# CS5014 Machine Learning
# Practical 1 Report

University of St Andrews – School of Computer Science

Student ID: 150014151

12th March, 2020

*Pages: 13 (excluding title page, table of contents, references and appendices)*

# Table of Contents

# 1 Introduction

This practical covers the exploration of various machine learning techniques used when dealing with real-life data. The data documents the critical temperatures required for different superconductors to conduct electrical current with no resistance [1]. Various techniques and design decisions considered and discussed for analysing the data, before extracting its features to make predictions on the critical temperatures. The predictions are made by training multiple regression models, which are all evaluated to determine their performance.

The report is separated in four sections. Starting with a general introduction, the report then explores all the methodology and design decisions made during the development of the code in Python, before evaluation the final performance of the regression model and critically reflecting on the findings.

## 1.1 Usage instructions

Before running the program, create a new virtual environment and install the Python libraries used in the code by running the following command:

```
1. pip install -r requirements.txt
```

To run the program, move to the "*src*" directory and run the following command:

```
1. python main.py -s <section> [-m <model>] [-g] [-d]
```

where:

- *"-s section"*: is a setting that executes different parts of the program. It must be one of the following: 'data_vis', 'train' or 'test'.
- *"-m model"*: is an optional setting that selects the regression model to use for training. It must be one of the following: 'linear', 'ridge', 'lasso', 'elastic_net', 'decision_tree', 'mlp', 'svm' or 'random_forest_generator'.
- *"-g"*: is an optional flag the grid search algorithm to determine the optimal hyperparameters for the selected regression model. The flag only takes effect when using linear regression with either Ridge or Lasso regularisation.
- *"-d"*: is an optional flag that enters debugging mode, printing additional statements on the command line.

## 1.2 Tools used

- Programming language: Python 3.7.

- Python libraries used: Scikit-Learn[1], Pandas[2], Matplotlib[3], NumPy[4] and Seaborn[5].
- Editor: PyCharm[6].
- Version controlling: Git and GitHub (private repository).
- Initial code prototyping: Jupyter Lab

# 2 System Architecture

## 2.1 Project structure

The project is organised into the following python modules:

- "main.py": functions to parse command line arguments and control the different sections of the code.
- "regression_models.py": all the functions to test Scikit-Learn's different regression models and functions to perform evaluation.
- "helper.py": functions to perform small operations on the data or print statements on the command line.
- "config.py": contains project-wide variables that are set by the command line arguments.

## 2.2 Execution flow

A command-line interface controls which part of the program to run, implemented through Python's native library *argparse*:

- Visualising the data set
- Training different models and running grid search
- Testing the final model

# 3 Methodology & Design Decisions

## 3.1 Initial data loading and data train/test split

The dataset used is provided by the CSV file related to the journal paper "A data-driven statistical model for predicting the critical temperature of a superconductor" [1]. This file

[1] Scikit-Learn: https://scikit-learn.org
[2] Pandas: https://pandas.pydata.org/
[3] Matplotlib: https://matplotlib.org/
[4] NumPy: https://numpy.org/
[5] Searborn: https://seaborn.pydata.org/index.html
[6] PyCharm: https://www.jetbrains.com/pycharm/

is directly loaded into a Pandas DataFrame[7] by using the *read_csv* function and immediately split into a training and a testing set.

```
1.  df = pd.read_csv("data/train.csv")
```

Before even viewing the data's features and values, the dataset is split in a training and a testing set to avoid any form of data snooping. Data snooping corresponds to the poor practice of making design decisions (either voluntary or involuntary) after viewing the data and detecting patterns that could lead to favouring certain models or hyperparameters above others. The opening sentence in H. White's abstract in his paper on data snooping reinforces how any form of data snooping could cause the final results to be questionable, thus reiterating the importance of splitting the data into two separate sets and forgetting about the testing data until the final evaluation [2]:

"*Data snooping occurs when a given set of data is used more than once for purposes of inference or model selection. When such data reuse occurs, there is always the possibility that any satisfactory results obtained may simply be due to chance rather than to any merit inherent in the method yielding the results*".

Various options to split the dataset into two sets are studied, including *random sampling* and *stratified sampling*.

- Random sampling consists of randomly selecting samples from the dataset with uniform probabilities, where each sample has the same probability as all other samples of being selected [3].
- Stratified sampling consists of maintaining representative samples from the data in both the training and the testing sets to avoid introducing sampling bias. To demonstrate this bias, an example of scatter plot containing random numbers between 0 and 100 is generated in Figure 1 below (see Appendix A for code used to generate chart). Under the assumption that the data can be partitioned. Into four relevant sections, the split needs to extract samples from each of the four sections to be considered as representative (green sample). The risk with random sampling is that the split does not sample from each representative section, causing the training set to contain samples from three sections and the testing set containing samples from the fourth section (red circle) for example. In this case, the model will not generalise well to the unseen data as it was not trained on representative data.

[7] Pandas DataFrame: https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html

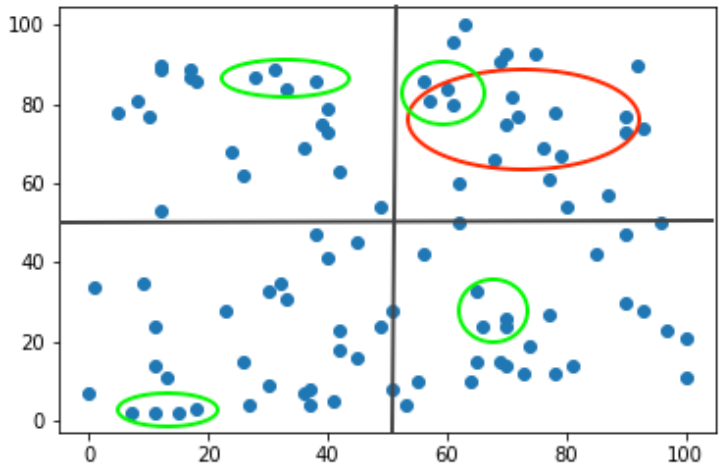



*Figure 1: High-level example demonstrating the concept behind stratified sampling. The scatter plot is generated in MatPlotLib, and the four representative separations and coloured samples are performed in a painting application.*

Considering both sampling methods to split the data into a training set and a testing set, stratified sampling is not always necessary, as random sampling is considered to be sufficient when the data is large enough. A. Géron mentions that stratified sampling is necessary for small datasets of about 1000 rows [4], and because the dataset used in this practical contains 21,263 rows, random sampling is therefore used instead. Additionally, stratified sampling requires strong knowledge of the dataset [4], which is not the case with this dataset as there are 81 features in total, and no unique feature is considered more important than the other (8 features are mentioned to be important) [1], thus rendering the task of performing a representative split of the data based on the most important feature impossible. Furthermore, random sampling completely eliminates the risk of introducing any form of bias [3].

The dataset is therefore randomly split using Scikit-Learn's *train_test_split* function with a 80%/20% distribution, which is the norm in machine learning [4], with the random number generator's seed set to a constant to ensure that it always generates the same shuffle indices when running the code multiple times to ultimately ensure reproducibility. The *shape* attribute of the Pandas DataFrame is used to double-check that the data was correctly split in a 80%/20% split (the training set size should be $21263 * 0.8 = 17010$ and the testing set size should be $21263 - 17010 = 4253$, which is the case).

```
1. from sklearn.model_selection import train_test_split
2. train_set, test_set = train_test_split(df, test_size=0.2, random_state=42)
3. print("Train set size = {}, Test set size = {}".format(train_set.shape, test_set.shape))
4. #Output: Train set size = (17010, 82), Test set size = (4253, 82)
```

At this stage of the process, with the data is separated into a training set and a testing set, only the training set is considered until the final evaluation, while the testing set is set aside and forgotten about.

It is important to note that the assumption that the data never changes during the entire practical is made, as using the random generator with a fixed seed would cause different samples to be extracted into the training and testing sets, thus neglecting the results [4].

## 3.2 Data visualisation and analysis

Before deciding how to fit a regression model to the training set, it is important to visualise and analyse the data in order to gain a deeper understanding of the data and its underlying patterns. To secure the training set from any undesired modifications, a copy of the training set, called the exploration set, is created and used for data visualisation.

```
1. exploration_set = train_set.copy()
```

### Data overview

Storing the data in a Pandas DataFrames provides a rich array of functions that can be used to quickly visualise the data and gain high-level insights. Using the *info* function to print a summary of the data reveals some information (see Figure 2), such as:

- There are no empty values in the entire dataset (each feature has 17,010 non-null values, which corresponds to the size of the training set)
- All values are numbers (either floats or integers)

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 17010 entries, 16546 to 15795
Data columns (total 82 columns):
 #   Column                        Non-Null Count  Dtype
---  ------                        --------------  -----
 0   number_of_elements            17010 non-null  int64
 1   mean_atomic_mass              17010 non-null  float64
 2   wtd_mean_atomic_mass          17010 non-null  float64
 3   gmean_atomic_mass             17010 non-null  float64
 4   wtd_gmean_atomic_mass         17010 non-null  float64
 5   entropy_atomic_mass           17010 non-null  float64
 6   wtd_entropy_atomic_mass       17010 non-null  float64
 7   range_atomic_mass             17010 non-null  float64
 8   wtd_range_atomic_mass         17010 non-null  float64
 9   std_atomic_mass               17010 non-null  float64
 10  wtd_std_atomic_mass           17010 non-null  float64
 11  mean_fie                      17010 non-null  float64
 12  wtd_mean_fie                  17010 non-null  float64
 13  gmean_fie                     17010 non-null  float64
 14  wtd_gmean_fie                 17010 non-null  float64
                                  ...
 75  entropy_Valence               17010 non-null  float64
 76  wtd_entropy_Valence           17010 non-null  float64
 77  range_Valence                 17010 non-null  int64
 78  wtd_range_Valence             17010 non-null  float64
 79  std_Valence                   17010 non-null  float64
 80  wtd_std_Valence               17010 non-null  float64
 81  critical_temp                 17010 non-null  float64
dtypes: float64(79), int64(3)
```

*Figure 2: Output of the info function called on the training set stored in a Pandas DataFrame.*

These two pieces of information mean that neither data encodings for non-numerical inputs are required, nor data cleaning operations are required as the data is already considered to be "*clean*".

### Data visualisation

Now that a high-level understanding of the data is acquired, more detail can be gathered from visualising the spatial distribution of each of the 81 features and the target column. This can be done by plotting each feature into individual small histograms (see Figure 3). This visualisation reveals two interesting characteristics about the data:

- Not all of the features' distributions are bell-shaped, some have a concentration of data points around specific regions (either for low values or high values at the edges of the distribution bins). This could indicate that feature transformation would be required to achieve more bell-shaped distributions.
- Observing the y-axes of the histograms, the features seem to have different scales, which may suggest that feature scaling is required.

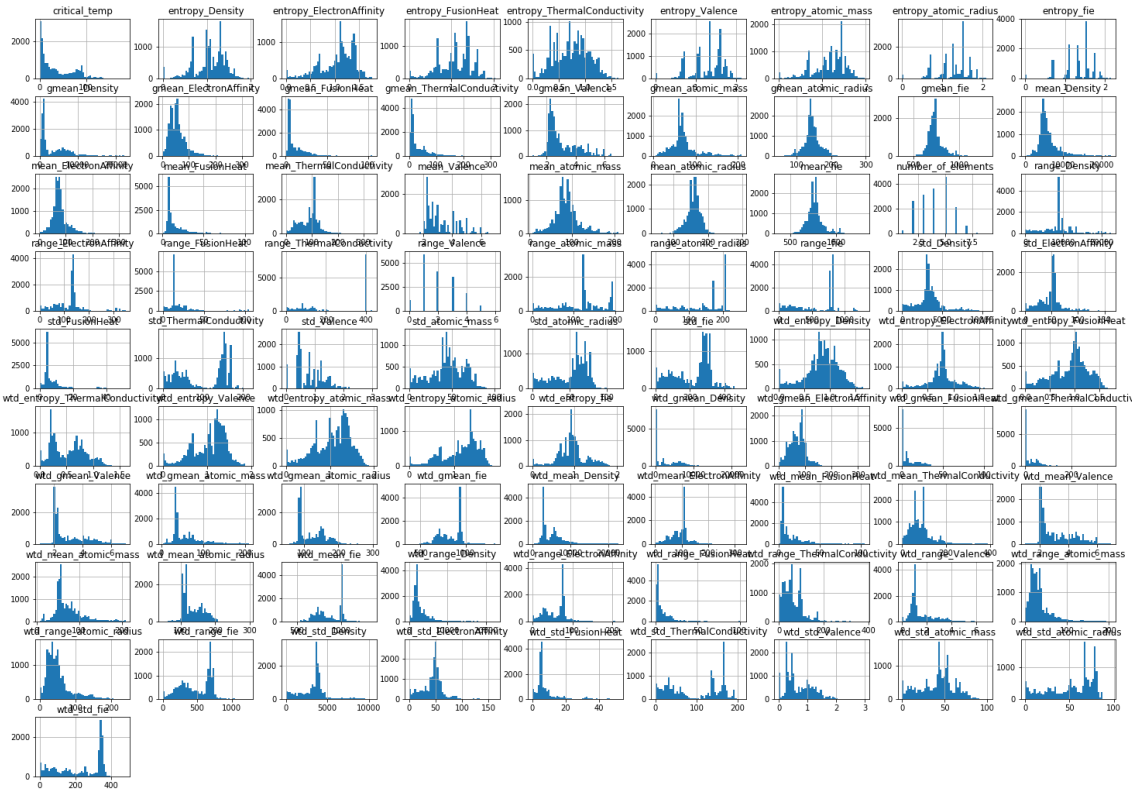



*Figure 3: Histogram distribution of each of the 81 features in the training data set.*

The solution to the two aforementioned problems is to standardise the data in order to transform the data into a distribution with unit variance and distribution shape more similar to a bell (the two properties of standardised distributions are a mean of all values equal to 0 and a standard deviation equal to 1) [4], [5]. This feature transformation is carried out in Section 3.4.

## 3.3 Feature selection

Although the K. Hanidieh identifies the eight most important features as being the atomic mass, the atomic radius, the electron affinity, the thermal conductivity and the valence [1]; it is important to establish our own data analysis to select features. This can be done by computing the standard correlation coefficient between the actual critical temperature and every feature in the training set. The correlation coefficient is a value ranging between -1 and 1. A correlation close to 1 indicates a strong positive linear correlation, while a correlation close to -1 indicates a strong negative linear correlation. However, a correlation close to 0 indicates no relationship between the data. According to existing literature, features are considered to have a high correlation when they have values in the region of $[\,|0.5,0.7|\,]$ [4], [6].

To determine which features show the highest correlation with the actual critical temperature, Pandas' *corr* function is called on the DataFrame, and the ten features with the highest positive correlation are displayed, as seen in Figure 4 (only the features with a positive correlation are displayed as scatter plots from the paper visually indicated that a model with a positive gradient would best fit the data). The high positive correlation coefficients for these features can be translated as a linear increase for the critical temperature when they increase.

```
10 features with the strongest correlation with 'critical_temp':
critical_temp                 1.000000
wtd_std_ThermalConductivity   0.720134
range_ThermalConductivity     0.686189
std_ThermalConductivity       0.652264
range_atomic_radius           0.652252
wtd_entropy_atomic_mass       0.626036
wtd_entropy_atomic_radius     0.603819
number_of_elements            0.601491
range_fie                     0.600777
entropy_Valence               0.598086
wtd_std_atomic_radius         0.595144
Name: critical_temp, dtype: float64
```

*Figure 4: Top ten features with the highest correlation in regard to the critical temperature.*

To better visualise the correlation between these, a correlation matrix of these top ten features is plotted, as seen in Figure 5. The matrix is constructed as a heatmap using the Seaborn library. This correlation matrix re-confirms that there is strong correlation between the critical temperature and the features, as their values range between 0.59 and 0.72.

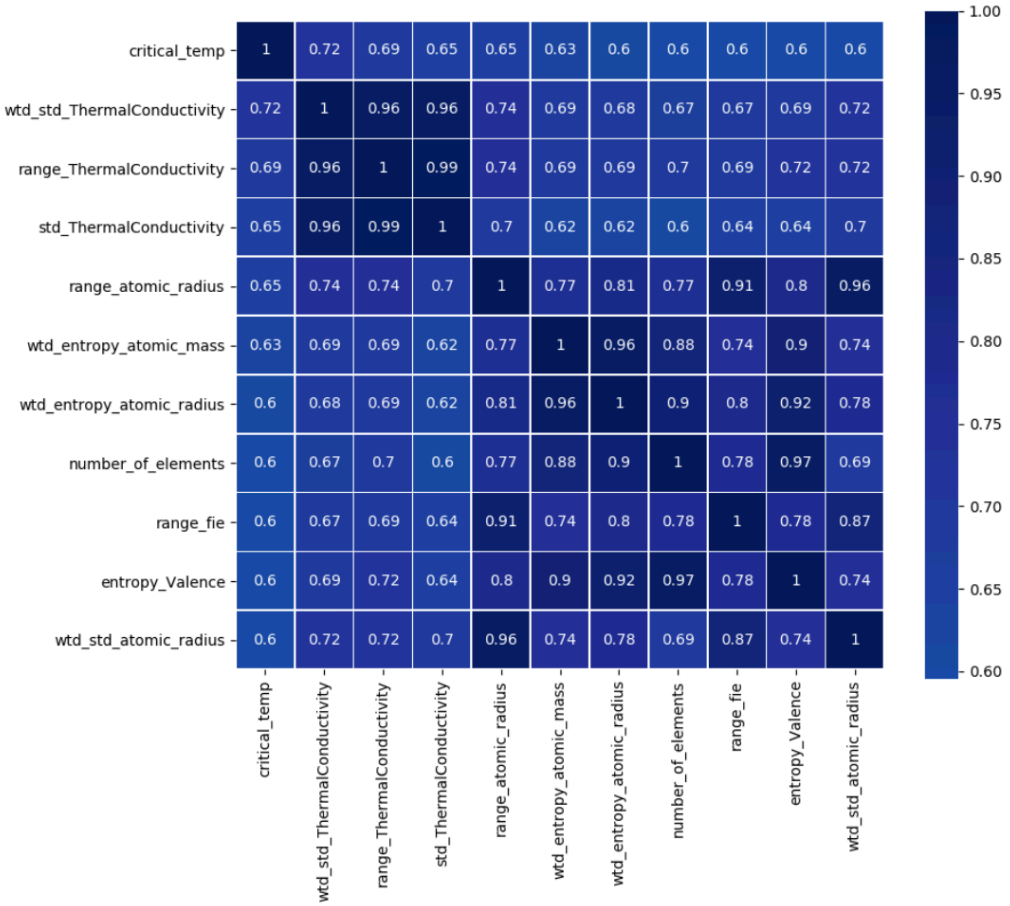



*Figure 5: Correlation matrix of the top ten features with the highest correlation in regard to the critical temperature.*

However, when observing the correlation between the features themselves, the correlation coefficient is much stronger than with the critical temperature for some pairs of features. This problem is known as multicollinearity, which occurs when multiple features can accurately predict the critical temperature, thus leading to misleading results [6]. A solution to this is either to use complex regression models such as decision trees, which can handle the multicollinearity in the data, or to drop some of the very closely correlated features. However, in order to keep all features in the training set, it is possible to stick to simple models like linear regression, where the weights of features that do not contribute towards reducing the error will be greatly diminished to the point where they can be considered as having been "dropped" from the data. For more in-depth visualisations of correlation between features, see Appendix B where scatter plots are used to visually describe the high linear correlation.

### 3.4 Input preparation

With the data visualised, the training set can be processed to maximise the regression models' fit on the data. The first steps consist in extracting the training set into inputs and outputs, which consists in dropping the critical temperature that the model needs to predict from the training set for the inputs and copying that same column for the output:

```
1.  X = train_set.drop("critical_temp", axis=1)
2.  y = train_set["critical_temp"].copy()
```

As mentioned earlier in the "Data overview" part of Section 3.2, no data encodings or cleaning operations are required as the data is already clean. However, because the distributions of the different features were not bell-shaped and had very different scales, feature scaling is required in the form of standardisation. To do so, Scikit-Learn's *StandardScaler* is used on the input features of the training set only.

```
1.  from sklearn.preprocessing import StandardScaler
2.  def input_preparation(train_set):
3.      X = train_set.drop("critical_temp", axis=1)
4.      y = train_set["critical_temp"].copy()
5.      X[X.columns] = StandardScaler().fit_transform(X[X.columns])
6.      return X, y
```

## 3.5 Training

This section consists of the various steps followed to fit the training data to various regression models by measuring the error between the model's predicted critical temperatures and the actual critical temperatures. In his paper, K. Hamidieh reaches an RMSE (Root Mean Squared Error) of about 17.6K and an $R^2$ score of about 0.74% using multiple regression models [1].

According to A. Géron, one of many approaches to selecting models is to test multiple regression models, often without tweaking the hyperparameters to keep default values recommended by Scikit-Learn. The goal of this approach is to get a high-level intuition of which model to use [4]. After applying this step to linear, decision tree, multi-layered perceptron, SVM, and random forest generator regression models; only the linear regression models were kept (general linear regression and regularised – with Lasso and Ridge regression). The Python functions for the other complex regression models is still kept in the code but are not considered in the report.

### Performance metrics

Many performance metrics exist to evaluate the quality of a fit on a regression model, such as MAE (Mean Absolute Error), MSE (Mean Squared Error) or RMSE [4]. However, the same metrics used in K. Hamidieh's paper will be used, which correspond to the RMSE and the $R^2$ score (coefficient of determination), in order to allow direct comparison between the models implemented in this practical and the paper's model:

- RMSE is a metric that keeps the same unit as the measure being predicted (the error is reported in Kelvins). It directly compares the predicted values and the actual values for the critical temperature, which is why it is always selected by predictive models [4].

- $R^2$ is a metric expressed in the form of a percentage that indicates how close the data is to the line of best fit (100% means that the model explains all the variation in the data, whereas 0% means that the model does not explain any of the variation in the data) [7].

### K-fold cross validation

K-fold cross validation consists of a more representative way of evaluating the trained regression model. It divides the training set in K (e.g. K=5) subsets and evaluates the model K times (calculates the RMSE and the $R^2$ score each time) on one of subsets, each time training on the K-1 other subsets. At the end, there are K different results, which can be averaged to get the mean error/scores across the K subsets [4].

The mean score is often worse as it is an average of all subsets, which may include some subsets with data that is not representative of the training data. However, the mean scores can be coupled with the standard deviation to get a more accurate estimate of the model's actual performance [4]. For example, fitting a linear regression model on the training data and evaluating its RMSE and $R^2$ score give the following results: $RMSE = 17.61K$ and $R2 = 0.74\%$. However, evaluating the same model using 10-fold cross validation yields the following: $RMSE = 17.68K \pm 0.45$ and $R^2 = 0.73\% \pm 0.01$ thanks to the standard deviation. This information would not have been available if a single validation set (single fold) was being used to evaluate the training model.

Despite the advantages of running k-fold cross validation, it is not always possible as it has the steep cost of multiplying execution times by the number of folds. In the case of Linear Regression, this added cost is acceptable, but using a larger dataset or more complex regression models such as neural networks or random forest generators would considerably increase the run time. [4]

### Regression models & regularisation

Scikit-Learn provides three linear regression models that are used and compared in this practical: Linear Regression, Linear Regression with Ridge Regression and Linear Regression with Lasso Regression.

The linear regression model is the basic regression model used, whereas the two others introduce regularisation to help counter overfitting, and according A. Géron, work well with standardised data [4], which is a feature transformation that was applied in Section 3.4:

- Ridge regularisation: keeps the weights applied to each feature as small as possible through a variable $\alpha$. If $\alpha = 0$, then basic linear regression is performed. A high $\alpha$ will cause the weights to be small, and therefore the line of best fit to be flat.
- Lasso regularisation: eliminates the weights of unimportant features by setting them to 0.

### Model fine-tuning

The hyperparameters of Scikit-Learn's three linear regression models (*LinearRegression*, *Ridge* and *Lasso*) are manually tried out to briefly evaluate which have an impact on the evaluation. Next, a grid search algorithm is used to fine-tune the models' hyperparameters based on the hyperparameters that have an impact on the model. The grid search evaluates the model using cross-validation with all possible combinations of hyperparameters that are desired, making it the most efficient way of fine-tuning the regression models as manual testing different combinations of parameters is very time-consuming.

After manually tweaking hyperparameters, the following combinations are tested using grid search (no grid search is run on the basic linear regression as there is no hyperparameter to tune for improvements). For example, 42 different combinations[8] are tested using cross validation for the Ridge model:

```
1.  linear_regression_parameters = {}
2.  ridge_parameters = {
3.          "alpha": [0.1, 1, 10],
4.          "normalize": [True, False],
5.          "solver": ["auto", "svd", "cholesky", "lsqr", "sparse_cg", "sag", "saga"]
6.      }
7.  lasso_parameters = {
8.          "alpha": [0.0001, 0.001, 0.01, 0.1, 0, 1, 10],
9.          "tol": [0.01, 0.1, 1],
10.         "positive": [True, False],
11.         "selection": ["cyclic", "random"]
12.     }
```

The results are reported back into a CSV file with information about each combination of hyperparameters. It is tempting to immediately choose the model that is ranked the highest (the one with the lowest mean test score), but it is important to double check that the model performs as well on the training set as it does on the testing set. If the model performs much better on the training than on the testing, then the model is overfitting the data with that combination of hyperparameter and should not be selected. If the model performs poorly on both the training and testing sets, then it is underfitting the data. The full results of each grid search are stored in csv files in the "*grid_search_results*" directory.

### Training execution flow

Training is tackled as follows: an instance of one of Scikit-Learn's linear regression models is created and grid search is run on it to determine optimal hyperparameters. These hyperparameters are determined by analysing the grid search results in Excel and are then used to perform 10-fold cross validation to get an idea of the model's performance on the

[8] 42 = 3*2*7

training data compared to the paper's results using the same metrics. The model's predicted values are then plotted against the actual values for the critical temperature and a line of best fit is plotted to show how closely the fit matches the data. The model is finally saved into a ".pkl" file for future reference in the "trained_models" directory.

# 4 Evaluation & Critical Discussion

## 4.1 Selecting the best model for evaluation

The training execution flow mentioned in Section 3.5 is followed for each of the three linear regression models used. The table below documents the scores found for each model using 10-fold cross validation using the optimal hyperparameters determined during the grid search (for ridge and lasso only).

| Model | RMSE | $R^2$ | Optimal Hyperparameters |
|---|---|---|---|
| **Linear (Appendix C)** | $17.68K \pm 0.45$ | $0.73\% \pm 0.01$ | • fit_intercept=True,<br>• normalize=True |
| **Ridge (Appendix D)** | $17.68K \pm 0.45$ | $0.73\% \pm 0.01$ | • alpha=0.1,<br>• solver='svd',<br>• normalize=False |
| **Lasso (Appendix E)** | $17.75 \pm 0.3$ | $0.73\% \pm 0.01$ | • alpha=0,<br>• tol=0.01,<br>• selection="random",<br>• positive=False,<br>• max_iter=1000,<br>• normalize=False |

This table indicates that there is minimal difference between the performance of each optimal linear regression model. Indeed, looking at the column of the optimal hyperparameters used for each optimal model, the alpha constant, which sets the strength of the regularisation for the Ridge and Lasso models, and therefore the penalty applied to irrelevant features, are near 0. This means that a general linear regression model without regularisation is actually being applied in all three cases. To break the tie, the linear regression model with Ridge regularisation is used as it includes some minor form of regularisation which may help generalise to unseen data.

## 4.2 Final model evaluation against test data

Finally, the test data set can be used for the final evaluation of the chosen model. Before predicting the critical temperatures based on the testing data, the testing data has to undergo the same steps covered in Section 3.4. The testing data is split between inputs and outputs, and the outputs are scaled in the same way.

Only then can the *predict* function be called on the inputs of the testing set for the first time. The results are the following:

- $RMSE = 17.38K$
- $R^2 = 0.74\%$

Compared to the paper's results [1], the RMSE improved by 0.22K, while the $R^2$ score remained the same. Observing the line of best fit on the testing data in Figure 6, it can be visually confirmed that the linear model.

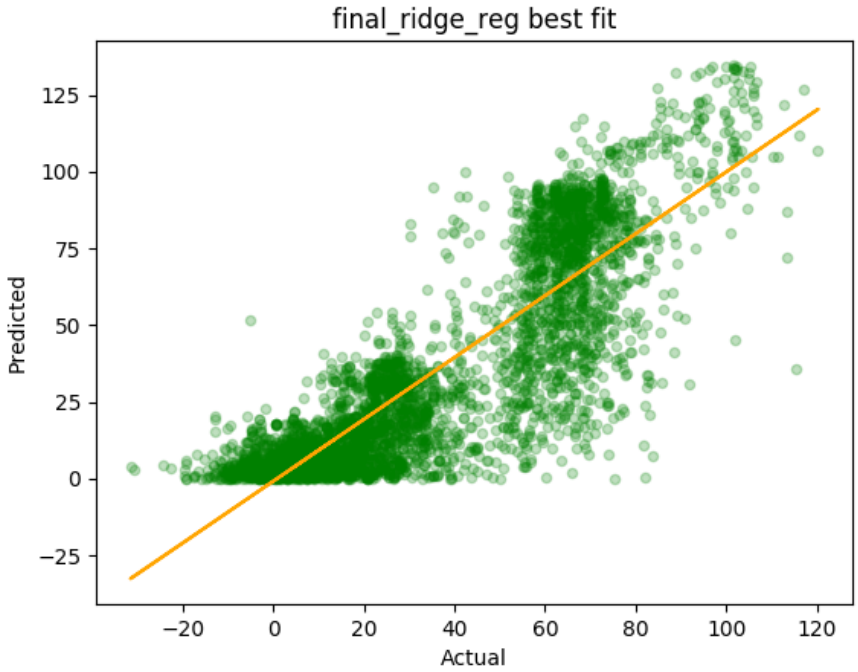



*Figure 6: Line of best fit on the testing dataset with the final selected model.*

## 4.3 Critical Discussion

As a final comparison, the difference between the training RMSE (17.68K) and the testing RMSE (17.38K) is very low, suggesting that there the model does not overfit nor underfit the data. Considering this critical observation [4] and the fact that all the steps followed throughout the practical are considered as "standard" in the field of machine learning, the assumption that the model produced in this practical is a potential improvement on the regression model used in the paper can be made. However, there is always a possibility that on one step along the way an error was made.

Further improvements could have been made by exploring more complex regression models such as Decision Trees or Random Forests to perhaps approach the RMSE found in the paper (9.5K) when using boosted models (XGBoost) [4]. Based on the brief testing conducted with Scikit-Learn's implementations of these more complex models, they often seemed to overfit the data, requiring deeper knowledge of these models to fine-tune their hyperparameters to correctly use them. For this reason, the practical's scope focused on linear regression models only.

# References


[1] K. Hamidieh, "A data-driven statistical model for predicting the critical temperature of a superconductor," *Computational Materials Science*, vol. 154, pp. 346–354, Nov. 2018.

[2] H. White, "A reality check for data snooping," *Econometrica*, vol. 68, no. 5, pp. 1097–1126, Sep. 2000.

[3] J. Dudovski, "Simple Random Sampling," *Research Methodology*, 2019. [Online]. Available: https://research-methodology.net/sampling-in-primary-data-collection/random-sampling/. [Accessed: 12-Mar-2020].

[4] A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow*, 2nd ed. O'Reilly Media, 2019.

[5] S. Glen, "Standardized Values: Example," *Statistics How To*, 2014. [Online]. Available: https://www.statisticshowto.datasciencecentral.com/standardized-values-examples/. [Accessed: 12-Mar-2020].

[6] W. Badr, "Why Feature Correlation Matters .... A Lot!," *Towards Data Science*, 2019.

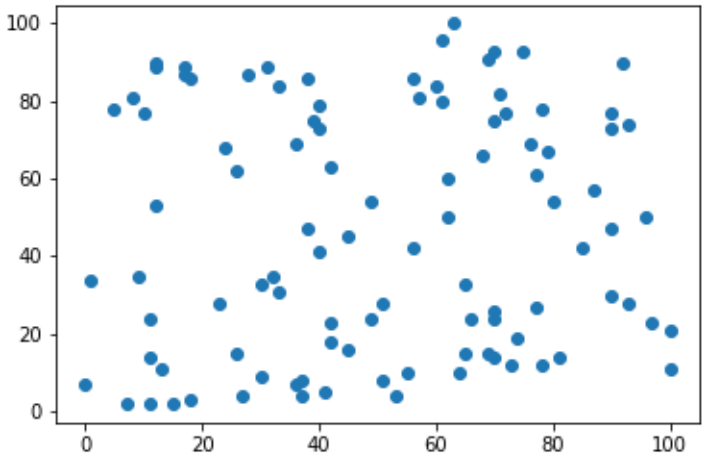
[7] "Regression Analysis: How Do I Interpret R-squared and Assess the Goodness-of-Fit?," *The Minitab Blog*, 2013. [Online]. Available: https://blog.minitab.com/blog/adventures-in-statistics-2/regression-analysis-how-do-i-interpret-r-squared-and-assess-the-goodness-of-fit. [Accessed: 12-Mar-2020].

# Appendix A: Random scatter plot for stratified example

Code used to generate the random scatter plot used to explain stratified sampling.

```
1.  import random
2.  def random_plot():
3.      random.seed(0)
4.      x = [random.randint(0,100) for i in range(0, 100)]
5.      y = [random.randint(0,100) for i in range(0, 100)]
6.      plt.scatter(x,y)
7.      plt.savefig("plot_stratified_example.png")
8.      plt.show()
9.  random_plot()
```

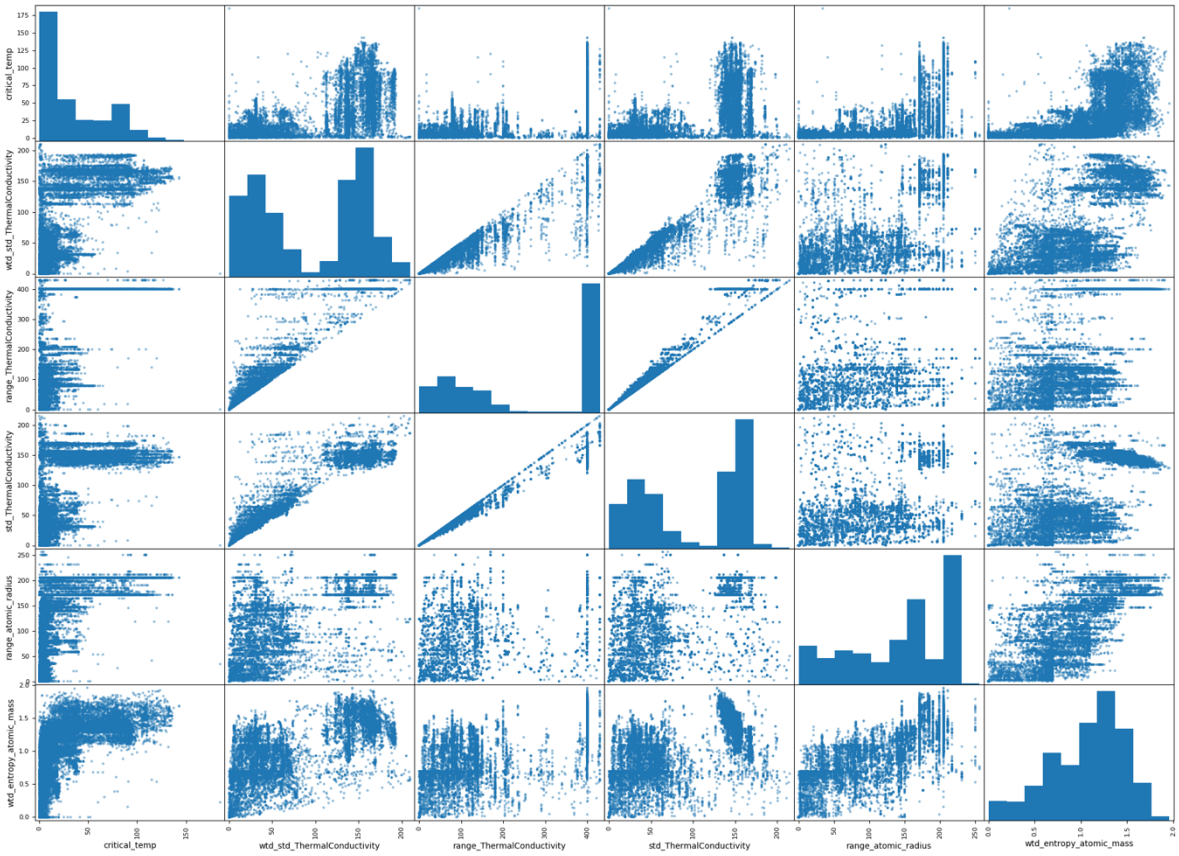
The plot generated by the code above:



# Appendix B: In-depth feature correlation visualisation

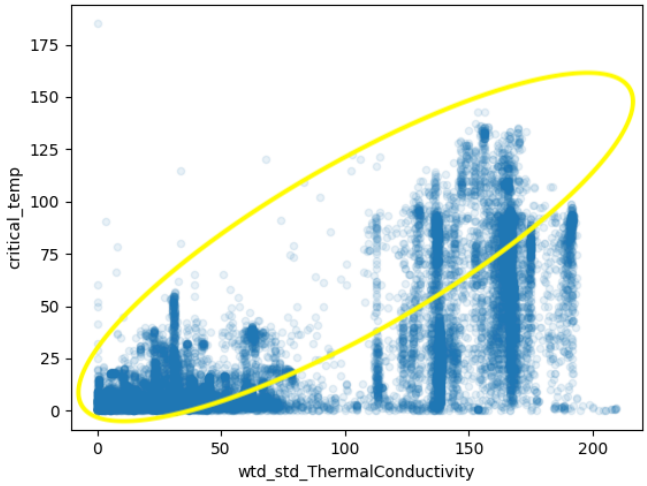
## Appendix B.1

Scatter plots of the top five features correlated to the critical temperature, which clearly betrays the positive linear correlation between the features and the data:



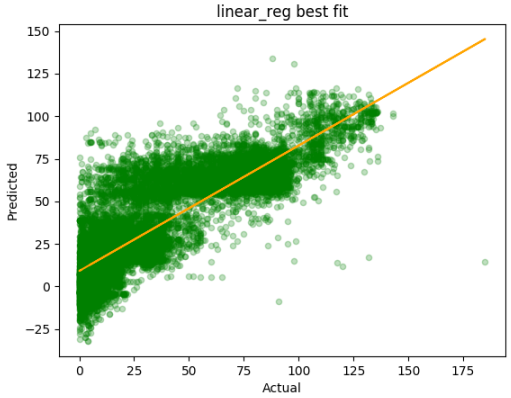

## Appendix B.2

Zooming in on the scatter plot (from Appendix B.1) of the feature with the highest correlation with the critical temperature (wtd_std_ThermalConductity), signals the linear relationship between them, as seen in the figure below:



# Appendix C: Linear Regression Model evaluation

Line of best fit achieved:




Command line output:

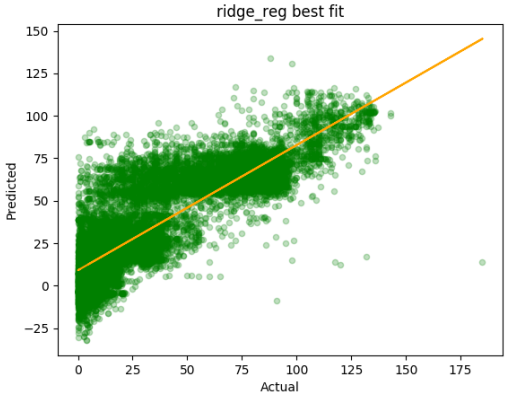
File - train linear

```
 1 /Users/[redacted]/Environments/ML-Predicting-
   Superconductivity-Critical-Temperature-CS5014/bin/python /
   Users/[redacted]Projects/ML-Predicting-Superconductivity-
   Critical-Temperature-CS5014/src/main.py -s train -m linear
 2
 3 Training Linear Regression model:
 4
 5 Predictions: [ 2.81982681 72.83853211 62.60904699 70.
   41616711 24.86911432]
 6 Real value: [5.0, 97.0, 62.1, 60.0, 4.0]
 7 RMSE=17.61K
 8 R^2=0.74%
 9 10-fold cross validation results:
10
11 RMSE scores: [18.45082018 18.00777982 16.95175983 18.
   14080001 17.15519339 17.90118777
12  17.86937038 17.5476235  17.29397962 17.50679636]
13
14 Mean RMSE: 17.68K (+/-0.45)
15
16 R2 scores: [0.7144509  0.72942555 0.75706526 0.70860898 0.
   7321999  0.72314711
17  0.74030987 0.7441437  0.74062472 0.75118776]
18
19 Mean R2: 0.73% (+/-0.01)
20
21 --- Runtime: 1.27 seconds ---
22
23 Process finished with exit code 0
24
```

Page 1 of 1

# Appendix D: Ridge Regression Model evaluation

Line of best fit achieved with fine-tuned model:




Command line output:

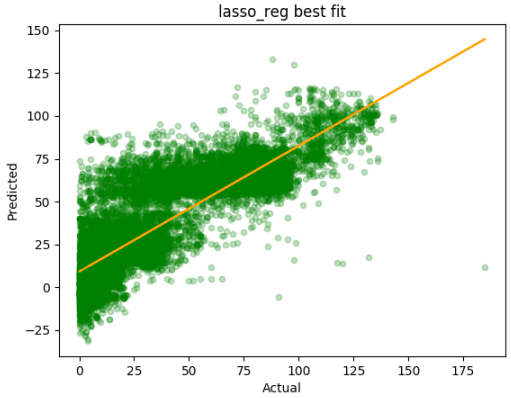
File - train ridge

```
 1 /Users/[redacted]/Environments/ML-Predicting-
   Superconductivity-Critical-Temperature-CS5014/bin/python /
   Users/[redacted]/Projects/ML-Predicting-Superconductivity-
   Critical-Temperature-CS5014/src/main.py -s train -m ridge
 2
 3 Training Linear Regression model with Ridge regularisation:
 4
 5 Predictions: [ 2.97208397 72.75729935 62.55443334 70.
   46167647 24.7970406 ]
 6 Real value: [5.0, 97.0, 62.1, 60.0, 4.0]
 7 RMSE=17.61K
 8 R^2=0.74%
 9 10-fold cross validation results:
10
11 RMSE scores: [18.44913509 18.00972228 16.94849773 18.
   13799875 17.1534157  17.90501772
12  17.87117531 17.55369892 17.29249674 17.50148474]
13
14 Mean RMSE: 17.68K (+/-0.45)
15
16 R2 scores: [0.71450306 0.72936717 0.75715875 0.70869897 0.
   7322554  0.72302863
17  0.74025741 0.7439665  0.7406692  0.75133871]
18
19 Mean R2: 0.73% (+/-0.01)
20
21 --- Runtime: 1.46 seconds ---
22
23 Process finished with exit code 0
24
```

Page 1 of 1

# Appendix E: Lasso Regression Model evaluation

Line of best fit achieved with fine-tuned model:




Command line output: too many warnings to include in the report.