CS5014 Machine Learning

Practical 1 Report

University of St Andrews – School of Computer Science

Student ID: 150014151

12th March, 2020

A close up of a logo

Description automatically generated

Pages: (starting from introduction to

Table of Contents

[1 Introduction 3](#_Toc34926489)

[1.1 Usage instructions 3](#_Toc34926490)

[1.2 Tools used 3](#_Toc34926491)

[2 System Architecture 4](#_Toc34926492)

[2.1 Project structure 4](#_Toc34926493)

[2.2 Execution flow 4](#_Toc34926494)

[3 Methodology & Design Decisions 5](#_Toc34926495)

[3.1 Initial data loading and data train/test split 5](#_Toc34926496)

[3.2 Data visualisation and analysis 7](#_Toc34926497)

[3.3 Feature selection 9](#_Toc34926498)

[3.4 Input preparation 10](#_Toc34926499)

[3.5 Training 11](#_Toc34926500)

[3.6 Model fine-tuning 11](#_Toc34926501)

[4 Evaluation 12](#_Toc34926502)

[5 Critical Discussion & Conclusion 13](#_Toc34926503)

[References 14](#_Toc34926504)

[Appendix A: Random scatter plot for stratified example 15](#_Toc34926505)

[Appendix B: In-depth feature correlation visualisation 16](#_Toc34926506)

[Appendix B.1 16](#_Toc34926507)

[Appendix B.2 16](#_Toc34926508)

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. 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. Tools used
* Programming language: Python 3.7.
* Python libraries used: Scikit-Learn[[1]](#footnote-1), Pandas[[2]](#footnote-2), Matplotlib[[3]](#footnote-3), NumPy[[4]](#footnote-4) and Seaborn[[5]](#footnote-5).
* Editor: PyCharm[[6]](#footnote-6).
* Version controlling: Git and GitHub (private repository).
* Initial code prototyping: Jupyter Lab

1. System Architecture
   1. Project structure

The project is organised into the following python modules:

* Main
* Regression\_models
* Helpers
* config
  1. Execution flow

Command line interface to run different parts of the program

1. Methodology & Design Decisions
   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 is directly loaded into a Pandas DataFrame[[7]](#footnote-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.

A screenshot of a cell phone

Description automatically generated

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 and the testing set size should be , 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].

* 1. 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)

A close up of text on a white background

Description automatically generated

…

A screenshot of a cell phone

Description automatically generated

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.

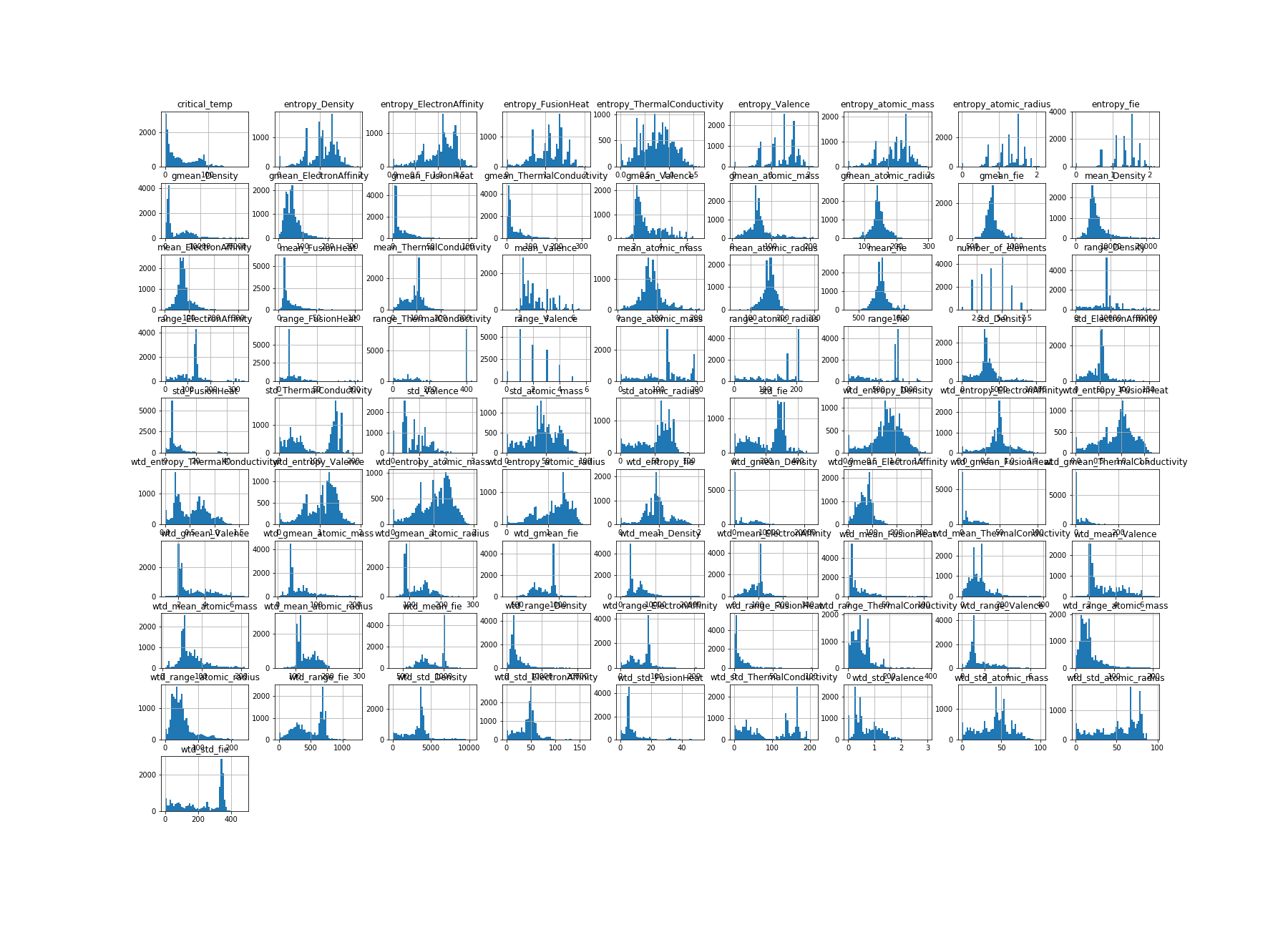


Figure 3: Histogram distribution of each of the 81 features in the training data set.

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.

* 1. 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 [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.

A screenshot of a cell phone

Description automatically generated

Figure 4

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.

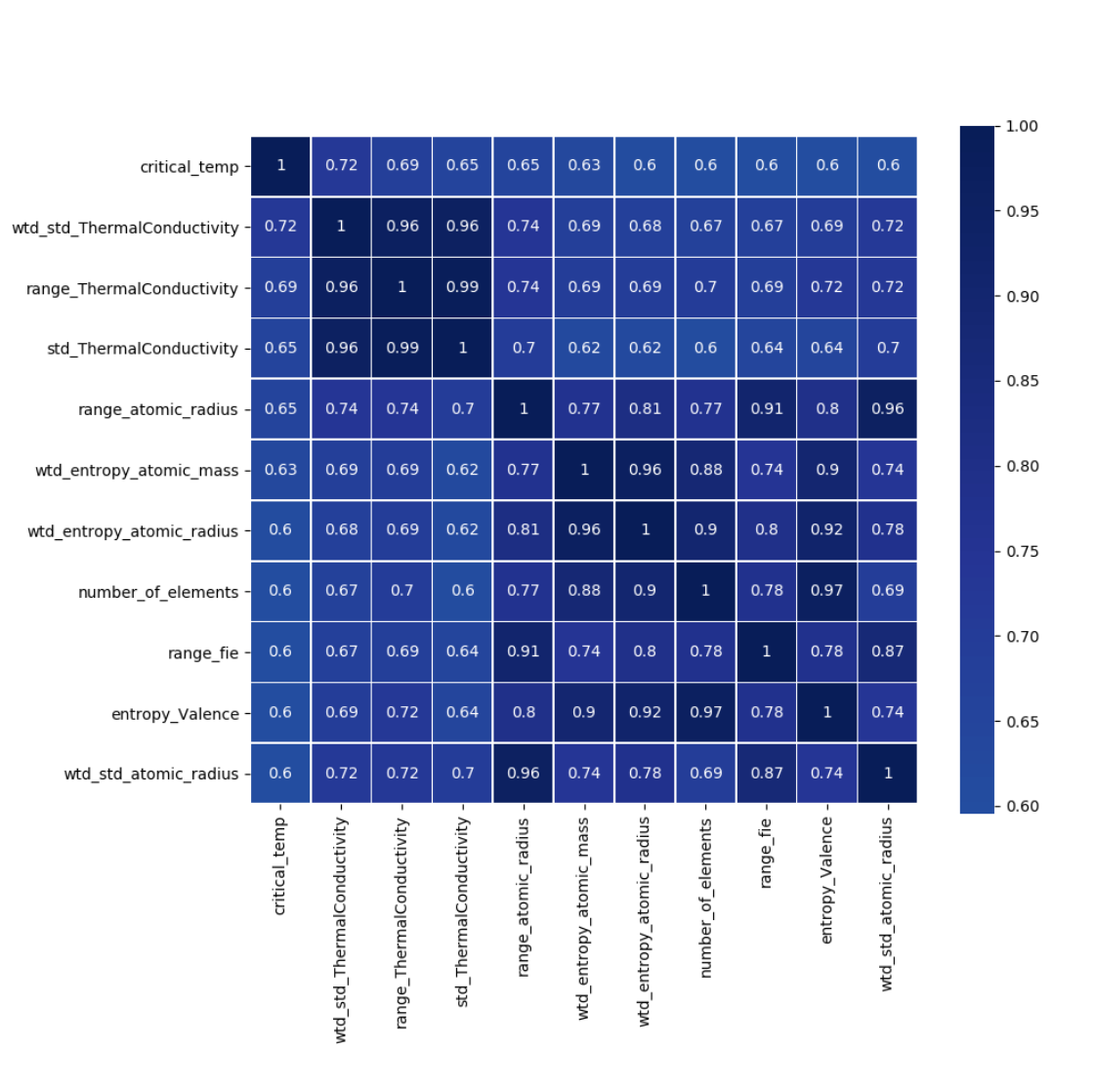


Figure 5: Correlation matrix.

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.

For more in-depth visualisations of correlation between features, see Appendix B where scatter plots are used to visually describe the high linear correlation.

* 1. 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

* 1. Training

This section consists of fitting 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 R2 score of about 0.74% [1].

According to A. Géron, it is ideal to test multiple regression models, often without tweaking the hyperparameters to keep default values. 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.

General Linear

* 1. Model fine-tuning

Look at whether model is overfitting or not in grid search

1. Evaluation

Evaluation of the final model

1. Critical Discussion & Conclusion

todo

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.

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:

A screenshot of a cell phone

Description automatically generated

Appendix B: In-depth feature correlation visualisation

Appendix B.1

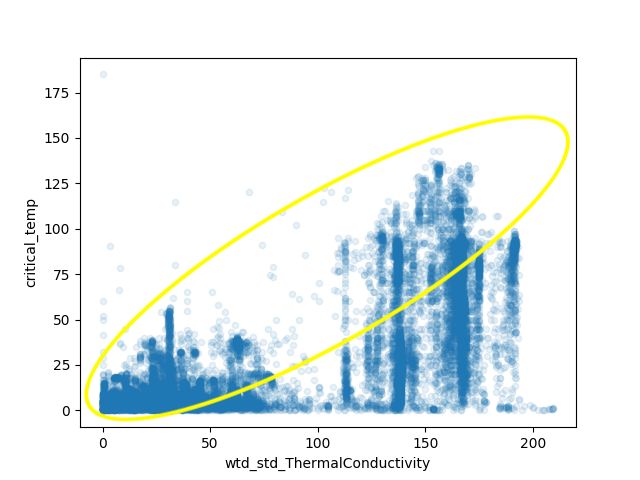
Scatter plots of the top five features correlated to the critical temperature, which clearly betrays the positive linear correlation between the features and the data:

A picture containing text

Description automatically generated

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:



1. Scikit-Learn: <https://scikit-learn.org> [↑](#footnote-ref-1)
2. Pandas: <https://pandas.pydata.org/> [↑](#footnote-ref-2)
3. Matplotlib: <https://matplotlib.org/> [↑](#footnote-ref-3)
4. NumPy: <https://numpy.org/> [↑](#footnote-ref-4)
5. Searborn: <https://seaborn.pydata.org/index.html> [↑](#footnote-ref-5)
6. PyCharm: <https://www.jetbrains.com/pycharm/> [↑](#footnote-ref-6)
7. Pandas DataFrame: <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html> [↑](#footnote-ref-7)