Data Analysis and Machine Learning, Project 3

Francesco Pogliano

December 2019

Department of Physics, University of Oslo, Norway

Abstract

In this report I have explored different machine learning techniques to model the nuclear mass data provided by the Atomic Mass Data Center. The goal was to order to reproduce (or at least compare to) the results of Utama et al. [6], these obtained using Bayesian Neural Networks. Linear regression, Feed Forward Neural Networks, decision trees and XGBoost methods were used in this work.
The results were satisfactory giving a standard deviation of 2.891 in the nuclear binding mass error with XGBoost, the best performing model. This is though six times as much as the theoretical-BNN results, and three times the purely theoretical model obtained by Utama et al. One of the reasons for this under-performance is probably due to the fact that only the neutron and proton numbers were used for the fitting, this not being the case for a theoretical model where actual physics is taken into account.

Contents

1 Introduction

One of the quantities of major interest in the field of nuclear physics and astrophysics is the nuclear mass. This, together with other properties such as the radii and the lifetime, gives us a way to probe into how nuclei behave in astrophysical conditions and help us understand how chemical elements are formed in the process called nucleosynthesis.
While we can measure the mass of stable nuclei relatively easily, this is not the case for the exotic ones far from the valley of stability, and the nuclear properties of such nuclei play a vital role in understand the processes of nucleosynthesis happening in sites such as supernovas and neutron star mergers, where we expect nuclei close to the neutron drip line to be formed. The difficulties in studying the properties of such nuclei make us reliant on the predictions from theoretical models, which to a certain degree can help us in this quest.
Recently, new effort has been put into looking for new ways of predicting such masses using machine learning algorithms. One of such ways is the use of Bayesian Neural Networks (BNN), which have been employed for example to predict both masses [6] and the separation energy for two neutrons [8].
In this report I will use different regression methods such as decision trees and XGBoost on the same datasets used in the work from R. Utama and J. Piekarewicz in order to try to reproduce their results, and compare these techniques to other ML-methods used in my other projects [10] [11], such as regression and Feed Forward Nuclear Networks (FFNN).
The report is structured in this way: First a general introduction of the theoretical concepts is given, both when it comes to physics, and the algorithms used. Then it follows a section dedicated to the new additions to the code from the previous reports. Finally, the results are given together with an analysis and a discussion in the conclusion. All of the relevant graphs are collected in the Appendix, while tables with the most important results can be found in the Results and Analysis section.

All scripts and datasets can be found in my GitHub repository at https://github.com/Cyangray/ML-project-3.

2 Formalism

This project will continue in our series of studies of different machine learning algorithms and data analysis, focusing on decision trees and boosting in order to reproduce the nuclear mass estimates in the paper of Utama et al. [6]. In this section I will introduce the theoretical background needed in order to understand the analysis part. Firstly I will explain the physics behind the datasets and the datasets themselves, for then taking care of the theoretical part behind decision trees and the XGBoost algorithm. Most of the content in this section is based on the lecture notes on decision trees by Morten Hjorth-Jensen [9] and the notes on XGBoost by Tianqi Chen and Carlos Guestrin [3].

2.1 Nuclear physics: An overview

An atomic nucleus is usually described as a system formed by two kinds of particles, protons and neutrons, interacting with each other via the strong nuclear force. Although the masses of a free proton and a free neutron are known, the resulting mass of a nucleus will not translate to the simple sum of the mass of its components. This mass will usually fall short of that sum by a certain value, called *mass deficit*, and this difference, translated into energy with Einstein's famous equation $E = mc^2$, is the binding energy holding the nucleons together.

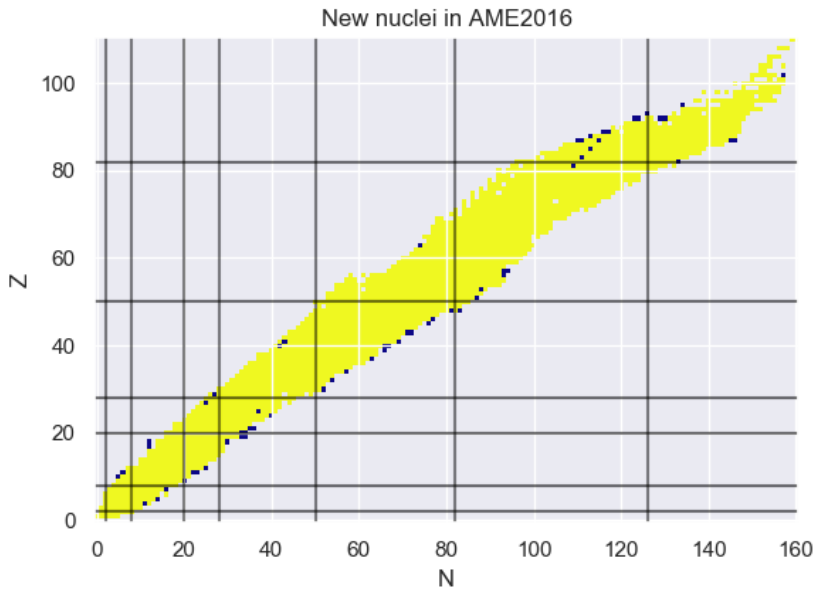

Figure 1: Plot of AME2012 and AME2016, here where the nuclei present in the older dataset are in yellow, while the newly added nuclei in dark blue. Here are the number of neutrons on the x-axis, while the number of protons on the y-axis. The black lines represent the magic numbers.

The mass of a nucleus then tells us then how tightly bound the nucleons are on average, and the masses of two different nuclei will tell us whether a transition between the two (say, a nuclear reaction) will give off or take in energy. This is of big importance when it comes to all nuclear physics and, when it comes to nuclei far from the valley of stability, to astrophysical applications such as the nucleosynthesis processes. As these masses are not readily available for experiments, we must rely on values from either theoretical models or, more recently, by extrapolating models fitted to the known masses.
Tables with mass measurements are published in different occasions by the Atomic Mass Data Center, with their most recent publication being the Atomic Masses Evaluation of 2016 [4] [7], and the one before the Atomic Masses Evaluation of 2012 [1] [2]. These datasets, from now on AME2012 and AME2016, can then be used to model known data in order to get predictions about nuclei with masses yet to be known. This has been the focus of the paper by Utama et al. [5] when it comes to AME2012 by using a Bayesian Neural Network, where they analyzed different theoretical models with machine learning (among others the Hartree-Fock-Bogolyubov HBF models, and the Duflo-Zucker DZ one, yielding the best results) and in 2018 [6] when evaluating the model against the new measurements appeared in AME2016 [4] [7], using them as a test set. In this project I will try to emulate their results by using Decision Trees and comparing it to other techniques already discussed in the other reports, such as Feed Forward Neural Networks and linear regression.

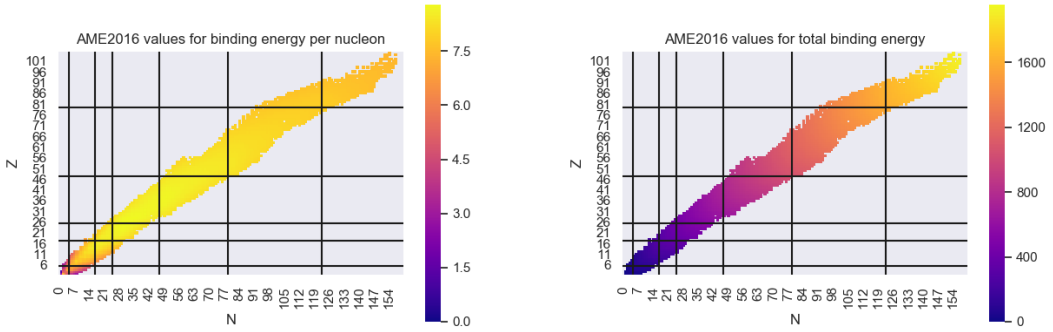

Figure 2: Two plots of the AME2016 dataset, where the color of each nucleus represent the binding energy per nucleon (B/A) in he left plot, while it represent the total binding energy in the right plot. We see that although the binding energy per nucleon is higher around the ^{56}Fe nucleus in the left plot, the total mass deficit is way higher for bigger nuclei in general. The black lines represent the magic numbers.

2.2 Decision Trees

Decision trees is a machine learning technique used for supervised learning which, put it simply, tries to find which features describe the dataset in the best way with respect to the target, for then making predictions according to those. For each found feature the dataset is split according to it and resulting in two dataset. The procedure is then repeated by finding the feature that contains most information in each of these datasets, splitting them, and going on until a certain criteria is met (such as, for example, when the resulting dataset is small enough, or after a certain number of splits has been done, this being the depth of a tree). A decision tree can be divided into a root node (where the first decision occurs), which divides the dataset into two branches. These, in turn, meet their internal nodes, where they split each into two more datasets, and so on until the leaf node is reached, where the dataset is no longer split.

Decision trees work for both classification and regression tasks. When it comes to regression, the algorithm can be seen as trying to find a way to divide the p-dimensional dataset space into $\boldsymbol{J}$ hypercubes, where p is the number of predictors in the dataset. The prediction will then be the average of the values inside these cubes, and the goal is to choose the limits of these cubes so that a cost function (such as the MSE) is minimized. The decision tree algorithm is a greedy algorithm, which means that it chooses each time the best way to split the dataset, without considering whether there is a better way which may look worse at first, but would eventually end up yielding a better predicting tree.
One of the problems with decision trees is that they are prone to overfit, or in other words divide the dataset into too many subsets, or leaves. This can be avoided with a pruning procedure, where we add a cost proportional to the amount of leaves. This helps the model get a smaller variance and reduce complexity.
I will mostly use the scikit-learn package in this project, and the decision tree regressor uses the CART algorithm. This corresponds to looking for the k-th feature and corresponding t_k threshold for which the following cost function is minimized:

$$C(k, t_k) = \frac{m_{\text{left}}}{m} MSE_{\text{left}} + \frac{m_{\text{right}}}{m} MSE_{\text{right}}, \tag{1}$$

where MSE_{left} and MSE_{right} are the mean square errors for the regions to the left and the right of the threshold respectively, m the total number of values in the dataset and m_{left} and m_{right} the values in the region on each side of the threshold.

2.2.1 Gradient Boosting and XGBoost

As decision trees are prone to overfitting, we are interested in ways to avoid that, for example, a slightly different dataset could result in a completely different tree. This is the case of gradient boosting, which employs techniques in order to make weak predictors have a say in the modeling. Adaptive boosting, which may be seen as gradient boosting for classification tasks, may be intuitively understood as iteratively adapting the dataset by making the misclassified predictions count more, for then combining the different predictions. Gradient boosting functions in a somewhat similar way, where the guess of a base learner (e.g. a decision tree) is supplied by an (iteratively guessed through gradient descent) correcting term in order to get closer and closer to the target. This increases the danger of overfitting, and a way in order to avoid it is to limit the number of iterations or to use a smaller learning rate. This last one can be seen as our regularization parameter which can be used on a validation subset of the training dataset in order to stop where the algorithm starts to overfit, or used directly on the test set on a grid search.
Gradient boosting may in practice be used on whichever base learner, may it be trees, neural networks or other methods. XGBoost is a development of gradient boosting when used with decision trees, and implements many of the computational and statistical tweaks in order to make the algorithm more flexible, adaptable and fast.

3 Code, implementation and testing

3.1 Program structure

The program written is this project is object oriented, meaning that the different algorithms are put together in classes. This makes it easier to structure and organize more advanced algorithms. The results in this project are obtained by first making the datasets in `make_datasets-py`, running the three `AME-xxx.py` files, and a comparison of the different methods and the published data in Utama et al. [6] is taken in `comparisons.py`. These files import and call upon other different instances of the program files. Since the code in many cases is the same as (or a slight variation

from) the one written for projects one and two [10] [11], I will simply refer to those reports when certain algorithms have already been explained.
In this section I will describe the main differences and developments since project two, for then illustrating their functioning.

3.2 Classes

When it comes to classes, most of them are exactly the same as for my previous two reports, these being classes `fit` in file `fit_matrix.py`, and class `NeuralNetwork` in file `neural_network.py`. The main addition are in the `dataset_objects.py` file, where a new child class of `dataset` has been created, namely `AtomicMasses`, which will be described in this section.

3.2.1 Datasets and AtomicMasses

The `dataset_objects.py` file contains two classes of interest, `dataset` and `AtomicMasses`, where the second is a child class of the first, with some hard-coded, ad-hoc methods for the task to be analysed in this project.
The `dataset` class has been thoroughly described in projects one and two [10] [11]. The child class `AtomicMasses` differs from dataset in a few ways, here described.
One of the main differences between this class and its parent is the fact that it will not work with other input files than the ones in the same format as the AME `.txt` files. Since the datasets not only record the binding energy but many other nuclear properties, the input `usealldata` tells the class whether to consider all these properties or not. The upside is that we have more predictors to use for the fitting, but the downside is that there is not necessarily data for all the nuclei, meaning that we would lose datapoints. Although the standard is `usealldata = True`, for this task it has mainly been set to `False`.
The `__init__` function finishes with a loop in line 290 where the datapoints in the newly made Pandas dataset are given their isotopes names. This will turn useful when comparing these against each other.
The function `AMPolishDivide` takes care of the data analysis and massaging part, as it deletes the nuclei with only estimated data, or no data at all. The condition in the input asks us whether we want to evaluate the total binding energy of nucleus, instead of its binding energy *per nucleon*. This just means multiplying B/A by the number of nucleons, and it is done in order to make the comparison to the Utama et al. [6] results easier, as this is the quantity analyzed in that paper.
The `sort_train_test` method is also a modified version of the parent class' method, as it takes as input the AME2012 dataset (we assume that this method is used on the AME2016 dataset) in order to find which nuclei are new, and these will be assigned to the test set. The condition `useAME12` asks us whether we want to use the nuclear data from the AME2012 or from the AME2016 dataset as the training set, as the data from the new one may be updated or more precise. In this task the data from the 2016 dataset has been used.

3.3 Functions

Most of the functions in the library `functions` are self-explanatory or well commented.

3.3.1 Statistical evaluations of results

Since the task at hand is a regression, I used the MSE and the R2 score to evaluate the same method for different hyperparameters in the grid searches and the different regression methods against each other. The standard deviation of the errors about the results was also used, as a way to compare the results obtained in the Utama et al. paper [6], see section 4.4.

3.4 Example run

The software is structured in this way:

1. Create the dataset objects for AME2012, AME2016 and the testset in `make_datasets.py`;
2. Run the regressions in `AME-Regression.py`, `AME-FFNN.py` and `AME-DecTrees.py`;
3. Compare the codes against each others and to the results from Utama et al. [6] in `comparisons.py`.

Each point in the list is discussed below.

3.4.1 Create the dataset objects

the datasets are created by running the file `make_datasets.py`, where all the steps are well commented. One thing to notice is the use of the `pickle` package in order to save the datasets to file, so that they can be easily opened by the other regression programs. The objects will be saved in `datasets.pkl`.

3.4.2 Run the regressions

The regression is done in the three files beginning with AME: `AME-Regression.py`, `AME-FFNN.py` and `AME-DecTrees.py`. These have the same structure, where the regression parameters are set in the header, then a grid search is carried for different values of the regression parameters, and in the end the results for the best fit are saved to a `.txt` file in order to be later opened by the comparison script. The structure here is also very similar to the one used in projects one and two, and I will rely on the thorough description in that project report, and otherwise on the comments in the script. Here I can though say that for the decision tree regression the grid search has been carried for different values of the pruning parameter α in the case of the plain decision tree algorithm, while for the XGBoost algorithm the same values are used for the regularization parameter. In the case of the FFNN, the same grid search procedure has been used as for project two, with the exception that the same architecture has been tried with two different activation functions between the layers (sigmoid and tanh) and two for the output layer (linear, and tanh). The choice of tanh for the output layer is sensible because the dataset is normalized using scikit-learn's MinMaxScaler, which constrains the values in the same region of the output of a tanh function, namely between -1 and 1. The results of the four FFNN evaluations are then saved in different files, to be later compared.
For the linear regression OLS, Ridge and LASSO were compared in a grid search for different polynomial degrees and regularization parameters λ. Since the results were always best when using OLS even for the test set, separate result files were written for the best fit when $\lambda = 0$ and for $\lambda \neq 0$.

3.4.3 Compare the results

After all the regressions are carried out, the outputs are collected by the file `comparisons.py`. This does nothing more than transforming the predictions in the same form as the results published in the paper from Utama et al.[6] and print the standard deviation of the differences between the fits and the experimental data. The regression results are then compared to the best results from the paper, namely a standard deviation of 0.479 on the differences between the predictions and the theoretical results for the BNN improvement of the Duflo-Zucker (DZ) model.

4 Results and analysis

When trying to fit to the total binding energy the results have been quite good, possibly because of the dataset being well behaved without sudden spikes or too many irregularities. Nevertheless, when trying to fit to other related quantities, such as B/A, the results went from being close to 0.999 to around 0.97 - still good, but it might show how simple modifications to the dataset, "helping" the regressor by making the data more regular and linear, could be a way of improving a fit.

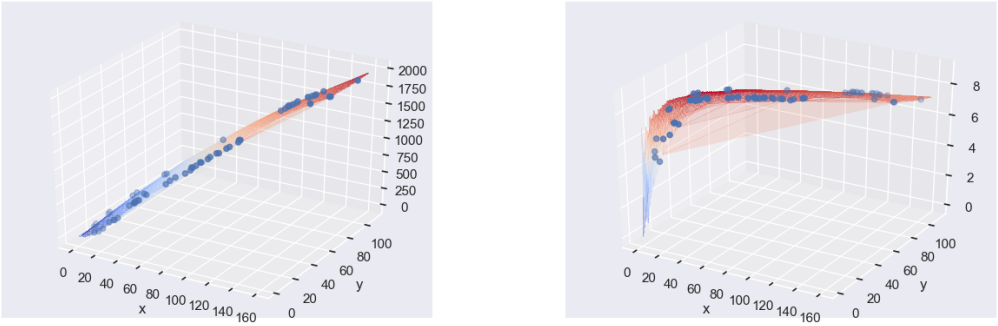

Figure 3: Plots of the best fits using decision trees, on the left to the total binding energy, while on the right to the binding energy per nucleon. Although showing exactly the same information (the binding energy per nucleon is simply the total binding energy divided by the sum of the number of protons and neutrons, i.e. the coordinate values), the fit to the graph on the right would be much better than then one to the left, even after normalization. the number of neutrons are along the x axis, the number of protons on the y axis, and the results are given in MeV.

4.1 Linear regression

The results from the grid search for linear regression are shown in the Appendix in figures 4 and 5, where we can see how the ridge regression yields better results than LASSO, but better than both does actually OLS, or when the regularization parameter is zero. This has motivated me to save the results for the OLS in a different file, so that the results for both Ridge and LASSO could be represented in the comparison between methods. When comparing the results to the experimental ones and taking the standard deviation of the differences, we would get the results in table 1.

Method	Best degree	Best λ	std (MeV)
OLS	7	0	7.885
Ridge	14	10^{-6}	7.895
LASSO	10	10^{-5}	36.280

Table 1: Overview of results from linear regression. We see how the result from ridge is very close to the OLS because of the small λ, while LASSO is completely outperformed.

4.2 Neural Networks

When it comes to neural networks, four different combinations of activation functions were tried for the same architecture (three layers of 20, 20 and 10 neurons respectively), where the sigmoid and

the tanh were used between the layers, and a linear and a tanh activation function was used for the output layer. The results are shown in the Appendix in figures 6, 7, 8 and 9. A quick view of the results tells us how the stability of the regression is much more sensible to different values of parameters than the other methods, but for values of η between 0.001 and 0.01, and regularization parameters λ below 0.01 we get useful results in line with what gotten in the other methods. The best results are collected in table 2.

act. fun.	act. fun. output	best η	Best λ	std (MeV)
sigmoid	linear	0.001	0.001	14.335
sigmoid	tanh	0.1	0.0001	19.705
tanh	linear	0.001	10^{-6}	28.932
tanh	tanh	0.01	0.01	14.077

Table 2: Overview of results from the FFNN regression. The results are quite coherent and all better than the LASSO regression, although the combined use of tanh as an activation function in between the layers and a linear output activation function somehow yields a worse result than else. The grid searches can be seen in the Appendix in figures 6, 7, 8 and 9. We see how the lowest standard deviation is yielded by the tanh-tanh neural network, this being though still almost double as much as what obtained with ridge and OLS.

4.3 Decision trees

Two algorithms were used for the decision trees part: one was the plain algorithm offered by the scikit-learn package, while the other was the XGBoost algorithm. The first one was evaluated in a grid search with varying depth and pruning parameter, while the other again with varying depth, but with different regularization parameters. These can be seen in figures 10 and 11. Both perform very well and gave the best standard deviation of all the ones tried before, as seen in table 3.

Method	Best depth	Best λ	std (MeV)
Dec. Trees	10	0	6.523
XGBoost	8	0.1	2.891

Table 3: Overview of results from the decision trees algorithms. We see how the results are both very good and better than all those tried before, with XGBoost shining against all others. The grid searches can be seen in 10 and 11.

4.4 Overview

The methods perform all quite well, even with LASSO giving a standard deviation of around 36 MeV, this corresponds to a R2 score of 0.99. Nevertheless when compared to Bayesian Neural Networks they do not compete, as the closest we get to the standard deviation of 0.479 MeV of the DZ-BNN model is with XGBoost, the best of my models, with a standard deviation of 2.891 MeV, around six times bigger. This is also worse than the purely theoretical DZ model by a factor of almost three. All data is collected and shown in table 4.

5 Conclusion

In this report I have used different machine learning techniques to find a fit of the nuclear mass data provided by the Atomic Mass Data Center. The goal was to reproduce or at least compare to

Method	std (MeV)
DZ [6]	1.018
DZ-BNN [6]	0.479
Regr. OLS	7.885
Regr. Ridge	7.895
Regr. LASSO	36.280
FFNN sig-lin	14.335
FFNN sig-tanh	19.705
FFNN tanh-lin	28.932
FFNN tanh-tanh	14.077
Dec. Trees	6.523
XGBoost	2.891

Table 4: Overview of all the results, compared to the results gotten by the paper on top. Although the XGBoost giving a very good result, is still far from competing with the theoretical-BNN methods used in Utama et al. [6], or even the purely theoretical DZ method, as shown for comparison.

the results obtained by Utama et al. [6]. While my methods were linear regression, Feed Forward Neural Networks, decision trees and XGBoost, they used Bayesian Neural Networks applied to known theoretical models. As this project took me a couple of weeks in my introductory course to machine learning and data analysis, my objective was more to see how close the different methods would get to what obtained by Utama et al., instead of actually reproducing the results.
These, collected in table 4, show that the results are good, but at best six times worse than the theoretical-BNN results, and three times worse than the purely theoretical model.
One of the reasons for this is probably the fact that only the neutron and proton numbers were used for the fitting, this not being the case for a theoretical model where actual physics is taken into account. Although more data was given in the mass tables used as dataset (such as the neutron and proton separation energies and so on), these properties were not present for all the nuclei, and many of these had to be discarded.
A possible improvement for these fitting would possibly be to try to import some of these properties (such as just the neutron or the proton separation energy) and see how the information gain from having these new features in the fitting process outweights the information loss from having to discard those nuclei where experimental values for these properties are not present.
Another possible improvement would be to add some physics to the models, and try to fit parameters from theoretical theories. It would be for example interesting to see how XGBoost would compete with BNN in the same task.

6 Appendix

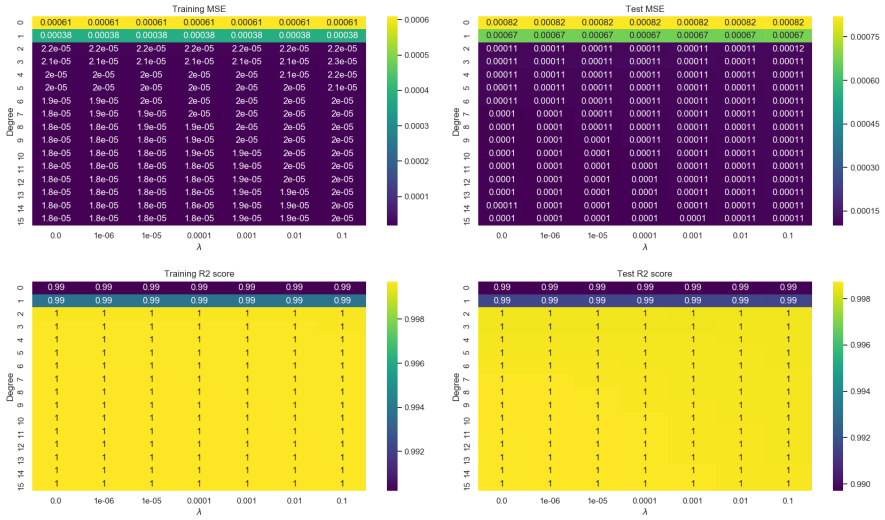

Figure 4: Grid search for the ridge regression, with varying polynomial degree and regularization parameter λ. A part for the first two lines, the fit was very good, and is more readily seen by looking at the MSE, since the R2 score was too close to one to be properly shown (the `matplotlib` package rounded everything to one, while the results are actually more vary, as one can maybe see from the slightly different shades of yellow). The best fit for the test set was for degree $= 7$ and $\lambda = 0$, so actually for the OLS solution. When neglecting the OLS solutions, the best fit was for degree $= 14$ and $\lambda = 10^{-06}$.

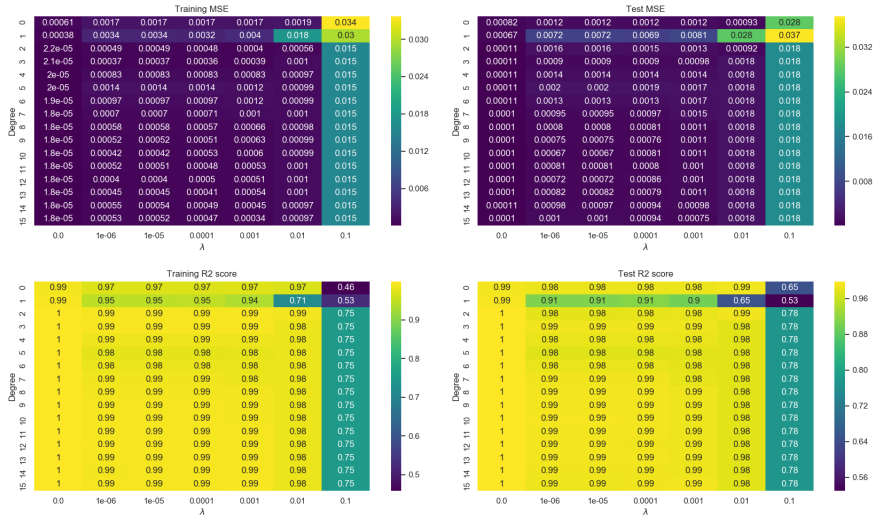

Figure 5: Grid search for the LASSO regression, with varying polynomial degree and regularization parameter λ. Here the results are more vary than with ridge, and the fit not as good. The best fit for the test set was again for degree $= 7$ and $\lambda = 0$, so actually for the OLS solution. When neglecting the OLS solutions, the best fit was for degree $= 10$ and $\lambda = 10^{-05}$.

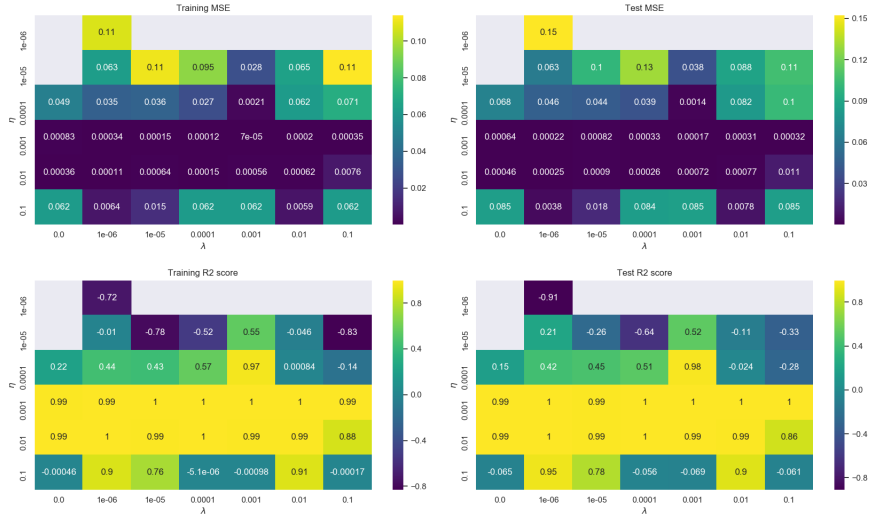

Figure 6: Grid search for FFNN, with a sigmoid activation function between the layers and linear for the output layer. The best results are for learning rate (η) values between 0.01 and 0.001, while most of the other results are way worse. The grey fields are either NaNs, or value so far off that have been ignored. The best result was obtained with $\eta = 0.001$ and $\lambda = 0.001$.

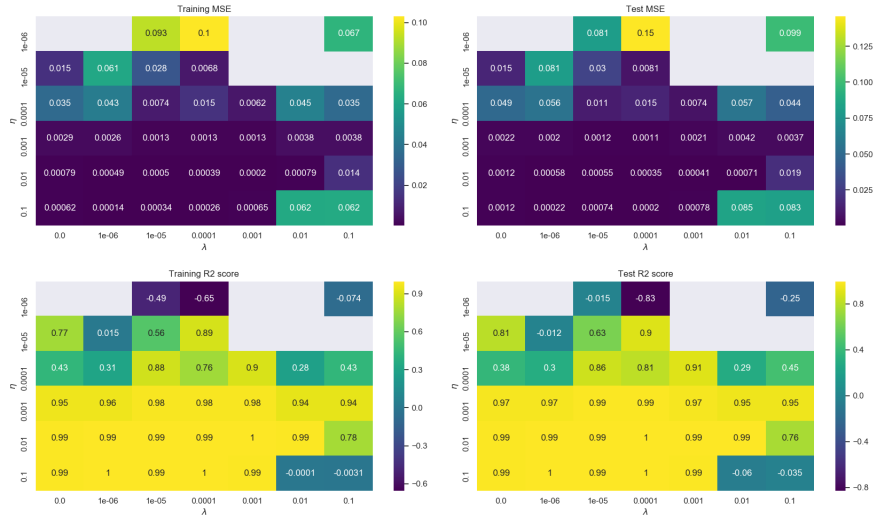

Figure 7: Grid search for FFNN, with a sigmoid activation function between the layers and tanh for the output layer. The best results are for learning rate (η) values between 0.1 and 0.001. The grey fields are either NaNs or value so far off that have been ignored. The best result was obtained with $\eta = 0.01$ and $\lambda = 0.00001$.

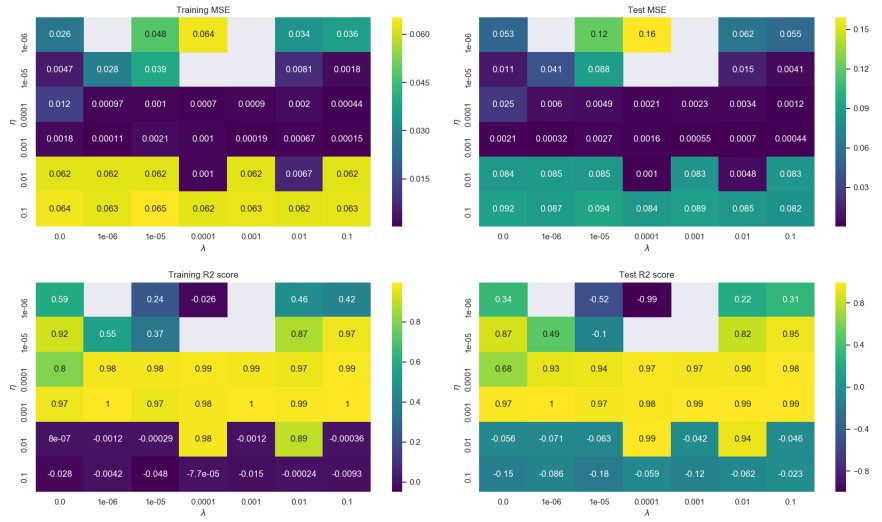

Figure 8: Grid search for FFNN, with a tanh activation function between the layers and linear for the output layer. The best results are for learning rate (η) values between 0.001 and 0.0001. The grey fields are either NaNs or value so far off that have been ignored. The best result was obtained with $\eta = 0.001$ and $\lambda = 0.000001$ (10^{-6}).

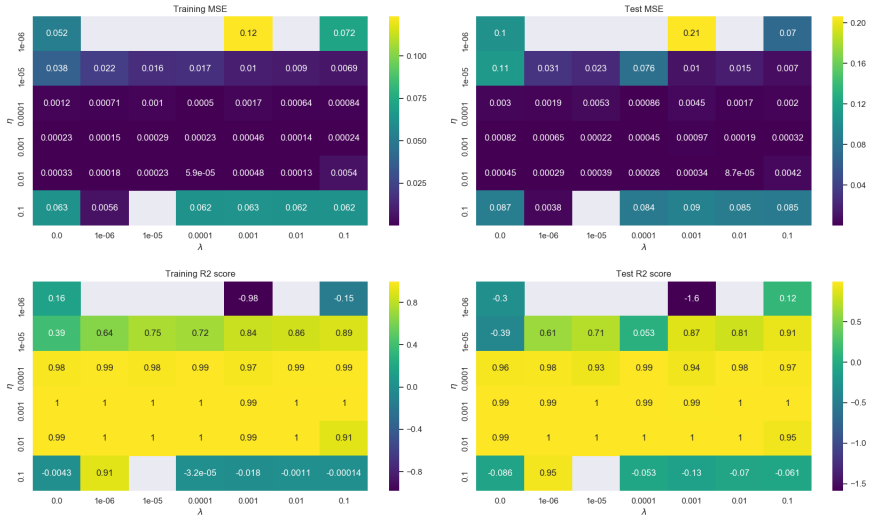

Figure 9: Grid search for FFNN, with a tanh activation function between the layers and tanh again for the output layer. The best results are for learning rate (η) values between 0.01 and 0.0001. The grey fields are either NaNs or value so far off that have been ignored. The best result was obtained with $\eta = 0.01$ and $\lambda = 0.01$.

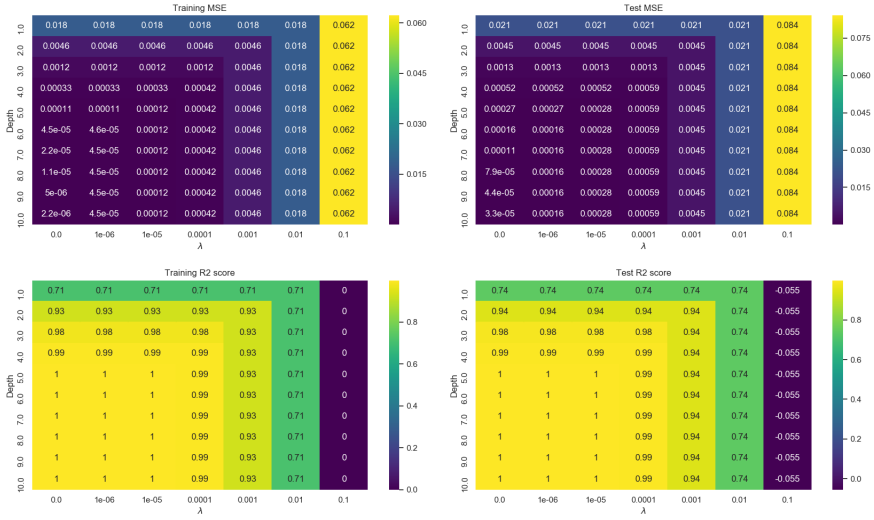

Figure 10: Grid search for the decision tree algorithm from scikit-learn, with pruning. We see at first eyesight how the algorithm is more stable than FFNN, and yields good results. The grid search has been done with varying depth and pruning parameter λ. We see how the best results are obtained generally with more depth and lower pruning parameters. The seeming correspondence between the two, is that a high pruning parameter reduces the size of the tree, giving a similar results as what gotten by building a smaller tre in the first place. The best result was obtained with depth $= 10$ and $\lambda = 0$.

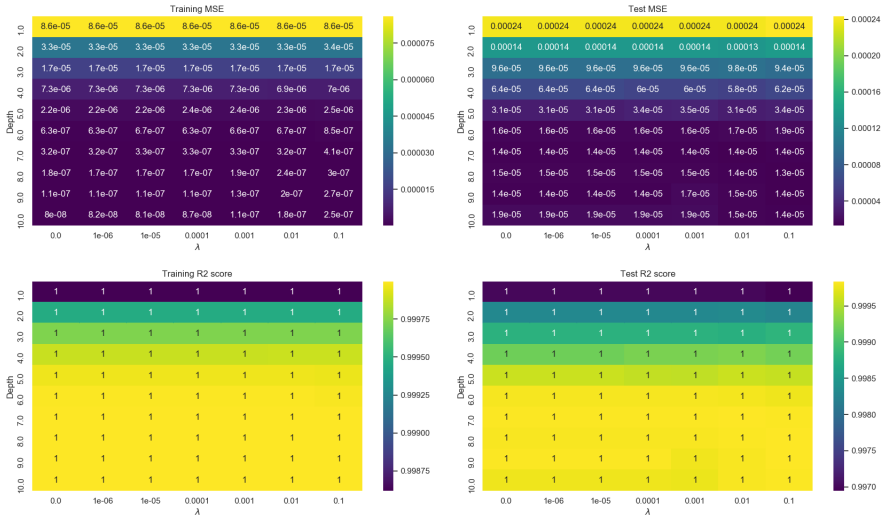

Figure 11: Grid search for the XGBoost regression algorithm. We see at first eyesight how the algorithm give very good and coherent results. The grid search has been done with varying depth and regularization parameter λ. We see how the best results are obtained generally with more depth. The best result was obtained with depth $= 8$ and $\lambda = 0.1$.

References

[1] G. Audi et al. "The Ame2012 atomic mass evaluation". In: *Chinese Physics C* 36.12 (Dec. 2012), pp. 1287–1602. DOI: 10.1088/1674-1137/36/12/002. URL: https://doi.org/10.1088%2F1674-1137%2F36%2F12%2F002.

[2] M. Wang et al. "The Ame2012 atomic mass evaluation". In: *Chinese Physics C* 36.12 (Dec. 2012), pp. 1603–2014. DOI: 10.1088/1674-1137/36/12/003. URL: https://doi.org/10.1088%2F1674-1137%2F36%2F12%2F003.

[3] Tianqi Chen and Carlos Guestrin. "XGBoost". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '16* (2016). DOI: 10.1145/2939672.2939785. URL: http://dx.doi.org/10.1145/2939672.2939785.

[4] W.J. Huang et al. "The AME2016 atomic mass evaluation (I). Evaluation of input data and adjustment procedures". In: *Chinese Physics C* 41.3 (Mar. 2017), p. 030002. DOI: 10.1088/1674-1137/41/3/030002. URL: https://doi.org/10.1088%2F1674-1137%2F41%2F3%2F030002.

[5] R. Utama and J. Piekarewicz. "Refining mass formulas for astrophysical applications: A Bayesian neural network approach". In: *Phys. Rev. C* 96 (4 Oct. 2017), p. 044308. DOI: 10.1103/PhysRevC.96.044308. URL: https://link.aps.org/doi/10.1103/PhysRevC.96.044308.

[6] Raditya Utama and Jorge Piekarewicz. "Validating neural-network refinements of nuclear mass models". In: *Physical Review C* 97 (Sept. 2017). DOI: 10.1103/PhysRevC.97.014306.

[7] Meng Wang et al. "The AME2016 atomic mass evaluation (II). Tables, graphs and references". In: *Chinese Physics C* 41.3 (Mar. 2017), p. 030003. DOI: 10.1088/1674-1137/41/3/030003. URL: https://doi.org/10.1088%2F1674-1137%2F41%2F3%2F030003.

[8] Léo Neufcourt et al. "Bayesian approach to model-based extrapolation of nuclear observables". In: *Phys. Rev. C* 98 (3 Sept. 2018), p. 034318. DOI: 10.1103/PhysRevC.98.034318. URL: https://link.aps.org/doi/10.1103/PhysRevC.98.034318.

[9] Morten Hjorth-Jensen. *Lecture notes, course FYS-STK4155*. compphysics.github.io/MachineLearning/doc/web/course.html. Department of Physics, University of Oslo et al., 2019.

[10] Francesco Pogliano. *Data Analysis and Machine Learning, Project 2*. https://github.com/Cyangray/ML-Project-2/blob/master/Project2-report.pdf. Department of Physics, University of Oslo, 2019.

[11] Francesco Pogliano and Marianne Bjerke. *Data Analysis and Machine Learning, Project 1*. https://github.com/Loopdiloop/fys-stk4155/blob/master/ML.pdf. Department of Physics, University of Oslo, 2019.