A Deeper Look into LassoNet

Sina Alsharif (z5311825)

Abstract

In the paper [1], the algorithm for a feature sparse neural network by the name of *LassoNet* is proposed. This method introduces $\ell 1$ regularisation into deep neural networks which models that of traditional Lasso. In turn, we see should see feature sparsity and a more interpretable model.

In this report, I will be diving into the ideology behind LassoNet, its implementation and testing its efficacy as a modern machine learning method. This report will be comparing its performance in various machine learning tasks against current methods and also examining the impact of its new method of regularisation. The testing is done over both a classification and regression task.

1 Introduction

Firstly, it is good to define the problem that *LassoNet* is attempting to solve and how it achieves this. From the name, *LassoNet* attempts to emulate the feature sparsity seen in Lasso regression [2] to neural networks, here we will explore this notion in further detail and outline its implementation.

1.1 Preamble - Artificial Neural Networks & Standard Problem Definition

To paraphrase, we can define a general or standard problem as function reconstruction. So say we have an original function of the data, $y = f(\mathbf{x})$ where $\mathbf{x}$ is the data and y is our *target*. We then try to construct a function $\hat{f}(\mathbf{x})$ to approximate $f(\mathbf{x})$. This is done by defining an empirical loss function and minimising this function to find the best $\hat{f}(\mathbf{x})$.

Neural networks approach this problem similar to a perceptron by adjusting weights using a gradient descent. In fact, neural networks are technically multi-layer perceptron with back propagation to adjust the weights and biases. This is just a simplified overview, the intricacies of the inner workings are not in the scope of discussion. Also note that in this paper, we will assume all neural networks to be a *feed-forward* network.

For later use, we can define important attributes of a neural network as follows:

- g_W is a feed-forward neural network with weights W
- $W^{(\ell)}$ are the weights for layer ℓ
- K is the number of units in the first hidden layer

1.2 Residual Neural Networks (ResNet)

Currently, a widely employed form of deep neural networks is the residual network. The residual network allows for the construction of deeper networks without burdens of more difficult training [3] and problems quickly degrading model accuracy.

It does this by introducing 'skip connections', which allows the output of a neuron to 'skip' past layers in the neural network architecture and influence layers deeper in the network. This architecture is typically implemented in convolutional neural networks, though we'll assume we are using a general case of it in a classic artificial neural network architecture (the reason behind this will become clear in LassoNet).

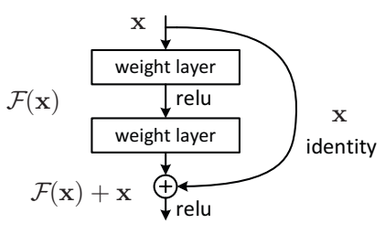


Figure 1: Basic structure of a ResNet. [3]

This means that the network is to fit a *residual* mapping of the original target mapping. So, say $\mathcal{H}(\mathbf{x})$ is the original mapping we desire to fit, we define a residual mapping as $\mathcal{F}(\mathbf{x}) := \mathcal{H}(\mathbf{x}) - \mathbf{x}$ where the original mapping can also be cast to $\mathcal{H}(\mathbf{x}) := \mathcal{F}(\mathbf{x}) + \mathbf{x}$. Then, in theory, the residual mapping should be easier to optimise [3].

To realise the residual mapping, we use the skip connections mentioned above to perform an identity mapping and add these outputs to the outputs of the layers. So, let $\mathbf{x}^{(\ell)}$ be the outputs for a layer ℓ. In the classic problem, $\mathbf{x}^{(\ell)} = A^{(\ell)}(\mathbf{x}^{(\ell-1)})$ for an arbitary non-linear transformation A used throughout the network such as ReLU. So, if we have a skip connection from layer $\ell - 1$ to ℓ, the output in the residual case becomes $\mathbf{x}^{(\ell)} = A^{(\ell)}(\mathbf{x}^{(\ell-1)}) + \mathbf{x}^{(\ell-1)}$ [4]. This therefore does not add any new parameters or complexity to the original

problem as the updates are still in terms of the previous layer's values.

In practice, this method has been proven to be highly effective and can facilitate deeper networks without classical problems such as degrading accuracy [3].

For a residual network we will denote:

- θ as the matrix of skip layer weights

1.3 LassoNet

Leveraging the architecture of residual networks, LassoNet tries to extend the feature-sparse regularisation seen in traditional Lasso regression to artificial neural networks. In turn, this will also introduce feature sparsity/selection, making the neural network more interpretable.

To acheive this, LassoNet includes an input to output skip layer which maps feature importance. If a skip layer weight for a feature reaches zero, it is not given any influence over the network (i.e its hidden unit weights are also set to zero) [1]. The proposed architecture is pictured below:

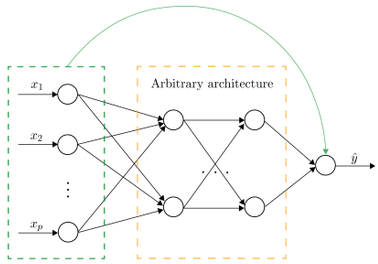


Figure 2: Proposed LassoNet architecture. [1]

To formulate the problem, we add a penalty λ to the L1 norm of the skip layer (Lasso) and also implement a proximal algorithm to optimise the objective function.

First, we note:

- $f_{\theta,W}$ as a general LassoNet architecture
- $L(\theta, W) = \frac{1}{n}\sum_{i=1}^{n} \ell(f_{\theta,W}(\mathbf{x}_i), y_i)$ as an empirical loss function (where n is the number of training samples)
- $M \in \mathbb{R}^+$ as the hierarchy coefficient, controlling the relative strength of linear and nonlinear components
- $\lambda \in \mathbb{R}^+$ as the regularisation/penalty parameter (similar to λ in classic Lasso [2])
- $d \in \mathbb{N}$ as the dimension of data

Note that we can define LassoNet as,

$$f_{\theta,W}(\mathbf{x}) := \theta^T \mathbf{x} + g_W(\mathbf{x})$$

We can then define the objective function as,

$$\arg\min_{\theta,W} L(\theta, W) + \lambda \|\theta\|_1$$

$$\text{subject to } \left\| W_j^{(1)} \right\|_\infty \leq M|\theta_j|, \quad j = \{1, \ldots, d\}$$

The constraint here enforces a proportional amount of relevance and non-linearity to feature based on the hierarchical parameter M. So, if a feature is more important (i.e θ_j is larger) it has more room to increase the size of its hidden layer weight and therefore allow the model to better represent non-linear relationships. As said before, when the skip layer weights go to zero ($\theta_j \to 0$), the hidden layer unit for that feature also goes to zero, removing the influence of this feature from the model. We can also spot that if $M \to \infty$, the skip layer will not limit the size of the hidden layer units, essentially making the model an general feed-forward neural network.

Training this network will simply involve running classic gradient descent on all of the model parameters, and then using a hierarchical proximal operator (outlined in the next section) on the skip layer and first hidden layer $(\theta, W^{(1)})$.

2 Psudeocode & Python Implementation

2.1 Python Package

LassoNet has its own experimental Python package (Github) with implementations of the following algorithms. The code base is relatively large and can be difficult to follow at a glance, I have made simpler versions of the core algorithms in the next section for illustrative purposes using the main LassoNet code as a reference.

My own fork of LassoNet is available on Github where I have added K-fold cross-validation in the training, a field to monitor validation accuracy/loss and feature importance plots. A comparision between my implementation and the original is attached in appendix D. Instructions for the installation of my fork are on the Github page, however I have included it under the name `lassonet_fork` in my submission.

2.2 Training Algorithm

Similar to the traditional Lasso, LassoNet uses proximal gradient descent to update its parameters under regularisation. However, the minimisation for LassoNet is of a non-convex function, opposed to the convex function seen in Lasso [5] and we need to define a more complex proximal operator (as seen in the next section).

To train LassoNet, we can follow the algorithm outlined in the original paper [1],

Algorithm 1 Training LassoNet

Input: training dataset $X \in \mathbb{R}^{n \times d}$, training labels Y, feed-forward neural network $g_W(\cdot)$, number of epochs B, hierarchy multiplier M, path multiplier ϵ, learning rate α
Initialize and train the feed-forward network on the loss $L(\theta, W)$
Initialize the penalty, $\lambda = \lambda_0$, and the number of active features, $k = d$
while $k > 0$ **do**
    Update $\lambda \leftarrow (1 + \epsilon)\lambda$
    for $b \in \{1 \dots B\}$ **do**
        Compute gradient of the loss w.r.t to (θ, W) using back-propagation
        Update $\theta \leftarrow \theta - \alpha \nabla_\theta L$ and $W \leftarrow W - \alpha \nabla_W L$
        Update $(\theta, W^{(1)}) \leftarrow$ HIER-PROX$(\theta, W^{(1)}, \alpha\lambda, M)$
    end for
    Update k to be the number of non-zero coordinates of θ
end while
where HIER-PROX is defined in Alg. 2

as stated before, this is simply the classic approach of gradient descent to optimise the model parameters, with an added hierarchical proximal operator to update the skip layer weights.

Instead of using auto-differentiation (i.e Pytorch, Keras) to optimise regularisation parameters, a proximal gradient descent is employed to theoretically guarantee the optimality of the solution and provide more accurate gradient computation [6].

The full code for this section is under the `_train` method of the `BaseLassoNet` class in /lassonet/interfaces.py with added batch learning and K-fold cross validation. I have made a simpler, more general version in lassonet/simplified_training.py as the class method `train_lassonet` using Pytorch. *Note*: The code is very incomplete and not as well tested as the main pipeline of LassoNet, this is just a basic demonstration of how the proposed method can be implemented.

I have modified the original implementation to add K-fold cross validation for training the model. Though this method is typically not applied in deep learning, I found a place for it in the experimentation outlined in further sections. The existing implementation used the sklearn function `train_test_split` which introduced a randomness in recreating models, where the same code and datasets would produce different models on different runs. Even though K-fold is computationally expensive, the datasets and models I am training them on here do not suffer from an unusable performance decrease.

2.3 Hierarchical Proximal Operator

First note

- $\mathcal{S}_\lambda = \text{sign}(x) \cdot \max\{|x| - \lambda, 0\}$ as the soft-thresholding operator

We then use the hierarchical proximal operator to find a global minimum of an objective of the form,

$$\underset{b \in \mathbb{R}, W \in \mathbb{R}^K}{\arg\min} \frac{1}{2}(v - b)^2 + \frac{1}{2}\|u - W\|_2^2 + \lambda|b|$$
$$\text{subject to } \|W\|_\infty \leq M|b|$$

To solve the above problem, we can define the algorithm for the operator as,

Algorithm 2 Hierarchical Proximal Operator

procedure HIER-PROX$(\theta, W^{(1)}; \lambda, M)$
    for $j \in \{1, \dots, d\}$ **do**
        Sort the entries of $W_j^{(1)}$ into $|W_{(j,1)}^{(1)}| \geq \dots \geq |W_{(j,K)}^{(1)}|$
        for $m \in \{0, \dots, K\}$ **do**
            Compute $w_m := \frac{M}{1+mM^2} \cdot \mathcal{S}_\lambda\left(|\theta_j| + M \cdot \sum_{i=1}^m |W_{(j,i)}^{(1)}|\right)$
        end for
        Find $\tilde{m}$, the first $m \in \{0, \dots, K\}$ such that $|W_{(j,m+1)}^{(1)}| \leq w_m \leq |W_{(j,m)}^{(1)}|$
        $\tilde{\theta}_j \leftarrow \frac{1}{M} \cdot \text{sign}(\theta_j) \cdot w_{\tilde{m}}$
        $\tilde{W}_j^{(1)} \leftarrow \text{sign}(W_j^{(1)}) \cdot \min(w_{\tilde{m}}, |W_j^{(1)}|)$
    end for
    return $(\tilde{\theta}, \tilde{W}^{(1)})$
end procedure
Notation: d denotes the number of features; K denotes the size of the first hidden layer.
Conventions: Ln. 6, $W_{(j,K+1)}^{(1)} = 0$, $W_{(j,0)}^{(1)} = +\infty$; Ln. 9, minimum and absolute value are applied coordinate-wise.

The algorithm can come across a little convoluted, my explanation in words is as follows:

1. Do the following for $j \in \{1, \dots, d\}$ (i.e. over all *features* for the respective data sample)
2. First sort all $W_j^{(1)}$ in descending order in terms of magnitude. This will be useful for proximal descent.
3. For all K units in the first hidden layer, calculate the next possible step in weights w_m based on the hierarchy M (this dictates the influence of previous units) and use the soft thresholding operator $\mathcal{S}_\lambda$ to optimise with respect to λ. Where

$$w_m := \frac{M}{1 + mM^2} \cdot \mathcal{S}_\lambda\Big(|\theta_j| + M \cdot \sum_{i=1}^{m} |W_{(j,i)}^{(1)}|\Big)$$

   For more on the soft thresholding operator, check out this Stackexchange post on deriving the close form solution to the tradtional Lasso. The solution shown in the post is essentially the soft thresholding operator. The optimality of w_m is proven in Appendix D of the original paper.
4. Find the first $w_{\tilde{m}}$ where $|W_{(j,m+1)}^{(1)}| \leq w_{\tilde{m}} \leq |W_{(j,m)}^{(1)}|$. This is the proximal descent step.
5. Define the optimal value for θ_j as $\tilde{\theta}_j \leftarrow \frac{1}{M} \cdot \text{sign}(\theta_j) \cdot w_{\tilde{m}}$. This updates the skip layer weights with the solution from the proximal descent above, influenced by the hierarchy parameter.
6. Define the optimal value for $W_j^{(1)}$ as $\tilde{W}_j^{(1)} \leftarrow \text{sign}(W_j^{(1)}) \cdot \min(w_{\tilde{m}}, |W_j^{(1)}|)$. Here, we update the weights in the first hidden layer with the minimum optimal value.

The final two steps make intuitive sense in the context of our original objective function, where the constraint is,

$$\left\| W_j^{(1)} \right\|_\infty \leq M|\theta_j| \quad j \in \{1, \dots, d\}$$

The updates we make here attempt to fit this constraint and therefore enforce the regularisation which is necessary for feature sparsity.

This algorithm runs in $O((dK + d) \cdot \log(dK + d))$ where d is the dimension of data and K is the number of units in the first hidden layer [1]. The mathematics and proof behind this algorithm is elegant, though much out of the scope of discussion. I highly recommend you look in the appendix D (page 24) of the original paper [1] for in-depth proofs of the optimality of this operator.

This algorithm in implemented in lassonet/prox.py under the function `prox`, though has different naming conventions from the algorithm shown above. Most of the variable names are from the solution to the problem shown at the start of this section (*2.3*) (in terms of u and v) and the definitions come in the proof in the appendix D of the original paper.

2.4 Other Intricacies of Training

Typically, $\ell 1$ regularised models exhibit bias due to regularisation [7]. LassoNet combats this by by re-fitting the model using only the features of interest and zeroing out others [1]. Also, LassoNet uses early stopping with a patience hyperparameter in its training to avoid overfitting. This can be seen in lassonet/interfaces.py (line 289-290) where the main training loop is broken when the patience is met.

Additionally, in the paper [1] (Section 4), it was found that LassoNet favours dense-to-sparse starts for optimisation, as opposed to the classic sparse-to-dense where regularisation is slowly removed to find where the model may start increasing in accuracy. This is effectively pruning the features to find the most valuable and generate a less-complex, more general predictor.

3 Experimentation

3.1 Sanity Check - Boston Housing

Before comparing LassoNet to other machine learning methods, it's good to sanity check and test the given LassoNet implementation. This also gives us an opportunity to examine the impact the regularisation of LassoNet and do a simple comparison it to other methods which also create some form of feature sparsity and selection, namely traditional Lasso regression and extreme gradient boosted trees (XGBoost).

Method

To test the feature selection of LassoNet, we will examine the importance it gives features for a dataset with 'dummy' features which add no information. This is done by using the Boston Housing dataset with added features containing random values and examining the importance of the features after the model is trained on this dataset. Additionally, we can duplicate features in the dataset to test how the model deals with multi-collinearity.

Both XGBoost and LassoNet have built in methods to retrieve feature importance, though Lasso regression does not. For the feature importance of Lasso, we can take the absolute value of the coefficients for a feature to get an approximate.

Results

The models were tested over different conditions, with different number of fake and collinear features. The format of the tests are as follows,

1. Control, 0 fake and 0 collinear features.
2. 5 fake and 0 collinear features.
3. 0 fake and 5 collinear features.
4. 5 fake and 5 collinear features.

Plots of the results and a discussion can be seen in Appendix A.

These basic tests prove the correct function of the implmenetation of LassoNet to be used throughout the experiments and also examine *how* it creates feature sparsity. We have seen that the regularisation used in LassoNet performs similar to classic Lasso in dealing with uninformative and collinear features.

3.2 Classification - Heart Disease

Now, we can compare LassoNet with other popular machine learning methods in a classification task. The task will be to predict the chance of heart attack (binary classifcation) in this Kaggle dataset on heart disease.

We will compare the efficacy of LassoNet, logistic regression, support vector machine, a classic multi-layer perceptron, random forests and extreme gradient boosted trees in the context of this task. As this is a binary classfication task, we use log loss in conjunction with classification accuracy to evaluate the quality of the models.

Results

I ran the above methods on the data after basic pre-processing (see `heart_class.py` for details) and used 5-fold cross validation. For LassoNet, I picked the model with the lowest validation loss as the final model to be tested.

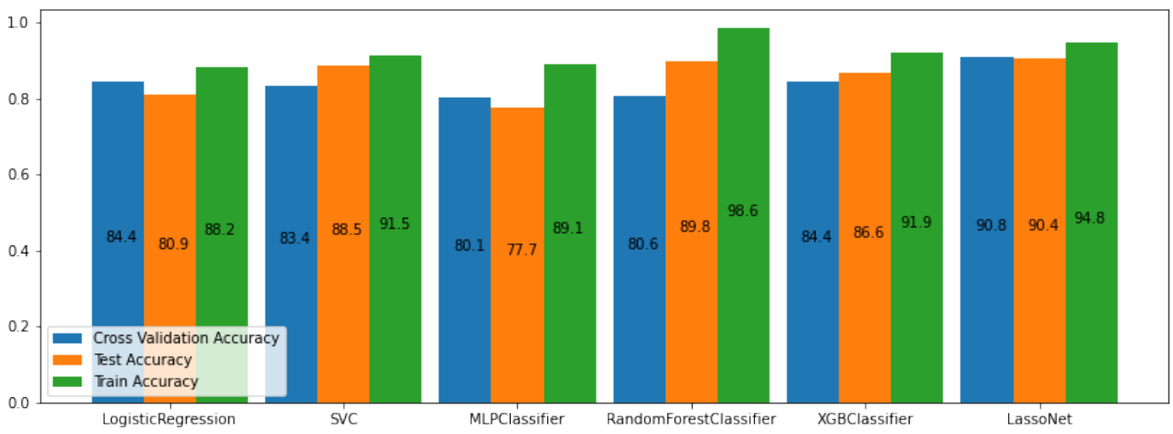


Figure 3: Model accuracies.

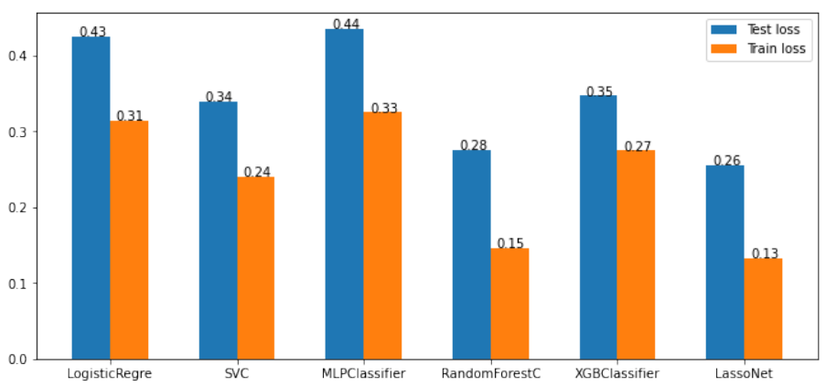


Figure 4: Model log losses.

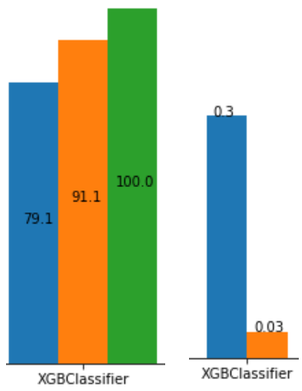
Where feasible, hyperparameters for the models were optimised using scikit-learn's `GridSearchCV`.

Analysis

From the results, we can see that LassoNet had the best test accuracy and test loss. As expected, it was by far the most computationally expensive to train and is the most complex model, with 6 hidden layers in the final model. However by design, it does not entirely sacrifice interpretability. Furthermore, LassoNet produced one of the most general models, with equal smallest difference between the validation accuracy and test accuracy.

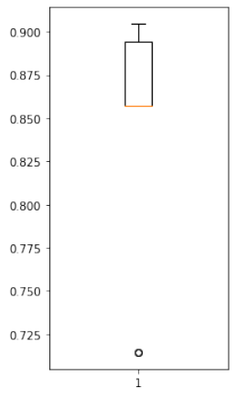
Also, neural networks, especially residual networks have been proven to effectively balance bias and variance [8].

Close behind LassoNet in terms of test loss and accuracy are the Random Forest and XGBoost classifiers. Compared to the other methods (Logistic Regression, MLP and SVM), the bagging and boosting used in the respective methods contribute to their generality by design, however they did not generalise very well in the testing without thorough hyperparameter tuning. It was difficult to find parameters allowed for errors in training (reducing overfitting) and did not make considerable impact on the validation or testing accuracy. For example, we can find the following XGBoost model with 91% test accuracy, though low validation accuracy.

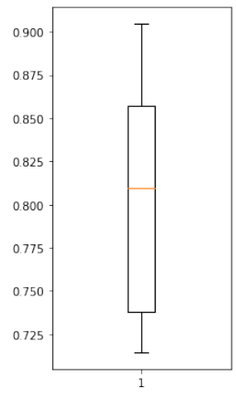


(a) Accuracy for overfit XGB

(b) Loss for overfit XGB



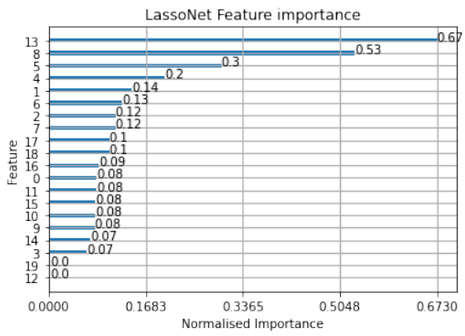
(c) Validation accuracy for final model



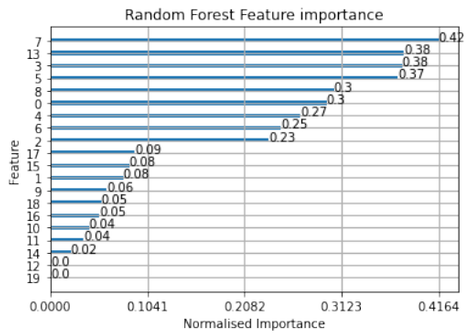
(d) Validation accuracy for overfit model

This is a textbook example of the bias-variance tradeoff, where by pure chance, the high variance (and consequently low bias) of the overfit model allowed it to achieve a better test accuracy than the model used for comparison. We can spot this variance in the plots of validation accuracies (in 10-Fold cross validation) shown above. It is vital we choose the most general model, and not just the model which performs the best on the test set.

In terms of interpretability, the decision tree based models (Random Forests and XGBoost) and logistic regression were the closest to LassoNet, all provide some indication of the importance of the features used for prediction. Though, the support vector machine and multi-layer perceptron do not provide grounds to easily interpret how they predict their results and act mainly as a black-box. The main feature of LassoNet is that it provides interpretability for complex deep networks and tries to remove the notion of a 'black-box'.



(e) Feature importance for LassoNet.



(f) Feature importance for Random Forest.

For example, in the plots above, we can compare the feature selection of the random forest model and LassoNet. We see that the feature selection is quite similar, with (7, 13, 5, 8, 3) appearing in the top 10 weighted features in both models and both models giving no weighting to feature 12 and 19. Such comparisons would not be easily achievable in a classic feed-forward deep neural network. I have attached the feature importance for models where it is possible in Appendix B for further reading.

In this test, we have seen the efficacy of LassoNet compared to classic machine learning methods and examined the usefulness of its interpretability. The sheer complexity of the deep network used was not burdened with poor interpretability.

3.3 Regression - Pyrim

To understand the full extent of LassoNet's capabilites, we can task it with non-linear regression. The task is to predict the activity of antibiotics from this OpenML dataset. To make the dataset more complex, we add Gaussian noise and 3 uninformative features.

Again, we will be pitching LassoNet up against current machine learning methods, namely random forests, K nearest neighbours, XGBoost and a multi-layer perceptron.

Results

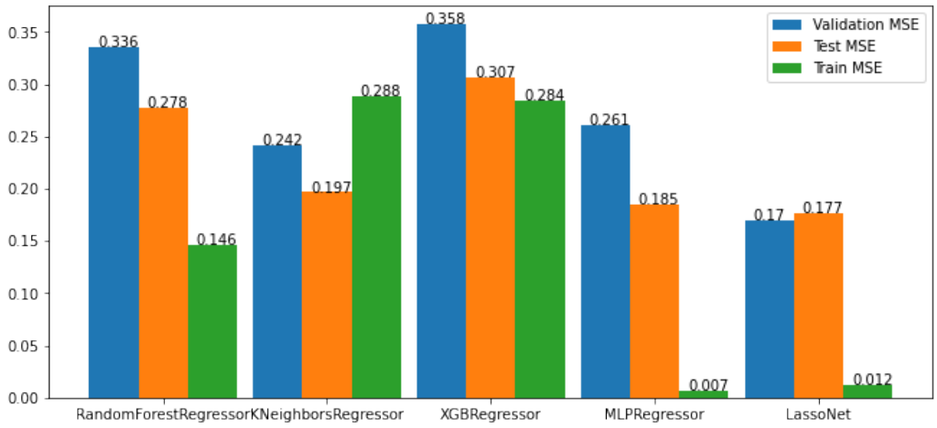


Figure 5: RMSE for models.

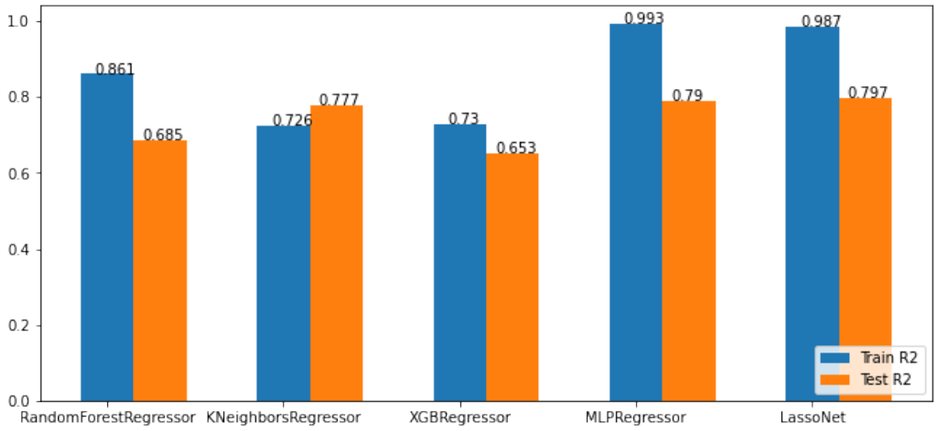
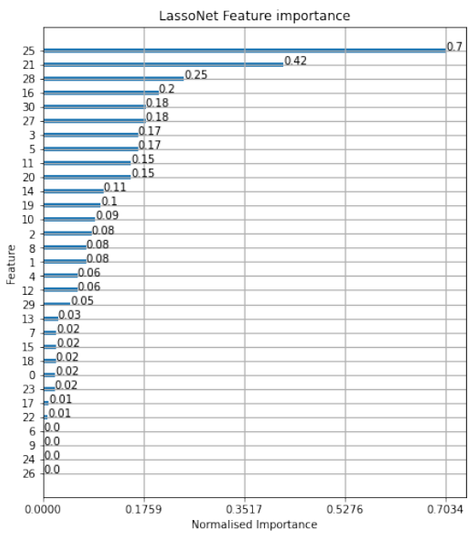


Figure 6: R^2 Score for models.

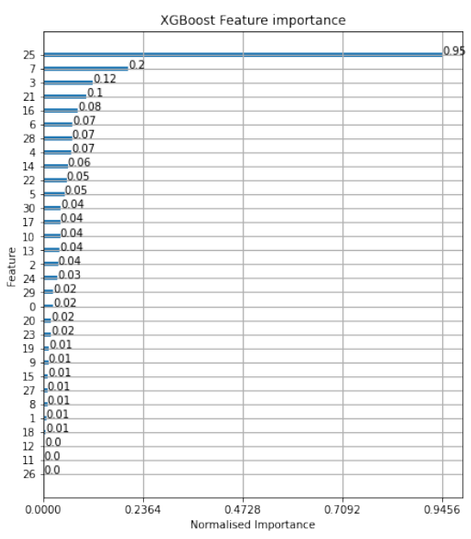
Analysis

Note: Here I took the *median* cross validation scores as there were many outliers, causing unreasonable mean scores.

In this test, LassoNet performed the best, with the highest test R^2 and a general model with only a small deviation between its train and test scores. With regularisation, the model allows for errors in training and consequently creates a more general model which does not fit noise or fake features. LassoNet and the classic multi-layer perceptron performed similarly. This is likely because I used a relatively high hierarchy multiplier ($M = 100$) to achieve the best model. However, LassoNet provides interpretability which the MLP does not have. This again allows us to analyse how the model is making its predictions and what features it uses.

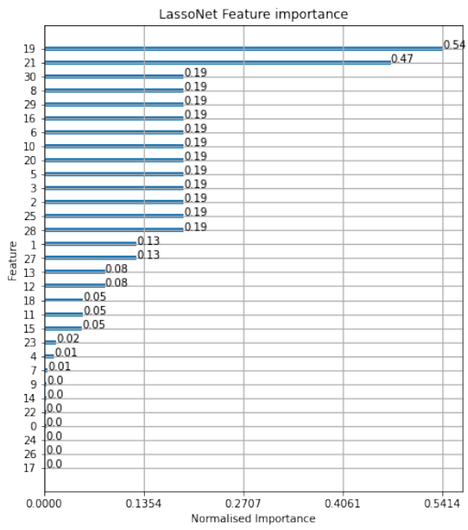


(a) Feature importance for LassoNet.

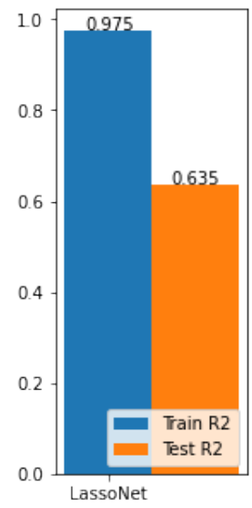


(b) Feature importance for XGBoost.

We can see in the plots above that LassoNet did not deal with the fake features (27-30) as well as XGBoost, putting features (27, 28, 30) in its 10 most important, opposed to only 28 appearing in XGBoost's top 10. The pruning and regularisation in XGBoost therefore allows it to identify informative features better than the proposed method in LassoNet. However, as we have used a high value of M, LassoNet is closer to a feed-forward neural network with as the skip layer does not have great influence (Section *1.3*). Though, even with a lower hierarchy parameter of $M = 16$, the model still gives weighting to fake features (28, 29, 30) and the overall accuracy of the model decreases.



(c) Feature importance for LassoNet with $M = 16$.



(d) R2 Scores for LassoNet with $M = 16$.

We can see above that the result is an overfit model, with no improvement in feature selection.

Surprisingly, the random forest did not perform as well on the test set in comparison with other methods. Though, it filtered fake features better than LassoNet and even XGBoost, not including any fake features in its top 10 (plot included in Appendix C). We can infer that the pruning methods used in the trees in random forest and XGBoost can be more effective than LassoNet at identifying uninformative features. It was a similar story of underacheivement in test scores for XGBoost.

There is not much to discuss about K nearest neighbours. It performed well (falling just short of the MLP) though is a method with high variance, especially with the number of neighbours that was optimal ($k = 2$). This method acts more as a baseline as it is the most basic method tested.

In this test of regression on a noisy dataset, we have seen that traditional tree pruning methods can be more effective than the LassoNet method in filtering uninformative features. However, the inherent complexity of LassoNet allows it to stay competitive with other estimators.

4 Conclusion

In this paper, we have been introduced to a new method of regularisation in neural networks which allows for enhanced interpretability (*LassoNet*). The new methods which have been introduced served well in our first experiment on classification and showed similarity to traditional Lasso in feature selection in the sanity check. In the second experiment however, the proposed method did not perform better than existing methods (XGBoost and random forests) in selecting informative features to make general predictions. This method is an innovative take on a new classic, and I feel may serve as a basis for new methods which try to demystify the predictions of deep neural networks.

5 Further Work

Personally, I would have loved to try using LassoNet in a stacking ensemble method to further analyse its value in modern machine learning. Though, I did not have the resources, time or expertise available to do this. This is definitely something I will look into in the near future.

Also, I would have loved to see a translation of LassoNet's regularisation in a more complex architecture such as a recurrent network with LSTM cells. This would be helpful in identifying what features are important in a time series task (i.e stock prediction) and give business facing engineers more insight into how their predictions are made.

Bibliography

[1] I. Lemhadri, F. Ruan, L. Abraham, and R. Tibshirani, "Lassonet: A neural network with feature sparsity," *Journal of Machine Learning Research*, vol. 22, no. 127, pp. 1–29, 2021.

[2] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society (Series B)*, vol. 58, pp. 267–288, 1996.

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.

[4] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," 2018.

[5] A. Beck and M. Teboulle, "A fast iterative shrinkage-thresholding algorithm for linear inverse problems," *SIAM journal on imaging sciences*, vol. 2, no. 1, pp. 183–202, 2009.

[6] F. Latorre, P. Rolland, N. Hallak, and V. Cevher, "Efficient proximal mapping of the 1-path-norm of shallow networks," 2020.

[7] J. D. Lee, D. L. Sun, Y. Sun, and J. E. Taylor, "Exact post-selection inference, with application to the lasso," *The Annals of Statistics*, vol. 44, Jun 2016.

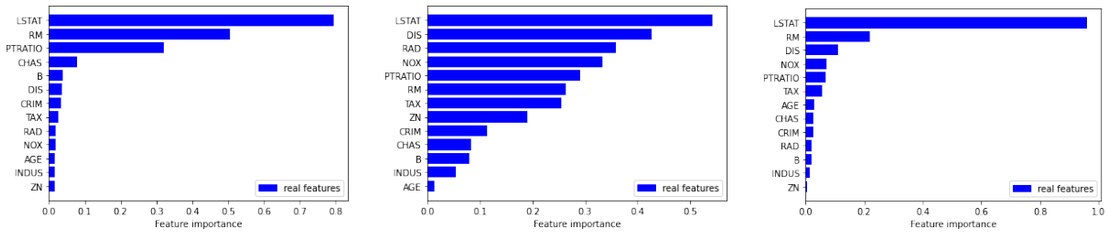
[8] Z. Yang, Y. Yu, C. You, J. Steinhardt, and Y. Ma, "Rethinking bias-variance trade-off for generalization of neural networks," 2020.

Appendix A

Sanity check results and analysis

** FI: Feature Importance*

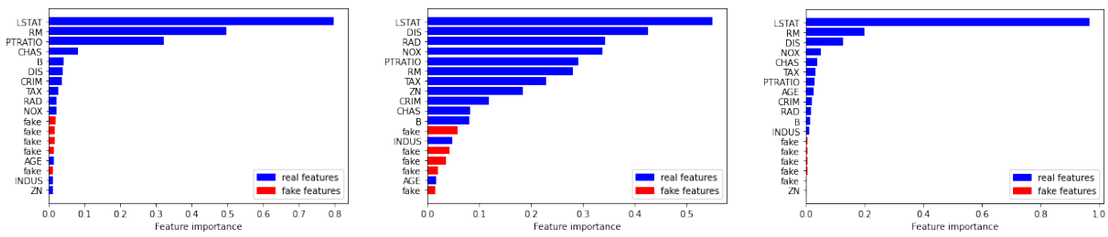
Test 1 0 fake and 0 collinear features. First, a baseline test with no fake or collinear features.



(a) FI for LassoNet (b) FI for Lasso Regression (c) FI For XGBoost

Figure A.1: No fake or collinear features.

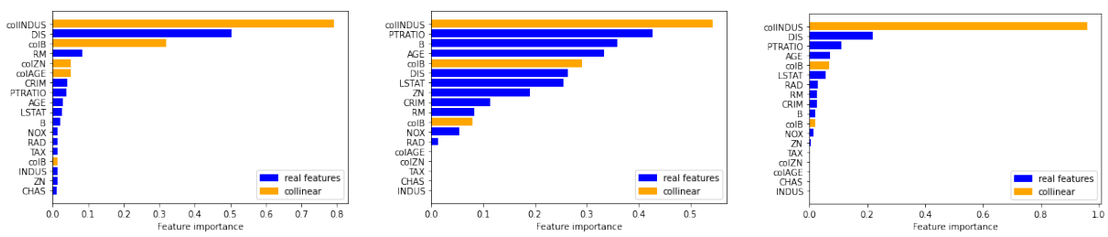
Test 2 5 fake and 0 collinear features.



(a) FI for LassoNet (b) FI for Lasso Regression (c) FI For XGBoost

Figure A.2: 5 fake features.

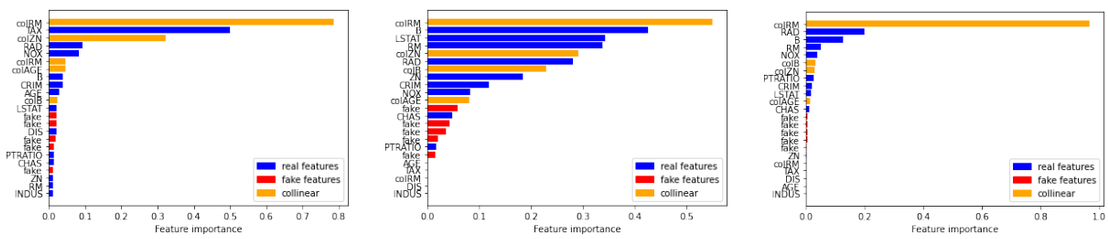
Test 3 0 fake and 5 collinear features.



(a) FI for LassoNet (b) FI for Lasso Regression (c) FI For XGBoost

Figure A.3: 5 collinear features.

Test 4 5 fake and 5 collinear features.



(a) FI for LassoNet (b) FI for Lasso Regression (c) FI For XGBoost

Figure A.4: 5 fake and 5 collinear features.

Analysis

From all 4 tests, we can see that the implementation of LassoNet works to create feature sparsity similar to other regularised methods.

Test 1 just acts as a baseline or control for the other tests. From this test, we can spot that all models favour the LSTAT feature as the most important, with RM and PTRATIO appearing in the top 5 features for all models.

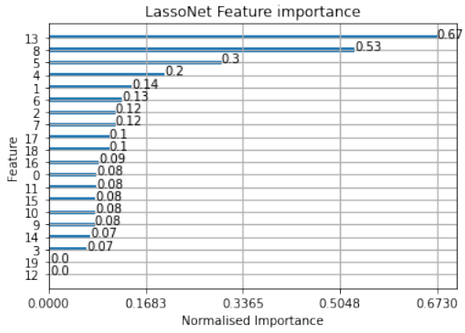
In test 2, we can see that all three methods do well in filtering out the influence of the fake features on the final model. The fake features all have low weightings and appear in the bottom 10 of most important features for all three models. Again, we cannot necessarily judge the 'scale' of importance for the three models as they calculate weighting differently, though their rankings of fake features are very similar. Also, the most important features do not differ greatly from the control test, with LSTAT appearing as the most important for all models, alongside RM and PTRATIO in the top 5.

Following on, in test 3 we can see that all 3 models deal with multi-collinearity similarly. In all models, the duplicate of the collinear features have inverse importance as expected. For example, colINDUS (the dupliate of INDUS) is given a very high importance in all models with its duplicate INDUS having almost no importance. The top features differ from the control, with INDUS ranking as the most important across all models, though PTRATIO still appearing in the top 5 across all models.

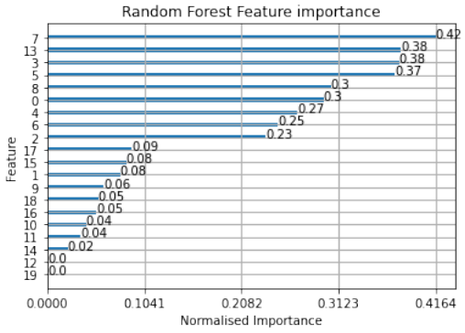
In the final test (test 4), the models again exhibit similar behaviour their choice of most important feature. However, the models all dealt with other features differently. Lasso regression picked both RM and one of its duplicates RM in its top 4 features, though it dealt with the other duplicate well, giving it no weighting. Lasso regression also dealt reasonably well with the fake features, placing them in the lower half of importance. LassoNet also struggled with mutli-collinearity, placing both instances of colRM (RM copies) in the top 6. However, with different collinear features (ZN) for instance, it worked as expected and placed negligible importance on one of the instances. LassoNet also dealt with fake features well, placing them all in the lower half. Finally, XGBoost performed reasonably well compared to the competition, removing fake features favourably, though had trouble dealing with multi-collinearity, placing colRM, RM and colB, B all in the top 6.

Appendix B

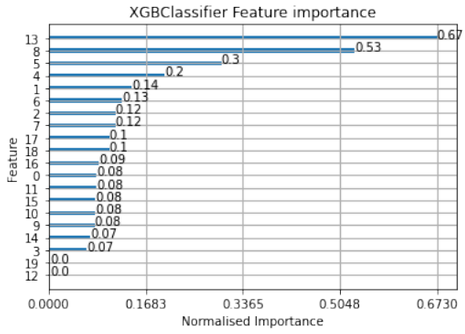
Feature importances for Heart Classification



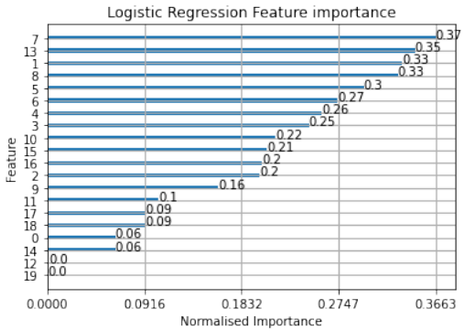
(a) LassoNet feature importance.



(b) Random Forest feature importance.



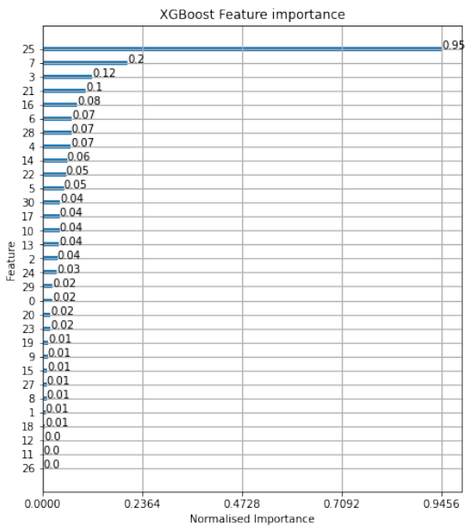
(c) XGBoost feature importance.



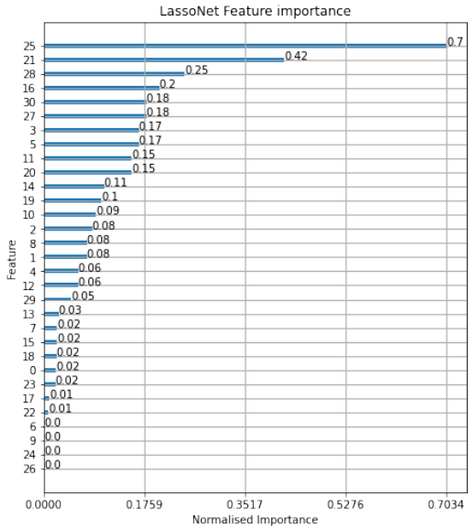
(d) Logistic Regression feature importance.

Appendix C

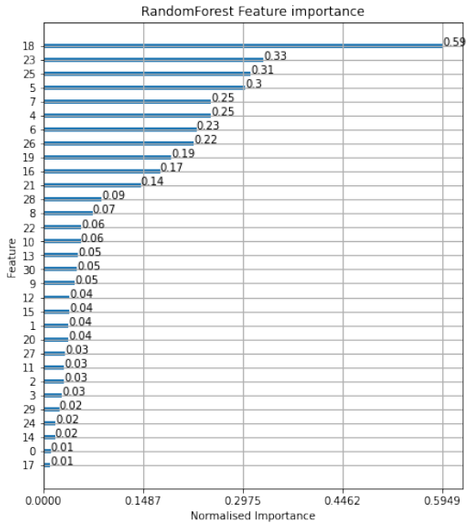
Feature importances for Pyrim Regression



(a) XGBoost feature importance.



(b) LassoNet feature importance.



(c) Random Forest feature importance.

Appendix D

Comparison of original and my implementation

Note: the model being trained here is complex, hence some erratic behaviours. The first plot (score vs number of selected features) may not serve useful for analysis, as the model uses the `backtrack` option to improve at every step.

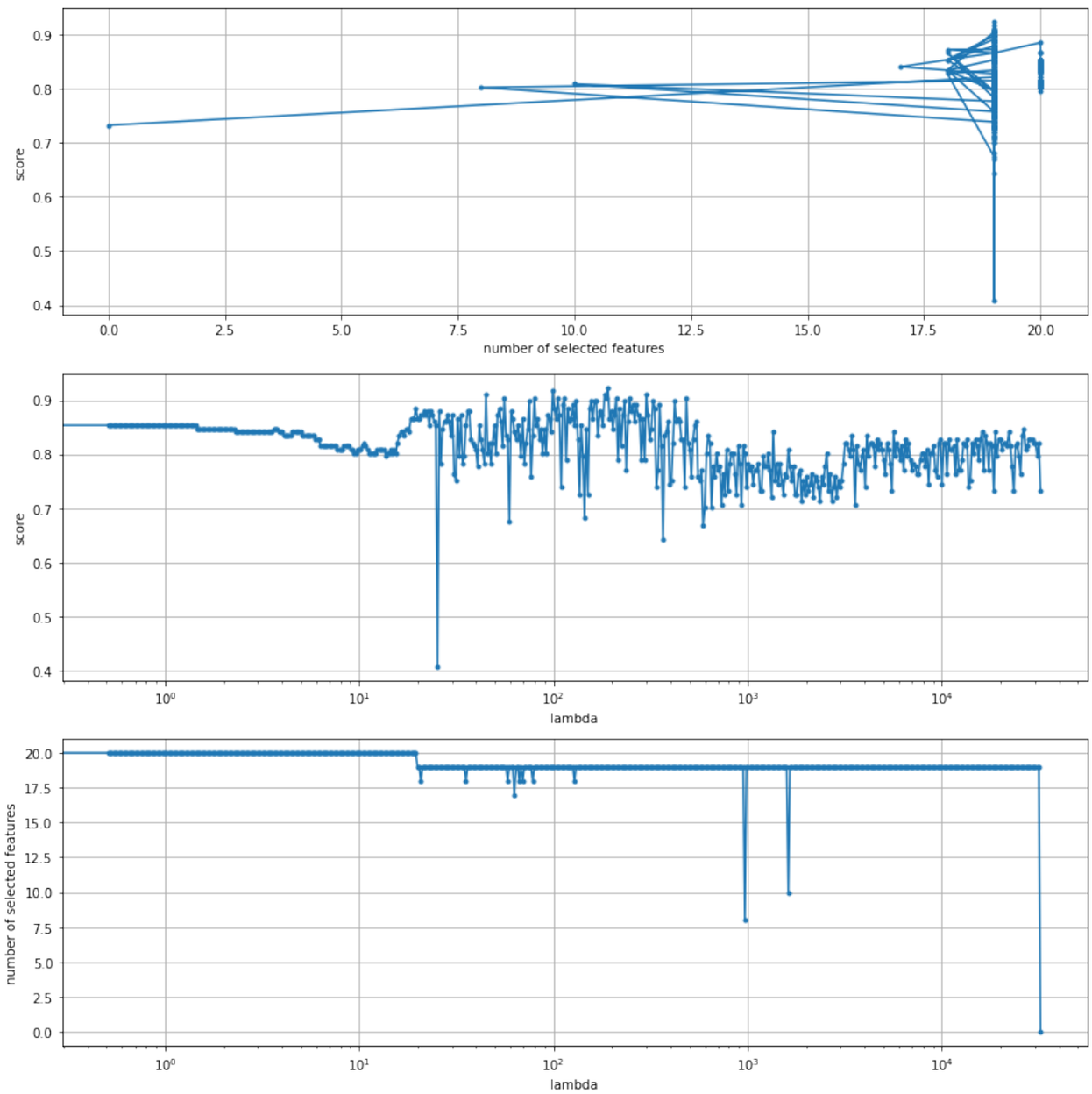


Figure D.1: Path plot of original implementation (80-20 split).

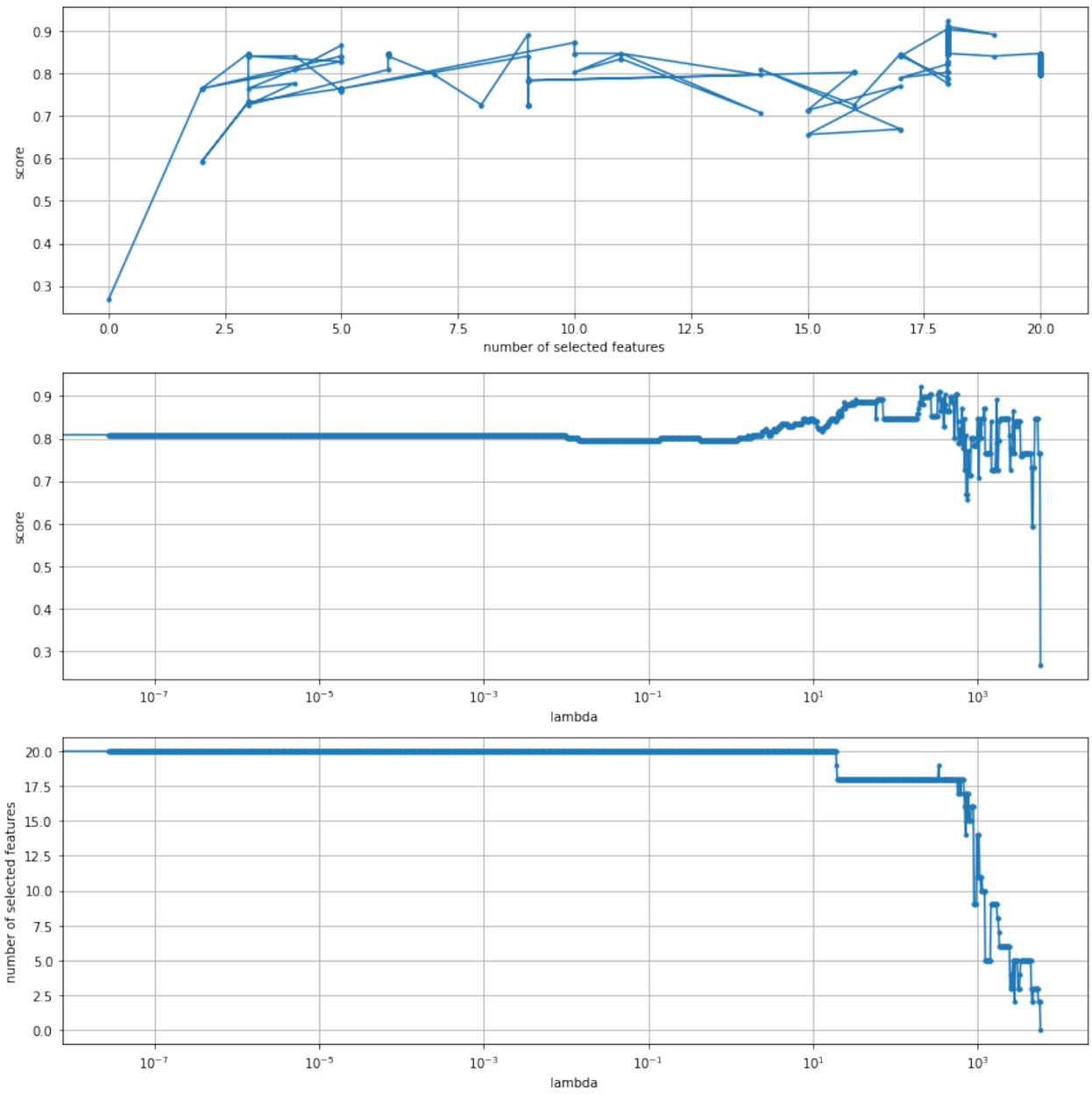
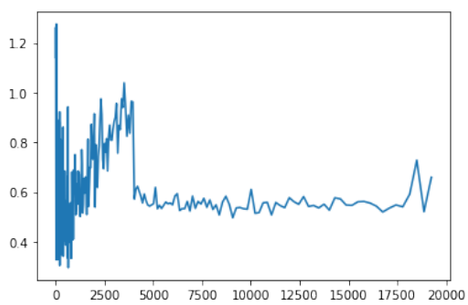
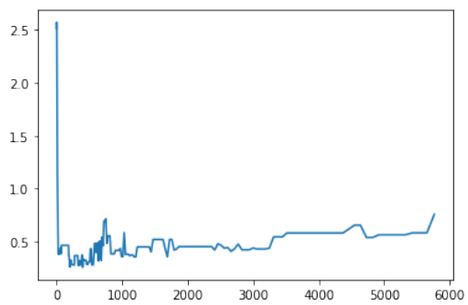


Figure D.2: Path plot of my implementation (5-fold cross validation).



(a) Log loss of original path.



(b) Log loss of my implementation's path.