Untitled

2025-09-28

Methods

For the modelling section of the project, **keras** was used for the training, tuning and testing. **keras** is an open-source python library that allows users to to build Neural Networks. It is known for modularity and it's relatively gentle learning curve compared to other libraries such as **pytorch**. The modularity makes **keras** a good library for projects requiring lots of experimentation such as this one. It should be noted that when working with **keras** in R, all the functions used are wrapper functions of the native python functions. This makes version control extremely important. For this project, the models were created, trained and tuned using R version 4.5.1 (2025-06-13) and python version 3.11.6.

The goal for this project was to train a neural network to predict the forecasted avalanche hazard(FAH). Since we are predicting the forecast, the observed avalanche hazard(OAH) needs to be removed otherwise the model would be trained on information that would not be available on new data, i.e, data leakage. Therefore the predictor sets are explicitly defined so that there is no data leakage.

A 70/30 training split was used in the data. Usually we would set aside a small portion for validation but as we will see later, **keras** handles this for us so we do not need to specify a validation set. Using a seed for this step is important as it ensures the split remains every time the code is ran. This is the first step toward achieving repeatable results.

Table 1: Comparison of percentage of each category in the training and test sets

	category 0	category 1	category 2	category 3	category 4
train	0.328	0.303	0.236	0.088	0.044
test	0.320	0.312	0.236	0.089	0.043

The table above reports the percentage of each category of FAH that is in the training and test data. It is important that the percentages are approximately equal in both sets and indeed this is the case. This is good but there is a problem. For the best results there should be approximately the same number of observations from each category in both the training and test sets, which is not the case for this data. Category 0 and 1 both are roughly the same at roughly 30% of total observations each. Category 2 is slightly lower with 23% but category 3 and 4 are hugely unrepresented in the data with only 9% and 4% respectively. This is an issue because big imbalances such as this will cause the model to optimise for correctly identifying observations from categories 0, 1 and 2.

There a few possible methods we can use to mitigate this issue but the method chosen for this project was to define custom class weights so that incorrect predictions have different penalties for each category. The idea is that underrepresented categories have higher weights so that in the training the model gets penalised more heavily and hopefully the final model is better at making predictions for the underpresented categories. The weights chosen were inversely proportional to the percentage of appearance in the training data. A possible area of further experimentation is using bootstrap sampling to equalise the percentages of each category.

For this project there were 4 predictor sets that were evaluated. Predictor set 1 contained variables relating to the location of the observation. Predictor set 2 contained variables relating to the weather conditions of

the observation. Predictor set 3 contained variables relating to a snow pack test. Predictor set 4 contained all the varibles in the dataset and thus contained all three predictor sets inside it. The final model will be built using all the available data, so predictor set 4. The remaining three predictor sets serve as yardstick to which we can compare which sets of variables are important.

Defining the model builder wrapper function

Neural networks are extremely flexible and configurable models. But this configurability means that there are several parameters that need to be defined before the model can be trained. In order to find the optimal configuration, hyperparameter tuning needs to be done first. Some of the hyperparameters we have control over include: number of layers, number of nodes per layer, the acitvation function to use on each layer, the dropout rate and the learning rate. **`keras`** is flexible and allows the user control of almost every aspect of the model through the parameters. In order to do the hyperameter tuning in R, the **`kerastuneR`** library is used. But in order to do the hyperparameter tuning with **`kerastuneR`** the model needs to be wrapped in a function that accepts as an input the hyperparameter configuration, and outputs a model.

For this project, it was decided to tune for the number of layers, number of nodes on each layer and the learning rate. The activation function was chosen to be rectified linear units(Relu) and the dropout rate was set to 0,1. Each dense layer was followed by a dropout layer. The specific values that were tuned across were 1-5 layers(step size of 1) with 30-50(step size of 10) nodes on each layer and 5 equally spaced learning rates

$$0,01 \ -0,0001$$

. Therefore, there were a total of 75 unique models that could be fitted. The metric the Neural Network tries to minimize is the **`metric_categorical_accuracy`**. This was chosen since the target variable has more than 1 category.

Hyperparameter tuning

`kerastuneR` has multiple tuning algorithms, we have used the **`RandomSearch`** algorithm. **`RandomSearch`** takes random combinations of the provided hyperparameters and fits the model each time. Since the combination of hyperparameters is random, there is a possibility that the same mode configuration is ran multiple times by the algorithm. The algorithm does attempt to mitigate this but it not guaranteed to stop duplicate runs. We do have some control over this though by setting the **`max_trials`** variable to the total number of unique models that can be specified from our selected tuning ranges. It has also been specified that each model should be fit 3 times by setting **`executions_per_trial = 3`**. This reduces variation in the results since there is an element of randomness in the initialisation of the model. A validation split of 20% was used and **`shuffle = T`** was used. Doing this shuffles the which observations get used as the validation set. This helps reduce the chances of overfitting.

All the results from tuning were saved into folders so that the results can be extracted and used for further analysis. The tuning was was undertaken is by no means exhaustive, no tuning can ever be, but the range of values tuned over is relatively small and therefore the results should be taken with a pinch of salt. With more time and perhaps more compute power a better result is possible.

Tuning results

`kerastuneR` saves the tuning results as .json files. Each trial will be its own folder and inside that folder there will be a .json file containing information about the configuration of the Neural Network and the validation accuracy it achieved. The results were compiled into a single table containing the top 3 configurations from each of the predictor sets.

Systematic hyperparameter tuning across four predictor sets revealed distinct performance patterns. As summarized in Table @ref(tab:tuning_table), Predictor Set 2 (weather conditions) achieved the highest

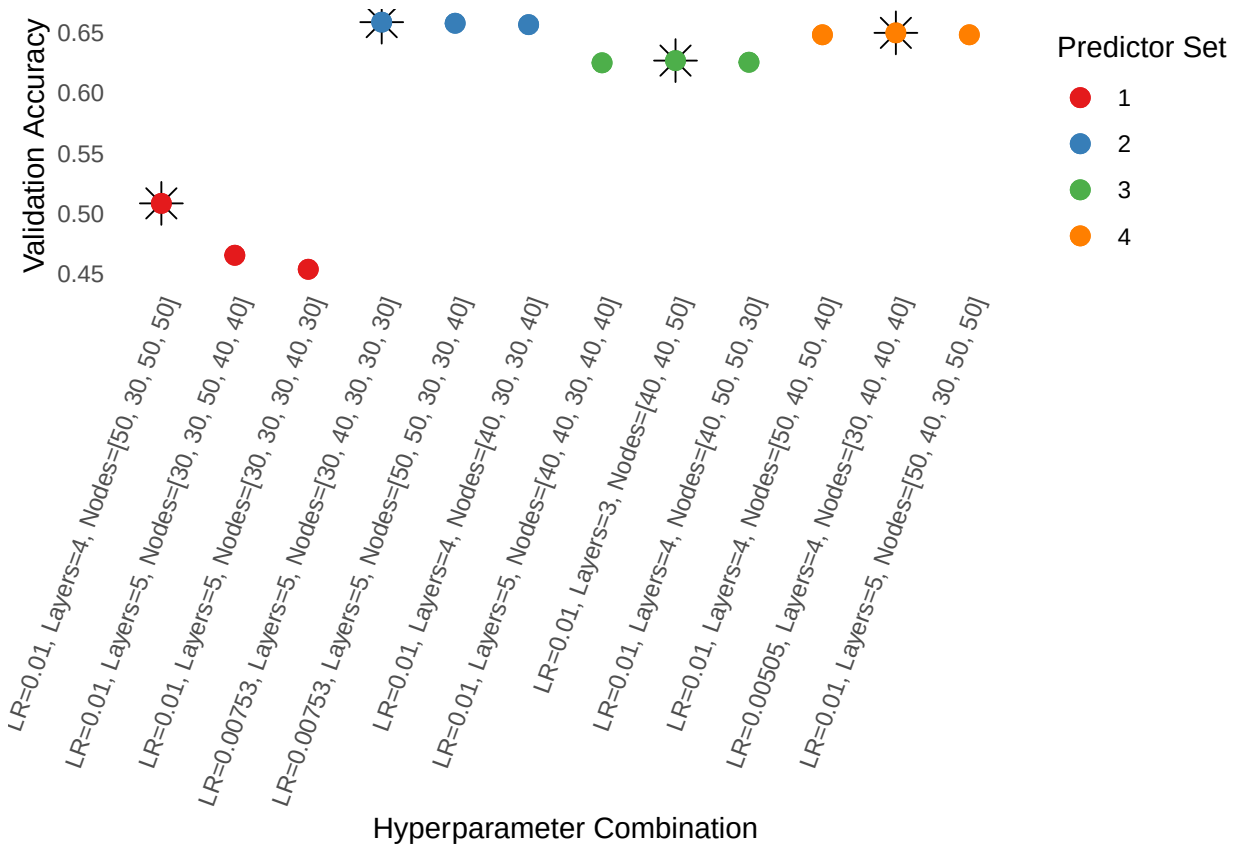

Figure 1: Validation accuracy by hyperparameter configuration across four predictor sets

validation accuracy (66%), slightly outperforming Predictor Set 4 (all variables) at 65%. This suggests that weather variables capture the most critical signals for avalanche hazard forecasting, with topographic and snow-pack variables providing only marginal incremental value.

Table 2: Top hyperparameter configurations per predictor set, ranked by validation accuracy.

Predictor set	Validation accuracy	Learning rate	Number of layers	nodes on layer 1	nodes on layer 2	nodes on l
1	0.50829	0.01000	4	50	30	
	0.46535	0.01000	5	30	30	
	0.45376	0.01000	5	30	30	
2	0.65826	0.00753	5	30	40	
	0.65735	0.00753	5	50	50	
	0.65621	0.01000	4	40	30	
3	0.62645	0.01000	3	40	40	
	0.62509	0.01000	4	40	50	
	0.62463	0.01000	5	40	40	
4	0.64940	0.00505	4	30	40	
	0.64781	0.01000	4	50	40	
	0.64781	0.01000	5	50	40	

Architecturally, models with 4–5 layers and 30–50 nodes per layer consistently outperformed others, with no clear gains beyond 5 layers. The optimal configuration from Predictor Set 4—selected for final evaluation—employed 4 hidden layers with

$$30, 40, 40, 40$$

nodes and achieved 64.9% validation accuracy.

The selected architecture (Figure @ref(fig:best_model_plot)) utilizes ReLU activation in hidden layers and softmax output activation, appropriate for the multi-class ordinal nature of avalanche hazard prediction. This configuration represents an optimal balance between model complexity and predictive performance for the comprehensive feature set.

Table 3: Confusion matrix

	Predicted				
Predicted	0	1	2	3	4
0	857	420	88	15	10
1	118	326	219	51	7
2	29	231	413	186	94
3	0	4	18	20	16
4	0	0	4	7	8

The confusion matrix of the the fitted model is reported above. It is clear that category 0 is the best estimated while categories 3 and 4 are estimated the worst. This result was expected since category 3 and 4 are very underepresented in both the training and test data. Concrete metrics of the model performance is given in the table below.

Table 4: Class metrics

	Sensitivity	Specificity	Pos Pred Value	Neg Pred Value	Precision	Recall	F1	Prevalence	Detecti
Class: 0	0.854	0.751	0.617	0.916	0.617	0.854	0.716	0.320	
Class: 1	0.332	0.817	0.452	0.729	0.452	0.332	0.383	0.312	
Class: 2	0.557	0.775	0.433	0.850	0.433	0.557	0.487	0.236	
Class: 3	0.072	0.987	0.345	0.916	0.345	0.072	0.119	0.089	
Class: 4	0.059	0.996	0.421	0.959	0.421	0.059	0.104	0.043	

Sensitivity gives the percent of the time the model predicted an observation as belonging to a category and it actually belonging to that category. The sensitivity for class 0 is the best with a value of 0,854. There is a steep drop off for the other classes but class 2 is the second highest with a value of 0,557 and the rest are all below 0.5 meaning that more often than not, the model is unable to identify the correct category. An analogy for sensitivity is a test with high sensitivity(close to the maximum of 1) will identify most of the patients with with the flu as having the flu but this may mean that lots of patients without the flu also get identified as having the flu. An extreme case may be if the model predicted all observations as belonging to category 0. Then the sensitivity would be 1,meaning that all observations that belong to category 0 are predicted to be category 0. So there need to be a balance because we do not want this

Specificity gives the percent of time the an observation does not belong to a specific category and the model predicts it as not belonging to that category. High specificity(close to the maximum of 1) is analogous to a test rarely every flagging someone as having the flu when they don't. A perfect model will have high sensitivity with a high specificity, meaning that it is able to identify when observations belong to a category and does not incorrectly predict other observations as belonging to that category. The table above indicates that all the categories have relatively high specificity. Class 4 has a specificity of 0.996 which on the surface looks great but because the sensitivity is so low, this high specificity just means that the model rarely every predicts any observations as belonging to category 4. The same follows for category 3. This is also seen in the extremely low detection rate and prevalence for these categories. These results, specifically for categories 3 and 4 are an indication that perhaps just reweighting the classes was not enough to overcome the imbalance in the data.

A better metric for instances such as this where the data is imbalanced is the F1 score. The F1 score is the harmonic mean of the precision and recall. The precision is the percentage of observations that belonged to a category and were correctly predicted as belonging to that category. The recall is just the sensitivity. The F1 score, same as the other metrics, ranges from 0 to 1 with 0 being the worst and 1 being the best. The F1 score is useful because the harmonic mean because it is less affected by extreme values than the arithmetic mean. Categories 4 and 3 are by far the worst with values of 0,104 and 0,119. Category 0 has the best F1 score with a value of 0,716. Then category 1 has a value of 0,383 and category 3 has a score of 0,487. All these values, except for category are quite poor and indicate lack of predictive power. This is reflected in the fact that the model achieved an accuracy of 51,7% on the test set. The model is better than blindly guessing which we expect to return an accuracy of 20% but the model lacks predictive power. Expanding the search are for tuning as well as utalising bootstrapping may yield better results.

Extra resources

`keras`

`kerastuneR`