Predicting Rainfall in Australia Using Neural Networks

Joanna Parker

University of Colorado Boulder, Boulder CO 80303, USA
jopa2386@colorado.edu

1 Libraries

– numpy, pandas for data preprocessing
– matplotlib for visualization
– scikit-learn for data preprocessing, feature selection, model evaluation
– PyTorch for neural network architecture and training

2 Exploratory Data Analysis

The Australian Rain dataset was loaded into a pandas data frame, and rows where the target variable 'RainTomorrow' was missing were dropped. The dataset was split into features and the target variable, and categorical columns were identified. The date column was broken into year, month, and day, and a periodic encoding was used to capture the cyclical nature of weather over time. Missing values were filled with mean values for quantitative columns, and randomly assigned for categorical columns. This introduces noise, but it is largely ignorable by the model and allows for the benefits of a larger training set. Once missing values were handled, categorical variables were given a numeric encoding. Several new features, as combinations of others, were added and the data was split into training and testing sets.

3 Feature Analysis

Lasso Regression was used with 5-fold cross-validation to select the most important features – those with a score $>= 0.01$. This typically allows the model to focus and prevents overfitting, but may have been too tight a bound for this scenario. A correlation matrix of selected features was visualized using Matplotlib.

4 Model Building

A fully connected neural network with two hidden layers using the hyperbolic tangent activation function, and a sigmoid activation function for binary classification was built using PyTorch. It was trained using binary cross-entropy loss and the Adam optimizer. Initial predictions were evaluated on the test set. Hyperparameter tuning was performed using 5-fold cross-entropy validation, to test different values for the number of nodes in each hidden layer and the learning rate. Once optimal parameters were found, an optimal network was trained and its performance was evaluated on the test set.

5 Model Evaluation

The performance of the model was assessed using accuracy, precision, recall, and f1 metrics. The confusion matrix was computed to shed light on the bias toward negative predictions.

6 Results

To break down the results of the model: Accuracy is the ratio of correct predictions to total predictions. This model correctly classified 85.06% of all instances, which is fairly good for a simple neural network and noisy data. Precision is the ratio of correct positive predictions (the model thinks it will rain tomorrow and it does) to total positive predictions (the model thinks it will rain tomorrow and it doesn't). For this model, 72.19% of the instances predicted positive were actually positive, Recall is the ratio of correct positive predictions to total predictions. This model correctly predicted 53.67% of the actual positives as positive, which is interestingly much worse than the precision. It suggests that the model is only slightly better than random guessing at positive instances, which makes sense considering that there are many more days where it does *not* rain tomorrow than days where it does. F1 Score is the weighted average of precision and recall, good for uneven class distribution as we have in this dataset. It is not surprisingly worse than the total accuracy at 0.6156 and raises the concern that the limitation in the model's overall performance is likely due to its inability to consistently classify positive instances.

The confusion matrix describes the model's predictions versus the actual class labels. There were 20787 true negatives and 3403 true positives. There were 1311 false positives: instances where the model thinks it will rain tomorrow and it doesn't. There were 1938 false negatives: the model thinks it won't rain tomorrow and it does.

In the case of this problem, false positive mitigation is slightly more important from a firefighting point of view – if the model predicts wetter weather, the fire danger will be lowered too much. However, if the model consistently predicts drier weather (false negatives), unnecessary and expensive precautions may be taken in an area where they are not needed. 85% accuracy is good from a machine learning point of view, but much more accurate weather prediction is necessary as a resource to governments trying to allocate scarce resources as effectively as possible.

7 Challenges

The cross validation of hyperparameters took by far the most amount of time to run. Originally, I wanted to include batch size as a hyperparameter but since I also had two hidden layers and learning rate, and wanted to test 3 values for each, it needed to run through 3*3*3*3 = 81 combinations. Each combination took approximately 7 minutes to run on my machine, so it would have taken about 9 hours to run through every combination. Without batch size, each fold took about 8 seconds to run, which was much more reasonable.

Batch size is an incredibly important parameter, representing the number of training samples used in one iteration. Smaller batch sizes use less memory but introduce more randomness and can help in avoiding local minima. Larger batch sizes can lead to faster convergence – the model's parameters aren't updated as frequently, so each update is more stable. If I figured out how to better utilize my computer's resources, I could take advantage of parallelization with CUDA and potentially significantly decrease the runtime for a more robust hyperparameter tuning.

8 Potential Improvements

The model seemed to perform at around 85% validation accuracy, with very slight improvements from hyperparameter tuning. This model architecture seems to have reached its ceiling of performance, and I am suspicious that the biggest improvement in accuracy would come not from a more complex version of this model but from better data preprocessing.

One improvement to this project would be in handling the missing values in the dataset with more finesse. One option is an approximation of the distribution of each feature and filling in values randomly about the mean of the distribution, assuming that these features follow a Gaussian distribution (once collapsed because they are time-series data). A second, more complex option would be to estimate the missing values of parameters based on the values of other parameters using a simple neural network. Starting with the parameters with the tightest distributions and moving towards the features with the most variance, this method would hopefully provide the final model with useful information rather than random noise it has to ignore.

Since the classes are unbalanced, leading to bad recall and F1 scores, the model would also benefit from training on a random subsample of the negative class that is more proportional to the positive class. This may slightly reduce overall accuracy, but the improvement in positive detections would be significant. It would be interesting to see if a more complex architecture could handle this class imbalance, or if it would have similar problems and also need to be trained on subsamples of the negative class to improve recall.

It would be interesting to do a deeper analysis of the optimal number of hidden layers, perhaps in a cross-validation loop of its own. Since the data has very complex underlying relationships, intuition suggests that more than two hidden layers may improve performance but it may not be by a significant amount. In addition, different sets of hyperparameters could be considered for cross-validation – I tried standard estimates but it could be that considering hidden sizes as functions of the size of the whole dataset would improve performance. It may also improve the precision and recall of the final model if instead of using the average accuracy to determine the best set of parameters, the f1 score was used.