Comparing classification algorithms for predicting forest cover types from cartographic variables

Virginia Frey and Patrick McCauley
School of Physics, The University of Sydney, NSW 2006 Australia

(Dated: June 4, 2018)

We present a comparison of classifier performance using the covertype data set [1, 2] to predict the predominant forest cover type of approximately 580,000 30×30 m patches inside the U.S. Roosevelt National Forest using on cartographic variables. Three classifiers, representing three very different algorithm types, were explored: Naive Bayes, Multi-Layer Perceptron (MLP), and Random Forest. We review the theory for each classifier and include a detailed evaluation of their associated hyperparameters. We also examine the relative feature responsibilities in the dataset to design preliminary data pre-processing that improves both classification performance and runtime. This includes feature compression, engineering, and regularisation. Our results demonstrate that the Random Forest algorithm performs best with an overall accuracy of 98 %, followed closely by the MLP at 96 %, while the Naive Bayes achieved merely 52 %. The Random Forest further outperformed the MLP in runtime by a factor of six. These results are consistent with previous studies using this dataset that also favour tree-based approaches [3–5].

I. INTRODUCTION

Accurately assessing natural resource information is essential for land monitoring and management purposes. The required resource data is routinely collected through either land-based personnel or remote sensing data, for example using satellite imagery [6]. However, the feasibility of these methods is often limited due to high costs and time consumption, which motivates the exploration of predictive estimation techniques using data that is more readily available, such as cartographic and other types of geographical information for the area of interest [2, 7]. Predictive models based on these resources may also assist in evaluating the past and future development of economical systems, and they may further help to identify human influences and predict natural hazards [7].

In this study, we examine the classification of forest cover types, meaning the predominant tree variety in a given area, based on purely cartographic data. Blackard et al. [2] assembled an exhaustive set of data from the US Geological Survey and the US Forest Service for this purpose in 1999, and they published a study comparing classification accuracies obtained using a discriminant analysis and an artificial neural network. Their dataset has since been made publicly available at standard machine learning databases like the UCI Machine Learning repository [1].

We use Blackard's original dataset evaluate classification performance using three popular algorithms, namely a Naive Bayesian model, a neural network, and a decision tree ensemble. We explore the relevant hyperparameters and discuss the advantages and disadvantages of each algorithm for this particular task. The supporting numeric analysis was carried out using the Python programming language and the open-source machine learning library `scikit-learn` [8].

A. Previous Work

J. Blackard et al. [2], who created and published the dataset in 1998, performed comparative classification studies using both a linear discriminant analysis and an artificial neural network with backpropagation. The former method yielded a classification accuracy of 58 %, while their neural net, consisting of a single hidden layer, yielded 70 %. Their work was motivated by recent successes in applying neural nets in natural resource modelling, and their dataset has since been used in many more classification studies. Interestingly, subsequent publications using this dataset have concentrated on tree-based methods rather than neural nets, which were found yield higher accuracies with shorter run times [3, 4].

Rank	Team name	Accuracy score
1	antgleb	1.00000
2	Ashish Singh	0.99998
3	ucbw207_2_forest	0.99751
	. . .	
11	Audere Labs	0.90391
	. . .	
176	London Machine Learning Learners	0.80021

TABLE I. Accuracy scores on the Kaggle leadership board for the forest covertype competition [5]. Note that this competition is still ongoing, so the ranks are subject to change. The competition used a subset of the full dataset that is used in this work.

Furthermore, a subset of this dataset is the subject of a current Kaggle competition [5], where related work may be found. The classification scores at the time this work was prepared are listed in Table I. Interestingly enough, the high-scoring teams that have made their analysis publicly available also seem to favour tree-based approaches for this dataset.

B. The covertype dataset

The studied dataset contains cartographic data from four distinct wilderness areas in Colorado's Roosevelt National Forest, along with the actual cover type for each region. The cover type is given for 30 × 30 m patches and was collected by the U.S. Forest Service through evaluation of large-scale areal photographs. The data was collected from areas that have seen very little human influence in the past, and their development may therefore be attributed to mostly natural causes.

Property	Value
Observations	581012
Attributes	54
Classes	7
Missing values	0
Baseline accuracy	48.8 %

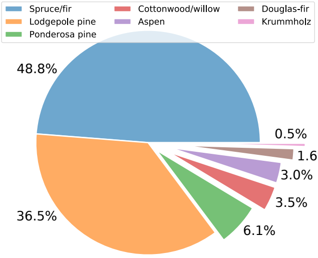


FIG. 1. The covertype dataset at a glance [1]. The table summarises the basic properties of the dataset, and the chart on the right visualises the class distributions. The baseline accuracy was estimated based on the predominant class.

An overview of the dataset is given in Figure 1. The dataset consists of approximately 580,000 observations with 54 attributes per observation and 7 associated classes. There are no missing values in the dataset and very few outliers. The attributes are summarised in Table II.

The seven forest cover types are unevenly represented in the dataset, with the Spruce/fir cover type comprising 48.8 % of the observations, followed closely by the Lodgepole pine at 36.5 %. The remaining five classes constitute 14.7 % of the dataset. This class distribution results in a fairly high baseline classification accuracy of 48.8 % if a classifier were to always predict the most common class. This also implies that special attention must be paid to the prediction of the less common classes when evaluating classifier performance.

II. METHODS

A. Data preparation and pre-processing

The dataset in its original form requires very little pre-processing as there are no missing values, all numeric values lie far away from machine precision limits, and the dimensionality is computationally feasible. However, we have experimented with several pre-processing steps to improve classification accuracy and runtime. This includes compressing the binary columns to reduce dimensionality, examining the responsibility of individual attributes to motivate feature engineering, and lastly feature regularisation.

Attribute	Description
1	Elevation (m)
2	Aspect angle (azimuth from true north)
3	Slope (degrees)
4	Horizontal distance to nearest water surface (m)
5	Vertical distance to nearest water surface (m)
6	Horizontal distance to nearest road (m)
7	Incident sunlight at 9 am
8	Incident sunlight at 12 pm
9	Incident sunlight at 3 pm
10	Horizontal distance to nearest wildfire ignition point
11-14	Wilderness area designation (binary columns)
15-54	Soil type designation (binary columns)

TABLE II. The original attributes of the dataset. The incident sunlight is a relative index measure (with values from zero to 255) of sunlight illumination on the day of the summer solstice, and the binary columns for both areas and soil types are indicator columns with four and 40 binary values respectively.

The binary columns for both wilderness areas and soil type are unique for each observation, meaning that there is only one area and soil type for a single observation. Therefore, we have compressed those columns into two new columns that hold indicator values of 0-4 for the wilderness areas and 0-40 for the soil types.

In terms of feature responsibility, we identified those features that are likely to be most relevant in the classification process by looking at how they *separate* classes. To this end, we binned the individual feature columns and counted the number of occurrences for each class, as shown in Figure 2 for the elevation and wilderness area feature. Features exhibiting large variation in class frequency will have a larger impact on the decision process during classification, whereas features for which these histograms are evenly distributed among all classes are likely to have less influence. Elevation is an example of the former and is likely to play the most important role in deciding which class an observation belongs to. On the other hand, features like the incident sunlight and distances to the closest water surfaces appear less likely to be influential. In the following sections, we show that classifiers that assign feature importances, like decision-tree algorithms, yield results that are consistent with these inferences. A full explanation of this process, along with histograms for each feature, is given in the IPython notebook that accompanies this report.

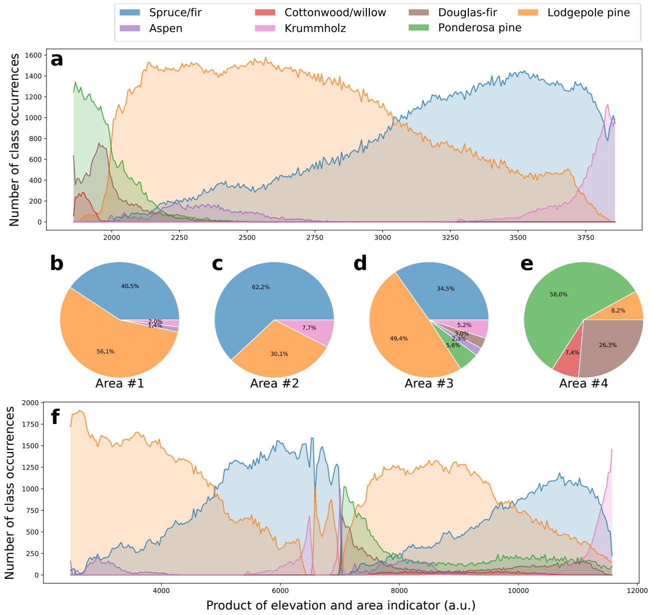


FIG. 2. Evaluating feature responsibility and feature engineering. **a** shows the number of class occurrences for different levels of elevations. The charts in **b-e** show the same quantity for wilderness areas. Both features show a clear separation of classes and are likely to be relevant in the classification process. **f** shows the class counts for a combination of these two features. This new, *engineered* feature provides the combined information and is likely to improve classification accuracy.

Once feature responsibilities are identified, it is possible to create new features by linear or non-linear combinations thereof to achieve better class separation. An example of a new feature is shown in Figure 2f, where we have formed the product of elevation and wilderness area indicator to craft a new feature with the combined information.

That said, we have consequently dropped the features labelled 11-40 and replaced them with the following compressed and engineered features:

11. Wilderness area indicator number (0-4)
12. Soil type indicator number (0-4)
13. Sum of elevation and road distance
14. Difference between elevation and road distance
15. Sum of elevation and distance to wildfire ignition point
16. Difference between elevation and distance to wildfire ignition point
17. Product of elevation and soil type indicator
18. Product of elevation and wilderness area indicator
19. Product of soil type and wilderness area indicators

Overall, the new set of features improve performance for both the Naive Bayes and Random Forest classifiers by several percent, which is consistent with the feature importances ratings that will be presented in Section III B. However, the feature engineering and compression pre-processing slightly diminished the Multi-Layer Perceptron (MLP) classifier's performance and was therefore not used in the final MLP results.

The MLP algorithm further requires that the data be regularised, which is not required for the other two techniques. This step is analogous to the normalisation used for k-Nearest Neighbour algorithms and is likewise due to sensitivity to feature scaling, meaning that biases are introduced if different features have very different ranges. The data may be regularised in a few different ways, and we have adopted the most common approach of scaling each feature to have zero mean with unit variance.

B. Naive Bayes

The Naive Bayes classifier implements Bayes' rule under the "naive" assumption that the classification parameters are independent for a given class. Bayes' rule is given by:

$$P(A|B_1,...B_n) = \frac{P(B_1,...B_n|A)P(A)}{P(B_1,...B_n)} \tag{1}$$

where A is a particular class and B_n are the features that characterise the data. The terms in Equation 1 are generally referred to as:

$$\text{Posterior} = \frac{\text{Likelihood} \cdot \text{Prior}}{\text{Marginal Probability}} \tag{2}$$

The goal is to evaluate Bayes' rule for each class and determine which has the largest posterior for a given sample. However, it is generally not possible to estimate the likelihood $P(B_1,...B_n|A)$ directly because the interdependencies of B_n may either be unknown or prohibitively costly to compute. The Naive Bayes classifier is so-named because it ignores these interdependences, allowing Equation 1 to become:

$$P(A|B_1,...B_n) \propto P(A)\prod_{n=1}^{k} P(B_n|A) \tag{3}$$

where k is the number of features. Note that the marginal probability has been dropped in Equation 3 because it is not needed for simply finding the class with the largest posterior.

The probability of a given class $P(A)$ can be estimated easily by dividing the number of instances of that class by the total number of points in the training data. Our features B_n comprise a mixture of continuous and discrete variables. To estimate $P(B_n|A)$, we can either discretise the continuous variables by separating them into some number of discrete bins, or we can model each parameter using a standard probability distribution. We have chosen to do the latter using Gaussian distributions:

$$P(B_n|A) = \frac{1}{\sqrt{2\pi\sigma_n^2}} \exp\left(-\frac{(x_n-\mu_n)^2}{2\sigma_n^2}\right) \tag{4}$$

where x_n is the value of feature n for a given test sample, σ_n^2 is the standard deviation of feature n for class A in the training data, and μ_n is the mean of feature n for class A in the training data.

Incorporating Equation 4 into Equation 3 directly may produce underflow errors due to the multiplicative sum of many small numbers. Most Naive Bayes implementations avoid this by instead using log probabilities, which allows the multiplicative sum to be cast as an additive sum:

$$\ln P(A|B_1,...B_n) \propto \ln P(A) - \sum_{n=1}^{k} \ln\sqrt{2\pi\sigma_n^2} + \frac{(x_n-\mu_n)^2}{2\sigma_n^2} \tag{5}$$

For a given test vector $(B_1,...B_n)$, Equation 5 is evaluated for each class A, and the assignment goes to the class with the largest value.

C. Multi-Layer Perceptron (MLP)

The Multi-Layer Perceptron (MLP) classifier is a basic neural network that, as its name implies, is composed of a number of individual perceptrons. Perceptrons are binary linear classification algorithms that were first introduced by Ronsenblatt (1958) [9]. The basic perceptron function is given by:

$$f(x) = \begin{cases} 1 & \text{if } \mathbf{w}\cdot\mathbf{x}+b \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

where $\mathbf{x}$ is a features (data) vector, $\mathbf{w}$ is a vector of corresponding weights, and the "bias" b moves the decision boundary.

The perceptron algorithm works by iteratively modifying the weights vector $\mathbf{w}$ to minimise classification error. A given data sample $\mathbf{x}$ is passed through Equation 6, and if the classification is incorrect, $\mathbf{w}$ is shifted in the direction of $\mathbf{x}$ by an amount defined by the *learning rate* η. This procedure is repeated until the solution converges or a stop criteria is met. Convergence can only be achieved if the data are linearly separable. Otherwise, as is generally the case, the algorithm will oscillate between solutions and is typically terminated after a certain number of iterations or when the classification error stops decreasing.

While a single perceptron is only appropriate for binary linear classification tasks, multiple perceptrons can be linked together to tackle non-linear and multi-class problems. Figure 3 visualises the MLP algorithm with one "hidden layer" composed of perceptrons a_k. The MLP algorithm behaves in much the same way as the single perceptron. Each node is connected to the input layer in the same manner as above, such that its output is given by:

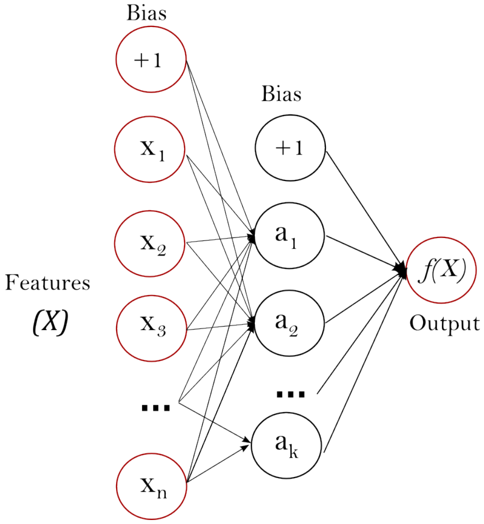


FIG. 3. MLP with one hidden layer [8]. The left side is referred to as the input layer, where the vector $\mathbf{x}$ is composed of the various features in the data. $\mathbf{x}$ is passed with a weights vector $\mathbf{w}_k$ to each node a_k of the middle, "hidden" layer, where each node is an individual perceptron. The output from the perceptrons is used to define the output layer $f(\mathbf{x})$ that assigns the class label.

$$a_k(\mathbf{x}) = \mathbf{w}_k \cdot a(\mathbf{x}) + b \tag{7}$$

where $\mathbf{w}_k$ are the weights associated with perceptron a_k, and $a(\mathbf{x})$ is referred to as the *activation function*. Here we experiment with the four activation functions shown in Figure 5c. The outputs from each perceptron can either be fed into the final output layer, as shown in Figure 3, or they can be passed to an adjacent hidden layer with a new set of weights.

As with the single perceptron, the MLP algorithm works by iteratively modifying the weights vectors $\mathbf{w}_k$ to minimise a *loss function* that characterises the classification error. One of the most common loss functions, and the one our classifier uses, is the *log-loss* function:

$$L_{log}(Y,P) = -\ln Pr(Y|P) = -\frac{1}{N}\sum_{i=0}^{N-1}\sum_{k=0}^{K-1} y_{i,k}\ln p_{i,k} \tag{8}$$

where N is the number of data samples, and k is the number class labels. $p_{i,k}$ is the probability assigned to sample i for class k, and $y_{i,k}$ is a binary value that equals zero if the class assignment for sample i, k is correct. This latter term means that the log loss function equals zero if all samples are classified correctly.

Minimising Equation 8 begins by randomly initialising the weights for each node. All the data samples are then passed through the network, new weights are derived that move the loss function closer to zero, and the process is repeated until the solution converges or a stop criteria is met. This procedure is referred to as *backpropagation* and can be accomplished using several related algorithms that estimate the loss function gradient from the partial derivatives of Equation 8 with respect to each of the weights. The gradient determines the direction in which to move the weights after each iteration, and a learning rate determines how far to move the weights in that direction.

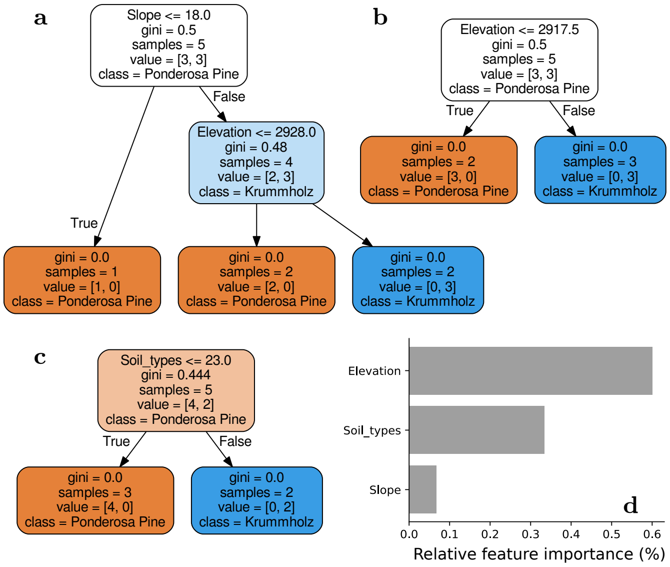
The MLP implementation we use here has three options for the algorithm that optimises the weights: Stochastic Gradient Descent (SGD), Adam [10], and Limited-memory Broyden-Fletcher-Goldfarb-Shanno (LBFGS). LBFGS is best suited to smaller datasets as it typically requires more iterations over the dataset to converge. SGD improves runtime by computing the gradient using only a subset of the training data, referred to as a *mini-batch*. To avoid biases that might be introduced by the ordering of the data, the training data are typically randomly shuffled after each iteration before choosing the mini-batch.

Learning rates for SGD solvers typically remain constant or may be updated in a simplistic way, such as by inverse scaling after each iteration or by decreasing the rate once the classification errors stop decreasing. Adaptive algorithms such as Adam, AdaGrad, and RMSProp attempt to improve SGD by automatically modulating the learning rate throughout the run. Although these methods have become very popular, recent results have cast doubt on how well they generalise as compared to standard SGD [11, 12]. In light of these results, we have chosen to use SGD for the hyperparameter tuning in Section III A.

D. Random Forests

Random forests are an ensemble learning technique based on multiple decision trees [13, 14]. Decision trees are incredibly powerful algorithms for data mining, as they can learn any given data structure arbitrarily well, though this fact also results in relatively poor predictive capabilities [15]. The combination of multiple individual trees, however, can mitigate this overfitting behaviour and offer better predictive performance than could be obtained from any of the individual trees.

During the training phase of a random forest, multiple decision trees are fitted on sub-samples of the training data. This procedure is called bootstrap-aggregating, or bagging [16], and is a form of model-averaging. For a set of B models trained on different sub-samples of the dataset (drawn with replacement), the individual probabilities P_b for a given observation x belonging to a certain class C, form the bagged prediction $\hat{P}_{\text{bag}}$ via:



e

	Ponderosa pine			Krummholz			Importance
Elevation (m)	2506	2447	2520	3552	3350	3329	0.60
Slope (°)	21	39	16	21	20	5	0.33
Soil type	3	9	3	39	38	37	0.07

FIG. 4. Example random forest using a small subset of the covertype dataset. The forest consists of three trees (labelled **a**, **b** and **c**) and the data that was used to build these trees (table **e**) contains six observations from two classes and three features. All trees use a different subset of features and data. Each split decision is made based on the minimum Gini impurity. The nodes are coloured based on which class the data point is more likely to belong to (orange for Ponderosa Pine and blue for Krummholz), and their opacity is inversely proportional to the impurity of the split. By adding up the impurity decrease for each feature, the Gini importance of each feature can be extracted (Table **e** and plotted in **d**).

$$P_{\text{bag}}(C|x) = \frac{1}{B}\sum_{b}^{B} P_b(C|x),$$

which is simply the mean over the individual predictions. Alternatively, many implementations including the one in this work consider the argmax function with respect to each model b. Random forests combine this bootstrap-aggregation with additional *feature bagging*, also known as the random subspace method [17]. Here, the individual models are constructed by systematically selecting random subsets of input feature vector.

For each random subsample of features and training data, a decision tree is grown. In this work, trees were built using the CART algorithm [18]. For a given input of feature vectors and labels, a decision tree recursively partitions the space such that samples with the same label are grouped together. The recursion ends either when all the data has been partitioned or when an early-stopping criterion such a maximum tree depth is reached. The split decision at each node in the tree is based on the *impurity* that the split causes, which is a measure of misclassification entropy. In this work, we consider the Gini impurity, a measure of the likelihood of an incorrect classification that is defined for a given node N as follows:

$$G_b(N,x) = \sum_i P(C_i|x) * (1 - P(C_i|x))$$

where the sum is taken over all possible outcomes. The Gini impurity is lower bounded by zero, which is the value it will assume if all the data x belong to a single class. Based on the Gini impurities calculated at each node, we can extract information about how relevant a given feature was in the splitting process by adding up the Gini impurity decreases for that feature over all the trees in the forest. This is also called Gini importance [13].

A sample random forest using a small fraction of the covertype dataset is shown in Figure 4. The forest consists of three trees, and the data contains only two classes (Krummholz and Ponderosa Pine), with three features for each point. The decision paths were visualised using the `graphviz` package [19] in combination with `scikit-learn`. At each node, the split is based on the minimum Gini impurity. The trees stop growing when a Gini impurity of zero is reached. Based on the three trees in this example, the most relevant features can be extracted.

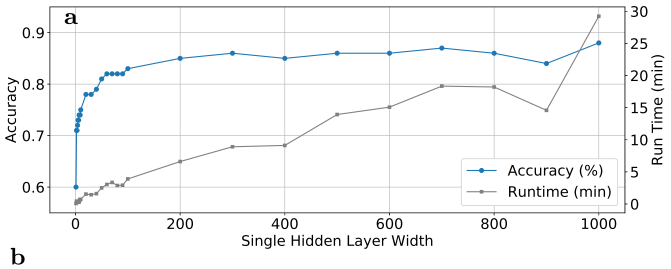
III. EXPERIMENT

A. MLP Parameter Tuning

The MLP implementation in `scikit-learn` includes many hyperparameters that can be tuned by the user to influence classification and runtime performance. There are three options for the algorithm that optimises the weights, and as we described in Section II C, we have chosen to use Stochastic Gradient Descent (SGD) for our analysis.

Table III lists all of the relevant hyperparameters, their default values, the ranges over which we tested them, and their "optimum" values given the test results. Each hyperparameter was varied over the indicated range while keeping the others at their default values. The optimum value refers to the one that maximised overall accuracy. If two test values achieved the same accuracy to within 1%, the value with the smaller runtime was chosen.

"Early stopping" was enabled for all of our results to improve runtime at minimal cost to accuracy. This works by internally reserving 10% of the training data as a test set so that training can be terminated when the test errors stop decreasing by a tolerance margin for two consecutive epochs. This splitting is internal to the algorithm and is in addition to the main training and test set partition used to score the results. "Warm starting", which can improve runtime by incorporating information from a previous run was also disabled for all of our results, as this would obviously bias our comparative runtime results.



b

N Layers (Depth)	Accuracy	Run Time (s)	Iterations
1x [100]	0.83	181	60
2x [100]	0.84	207	37
3x [100]	0.86	253	31
4x [100]	0.86	264	25
5x [100]	0.81	125	10

c

Activation function	$a(\mathbf{x}) =$	Accuracy	Runtime (s)
Identity	x	0.72	20
Logistic (Sigmoid)	$(1 + e^{-\mathbf{x}})^{-1}$	0.78	390
Hyperbolic Tangent	$\tanh(\mathbf{x})$	0.84	285
Rectified Linear Unit	$\max(0, \mathbf{x})$	0.83	205

FIG. 5. Multilayer Perceptron parameter tuning. The plot in panel **a** shows the accuracy and runtime versus the number of nodes (width) in a single hidden layer. Table **b** compares different hidden layer depths for networks composed of 1–5 layers of 100 nodes each, and Table **c** demonstrates the effect that different activation functions have on classification accuracy and runtime.

What follows is a brief description of the most impactful parameters.

- Hidden layer with: The number of hidden layers and the number of nodes in each layer is perhaps the most important single hyperparameter to consider. Figure 5a shows the accuracy and runtime results considering a single hidden layer with widths ranging from 1 to 1000 nodes. The highest accuracy was achieved using the largest width, requiring a runtime of around 30 minutes. However, above a few hundred nodes, the increases in accuracy come at a large runtime cost. For instance, 300 nodes performed only 2% worse in accuracy with a factor of ∼4 smaller runtime compared to 1000 nodes.

- Hidden layer depth: Figure 5b shows results for different numbers of hidden layers, each with 100 nodes. Unlike the hidden layer width, the accuracy improvements saturate at 3 hidden layers. The runtime results here also suggest that increasing the depth of a neural network is the more computationally efficient way to improve accuracy. Three 100-node layers perform similarly to one 300-node layer with around half the runtime.

- Activation function: Figure 5c shows the result of using each of the available activation functions. The hyperbolic tangent (tanh) and Rectified Linear Unit (ReLU) functions performed similarly, with tanh producing a slightly better accuracy somewhat more slowly. This is consistent with expectations based on previous work that shows ReLU generally achieves similar (if somewhat worse) results with significantly fewer iterations compared to tanh [20].

While the hidden layer configuration and activation function choices were perhaps most important, the parameters related to the SGD solver also had significant effects on both runtime and accuracy. In particular, the learning rate, L2 penalty term, and mini-batch sizes are all very influential. The full results for each hyperparameter are given in the IPython notebook that accompanies this report.

This analysis constitutes an incomplete grid search that informed, but did not completely dictate, our final hyperparameter selection. We achieved a final accuracy of 96%, which is 13% better than the default settings at a 3.6× cost in additional runtime. We will discuss limitations to our tuning approach in Section IV.

B. Random Forest Parameter Tuning

The hyperparameters of the random forest algorithm are a combination of those of the individual decision trees and those used for determining the random sampling routine. Optimal values for each parameter were determined using the same approach as outlined in the previous section for the MLP tuning. The results are summarised in Table IV. Below, we again give an overview over those hyperparameters that have the arguably biggest impact on the classification accuracy, while full results and descriptions can be found in the IPython notebook.

The number of trees in the Random Forest has the biggest impact on classification accuracy. In general, the accuracy is expected to increase with the number of trees, and Breiman has shown that in the limit of $B \longrightarrow \infty$, the misclassification error converges to a minimal value such that adding more trees will not increase overfitting [13]. We can reproduce this behaviour by observing that the classification accuracy saturates after about 40 trees, indicating that the misclassification error has converged. This is visualised in Figure 6a.

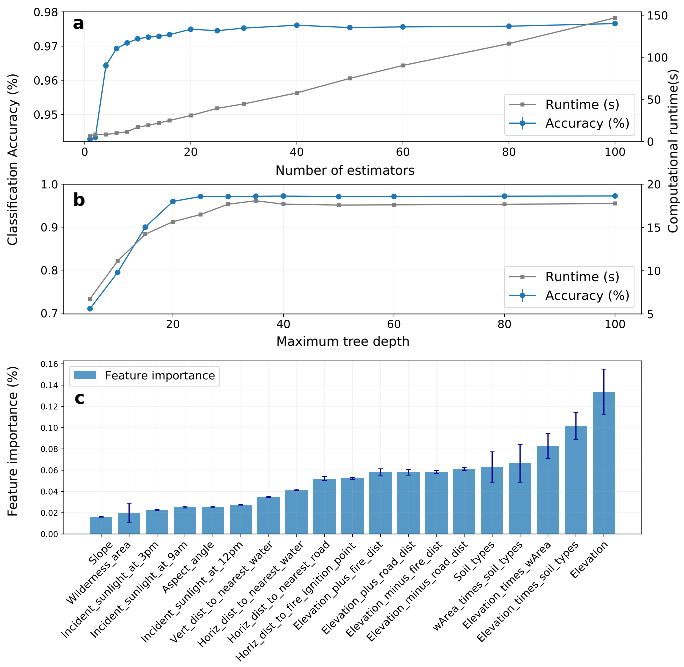


FIG. 6. Random forest parameter tuning and feature relevance. Panels **a** and **b** show exemplary tuning curves for the hyperparameter `n_estimators` (number of decision trees) and `max_depth` (maximum tree depth). Accuracy and runtime are plotted, averaged over ten folds in cross-validation. Panel **c** shows the ten-fold averaged feature importance as rated by the classifier. The error bars represent standard deviations from those averages.

Most hyperparameters for the individual decision trees set bounds on when to stop the tree growth, by for example defining a maximum tree depth or allowing nodes to become leaves once the impurity has reached a lower threshold (other than zero). Therefore, in our experiments with those parameters we predominantly observe a clearly defined upper bound of the achievable accuracy. For instance, increasing the maximum tree depth initially increases accuracy until it reaches an upper threshold. Interestingly enough, we do not observe an accuracy decrease that might be expected from overfitting, indicating that the trees stop growing naturally after about 30 successive nodes – an observation that is confirmed by the saturation in runtime as well. Through exploration of the remaining continuous hyperparameter we find that minimal restrictions on tree growth yield the highest accuracy scores. All values are listed in Table IV.

Additionally, disabling the default bootstrapping routine to bag the training samples improves the accuracy by about 0.5 % on average, which we believe can be explained by an unfavourable representation of the less common Cottonwood-willow class in the training samples, and because it hardly distinguishes itself from other classes through relevant features. Indeed, our tests have shown lower precision and recall values for this class under bootstrapping.

Lastly, during the tuning process we also examined the feature importances rated by this classifier. As ex-

Parameter	Default Value	Test Range	Optimum Value
Single Hidden Layer (Width)	[100]	[1] – [1000]	1000
Multiple Hidden Layers (Depth)	[100]	[100] – [100,...,x5]	[100, 100, 100]
Activation Function	ReLU	See Figure 5c	tanh
Alpha (L2 penalty)	0.0001	[1e-6, 0.5]	1e-5
Mini Batch Size	200	[10, 1e5]	100
Learning Rate	Constant	[Constant, Inverse Scaling, Adaptive]	Adaptive
Initial Learning Rate	0.001	[1e-5, 1]	0.1
Inverse Scaling Exponent	0.5	[0.1, 2]	1.4
Shuffle	True	[True, False]	True
Tolerance	1e-4	[1e-6, 10]	5e-5
Momentum	0.9	[0.01, 0.99]	0.99
Nesterov's Momentum	True	[True, False]	True

TABLE III. Tunable MLP hyperparameters. Each was tested over the prescribed range with all others set to the default. "Optimum" is defined as the test value with highest average of accuracy and F1 score.

Parameter	Default Value	Test Range	Optimum Value
Number of estimators	[10]	[1] – [100]	60
Impurity criterion	Gini	Gini and Entropy	Gini
Maximum features per split	$\sqrt{n_{\text{features}}}$	[1] – [n_{features}]	5
Maximum tree depth	Unlimited	[2] – [100]	30
Minimum samples per split	2	[2] –[10]	2
Minimum impurity decrease	0	[0] – [0.001]	0
Minimum samples per leaf	1	[1] – [10]	1
Use bootstrap samples	True	[True, False]	False
Use out-of-bag samples	False	[True, False]	False

TABLE IV. Tunable Random Forest hyperparameters. Each was tested over the prescribed range with all others kept fixed. "Optimum" is defined as the test value with highest average of accuracy and F1 score, and minimum runtime.

pected from our preliminary data exploration, the elevation seems to indeed be the most relevant parameter in split decisions, followed closely by our new, engineered features. This observation is consistent with the increase in accuracy that we observed for the Naive Bayes and the Random Forest classifiers after adding the engineered features.

C. Classifier Comparison

The three classifiers compared in this work all yield classification accuracies higher than the baseline accuracy (based on predicting the most common class) of the dataset. A detailed comparison of class-specific precision, recall and F1 scores is shown in Table V, and the final overall results are shown in Table VI.

The performance of the Naive Bayes classifier is by far the worst with only 58 % overall accuracy, which is achieved mainly due to correct predictions of the most common classes in the dataset (see Table V). All other classes are frequently misclassified by Naive Bayes. The behaviour of this classifier can most likely be explained by the strong correlations of individual features in the dataset, which violates the naive feature independence assumption, and the non-Gaussian feature distributions.

The MLP and Random Forest classifier perform substantially better, with overall accuracies of 96 % and 98 %, respectively. They classify the least common classes with a worst-case precision of 87 %. However, in terms of computational runtime and memory usage, the matrix multiplication performed by the Naive Bayes classifier is far superior. For the more complex algorithms, we find the Random Forest to outperform the MLP in terms of runtime by a factor of six (see Table VI). This is partly because the Random Forest classifier can easily be parallelised because the different trees can grow independently of each other, and we have taken advantage of the parallelisation implemented in `scikit-learn` that utilises a fast Cython implementation [8] to work

	Naive Bayes			MLP			Random Forest		
Class	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall
Spruce/Fir	0.50	0.74	0.38	0.96	0.96	0.96	0.98	0.98	0.97
Lodgepole Pine	0.59	0.58	0.60	0.96	0.96	0.96	0.98	0.98	0.98
Ponderosa Pine	0.26	0.15	0.95	0.95	0.94	0.95	0.97	0.97	0.98
Cottonwood/Willow	0.16	0.10	0.38	0.87	0.87	0.87	0.91	0.92	0.89
Aspen	0.22	0.17	0.30	0.87	0.88	0.86	0.92	0.94	0.90
Douglas-fir	0.47	0.34	0.79	0.89	0.90	0.89	0.96	0.96	0.95
Krummholz	0.61	0.54	0.69	0.96	0.97	0.95	0.97	0.98	0.97

TABLE V. Extensive classification results for each algorithm. The results include class-specific precisions, recalls and F1 scores averaged over a 10-fold cross validation. Cell colouring marks the maximum and minimum value of each column and visually highlights the fact that the classifiers mostly perform best on the best-represented classes (Spruce/Fir and Lodgepole Pine) and worst on the less well represented classes (Cottonwood/Willow and Aspen). The Krummholz class, even though also achieves high scores because relevant features like elevation and soil type separate this class well from the others.

around Python's global interpreter lock which is notoriously cited for slowing down Python routines. Neural networks can be parallelised in principle, but this has not be implemented in `scikit-learn`. Therefore, although we find the Random Forest algorithm to be significantly more efficient than the MLP, our results may not entirely reflect the true differences in computational efficiency for these types of algorithms in general.

While classification performance of both the MLP and Random Forest are similar, the Random Forest classifier requires significantly less tuning to perform well; the accuracy with all parameters set to their defaults was about 95 % for the Random forest and 83 % for the MLP. Furthermore, the tuning process was also considerably faster for the Random Forest because of shorter runtimes and little interdependence between the parameters. However, we note that we have reached an upper bound for classification accuracy after all parameter ranges have been exploited, suggesting that further tuning will not improve the overall accuracy any further. Parameter tuning for the MLP algorithm on the other hand, and neural networks in general is somewhat less straightforward, and we suspect that higher accuracies could be achieved with better tuning, as discussed in the following section.

	Naive Bayes	MLP	Random Forest
Accuracy	0.52	0.96	0.98
Precision	0.38	0.93	0.96
Recall	0.58	0.92	0.95
F1-Score	0.40	0.92	0.95
Runtime (s)	0.68	636	112

TABLE VI. Final classifier comparison. Each classifier was appropriately tuned and the classification results were averaged over a 10-fold cross validation. The accuracy is an overall average, while the F1-score, precision, and recall are averaged over the individual classes.

IV. DISCUSSION AND FUTURE WORK

Our MLP tuning results from Section III A and the accompanying IPython notebook illustrate one of the main limitations of neural networks in general, which is that they possess a large number of hyperparameters that must be tuned to the specific task. How to choose the "best" value for a given hyperparameter is often not obvious and has traditionally been left to the user. Our approach was to perform a limited grid search where each hyperparameter was varied while keeping the others at their respective defaults. Although we were able to improve our accuracy from the default settings by ∼13% using this method, we could not simply combine the "best" values from Table III to do so.

This is because the different hyperparameters are not independent of each other and therefore had to be retuned somewhat once the optimal values were combined. A more complete grid search could accomplish this automatically, but the run time required to perform that operation would be unreasonably large. This issue has received a lot of attention in recent years, and Bayesian optimisation using Gaussian processes is emerging as perhaps the most promising method for hyperparameter optimisation [21, 22]. These methods work in a manner that is analogous to SGD in that hyperparameter values can be initialised randomly and stochastically varied to eventually converge on an optimal configuration. It is highly likely that the 96% accuracy we achieved with the MLP could be improved in this way, which would be a productive direction for future work.

Our results also highlight the sensitivity of machine learning algorithms to pre-processing. The feature engineering and compression described in Section II A resulted in a modest improvement (∼1%) to the Random Forest results and a significant improvement (∼6%) for Naive Bayes. It was surprising to find then that the same steps actually diminished the MLP accuracy slightly by around 1%. While it is not clear exactly why this was the case, it underscores the point that pre-processing choices must be tailored to the specific algorithm in use. Future work could explore why the feature engineering was not successful for MLP and what pre-processing steps might be used alongside regularisation to improve those results. There are also, of course, many other neural network and decision tree implementations, along with entirely different algorithms, that could be explored in this context.

V. CONCLUSION

We have compared the performance of three distinct classification algorithms on the covertype dataset. Our highest 10-fold cross-validation accuracy (98 %) was achieved using a Random Forest approach, followed by the Multi-Layer Perceptron (96 %) and Naive Bayes (52 %). Our results show that complex classification models like neural networks and decision tree ensembles outperform simple linear models like the Naive Bayes classifier. We also found that the Random Forest approach appears to be better suited to this dataset than the MLP algorithm, as the Random Forest achieved marginally higher accuracies with significantly lower runtimes. While we acknowledge that the MLP hyperparameters can likely be better tuned so as to rival the accuracy of Random Forest, a more finely-tuned MLP implementation is still unlikely to match Random Forest in terms of computational efficiency. We also experimented with different pre-processing techniques, employing feature engineering to successfully improve both the Random Forest and Naive Bayes results, while it slightly decreased the accuracy of the MLP classifier, an observation that shows that pre-processing has to be adapted to different algorithms in order to extract maximum classification accuracy for a given problem set.

Coding and runtime environment

The classification studies in this work were performed using the Python programming language and additional open-source libraries including `scikit-learn` [8], `numpy` [23], `pandas` [24] and `matplotlib` [25]. The report is accompanied by two IPython notebooks which contain the exploratory data analysis and pre-processing, as well as the classification comparisons. The notebooks and all accessory scripts are publicly accessible at the University of Sydney GitHub Enterprise in the classifier_comparison_on_covertype_data repository.

The hardware environment used for the classifier comparison results was a MacBook Pro with an Intel i5 dual-core processor with 2.7 GHz maximum clock rate, and 8 GB memory on a 1867 MHz DDR3 RAM.

Author contributions

V. Frey performed most of the pre-processing work and all of the Random Forest analysis. P. McCauley did the work related to the Naive Bayes and MLP classifiers. Accessory scripts for data processing, cross-validation, and display were written collaboratively. The manuscript was written by both V. Frey and P. McCauley.

[1] Dheeru D, Karra Taniskidou E. UCI Machine Learning Repository; 2017. Available from: `http://archive.ics.uci.edu/ml`.

[2] Blackard JA, Dean DJ. Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables. Computers and Electronics in Agriculture. 1999;24:131–151.

[3] Gama J, Rocha R, Medas P. Accurate Decision Trees for Mining High-speed Data Streams. In: Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. KDD '03. New York, NY, USA: ACM; 2003. p. 523–528. Available from: `http://doi.acm.org/10.1145/956750.956813`.

[4] Oza NC, Russell S. Experimental Comparisons of Online and Batch Versions of Bagging and Boosting. In: Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. KDD '01. New York, NY, USA: ACM; 2001. p. 359–364. Available from: `http://doi.acm.org/10.1145/502512.502565`.

[5] Competitions K. Forest covertype leadership board; May 2018. Available from: `https://www.kaggle.com/c/forest-cover-type-prediction/leaderboard`.

[6] Shao Ge. Forest cover types derived from Landsat thematic mapper imagery for Changbai mountain area of China. Canadian Journal for Forest Research. 1996;26:201–206.

[7] Dou Je. An integrated artificial neural network model for the landslide susceptibility assessment of Osado Island, Japan. Nature Hazards. 2015;.

[8] Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, et al. Scikit-learn: Machine Learning in Python. Journal of Machine Learning Research. 2011;12:2825–2830.

[9] Ronsenblatt F. The perceptron: a probabilistic model for information storage and organization in the brain. Psychological review. 1958;65:386–408.

[10] Kingma DP, Ba J. Adam: A Method for Stochastic Optimization. CoRR. 2014;abs/1412.6980. Available from: `http://arxiv.org/abs/1412.6980`.

[11] Wilson AC, Roelofs R, Stern M, Srebro N, Recht B. The marginal value of adaptive gradient methods in machine learning. In: Advances in Neural Information Processing Systems; 2017. p. 4151–4161.

[12] Keskar NS, Socher R. Improving Generalization Performance by Switching from Adam to SGD. arXiv preprint arXiv:171207628. 2017;.

[13] Breiman L. Random Forests. Machine Learning;45.

[14] Ho TK. Random Decision Forests. Proceedings of the 3rd International Conference on Document Analysis and

Recognition. 1995;p. 278–282.

[15] Hastie T, Tibshirani R, Friedman J. The Elements of Statistical Learning. Springer;.

[16] Breiman L. Bagging predictors. Machine Learning;24.

[17] Ho TK. The Random Subspace Method for Constructing Decision Forests. IEEE Transactions on Pattern Analysis and Machine Intelligence. 1998;20:834–844.

[18] Breiman L, Friedman J, Olshen R, Stone C. Classification and Regression Trees. Wadsworth Belmont;.

[19] Ellson J, Gansner ER, Koutsofios E, North SC, Woodhull G. Graphviz and dynagraph âĂŞ static and dynamic graph drawing tools. In: GRAPH DRAWING SOFTWARE. Springer-Verlag; 2003. p. 127–148.

[20] Krizhevsky A, Sutskever I, Hinton GE. Imagenet classification with deep convolutional neural networks. In: Advances in neural information processing systems; 2012. p. 1097–1105.

[21] Snoek J, Rippel O, Swersky K, Kiros R, Satish N, Sundaram N, et al. Scalable bayesian optimization using deep neural networks. In: International conference on machine learning; 2015. p. 2171–2180.

[22] Domhan T, Springenberg JT, Hutter F. Speeding Up Automatic Hyperparameter Optimization of Deep Neural Networks by Extrapolation of Learning Curves. In: IJCAI. vol. 15; 2015. p. 3460–8.

[23] Oliphant TE. A guide to NumPy. vol. 1. Trelgol Publishing USA; 2006.

[24] McKinney W. Data Structures for Statistical Computing in Python. In: van der Walt S, Millman J, editors. Proceedings of the 9th Python in Science Conference; 2010. p. 51 – 56.

[25] Hunter JD. Matplotlib: A 2D graphics environment. Computing in science & engineering. 2007;9(3):90–95.