Remote Sensing for Forest Recovery: Technical Report

Benjamin Frizzell Zanan Pech Mavis Wong Hui Tang
Piotr Tompalski Alexi Rodríguez-Arelis

2025-06-18

Table of contents

1 Introduction

Monitoring the success of large-scale afforestation initiatives is a critical yet complex challenge especially for early growth stages. Young trees often produce weak spectral signals due to sparse canopies, making them difficult to detect using traditional remote sensing methods. As part of Canada's 2 Billion Trees program—which aims to plant two billion trees across Canadian provinces by 2031—Natural Resources Canada must track survival rates across hundreds of ecologically diverse and often remote planting sites.

This problem is crucial because the ecological and climate benefits of afforestation—such as carbon reduction, and biodiversity restoration can only be realized if newly planted trees survive and grow.

In this study, we aim to investigate two main research questions:

- Can satellite-derived vegetation indices and site-level data be used to accurately predict tree survival over time in large-scale afforestation programs?
- Which modelling approach is most effective, and how long after planting is needed before accurate survival predictions can be made?

To address these qestions, this study leverages satellite-derived vegetation indices and site-level data to train machine learning models. By evaluating multiple modelling approaches—including logistic regression, random forests, and deep learning architectures—we aim to determine which techniques provide the most accurate predictions of tree survival rate to support the mission of supporting sustainable forest management and addressing climate change.

2 Data & Pre-processing

We structure our workflow into two phases—static feature modelling and sequence modelling—each with its own preprocessing steps, followed by evaluation using grouped cross-validation.

2.1 Phase 1: Static Preprocessing & Modelling

Preprocessing Steps - load_data.py: Import RDS (~630 MB) and export Parquet (`afforestation_rasters.parquet`) plus CSVs (`planting_records.csv`, `survival_survey.csv`). - **preprocess_features.py:** Drop invalid sites, one-hot encode `Type`, scale `Density` & `Age`; output `clean_feats_data.parquet`. - **data_split.py:** Binarize Year-7 retention (70 %), stratify by `Type`, and split into `train_data70.0.parquet` and `test_data70.0.parquet`.

Static Models

Model	Implementation	Tuned Hyperparameters
Logistic Regression	`LogisticRegression(class_weight="`	`C` (inverse regularization strength)
Random Forest	`RandomForestClassifier(class_weig`	`n_estimators`, `max_depth`
XGBoost	`XGBClassifier(monotone_constraint`	`max_depth`, `learning_rate`, `n_estimators`, `reg_alpha`, `reg_lambda`

2.2 Phase 2: Sequence Preprocessing & Modelling

Preprocessing Steps - load_data.py: Same raw ingest as Phase 1. - **pivot_data.py:** Mask annual rasters by polygon, compute mean of 12 indices per year, save ~15 000 `site_<ID>.parquet` sequence files. - **data_split.py:** Build lookup table (static features + filename + target) and split identically to Phase 1.

Sequence Models

Model	Implementation	Tuned Hyperparameters
GRU	1-layer GRU (hidden_size=32, dropout=0.2) + static features → dense → logit	`hidden_size`, `dropout`, `learning_rate`
LSTM	1-layer LSTM (hidden_size=32, dropout=0.2) + static features → dense → logit	`hidden_size`, `dropout`, `learning_rate`

2.3 Evaluation Metrics

We use grouped 5-fold CV by site ID (random_state=591) on `train_data70.0.parquet`, optimizing **F1 Score**. Final evaluation on `test_data70.0.parquet` reports:

- **Precision & Recall:** trade-off between false positives and negatives.
- **F1 Score:** harmonic mean of Precision and Recall.
- **AUC:** area under the ROC curve for ranking quality.

3 Modelling Specifications

This phase of modeling began with three classical machine learning models: Logistic Regression (as a baseline), Random Forest, and Gradient Boosting Machines. To better capture the temporal structure of the remote sensing time series, two recurrent neural networks—GRU and LSTM—were subsequently developed. This section provides detailed descriptions of each model's architecture and the rationale for their selection. Methods for training, tuning, and evaluating model performance will also be thoroughly outlined.

3.1 Classial Modelling

3.1.1 Logistic Regression

Logistic regression is a generalized linear model widely used for binary classification tasks, valued for its simplicity and interpretability. It models the **log-odds** of the probability, or the **logit** that a given record belongs to class 1 as a linear combination of the input features (Wikipedia contributors 2025):

$$\log\left(\frac{p}{1-p}\right) = \beta_0 + \beta^\top \mathbf{x}_i, \quad i = 1, \dots, n \tag{1}$$

Here,

- n denotes the sample size,

- $\mathbf{x}_i = [x_{i1}, x_{i2}, \dots, x_{iD}]$ is the D-dimensional feature vector for the ith observation (e.g., site-level features and aggregated vegetation indices),

- p is the probability that the target label y_i corresponds to the high survival class: $p = P(y_i = 1 \mid \mathbf{x}_i)$

The coefficient vector $\beta = [\beta_1, \beta_2, \dots, \beta_D]$ represents the influence of the features on each prediction. The jth entry of β corresponds to the change in the log-odds associated with a one-unit increase in the jth feature, holding all other features constant.

An optimal estimate of β is determined by minimizing the **cross-entropy loss**:

$$\mathcal{L} = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \log(\hat{p}_i) + (1 - y_i)\log(1 - \hat{p}_i)\right], \tag{2}$$

Where $\hat{p}_i$ is the estimated class probability obtained from the inverse of Equation 1, which can be shown to be the **sigmoid function**:

$$\hat{p}_i = \sigma(\beta_0 + \beta^\top \mathbf{x}_i) = \frac{1}{1 + \exp\left(-(\beta_0 + \beta^\top \mathbf{x}_i)\right)} \tag{3}$$

These probabilistic predictions can be converted to binary class labels by applying a specified decision threshold, typically 0.5. Model performance across different thresholds can be evaluated using ROC and precision–recall (PR) curves, which are discussed in Section 3.6.

Overall, logistic regression provides an interpretable, statistically grounded baseline and serves as a proxy for the classical statistical modeling used prior to this analysis. To demonstrate the value of more sophisticated machine learning models in predicting survival rates, any subsequent models should achieve performance that exceeds that of logistic regression.

3.1.2 Tree-Based Modelling

Many high-performing machine learning models are composed of simple, rule-based structures known as decision trees. These models make predictions by recursively partitioning the input dataset into distinct regions based on selected features and threshold values. An example of a decision tree is shown in Figure 1.

Each internal node in the tree represents a decision based on a specific feature and a corresponding threshold, and each leaf node corresponds to a unique subset of the data, defined by the path of decision rules leading to it. In binary classification, the majority label of samples in a leaf node is used as the prediction, but for regression, the mean of the target within the leaf node is given. Feature-threshold pairs are selected using a greedy algorithm: starting from the root node, the tree is grown iteratively by choosing the split that most effectively reduces

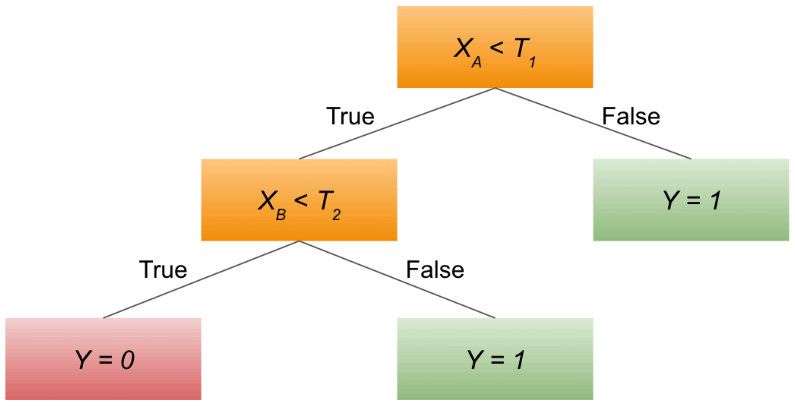

Figure 1: A simple example of a Decision Tree with a depth of 2. The predictions Y are made by comparing selected features X_A and X_B via comparison with threshold values T_1 and T_2.

the value of a given loss function. The cross-entropy loss defined in Equation 2 is commonly used for binary classification tasks; however, Gini impurity is another frequently used criterion (Pedregosa et al. 2011). Alternatively, regression loss functions such as MSE can be used for Regression Tree tasks. Tree construction halts either when a leaf node contains only one class (resulting in zero loss for that subset) or when a predefined stopping criterion, such as the maximum depth, is met. See Section 3.3.2 for guidance on selecting an appropriate maximum tree depth; choosing a higher max depth generally leads to a greater number of decisons and an overall more complex model.

3.1.3 Random Forest Classifier

The Random Forest model is an aggregate model composed of many decision trees, each trained on a bootstrapped subset of the training data and a randomly selected subset of the features. Typically, the maximum allowable depth for each tree in a Random Forest is quite high, resulting in individual trees that are often overfitted and exhibit high variance. However, this high variance is mitigated through aggregation: by combining the predictions of many diverse trees, the overall model can generalize effectively to unseen data. For binary classification tasks, the final prediction is determined by majority vote among the individual trees.

Although training Random Forests can be computationally intensive, each tree is trained independently, enabling efficient parallelization and scalability. Previous studies from Bergmüller and Vanderwel (2022), have demonstrated that Random Forests perform well when using vegetation indices to predict canopy tree mortality. Becuase of this, this model was selected as a candidate for the present analysis.

3.1.4 Gradient Boosting Classifier

The Gradient Boosting model is a popular model that exists in a collection of 'boosting' models, which -unlike Random Forests- consists of a sequence of underfit and biased 'weak learner' models which converge to a high-performing 'strong learner' model when combined (Zhou 2025). This model was selected as a candidate model due to fast implementation and strong performance across a wide variety of machine learning tasks (Chen and Guestrin 2016).

Convergence to a strong learner from a series of weak learners is performed by iteratively fitting a regression tree to the errors of the previous model estimate. To understand this, we firt define the per-sample the loss to be the negative of Equation 2 evaluated for a particular class prediction $\widehat{p}_i$:

$$\ell_i(\widehat{p}_i, y_i) = -\left[y_i \log(\widehat{p}_i) + (1 - y_i)\log(1 - \widehat{p}_i)\right] \tag{4}$$

The model outputs raw logit predictions $f_i(\mathbf{x}_i)$, which can be converted to probabilistic predictions via the sigmoid function shown in Equation 3:

$$\widehat{p}_i = \sigma(f_i(\mathbf{x}_i))$$

The errors associated to each prediction are quantified by the **gradient** g_i and **Hessian** h_i of the loss with respect to the model estimate:

$$g_i = \frac{\partial \ell_i}{\partial f(\mathbf{x}_i)} = \widehat{p}_i - y_i \tag{5}$$

$$h_i = \frac{\partial^2 \ell_i}{\partial f(\mathbf{x}_i)^2} = \widehat{p}_i(1 - \widehat{p}_i) \tag{6}$$

3.1.4.1 Initialization

The model initializes with a constant prediction f_0 across all training sample, usually taken as the logit function (ie. the left-hand side of Equation 1) evaluated over the proportion of samples with label 1:

$$f_0 = \log\left(\frac{P(Y=1)}{1 - P(Y=1)}\right)$$

3.1.4.2 Update step

To update the model prediction after initialization, a regression tree is fitted with the gradients given by Equation 5 as the target predictor. Using Newton's method, the output for a particular leaf node j is given by the sum of g_i and h_i for all samples that reach that leaf node.

$$\omega_j^{(1)} = \frac{\sum_{i\in j} g_i}{\sum_{i\in j} h_i} \tag{7}$$

The overall model prediction is then updated:

$$f_1(\mathbf{x}_i) = f_0 + \eta\omega_{\mathbf{x}_i}^{(1)}$$

Where $\omega_{\mathbf{x}_i}$ denotes the leaf node that sample $\mathbf{x}_i$ is assigned.

Where η is a predefined **learning rate** which controls the degree to which each weak learner can make contributions to the overall model estimate. See Section 3.3.2 for further details.

This update process is repeated iteratively, producing a final estimate of the log-odds which can be converted to a class probability and class labels through the same process as that of the logistic regression model:

$$F(\mathbf{x}_i) = f_0 + \eta\sum_{k=1}^{K}\omega_{\mathbf{x}_i}^{(k)}$$

Where K is the total number of iterations of the algorithm.

3.2 Feature Selection Methods

To address collinearity among vegetation indices and evaluate the importance of both site-based and remote sensing features, we applied three feature selection methods:

3.2.1 Permutation Importance

We estimate each feature's importance by randomly shuffling its values across samples before training, then measuring the resulting change in the model's performance (F_1 score; see Section 3.6). This yields an interpretable, global importance metric. However, when predictors are highly correlated, it can misattribute importance—because different features may serve as proxies for one another—leading to misleading rankings (Pedregosa et al. 2011).

3.2.2 SHAP Values

SHAP (SHapley Additive exPlanations) return per-prediction feature contributions based on Shapley values from cooperative game theory (Lundberg and Lee 2017). Concretely:

1. A baseline expectation is defined (e.g., the average model output when no features are known).

2. For each feature, all possible subsets of features including and excluding that feature are considered, and the marginal contribution is averaged.

3. Taking the mean absolute SHAP values across all samples yields a global importance ranking.

This method provides both local (per-prediction) and global interpretability. However, SHAP may tacitly distribute credit among highly correlated features—sometimes giving a near-zero or inflated value to one feature over another—depending on whether the model uses marginal or conditional expectations when computing the baseline.

3.2.3 Recursive Feature Elimination with Cross-Validation (RFECV)

Finally, RFECV is used to iteratively train the model and remove the least important features based on model-derived importance metrics (e.g., coefficients or feature gains). Each reduced feature subset was evaluated by its F_1 performance using cross-validation (see Section 3.3.3). This method directly handles correlated features by eliminating them if they do not contribute to the model performance, however it can be quite computationally exhaustive. Feature rankings based on how early features were removed are used as importance metrics.

3.3 Training and Tuning Classical Models

Most machine learning models involve a set of hyperparameters—values specified *a priori*—that govern model complexity and influence training behavior. Inappropriate hyperparameter choices can result in models that are either overly biased or unnecessarily complex, leading to poor generalization on unseen data. This section provides a detailed overview of the key hyperparameters for each candidate model in this analysis, along with the methodology used for their selection.

3.3.1 Regularization

In general, regularization involves a penalty to the loss function of that is proportional to the magnitude of the model parameters; stronger regularization leads to smaller parameters and more conservative predictions, which often aids in decreasing overfitting and variance. In Logistic Regression, this is implemented through an additional term in Equation 2:

$$\mathcal{L} = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \log(\hat{p}_i) + (1-y_i)\log(1-\hat{p}_i)\right] + \lambda R(\beta) \tag{8}$$

Where λ controls the strength of regularization (larger values lead to stronger regularization), and $R(\beta)$ is some function of the model parameter magnitude. In L_1 regularization, $R(\beta) = \sum_j |\beta_j|$, and for L_2 regularization, $R(\beta) = \sum_j (\beta_j)^2$. L_1 tends to decrease parameter values to 0 in a linear fashion, whereas L_2 causes parameters to asymptotically decrease towards, but never exactly to 0.

In the context of Gradient Boosting with XGBoost, regularization is applied to the loss function in the form:

$$\mathcal{L} = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \log(\hat{p}_i) + (1-y_i)\log(1-\hat{p}_i)\right] + \lambda\left(T + R(\beta)\right) \tag{9}$$

Where T is the number of leaves in the tree. Regularization is also applied to the weights directly, via modification of Equation 7 as implemented by Chen and Guestrin (2016):

$$\omega_j = \frac{\sum_{i\in j} g_i}{\sum_{i\in j} h_i + \lambda} \tag{10}$$

Generally, model performance varies logarithmically with λ, therefore it is advised that test values be sampled on a logarithmic scale when optimizing for performance.

3.3.2 Tree Hyperparameters

Nonparametric models such as Random Forest do not incorporate explicit regularization terms. Instead, they are controlled through structural hyperparameters that constrain model complexity. As discussed in Section 3.1.2, **maximum depth** is a key hyperparameter that limits the number of hierarchical decision rules in each tree, thereby directly affecting overfitting. Additional parameters—such as the **minimum number of samples per leaf**, the **cost-complexity pruning parameter** (α), and the **number of estimators** (trees)—can also be tuned to control generalization error (Pedregosa et al. 2011). However, to reduce computational cost and simplify the tuning process, only maximum depth and the number of estimators were optimized in this analysis.

3.3.3 Random Search Cross-Validation

Given a candidate model and a set of tunable hyperparameters, an optimization problem naturally arises: which hyperparameter configuration yields the best model performance? To address this, the present analysis employed random search cross-validation to tune hyperparameters. The process is illustrated in Figure 2.

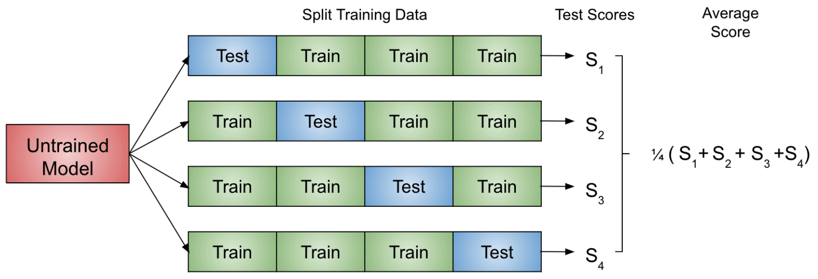

Figure 2: An example of four-fold cross-validation. A given model with a configuration of hyperparameters is trained four times, each time leaving out one subset of the data as a hold-out validation set. The model is evaluated on the hold-out fold, and the resulting scores are averaged. This process is repeated for multiple hyperparameter configurations. The configuration with the best average score is selected for further evaluation. Many scoring metrics exist depending on the use case and data characteristics; see Section 3.6 for details on the metrics used in this analysis.

Cross-validation mitigates the risk of overfitting by simulating model performance on unseen data through repeated training on subsets of the data while reserving a separate fold for validation. Averaging the resulting scores provides a more realistic estimate of generalization performance than fitting on the entire training set alone.

In random search, hyperparameter values are sampled from predefined distributions at each iteration, offering an efficient alternative to grid search, which exhaustively evaluates all combinations of specified values. Although grid search can be effective for low-dimensional hyperparameter spaces, it quickly becomes computationally prohibitive as the number of parameters increases. Accordingly, random search was chosen for its efficiency and scalability in this analysis.

3.4 Sequential Deep Learning Models

While the previously discussed models perform well across a range of supervised learning tasks and provide a strong performance baseline, they are limited by their assumption that each input instance is independent. This assumption is ill-suited to the sequential structure of the vegetation index data in this study, which exhibits temporal dynamics and potential spatial correlations between pixels within sites. To better model these dependencies, the final phase of the analysis employed sequential deep learning architectures based on Recurrent Neural Networks (RNNs), specifically Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) models. Despite their increased complexity and computational demands, these models are efficiently implemented using modern deep learning libraries in Python, such as PyTorch (Paszke et al. 2019).

3.4.1 Recursive Neural Network (RNN)

The simplest deep learning model that supports sequential modelling is the RNN. Figure 3 outlines the architecture of this model.

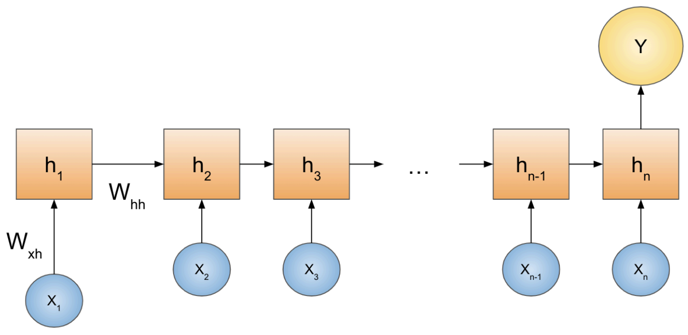

Figure 3: Basic architecture of a many-to-one RNN. Inputs from a sequence of vectors $(X_1, X_2, ..., X_n)$ are taken one-by-one, which updates a hidden state vector h_i according to a linear transformation with a weight matrix W_{xh}. As furher inputs are processed, the hidden state recursively updates according to the input as well as the previous hidden state through the weight matrix W_{hh}. A many-to-one RNN outputs one prediction Y after the final entry of the sequence is processed.

The key component of the RNN is the **hidden state**, which encodes the 'memory' of previous instances in the sequence. The transformation of the hidden state is governed by weight matrices W_{hh} and W_{xh}. Additionally, bias vectors b_{xh} and b_{hh} are included, and the linear transformation is passed through the hyperbolic tangent (tanh) function to introduce nonlinearity. Therefore, the hidden state h_t at time t in the sequence is updated given the previous hidden state h_{t-1} and current sequence entry x_t according to the transformation:

$$h_t = \tanh\left(x_t W_{xh}^T + b_{xh} + h_{t-1} W_{hh}^T + b_{hh}\right) \tag{11}$$

Although the RNN is capable of capturing short term dependencies in sequential data, long-term trends are difficult to capture due to issues of 'vanishing' and 'exploding' gradients during training (Pascanu, Mikolov, and Bengio 2013). See Section 3.5 for further details regarding this.

3.4.2 Long-Term Short Term Memory (LSTM)

To address the long-term dependency issue regarding RNNs, several models of similar, but more complex architecture have been proposed. One such model is the LSTM, which includes additional weights in the form of **input**, **output**, **cell**, and **forget** gates (i_t, o_t, g_t, f_t) respectively. These gates determine which aspects of the prior hidden state and current input are 'important' for prediction. These gates are used to update the cell state c_t, which is then used to update the current hidden state according to the equation:

$$\begin{aligned}
i_t &= \sigma(W_{xi}x_t + b_{xi} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{xf}x_t + b_{xf} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{xg}x_t + b_{xg} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{xo}x_t + b_{xo} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned} \tag{12}$$

Where $\odot$ represents the Hadamard product (elementwise multiplication of vector entries) and σ represents the sigmoid function introduced in Equation 3. The cyclical behaviour of the hidden state update helps to control for problematic gradients during training, making the LSTM suitable for many long-term sequential modelling and prediction tasks (Sak, Senior, and Beaufays 2014). However, the introducion of several new weight matrices and bias vectors can lead to excessively complex models that take extensive time to train.

3.4.3 Gated Recurrent Unit (GRU)

Like LSTMs, GRUs were developed to handle the vanishing gradient problem of RNNs. However, GRUs only utilize a **reset**, **update**, and **candidate** hidden state gate: $(r_t, z_t, \hat{h}_t)$. This allows for a lighter model than the LSTM, often with comparable performance on certain tasks (Ravanelli et al. 2018). Hidden states are updated according to the equation:

$$\begin{aligned}
r_t &= \sigma(W_{ir}x_t + b_{ir} + W_{hr}h_{t-1} + b_{hr}) \\
z_t &= \sigma(W_{iz}x_t + b_{iz} + W_{hz}h_{t-1} + b_{hz}) \\
\hat{h}_t &= \tanh(W_{in}x_t + b_{in} + r_t \odot (W_{hn}h_{t-1} + b_{hn})) \\
h_t &= (1 - z_t) \odot \hat{h}_t + z_t \odot h_{t-1}
\end{aligned} \tag{13}$$

3.4.4 Bidirectional RNNs

In addition to the usage of GRUs and LSTMs, additional hidden layers that update in reverse sequential direction may also be used to capture more complex, past and future dependencies. The hidden states can then be concatenated and used for prediction, as shown in Figure 4.

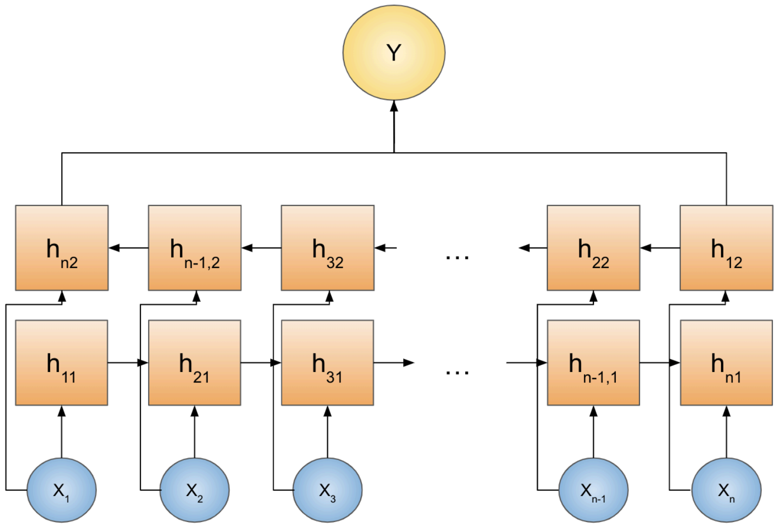

Figure 4: A schematic of a bidirectional RNN, where inputs are passed to hidden states in the forward and reverse temporal direction. The produced hidden states can be combined via concatenation, addition, or averaging to produce a final vector to be used for predicion.

In this analysis, including bidirectional layers appeared to slightly improve model performance. See Section 4 for further details.

3.4.5 Fully Connected Neural Network (FCNN)

The RNN model produces a hidden state which acts as a vectorization the processed vegetation index sequence. To convert this to a single scalar prediction, a **Fully Connected Neural Network (FCNN)** layer can be used. The architecture of this model is shown in Figure 5.

Inputs are passed between layers through a linear transformation using weight matrices and bias vectors, wrapped by an activation function to scale outputs and introduce nonlinearity into the system. For example, the first hidden layer $h^{(1)}$ in Figure 5 is produced by the transformation:

$$h^{(1)} = \text{ReLU}\left(W^{(1)}x + b^{(1)}\right)$$

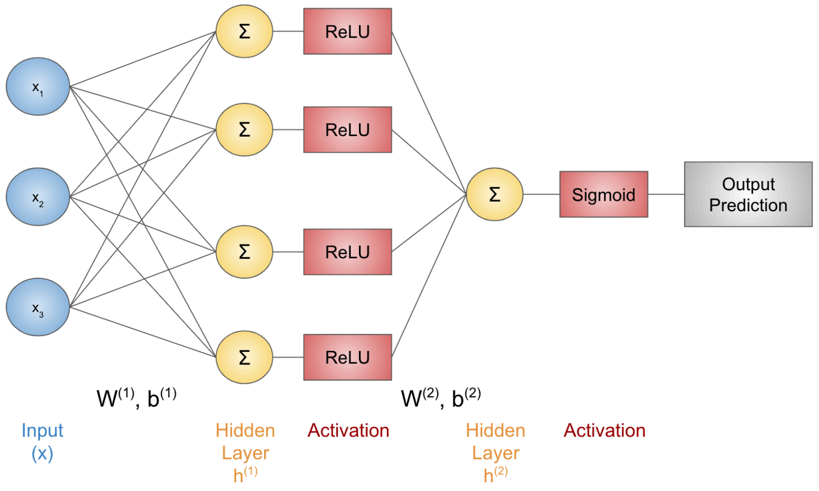

Figure 5: An example of an FCNN of two layers, with a three-dimensional input and one-dimensional output.

Where $W^{(1)}$ is a 4x3 matrix and $b^{(1)}$ is a vector of length 4, both of which have values that are learned during training. A common activation function is the **Rectifier Linear Unit (ReLU)** which simply has the form:

$$\text{ReLU}(x) = \max(0, x) \tag{14}$$

Although Equation 3 is also a common activation function due to its controlled range of (0,1). Generally, more hidden layers and higher dimensional hidden layers allow for higher performing models, at the cost of model complexity and training time.

3.4.6 Proposed Model Architecture

Following preprocessing (Section 2), the deep learning pipeline —comprising components from Section 3.4.2 through Section 3.4.5— proceeds as follows:

1. The sequence of vegetation indices and engineered features is passed through a bidirectional GRU or LSTM, producing a hidden state.
2. The static site features are concatenated onto the hidden state vector and passed to a multilayer FCNN.
3. The final layer of the FCNN output is a scalar, which is passed through a sigmoid activation and multiplied by 100 to produce an estimate of the survival rate of the site pixel.

In addition to these steps, **layer normalization** -which normalizes hidden layer output as a method of stabilizing predictions- was experimented with, although no improvement to predictions was observed. **Dropout** -which randomly sets some parameter values to zero as a method of regularization- was also added within the FCNN layers. Modelling was attemped with and without site features to assess their usage in predicting survival rate. Hidden state and hidden layer sizes, and the number of hidden layers are all variable hyperparameters than may effect model performance (Greff et al. 2017).

3.5 Training Deep Learning Models

Training relied on mini-batch stochastic gradient descent (SGD)—specifically the Adam optimizer, which combines momentum and adaptive learning rates. Gradients of model parameters are computed via backpropagation (chain rule) and updated in the direction opposite to the gradient, scaled by the learning rate. Mini-batch SGD, using shuffled subsets of training data, was employed to improve convergence speed and generalization.

Deep or recurrent architectures (e.g., long sequences or many layers) can suffer **vanishing gradients**, where gradient magnitudes shrink exponentially, or **exploding gradients**, where they grow uncontrollably—both impeding effective training. To mitigate these, we applied regularization techniques from Section 3.4.6 (e.g., dropout, layer normalization) and utilized GRU/LSTM architectures designed to ease gradient propagation.

Training used the **Mean Squared Error (MSE)** loss:

$$\mathcal{L}_{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

where y_i and $\hat{y}_i$ are true and predicted survival rates. MSE is appropriate for continuous targets -especially when additional penalization of drastic errors is reqiured- however this introduced a methodological discrepancy between this phase of deep modeling and the classification-oriented baselines, which is addressed in Section 3.6. Further discussion on transforming targets into binary form appears in Section 2.

3.6 Error Metrics

Model performance was primarily evaluated using F_1 score, **precision**, and **recall**, with classification accuracy treated as a secondary metric due to class imbalance favoring high-survival sites. To enable comparison between classical classification models and deep learning regression models, continuous survival rate predictions were thresholded to produce binary labels. In the context of these metrics, **low-survival sites are treated as the positive class**, reflecting the goal of identifying potentially failing afforestation sites for targeted intervention.

3.6.1 Precision, Recall, F_β Score

The **precision** of a classifier measures the proportion of true positive (TP) predictions among all instances predicted as positive, including both true positives and false positives (FP):

$$\text{Precision} = \frac{\sum \text{TP}}{\sum(\text{TP} + \text{FP})}$$

In contrast, **recall** (also known as sensitivity or true positive rate) measures the proportion of true positive predictions among all actual positive instances, including false negatives (FN):

$$\text{Recall} = \frac{\sum \text{TP}}{\sum(\text{TP} + \text{FN})}$$

Recall is particularly important in this imbalanced classification task, as it reflects the model's ability to correctly identify **low-survival (high-risk)** afforestation sites—those most in need of intervention.

To balance both precision and recall in a single metric, the F_β **score** is used, defined as the weighted harmonic mean:

$$F_\beta = (1 + \beta^2) \cdot \frac{\text{Precision} \cdot \text{Recall}}{(\beta^2 \cdot \text{Precision}) + \text{Recall}}$$

Larger values of β emphasize recall more heavily. This analysis primarily used the F_1 **score** ($\beta = 1$), which equally weights precision and recall, and also reported the F_2 **score** as a secondary metric to emphasize recall in support of the primary goal of detecting unhealthy sites.

3.6.2 ROC and PR Curves

To evaluate classifier performance across a range of decision thresholds, two widely used diagnostic tools are the **Receiver Operating Characteristic (ROC) curve** and the **Precision–Recall (PR) curve**.

The **ROC curve** plots the **true positive rate** (recall) against the **false positive rate** (FPR) at various classification thresholds:

$$\text{TPR} = \frac{TP}{TP + FN}, \qquad \text{FPR} = \frac{FP}{FP + TN}$$

It summarizes a model's ability to distinguish between the positive and negative classes, regardless of their prevalence. **Note:** this treshold is intrinstic to the probabilistic predictions of the model, and is **not** the same threshold used in the preprocessing target conversion.

The **area under the ROC curve (AUC-ROC)** provides a scalar summary of this performance; a value of 1 indicates perfect separation, while 0.5 indicates no discriminative ability.

ROC curves can be misleading in **imbalanced classification problems**, where the majority class dominates performance metrics. In this analysis, most afforestation sites exhibit **high survival**, so the model may appear to perform well overall even if it fails to identify the few low-survival sites of interest. The **Precision–Recall (PR) curve** more directly reflects performance on the **minority (positive) class**, which in this case is defined as **low-survival** sites. This curve focuses on the trade-off between **false positives** and **false negatives** when trying to identify high-risk sites. The **area under the PR curve (AUC-PR)** is more informative than AUC-ROC in this context because it penalizes false positives more explicitly and does not assume a balanced label distribution.

4 Data Product & Results

Our primary data product is a self-contained, reproducible Python/Quarto repository that provides partner analysts with a turnkey mechanism to (1) preprocess new satellite and silviculture data, (2) train or update models, and (3) evaluate survival-risk predictions for newly planted forest sites. The repository includes:

1. **Python Package (`src/`)**
   - Implements all data-preprocessing steps (`load_data.py`, `preprocess_features.py`, `pivot_data.py`, `data_split.py`), modelling pipelines (`gradient_boosting.py`, `logistic_regression.py`, `rnn.py`), and evaluation routines (`error_metrics.py`).
   - Exposes high-level Make targets so that running `make time_series_train_data` or `make train_models` executes the entire workflow end-to-end.
   - Facilitates future extension: partners can add new features, incorporate additional spectral indices, or replace models without rewriting core scripts.
2. **Versioned Model Artifacts (`models/`)**
   - For the 70% survival threshold, we provide five trained model files:
     - `logreg_model_70.pickle` (Logistic Regression)
     - `rf_model_70.pickle` (Random Forest)
     - `gbm_model_70.pickle` (XGBoost)

  - `gru_model_70.pth` (GRU network)
  - `lstm_model_70.pth` (LSTM network)

- Each artifact is accompanied by its hyperparameter configuration and training logs, enabling reproducibility and auditability.
- Partners can load any artifact in a separate environment for inference or further fine-tuning.

4.1 Intended Usage

Partner analysts should leverage this product to:

- **Rapidly assess survival risk** for new planting cohorts as soon as the latest Landsat composites and silviculture records are available.
- **Prioritize field surveys** by focusing on sites flagged as "high-risk" (predicted low survival) to allocate limited labour and remediation resources.
- **Iteratively retrain** models when new Year-7 survey data arrive, ensuring that the classifier adapts to evolving geographic or stand conditions.

4.2 Pros & Cons of the Current Interface

Pros
- **Simplicity**: A single `Makefile` orchestrates the entire pipeline, minimizing command-line complexity.
- **Modularity**: Individual scripts (`src/data/*.py`, `src/models/*.py`, `src/training/*.py`) can be modified or replaced without breaking the workflow.
- **Reproducibility**: Version control of code, hyperparameters, and environment specifications (via `environment.yml`) ensures partners can recreate any result on a new machine.
- **Transparency**: All intermediate Parquet files and logs are stored, making it easy to trace back from final predictions to raw inputs.

Cons
- **Compute Requirements**: Full data-preprocessing and model training (especially the GRU) require significant CPU/GPU resources. Partners without a compatible HPC or GPU-equipped workstation may experience long runtimes.
- **Command-Line Interface**: Although `Makefile` targets simplify execution, partners unfamiliar with command-line tools may face a learning curve.
- **Monolithic Outputs**: The current product writes large Parquet files (~15k per-site series)

which can strain disk space.
- **Minimal Web Interface**: There is no interactive dashboard; partners must inspect outputs in Python or Quarto. Developing a web-based front end (e.g., Streamlit or Dash) could improve user experience.

4.3 Justification & Comparison

Compared to alternative approaches—such as distributing only a standalone Jupyter notebook or providing a cloud-hosted API—our package-based product:

- **Reduces dependency on continuous internet access**: All code and data live locally once cloned, so partners in remote offices can run the pipeline offline.

- **Enables customization**: Internal data scientists can incorporate new sensors (e.g., Sentinel-2 or LiDAR) by extending `get_time_series.py` rather than rewriting a monolithic notebook.

- **Avoids vendor lock-in**: No reliance on commercial platforms or paid APIs; the entire stack uses open-source Python libraries.

However, a cloud-hosted API might be preferable if partners require real-time web access or integration with other information systems. Our current approach trades off ease of immediate integration for maximal transparency and reproducibility.

4.4 Results Overview

On held-out (20%) test data for the 70% canopy-retention threshold:

Phase 1: Classical Models

- **Logistic Regression**: performance metrics (Precision, Recall, F1, AUC) will be inserted here.
- **Random Forest**: performance metrics (Precision, Recall, F1, AUC) will be inserted here.
- **XGBoost (GBM)**: performance metrics (Precision, Recall, F1, AUC) will be inserted here.

4.5 Phase 2: Sequence Model Evaluation

- **GRU (spectral+static):** performance metrics (Precision, Recall, F1, AUC) will be inserted here.

- **LSTM (spectral+static):** performance metrics (Precision, Recall, F1, AUC) will be inserted here.

  Insight: Placeholder comment on sequence model performance.

- The **XGBoost model** strikes the best balance (highest F1) for identifying low- survival sites, making it the recommended default for partner deployment.

- The **GRU** achieves comparable AUC and better captures temporal signals, suggesting a potential future improvement once partners can provision GPU resources.

- **Logistic regression** serves as a transparent baseline; its lower AUC and F1 indicate that non-linear interactions among spectral indices and stand attributes are important.

The complete confusion matrix for XGBoost (70% threshold) is:

- **LSTM (spectral+static):** performance metrics (Precision, Recall, F1) will be inserted here.

4.6 Potential Enhancements

- **CNN-LSTM Hybrid:** Add convolutional layers that learn spatial correlations among spectral indices (e.g., local band interactions) before passing the extracted feature maps into an LSTM layer for temporal modeling. This approach can improve predictive accuracy by jointly capturing spatial and temporal patterns in the data.

By focusing on a reproducible, modular product now, we enable partners to adopt and extend the workflow flexibly, while future iterations can add web interfaces and advanced features as computational resources permit.

5 Conclusions & Recommendations

We developed a fully reproducible pipeline covering:

1. **Data Ingestion & Preparation**
   - Loaded raw RDS rasters and planting/survey tables via `load_data.py`.
   - Cleaned static features (`preprocess_features.py`) and extracted per-site time series (`pivot_data.py`).
   - Created consistent train/test datasets with `data_split.py`.

2. **Model Development & Benchmarking**
    - Phase 1: Trained and compared classical classifiers (Logistic Regression, Random Forest, XGBoost) using fixed-length features.
    - Phase 2: Trained GRU and LSTM networks on temporal sequences, demonstrating the benefit of sequence data.
3. **Evaluation & Results Interpretation**
    - Employed grouped 5-fold CV to avoid spatial leakage, optimized F1 for model selection, and reported Precision, Recall, F1, and AUC on held-out data.
    - XGBoost emerged as the top static model; GRU showed promise for temporal modeling.
4. **Operational Recommendations**
    - **Use XGBoost models as a reference** for identifying high-risk sites, rather than for direct operational deployment.
    - **Explore GRU models** as proof-of-concept for temporal analysis, acknowledging their current performance limitations.
    - **Treat all model outputs as advisory** insights; validate predictions with local surveys before taking field action.
    - **Enhance data** through expanded Year-7 surveys and new covariates (soil, LiDAR).
    - **Automate retraining** via Makefile and CI to adapt to new data.

By integrating data engineering, machine learning, and reproducible DevOps practices, this workflow provides a scalable tool for early afforestation monitoring and resource prioritization.

Tip

Rendering Instructions

```
cd reports/technical
conda activate mds-afforest-dev
quarto render report.qmd
open report.pdf
```

Bergmüller, Kai O, and Mark C Vanderwel. 2022. "Predicting Tree Mortality Using Spectral Indices Derived from Multispectral UAV Imagery." *Remote Sensing* 14 (9): 2195.

Chen, Tianqi, and Carlos Guestrin. 2016. "XGBoost: A Scalable Tree Boosting System." In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 785–94. KDD '16. ACM. https://doi.org/10.1145/2939672.2939785.

Greff, Klaus, Rupesh K. Srivastava, Jan Koutník, Bas R. Steunebrink, and Jürgen Schmidhuber. 2017. "LSTM: A Search Space Odyssey." *IEEE Transactions on Neural Networks and Learning Systems* 28 (10): 2222–32. https://doi.org/10.1109/TNNLS.2016.2582924.

Lundberg, Scott M, and Su-In Lee. 2017. "A Unified Approach to Interpreting Model Predictions." In *Advances in Neural Information Processing Systems 30*, edited by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, 4765–74. Curran Associates, Inc. http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf.

Pascanu, Razvan, Tomas Mikolov, and Yoshua Bengio. 2013. "On the Difficulty of Training Recurrent Neural Networks." https://arxiv.org/abs/1211.5063.

Paszke, Adam, Sam Gross, Francesco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, et al. 2019. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." In *Advances in Neural Information Processing Systems*. Vol. 32. Curran Associates, Inc.

Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, et al. 2011. "Scikit-Learn: Machine Learning in Python." *Journal of Machine Learning Research* 12: 2825–30.

Ravanelli, Mirco, Philemon Brakel, Maurizio Omologo, and Yoshua Bengio. 2018. "Light Gated Recurrent Units for Speech Recognition." *IEEE Transactions on Emerging Topics in Computational Intelligence* 2 (2): 92–102. https://doi.org/10.1109/tetci.2017.2762739.

Sak, Haşim, Andrew Senior, and Françoise Beaufays. 2014. "Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition." https://arxiv.org/abs/1402.1128.

Wikipedia contributors. 2025. "Logistic Regression — Wikipedia, the Free Encyclopedia." https://en.wikipedia.org/w/index.php?title=Logistic_regression&oldid=1291686859.

Zhou, Zhi-Hua. 2025. *Ensemble Methods: Foundations and Algorithms.* CRC press.