

Remote Sensing for Forest Recovery: Technical Report

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

2025-06-16

Table of contents

1	Introduction	2
2	Data & Pre-processing	3
2.1	Phase 1: Static Preprocessing & Modelling	3
2.2	Phase 2: Sequence Preprocessing & Modelling	3
2.3	Evaluation Metrics	4
3	Modelling Specifications	4
3.1	Classial Modelling	4
3.1.1	Logistic Regression	4
3.1.2	Tree-Based Modelling	5
3.1.3	Random Forest Classifier	5
3.1.4	Gradient Boosting Classifier	6
3.2	Training Classical Models	6
3.2.1	Tree Hyperparameters	6
3.2.2	L1, L2, ElasticNet Regularization	6
3.2.3	Cross-Validation	6
3.3	Sequential Deep Learning Models	6
3.3.1	Recursive Neural Network (RNN)	7
3.3.2	Long-term Short Term Memory (LSTM)	7
3.3.3	Gated Recurrent Unit (GRU)	7
3.3.4	Bidirectional RNNs	7
3.3.5	Fully Connected Neural Network (FCNN)	7
3.3.6	Final Model architecture	7
3.4	Training Deep Learning Models	8

3.5	Error Metrics	8
3.5.1	Precision, Recall, F_β Score	8
3.5.2	ROC and PR Curves	8
4	Data Product & Results	8
4.1	Intended Usage	9
4.2	Pros & Cons of the Current Interface	9
4.3	Justification & Comparison	10
4.4	Results Overview	10
4.5	Phase 2: Sequence Model Evaluation	11
4.6	Potential Enhancements	11
5	Conclusions & Recommendations	12

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 questions, 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_weight="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 that a given record belongs to class 1 as a linear combination of the input features (Wikipedia contributors 2025):

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

Here,

- n denotes the sample size,
- $\mathbf{x}_i = [x_{i1}, x_{i2}, \dots, x_{iD}]$ is the D -dimensional feature vector for the i th 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 j th entry of β corresponds to the change in the log-odds associated with a one-unit increase in the j th feature, holding all other features constant.

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

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

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**:

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

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 [?@sec-metrics](#).

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

- Rule based modelling: branches represent decisions that split feature space, leaves represent subsets
- Greedy algorithm: at each step, get split feature and threshold that best minimizes loss
- Loss can be log loss, entropy, Gini Impurity
- Stop splitting when leaf has one target, or max depth/ min samples per leaf is reached
- max depth/min samples per leaf is user-specified hyperparameter, see more in cross-validation section

3.1.3 Random Forest Classifier

- Series of decision trees trained on subset of features and bootstrapped sample of data
- Typically large depth: each tree becomes ‘expert’ on particular subset of data
- Computationally expensive but each tree is independent, can be parallelized and is highly scalable.

3.1.4 Gradient Boosting Classifier

- Opposite logic of RF: each tree is low-depth, underfit
- First model $F_0(x)$ fit over entire dataset, predicts constant minimum of log-odds
- Next model $F_1(x)$ uses gradient of loss WRT previous prediction as features
- new prediction is $F_0(x) + \text{learning_rate} * F_1(x)$
- repeat until max number of estimators is reached: final prediction = $F_0(x) + \text{learning_rate} * \sum(F_i(x))$
- XGBoost implementation provides faster training, leaf weights as parameters for regularization (see [?@sec-regularization](#))

3.2 Training Classical Models

3.2.1 Tree Hyperparameters

- Control number of estimators and tree depth for model complexity
- Random Forest: number of features per tree (usually square root of total)
- Gradient Boosting: Learning rate, smaller = less contribution per tree and less overfitting

3.2.2 L1, L2, ElasticNet Regularization

- penalize loss by parameter magnitude for logistic regression
- Logistic Regression: parameters are β
- Gradient Boosting: weights calculated at each leaf in each tree using higher order derivatives of loss, see paper
- Larger parameter value = smaller weights, less overfitting

3.2.3 Cross-Validation

- Grid Search vs Random Search over hyperparameters
- Split data into folds, use one as simulated test fold
- Split by site to avoid leakage between folds
- Choose model with best predefined score, F1 in our case (see metrics section)
- CV feasible for NN training, use holdout validation set instead

3.3 Sequential Deep Learning Models

- Classical models do not naturally handle sequential data
- Literature indicates changes in VIs across time are vital to predicting survival rates
- Problem: require large and diverse datasets.

3.3.1 Recursive Neural Network (RNN)

- Inputs passed sequentially to model
- Each input updates a hidden state via a linear combination: $h_t = (W_{\{xh\}} * X_t + W_{\{hh\}} * h_{t-1})$
- Hidden state represents memory data across time
- Vanilla RNNs suffer from vanishing/exploding gradients (see section [?@sec-backpropagation](#))

3.3.2 Long-term Short Term Memory (LSTM)

- Uses keep and forget gates to capture long term memory more effectively
- Show equation for updating hidden state with image of cell

3.3.3 Gated Recurrent Unit (GRU)

- Simplified form of LSTM, one combined gate for forget and keep
- Show equation for updating hidden state with image of cell

3.3.4 Bidirectional RNNs

- Add concatenated backward layer for encoding time in both directions

3.3.5 Fully Connected Neural Network (FCNN)

- Linear transformation of input
- Activation functions: control output scales and add nonlinearity (show ReLU function and justify use)

3.3.6 Final Model architecture

- time series into GRU or LSTM
- concatenate with site features and pass to FCNN
- FCNN output goes to sigmoid function * 100 to predict survival rate
- Dropout Rate before prediction, show paper for evidence of improvement
- hidden layers size, size of hidden state, all hyperparameters

3.4 Training Deep Learning Models

- Loss Function: MSE
- Compute gradient of loss WRT to each parameter in model
- Show equation for FCCN params
- show time dependence for sequential parts
- Show gradient descent equation for updating parameters
- Mini-batch: Compute gradients on subsets of data per update step to reduce computational load
- Repeat over successive epochs (epoch = 1 run through of data)

3.5 Error Metrics

3.5.1 Precision, Recall, F_β Score

- definition of each
- justification of F1 over other Fbeta and direct accuracy

3.5.2 ROC and PR Curves

- define TPR, FPR, and curves
- AUROC, AUC to measure model performance
- reference curves in results

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
```

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