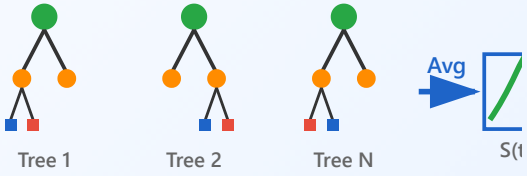
Time-to-Event Prediction with Machine Learning

Comprehensive Guide to Modern Survival Analysis Methods

Table of Contents

Survival analysis, also known as time-to-event analysis, focuses on predicting the time until an event of interest occurs. Machine learning has revolutionized this field by providing powerful methods that can handle complex, non-linear relationships and high-dimensional data.



Ensemble of survival trees
Bootstrap aggregation for robust predictions

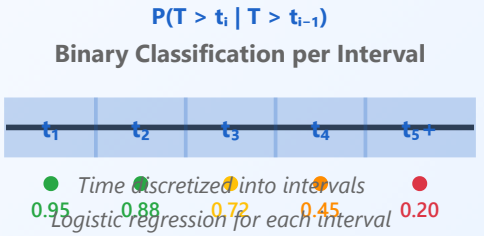
Random Survival Forests

Ensemble learning approach that combines multiple survival trees. Handles non-linear relationships and provides feature importance rankings.

Deep neural network architecture
Cox partial likelihood loss function

DeepSurv

Deep learning approach combining neural networks with Cox regression. Captures complex non-linear patterns in high-dimensional data.



Time discretized into intervals
Logistic regression for each interval

Discrete Time Models

Transforms survival analysis into sequence of binary classification problems. Flexible and easy to implement with standard ML tools.

Evaluation Metrics for Survival Analysis

- C-index (Concordance)
- Time-dependent AUC
- Calibration Plots
- Brier Score
- Integrated Brier Score

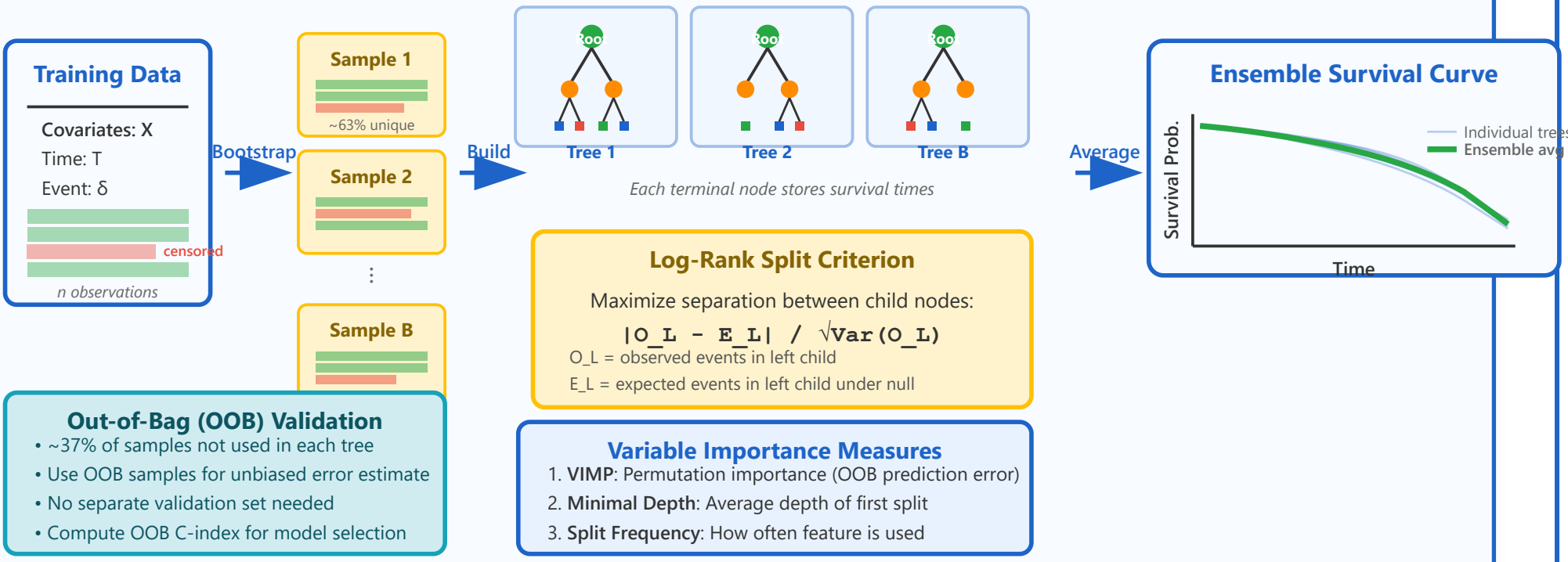
METHOD 1

Random Survival Forests (RSF)

Overview & Core Concept

Random Survival Forests extend the traditional Random Forest algorithm to handle censored survival data. Instead of predicting a class label or continuous value, RSF estimates the entire survival function S(t) for each individual, representing the probability of surviving beyond time t. The method builds an ensemble of survival trees, each trained on a bootstrap sample of the data, and aggregates their predictions to produce robust, non-parametric survival estimates that can capture complex patterns without assuming proportional hazards or specific distributional forms.

Complete RSF Architecture: From Data to Prediction



Complete workflow of Random Survival Forests: Data is bootstrap sampled, survival trees are grown using log-rank splitting, predictions from all trees are averaged to create ensemble survival function. OOB samples provide unbiased validation, and feature importance can be extracted.

▸ Key Features & Advantages

🌳 Non-parametric & Flexible

Makes no assumptions about the baseline hazard or proportional hazards. Can model complex, non-linear relationships between

🔍 Automatic Feature Selection

Automatically identifies important predictors through variable importance measures (VIMP, minimal depth). Robust to irrelevant features and multicollinearity.

covariates and survival times, including interactions that would be difficult to specify manually.

💔 Handles Missing Data

Can use surrogate splits when features have missing values, maintaining prediction accuracy even with incomplete data. No need for imputation in many cases.

📊 Built-in Cross-validation

Out-of-bag (OOB) samples (~37% per tree) provide unbiased estimate of prediction error without requiring a separate validation set. Efficient use of data.

⚡ Parallelizable Training

Trees are grown independently, allowing for parallel computation across multiple cores/machines. Scales well to large datasets.

🎯 Robust Predictions

Ensemble averaging reduces variance and overfitting. More stable predictions compared to single trees, especially with complex or noisy data.

▸ Algorithm Details & Implementation Steps

The RSF algorithm follows these key steps for building each survival tree in the forest:

- **Bootstrap Sampling:** Draw a random sample with replacement from the training data. Typically results in about 63% unique observations per tree, with remaining 37% as out-of-bag (OOB) samples for validation.
- **Random Feature Selection:** At each node during tree growing, randomly select a subset of m features from all p available features. Common choices: m = √p for survival data or m = p/3 for regression-type problems.
- **Split Point Selection:** For each selected feature, find the optimal split point that maximizes survival difference between daughter nodes using the log-rank test statistic. The split maximizes |O_L - E_L| / √Var(O_L), where O_L is observed events and E_L is expected events in the

left child.

- **Recursive Partitioning:** Recursively apply splitting until a stopping criterion is met: minimum node size reached (e.g., 3-15 observations), no significant splits available (p-value threshold), or maximum depth achieved.
- **Terminal Node Estimation:** In each leaf node, estimate the survival function using the Kaplan-Meier estimator based on the training samples that fall into that node. This gives $S_{leaf}(t)$ for all time points.
- **Ensemble Prediction:** For a new observation x, drop it down each of the B trees to obtain B terminal nodes. Collect the B survival function estimates and average them: $S_{ensemble}(t|x) = (1/B) \Sigma S_b(t|x)$.
- **Variable Importance:** Calculate feature importance using permutation: for each feature, randomly permute its values in OOB samples and measure decrease in prediction accuracy (C-index). Larger decreases indicate more important features.

⚡ Hyperparameter Tuning Guide

Number of trees (ntree): 500-1000 typically sufficient. More trees = more stable predictions but longer training. Error plateaus after sufficient trees.

Features per split (mtry): $\sqrt{p}$ is default and works well. Try $p/3$ or $\log_2(p)$ for alternatives. Smaller values = more diversity, larger values = stronger individual trees.

Minimum node size (nodesize): 3-15 typically. Larger values = simpler trees, less overfitting, faster training. Smaller values = more complex trees, potential overfitting.

Splitting rule: Log-rank (default), log-rank score, or conservation of events. Log-rank typically performs best for standard survival analysis.

Computational cost: Training time is $O(ntree \times n \times p \times \log n)$. Use parallel processing with multiple cores. Prediction is fast: $O(ntree \times \log n)$ per observation.

▸ Practical Example & Code Implementation

Clinical Application: Predicting cancer patient survival after diagnosis using clinical features (age, tumor stage, biomarkers), treatment information, and patient characteristics. RSF can identify complex interactions between age, tumor characteristics, and treatment response that proportional hazards models might miss.

```
# Python Implementation using scikit-survival
from sksurv.ensemble import RandomSurvivalForest
from sksurv.metrics import concordance_index_censored
import numpy as np
import matplotlib.pyplot as plt

# Initialize Random Survival Forest
rsf = RandomSurvivalForest(
    n_estimators=1000,          # Number of trees
    min_samples_split=10,       # Min samples to split node
    min_samples_leaf=15,        # Min samples in leaf
    max_features="sqrt",        # Features to consider per split
    n_jobs=-1,                  # Use all CPU cores
    random_state=42
)

# Train model
# y_train is structured array with fields 'event' (bool) and 'time' (float)
rsf.fit(X_train, y_train)

# Get survival function for new patients
surv_funcs = rsf.predict_survival_function(X_test)

# Plot survival curves for first 5 patients
```

```
for i, surv_func in enumerate(surv_funcs[:5]):
    plt.step(surv_func.x, surv_func.y, where="post",
             label=f"Patient {i+1}")
plt.xlabel("Time")
plt.ylabel("Survival Probability")
plt.title("Predicted Survival Curves")
plt.legend()
plt.show()

# Extract risk scores (higher = worse prognosis)
risk_scores = rsf.predict(X_test)

# Calculate C-index (concordance)
c_index = rsf.score(X_test, y_test)
print(f"C-index: {c_index:.3f}")

# Get feature importance
feature_importance = rsf.feature_importances_
important_features = np.argsort(feature_importance)[::-1][:10]
print("Top 10 important features:", important_features)
```

📚 When to Use Random Survival Forests

Ideal scenarios:

- Non-linear relationships between predictors and survival
- Need for variable importance/feature selection
- Moderate sample size (n > 100) with many features
- Missing data in predictor variables
- No strong prior belief in proportional hazards
- Want interpretable feature rankings

Less ideal for:

- Very small samples (n < 50)
- Need for explicit hazard ratios or covariate effects
- Real-time predictions with limited compute resources
- When linear Cox model performs adequately

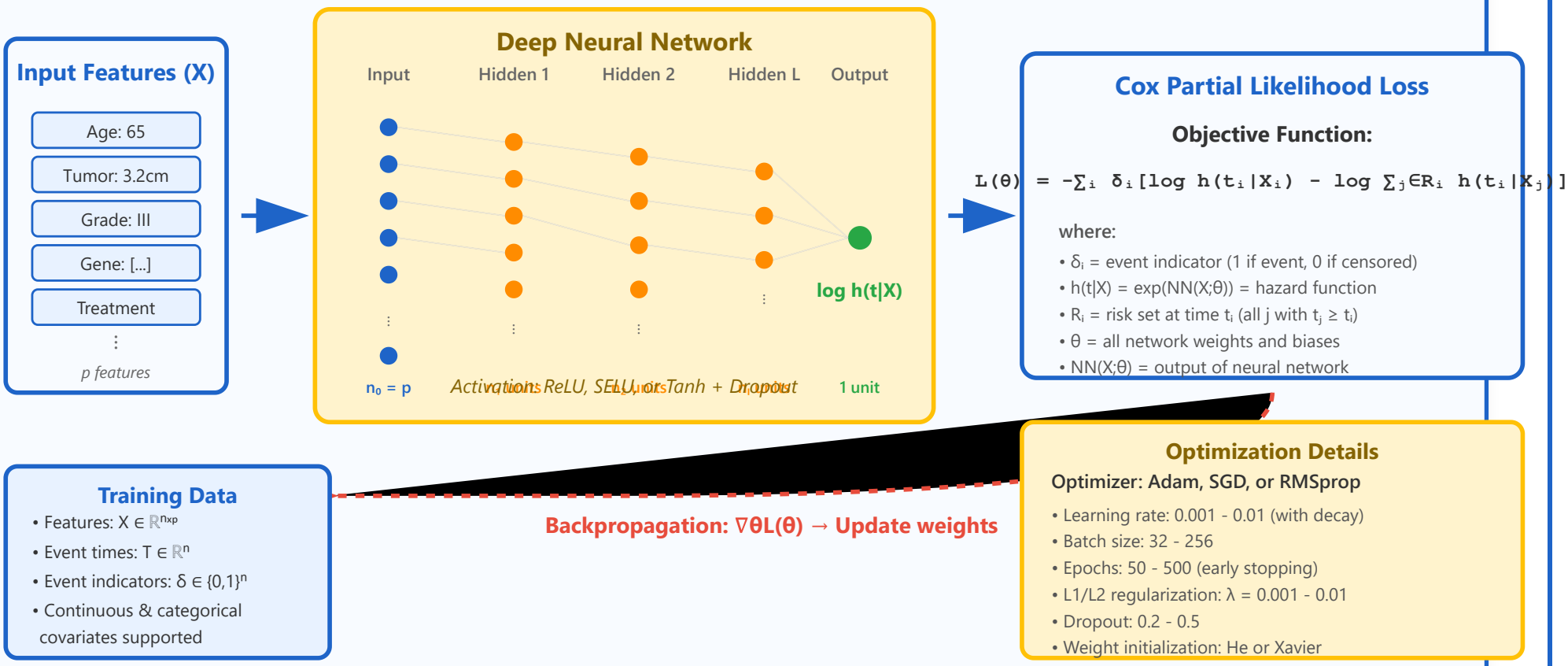
METHOD 2

DeepSurv: Deep Learning for Survival Analysis

Overview & Core Concept

DeepSurv is a deep neural network approach that extends the Cox proportional hazards model using deep learning architectures. Instead of assuming linear relationships between covariates and log-hazard like traditional Cox regression, DeepSurv uses a multi-layer feed-forward neural network to learn highly complex, non-linear representations of the input features. The network is trained using the Cox partial likelihood loss function, which maintains the semi-parametric nature and relative risk interpretation of Cox models while allowing the model to capture intricate patterns in high-dimensional data. This makes DeepSurv particularly powerful for genomic data, medical imaging features, or any scenario where relationships between predictors and survival are highly non-linear.

DeepSurv Architecture & Training Process



DeepSurv architecture showing input features processed through multiple hidden layers with non-linear activations, producing log-hazard predictions. The Cox partial likelihood loss enables end-to-end training via backpropagation while maintaining the interpretable hazard ratio framework.

▸ Key Features & Advantages

🧠 Deep Non-linear Representations

📈 High-dimensional Data Excellence

Multiple hidden layers can learn hierarchical, highly complex non-linear relationships between covariates and survival that shallow models cannot capture. Particularly powerful for discovering subtle patterns in high-dimensional data.

Excels with genomic data, medical imaging features, or other settings where p >> n (features greatly exceed samples). Deep architectures can perform implicit dimensionality reduction and feature learning.

🖇️ Automatic Interaction Learning

Automatically discovers complex interactions between features without manual specification. Hidden layers naturally combine features in non-linear ways that would be intractable to enumerate manually.

⚖️ Maintains Cox Interpretability

Uses Cox partial likelihood, so hazard ratios between patients can still be computed for risk stratification. Predictions have clear clinical interpretation as relative risks.

🎯 Transfer Learning Potential

Pre-trained networks from related tasks can be fine-tuned, leveraging knowledge from larger datasets. Particularly useful when labeled survival data is limited.

🔧 Flexible Architecture

Can incorporate various input types (continuous, categorical, images, text) with appropriate preprocessing layers. Architecture can be customized for domain-specific requirements.

▸ Network Architecture Design

Typical architecture components for DeepSurv:

- Input Layer: Dimensionality equals number of features (p). Common preprocessing: standardization (zero mean, unit variance) or normalization (min-max scaling to [0,1]). One-hot encoding for categorical variables.

- **Hidden Layers:** Usually 1-4 hidden layers with decreasing width pattern (e.g., 128 → 64 → 32 or 256 → 128 → 64). Deeper networks for very complex patterns, shallower for moderate complexity. Each layer has fewer neurons than previous to create information bottleneck.
- **Activation Functions:** ReLU (most common, fast, avoids vanishing gradients), SELU (self-normalizing, good for deep networks), or Tanh (bounded output). Applied element-wise after each hidden layer's linear transformation.
- **Dropout Layers:** Regularization via random neuron deactivation during training. Drop rate typically 0.1-0.5. Higher dropout for smaller datasets. Reduces overfitting by preventing co-adaptation of neurons.
- **Batch Normalization:** Normalizes layer inputs to have zero mean and unit variance. Stabilizes training, allows higher learning rates, acts as regularizer. Applied before or after activation.
- **Output Layer:** Single neuron with linear activation producing log-hazard ratio log h(t|X). No softmax or sigmoid needed since we're not doing classification. Output can be any real number.

```
# Example architecture specification
Layer 1 (Input):         p neurons  (features)
Layer 2 (Hidden 1):      128 neurons + ReLU + Dropout(0.3) + BatchNorm
Layer 3 (Hidden 2):      64 neurons  + ReLU + Dropout(0.3) + BatchNorm
Layer 4 (Hidden 3):      32 neurons  + ReLU + Dropout(0.2)
Layer 5 (Output):        1 neuron    (linear activation)

Forward pass:
  z1 = X
  z2 = ReLU(BatchNorm(W1·z1 + b1))  # Apply dropout in training
  z3 = ReLU(BatchNorm(W2·z2 + b2))
  z4 = ReLU(W3·z3 + b3)
  output = W4·z4 + b4  # log hazard
```

▸ Training Process & Optimization

DeepSurv is trained using mini-batch stochastic gradient descent with the Cox partial likelihood loss:

- **Batch Formation:** Divide training data into mini-batches of size B (typically 32-256). Each batch should ideally contain a mix of events and censored observations.
- **Forward Propagation:** Pass batch through network to compute log-hazards: $\hat{y}_i = NN(X_i; \theta)$ for each sample i in batch.
- **Loss Computation:** Calculate negative Cox partial log-likelihood on batch. For each observed event at time t_i with $\delta_i=1$, compute log likelihood contribution: $\hat{y}_i - \log(\sum_{j \in R_i} \exp(\hat{y}_j))$ where R_i is the risk set.
- **Backpropagation:** Compute gradients $\nabla\theta\ L(\theta)$ via automatic differentiation (chain rule applied through all layers). Modern frameworks (PyTorch, TensorFlow) handle this automatically.
- **Weight Update:** Update parameters using optimizer (Adam recommended): $\theta \leftarrow \theta - \alpha \cdot \nabla\theta L(\theta)$ where α is learning rate. Adam adapts learning rate per parameter based on gradient history.
- **Iteration:** Repeat for all batches (one epoch), then repeat for multiple epochs (50-500 typical) until convergence or early stopping triggered.

⚡ Training Best Practices

Learning Rate Strategy: Start with 0.001 (Adam default). Use ReduceLROnPlateau to decrease by factor of 10 when validation loss plateaus for 10+ epochs. Or use cyclic learning rates for better convergence.

Early Stopping: Monitor validation C-index every epoch. Stop if no improvement for 20-50 consecutive epochs. Save model with best validation performance, not the final epoch.

Regularization Balance: Start with moderate dropout (0.3) and L2 penalty (λ=0.01). If overfitting, increase. If underfitting, decrease. Can also use

L1 for feature selection.

Batch Size Selection: Larger batches (128-256) give more stable gradients but less regularization. Smaller batches (32-64) add noise that can help escape local minima. GPU memory constrains maximum size.

Initialization Matters: Use He initialization for ReLU (variance scales with $2/n_{in}$) or Xavier/Glorot for Tanh/Sigmoid. Poor initialization can cause vanishing/exploding gradients.

Validation Strategy: Use k-fold cross-validation or hold-out validation set. Never tune on test set. Monitor both C-index and calibration during training.

Implementation Example

```
# PyTorch DeepSurv Implementation
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.preprocessing import StandardScaler

class DeepSurv(nn.Module):
    def __init__(self, in_features, layers=[128, 64, 32], dropout=0.3):
        super(DeepSurv, self).__init__()

        # Build network layers
        modules = []
        prev_size = in_features

        for i, layer_size in enumerate(layers):
            # Fully connected layer
```

```
            modules.append(nn.Linear(prev_size, layer_size))
            # Batch normalization
            modules.append(nn.BatchNorm1d(layer_size))
            # Activation
            modules.append(nn.ReLU())
            # Dropout (less in final layers)
            drop_rate = dropout if i < len(layers)-1 else dropout*0.5
            modules.append(nn.Dropout(drop_rate))
            prev_size = layer_size

        # Output layer (log hazard)
        modules.append(nn.Linear(prev_size, 1))

        self.network = nn.Sequential(*modules)

    def forward(self, x):
        return self.network(x)

# Cox Partial Likelihood Loss
def cox_loss(log_hazards, times, events):
    # Sort by time (descending for risk set computation)
    idx = torch.argsort(times, descending=True)
    log_hazards = log_hazards[idx]
    events = events[idx]

    # Compute log risk = log(sum of exp(log_hazard) for risk set)
    # Use logsumexp for numerical stability
    hazard_ratio = torch.exp(log_hazards)
    log_risk = torch.log(torch.cumsum(hazard_ratio, dim=0))

    # Negative log partial likelihood (only for observed events)
    uncensored_likelihood = log_hazards - log_risk
    loss = -torch.sum(uncensored_likelihood * events) / torch.sum(events)
```

```
    return loss

# Training setup
model = DeepSurv(in_features=100, layers=[128, 64, 32], dropout=0.3)
optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=0.01)
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, patience=10)

# Standardize features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)

# Training loop
best_c_index = 0
patience_counter = 0

for epoch in range(500):
    model.train()
    epoch_loss = 0

    # Mini-batch training
    for batch_idx in range(0, len(X_train_scaled), batch_size):
        batch_X = torch.FloatTensor(
            X_train_scaled[batch_idx:batch_idx+batch_size])
        batch_times = torch.FloatTensor(
            times_train[batch_idx:batch_idx+batch_size])
        batch_events = torch.FloatTensor(
            events_train[batch_idx:batch_idx+batch_size])

        optimizer.zero_grad()
        log_hazards = model(batch_X)
        loss = cox_loss(log_hazards, batch_times, batch_events)
```

```
        loss.backward()
        optimizer.step()

        epoch_loss += loss.item()

    # Validation
    model.eval()
    with torch.no_grad():
        val_log_hazards = model(torch.FloatTensor(X_val_scaled))
        c_index = concordance_index(
            times_val, -val_log_hazards.numpy(), events_val)

    scheduler.step(epoch_loss)

    if c_index > best_c_index:
        best_c_index = c_index
        torch.save(model.state_dict(), 'best_model.pth')
        patience_counter = 0
    else:
        patience_counter += 1

    if patience_counter >= 30:
        print(f"Early stopping at epoch {epoch}")
        break

    if epoch % 10 == 0:
        print(f"Epoch {epoch}, Loss: {epoch_loss:.4f}, "
              f"Val C-index: {c_index:.4f}")

# Load best model for final predictions
model.load_state_dict(torch.load('best_model.pth'))
model.eval()
```

When to Use DeepSurv

Ideal scenarios:

- High-dimensional data (p >> n): genomics, imaging features
- Highly non-linear relationships expected
- Complex feature interactions present
- Sufficient training data available (n > 500 ideally)
- Computational resources available (GPU helpful)
- Want to leverage transfer learning from related domains

Less ideal for:

- Small sample sizes (n < 100)
- Simple linear relationships (Cox PHM may suffice)
- Need for explicit feature effect estimates
- Limited computational resources
- Interpretability is paramount over prediction accuracy
- Quick prototyping (training can take hours/days)

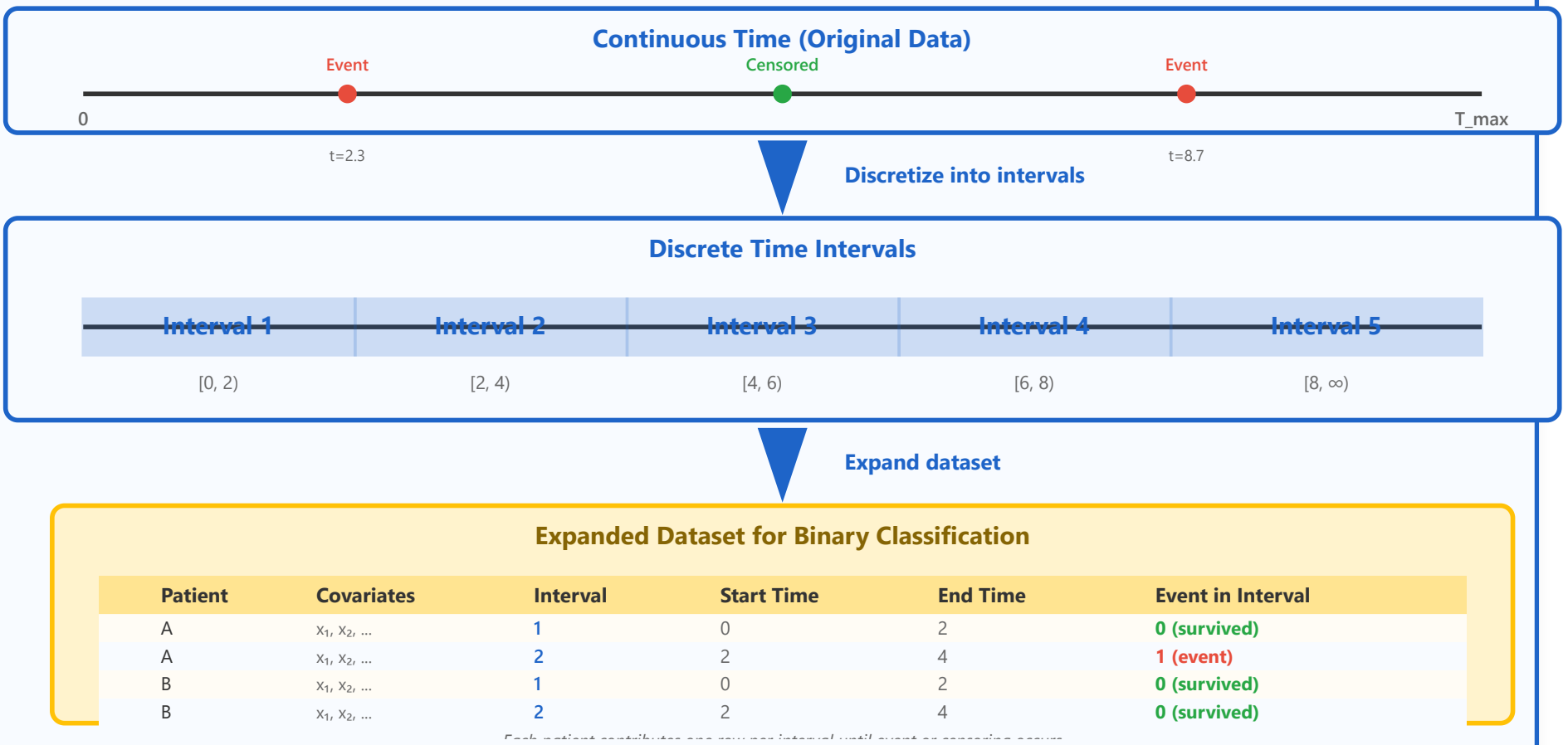
METHOD 3

Discrete Time Survival Models

Overview & Core Concept

Discrete time models reformulate survival analysis as a sequence of binary classification problems. Instead of modeling continuous time directly, the follow-up period is divided into discrete intervals, and for each interval, we predict whether an individual will experience the event during that specific interval, given they survived up to that point. This transforms the complex censored survival problem into a series of easier-to-model conditional probabilities: P(event in interval j | survived to interval j). The beauty of this approach is that it allows us to use any standard binary classification algorithm (logistic regression, random forests, gradient boosting, neural networks) while properly handling censored data through the data expansion process.

Discrete Time Framework: From Continuous to Intervals



Transformation from continuous survival data to discrete time format. Continuous timeline is divided into intervals, and the dataset is expanded so each patient contributes one observation per interval they are at risk. Binary outcome indicates whether event occurred in that interval.

▸ Mathematical Framework

The discrete time model estimates the **discrete hazard** $\lambda_j(X)$ for each interval j:

Discrete Hazard (probability of event in interval j):

$$\lambda_j(X) = P(T = t_j \mid T \geq t_j, X)$$

This is the conditional probability that an individual with covariates X experiences the event in interval j, given they survived until the start of that interval.

We model this using logistic regression:

$$\text{logit}[\lambda_j(X)] = \log[\lambda_j(X) / (1 - \lambda_j(X))] = \alpha_{0j} + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p$$

where:

- α_{0j} = baseline log-odds for interval j (interval-specific intercept)
- β = covariate effects (shared across all intervals by default)
- Can allow time-varying effects: β_j different for each interval

Survival function at time t_j:

$$S(t_j|X) = P(T > t_j|X) = \prod_{i=1}^{j} [1 - \lambda_i(X)]$$

This is the product of surviving each interval from 1 to j.

Probability of event in interval j:

$$P(T = t_j|X) = \lambda_j(X) \times S(t_{j-1}|X) = \lambda_j(X) \times \prod_{i=1}^{j-1} [1 - \lambda_i(X)]$$

▸ Key Features & Advantages

🎯 Simplicity & Flexibility

Converts survival analysis into standard binary classification. Can use ANY classification algorithm: logistic regression, random forests, gradient boosting, neural networks, SVM. No specialized survival software needed.

⏱️ Time-varying Effects

Easily model non-proportional hazards by allowing covariate effects to vary across intervals. Include interval indicators or interval × covariate interactions.

📊 Interpretable Probabilities

Direct probability predictions for each interval. Easy to communicate: "30% chance of event in next 6 months given survival so far." More intuitive than hazard ratios for some audiences.

🔧 Easy Implementation

Simple data preparation (expand to person-period format). Standard ML libraries (scikit-learn, XGBoost) work directly. Mature ecosystem of tools and techniques.

📈 Handles Competing Risks

Naturally extends to competing risks by using multinomial classification instead of binary. Can model multiple event types simultaneously with separate probabilities.

⚡ Computational Efficiency

Training is typically faster than RSF or DeepSurv. Standard optimized libraries. Predictions are very fast: just evaluate classifier for each interval.

▸ Implementation Steps

Step-by-step guide to implementing discrete time models:

- Step 1 - Define Intervals: Choose interval boundaries based on domain knowledge or data distribution. Options: equal width (e.g., every 6 months), equal frequency (quantiles of event times), or clinically meaningful periods.

- **Step 2 - Expand Dataset:** Transform data to person-period format. Each individual contributes one row per interval they are at risk. Include covariates and interval identifier for each row.
- **Step 3 - Create Binary Outcome:** For each row, outcome = 1 if event occurred in that interval, 0 if individual survived the interval. Exclude rows after event or censoring.
- **Step 4 - Add Interval Indicators:** Include dummy variables or numeric indicators for each interval. This allows baseline hazard to vary across time. Can use one-hot encoding or ordinal encoding.
- **Step 5 - Fit Classification Model:** Train any binary classifier on expanded dataset. Logistic regression for interpretability, tree-based methods for flexibility, neural networks for complex patterns.
- **Step 6 - Predict Survival Curves:** For new individual, predict probability for each interval sequentially. Multiply survival probabilities: $S(t_j) = \prod_{i=1}^{j} (1 - \hat{P}_i)$ where $\hat{P}_i$ is predicted probability for interval i.
- **Step 7 - Model Evaluation:** Use time-dependent C-index, Brier score, or calibration plots. Can evaluate within specific intervals or overall. Assess calibration by comparing predicted vs. observed event rates.

⚙️ Design Choices & Considerations

Interval Selection: Too few intervals = loss of temporal resolution. Too many intervals = sparse data per interval, increased variance. Typical: 5-20 intervals. Use event distribution to guide choice.

Handling Ties: Multiple events at same time are naturally handled since they fall in same interval. Unlike Cox models which require special handling.

Time-varying Covariates: Easy to incorporate! Just include time-updated covariate values in each row. Example: treatment changes, lab values, biomarker levels.

Class Imbalance: Typically few events per interval = imbalanced classes. Use: class weights, SMOTE, focal loss, or stratified sampling to address.

Interval Effects: Shared effects (β constant) assumes proportional odds. Time-varying effects (β_j) allows non-proportional hazards. Trade-off: flexibility vs. overfitting.

▸ Practical Example & Code

```
# Python implementation with pandas and scikit-learn
import pandas as pd
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import GradientBoostingClassifier
from lifelines.utils import concordance_index

def create_person_period(df, time_col, event_col, interval_breaks):
    """
    Expand survival data to person-period format

    Parameters:
    -----------
    df : DataFrame with survival data
    time_col : name of time column
    event_col : name of event indicator column (1=event, 0=censored)
    interval_breaks : list of interval boundaries [0, 2, 4, 6, 8, inf]

    Returns:
    --------
    Expanded DataFrame in person-period format
    """
    expanded_rows = []
```

```
    for idx, row in df.iterrows():
        time = row[time_col]
        event = row[event_col]

        # Determine which interval the event/censoring falls in
        interval_idx = np.searchsorted(interval_breaks, time, side='right') - 1

        # Create row for each interval up to event/censoring
        for j in range(interval_idx + 1):
            new_row = row.copy()
            new_row['interval'] = j
            new_row['interval_start'] = interval_breaks[j]
            new_row['interval_end'] = interval_breaks[j+1]

            # Event indicator for this interval
            if j < interval_idx:
                new_row['event_in_interval'] = 0  # Survived this interval
            else:
                new_row['event_in_interval'] = event  # Event or censored

            expanded_rows.append(new_row)

    return pd.DataFrame(expanded_rows)

# Example usage
# Original data
survival_data = pd.DataFrame({
    'time': [2.3, 5.1, 8.7, 3.2, 6.8],
    'event': [1, 0, 1, 1, 0],
    'age': [65, 72, 58, 61, 69],
    'treatment': ['A', 'B', 'A', 'B', 'A'],
    'biomarker': [2.3, 1.8, 3.1, 2.5, 1.9]
})
```

```
# Define intervals
interval_breaks = [0, 2, 4, 6, 8, np.inf]

# Expand to person-period format
pp_data = create_person_period(
    survival_data, 'time', 'event', interval_breaks
)

# Prepare features
# One-hot encode categorical variables
pp_data = pd.get_dummies(pp_data, columns=['treatment'])

# Create interval dummy variables (baseline hazard)
pp_data = pd.get_dummies(pp_data, columns=['interval'], prefix='int')

# Select features for modeling
feature_cols = ['age', 'biomarker', 'treatment_A', 'treatment_B',
                'int_0', 'int_1', 'int_2', 'int_3', 'int_4']
X = pp_data[feature_cols]
y = pp_data['event_in_interval']

# Fit logistic regression (or any classifier)
model = LogisticRegression(class_weight='balanced', max_iter=1000)
# Alternative: Gradient Boosting for non-linear relationships
# model = GradientBoostingClassifier(n_estimators=100)

model.fit(X, y)

# Predict survival curve for new patient
def predict_survival_curve(model, patient_data, interval_breaks):
    """Predict survival probability for each interval"""
    n_intervals = len(interval_breaks) - 1
```

```
    survival_probs = []
    survival_cum = 1.0

    for j in range(n_intervals):
        # Create features for interval j
        patient_data['interval'] = j
        patient_data_encoded = pd.get_dummies(
            patient_data, columns=['interval'], prefix='int'
        )

        # Predict probability of event in interval j
        prob_event = model.predict_proba(
            patient_data_encoded[feature_cols]
        )[0, 1]

        # Update cumulative survival
        survival_cum *= (1 - prob_event)
        survival_probs.append(survival_cum)

    return np.array(survival_probs)

# Predict for new patient
new_patient = pd.DataFrame({
    'age': [60],
    'biomarker': [2.0],
    'treatment': ['A']
})

surv_curve = predict_survival_curve(
    model, new_patient, interval_breaks
)
```

```
print("Survival probabilities by interval:", surv_curve)
```

📚 When to Use Discrete Time Models

Ideal scenarios:

- Time naturally discrete (monthly follow-ups, yearly events)
- Want to use standard ML libraries and tools
- Need time-varying covariate effects
- Prefer probability interpretations over hazards
- Have competing risks to model
- Want fast training and simple implementation
- Team familiar with classification but not survival analysis

Less ideal for:

- Precisely measured continuous time is critical
- Need smooth hazard function estimates
- Very fine time resolution required
- Small sample size with many intervals (overfitting risk)
- Strong preference for proportional hazards framework
- Want traditional hazard ratio interpretations

SECTION 5

Method Comparison & Selection Guide

Comprehensive Comparison Table

Aspect	Random Survival Forests	DeepSurv	Discrete Time Models
Core Approach	Ensemble of survival trees with bootstrap aggregation	Deep neural network with Cox loss	Sequential binary classification per interval
Assumptions	None (fully non-parametric)	Proportional hazards (like Cox)	Proportional odds (can relax with interactions)
Handles Non-linearity	✅ Excellent (tree splits)	✅ Excellent (deep layers)	✅ Good (depends on classifier choice)
Feature Interactions	✅ Automatic (implicit in trees)	✅ Automatic (hidden layers)	⚠️ Manual specification often needed
High-dimensional Data (p >> n)	✅ Good (random feature selection)	✅ Excellent (designed for this)	⚠️ Moderate (needs regularization)

Aspect	Random Survival Forests	DeepSurv	Discrete Time Models
Sample Size Required	Moderate (n > 100 preferable)	Large (n > 500 ideal)	Small-Moderate (n > 50 often sufficient)
Missing Data Handling	✅ Native support (surrogate splits)	❌ Requires imputation	❌ Requires imputation
Interpretability	⚠️ Variable importance, partial dependence	❌ Black box (can use SHAP/LIME)	✅ Good (especially with logistic regression)
Training Time	Moderate (parallelizable)	Long (GPU helps significantly)	Fast (standard classification)
Prediction Speed	Fast	Very fast (once trained)	Very fast
Implementation Complexity	Low (good packages available)	Moderate-High (need DL framework)	Low (standard ML tools)
Time-varying Effects	⚠️ Possible but complex	⚠️ Requires architecture changes	✅ Easy (interval × covariate interactions)
Best For	Moderate data, need variable importance, non-linear patterns	Large datasets, genomics, complex non-linearity, deep patterns	Discrete time, simple implementation, time-varying effects

▸ Decision Flowchart

When choosing a method, consider these key questions:

- **Q1: What is your sample size?**
  - Small (n < 100): Discrete Time Models or traditional Cox
  - Moderate (100 < n < 500): Random Survival Forests
  - Large (n > 500): DeepSurv becomes viable
- **Q2: How many features do you have relative to sample size?**
  - p >> n (high-dimensional): DeepSurv or RSF with feature selection
  - p < n (low-dimensional): Any method works
- **Q3: Do you suspect strong non-linear relationships?**
  - Yes: RSF or DeepSurv
  - No/Uncertain: Start with Discrete Time + Logistic Regression
- **Q4: How important is interpretability?**
  - Critical: Discrete Time with Logistic Regression
  - Moderate: RSF (variable importance available)
  - Not critical: DeepSurv (focus on prediction)
- **Q5: Do you need to model time-varying effects?**
  - Yes: Discrete Time Models (easiest)
  - No: Any method works
- **Q6: What computational resources do you have?**

- Limited: Discrete Time Models (fastest)
- Moderate: RSF (parallelizable)
- Substantial (GPU access): DeepSurv

💡 Practical Recommendations

For most applications, start with Random Survival Forests: Good balance of performance, interpretability, and ease of use. Handles non-linearity well and provides variable importance.

Use DeepSurv when you have: Very large sample size (n > 1000), high-dimensional features (genomics, images), or complex non-linear patterns that RSF doesn't capture well.

Choose Discrete Time Models when: Time is naturally discrete, you need simple implementation with standard tools, or you want to model time-varying covariate effects easily.

Best practice: Try multiple methods! Use cross-validation to compare C-index, calibration, and Brier scores. The "best" method is dataset-specific. Ensemble predictions from multiple methods often works best.

🎯 Key Takeaways

- **Random Survival Forests** offer robust, non-parametric predictions with automatic variable importance. Great default choice for moderate-sized datasets.
- **DeepSurv** leverages deep learning to capture highly complex patterns in high-dimensional data. Best for large samples and genomic/imaging applications.
- **Discrete Time Models** simplify survival analysis to binary classification, enabling use of any ML algorithm. Ideal for discrete time and time-varying effects.
- **No universally "best" method** exists. Choice depends on: sample size, dimensionality, interpretability needs, computational resources, and domain requirements.
- **Evaluation is critical:** Use time-dependent C-index, calibration plots, and Brier scores. Validate on held-out data or via cross-validation.
- **Consider ensembles:** Combining predictions from multiple methods often improves performance and robustness.