# Survival Analysis Methodology

**Last Updated:** 2025-10-23

**Framework Version:** 1.0

This document describes the theoretical foundations and implementation approach for the survival analysis framework. It is maintained alongside code changes to ensure consistency between theory and implementation.

## Table of Contents

Table of Contents

[Survival Analysis Methodology 1](#_Toc212141110)

[Table of Contents 1](#_Toc212141111)

[Introduction 2](#_Toc212141112)

[Purpose 2](#_Toc212141113)

[Key Advantages 2](#_Toc212141114)

[Survival Analysis Fundamentals 2](#_Toc212141115)

[Core Concepts 2](#_Toc212141116)

[Model Implementations 4](#_Toc212141117)

[1. Cox Proportional Hazards (Cox PH) 4](#_Toc212141118)

[2. Elastic Net Cox (Coxnet) 5](#_Toc212141119)

[3. Weibull Accelerated Failure Time (AFT) 6](#_Toc212141120)

[4. Gradient Boosting Survival Analysis (GBSA) 7](#_Toc212141121)

[5. Random Survival Forest (RSF) 8](#_Toc212141122)

[Data Preprocessing Pipeline 8](#_Toc212141123)

[Pipeline Architecture 9](#_Toc212141124)

[Stage 1: Column Transformation 9](#_Toc212141125)

[Stage 2: Variance Threshold 9](#_Toc212141126)

[Stage 3: Survival Model 10](#_Toc212141127)

[Why This Design? 10](#_Toc212141128)

[Evaluation Metrics 10](#_Toc212141129)

[1. Concordance Index (C-index) 10](#_Toc212141130)

[2. Integrated Brier Score (IBS) 11](#_Toc212141131)

[3. Time-Dependent AUC 12](#_Toc212141132)

[Cross-Validation Strategy 13](#_Toc212141133)

[Event-Balanced Stratified K-Fold 13](#_Toc212141134)

[Fold-Aware Time Grid Constraints 14](#_Toc212141135)

[Cross-Validation Workflow 14](#_Toc212141136)

[Prediction Methodology 15](#_Toc212141137)

[Model Selection 15](#_Toc212141138)

[Survival Probability Predictions 15](#_Toc212141139)

[Expected Survival Time 15](#_Toc212141140)

[Output Format 16](#_Toc212141141)

[Implementation Architecture 17](#_Toc212141142)

[Module Structure 17](#_Toc212141143)

[Design Principles 17](#_Toc212141144)

[Key Abstractions 18](#_Toc212141145)

[References 19](#_Toc212141146)

[Theoretical Foundations 19](#_Toc212141147)

[Implementation References 20](#_Toc212141148)

[Changelog 20](#_Toc212141149)

[2025-10-23 - Initial Version 20](#_Toc212141150)

## Introduction

### Purpose

This framework implements survival analysis for predicting customer churn/termination events. Unlike binary classification (will churn: yes/no) or time-series forecasting, survival analysis jointly models:

1. **Whether** an event will occur (termination)
2. **When** the event will occur (survival time)
3. **Censored observations** where the event has not yet occurred

### Key Advantages

* Handles right-censored data (customers still active at observation end)
* Provides time-dependent risk predictions
* Estimates expected survival time for each customer
* Quantifies uncertainty through survival probabilities

## Survival Analysis Fundamentals

### Core Concepts

**Survival Function S(t)**

The survival function represents the probability that an individual survives beyond time t:

S(t) = P(T > t)

Where:

* T is the random variable representing survival time
* S(t) is monotonically decreasing: S(0) = 1, lim(t→∞) S(t) = 0
* In our context: probability that a customer remains active beyond t months

**Hazard Function h(t)**

The hazard function represents the instantaneous risk of the event occurring at time t, given survival up to t:

h(t) = lim(Δt→0) P(t ≤ T < t + Δt | T ≥ t) / Δt

Properties:

* h(t) ≥ 0 for all t
* Higher hazard = higher instantaneous risk
* Relationship to survival: S(t) = exp(-∫₀ᵗ h(u)du)

**Cumulative Hazard Function H(t)**

H(t) = ∫₀ᵗ h(u)du = -log(S(t))

**Censoring**

**Right Censoring** occurs when the event has not been observed by the end of the study period:

* **Event indicator**: δ = 1 if event observed, δ = 0 if censored
* **Observed time**: T\_obs = min(T\_event, T\_censoring)
* Censored observations provide partial information: "survived at least until T\_obs"

## Model Implementations

### 1. Cox Proportional Hazards (Cox PH)

**Theory**

The Cox PH model assumes the hazard function can be written as:

h(t|X) = h₀(t) · exp(β'X)

Where:

* h₀(t) is the baseline hazard (unspecified, non-parametric)
* β is the vector of coefficients
* X is the covariate vector
* exp(β'X) is the risk score (hazard ratio)

**Proportional Hazards Assumption**: The ratio of hazards for two individuals is constant over time.

**Implementation Details**

**Module**: `CoxPHWrapper` in `models.py`

* Uses `CoxPHSurvivalAnalysis` from scikit-survival
* **L2 Regularization** (Ridge): α = 1.0 by default to prevent numerical instability
* Optimization: Newton-Raphson with 200 iterations, tolerance 1e-9
* Input validation: checks for finite values, minimum sample size, non-zero variance

**Why L2 Regularization?**

* Prevents LAPACK errors from multicollinearity
* Stabilizes coefficient estimates with high-dimensional data
* Default α=1.0 balances stability and model flexibility

### 2. Elastic Net Cox (Coxnet)

**Theory**

Extends Cox PH with elastic net penalty (L1 + L2 regularization):

minimize: -log-likelihood + λ[(1-α)/2 ||β||₂² + α||β||₁]

Where:

* λ controls overall regularization strength
* α ∈ [0,1] controls L1/L2 mix
* α=1: Lasso (feature selection)
* α=0: Ridge (shrinkage only)
* 0<α<1: Elastic net (both)

**Implementation Details**

**Module**: `CoxnetWrapper` in `models.py`

* Uses `CoxnetSurvivalAnalysis` from scikit-survival
* Default: l1\_ratio=0.5 (balanced L1/L2)
* Automatic regularization path selection via cross-validation
* Feature selection through L1 penalty

### 3. Weibull Accelerated Failure Time (AFT)

**Theory**

AFT models assume covariates act multiplicatively on survival time:

log(T) = β'X + σε

Where:

* ε follows a specified distribution (Weibull in our case)
* σ is a scale parameter
* exp(β'X) is the acceleration factor

Weibull distribution:

S(t|X) = exp(-(λt)ᵖ) where λ = exp(-β'X/σ)

* p > 1: increasing hazard (aging)
* p < 1: decreasing hazard
* p = 1: constant hazard (exponential)

**Implementation Details**

**Module**: `WeibullAFTWrapper` in `models.py`

* Uses `WeibullAFTFitter` from lifelines
* Parametric model with interpretable acceleration factors
* Requires DataFrame input with 'time' and 'event' columns
* Directly models survival time distribution

### 4. Gradient Boosting Survival Analysis (GBSA)

**Theory**

Ensemble method that builds an additive model:

h(t|X) = h₀(t) · exp(∑ᵢ fᵢ(X))

Where each fᵢ is a regression tree fitted to negative gradients of the partial likelihood loss.

**Advantages**:

* Captures non-linear relationships
* Handles interactions automatically
* Robust to irrelevant features
* No proportional hazards assumption

**Implementation Details**

**Module**: `GBSAWrapper` in `models.py`

* Uses `GradientBoostingSurvivalAnalysis` from scikit-survival
* Tree-based ensemble with boosting
* Flexible non-parametric approach
* Often achieves best predictive performance

### 5. Random Survival Forest (RSF)

**Theory**

Extension of random forests to survival data:

1. Bootstrap sample from training data
2. At each node, randomly select subset of features
3. Split node to maximize survival difference (log-rank test)
4. Aggregate predictions across trees

**Cumulative Hazard Estimate**:

Ĥ(t|X) = average over trees of cumulative hazard in terminal node

**Implementation Details**

**Module**: `RSFWrapper` in `models.py`

* Uses `RandomSurvivalForest` from scikit-survival
* Ensemble of survival trees
* Handles non-linear patterns and interactions
* Provides feature importance measures

## Data Preprocessing Pipeline

### Pipeline Architecture

The preprocessing pipeline (`make\_pipeline()` in `data.py`) consists of three stages:

Input Data → Preprocessing → Variance Filtering → Survival Model

### Stage 1: Column Transformation

**Numeric Features**:

Pipeline([  
 SimpleImputer(strategy='median', add\_indicator=True),  
 StandardScaler()  
])

* **Imputation**: Missing values replaced with median
* **Missing indicators**: Binary flags for missingness (preserves information)
* **Scaling**: Zero mean, unit variance (required for Cox models)

**Categorical Features**:

Pipeline([  
 SimpleImputer(strategy='most\_frequent', add\_indicator=True),  
 OneHotEncoder(drop='first', handle\_unknown='ignore')  
])

* **Imputation**: Missing values replaced with mode
* **Missing indicators**: Binary flags for missingness
* **One-hot encoding**: drop='first' to avoid multicollinearity
* **Unknown handling**: Assigns all-zero vector for new categories

### Stage 2: Variance Threshold

VarianceThreshold(threshold=1e-12)

**Purpose**: Remove effectively constant features that cause:

* Singular covariance matrices
* LAPACK numerical errors
* Unstable coefficient estimates

**Rationale**: Features with variance < 1e-12 provide no discriminative information and destabilize matrix operations in Cox models.

### Stage 3: Survival Model

The fitted model from Stage 1-2 is passed to the survival estimator.

### Why This Design?

1. **Missing value handling**: Prevents NaN propagation while preserving missingness as a signal
2. **Numerical stability**: Scaling + variance filtering prevent matrix singularity
3. **Regularization compatibility**: Scaled features enable fair L1/L2 penalties
4. **Censoring-aware**: All stages preserve sample indices for proper y alignment

## Evaluation Metrics

### 1. Concordance Index (C-index)

**Definition**

Measures the proportion of all comparable pairs where predictions and outcomes are concordant:

C-index = P(risk\_i > risk\_j | time\_i < time\_j, event\_i = 1)

**Interpretation**:

* 0.5: Random predictions
* 1.0: Perfect discrimination
* >0.7: Generally considered good
* Analogous to AUC-ROC for survival data

**Implementation**

* Uses Harrell's C-index with IPCW (inverse probability of censoring weighting)
* Handles censored observations appropriately
* Primary metric for model comparison

### 2. Integrated Brier Score (IBS)

**Definition**

Measures prediction accuracy across time by averaging Brier scores:

BS(t) = (1/n) ∑ᵢ [S(t|Xᵢ) - I(Tᵢ > t)]² · W(Tᵢ, δᵢ, t)  
IBS = ∫ BS(t) dt / (t\_max - t\_min)

Where W is the IPCW weight accounting for censoring.

**Interpretation**:

* 0: Perfect calibration
* Lower is better
* Measures both discrimination and calibration
* Time-dependent metric

**Implementation**

* Evaluated at time horizons: 3, 6, 12, 18, 24 months
* Uses IPCW to handle censoring
* Secondary metric for model selection

### 3. Time-Dependent AUC

**Definition**

Extension of ROC AUC for time-varying predictions:

AUC(t) = P(risk\_i > risk\_j | Tᵢ = t, Tⱼ > t)

**Interpretation**:

* Discrimination at specific time point t
* 0.5: Random classifier
* 1.0: Perfect discrimination
* Evaluated at multiple time horizons

**Implementation**

* Uses cumulative/dynamic AUC definition
* Evaluated at CV time horizons
* Provides time-specific performance insights

## Cross-Validation Strategy

### Event-Balanced Stratified K-Fold

**Motivation**

Standard K-fold can create folds with:

* Imbalanced event rates (e.g., 5% vs 15%)
* Insufficient events for model fitting
* Biased performance estimates

**Implementation**

**Function**: `event\_balanced\_splitter()` in `validation.py`

StratifiedKFold(n\_splits=5, shuffle=True, random\_state=42)

**Stratification variable**: Event indicator (δ)

**Ensures**:

* Each fold has approximately equal proportion of events
* Both censored and event samples in train/test
* Representative folds for unbiased evaluation

### Fold-Aware Time Grid Constraints

**Problem**: Survival metrics require time points within the test set's observed range.

**Solution**: `default\_time\_grid()` with test set constraints:

t\_min = max(train\_5th\_percentile, test\_min + 0.1)  
t\_max = min(train\_95th\_percentile, test\_max - 0.1)

**Benefits**:

* Prevents extrapolation errors
* Avoids "all times must be within follow-up" errors
* Ensures valid metric computation

### Cross-Validation Workflow

For each model:

1. Create 5 event-balanced folds
2. For each fold:

* Fit preprocessing + model on training data
* Generate fold-aware time grid
* Predict on test fold
* Compute C-index, IBS, time-dependent AUC
* Save predictions to `artifacts/<model>/<model>\_fold{i}\_\*.npy`

1. Aggregate metrics across folds
2. Rank models by mean C-index (primary), IBS (secondary)

## Prediction Methodology

### Model Selection

**Criterion**: Highest mean C-index from cross-validation

**Process**:

1. Read `artifacts/model\_summary.csv`
2. Select top-ranked model (sorted by rank\_cindex, rank\_ibs)
3. Load most recent saved model from `models/`

### Survival Probability Predictions

**Point Estimates at Fixed Horizons**:

Given feature vector X and time point t:

Ŝ(t|X) = model.predict\_survival\_function(X, times=[t])

**Current horizons**: 3, 6, 12, 18, 24, 36 months

**Interpretation**:

* P(customer remains active beyond t months | features X)
* Higher probability = lower churn risk
* Used for threshold-based interventions (e.g., contact if Ŝ(12) < 0.7)

### Expected Survival Time

**Restricted Mean Survival Time (RMST)**:

E[T|X] = ∫₀ᵗᵐᵃˣ Ŝ(u|X) du

**Implementation**:

def calculate\_expected\_survival\_time(pipeline, X):  
 times = np.linspace(0, t\_max, 200)  
 surv\_probs = pipeline.predict\_survival\_function(X, times)  
 expected\_time = trapezoid(surv\_probs, times) # area under curve

**Where**:

* t\_max = maximum observed time in training data (auto-detected)
* 200 points for numerical integration (trapezoidal rule)

**Properties**:

* **Intuitive interpretation**: Average months until termination
* **Single summary metric**: Easier for business stakeholders than probability curves
* **Restricted**: Lower bound if survival curve doesn't reach 0
* **Higher value = lower risk**: Directly ranks customers by expected retention

**Use Cases**:

* Customer lifetime value estimation
* Prioritizing retention interventions
* Segmentation by risk level

### Output Format

Each prediction CSV contains per-account:

| Column | Description | Range |

| `account\_entities\_key` | Account identifier | - |

| `model\_name` | Best model used | - |

| `expected\_survival\_months` | RMST | [0, t\_max] |

| `survival\_prob\_3m` | P(survive > 3 months) | [0, 1] |

| `survival\_prob\_6m` | P(survive > 6 months) | [0, 1] |

| `survival\_prob\_12m` | P(survive > 12 months) | [0, 1] |

| `survival\_prob\_18m` | P(survive > 18 months) | [0, 1] |

| `survival\_prob\_24m` | P(survive > 24 months) | [0, 1] |

| `survival\_prob\_36m` | P(survive > 36 months) | [0, 1] |

## Implementation Architecture

### Module Structure

survival\_framework/  
├── data.py # Data loading, preprocessing pipelines  
├── models.py # Survival model wrappers  
├── validation.py # Cross-validation, evaluation  
├── metrics.py # Survival-specific metrics  
├── predict.py # Prediction generation  
├── train.py # Training orchestration  
├── tracking.py # MLflow experiment tracking  
└── utils.py # Helper utilities

### Design Principles

1. **Unified Interface**: All models inherit from `BaseSurvivalModel` with standard `fit()`, `predict\_survival\_function()`, `score()` methods
2. **Pipeline Composition**: sklearn Pipeline pattern for reproducible preprocessing + modeling
3. **Separation of Concerns**:

* `data.py`: Feature engineering
* `models.py`: Algorithmic logic
* `validation.py`: Evaluation protocol
* `train.py`: Workflow orchestration

1. **Type Safety**: Type hints on all function signatures
2. **Comprehensive Docstrings**: Google-style docstrings with Args, Returns, Examples
3. **Experiment Tracking**: MLflow integration for reproducibility

### Key Abstractions

**Structured Array for Survival Data**

y = np.array(  
 [(event\_1, time\_1), (event\_2, time\_2), ...],  
 dtype=[('event', bool), ('time', float)]  
)

**Rationale**:

* scikit-survival standard format
* Type-safe event/time pairing
* Prevents index misalignment

**Model Wrapper Pattern**

class BaseSurvivalModel:  
 def fit(self, X, y) -> self  
 def predict\_survival\_function(self, X, times) -> np.ndarray  
 def score(self, X, y) -> float

**Benefits**:

* Consistent API across libraries (scikit-survival, lifelines)
* Easy to add new models
* Enables pipeline composition

## References

### Theoretical Foundations

1. **Cox, D. R. (1972)**. "Regression Models and Life-Tables." \*Journal of the Royal Statistical Society\*, Series B, 34(2), 187-220.
2. **Klein, J. P., & Moeschberger, M. L. (2003)**. \*Survival Analysis: Techniques for Censored and Truncated Data\* (2nd ed.). Springer.
3. **Ishwaran, H., et al. (2008)**. "Random Survival Forests." \*The Annals of Applied Statistics\*, 2(3), 841-860.
4. **Royston, P., & Altman, D. G. (2013)**. "External validation of a Cox prognostic model: principles and methods." \*BMC Medical Research Methodology\*, 13, 33.

### Implementation References

1. **Pölsterl, S. (2020)**. "scikit-survival: A Library for Time-to-Event Analysis Built on Top of scikit-learn." \*Journal of Machine Learning Research\*, 21(212), 1-6.
2. **Davidson-Pilon, C. (2019)**. \*lifelines: survival analysis in Python\*. Journal of Open Source Software, 4(40), 1317.
3. **Harrell, F. E., et al. (1996)**. "Multivariable prognostic models: issues in developing models, evaluating assumptions and adequacy, and measuring and reducing errors." \*Statistics in Medicine\*, 15(4), 361-387.

## Changelog

### 2025-10-23 - Initial Version

* Documented survival analysis theory and fundamentals
* Described all five model implementations (Cox PH, Coxnet, Weibull AFT, GBSA, RSF)
* Detailed preprocessing pipeline with L2 regularization and variance filtering
* Explained evaluation metrics (C-index, IBS, time-dependent AUC)
* Documented event-balanced cross-validation strategy
* Added prediction methodology including RMST calculation
* Outlined implementation architecture and design principles

**Maintenance Note**: This document should be updated whenever:

* New models are added to the framework
* Preprocessing steps are modified
* Evaluation metrics are changed
* Prediction outputs are altered
* Theoretical assumptions are revised