Survival Analysis Pipeline

This notebook implements a survival analysis workflow using clinical and molecular data. We will preprocess the data, create enhanced features, train **Coxnet** and **Random Survival Forest** models, evaluate performance using **concordance index**, and generate predictions for the test set.

In [48]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sksurv.util import Surv
from sksurv.linear_model import CoxnetSurvivalAnalysis
from sksurv.ensemble import RandomSurvivalForest
from sksurv.metrics import concordance_index_ipcw, concordance_index_censored
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, PolynomialFeatures
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.pipeline import Pipeline
```

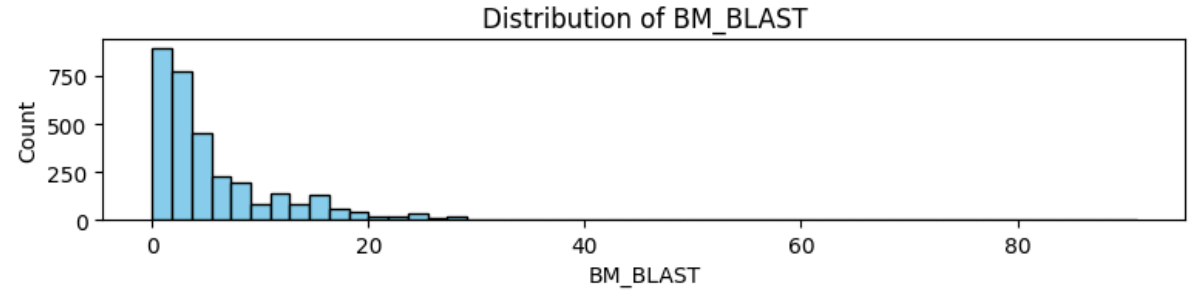
Load Data

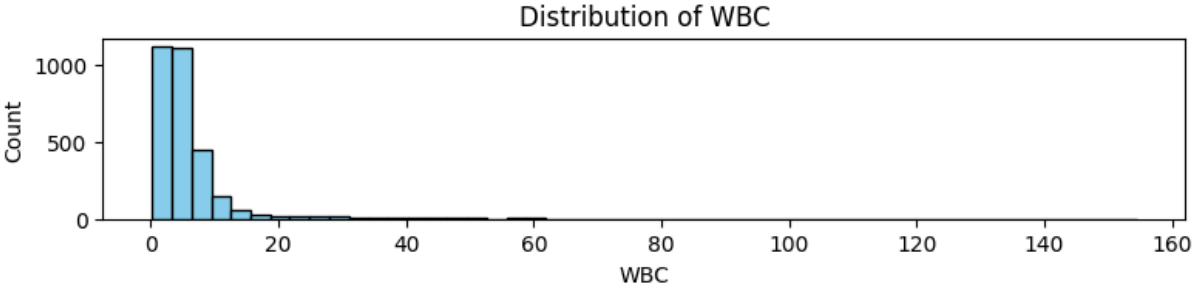
We load the clinical, molecular (MAF), and target datasets.
We also clean the target by converting `OS_YEARS` to numeric and `OS_STATUS` to boolean.

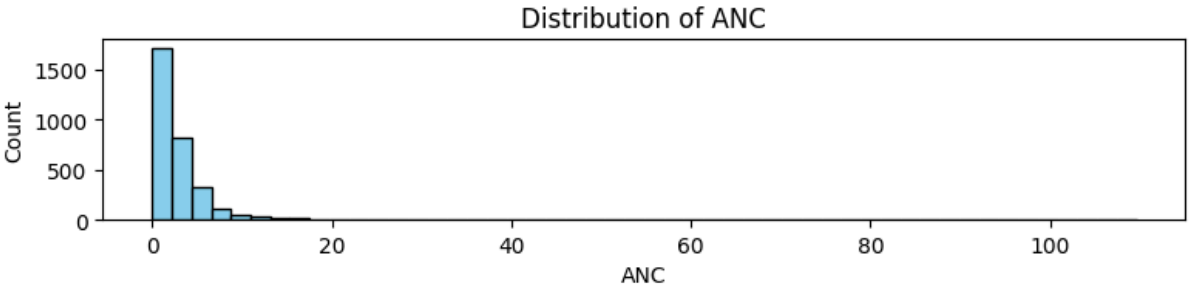
In [ ]:
```
df = pd.read_csv("./X_train/clinical_train.csv")
df_eval = pd.read_csv("./X_test/clinical_test.csv")
maf_df = pd.read_csv("./X_train/molecular_train.csv")
maf_eval = pd.read_csv("./X_test/molecular_test.csv")
y_df = pd.read_csv("./target_train.csv")

# Clean target
y_df.columns = y_df.columns.str.strip()
y_df.dropna(subset=['OS_YEARS','OS_STATUS'], inplace=True)
y_df['OS_YEARS'] = pd.to_numeric(y_df['OS_YEARS'], errors='coerce')
y_df['OS_STATUS'] = y_df['OS_STATUS'].astype(bool)
```

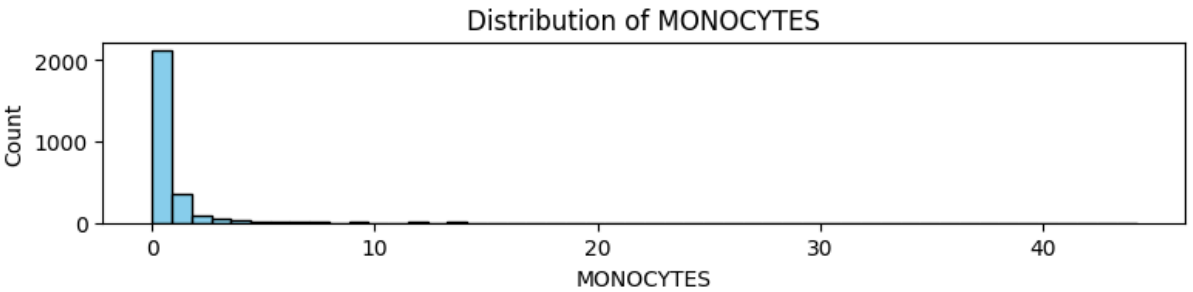
In [53]:
```
# Select numeric columns in your clinical data
numeric_cols = df.select_dtypes(include='number').columns.tolist()

# Plot distributions
plt.figure(figsize=(8, len(numeric_cols)*2))
for i, col in enumerate(numeric_cols, 1):
    plt.subplot(len(numeric_cols), 1, i)
    plt.hist(df[col].dropna(), bins=50, color='skyblue', edgecolor='black')
    plt.title(f'Distribution of {col}')
    plt.xlabel(col)
    plt.ylabel('Count')
    plt.tight_layout()
```

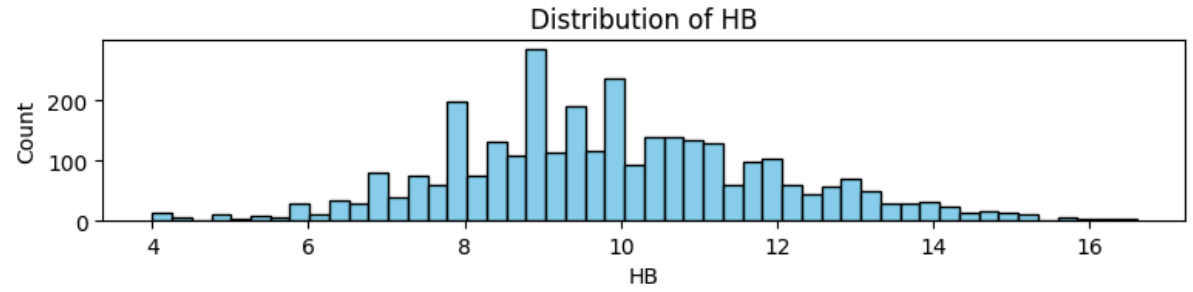
```
plt.show()
```

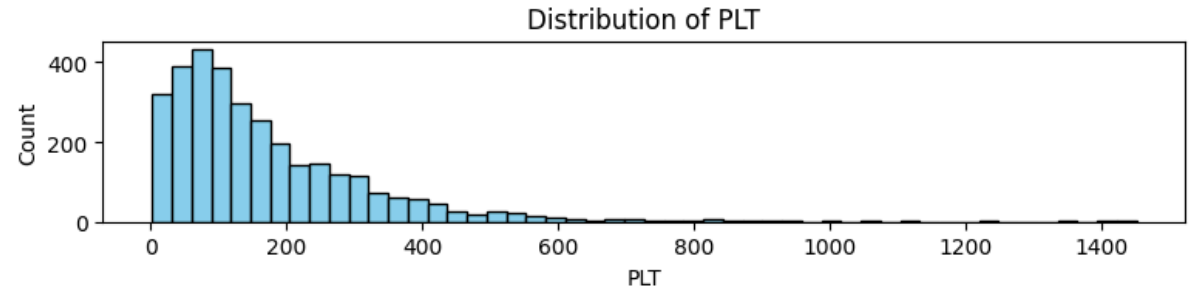

Distribution of BM_BLAST
Count
BM_BLAST


Distribution of WBC
Count
WBC


Distribution of ANC
Count
ANC


Distribution of MONOCYTES
Count
MONOCYTES


Distribution of HB
Count
HB


Distribution of PLT
Count
PLT

Preprocessing Function

This function performs:

- Adding mutation counts (`Nmut`)
- Biochemical feature transformations (log, sqrt, square)
- Feature ratios
- Cytogenetic feature extraction
- High-risk and epigenetic gene mutations
- Variant allele frequency (VAF) features
- Imputation and scaling

It can be used both for training (fit=True) and evaluation (fit=False).

In [ ]:
```
def preprocess(df_proc, maf_df, imputer=None, scaler=None, fit=False):
    if 'Nmut' in df_proc.columns:
        df_proc = df_proc.drop(columns=['Nmut'])
    tmp = maf_df.groupby('ID').size().reset_index(name='Nmut')
    df_proc = df_proc.merge(tmp, on='ID', how='left').fillna({'Nmut':0})

    for col in ['BM_BLAST','WBC','PLT','ANC','HB']:
        if col in df_proc.columns:
            df_proc[col+'_log'] = np.log1p(df_proc[col])
            df_proc[col+'_sqrt'] = np.sqrt(df_proc[col])
            df_proc[col+'_square'] = df_proc[col]**2

    basic_features = []
    for col in ['BM_BLAST','HB','PLT','WBC','ANC','MONOCYTES','Nmut']:
        if col in df_proc.columns:
            basic_features.append(col)
            for suffix in ['_log','_sqrt','_square']:
                feat = col + suffix
                if feat in df_proc.columns:
                    basic_features.append(feat)

    for col in ['WBC','HB','PLT','ANC','BM_BLAST','MONOCYTES']:
        if col not in df_proc.columns:
            df_proc[col] = 0

    df_proc['WBC_to_HB'] = df_proc['WBC'] / (df_proc['HB']+1e-5)
    df_proc['PLT_to_ANC'] = df_proc['PLT'] / (df_proc['ANC']+1e-5)
    df_proc['BLAST_to_WBC'] = df_proc['BM_BLAST'] / (df_proc['WBC']+1e-5)
    df_proc['ANC_to_MONOCYTES'] = df_proc['ANC'] / (df_proc['MONOCYTES']+1e-5)
    df_proc['PLT_to_HB'] = df_proc['PLT'] / (df_proc['HB']+1e-5)
    df_proc['BLAST_to_ANC'] = df_proc['BM_BLAST'] / (df_proc['ANC']+1e-5)
    df_proc['BLAST_to_PLT'] = df_proc['BM_BLAST'] / (df_proc['PLT']+1e-5)
    df_proc['MONO_to_WBC'] = df_proc['MONOCYTES'] / (df_proc['WBC']+1e-5)

    ratio_cols = ['WBC_to_HB','PLT_to_ANC','BLAST_to_WBC','ANC_to_MONOCYTES',
                  'PLT_to_HB','BLAST_to_ANC','BLAST_to_PLT','MONO_to_WBC']
    for col in ratio_cols:
```

```
        df_proc[col+'_log'] = np.log1p(df_proc[col])

    ratio_log_cols = [col+'_log' for col in ratio_cols]

    df_proc['CYTOGENETICS'] = df_proc.get('CYTOGENETICS','').fillna('')
    df_proc['MONOSOMY_7'] = df_proc['CYTOGENETICS'].str.contains(r'-7', na=False).a
    df_proc['TRISOMY_8'] = df_proc['CYTOGENETICS'].str.contains(r'\+8', na=False).a
    df_proc['COMPLEX_KARYO'] = df_proc['CYTOGENETICS'].str.count(',').ge(3).astype(
    df_proc['CYTO_SCORE'] = df_proc[['MONOSOMY_7','TRISOMY_8','COMPLEX_KARYO']].sum
    common_cytos = ['del5q','t(8;21)','inv(16)']
    for ab in common_cytos:
        df_proc[ab] = df_proc['CYTOGENETICS'].str.contains(ab, regex=False, na=Fals
    df_proc['MONO7_COMPLEX'] = df_proc['MONOSOMY_7'] * df_proc['COMPLEX_KARYO']

    cyto_features = ['MONOSOMY_7','TRISOMY_8','COMPLEX_KARYO','CYTO_SCORE'] + commo

    high_risk_genes = ['TP53','RUNX1','ASXL1']
    for gene in high_risk_genes:
        df_proc[gene+'_MUT'] = df_proc['ID'].map(
            maf_df.loc[maf_df['GENE'] == gene].groupby('ID').size()
        ).fillna(0)
    df_proc['HIGH_RISK_MUT'] = df_proc[[g+'_MUT' for g in high_risk_genes]].max(axi

    epigenetic_genes = ['ASXL1','TET2','DNMT3A']
    for g in epigenetic_genes:
        df_proc[g+'_MUT'] = df_proc['ID'].map(
            maf_df.loc[maf_df['GENE'] == g].groupby('ID').size()
        ).fillna(0)
    df_proc['EPIGENETIC_MUT'] = df_proc[[g+'_MUT' for g in epigenetic_genes]].sum(a

    mut_features = [g+'_MUT' for g in high_risk_genes] + ['HIGH_RISK_MUT','EPIGENET
    df_proc['TP53_RUNX1'] = df_proc['TP53_MUT'] * df_proc['RUNX1_MUT']
    df_proc['TP53_ASXL1'] = df_proc['TP53_MUT'] * df_proc['ASXL1_MUT']
    df_proc['RUNX1_ASXL1'] = df_proc['RUNX1_MUT'] * df_proc['ASXL1_MUT']
    mut_features += ['TP53_RUNX1','TP53_ASXL1','RUNX1_ASXL1']

    vaf_features = []
    if 'VAF' in maf_df.columns:
        for gene in high_risk_genes:
            vaf_gene = maf_df[maf_df['GENE'] == gene].groupby('ID')['VAF'].mean()
            df_proc[gene+'_MUT_VAF'] = df_proc['ID'].map(vaf_gene).fillna(0)
            vaf_features.append(gene+'_MUT_VAF')

        vaf_stats = maf_df.groupby('ID')['VAF'].agg(['max','mean','sum']).reset_ind
        vaf_stats.rename(columns={'max':'VAF_MAX','mean':'VAF_MEAN','sum':'VAF_SUM'
        df_proc = df_proc.merge(vaf_stats, on='ID', how='left').fillna({'VAF_MAX':0
        for col in ['VAF_MAX','VAF_MEAN','VAF_SUM']:
            df_proc[col+'_log'] = np.log1p(df_proc[col])
        vaf_features += [col for col in ['VAF_MAX','VAF_MEAN','VAF_SUM']] + [col+'_
    else:
        for col in ['VAF_MAX','VAF_MEAN','VAF_SUM']:
            df_proc[col] = 0
            df_proc[col+'_log'] = 0
            vaf_features += [col, col+'_log']

    features = basic_features + cyto_features + ratio_log_cols + mut_features + vaf
```

```
    df_features = df_proc[features].copy()

    if fit:
        imputer = SimpleImputer(strategy='median')
        df_features = pd.DataFrame(imputer.fit_transform(df_features), columns=df_f
        scaler = StandardScaler()
        df_features = pd.DataFrame(scaler.fit_transform(df_features), columns=df_fe
        return df_features, imputer, scaler
    else:
        df_features = pd.DataFrame(imputer.transform(df_features), columns=df_featu
        df_features = pd.DataFrame(scaler.transform(df_features), columns=df_featur
        return df_features
```

Preprocess Training and Evaluation Data

We apply the preprocessing function to both training and test datasets.
We also align the target data (`y`) and prepare the `Surv` object for survival analysis.

In [ ]:
```
X_train_final, imputer, scaler = preprocess(df.copy(), maf_df.copy(), fit=True)
X_eval_final = preprocess(df_eval.copy(), maf_eval.copy(), imputer=imputer, scaler=

df_2 = df.copy().merge(y_df[['ID','OS_STATUS','OS_YEARS']], on='ID', how='left').dr
df_2['OS_STATUS'] = df_2['OS_STATUS'].astype(bool)
y = Surv.from_dataframe('OS_STATUS','OS_YEARS',df_2)
X_train_final_aligned = X_train_final.loc[df_2.index].reset_index(drop=True)
```

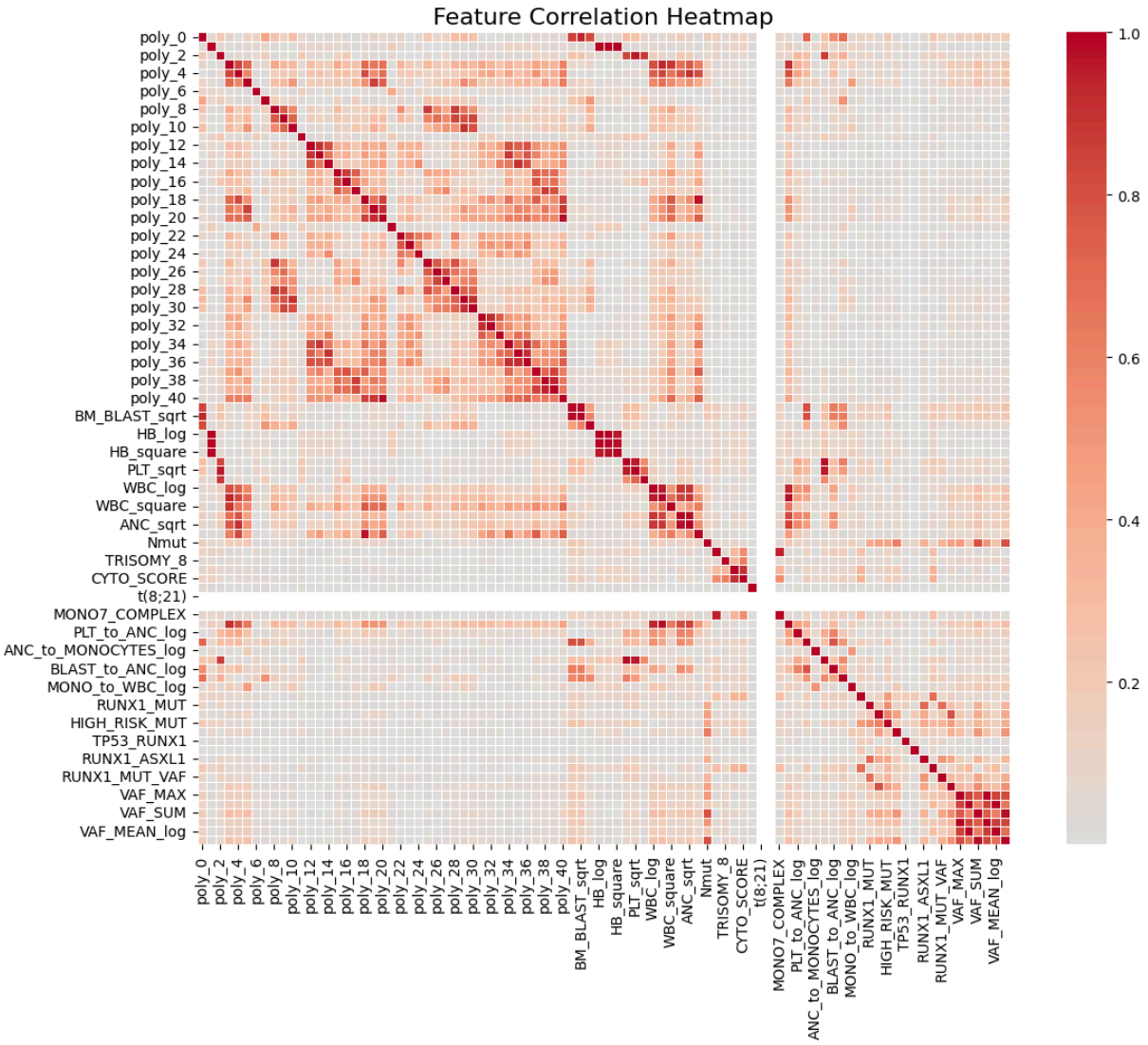
Polynomial Features

We enhance biochemical features (`BM_BLAST, HB, PLT, WBC, ANC, MONOCYTES`) using polynomial transformations.
This helps capture nonlinear relationships and interactions.

In [39]:
```
# Polynomial features
biochem_features = ['BM_BLAST','HB','PLT','WBC','ANC','MONOCYTES']
poly = PolynomialFeatures(degree=3, interaction_only=True, include_bias=False)
X_poly = poly.fit_transform(X_train_final_aligned[biochem_features])
X_rest = X_train_final_aligned.drop(columns=biochem_features).reset_index(drop=True
X_poly_df = pd.DataFrame(X_poly, columns=[f'poly_{i}' for i in range(X_poly.shape[1
X_train_enhanced = pd.concat([X_poly_df, X_rest], axis=1)

# Eval set
X_eval_poly = poly.transform(X_eval_final[biochem_features])
X_eval_rest = X_eval_final.drop(columns=biochem_features).reset_index(drop=True)
X_eval_poly_df = pd.DataFrame(X_eval_poly, columns=[f'poly_{i}' for i in range(X_ev
X_eval_enhanced = pd.concat([X_eval_poly_df, X_eval_rest], axis=1)
```

Correlation heatmap

In [40]:
```
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np

# Compute the absolute correlation matrix
corr_matrix = X_train_enhanced.corr().abs()

# Plot correlation heatmap
plt.figure(figsize=(14, 10))
sns.heatmap(corr_matrix, cmap='coolwarm', center=0, square=True, linewidths=0.5)
plt.title("Feature Correlation Heatmap", fontsize=16)
plt.show()

# Identify highly correlated features (corr > 0.9)
upper_tri = corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k=1).astype(bool))
to_drop = [col for col in upper_tri.columns if any(upper_tri[col] > 0.9)]

# Drop them from the dataset
X_train_filtered = X_train_enhanced.drop(columns=to_drop, errors='ignore')
X_eval_filtered = X_eval_enhanced.drop(columns=to_drop, errors='ignore')

print(f"\nDropped {len(to_drop)} highly correlated features (|corr| > 0.9):")
if to_drop:
    print(to_drop)
else:
    print("No features dropped.")
```




```
Dropped 20 highly correlated features (|corr| > 0.9):
['poly_20', 'poly_30', 'poly_32', 'poly_36', 'poly_39', 'poly_40', 'BM_BLAST_sqrt',
'HB_log', 'HB_sqrt', 'HB_square', 'PLT_sqrt', 'WBC_sqrt', 'ANC_sqrt', 'ANC_square',
'MONO7_COMPLEX', 'WBC_to_HB_log', 'PLT_to_HB_log', 'VAF_MAX_log', 'VAF_MEAN_log', 'V
AF_SUM_log']
```

1. Train/Test Split

We split the enhanced training data into training and validation sets.

2. Coxnet Model (Elastic Net Regularized Cox Proportional Hazards)

We use the **Cox Proportional Hazards model**:

$$h(t \mid X) = h_0(t)\ \exp(\beta^T X)$$

- $h(t \mid X)$: hazard at time t given covariates X
- $h_0(t)$: baseline hazard
- β: regression coefficients

Coxnet applies **Elastic Net regularization** to the partial likelihood:

$$\hat{\beta} = \arg\min_{\beta} \left(-\ell(\beta) + \lambda \left(\alpha \|\beta\|_1 + (1-\alpha)\frac{1}{2}\|\beta\|_2^2 \right) \right)$$

- $\ell(\beta)$: partial log-likelihood of the Cox model
- λ: overall regularization strength
- α (l1_ratio): balance between L1 (Lasso) and L2 (Ridge) penalties

We optimize over **l1_ratio (α)** and **alpha_min_ratio (controls λ grid)**.

Evaluation metric: Concordance index (C-index), measuring how well predicted risk rankings agree with observed survival times.

In [ ]:
```
# Split filtered features into train/test
X_train_final, X_test_final, y_train_final, y_test_final = train_test_split(
    X_train_filtered, y, test_size=0.3, random_state=42
)

# COXNET PIPELINE
pipe = Pipeline([
    ('coxnet', CoxnetSurvivalAnalysis(max_iter=5000))
])

param_grid = {
    'coxnet__l1_ratio': [0.1, 0.3, 0.5, 0.7],
    'coxnet__alpha_min_ratio': [0.001, 0.01, 0.1]
}

# Concordance index scorer
def cindex_scorer(estimator, X, y):
    pred = estimator.predict(X)
    return concordance_index_censored(y['OS_STATUS'], y['OS_YEARS'], pred)[0]

# Grid Search
grid_search = GridSearchCV(
    pipe,
    param_grid,
    cv=3,
    scoring=cindex_scorer,
    n_jobs=-1,
    verbose=1
)

grid_search.fit(X_train_final, y_train_final)

print(f"Best parameters: {grid_search.best_params_}")
best_coxnet = grid_search.best_estimator_

# Predictions & Concordance Index
train_preds = best_coxnet.predict(X_train_final)
test_preds = best_coxnet.predict(X_test_final)

cindex_train = concordance_index_censored(y_train_final['OS_STATUS'], y_train_final
```

```
cindex_test = concordance_index_censored(y_test_final['OS_STATUS'], y_test_final['O

print(f"Coxnet Concordance Index | Train: {cindex_train:.3f}, Test: {cindex_test:.3
```

```
Fitting 3 folds for each of 12 candidates, totalling 36 fits
Best parameters: {'coxnet__alpha_min_ratio': 0.01, 'coxnet__l1_ratio': 0.1}
Coxnet Concordance Index | Train: 0.740, Test: 0.733
```

Random Survival Forest (RSF)

The **Random Survival Forest (RSF)** builds an ensemble of survival trees and aggregates their survival estimates.

The ensemble survival function is:

$$\hat{S}(t \mid X) = \frac{1}{B} \sum_{b=1}^{B} \hat{S}_b(t \mid X)$$

- B: number of trees
- $\hat{S}_b(t \mid X)$: survival curve estimated by tree b

Splitting in each tree commonly uses a **log-rank statistic** to find partitions that separate survival outcomes.

Advantages:

- Captures **nonlinearities and interactions**
- No proportional hazards assumption required
- Robust with high-dimensional or noisy data

Evaluation metric: Concordance index (C-index), comparing predicted risk rankings with observed survival outcomes.

In [44]:
```
rsf = RandomSurvivalForest(
    n_estimators=1500,
    min_samples_split=8,
    min_samples_leaf=4,
    max_features="sqrt",
    n_jobs=-1,
    random_state=42,
)

# Fit model
rsf.fit(X_train_final, y_train_final)

# Predict
train_preds_rsf = rsf.predict(X_train_final)
test_preds_rsf = rsf.predict(X_test_final)

# Compute concordance index
cindex_train_rsf = concordance_index_censored(
    y_train_final['OS_STATUS'], y_train_final['OS_YEARS'], train_preds_rsf
```

```
)[0]

cindex_test_rsf = concordance_index_censored(
    y_test_final['OS_STATUS'], y_test_final['OS_YEARS'], test_preds_rsf
)[0]

print(f"RSF Concordance Index | Train: {cindex_train_rsf:.3f}, Test: {cindex_test_r
```

```
RSF Concordance Index | Train: 0.894, Test: 0.731
```

Predictions on Test Set

We use the best Coxnet model to predict risk scores for the evaluation dataset and save the submission file.

In [47]:
```
# Predict on evaluation set using filtered features
prediction_on_eval = best_coxnet.predict(X_eval_filtered)

# Prepare submission
submission = pd.Series(prediction_on_eval, index=df_eval['ID'], name='risk_score')
submission.to_csv('./final_submission.csv', index=True)

print("Submission file saved as 'final_submission.csv'.")
```

```
Submission file saved as 'final_submission.csv'.
```