Prediction vs. Inference in Life Insurance Contracts

Mark Szekacs and Marcell Hajdú

Problem Overview

The task consists of two fundamentally different statistical problems:

- Task b: Prediction – predicting whether a contract is cancelled due to overdue fees (`Event_NP`)
- Task c: Inference – identifying factors influencing survival time of cancelled contracts

Data Preparation

The dataset is provided in two separate CSV files:

- `Dependent.csv` – dependent variables (`Event_NP`, `Time`)
- `Explanatory.csv` – explanatory variables related to contract and policyholder characteristics

These files are merged column-wise, as they correspond to the same set of contracts.

```
import pandas as pd

dep = pd.read_csv("data/Dependent.csv", sep=";", decimal=",")
exp = pd.read_csv("data/Explanatory.csv", sep=";", decimal=",")

df = pd.concat([dep, exp], axis=1)
df.head()
```

Task b:

Target Variable

- Event_NP = 1: contract cancelled due to overdue fees
- Event_NP = 0: otherwise

This variable is binary and defines a classification problem.

Methodology: Why This Is a Prediction Task

The goal of Task b is to predict future cancellations, not to interpret causal effects.

Therefore: - the dataset is split into training and test sets, - models are evaluated on out-of-sample performance, - and model comparison is based on predictive metrics.

Interpretability is secondary in this task.

Model Setup

We compare several standard machine learning methods suitable for tabular insurance data:

- Logistic Regression (baseline)
- Lasso Logistic Regression
- Random Forest
- Gradient Boosting
- Histogram-based Gradient Boosting

```
from sklearn.model_selection import train_test_split
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier,
GradientBoostingClassifier, HistGradientBoostingClassifier
from sklearn.metrics import roc_auc_score, f1_score

X = df.drop(columns=["Event_NP"])
y = df["Event_NP"].astype(int)

cat_cols = [c for c in X.columns if X[c].dtype == "object"]
num_cols = [c for c in X.columns if c not in cat_cols]

preprocess = ColumnTransformer([
    ("num", Pipeline([
        ("imp", SimpleImputer(strategy="median")),
        ("sc", StandardScaler())
    ]), num_cols),
    ("cat", Pipeline([
        ("imp", SimpleImputer(strategy="most_frequent")),
        ("oh", OneHotEncoder(handle_unknown="ignore"))
    ]), cat_cols)
])

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)
```

```
#| echo: false

models = {
    "Logistic": LogisticRegression(max_iter=2000,
    class_weight="balanced"),
    "Lasso Logistic": LogisticRegression(penalty="l1",
```

```
    solver="saga", max_iter=3000, class_weight="balanced"),
    "Random Forest": RandomForestClassifier(n_estimators=300,
    random_state=42, class_weight="balanced"),
    "Gradient Boosting":
    GradientBoostingClassifier(random_state=42),
    "Hist Gradient Boosting":
    HistGradientBoostingClassifier(random_state=42)
}

results = []

for name, model in models.items():
    pipe = Pipeline([("prep", preprocess), ("model", model)])
    pipe.fit(X_train, y_train)

    proba = pipe.predict_proba(X_test)[:,1]
    auc = roc_auc_score(y_test, proba)
    f1 = f1_score(y_test, (proba >= 0.5).astype(int))

    results.append((name, auc, f1))

summary = pd.DataFrame(results,
columns=["Model", "ROC_AUC", "F1"])
summary.sort_values("ROC_AUC", ascending=False)
```

Interpretation of Results

Tree-based boosting models achieve the highest predictive performance, with ROC-AUC values close to 0.99 and strong F1-scores.

Logistic regression remains a strong baseline, indicating that the explanatory variables already contain substantial predictive information.

Differences between the top-performing models are small, suggesting robustness of the results.

Conclusion for Task b

Task b is successfully addressed as a prediction problem.

- Multiple machine learning models achieve strong out-of-sample performance.
- Tree-based boosting methods perform best in terms of ROC-AUC and F1-score.
- Logistic regression provides a competitive and interpretable benchmark.
- No extensive hyperparameter tuning was performed due to time constraints.

Task (c): Inference on Survival Time for Overdue Cancellations

Target and sample restriction

We focus on contracts cancelled due to overdue fees, i.e. we restrict the dataset to observations with *Event_NP = 1*.
The outcome of interest is the *survival time* Time (time until cancellation).

Why this is an inference task (not prediction)

The objective is to identify *significant determinants* of survival time and to justify a *model selection method*, rather than maximizing predictive accuracy. Therefore, we use an interpretable time-to-event model and report statistical significance (p-values).

Model choice: Weibull Accelerated Failure Time (AFT)

We estimate a *Weibull Accelerated Failure Time (AFT)* model on the restricted sample.
In an AFT model, covariates act multiplicatively on time:

- *Positive coefficients* imply *longer survival times* (slower cancellation),
- *Negative coefficients* imply *shorter survival times* (faster cancellation).

A useful interpretation is via exp(coef), which approximates the multiplicative change in survival time for a one-unit increase in the covariate (or relative to a baseline category for dummies).

Model selection

To avoid overfitting under time constraints, we first perform *Lasso screening* on log(Time) to reduce dimensionality, and then refit the final AFT model on the selected variables to obtain standard errors and p-values.

```
#| echo: false
#| warning: false
#| message: false
#| results: asis

import numpy as np
import pandas as pd

from lifelines import WeibullAFTFitter
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LassoCV


# -----------------------
# 1) Load + merge
# -----------------------
```

```
dep = pd.read_csv("Dependent (1).csv", sep=";", decimal=",")
exp = pd.read_csv("Explanatory (1).csv", sep=";", decimal=",")
df = pd.concat([dep, exp], axis=1)

# -----------------------
# 2) Filter: only overdue cancellations
# -----------------------
df["Event_NP"] = pd.to_numeric(df["Event_NP"], errors="coerce")
.astype(int)
df = df[df["Event_NP"] == 1].copy()

# Time must be positive
df["Time"] = pd.to_numeric(df["Time"], errors="coerce")
df = df[df["Time"].notna() & (df["Time"] > 0)].copy()

# Optional drop
if "UzletiKedv" in df.columns:
    df = df.drop(columns=["UzletiKedv"])

# -----------------------
# 3) Build X, y
# -----------------------
y = np.log(df["Time"].values)  # for screening only
X = df.drop(columns=["Event_NP", "Time"], errors="ignore")

cat_cols = [c for c in X.columns if X[c].dtype == "object"]
num_cols = [c for c in X.columns if c not in cat_cols]

preprocess = ColumnTransformer([
    ("num", Pipeline([
        ("imp", SimpleImputer(strategy="median")),
        ("sc", StandardScaler())
    ]), num_cols),
    ("cat", Pipeline([
        ("imp", SimpleImputer(strategy="most_frequent")),
        ("oh", OneHotEncoder(handle_unknown="ignore"))
    ]), cat_cols)
])

# -----------------------
# 4) Fast screening with Lasso on log(Time)
# -----------------------
Xmat = preprocess.fit_transform(X)

# feature names for interpretability
```

```
feature_names = list(num_cols)
if len(cat_cols) > 0:
    oh = preprocess.named_transformers_["cat"].named_steps["oh"]
    feature_names += list(oh.get_feature_names_out(cat_cols))

lasso = LassoCV(cv=5, random_state=42, n_alphas=40, max_iter=20000)
lasso.fit(Xmat, y)

coef = pd.Series(lasso.coef_, index=feature_names)
selected = coef[coef.abs() > 1e-8].index.tolist()

print("\nSelected variables (Lasso screening):")
for v in selected[:50]:
    print(" -", v)
print(f"\nTotal selected: {len(selected)}")

# -----------------------
# 5) Final inference: Weibull AFT on selected vars
# -----------------------
# Build a clean numeric dataframe for lifelines:
df_model = df[["Time"]].copy()

# Add selected features
# For numeric features: directly from df
for col in num_cols:
    if col in selected:
        df_model[col] = pd.to_numeric(df[col], errors="coerce")

# For categorical: add dummies and take selected
if len(cat_cols) > 0:
    dummies = pd.get_dummies(df[cat_cols], drop_first=True)
    for col in dummies.columns:
        # lifelines uses exact column names, so match the
        # same naming as pandas get_dummies
        if col in selected:
            df_model[col] = dummies[col]

df_model = df_model.dropna().reset_index(drop=True)

aft = WeibullAFTFitter()
aft.fit(df_model, duration_col="Time")

out = aft.summary.reset_index()

print("DEBUG columns:", list(out.columns)) # ezt 1x nézd meg
```

```
# Case 1: lifelines MultiIndex -> columns like
# ['param', 'covariate', 'coef', 'se(coef)', 'p', ...]
if "param" in out.columns and "covariate" in out.columns:
    out = out[out["param"] == "lambda_"].copy()
    out["variable"] = out["covariate"].astype(str)

# Case 2: single index: first column contains terms like 'lambda_XYZ'
else:
    out = out.rename(columns={out.columns[0]: "term"})
    out = out[out["term"].astype(str).
    str.startswith("lambda_")].copy()
    out["variable"] = out["term"].astype(str)
    .str.replace("lambda_", "", regex=False)

# Ensure p column name
if "p" not in out.columns:
    p_candidates = [c for c in out.columns if c.lower() in
    ["p", "pvalue", "p-value", "p_value"]]
if len(p_candidates) == 1:
    out = out.rename(columns={p_candidates[0]: "p"})
else:
    raise ValueError(f"Could not find p-value column.
    Columns: {list(out.columns)}")

out = out.sort_values("p")

print("\nTop determinants by p-value:")
print(out[["variable", "coef", "se(coef)", "p"]].
head(20).to_string(index=False))

sig = out[out["p"] < 0.05][["variable", "coef", "se(coef)", "p"]]
sig.sort_values("p").head(20)
sig

print(sig.to_latex(index=False))
```

Significant determinants of survival time (Weibull AFT)

Significant determinants of survival time (Weibull AFT)

Variable	Coefficient	SE	p-value
Intercept	2.707	0.102	$<$ 1e-154
INTERVENTIALO_ELTERES	-0.287	0.038	4.68e-14
OTHER_LIFE	0.252	0.040	3.44e-10

Variable	Coefficient	SE	p-value
PAYMENT_METHOD_csekkes	-0.182	0.034	6.98e-08
TOROLT_LIFE	-0.239	0.047	2.76e-07
EUG_KIEG	-0.170	0.037	3.13e-06
ANNUAL_PREMIUM	-0.000	0.000	9.38e-06
REGIO_EszakAlfold	-0.203	0.051	7.44e-05
KEZD_BOSSZEG	0.000	0.000	6.83e-04
TOROLT_NON_LIFE	0.027	0.008	9.28e-04
HITELFEDEZETI	0.146	0.046	1.40e-03
KOZLEKEDESI_DIJ	0.000	0.000	2.21e-03
CRIT_ILLNESS_DIJ	0.000	0.000	2.33e-03
TARTAM_EV	0.011	0.004	1.28e-02
KOZLEKEDESI	-0.127	0.055	2.00e-02

Evaluation of Results

The estimation results provide clear evidence that the timing of contract cancellation due to overdue fees is systematically related to both contract characteristics and customer attributes.

Several variables exhibit statistically significant effects on survival time. In particular, *negative coefficients* (e.g. INTERVENTIALO_ELTERES, PAYMENT_METHOD_csekkes, TOROLT_LIFE, EUG_KIEG) are associated with *shorter survival times*, indicating faster cancellation once overdue fees occur. These effects suggest that administrative frictions, payment methods, and certain contract structures accelerate termination.

Conversely, *positive coefficients* (e.g. OTHER_LIFE, HITELFEDEZETI, and selected premium components such as KOZLEKEDESI_DIJ and CRIT_ILLNESS_DIJ) are associated with *longer survival times*, implying slower cancellation. This indicates that contracts with broader coverage or additional premium components tend to remain active longer even after payment issues arise.

Overall, the results highlight the importance of *contract composition and payment-related characteristics* in explaining the duration until cancellation. The consistency of coefficient signs and the presence of multiple highly significant effects suggest that the findings are robust and economically meaningful, despite the relatively simple model structure and limited tuning due to time constraints.