Notes for ISL Chapter 11: Survival Analysis and Censored Data

Justin Burruss

2026-04-13

Background

The Python code and notes below are for the *Introduction to Statistical Learning* (ISL) study group. This is to try out some of the concepts from Chapter 11 using the `BrainCancer` data set. The document was created in RMarkdown with the Python code running via the `reticulate` library plus a little LaTeX.

Our ground rule: when implementing concepts from the chapter, just use basic Python + Pandas + NumPy. It's OK to use more when visualizing or evaluating results.

Loading the data set

The Pandas library ("Panda" as in "**Pan**el **da**ta") in conjunction with `openpyxl` makes it easy to load the Excel file.

```
import openpyxl
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# read Excel
df = pd.read_excel(r"E:\docs\Classes\ISL\S1Dataset.XLSX", sheet_name="data")
```

We can use the `.info()` method to check the column names and that we have the same record count as described in the text (88 patients).

```
df.info()
```

```
## <class 'pandas.core.frame.DataFrame'>
## RangeIndex: 88 entries, 0 to 87
## Data columns (total 9 columns):
##  #   Column               Non-Null Count  Dtype
## ---  ------               --------------  -----
##  0   ID                   88 non-null     int64
##  1   Sex                  88 non-null     int64
##  2   Diagnosis            87 non-null     float64
##  3   Location             88 non-null     int64
##  4   KI                   88 non-null     int64
##  5   GTV                  88 non-null     float64
##  6   Stereotactic methods  88 non-null     int64
```

```
##  7   status                88 non-null     int64
##  8   OS                    88 non-null     float64
## dtypes: float64(3), int64(6)
## memory usage: 6.3 KB
```

The "OS" name here indicates Overall Survival, as opposed to say Progression-Free Survival (PFS). We're expecting units of months per the book.

Survival function

The text gives us

$$\hat{S}(d_k) = \prod_{j=1}^{k} (\frac{r_j - q_j}{r_j})$$

where r_j is the number at risk at step k and q_j is the number of deaths at step k. Let's make just one simplification before we code this up.

$$\begin{aligned} \hat{S}(d_k) &= \prod_{j=1}^{k} (\frac{r_j - q_j}{r_j}) \\ &= \prod_{j=1}^{k} (\frac{r_j}{r_j} - \frac{q_j}{r_j}) && \text{decompose} \\ &= \prod_{j=1}^{k} (1 - \frac{q_j}{r_j}) && \frac{r_j}{r_j} = 1 \end{aligned}$$

This should not underflow given the small row count and percentages involved—no need to rewrite as sum of logs. We'll implement this using the Pandas built-in `.cumprod()`.

Building survival curves

To build a Kaplan–Meier survival curve we'll want to count deaths at all K of the d_k death times. A convenient way to structure this is as a Pandas DataFrame.

Counting events at each time

Let's build a DataFrame having OS as the index: each row will have OS and, for that OS, the number of each type of event. We have just the two events: deaths and censorings. At a minimum we need all K of the d_k death times, and we may as well save censorings as well. A combination of `.groupby()` and `.size()` will give us the counts at each even time. Then we can use `.unstack()` to take us from long to wide, just like `dcast` with `data.table`.

```
pd.DataFrame({"DeathCnt": 0, "CensorCnt": 0}, index=[0]),
  df.groupby(["OS", "status"]) \
    .size() \
    .unstack(fill_value=0) \
    .rename(columns={1: "DeathCnt", 0: "CensorCnt"})
```

The only subtlety is that we'll want to start at zero. One approach is to use `pd.concat` to prepend a zero entry, like this:

```
pd.concat([
        pd.DataFrame({"DeathCnt": 0, "CensorCnt": 0}, index=[0]),
        df.groupby(["OS", "status"]) \
          .size() \
          .unstack(fill_value=0) \
          .rename(columns={1: "DeathCnt", 0: "CensorCnt"})
    ])
```

We can't quite do "sum(X) over (partition by Y order by Z desc)" directly, but it's still very straightforward to use `.cumsum()` on reversed events. As mentioned, Pandas has a built-in cumulative product method: `.cumprod()`. Let's put these all together in a function.

```
def build_ev(df):
    """Return a Kaplan_Meier Survival DataFrame"""

    # structure a DataFrame with Overall Survival (OS) as the index and columns
    # to indicate the kind and number of events at each OS point in time.
    # note that unstack takes us from long to wide, like data.table dcast
    ev = pd.concat([
            pd.DataFrame({"DeathCnt": 0, "CensorCnt": 0}, index=[0]),
            df.groupby(["OS", "status"]) \
            .size() \
            .unstack(fill_value=0) \
            .rename(columns={1: "DeathCnt", 0: "CensorCnt"})
        ])
    ev.index.name = "Months" # OS is in Months

    # compute at-risk counts via reverse cumulative sum; the (::-1) lets us
    # reverse to accumulate from largest time to smallest (i.e., "desc window")
    ev["AtRiskCnt"] = ev.loc[::-1].cumsum()[::-1].sum(axis=1)

    # equation 11.3: cumprod((r_j - q_j)/r_j) rewrite:
    # (r_j - q_j)/r_j) = r_j / r_j - q_j / r_j = 1 - q_j / r_j
    # term is 1 at each censoring-only time, so OK to include in cumprod()
    ev["S(t)"] = (1 - ev["DeathCnt"] / ev["AtRiskCnt"]).cumprod()

    return ev

ev = build_ev(df)
```

We should acknowledge that by doing it this way we're multiplying by 1 at time zero and at each event time having censorings only, whereas we only need to multiply at each of the K death times (i.e., where the term is non-zero). It's more convenient to include all events and the zero row in the one-liner as we've done here.

Validating

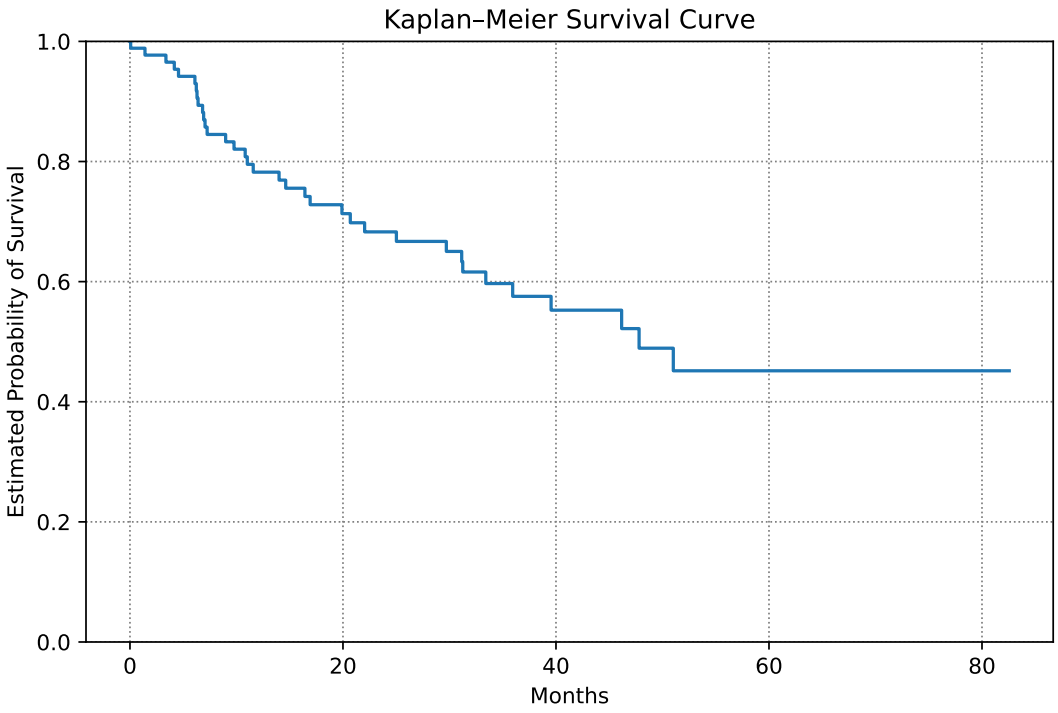
Per the book the "estimated probability of survival past 20 months is 71%"—did we get the same result? We can use `.asof()` to look up our answer.

```
print(f"S(20) = {100 * ev['S(t)'].asof(20):.1f}%")
```

```
## S(20) = 71.3%
```

We also get 71%. So far, so good. Now let's plot the result using `step()` and compare with Figure 11.2 from the text.

```
plt.figure(figsize=(8, 5))
plt.step(ev.index, ev["S(t)"], where="post")
plt.xlabel(ev.index.name)
plt.ylabel("Estimated Probability of Survival")
plt.title("Kaplan-Meier Survival Curve")
plt.ylim(0, 1);
plt.grid(visible=True, color='grey', linestyle=':')
plt.show()
```

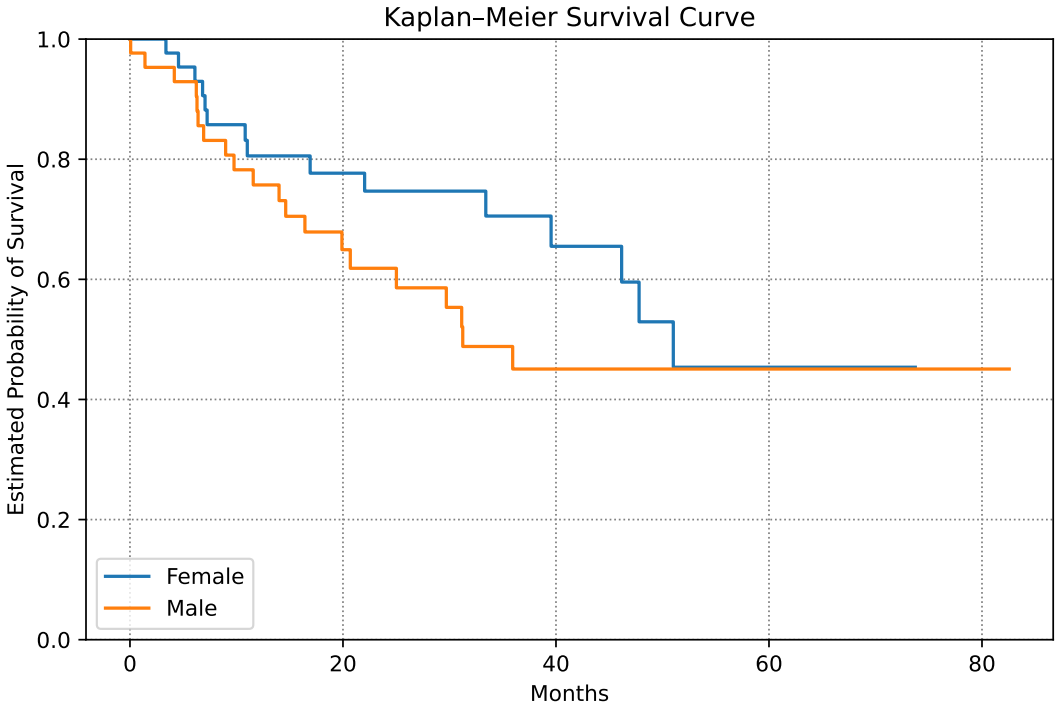


The curve matches what shows in the book. Finally, let's try to recreate Figure 11.2 by creating two event DataFrames.

```
# separate: Male=1, Female=0
ev_f = build_ev(df[df["Sex"] == 0])
ev_m = build_ev(df[df["Sex"] == 1])

# recreate figure 11.2
plt.figure(figsize=(8, 5))
plt.step(ev_f.index, ev_f["S(t)"], where="post", label="Female")
plt.step(ev_m.index, ev_m["S(t)"], where="post", label="Male")
plt.xlabel(ev.index.name)
plt.ylabel("Estimated Probability of Survival")
plt.title("Kaplan-Meier Survival Curve")
plt.ylim(0, 1);
plt.grid(visible=True, color='grey', linestyle=':')
```

```
plt.legend(loc="lower left")
plt.show()
```



Everything looks to match the text. Ultimately there's wasn't much to it: use `.groupby()` and `.size()` to find the counts at distinct OS times, use `.cumsum()` on reversed events to tally the at-risk counts r_j at each time, and apply the built-in cumulative product method `.cumprod()` to find $\hat{S}(t)$ at each step.

Proportional Hazards Model

We'll need to filter out patients with incomplete data. Let's save patients having complete data into a new DataFrame "p" for "patients", sorting by `Time` with deaths coming before censorings. For easier reading we will put things in the same order as in Table 11.2 of the book.

```
c = df.dropna()
p = pd.concat(
    [
        c["Sex"].eq(1).rename("IsMale"),
        pd.get_dummies(c["Diagnosis"], drop_first=True)
          .rename(columns={
              1.0: "IsDiagLG",
              2.0: "IsDiagHG",
              3.0: "IsDiagOth"
          }),
        c["Location"].eq(1).rename("IsLocSupr"),
        c[["KI", "GTV"]],  # keep as DataFrame
        c["Stereotactic methods"].eq(1).rename("IsSRT"),
```

```
        c["status"].rename("Event"),
        c["OS"].rename("Time")
    ],
    axis=1
).sort_values(by=["Time", "Event"], ascending=[True, False])
```

One thing to check before we move on to partial likelihood. On page 481 the book mentions that we've assumed "there are no tied failure times", so we ought to check if that's really the case with our data. We can use `.all()` to check: it will return `True` only if all values are true.

```
print((p[p["Event"] == 1]["Time"].value_counts() == 1).all())
```

```
## True
```

No ties, so it should be straightforward to use equation 11.16 to code up partial likelihood. As the book notes, no closed form solution is available for partial likelihood, so we'll code up an equivalent function that we can minimize in a straightforward way.

Negative Log Partial Likelihood

Equation 11.16 of the text gives us a formula for partial likelihood, which if we rewrite using vectors and take the log gets us to *log partial likelihood*:

$$
\begin{aligned}
PL(\beta) &= \prod_{i:\delta_i=1} \frac{\exp(\sum_{j=1}^{p} x_{ij}\beta_j)}{\sum_{i':y_{i'}\geq y_i} \exp(\sum_{j=1}^{p} x_{i'j}\beta_j)} && \text{given in eq 11.16} \\
&= \prod_{i:\delta_i=1} \frac{\exp(X_i \cdot \beta)}{\sum_{i':y_{i'}\geq y_i} \exp(X_{i'} \cdot \beta)} && \text{rewrite as vector} \\
\log(PL(\beta)) &= \log\Big(\prod_{i:\delta_i=1} \frac{\exp(X_i \cdot \beta)}{\sum_{i':y_{i'}\geq y_i} \exp(X_{i'} \cdot \beta)} \Big) && \text{log of both sides} \\
&= \sum_{i:\delta_i=1} \Big[\log(\exp(X_i \cdot \beta)) - \log(\sum_{i':y_{i'}\geq y_i} \exp(X_{i'} \cdot \beta)) \Big] && \text{product rule of logarithms} \\
&= \sum_{i:\delta_i=1} \Big[X_i \cdot \beta - \log(\sum_{i':y_{i'}\geq y_i} \exp(X_{i'} \cdot \beta)) \Big] && \log(e^a) = a
\end{aligned}
$$

Then our final *negative log partial likelihood* as:

$$
NLPL(\beta) = \sum_{i:\delta_i=1} \Big[X_i \cdot \beta - \log(\sum_{i':y_{i'}\geq y_i} \exp(X_{i'} \cdot \beta)) \Big]
$$

Now to code it up. We can use variable name `Beta` for our 1-dimensional vector β, `X` for the full 2-dimensional X. Let's use `E` for "events" and pass that as an input, no need to write up an indicator function. Just as we did before with the event table, we can get a reversed cumulative sum by using `.cumsum()` on reversed elements then reversing the result.

```
def nlpl(Beta, X, E):
    """Return Negative Log Partial Likelihood given df and candidate beta
       assuming no tied death times."""

    X_Beta = X @ Beta
    exp_X_Beta = np.exp(X_Beta)

    at_risk = np.cumsum(exp_X_Beta[::-1])[::-1]

    lpl = (X_Beta[E == 1] - np.log(at_risk[E == 1])).sum()

    return -lpl
```

Another way to do that reverse cumulative sum would have been to use `np.flip`, like this:

```
at_risk = np.flip(np.cumsum(np.flip(exp_X_Beta)))
```

Now to structure our inputs. Our input `X` contains all the patient variables other than `Time` and `Event`. For `Beta` we have one element per column of `X`. Then as discussed, we'll load our events into `E`.

```
X = p.drop(columns=["Time", "Event"]).astype(float).values
Beta = np.zeros(X.shape[1])
E = p["Event"].values
```

Validating

We'll use `minimize` from `scipy.optimize` to help validate that we've coded `nlpl()` correctly.

```
from scipy.optimize import minimize
res = minimize(nlpl, Beta, args=(X, E))

book_coeff = [0.18, 0.92, 2.15, 0.89, 0.44, -0.05, 0.03, 0.18] # page 482

comparison = pd.DataFrame({'Coeff (book)': book_coeff,
                           'Coeff (hand-code)': res.x},
                          index=p.drop(columns=["Time", "Event"]).columns)

print(comparison.round(2))
```

```
##            Coeff (book)  Coeff (hand-code)
## IsMale             0.18               0.18
## IsDiagLG           0.92               0.92
## IsDiagHG           2.15               2.15
## IsDiagOth          0.89               0.89
## IsLocSupr          0.44               0.44
## KI                -0.05              -0.05
## GTV                0.03               0.03
## IsSRT              0.18               0.18
```

Our results tie perfectly with Table 11.2 from the text.