Time-To-Event Processing in the WDMOC

Time-to-event (TTE) sampling is accomplished through one of two processes.

1. Sampled from a transition probability

   Some TTE parameters are encoded as a mean (x) and a standard deviation (y) of a transition probability. In those cases, a random sample is drawn from an exponential distribution based on those values using the method of moments. The Python code for this process is as follows:

```
# Step 1: generate random estimate of the transition probability
   x = self.mean
   y = self.se
   bdist_alpha = x*((x*(1-x)/y**2) - 1)
   bdist_beta = (1-x)*(x/y**2*(1-x) - 1)
   est_tp = numpy.random.beta(bdist_alpha, bdist_beta)

# Step 2: generate random draw from exponential distribution
   lmbd = -(math.log(1.0 - est_tp)/365.0)
   beta = 1/lmbd
   samp_value = numpy.random.exponential(beta)
   return samp_value
```

2. Sampled from a parametric (Weibull) distribution

   Other TTE parameters are derived from a generalized linear model (GLM) regression with a Weibull link function. The output of the GLM is an intercept β_0 with coefficients (β_1 ... β_n). Each coefficient corresponds with values (X_1 … X_n) stored on the entity (the entity's age, sex, smoking status, treatment type, etc.). The GLM also outputs a Sigma value (σ) that corresponds to the shape of the distribution.

   Sampling values from the empirical distribution described by the GLM function (based on an entity's individual characteristics) can be accomplished through the use of the function `numpy.random.weibull(self.shape)*self.scale` where:

   Shape = $1/\sigma$

   Scale = $\exp(\beta_0 + X_1\beta_1 + \ldots + X_n\beta_n)$

Both of these methods produce a TTE estimate t, expressed in days.

Competing Risks

The GLM-based analysis allowed the WDMOC to account for competing risks – the risk of death vs. recurrence following primary treatment for cancer, and the similar risk of death vs. recurrence following recurrence treatment. This was accomplished in three steps:

1. Estimate the time to the next event through parametric sampling

   The time to the next event (*t*) is estimated using the parametric sampling process described above. The distribution coefficients describing intermediate event (in this case, recurrence) are used.

2. Estimate the probability of each risk occurring at that time

   The probability of an event occurring at a given time is described by the cumulative distribution function (CDF) of the parametric distribution for that event, based on the shape and scale of the parametric function and the value of *t* from Step 1:

   ```
   prob_event = 1 - numpy.math.exp(-(t/self.scale)**self.shape)
   ```

   Probabilities for each event (i.e., recurrence or death following treatment) are generated in this way.

3. Determine which event occurs, based on relative probability

   Once probabilities for competing events have been estimated, the relative probability is compared to a randomly-sampled value from a uniform distribution:

   ```
   event_prob = prob2/prob1
   self.probEst = random.random()
   if probEst < event_prob:
       event_type = 2
   elif probEst >= event_prob:
       event_type = 1
   ```

   If the relative probability is less than or equal to the randomly-sampled probability (i.e., event type = 1), then the intermediate event (recurrence) occurs at time *t*. Otherwise, the competing event (death) occurs at time *t*.

The code governing the parametric sampling and competing risk functions follows:

Glb_GenTime – A Function to Generate Parametric Sampling of TTE Values

```
class GenTime:
    def __init__(self, estimates, regcoeffs):
        self._estimates = estimates
        self._regcoeffs = regcoeffs

    def readVal(self, entity, param):
        # Is the parameter being estimated contained within the Excel sheet?
        if param in self._regcoeffs:

            # The sum of the coefficients starts at zero
            coeff = 0

            # For a given factor of a parameter within the Excel sheet
            for factor in self._regcoeffs[param].keys():

                # Identify the intercept
                if factor == 'Intercept':
                    Intercept = self._regcoeffs[param]['Intercept']['mean']

                # Identify the shape parameter from the output
                elif factor == 'Sigma':
                    Sigma = self._regcoeffs[param]['Sigma']['mean']

                # Identify values for all other coefficients
                elif factor in entity.__dict__.keys():
                    value = getattr(entity, factor)

                    if self._regcoeffs[param][factor]['vartype'] == 2:
                        coeff += self._regcoeffs[param][factor]['mean'] * value
                    else:
                        coeff += self._regcoeffs[param][factor][value]['mean']

            # Produce an estimate of time from the regression
            mu = Intercept + coeff
            shape = 1/Sigma
            scale = math.exp(mu)

            self.mu = mu
            self.shape = shape
            self.scale = scale

    # Randomly sample an event time for the entity from a Weibull distribution
    def estTime(self):
        estimate_time = numpy.random.weibull(self.shape)*self.scale
        return estimate_time

    # Estimate the probability (CDF) of being alive at a given time
    def estProb(self, time):
        estimate_probability = numpy.math.exp(-(time/self.scale)**self.shape)
        return estimate_probability
```

Glb_CompTime – A Function to Evaluate Competing Risks

```
class CompTime:
    def __init__(self, estimates, regcoeffs):
        self._estimates = estimates
        self._regcoeffs = regcoeffs
        self.probEst = random.random()

    def Process(self, entity, tte1, tte2):
        # Draw two survival functions for the entity
        event1 = GenTime(self._estimates, self._regcoeffs)
        event2 = GenTime(self._estimates, self._regcoeffs)
        # Any event
        event1.readVal(entity, str(tte1))
        # Competing event
        event2.readVal(entity, str(tte2))

        # 1 - Draw random value for time to next event
        event_time = event1.estTime()
        # 2 - Estimate probability of that value occurring within first event
        prob1 = 1 - event1.estProb(event_time)
        # 3 - Estimate probability of that value occurring within second event
        prob2 = 1 - event2.estProb(event_time)
        # 4 - Calculate relative probability that event is the competing event
        event_prob = prob2/prob1
        # 5 - Evaluate relative probability against random probability
        if self.probEst < event_prob:
            event_type = 2
        elif self.probEst >= event_prob:
            event_type = 1

        return (event_time, event_type)
```