Three paradigms for modeling: curve-fitting, probabilistic models and uncertainty quantification, illustrated using the MMI data

By Charles Zheng, using data from Keren et al. (2020)

Importing the data

In [1]:
```
import numpy as np
from matplotlib import pyplot as plt
%matplotlib inline
import pandas as pd
```

In [2]:
```
# Import the data
ts = pd.read_csv('../mturk.csv', index_col=0)
```

In [3]:
```
# Columns
ts.columns
```

Out[3]:
```
Index(['CertainAmount', 'Outcome1', 'Outcome2', 'isHappyBlock', 'Happin
ess',
       'Outcome', 'time', 'subject_id', 'InterpHappiness', 'MoodTarge
t',
       'Expected', 'Actual', 'CertCol', 'ExpectedCol', 'RPECol', 'Gambl
e',
       'RutC', 'RutE', 'RutR', 'block'],
      dtype='object')
```

In [4]:
```
n_trials = 81
n_subjects = int(len(ts)/n_trials)
# reformat data into separate variables that are #trials x #subjects
all_trial_nos = ts.time.values.reshape((-1, n_trials)).T
all_participant = ts.subject_id.values.reshape((-1, n_trials)).T
all_outcome1 = ts.Outcome1.values.reshape((-1, n_trials)).T
all_outcome2 = ts.Outcome2.values.reshape((-1, n_trials)).T
all_certainAmount = ts.CertainAmount.values.reshape((-1, n_trials)).T
all_choice = ts.Gamble.values.reshape((-1, n_trials)).T
all_outcomeAmount = ts.Actual.values.reshape((-1, n_trials)).T
all_mood_rating = ts['Happiness'].values.reshape((-1, n_trials)).T
n_subjects
```

Out[4]:
```
80
```

In [5]:
```
all_winAmount = np.maximum(all_outcome1, all_outcome2)
all_loseAmount = np.minimum(all_outcome1, all_outcome2)
```

I. Curve-fitting

The LTA model with time drift is defined as follows

$$E(t) = \frac{1}{t-1} \sum_{u=1}^{t-1} A(u)$$

$$\hat{M}(t) = M_0 + \beta_E \sum_{u=1}^{t} \lambda^{t-u} E(u) + \beta_A \sum_{u=1}^{t} \lambda^{t-u} A(u) + \beta_T T(t)$$

where $A(t)$ is the actual outcome of trial t, $T(t)$ is the time stamp (here just the trial number), $M(t)$ is the mood rating.

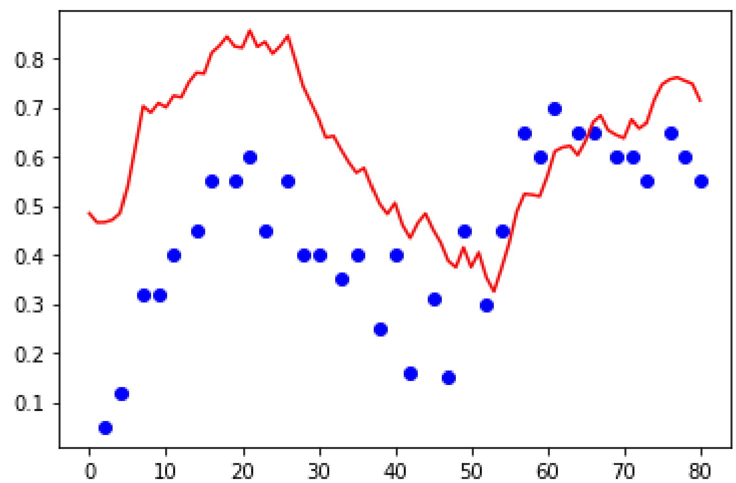
λ and M_0 are constrained to lie in [0,1]. β_A and β_E are constrained to be nonnegative. There are no constraints on β_T.

In [182]:

```
## Define a class for the LTA model

class CurveLTA(object):

    def __init__(self):
        pass

    # intializes with some default parameters
    def initialize(self):
        self.m0 = 0.5
        self.lam = 0.8
        self.betaE = 0.01
        self.betaA = 0.005
        self.betaT = 0.0001

    def predict(self, actual, timestamps):
        n_trials = len(actual)
        # holds the predicted moods
        mood_pred = np.zeros(n_trials)
        # Holds the exponentially weighted sums for E(t) and A(t)
        sum_E = 0
        sum_A = 0
        for trial_no in range(n_trials):
            if trial_no == 0:
                lte = 0
            else:
                lte = np.mean(actual[:trial_no])
            sum_E = sum_E * self.lam + lte
            sum_A = sum_A * self.lam + actual[trial_no]
            mood_mu = self.m0 + self.betaE * sum_E + self.betaA * sum_A
+ self.betaT * timestamps[trial_no]
            mood_pred[trial_no] = mood_mu
        return mood_pred
```

In [197]:
```
# use a single subject for all demonstrations
subject_index = 7
time = all_trial_nos[:, subject_index]
actual = all_outcomeAmount[:, subject_index]
mood = all_mood_rating[:, subject_index]
highGamble = all_winAmount[:, subject_index]
lowGamble = all_loseAmount[:, subject_index]
certain = all_certainAmount[:, subject_index]
choice = all_choice[:, subject_index]

CL = CurveLTA()
CL.initialize()
plt.plot(time, CL.predict(actual, time), c="r")
plt.scatter(time, mood, c="b")
```

Out[197]: <matplotlib.collections.PathCollection at 0x7fc0e0f51d00>



Defining a loss function for optimization

In [198]:
```
from scipy.optimize import minimize, Bounds
```

In [199]:
```
CL = CurveLTA()
def loss_func(par):
    CL.m0, CL.lam, CL.betaE, CL.betaA, CL.betaT = par
    mood_pred = CL.predict(actual, time)
    return np.nansum(np.abs(mood - mood_pred))
```

In [200]:
```
# What's the loss on the default parameters?
loss_func([0.5,0.8,0.01,0.005,0.0001])
```

Out[200]: 6.919534885155892

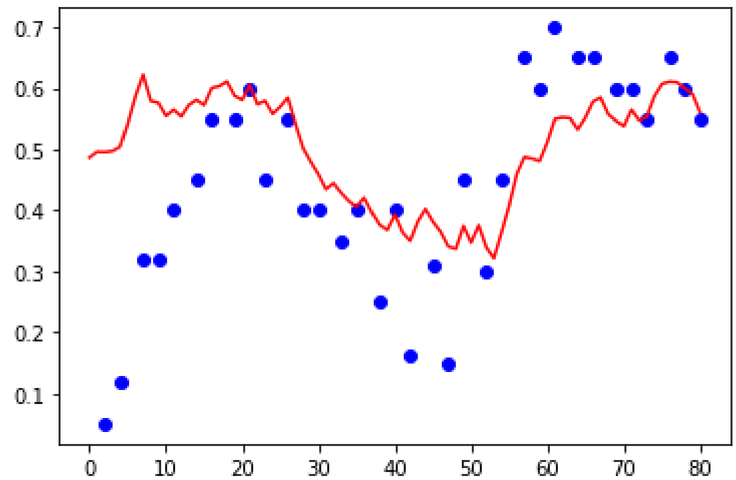
In [201]:
```
# call to minimize the function
minimize(loss_func, [0.5,0.8,0.01,0.005,0.0001],
         bounds=Bounds([0.0,0.0,0.0,0.0,-np.inf],[1.0, 1.0, np.inf,np.in
f,np.inf]))
```

Out[201]:
```
      fun: 3.7367261254672646
 hess_inv: <5x5 LbfgsInvHessProduct with dtype=float64>
      jac: array([ 12.00000002,   4.06740279, 237.14949782,  41.8678734
8,
       -57.39448903])
  message: b'ABNORMAL_TERMINATION_IN_LNSRCH'
     nfev: 432
      nit: 11
     njev: 72
   status: 2
  success: False
        x: array([ 4.99402509e-01,  7.99328776e-01,  0.00000000e+00,
3.93384334e-03,
       -6.09039431e-04])
```

In [202]:
```
# Now add the minimization to the class

class CurveLTA(object):

    _par_names = ['m0','lam','betaE','betaA','betaT']
    _default_pars = [0.5,0.8,0.01,0.005,0.0001]
    _lower_bounds = [0.0,0.0,0.0,0.0,-np.inf]
    _upper_bounds = [1.0, 1.0, np.inf,np.inf,np.inf]

    def __init__(self):
        pass

    # intializes with some default parameters
    def initialize(self):
        self.m0, self.lam, self.betaE, self.betaA, self.betaT = self._de
fault_pars

    def fit(self, actual, timestamps, mood):
        def loss_func(par):
            self.m0, self.lam, self.betaE, self.betaA, self.betaT = par
            mood_pred = self.predict(actual, time)
            return np.nansum(np.abs(mood - mood_pred))
        res = minimize(loss_func, self._default_pars,
            bounds=Bounds(self._lower_bounds,self._upper_bounds))
        self.m0, self.lam, self.betaE, self.betaA, self.betaT = res.x
        return res

    def predict(self, actual, timestamps):
        n_trials = len(actual)
        # holds the predicted moods
        mood_pred = np.zeros(n_trials)
        # Holds the exponentially weighted sums for E(t) and A(t)
        sum_E = 0
        sum_A = 0
        for trial_no in range(n_trials):
            if trial_no == 0:
                lte = 0
            else:
                lte = np.mean(actual[:trial_no])
            sum_E = sum_E * self.lam + lte
            sum_A = sum_A * self.lam + actual[trial_no]
            mood_mu = self.m0 + self.betaE * sum_E + self.betaA * sum_A
+ self.betaT * timestamps[trial_no]
            mood_pred[trial_no] = mood_mu
        return mood_pred
```

In [203]:
```
CL = CurveLTA()
CL.fit(actual, time, mood)
plt.plot(time, CL.predict(actual, time), c="r")
plt.scatter(time, mood, c="b")
```

Out[203]:
```
<matplotlib.collections.PathCollection at 0x7fc0e0141820>
```



In [204]:
```
# MAE
np.nanmean(np.abs(mood - CL.predict(actual, time)))
```

Out[204]:
```
0.10990370957256661
```

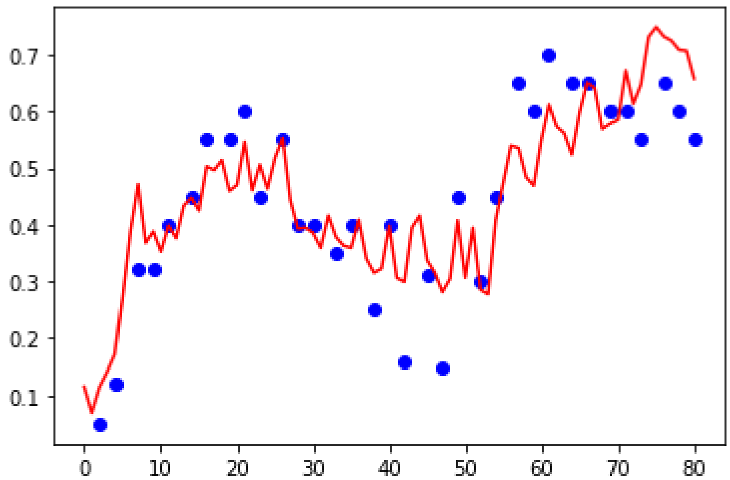
Regularized curve-fitting

Now we add L1 penalties to the coefficients

In [205]:
```
# Now add the minimization to the class

class CurveLTA(object):

    _par_names = ['m0','lam','betaE','betaA','betaT']
    _default_pars = [0.5,0.8,0.01,0.005,0.0001]
    _lower_bounds = [0.0,0.0,0.0,0.0,-np.inf]
    _upper_bounds = [1.0, 1.0, np.inf,np.inf,np.inf]
    pen_betaE = 0.0
    pen_betaA = 0.0
    pen_betaT = 0.0

    def __init__(self, pen_betaE = 0.0, pen_betaA = 0.0, pen_betaT = 0.0
):
        self.pen_betaE = pen_betaE
        self.pen_betaA = pen_betaA
        self.pen_betaT = pen_betaT

    # intializes with some default parameters
    def initialize(self):
        self.m0, self.lam, self.betaE, self.betaA, self.betaT = self._de
fault_pars

    def fit(self, actual, timestamps, mood):
        def loss_func(par):
            self.m0, self.lam, self.betaE, self.betaA, self.betaT = par
            mood_pred = self.predict(actual, time)
            pen_term = np.abs(self.pen_betaE * self.betaE) + \
                np.abs(self.pen_betaA * self.betaA) + \
                np.abs(self.pen_betaT * self.betaT)
            return np.nansum(np.abs(mood - mood_pred)) + pen_term
        res = minimize(loss_func, self._default_pars,
            bounds=Bounds(self._lower_bounds,self._upper_bounds))
        self.m0, self.lam, self.betaE, self.betaA, self.betaT = res.x
        return res

    def predict(self, actual, timestamps):
        n_trials = len(actual)
        # holds the predicted moods
        mood_pred = np.zeros(n_trials)
        # Holds the exponentially weighted sums for E(t) and A(t)
        sum_E = 0
        sum_A = 0
        for trial_no in range(n_trials):
            if trial_no == 0:
                lte = 0
            else:
                lte = np.mean(actual[:trial_no])
            sum_E = sum_E * self.lam + lte
            sum_A = sum_A * self.lam + actual[trial_no]
            mood_mu = self.m0 + self.betaE * sum_E + self.betaA * sum_A
+ self.betaT * timestamps[trial_no]
            mood_pred[trial_no] = mood_mu
        return mood_pred
```

In [206]:
```
CL = CurveLTA(pen_betaE = 0.1, pen_betaA = 0.1)
CL.fit(actual, time, mood)
plt.plot(time, CL.predict(actual, time), c="r")
plt.scatter(time, mood, c="b")
```

Out[206]: <matplotlib.collections.PathCollection at 0x7fc0e0e47c10>



In [207]:
```
# MAE
np.nanmean(np.abs(mood - CL.predict(actual, time)))
```

Out[207]: 0.061526095054160186

Held-out prediction

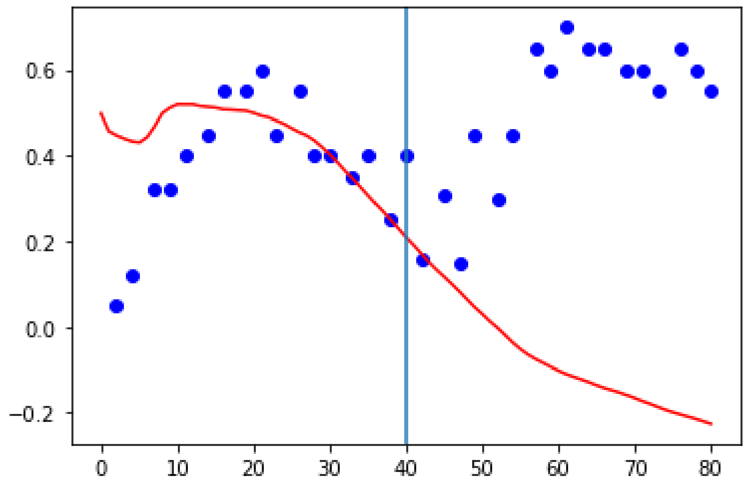
We fit to the first 40 trials and predict on the remaining

In [297]:
```
CL.fit(actual[:40], time[:40], mood[:40])
```

Out[297]:
```
      fun: 1.690281251212084
 hess_inv: <5x5 LbfgsInvHessProduct with dtype=float64>
      jac: array([  4.        ,  -0.66621279,   1.43200838,  49.4497209
2,
       -42.68056697])
  message: b'ABNORMAL_TERMINATION_IN_LNSRCH'
     nfev: 900
      nit: 21
     njev: 150
   status: 2
  success: False
        x: array([ 0.49846875,  0.80032608,  0.00993689,  0.        , -
0.00986397])
```

In [298]:
```
plt.plot(time, CL.predict(actual, time), c="r")
plt.axvline(40)
plt.scatter(time, mood, c="b")
```

Out[298]: <matplotlib.collections.PathCollection at 0x7fc0e310bd90>



In [210]:
```
# MAE (training, test)
errs = np.abs(mood - CL.predict(actual, time))
np.nanmean(errs[:40]), np.nanmean(errs[40:])
```

Out[210]: (0.04661197359621433, 0.0747830919056676)

IIa. Likelihood model

Now we add a utility function and jointly model mood with decision-making.

Data variables:

- $H(t)$ high gamble
- $L(t)$ low gamble
- $C(t)$ deterministic amount
- $A(t)$ actual outcome
- $G(t)$ whether gambling occurred
- $T(t)$ time elapsed

Define $W(t) = 1$ if $A(t) = H(t)$ and $G(t) = 1$, and $W(t) = 0$ otherwise.

Define the subjective win probability as

$$p(t) = \frac{\sum_{i=1}^{t-1} W(i)}{\sum_{i=1}^{t-1} G(i)}$$

with $p(t) = 0.5$ when it would otherwise be undefined.

Model parameters:

- $M_0 \in (-\infty, \infty)$ baseline logit-mood
- $\lambda \in [0, 1]$ discount factor
- $\beta_E \in [0, \infty)$ coefficient for LTA
- $\beta_A \in [0, \infty)$ coefficient for actual outcome
- $\beta_T \in (-\infty, \infty)$ coefficient for time trend
- $\gamma \in [0, 2]$ utility function exponent
- $\rho_C \in [0, \infty)$ choice inverse temperature
- $\sigma \in [0, \infty)$ Gaussian noise standard deviation for logit-mood

Latent variables:

- *Choice bias*: $V(t) = \rho_C(p(t)H(t)^\gamma + (1 - p(t))L(t)^\gamma - C(t)^\gamma)$
- *Long-term average*: $E(t) = \frac{1}{t-1} \sum_{u=1}^{t-1} A(u)^\gamma$
- *Predicted logit-mood*: $\mu(t) = M_0 + \beta_E \sum_{u=1}^{t} \lambda^{t-u} E(u) + \beta_A \sum_{u=1}^{t} \lambda^{t-u} A(u)^\gamma + \beta_T T(t)$

Model:

- $G(t) \sim \text{Binomial}(\text{expit}(V(t))$
- $M(t) \sim \text{LogitNormal}(\mu(t), \sigma)$

where $\text{expit}(x) = \frac{1}{1+e^{-x}}$

In [211]:
```
from scipy.stats import norm, bernoulli
from scipy.special import logit, expit
```

In [212]:
```
def signedPower(x, y):
    return np.power(np.abs(x), y) * np.sign(x)

class MoodChoiceLTA(object):

    _model_name = 'LTA nonlinear with simple win prob. and choice'
    _par_names = ['m0','lam','betaE','betaA','betaT','gamma','rhoC','sig
ma']
    _default_pars = [0.0,0.8,0.01,0.005,0.0001,1.0,1.0,0.5]
    _lower_bounds = [-np.inf,0.0,0.0,0.0,-np.inf,0.0,0.0,0.0]
    _upper_bounds = [np.inf, 1.0, np.inf,np.inf,np.inf,2.0,np.inf,np.inf
]

    def __init__(self):
        pass

    # prints parameters
    def __str__(self):
        s = self._model_name
        for par in self._par_names:
            s = s + '\n' + par + ': %.4f' % self.__dict__[par]
        return s

    # intializes with some default parameters
    def initialize(self, params = None):
        if params is None:
            params = self._default_pars
        self.m0, self.lam, self.betaE, self.betaA, self.betaT, \
            self.gamma, self.rhoC, self.sigma = params

    def fit(self, actual, certain, highGamble, lowGamble, choice, timest
amps, mood):
        def loss_func(par):
            self.m0, self.lam, self.betaE, self.betaA, self.betaT, \
                self.gamma, self.rhoC, self.sigma = par
            return -self.loglike(actual, certain, highGamble, lowGamble,
choice, timestamps, mood)
        res = minimize(loss_func, self._default_pars,
            bounds=Bounds(self._lower_bounds,self._upper_bounds))
        self.m0, self.lam, self.betaE, self.betaA, self.betaT, \
            self.gamma, self.rhoC, self.sigma = res.x
        return res

    def loglike(self, actual, certain, highGamble, lowGamble, choice, ti
mestamps, mood):
        mood_logit, choice_logit = self.predict(actual, certain, highGam
ble, lowGamble, choice, timestamps)
        choice_ll = bernoulli.logpmf(choice, expit(choice_logit))
        mood_ll = norm.logpdf(logit(mood), loc=mood_logit, scale=self.si
gma)
        return np.nansum(choice_ll) + np.nansum(mood_ll)

    def predict(self, actual, certain, highGamble, lowGamble, choice, ti
mestamps):
        n_trials = len(actual)
        # compute the win probabilities
```

```
        winIndicator = (highGamble == actual) * choice
        pwin = 0.5 * np.ones(n_trials)
        temp1 = np.cumsum(winIndicator, axis = 0)
        temp2 = np.cumsum(choice, axis = 0)
        pwin[temp2 > 0] = temp1[temp2 > 0]/temp2[temp2 > 0]
        pwin = np.concatenate(([0.5], pwin[:-1]))
        # holds the predicted moods and choices
        mood_logit = np.zeros(n_trials)
        choice_logit = np.zeros(n_trials)
        # Holds the exponentially weighted sums for E(t) and A(t)
        sum_E = 0
        sum_A = 0
        for trial_no in range(n_trials):
            if trial_no == 0:
                lte = 0
            else:
                lte = np.mean(signedPower(actual[:trial_no], self.gamma))
            sum_E = sum_E * self.lam + lte
            sum_A = sum_A * self.lam + signedPower(actual[trial_no], self.gamma)
            choice_bias = self.rhoC * \
                (pwin[trial_no] * signedPower(highGamble[trial_no], self.gamma) + \
                 (1-pwin[trial_no]) * signedPower(lowGamble[trial_no], self.gamma) - \
                 signedPower(certain[trial_no], self.gamma))
            mood_mu = self.m0 + self.betaE * sum_E + self.betaA * sum_A + self.betaT * timestamps[trial_no]
            mood_logit[trial_no] = mood_mu
            choice_logit[trial_no] = choice_bias
        return mood_logit, choice_logit

    def sample(self, actual, certain, highGamble, lowGamble, choice, timestamps):
        mood_logit, choice_logit = self.predict(actual, certain, highGamble, lowGamble, choice, timestamps)
        mood_sample = expit(norm.rvs(loc=mood_logit, scale=self.sigma))
        choice_sample = bernoulli.rvs(expit(choice_logit))
        return mood_sample, choice_sample
```

In [217]:
```
# use a single subject for all demonstrations
subject_index = 7
time = all_trial_nos[:, subject_index]
actual = all_outcomeAmount[:, subject_index]
mood = all_mood_rating[:, subject_index]
highGamble = all_winAmount[:, subject_index]
lowGamble = all_loseAmount[:, subject_index]
certain = all_certainAmount[:, subject_index]
choice = all_choice[:, subject_index]


MCL = MoodChoiceLTA()
MCL.initialize()
#MCL.predict(actual, certain, highGamble, lowGamble, choice, time)
MCL.loglike(actual, certain, highGamble, lowGamble, choice, time, mood)
```

Out[217]:
```
-99.82923444303935
```

In [218]:
```
MCL.fit(actual, certain, highGamble, lowGamble, choice, time, mood)
```

Out[218]:
```
      fun: 40.08111367796036
 hess_inv: <8x8 LbfgsInvHessProduct with dtype=float64>
      jac: array([ 0.00021743,  0.0007482 , -0.00091873,  0.00686526, -
0.01013589,
        0.00192344,  0.00022311,  0.00174936])
  message: b'CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH'
     nfev: 981
      nit: 93
     njev: 109
   status: 0
  success: True
        x: array([-1.97364268,  0.2770499 ,  0.4732268 ,  0.11227632,
0.02722128,
        0.56262762,  2.21865169,  0.37428349])
```

In [219]:
```
MCL.loglike(actual, certain, highGamble, lowGamble, choice, time, mood)
```

Out[219]:
```
-40.08111367796036
```

In [220]:
```
print(MCL)
```

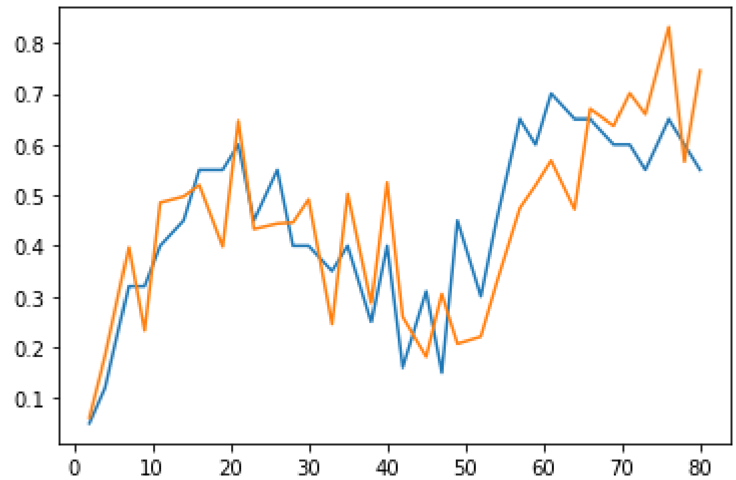
```
LTA nonlinear with simple win prob. and choice
m0: -1.9736
lam: 0.2770
betaE: 0.4732
betaA: 0.1123
betaT: 0.0272
gamma: 0.5626
rhoC: 2.2187
sigma: 0.3743
```

Parameter recovery example

In [221]:
```
# Simulate data
mood_s, choice_s = MCL.sample(actual, certain, highGamble, lowGamble, ch
oice, time)
# copy missing pattern of original
mood_s[np.isnan(mood)] = np.nan
```

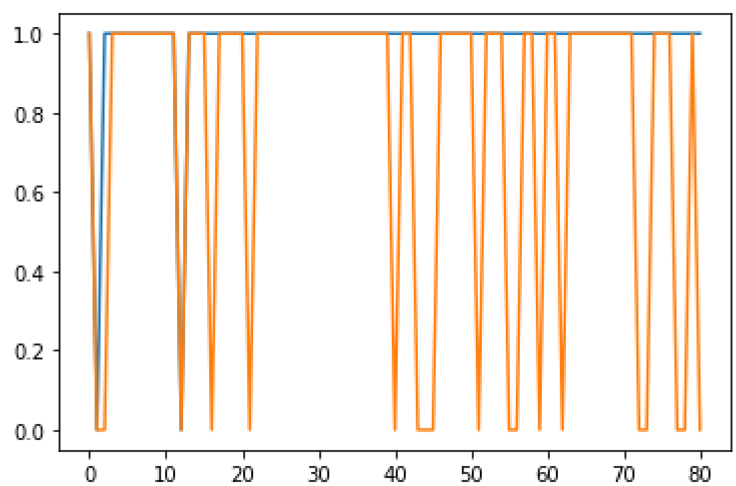
In [222]:
```
# plot original vs simulated mood
plt.plot(time[~np.isnan(mood)], mood[~np.isnan(mood)])
plt.plot(time[~np.isnan(mood)], mood_s[~np.isnan(mood)])
```

Out[222]:
```
[<matplotlib.lines.Line2D at 0x7fc0e0f9e070>]
```



In [223]:
```
# plot choice
plt.plot(time, choice)
plt.plot(time, choice_s)
```

Out[223]:
```
[<matplotlib.lines.Line2D at 0x7fc0e056f460>]
```



In [224]:
```
# fit on simulated
MCL2 = MoodChoiceLTA()
MCL2.fit(actual, certain, highGamble, lowGamble, choice_s, time, mood_s)
```

Out[224]:
```
      fun: 46.168111630687804
 hess_inv: <8x8 LbfgsInvHessProduct with dtype=float64>
      jac: array([-0.00100471, -0.00409059, -0.00233698, -0.00676579,
0.00619877,
       -0.00301412,  0.00015561, -0.00622364])
  message: b'CONVERGENCE: REL_REDUCTION_OF_F_<=_FACTR*EPSMCH'
     nfev: 1305
      nit: 124
     njev: 145
   status: 0
  success: True
        x: array([-2.17761285,  0.10234372,  1.09588124,  0.14728704,
0.02959408,
        0.37319938,  2.62004156,  0.41533034])
```

In [225]:
```
# simulated
print(MCL2)
```

```
LTA nonlinear with simple win prob. and choice
m0: -2.1776
lam: 0.1023
betaE: 1.0959
betaA: 0.1473
betaT: 0.0296
gamma: 0.3732
rhoC: 2.6200
sigma: 0.4153
```

In [226]:
```
# original
print(MCL)
```

```
LTA nonlinear with simple win prob. and choice
m0: -1.9736
lam: 0.2770
betaE: 0.4732
betaA: 0.1123
betaT: 0.0272
gamma: 0.5626
rhoC: 2.2187
sigma: 0.3743
```

IIb: Bayesian model

We now add prior distributions to the previous model parameters

In [227]:
```
from RunDEMC import Model, Param, dists, calc_bpic, joint_plot
```

In [241]:
```
# Priors for parameters
params = [Param(name='m0',
                display_name=r'm0',
                prior=dists.normal(0, 1)),
          Param(name='lam',
                display_name=r'$lambda',
                prior=dists.normal(0, 1.4),
                transform=dists.invlogit
                ),
          Param(name='betaE',
                display_name=r'beta_E',
                prior=dists.normal(-3, 0.7),
                transform=np.exp,
                inv_transform=np.log),
          Param(name='betaA',
                display_name=r'beta_A',
                prior=dists.normal(-3, 0.7),
                transform=np.exp,
                inv_transform=np.log),
          Param(name='betaT',
                display_name=r'beta_T',
                prior=dists.normal(0, 0.00001)),
          Param(name='gamma',
                display_name=r'$\gamma$',
                prior=dists.gamma(1.5, 0.5),
                ),
          Param(name='rhoC',
                display_name=r'$rho_C$',
                prior=dists.gamma(1.5, 0.5),
                ),
          Param(name='sigma',
                display_name=r'sigma',
                prior=dists.exp(1))
        ]
```

In [242]:
```
# use a single subject for all demonstrations
subject_index = 7
time = all_trial_nos[:, subject_index]
actual = all_outcomeAmount[:, subject_index]
mood = all_mood_rating[:, subject_index]
highGamble = all_winAmount[:, subject_index]
lowGamble = all_loseAmount[:, subject_index]
certain = all_certainAmount[:, subject_index]
choice = all_choice[:, subject_index]
```

In [243]:
```
def eval_fun(params):
    md = MoodChoiceLTA()
    params2 = np.array([params[n] for n in md._par_names])
    n_params, n_particles = params2.shape
    ll = -np.inf * np.ones(n_particles)
    valid1 = (params2 > np.reshape(np.array(md._lower_bounds), (n_params
, 1)))
    valid2 = (params2 < np.reshape(np.array(md._upper_bounds), (n_params
, 1)))
    valid = np.logical_and(valid1, valid2)
    for ind_part in np.nonzero(valid)[0]:
        md.initialize(params2[:, ind_part])
        ll[ind_part] = md.loglike(actual, certain, highGamble, lowGamble
, choice, time, mood)
    return ll
```

In [244]:
```
m = Model('mmi', params=params,
    like_fun=eval_fun,
    init_multiplier = 3,
    use_priors = False,
    verbose=True)
```

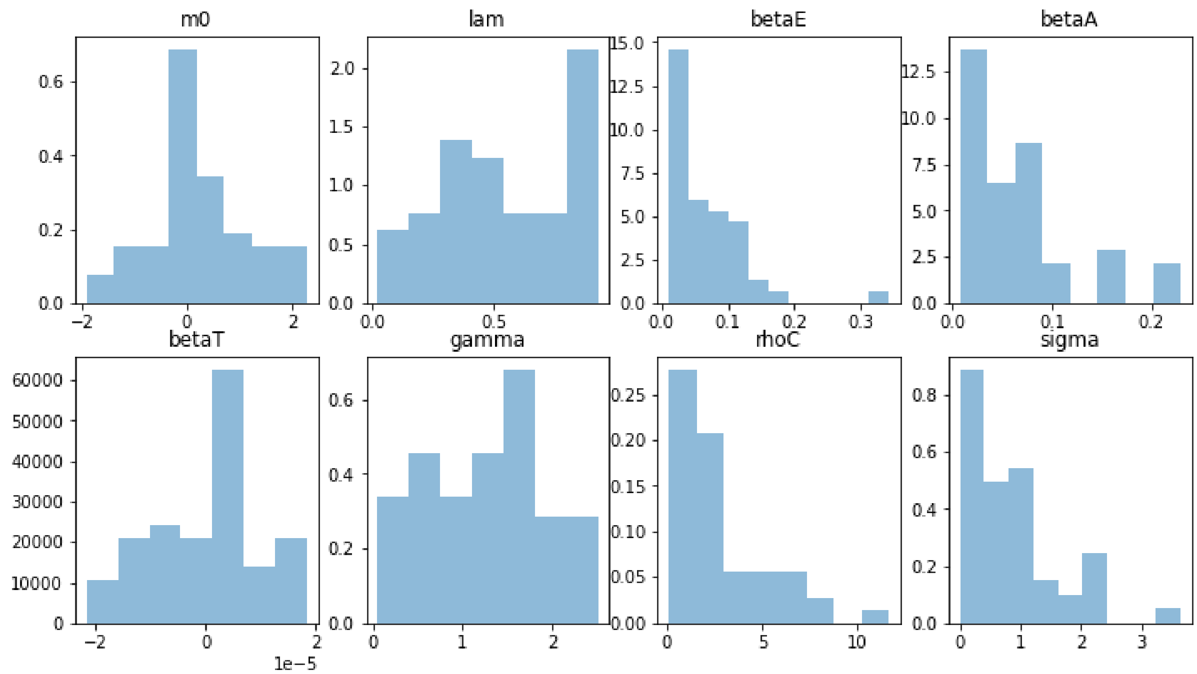
In [245]:
```
m._initialize(num_chains=50)
```

```
Initializing: 150(50) 145(45) 142(42) 139(39) 136(36) 132(32) 126(26) 1
20(20) 117(17) 113(13) 110(10) 106(6) 103(3)
```

In [246]:
```
np.min(m.weights[-1]), np.max(m.weights[-1])
```

Out[246]:
```
(-17322870.812327046, -68.18925915511545)
```

In [247]:
```
plt.figure(figsize=(12,10))
for i in range(8):
    plt.subplot(3,4,i+1)
    plt.hist(m.particles[:, :, i].flatten(), bins='auto', alpha=.5, dens
ity=True)
    plt.title(MCL._par_names[i])
```



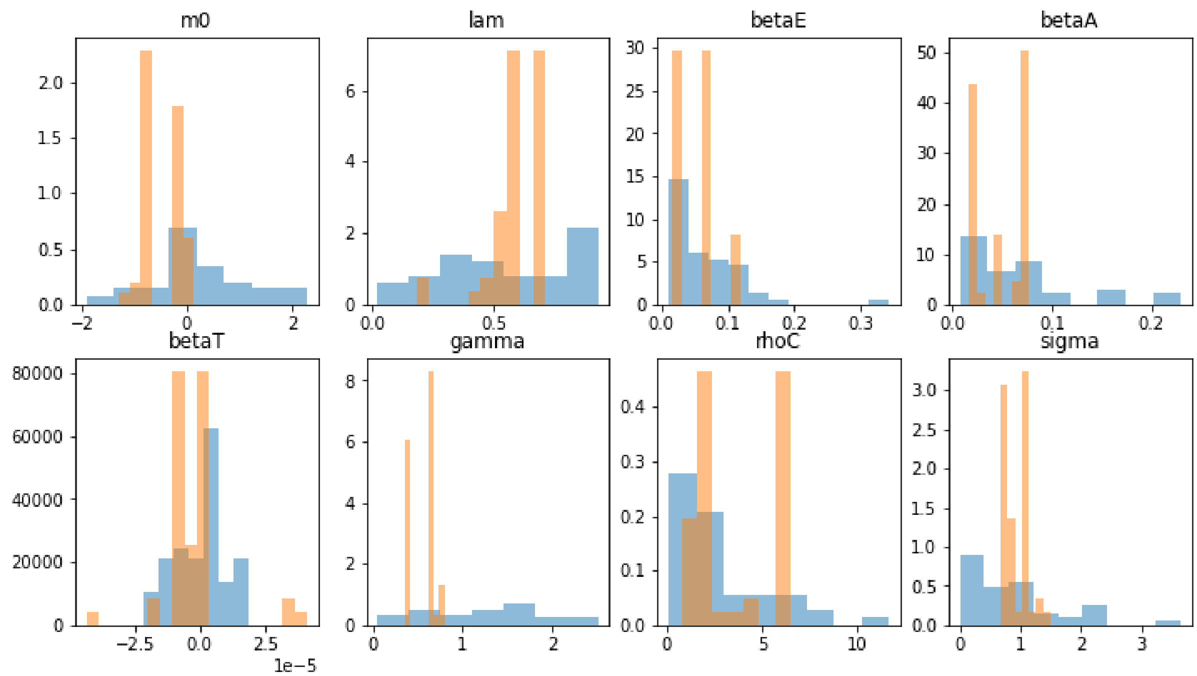
In [248]:
```
times = m.sample(100, burnin=True, migration_prob = 0.1)
```

```
Iterations (100): 1 2 3 4 5 6 7 8 x 9 10 11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30 31 32 33 34 x 35 36 37 38 39 40 41 42 43
44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67
68 69 70 71 72 73 x 74 x 75 76 77 x 78 79 80 81 82 83 84 85 86 x 87 88
89 90 91 92 93 94 95 96 97 98 99 100
```

In [249]:
```
np.min(m.weights[-1]), np.max(m.weights[-1])
```

Out[249]:
```
(-101.60792264263635, -68.18925915511545)
```

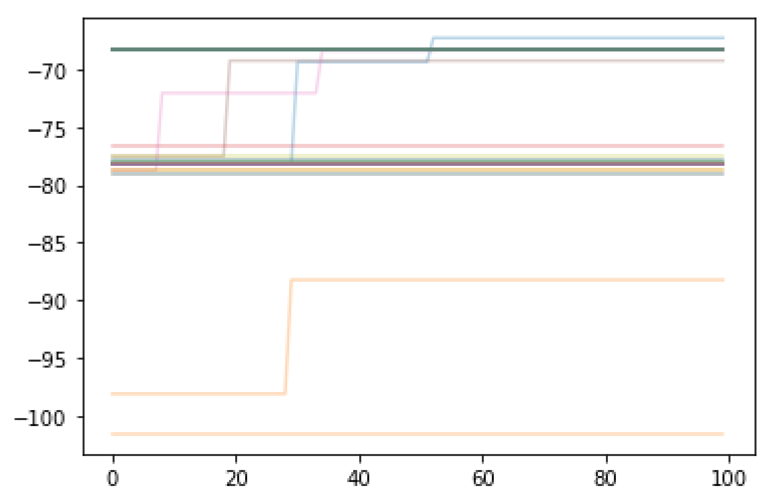
In [250]:
```
plt.figure(figsize=(12,10))
for i in range(8):
    plt.subplot(3,4,i+1)
    plt.hist(m.particles[0, :, i].flatten(), bins='auto', alpha=.5, dens
ity=True)
    plt.hist(m.particles[-1, :, i].flatten(), bins='auto', alpha=.5, den
sity=True)
    plt.title(MCL._par_names[i])
```



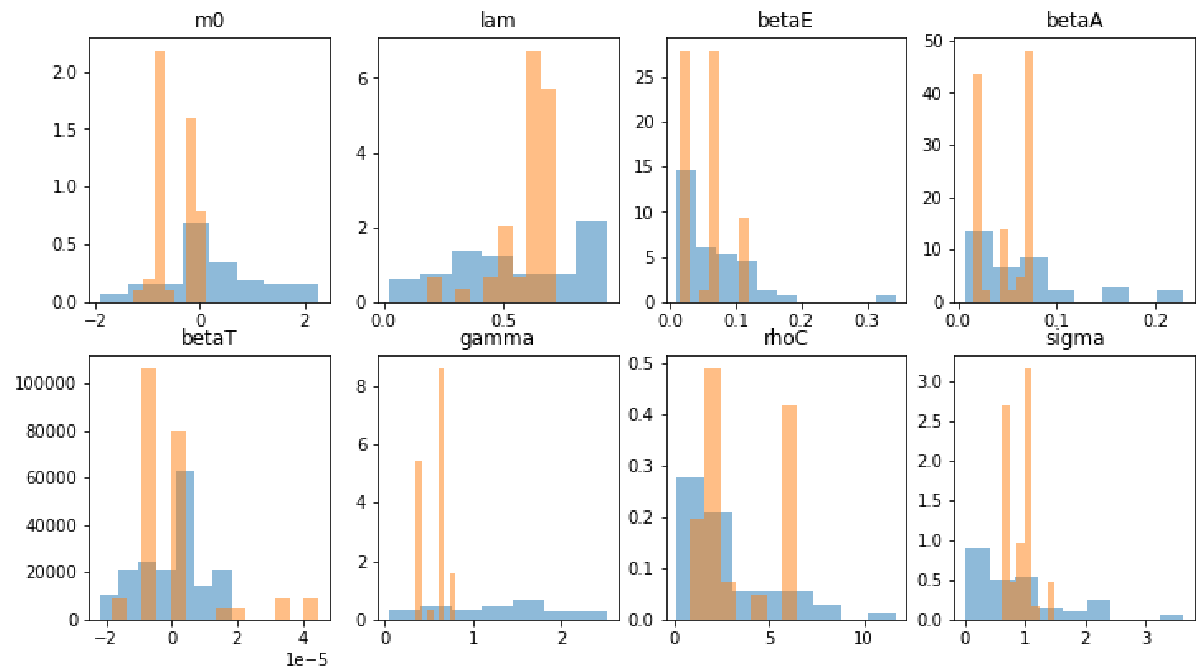
In [251]:
```
times = m.sample(100, burnin=False)
```

```
Iterations (100): 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93
94 95 96 97 98 99 100
```

In [252]:
```
# show the chains are mixing and converging
plt.plot(m.weights[-100:], alpha=.3);
```



In [253]:
```
plt.figure(figsize=(12,10))
for i in range(8):
    plt.subplot(3,4,i+1)
    plt.hist(m.particles[0, :, i].flatten(), bins='auto', alpha=.5, dens
ity=True)
    plt.hist(m.particles[-1, :, i].flatten(), bins='auto', alpha=.5, den
sity=True)
    plt.title(MCL._par_names[i])
```



In [261]:
```
# debugging
```

In [289]:
```
p1=m.particles[-1, 0]
p1
```

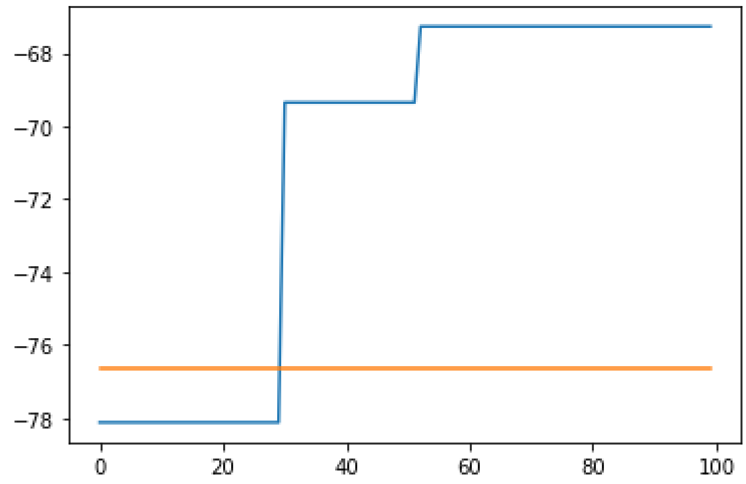
Out[289]:
```
array([-9.97538338e-05,  6.00347560e-01,  5.43897946e-02,  5.24324942e-
02,
        1.91922978e-05,  5.45469232e-01,  2.89763620e+00,  7.88518713e-
01])
```

In [290]:
```
p2=m.particles[-1, 3]
p2
```

Out[290]:
```
array([-7.31339565e-01,  5.44584541e-01,  1.55575456e-02,  7.67690370e-
02,
        4.10651593e-05,  6.73260648e-01,  2.64165594e+00,  1.15006514e+
00])
```

In [291]:
```
plt.plot(m.weights[-100:, [0,3]])
```

Out[291]:
```
[<matplotlib.lines.Line2D at 0x7fc0e0ec0eb0>,
 <matplotlib.lines.Line2D at 0x7fc0e0ec0fa0>]
```



In [292]:
```
def ll(params):
    MCL = MoodChoiceLTA()
    MCL.initialize(params)
    return MCL.loglike(actual, certain, highGamble, lowGamble, choice, t
ime, mood)
```

In [296]:
```
[ll(p) for p in m.particles[-1]]
```

Out[296]:
```
[-68.55036214985282,
 -80.854936970193,
 -69.23349712768768,
 -69.29013840688418,
 -78.90207021908176,
 -70.29094947429809,
 -70.0605273538265,
 -69.23349712768768,
 -80.81496092937205,
 -69.23349712768768,
 -78.90207021908176,
 -78.90207021908176,
 -78.90207021908176,
 -69.23349712768768,
 -69.23349712768768,
 -69.23349712768768,
 -69.23349712768768,
 -69.23349712768768,
 -78.84060454146022,
 -78.90207021908176,
 -69.23349712768768,
 -100.71895772569297,
 -78.29758245238412,
 -69.23349712768768,
 -78.29758245238412,
 -69.23349712768768,
 -78.90207021908176,
 -69.23349712768768,
 -78.90207021908176,
 -69.23349712768768,
 -80.81496092937205,
 -80.81496092937205,
 -69.23349712768768,
 -78.90207021908176,
 -80.81496092937205,
 -78.90207021908176,
 -78.90207021908176,
 -69.23349712768768,
 -78.90207021908176,
 -80.81496092937205,
 -78.90207021908176,
 -78.84060454146022,
 -69.23349712768768,
 -69.23349712768768,
 -78.90207021908176,
 -78.90207021908176,
 -78.90207021908176,
 -69.23349712768768,
 -78.90207021908176,
 -69.23349712768768]
```

In [293]:
```
ll(p1)
```

Out[293]:
```
-68.55036214985282
```

In [294]:
```
ll(p2)
```

Out[294]:
```
-69.29013840688418
```

In [295]:
```
ll(0.5 * p1 + 0.5 * p2)
```

Out[295]:
```
-64.68052793018006
```

In [ ]:
```

```