FinalThesisFunctions

March 28, 2017

In [ ]:
```
import pandas as pd
import numpy as np
import scipy as sp
import sklearn as sk
import math
import csv
import statsmodels.api as sm
import statsmodels.formula.api as smf
import random
import matplotlib.pyplot as plt
import pylab as plt
import plotly.plotly as py
import plotly
import plotly.graph_objs as go
from scipy.stats.stats import pearsonr
from sklearn import linear_model, datasets
import itertools


##########################################################################
##[FUNCTION] data_creation simulates data for a given number of
## individuals(indiv) over a set amount of time (max_time), and can
## include as many covariates as desired (number_of_covariates)
##########################################################################


def data_creation2(indiv, max_time, number_of_covariates, Y_full, alpha,\
    beta):

    columns = ["indiv", "time","U", "A", "Y",  "L1"]
    df = pd.DataFrame(columns = columns)

    ## creating an unobserved variable that affects covariates
    U = np.random.uniform(low = 0.1, high = 1, size = indiv)

    for jj in range(0, max_time+1):
        if jj == 0:
            x_L = alpha[0] + alpha[5]*U
```

```
        L1 = np.random.binomial(n=1, p = np.exp(x_L)/(1+np.exp(x_L)))

        x_A = beta[0] + beta[1]*L1
        A = np.random.binomial(n=1, p = np.exp(x_A)/(1+np.exp(x_A)))

        df = pd.DataFrame({"indiv":range(1,indiv+1), "time":jj,"U":U,\
            "A":A, "Y":[math.nan]*indiv, "L1":L1})

    elif jj == 1:
        x_L = np.sum(alpha*np.transpose(np.array([[1.0]*indiv, \
              df["L1"][(df.time == jj-1)], [0.0]*indiv, \
              df["A"][(df.time == jj-1)],[0.0]*indiv, U])), axis = 1)

        L1 = np.random.binomial(n=1, p = np.exp(x_L)/(1+np.exp(x_L)))


        x_A = np.sum(beta*np.transpose(np.array([[1.0]*indiv, L1, \
              df["L1"][(df.time == jj-1)],df["A"][(df.time == \
              jj-1)], [0.0]*indiv ])), axis = 1)

        A = np.random.binomial(n=1, p = np.exp(x_A)/(1+np.exp(x_A)))

        temp_df = pd.DataFrame({"indiv":range(1,indiv+1), "time":jj,\
                  "U":U, "A":A,"Y":[math.nan]*indiv, "L1":L1})
        df = pd.concat([df, temp_df])

    else:
        x_L = np.sum(alpha*np.transpose(np.array([[1.0]*indiv, \
              df["L1"][(df.time == jj-1)],df["L1"][(df.time == \
              jj-2)], df["A"][(df.time == jj-1)], \
              df["A"][(df.time == jj-2)], U])), axis = 1)

        L1 = np.random.binomial(n=1, p = np.exp(x_L)/(1+np.exp(x_L)))


        x_A = np.sum(beta*np.transpose(np.array([[1.0]*indiv,L1,\
              df["L1"][(df.time == jj-1)],df["A"][(df.time == jj-1)]\
            , df["A"][(df.time == jj-2)]])), axis = 1)

        A = np.random.binomial(n=1, p = np.exp(x_A)/(1+np.exp(x_A)))

        if jj == max_time:
            ## no treatment effect (null hypothesis)
            x_Y = 0.5 + U
            ## treatment effect (alternative hypothesis)
            x_Y = [-1]*indiv + U + A + df.groupby(["indiv"]).A.mean()

            Y = np.random.binomial(n=1, p = np.exp(x_Y)/\
```

```python
                        (1+np.exp(x_Y)))
                    temp_df = pd.DataFrame({"indiv":range(1,indiv+1), \
                              "time":jj,"U":U, "A":A,"Y":Y, "L1":L1})
                    df = pd.concat([df, temp_df])


                else:
                    temp_df = pd.DataFrame({"indiv":range(1,indiv+1), \
                              "time":jj,"U":U, "A":A,"Y":[math.nan]*\
                              indiv, "L1":L1})
                    df = pd.concat([df, temp_df])


    # creating shifted values
    if Y_full == True:
        for kk in range(1,max_time+1):
            df["L1_"+str(kk)] = df.L1.shift(kk)
            df["A_"+str(kk)] = df.A.shift(kk)
    else:
        for kk in range(1,4):
            df["L1_"+str(kk)] = df.L1.shift(kk)
            df["A_"+str(kk)] = df.A.shift(kk)

    df.sort_values(by=['time', 'indiv'], ascending=[True, True])

    return(df);

#########################################################################
##[FUNCTION] Y_model_creation creates the linear regression model for
## the observed Ys based on the treatments (A) and covariates (L)
#########################################################################

def Y_model_creation(df, max_time):
    temp_df = df[df.time == max_time]
    train_columns = list(df)[0:2]+list(df)[6:]
    temp_df = temp_df.astype(float)
    Y_model = sm.Logit(np.asarray(temp_df["Y"]), \
              np.asarray(sm.add_constant(temp_df[train_columns]))).fit();
    return(Y_model)


#########################################################################
##[FUNCTION] covariate_model_creation creates the logistic regression
## for the observed covariate (L) data from the previous covariates
## and the previous treatments (A)
#########################################################################
```

```python
def covariate_model_creation(df, max_time):
    train_columns = ["L1_1", "L1_2", "L1_3", "A_1", "A_2",  "A_3"]
    L1_model = {}
    poly = PolynomialFeatures(1)

    for ii in range(1, (max_time+1)):
        temp_df = df[df.time == ii]
        if ii == 1:
            x = temp_df[["L1_1", "A_1"]]
        elif ii == 2:
            x = temp_df[["L1_1", "L1_2", "A_1", "A_2"]]
        else:
            x = temp_df[train_columns]
        L1_model[ii] = sm.Logit(np.asarray(temp_df["L1"]), \
                           poly.fit_transform(x)).fit();
    return(L1_model)


##########################################################################
##[FUNCTION] treatment_model_creation creates the logistic regression
## for the observed treatment (A) data from the current and previous
## covariates and the previous treatments (A)
##########################################################################


def treatment_model_creation(df, max_time):
    train_columns = ["L1", "L1_1", "L1_2", "A_1", "A_2", "A_3"]
    A_model = {}
    poly = PolynomialFeatures(1)

    for ii in range(0, (max_time+1)):
        temp_df = df[df.time == ii]
        if ii == 0:
            x = temp_df[["L1"]]
            A_model[ii] = sm.Logit(np.asarray(temp_df["A"]), sm.add_\
                              constant(x, has_constant = "add")).fit()
        elif ii == 1:
            x = temp_df[["L1", "L1_1", "A_1"]]
            A_model[ii] = sm.Logit(np.asarray(temp_df["A"]), poly.fit\
                              _transform(x)).fit()
        elif ii == 2:
            x = temp_df[["L1", "L1_1", "L1_2", "A_1", "A_2"]]
            A_model[ii] = sm.Logit(np.asarray(temp_df["A"]), poly.fit\
                              _transform(x)).fit()
        else:
            x = temp_df[train_columns]
            A_model[ii] = sm.Logit(np.asarray(temp_df["A"]), poly.fit\
                              _transform(x)).fit()
```

```python
    return(A_model)


########################################################################
##[FUNCTION] simulation_run calculates the causal effect over an
## established number of Monte Carlo repetitions (10,000)
## using the models for outcome (Y) and the covariates (L)
########################################################################


def simulation_run(df, Y_model, L1_model_df, max_time, Y_full, \
    test_value):

    reps = 10000
    final_results = np.empty(reps)

    L_model = covariate_model_creation(df, max_time)
    poly = PolynomialFeatures(1)

    ### establishing treatment of interest
    A_test = [test_value]*(max_time+1)

    values = pd.DataFrame(np.random.choice(np.array(df["L1"][df["time"]\
              == 0]), reps))
    prod = np.empty(reps)

    prod[np.where(values[0] == 0)] = 1-np.mean(list(df["L1"][df["time"]\
                                      == 0]))
    prod[np.where(values[0] != 0)] = np.mean(list(df["L1"][df["time"]\
                                      == 0]))

    x = np.transpose(np.array([list(values[0]),[A_test[0]]*reps]))
    values[1] = L_model[1].predict(poly.fit_transform(x))

    p_v = sp.special.expit(values[1])
    values[1] = np.random.binomial(n=1, p = p_v)
    prod = prod*p_v

    x = np.transpose(np.array([list(values[1]),list(values[0]), \
        [A_test[1]]*reps, [A_test[0]]*reps]))
    values[2] = L_model[2].predict(poly.fit_transform(x))
    p_v = sp.special.expit(values[2])
    values[2] = np.random.binomial(n=1, p=p_v)
    prod = prod*p_v



    for jj in range(3, max_time+1):
```

```
        x = np.transpose(np.array([list(values[jj-1]),\
            list(values[jj-2]), list(values[jj-3]), [A_test[jj-1]]*reps,\
            [A_test[jj-2]]*reps, [A_test[jj-3]]*reps]))
        values[jj] = L_model[jj].predict(poly.fit_transform(x))

        p_v = sp.special.expit(values[jj])
        values[jj] = np.random.binomial(n=1, p=p_v)
        prod = prod*p_v

    if Y_full == "TRUE":
        Y_A = [A_test]*reps
        Y_L = np.array(values)
        Y_exp = np.array(Y_model.params[0])*([1.0]*reps) + np.sum(Y_A\
              *np.array([Y_model.params[i] for i in [1,4,6,8,10,12,\
              14,16,18,20,22,24]]), axis = 1)+np.sum([Y_model.params\
              [i] for i in [2,3,5,7,9,11,13,15,17,19,21,23]]*Y_L, \
              axis = 1)
        Y_exp = sp.special.expit(Y_exp)

    else:
        Y_A = [A_test*4]*reps
        Y_L = np.array([values[0], values[1], values[2], values[3], \
            values[4]])
        Y_exp = np.array(Y_model.params[0])*([1.0]*reps) + np.sum(Y_A\
              *np.array([Y_model.params[i] for i in [1,4,6,8]]), \
              axis = 1)+np.sum([Y_model.params[i] for i in [2,3,5,\
              7]]*Y_L, axis = 1)
        Y_exp = (np.exp(Y_exp)/(1+np.exp(Y_exp)))

    return(np.mean(prod*Y_exp))



########################################################################
##[FUNCTION] natural_course_test creates a second dataset from the
## models (L and Y) used in the g-formula to test their
## accuracy at modeling the underlying data (input df)
########################################################################
def natural_course_test(df):
    max_time = 11
    indiv = 10000
    results_mean_df = pd.DataFrame(columns = list(df))
    results_var_df = pd.DataFrame(columns = list(df))
    Y_model = Y_model_creation(df, max_time)
    L_model = covariate_model_creation(df, max_time)
    A_model = treatment_model_creation(df, max_time)
    poly = PolynomialFeatures(1)
    poly2 = PolynomialFeatures(1)
```

```python
new_df = pd.DataFrame(columns = ["indiv", "time", "A", "Y",  "L1"])
for ii in range(0, max_time+1):
    if ii == 0:
        L = np.random.choice(np.array(df["L1"][df["time"] == 0]), \
            indiv)

        A = A_model[ii].predict(sm.add_constant(L, has_constant=\
            'add'))

        temp_df = pd.DataFrame({"indiv": range(0, indiv), "time": \
                  [0.0]*indiv, "A": A, "Y": [float('nan')]*indiv, \
                  "L1":L})
        new_df = pd.concat([new_df, temp_df])

    elif ii == 1:
        y = np.transpose(np.array([new_df[new_df["time"] == 0].L1,\
            new_df[new_df["time"] == 0].A]))
        L = L_model[ii].predict(poly2.fit_transform(y))

        x = np.transpose(np.array([L, new_df[new_df["time"] == 0].L1\
            ,new_df[new_df["time"] == 0].A]))
        A = A_model[ii].predict(poly.fit_transform(x))

        temp_df = pd.DataFrame({"indiv": range(0, indiv), "time": \
                  [ii]*indiv, "A": A, "Y": [float('nan')]*indiv, \
                  "L1":L})
        new_df = pd.concat([new_df, temp_df])

    elif ii == 2:

        y = np.transpose(np.array([new_df[new_df["time"] == ii-1].L1,\
            new_df[new_df["time"] == ii-2].L1, new_df[new_df["time"]\
            == ii-1].A, new_df[new_df["time"] == ii-2].A]))
        L = L_model[ii].predict(poly2.fit_transform(y))

        x = np.transpose(np.array([L, new_df[new_df["time"] == ii-1]\
            .L1,new_df[new_df["time"] == ii-2].L1,new_df[new_df\
            ["time"] == ii-1].A,new_df[new_df["time"] == ii-2].A]))
        A = A_model[ii].predict(poly.fit_transform(x))
        temp_df = pd.DataFrame({"indiv": range(0, indiv), "time": \
                  [ii]*indiv,"A": A, "Y": [float('nan')]*indiv, \
                  "L1":L})
        new_df = pd.concat([new_df, temp_df])

    else:

        y = np.transpose(np.array([new_df[new_df["time"] == ii-1].L1,\
```

```
                new_df[new_df["time"] == ii-2].L1, new_df[new_df["time"]\
                == ii-3].L1, new_df[new_df["time"] == ii-1].A, new_df[new\
                _df["time"] == ii-2].A, new_df[new_df["time"] == ii-3]\
                .A]))

            L = L_model[ii].predict(poly2.fit_transform(y))

            x = np.transpose(np.array([L, new_df[new_df["time"] == ii-1].\
                L1, new_df[new_df["time"] == ii-2].L1,\
                new_df[new_df["time"] == ii-1].A, new_df[new_df["time"]==\
                ii-2].A, new_df[new_df["time"] == ii-3].A]))
            A = A_model[ii].predict(poly.fit_transform(x))

            temp_df = pd.DataFrame({"indiv": range(0, indiv), "time": \
                      [ii]*indiv, "A": A, "Y": [float('nan')]*indiv,  \
                      "L1":L})
            new_df = pd.concat([new_df, temp_df])
    for kk in range(1,max_time+1):
        new_df["L1_"+str(kk)] = new_df.L1.shift(kk)
        new_df["A_"+str(kk)] = new_df.A.shift(kk)

    small_df = new_df[new_df["time"] == 11.0]
    cols = ['Y']+ ["time"] + ["indiv"]  + [col for col in small_df if \
           col not in  ['Y', "time", "indiv"]]
    small_df = small_df[cols]
    p_Y = np.sum(Y_model.params*sm.add_constant(small_df.ix[:,3:]), \
          axis = 1)
    new_df.Y[new_df["time"] == 11.0]= np.random.binomial(n=1, p = \
                                      sp.special.expit(p_Y)).astype(int)

    return(new_df)



#########################################################################
##[FUNCTION] pi_function creates the w_m function given the following:
## the alpha model of A_{m,i}, the dataframe, the time (m), and an
## indicator of whether this is the correct or incorrect model
#########################################################################

def pi_function(m, alpha_model, df, indiv, alpha_wrong):
    product = [1]*indiv
    for jj in range(3, m+1):
        if alpha_wrong[jj] == False:
            x = alpha_model[jj].predict(sm.add_constant(df[df.time ==\
                jj][["L1", "L1_1", "L1_2", "A_1", "A_2"]], \
                has_constant='add'))
        else:
```

```
                x = alpha_model[jj].predict(sm.add_constant(df[df.time ==\
                    jj][["L1_3","A_3"]], has_constant='add'))
            product = product*x

        x = np.array(np.divide([1]*indiv, product))
        x[np.where(df[df.time == m]["A_1"] == 0.0)] = 1 - x[np.where(df\
                [df.time == m]["A_1"] == 0.0)]
        return(x)

##########################################################################
##[FUNCTION] alpha_model_creation creates the logistic regression
## for the observed treatment (A) data from the current and previous
## covariates and the previous treatments (A) over all time periods and
## individuals
##########################################################################


def alpha_model_creation(df, wrong):
    temp_df = df[df["time"]>2.0]
    if wrong == True:
        alpha_model = sm.Logit(np.asarray(temp_df.A),np.asarray(sm.add\
                      _constant(temp_df[["L1_3", "A_3"]], has_constant\
                      ='add'))).fit()

    else:
        alpha_model = sm.Logit(np.asarray(temp_df.A),np.asarray(sm.add\
                      _constant(temp_df[["L1", "L1_1", "L1_2", "A_1", \
                      "A_2"]], has_constant='add'))).fit()

    return(alpha_model)


##########################################################################
##[FUNCTION] DR_estimate_creation calculates the causal effect for a
## given treatment of interest (test_value), including an indicator
## of whether the correct or incorrect model is being used
##########################################################################

def DR_estimate_creation_bin_time(test_value, max_time, df, indiv, \
    wrong_alpha_model, wrong_s_model, alpha_model, int_term):

    A_test = [test_value]*indiv
    model_df = pd.DataFrame(columns = ["time", "beta_0", "beta_1",
                 "beta_2","beta_3", "beta_4", "beta_5", "beta_6", "phi"])
    time_counter = max_time+1
    T = df[df.time == max_time]["Y"]

    poly = sk.preprocessing.PolynomialFeatures(interaction_only = True)
```

```
while(time_counter > 3.0):
    time_df = df.loc[df.time == time_counter-1]
    pi = pi_function(time_counter-1, alpha_model, df, indiv, \
         wrong_alpha_model)
    time_df["pi"] = pi
    if wrong_s_model[time_counter-1] == True:
        train_columns = list(time_df)[0:2] + list(time_df)[12:14]\
        +["pi"]
        reg_columns = '+'.join(map(str, np.append(list(time_df)\
                    [0:2],np.append(list(time_df)[12:14], ["pi"]))))
    else:
        train_columns = list(time_df)[0:2] + list(time_df)[6:10]+\
                        ["pi"]
        if int_term == True:
            x = list(itertools.combinations(np.append(list(time_df)\
                [0:2], list(time_df)[6:10]), 2))
            y = ['*'.join(map(str, np.array([x[i][0], x[i][1]]))) \
                 for i in range(len(x))]
            z = '+'.join(map(str, y))
            reg_mid_columns = '+'.join(map(str, np.append(list(\
                              time_df)[0:2], np.append(list(time_df)\
                              [6:10],["pi"]))))
            reg_columns = '+'.join(map(str, np.array([reg_mid_
                          columns,z])))
        else:
            reg_columns = '+'.join(map(str, np.append(list(time_df)\
                          [0:2],np.append(list(time_df)[6:10],["pi"]))))
    time_df = time_df.astype(float)

    formula = "T~"+reg_columns
    glm_model = smf.glm(formula = formula, data = time_df, family=\
                sm.families.Binomial(link=sm.families.links.logit))
    try:
         glm_results = glm_model.fit()
    except Exception as ex:
        return(float("nan"), float("nan"))

    pi2 = pi_function(time_counter-2, alpha_model, df, indiv, \
          wrong_alpha_model)

    time_df["A"] = np.array(A_test)

    if test_value == 1:
        if wrong_alpha_model[time_counter-1] == True:
            pi2 = pi2*alpha_model[time_counter-1].predict(\
                  sm.add_constant(time_df[["L1_3", "A_3"]], \
                  has_constant = "add"))
```

```python
            else:
                pi2 = pi2*alpha_model[time_counter-1].predict(\
                    sm.add_constant(time_df[["L1", "L1_1", "L1_2", \
                    "A_1", "A_2"]], has_constant = "add"))

        elif test_value == 0:
            if wrong_alpha_model[time_counter-1] == True:
                pi2 = pi2*(1-alpha_model[time_counter-1].predict(\
                      sm.add_constant(time_df[["L1_3", "A_3"]], \
                      has_constant = "add")))
            else:
                pi2 = pi2*(1-alpha_model[time_counter-1].predict(\
                      sm.add_constant(time_df[["L1", "L1_1", "L1_2",\
                      "A_1", "A_2"]], has_constant = "add")))


        time_df["pi"] = pi2
        T = glm_results.predict(time_df[train_columns])
        time_counter = time_counter-1

    values = np.array([np.mean(df.Y), np.mean(df.A), np.mean(df.L1), \
             np.mean(df.U), pearsonr(df.Y[df.time == 11], \
             df.A[df.time == 11])[0], pearsonr(df.Y[df.time == 11], \
             df.L1[df.time == 11])[0], pearsonr(df.Y[df.time == 11], \
             df.U[df.time == 11])[0], pearsonr(df.A, df.L1)[0], \
             pearsonr(df.U, df.L1)[0], pearsonr(df.A, df.U)[0]])
    return(np.nanmean(T), values)
```