AppendixVIII

March 27, 2020

[ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from model import Fuzzification, InferenceEngine
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

plt.rcParams['font.weight'] = 'bold'
plt.rcParams['font.size'] = 12
plt.rcParams['axes.labelweight'] = 'bold'
plt.rcParams['lines.linewidth'] = 2
plt.rcParams['axes.titleweight'] = 'bold'

class Fuzzification:
    dx = 0.001 # dx for linguistic variable function
    nd = 3
    #N = 1000

    @staticmethod
    def norm_type(iv,typeofnorm,discourse):

        if typeofnorm == 'lower_is_better':
            x = Fuzzification.trapezoid(z= discourse,
                                        c_l=iv.min(),
                                        tc=iv.min(),
                                        Tc=iv.min(),
                                        c_u=iv.max());

        elif typeofnorm == 'middle_is_better':
            x = Fuzzification.trapezoid(z= discourse,
                                        c_l=iv.min(),
                                        tc=iv.mean(),
                                        Tc=iv.mean(),
                                        c_u=iv.max());

        elif typeofnorm == 'higher_is_better':
```

```
            x = Fuzzification.trapezoid(z=discourse,
                                        c_l=iv.min(),
                                        tc=iv.max(),
                                        Tc=iv.max(),
                                        c_u=iv.max())
        else:
            raise 'Error'
        return x

    @classmethod
    def trapezoid(cls,z,c_l,tc,Tc,c_u):
        """
        Trapezoid function that can be used to create linguistic values or normalization curves.

        Rule #1 of Fuzzy System: Completeness of Inputs - Make sure z covers the full universe of discourse.
        """
#         assert c_l<=tc, 'input values dont make sense for trapezoid'
#         assert tc<=Tc, 'input values dont make sense for trapezoid'
#         assert Tc<=c_u, 'input values dont make sense for trapezoid'

        if (c_l==tc) and (c_l==Tc): # DECREASING LINE
            #print('decreasing line')
            out = np.piecewise(z,
                               [z<c_l, z>c_u, (z>=Tc)&(z<=c_u)],
                               [0    , 0    , lambda z: -z/(c_u-c_l) + c_u/(c_u-c_l)])

        elif (c_u==tc) and (c_u==Tc): # INCREASING LINE
            #print('increasing line')
            out = np.piecewise(z,
                               [z<c_l, z>c_u, (z>=c_l)&(z<=tc)],
                               [0    , 0    , lambda z: z/(c_u-c_l) - c_l/(c_u-c_l)])

        else: # TRAPEZOID AND TRIANGLE
            #print('trapezoid')
            out = np.piecewise(z,
                               [z<=c_l, z>=c_u, (z>c_l)&(z<tc)          ,(z>=tc)&(z<=Tc), (z>Tc)&(z<c_u)],
                               [0    , 0    , lambda z: (z-c_l)/(tc-c_l), 1          , lambda z: (c_u-z)/(c_u-Tc)])
        out = out.round(cls.nd) #getrid of annoying decimals
        return out
```

```
    @classmethod
    def wms(cls):
        """
        Linguistic Variable WMS
        """
        x= np.arange(0,1+cls.dx,cls.dx).round(cls.nd)

        medium_val = 0.7
        df = pd.DataFrame(data = {'w':Fuzzification.
↪trapezoid(x,0,0,0,medium_val),
                                  'm':Fuzzification.
↪trapezoid(x,0,medium_val,medium_val,1),
                                  's':Fuzzification.
↪trapezoid(x,medium_val,1,1,1)},
                          index = x)
        Fuzzification.check_ruspini_partition(df)
        Fuzzification.check_consistency(df)
        return df.round(cls.nd)

    @classmethod
    def vbbagvg(cls):
        """
        Linguistic Variable VBBAGVG
        """
        x=np.arange(0,1+cls.dx,cls.dx).round(cls.nd)
        d = 0.25
        data = {}
        for i,l in enumerate(['vb','b','a','g','vg']):
            data[l] =Fuzzification.trapezoid(x,d*(i-1),d*(i),d*(i),d*(i+1))
        df = pd.DataFrame(data = data,index = x)
        cls.check_ruspini_partition(df)
        cls.check_consistency(df)
        return df.round(cls.nd)

    @classmethod
    def elvllflifhhvheh(cls):
        """
        Linguistic Variable elvllflifhhvheh
        """
        x=np.arange(0,1+cls.dx,cls.dx).round(cls.nd)
        d = 0.125
        data = {}
        for i,l in enumerate(['el','vl','l','fl','i','fh','h','vh','eh']):
            data[l] =Fuzzification.trapezoid(x,d*(i-1),d*(i),d*(i),d*(i+1))
        df = pd.DataFrame(data = data,index = x)
        cls.check_ruspini_partition(df)
        cls.check_consistency(df)
```

```
        return df.round(cls.nd)

    @staticmethod
    def check_ruspini_partition(df):
        """
        Special case of Rule #2 of Fuzzy System: Consistency of Unions for WMS␣
↪and VBBAGVG
        """
        assert any(df.sum(axis='columns').apply(lambda x: round(x,2)==1)) ==␣
↪True , 'Ruspini Partition Not Satisfied'

    @staticmethod
    def check_consistency(df):
        """
        Rule #2 of Fuzzy System: Consistency of Unions - for any input, the␣
↪membership functions of all the fuzzy sets it belongs too should be less␣
↪than or equal to 1.
        """
        assert any(df.sum(axis='columns').apply(lambda x: round(x,2)<=1)) ==␣
↪True , 'Not Consistent'

    @staticmethod
    def primary(primary_indicator,basic_indicators,indicator_type):
        #construct linguistic value lookuptables
        wms = Fuzzification.wms()
        #vbbagvg = Fuzzification.vbbagvg()


        #primary indicator db (the right excel tab)
        pi_db = pd.read_excel('./databases/indicator_db.xlsx',␣
↪sheet_name=primary_indicator).round(Fuzzification.nd)

        # CONSTRUCT NORMALIZATION CURVES
        curves = []
        for basic_indicator in basic_indicators.keys():
            assert pi_db[pi_db[indicator_type]==basic_indicator].shape[1] == 9␣
↪, 'Warning, not expected shape' #check,sensitve to number of basic␣
↪indicators, CHANGE ME
            basic_df = pi_db[pi_db[indicator_type]==basic_indicator]
            #print(basic_indicator)
            iv = basic_df['intensive_value']

            discourse = np.arange(iv.min(),iv.max()+Fuzzification.
↪dx,Fuzzification.dx).round(Fuzzification.nd)
            x = Fuzzification.norm_type(iv=iv,
```

```
                                        ␣
↪typeofnorm=basic_indicators[basic_indicator],
                                         discourse=discourse)

            ncurve = pd.Series(data=x.round(Fuzzification.nd),
                               index=discourse.round(Fuzzification.nd),
                               name=basic_indicator)
            curves.append(ncurve)

        # PLOT/SAVE NORMALIZATION CURVES
        for curve in curves:
            fig = plt.figure(figsize=(10,5));plt.title('Normalization Curve:␣
↪{}'.format(curve.name));
            curve.plot();
            plt.grid();plt.xlabel('z [{}]'.
↪format(pi_db[pi_db[indicator_type]==curve.name].iloc[0]['intensive_units']));
↪plt.ylabel('x'); #plt.legend(basic_indicators.keys());
            fig.savefig('./outputs/normalization_curves/{}.png'.format(curve.
↪name), dpi=300, bbox_inches='tight')


        #FUZZIFICATION
        frames = []
        for curve in curves:
            df_base = pi_db[pi_db[indicator_type]==curve.name].
↪drop(['raw_value','raw_value_units','intensive_units','source'],axis='columns').
↪reset_index(drop=True) #reset index so that it concats properly
            z = df_base['intensive_value'].values
            x = curve.loc[z] # pass through normalization curve
            wms_v = wms.loc[x.values].reset_index(); #reset index to make␣
↪concat work
            df_out = pd.concat([df_base,wms_v],axis='columns')
            assert df_base.shape[0] == df_out.shape[0], 'different amount of␣
↪rows, error'
            frames.append(df_out)
        fuzz = pd.concat(frames).reset_index(drop=True)

        #INFERENCE
        frames2=[]
        for company in fuzz['company'].unique():
            frames = []
            for year in fuzz['year'].unique():
                company_year = fuzz[(fuzz['company'] == company) &
                                    (fuzz['year'] == year)]
                #print(year,company)
```

```
                b_indicators =␣
↪[company_year[company_year[indicator_type]==indicator].iloc[0] for indicator␣
↪in basic_indicators]

                frames.append(InferenceEngine.b_s(b_indicators)) #apply␣
↪inference engine
            frames2.append(pd.concat(frames,axis='columns').T)
        secondary = pd.concat(frames2).reset_index(drop=True)
        return secondary



class InferenceEngine:
    dx = 0.01 # dx for linguistic variable function
    nd = 2

    @staticmethod
    def osus():
        return None

    @staticmethod
    def defuzzification():
        return None

    #primary indicators

    @classmethod
    def s_p(cls,secondary_indicators,primary_ind_name,indictor_type):
        """
        SECONDARY TO PRIMARY INFERENCE ENGINE &
        PRIMARY TO OSUS

        *** ONLY FOR 2 secondary INDICATOR INPUTS
        """
        #checks
        assert all(i['company']==secondary_indicators[0]['company'] for i in␣
↪secondary_indicators), 'secondary indicators included dont have the same␣
↪company'
        secondary_ind_company = secondary_indicators[0]['company']


        assert all(i['year']==secondary_indicators[0]['year'] for i in␣
↪secondary_indicators), 'secondary indicators included dont have the same␣
↪year'
        secondary_ind_year = secondary_indicators[0]['year']

        #search the rule base for the secondary indicator
```

```
        rb = pd.read_excel('./databases/rulebase.xlsx',␣
↪sheet_name=primary_ind_name,skiprows=13)
        lval = []
        indices = []

        if len(secondary_indicators)==2: #if there are two secondary indicators
            for in1_lv in ['vb','b','a','g','vg']:
                for in2_lv in ['vb','b','a','g','vg']:
                    lval.append(secondary_indicators[0][in1_lv] *␣
↪secondary_indicators[1][in2_lv]) #LARSEN IMPLICATIONsensitive to the number␣
↪of indicators
                    rule = rb[(rb[secondary_indicators[0][indictor_type]] ==␣
↪in1_lv) &
                              (rb[secondary_indicators[1][indictor_type]] ==␣
↪in2_lv)] #selecting the right row from the rulebase, sensitive to the number␣
↪of indicators
                    indices.append(rule[primary_ind_name].iloc[0]) #

        elif len(secondary_indicators)==3: #if there are three secondary␣
↪indicators
            for in1_lv in ['vb','b','a','g','vg']:
                for in2_lv in ['vb','b','a','g','vg']:
                    for in3_lv in ['vb','b','a','g','vg']:
                        lval.append(secondary_indicators[0][in1_lv] *␣
↪secondary_indicators[1][in2_lv] * secondary_indicators[2][in3_lv]) #LARSEN␣
↪IMPLICATIONsensitive to the number of indicators
                        rule = rb[(rb[secondary_indicators[0][indictor_type]]␣
↪== in1_lv) &
                                  (rb[secondary_indicators[1][indictor_type]]␣
↪== in2_lv) &
                                  (rb[secondary_indicators[2][indictor_type]]␣
↪== in3_lv)] #selecting the right row from the rulebase, sensitive to the␣
↪number of indicators
                        indices.append(rule[primary_ind_name].iloc[0]) #
        else:
            print('WARNING, THER ARE SOME OTHER NUMBER OF secondary INDICATORS')

        s = pd.Series(data=lval,
                      index=indices).round(cls.nd)
        s = s.groupby(s.index).sum() #add up all similar vals

        s['primary'] = primary_ind_name
        s['year'] = secondary_ind_year
        s['company'] = secondary_ind_company
        return s
```

```
    #basic indicators to secondary indicators
    @classmethod
    def b_s(cls,basic_indicators):
        """
        BASIC TO SECONDARY INFERENCE ENGINE

        *** ONLY FOR 2 BASIC INDICATOR INPUTS
        """
        #checks
        assert all(i['company']==basic_indicators[0]['company'] for i in␣
↪basic_indicators), 'basic indicators included dont have the same company'
        secondary_ind_company = basic_indicators[0]['company']

        assert all(i['secondary']==basic_indicators[0]['secondary'] for i in␣
↪basic_indicators), 'basic indicators included dont have the same secondary␣
↪indicator'
        secondary_ind_name = basic_indicators[0]['secondary'] #status

        assert all(i['year']==basic_indicators[0]['year'] for i in␣
↪basic_indicators), 'basic indicators included dont have the same year'
        secondary_ind_year = basic_indicators[0]['year']

        #search the rule base for the secondary indicator
        rb = pd.read_excel('./databases/rulebase.xlsx',␣
↪sheet_name=secondary_ind_name,skiprows=13)
        lval = []
        indices = []

        if len(basic_indicators)==2: #if there are two basic indicators
            for in1_lv in ['w','m','s']:
                for in2_lv in ['w','m','s']:
                    lval.append(basic_indicators[0][in1_lv] *␣
↪basic_indicators[1][in2_lv]) #LARSEN IMPLICATIONsensitive to the number of␣
↪indicators
                    rule = rb[(rb[basic_indicators[0]['basic']] == in1_lv) &
                              (rb[basic_indicators[1]['basic']] == in2_lv)]␣
↪#selecting the right row from the rulebase, sensitive to the number of␣
↪indicators
                    indices.append(rule[secondary_ind_name].iloc[0]) #

        elif len(basic_indicators)==3: #if there are three basic indicators
            for in1_lv in ['w','m','s']:
                for in2_lv in ['w','m','s']:
                    for in3_lv in ['w','m','s']:
```

```
                        lval.append(basic_indicators[0][in1_lv] *␣
↪basic_indicators[1][in2_lv] * basic_indicators[2][in3_lv]) #LARSEN␣
↪IMPLICATIONsensitive to the number of indicators
                        rule = rb[(rb[basic_indicators[0]['basic']] == in1_lv) &
                                  (rb[basic_indicators[1]['basic']] == in2_lv) &
                                  (rb[basic_indicators[2]['basic']] == in3_lv)]␣
↪#selecting the right row from the rulebase, sensitive to the number of␣
↪indicators
                        indices.append(rule[secondary_ind_name].iloc[0]) #
        else:
            print('WARNING, THER ARE SOME OTHER NUMBER OF BASIC INDICATORS')

        s = pd.Series(data=lval,
                      index=indices).round(cls.nd)
        s = s.groupby(s.index).sum() #add up all similar vals

        s['secondary'] = secondary_ind_name
        s['year'] = secondary_ind_year
        s['company'] = secondary_ind_company
        return s[['secondary','company','year','vb','b','a','g','vg']]



##################### MAIN.PY ###############

struct = {'wealth':{
                    'status':{'roe':'middle_is_better',
                              'roa':'higher_is_better'},
                    'pressure':{'CF_CAPEX':'higher_is_better',
                                'dividend payout ratio':'middle_is_better'},
                    'response':{'debt to equity ratio':'middle_is_better',
                                'operating expenses':'lower_is_better',
                                'effective tax rate':'lower_is_better'}
                   },
          'ecos':{
                    'air':{'clean generation':'lower_is_better', #bc we want to␣
↪see a low percentage of coal
                           'CO2 emissions':'lower_is_better'},
                    'land':{'spills':'lower_is_better',
                            'solid waste':'lower_is_better',
                            'hazardous waste':'lower_is_better'},
                    'water':{'total water withdrawal':'lower_is_better',
                             'percent water consumed':'lower_is_better'}
                   },
          'hums':{
                    'health':{'fatalities':'lower_is_better', #bc we want to␣
↪see a low percentage of coal
```

```
                                'osha recordable rate (ticr)':'lower_is_better'},
                     'polic':{'lobbying spending':'lower_is_better',
                              'charitable giving':'higher_is_better'}
                    }
         }

#=============================================================================
#=============================================================================
print('Starting Basic -> Secondary')

frames2 = []
for primary_indicator in struct.keys():
#     if (primary_indicator == 'wealth') or (primary_indicator == 'ecos'):
#         continue # skip for now
    frames = []
    for secondary_indicator in struct[primary_indicator]:
        print('{}-{}'.format(primary_indicator ,secondary_indicator))
        frames.append(Fuzzification.primary(primary_indicator=primary_indicator,
                                            ␣
 ↪basic_indicators=struct[primary_indicator][secondary_indicator],
                                             indicator_type='basic'))
    secondary_agg = pd.concat(frames,axis='index').reset_index(drop=True)
    secondary_agg.to_excel('./outputs/{}_secondary_agg.xlsx'.
 ↪format(primary_indicator))
    frames2.append(secondary_agg)

#=============================================================================
#=============================================================================



#=============================================================================
print('Starting Secondary -> Primary')

frames3=[]
for primary_ind_name in struct.keys():

    print(primary_ind_name)
    secondary_agg = pd.read_excel('./outputs/{}_secondary_agg.xlsx'.
 ↪format(primary_ind_name))

    indicator_type = 'secondary'
    #INFERENCE
    frames2=[]
    for company in secondary_agg['company'].unique():
        frames = []
        for year in secondary_agg['year'].unique():
```

```
            company_year = secondary_agg[(secondary_agg['company'] == company) &
                                         (secondary_agg['year'] == year)]
            #print(year,company)
            s_indicators =␣
 ↪[company_year[company_year[indicator_type]==indicator].iloc[0] for indicator␣
 ↪in secondary_agg[indicator_type].unique()]

            frames.append(InferenceEngine.
 ↪s_p(s_indicators,primary_ind_name,indicator_type)) #apply inference engine
        frames2.append(pd.concat(frames,axis='columns').T)
    primary = pd.concat(frames2).reset_index(drop=True)
    frames3.append(primary)
primary_agg = pd.concat(frames3)

primary_agg.to_excel('./outputs/primary_agg.xlsx')
#================================================================================



#================================================================================
print('Starting Primary  -> OSUS')

primary_agg = pd.read_excel('./outputs/primary_agg.xlsx')

osus_ind_name = 'osus'
indicator_type = 'primary'
#INFERENCE
frames2=[]
for company in primary_agg['company'].unique():
    frames = []
    for year in primary_agg['year'].unique():
        company_year = primary_agg[(primary_agg['company'] == company) &
                                   (primary_agg['year'] == year)]
        #print(year,company)
        s_indicators = [company_year[company_year[indicator_type]==indicator].
 ↪iloc[0] for indicator in primary_agg[indicator_type].unique()]

        frames.append(InferenceEngine.
 ↪s_p(s_indicators,osus_ind_name,indicator_type)) #apply inference engine
    frames2.append(pd.concat(frames,axis='columns').T)

osus = pd.concat(frames2).reset_index(drop=True)
osus.to_excel('./outputs/osus_fuzz.xlsx')
#================================================================================


# ================================================================================
print('Defuzzification')
```

```
osus = pd.read_excel('./outputs/osus_fuzz.xlsx')
osus.index= pd.to_datetime(osus['year'].astype(str), format='%Y')
elvllflifhhvheh = Fuzzification.elvllflifhhvheh()


osus['crisp']= (elvllflifhhvheh['el'].idxmax()*osus['el'] +
                elvllflifhhvheh['vl'].idxmax()*osus['vl'] +
                elvllflifhhvheh['l'].idxmax()*osus['l'] +
                elvllflifhhvheh['fl'].idxmax()*osus['fl'] +
                elvllflifhhvheh['i'].idxmax()*osus['i'] +
                elvllflifhhvheh['fh'].idxmax()*osus['fh'] +
                elvllflifhhvheh['h'].idxmax()*osus['h'] +
                elvllflifhhvheh['vh'].idxmax()*osus['vh'] +
                elvllflifhhvheh['eh'].idxmax()*osus['eh'])/
 ↪osus[['el','vl','l','fl','i','fh','h','vh','eh']].sum(axis='columns')

crisp = osus[['company','crisp']]
crisp.to_excel('./outputs/osus_crisp.xlsx')

fig, ax = plt.subplots(figsize=(8,4))
for idx, gp in crisp.groupby('company'):
    gp.plot(ax=ax,label=idx)
plt.grid(); plt.xlabel('Year'); plt.ylabel('OSUS');plt.legend(crisp.
 ↪groupby('company').indices)
fig.savefig('./outputs/final_output.png', dpi=300, bbox_inches='tight')

print('COMPLETE')
#==============================================================================
```