Challenge: When Will the Sakura Bloom?

Problem definition: Basics of the Sakura Bloom-cycle

In a year, sakura trees basically go through 4 phases: energy production, hibernation, growth, and of course flowering. These phases roughly follow the seasons, but not exactly.

Production phase： Initial development of the buds（Summer-Fall）
Hibernation phase： Bud growth stops while the tree goes into hibernation（Late Fall-Winter）
Growth phase： Buds once again continue to grow when the tree comes out of its winter hibernation（Late Winter-Spring）
Flowering phase： The buds finally bloom in spring (as climate conditions allow), once they have been able to fully develop.（Spring）

Each year, near the end of winter but before the trees finally bloom, the hibernation period ends. The sakura that rested through the winter once gain become metabolically active, and the buds continue to grow (though we may not immediately notice when this happens.) However, the cycle is not simply clockwork- for example, in places where the temperature is above 20℃ year-round, the trees are unable to hibernate sufficiently, and thus cannot blossom.

In this challenge, we have outlined the basic mechanism by which the sakura reach their eventual bloom-date. We consider building a bloom-date prediction model for the case of sakura in Tokyo, with the data split as follows:

Test years： 1966, 1971, 1985, 1994, and 2008
Training years: 1961 to 2017 (Excluding the test years)

You should fit the model to the data from the training years, then use the model to predict the bloom-date for each of the test years.

Abstract:

Based on different features of weather data of Tokyo, the blooming date of Sakura flower(cherry blossom) is to be predicted by diffrent models and accuracies are to be compared based on r2 score. Along with two given models, an ANN model has been proposed here. Required processing and feature engineering has been done and explained with proper tuning of hyper parameters.

Problem 0-1: (5pts)

Acquire data of sakura blooming date (桜の開花日) for Tokyo from 1961 to 2018 using the Japanese Meteorological Agency website (気象庁).

Data collection:

Weather data(Tokyo) of everyday from January,1961 to March,2017 has been collected from Japanese Meteorological Agency website(https://www.jma.go.jp/jma/en/menu.html (https://www.jma.go.jp/jma/en/menu.html)) . The collected features in the dataset are-

1. Local pressure
2. Sea pressure
3. Precipitation data
4. Maximum,minimum and average temperature data
5. Sun hours
6. Average and minimum humidity
7. Blooming date of Sakura in Tokyo for each year

Importing libraries

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import r2_score, mean_squared_error
from sklearn.preprocessing import RobustScaler, MinMaxScaler
import tensorflow as tf
```

Functions

In [2]:
```
def UpdateDayCount(dframe,y):
    #Takes dataframe and list of years
    #Returns updated dataframe with numbered days for each year
    grouped_df = dframe.groupby('year')
    last =0
    for i in range(len(y)):
        sub_df = grouped_df.get_group(y[i])
        for j in range(sub_df.shape[0]):
            k = j+last
            dframe.loc[k,'day_count'] = j+1
        last = k+1
    return dframe
```

In [3]:
```
def ActualBD(dframe,y):
    actual_days=[]
    grouped = dframe.groupby('year')
    for i in range(len(y)):
        actual_day = 0
        tempo = grouped.get_group(y[i])
        total_days = tempo.shape[0]
        tempo = tempo.reset_index(drop=True)
        for p in range(total_days):
            if tempo.loc[p,'bloom']==0:
                actual_day = 0
            else:
                actual_day = tempo.loc[p,'day_count']
                break
        actual_days.append(actual_day)

    bd_df = pd.DataFrame({'years':y,'day:actual':actual_days})
    return bd_df
```

In [4]:
```
def SplitDataSet(dframe,t_years):
    #Takes total dataframe and values of 'test' years
    #Returns separate train and test dataframes
    train_dframe = dframe[~dframe['year'].isin(t_years)]
    train_dframe = train_dframe.reset_index(drop=True)
    test_dframe = dframe[dframe['year'].isin(t_years)]
    test_dframe = test_dframe.reset_index(drop=True)
    return train_dframe,test_dframe
```

In [5]:
```
def ThresholdDegreeRule(dframe,degree,y):
    #Takes a dataframe,Threshold temperature value for blooming and a list of years
    #Returns prediction of blooming date for each year
    b_count =[]
    b_month =[]
    b_day = []
    sum_values = []
    actual_days=[]
    grouped = dframe.groupby('year')
    for i in range(len(y)):
        actual_day = 0
        tempo = grouped.get_group(y[i])
        total_days = tempo.shape[0] #Needs to be counted before dropping January
        tempo =tempo[tempo.month != 1] #Excluding January to start from February 1st
        tempo = tempo.reset_index(drop=True)
        sum_deg = 0
        for j in range(total_days):
            day_no = tempo.loc[j,'day_count']
            sum_deg= sum_deg+ tempo.loc[j,'max temp']
            if sum_deg>degree:
               break
        b_count.append(day_no)
        for p in range(total_days):
            if tempo.loc[p,'bloom']==0:
                actual_day = 0
            else:
                actual_day = tempo.loc[p,'day_count']
                break
        actual_days.append(actual_day)
        sum_values.append(sum_deg)
    pred = pd.DataFrame({'years':y,'day:actual':actual_days,'day:predicted':b_count,'sum':sum_values})

    return pred
```

In [6]:
```
def AccMaxTempCounter(dframe,y):
    #Takes dataframe and list of years
    #Returns dataframe of accumulated maximum temperature for each year
    acc_temp_list = []
    grouped = dframe.groupby('year')
    for i in range(len(y)):
        tempo = grouped.get_group(y[i])
        total_days = tempo.shape[0]
        tempo = tempo[tempo.month != 1] #Excluding January because calculation starts from February 1st
        tempo = tempo.reset_index(drop=True)
        acc_temp = 0
        for j in range(total_days):
            if tempo.loc[j,'bloom'] == 0:
                acc_temp = acc_temp + tempo.loc[j,'max temp']
            else:
                acc_temp = acc_temp + tempo.loc[j,'max temp']
                break
        acc_temp_list.append(acc_temp)
    df = pd.DataFrame({'years':y,'AccMaxTempLimit':acc_temp_list})
    return df
```

In [7]:
```
def CalcTmean(acc_df,y):
    T = acc_df.loc[:,'AccMaxTempLimit'].sum(axis=0)
    Tmean = T/len(train_years)
    return Tmean
```

In [8]:
```
def CalcR2Score(actual,pred):
    R2= r2_score(actual,pred)
    return R2
```

In [9]:
```
def CalcMSE(actual,pred):
    mse = mean_squared_error(actual,pred)
    return mse
```

In [10]:
```
def CalcTf(dframe,y):
    avg_temp_list = []
    grouped = df.groupby('year')
    for i in range(len(y)):
        tempo = grouped.get_group(y[i])
        tempo = tempo.reset_index(drop = True)
        sum_temp =0
        for j in range(tempo.shape[0]):
            if tempo.loc[j,'month']<4:
                sum_temp = sum_temp + tempo.loc[j,'avg temp']
            else:
                break
        avg_temp_list.append(sum_temp/j) #Will be divided by j because j starts from 0
    avg_temp_df = pd.DataFrame({'YEARS':y,'AVG_TEMP3m': avg_temp_list})
    return avg_temp_df
```

In [11]:
```
def CalcDj(fi,L,Tf_dframe,y):
    #Takes values to calculate Dj for each year and year list
    #Returns Dj value of each year in a dataframe
    dj_list = []
    x1 = 136.75 -(7.689*fi)+(0.133*fi*fi)-(1.307*(np.log(L)))
    for i in range(Tf_dframe.shape[0]):
        q =Tf_dframe.loc[i,'AVG_TEMP3m']
        x2 = 0.144*q + 0.285*q*q
        dj = x1+x2
        dj = int(dj)
        #dj = round(dj)
        #dj = int(dj)
        dj_list.append(dj)
    Dj_df= pd.DataFrame({'Dj':dj_list,'YEARS':y})
    return Dj_df
```

In [12]:
```
def CalcDTSj(Ea,dj_df,act_bd_df,dframe,y):
    R = 8.314
    Ts = 290 #in Kelvin
    Ea = Ea*4184 #in joule
    dj_list = np.array(dj_df['Dj'])
    ABD_list = np.array(act_bd_df['day:actual'])
    DTSj_list = []

    grouped = dframe.groupby('year')
    for i in range(len(y)):
        DTSj = 0
        tempo = grouped.get_group(y[i])
        tempo = tempo.reset_index(drop=True)
        Dji = dj_list[i]
        BD = ABD_list[i]
        for j in range(Dji,BD+1):
            Tij=tempo.loc[j-1,'avg temp']
            Tij = Tij+273
            ts=Calc_ts(Ea,Tij,Ts,R)
            DTSj = DTSj +ts
            #DTSj = round(DTSj)
            #DTSj = int(DTSj)
        DTSj_list.append(DTSj)
    return DTSj_list
```

In [13]:
```
def Calc_ts(Ea,Tij,Ts,R):
    A = Tij-Ts
    B = Ea*A
    C= R*Tij*Ts
    D = B/C
    ts = np.exp(D)
    return ts
```

In [14]:
```
def FindingMSE(dframe,dj_df,actBD_df,DTSmean_df,y):
    #Calculated mean square error between actual and predicted values
    R = 8.314
    Ts = 290

    MSE_perEa = []
    Dj_list = np.array(dj_df['Dj'])  #This is per year
    ABD_list = np.array(actBD_df['day:actual']) #This is per year
    DTSi = DTSmean_df['DTSmean'].values.tolist()

    grouped = dframe.groupby('year')
    for i in range(len(Ea_list)):
        DTSmean_ = DTSi[i]
        Ea = Ea_list[i]*4184
        pred_BD =[]
        for i in range(len(y)):
            DTSj = 0
            tempo = grouped.get_group(y[i])
            tempo = tempo.reset_index(drop=True)
            Dji = Dj_list[i]
            for j in range(Dji,tempo.shape[0]):
                Tij=tempo.loc[j-1,'avg temp']
                Tij = Tij+273
                ts=Calc_ts(Ea,Tij,Ts,R)
                if DTSj<DTSmean_:
                    DTSj = DTSj+ts
                else:
                    BD_pred = j
                    break
            pred_BD.append(BD_pred)
        PRED_BD=np.array(pred_BD)
        #print('ABD_list:',ABD_list)
        #print('PRED_BD:',PRED_BD)
        #print(len(ABD_list),len(PRED_BD))
        #MSE = CalcMSE(ABD_list,PRED_BD)
        MSE = mean_squared_error(ABD_list,PRED_BD)
        #print(MSE)
        MSE_perEa.append(MSE)
    MSE_Ea_df = pd.DataFrame({'Ea value(Kcal)':Ea_list,'MSE Score':MSE_perEa})
    return MSE_Ea_df
```

In [15]:
```
def FindingEabyR2(dframe,dj_df,actBD_df,DTSmean_df,y):
    R = 8.314
    Ts = 290

    r2_perEa = []
    Dj_list = np.array(dj_df['Dj'])  #This is per year
    ABD_list = np.array(actBD_df['day:actual']) #This is per year
    DTSi = DTSmean_df['DTSmean'].values.tolist()

    grouped = dframe.groupby('year')
    for i in range(len(Ea_list)):
        DTSmean = DTSi[i]
        Ea = Ea_list[i]*4184
        pred_BD =[]
        for i in range(len(y)):
            DTSj = 0
            tempo = grouped.get_group(y[i])
            tempo = tempo.reset_index(drop=True)
            Dji = Dj_list[i]
            for j in range(Dji,tempo.shape[0]):
                Tij=tempo.loc[j-1,'avg temp']
                Tij = Tij+273
                ts=Calc_ts(Ea,Tij,Ts,R)
                if DTSj<DTSmean:
                    DTSj = DTSj+ts
                else:
                    BD_pred = j
                    break
            pred_BD.append(BD_pred)
        PRED_BD=np.array(pred_BD)
        #print('ABD_list:',ABD_list)
        #print('PRED_BD:',PRED_BD)
        #print(len(ABD_list),len(PRED_BD))
        r2 = CalcR2Score(ABD_list,PRED_BD)
        #print(MSE)
        r2_perEa.append(r2)
    r2_Ea_df = pd.DataFrame({'Ea value(Kcal)':Ea_list,'r2 Score':r2_perEa})
    return r2_Ea_df
```

In [59]:
```
def PREDByBestEaDTSmeas(dframe,dj_df,actBD_df,Ea,DTSmean,y):
    R = 8.314
    Ts = 290

    Dj_list = np.array(dj_df['Dj'])  #This is per year
    ABD_list = actBD_df['day:actual'].values.tolist() #This is per year


    grouped = dframe.groupby('year')
    Ea = Ea*4184
    pred_BD =[]
    for i in range(len(y)):
        DTSj = 0
        tempo = grouped.get_group(y[i])
        tempo = tempo.reset_index(drop=True)
        Dji = Dj_list[i]
        for j in range(Dji,tempo.shape[0]):
            Tij=tempo.loc[j-1,'avg temp']
            Tij = Tij+273
            ts=Calc_ts(Ea,Tij,Ts,R)
            if DTSj<DTSmean:
                DTSj = DTSj+ts
            else:
                BD_pred = j-1
                break
        pred_BD.append(BD_pred)
    pred_df = pd.DataFrame({'day:predicted':pred_BD,'day:actual':ABD_list,'years':y})

    return pred_df
```

In [17]:
```
def CreateFeat(dframe,y,S_M,E_M,feat_name):
    c_dframe=dframe.copy()
    grouped = c_dframe.groupby('year')
    year_values = []
    for i in range(len(y)):
        tempo = grouped.get_group(y[i])
        value_list = []
        for m in range(S_M,E_M+1):
            feat_sum = 0
            value =0
            sub_df =tempo[tempo.month == m]
            sub_df = sub_df.reset_index(drop=True)
            feat_sum = sub_df[feat_name].sum()
            value = feat_sum/sub_df.shape[0]
            value_list.append(value)
        year_values.append(value_list)
    new_feat = E_M-S_M+1
    col = list(range(S_M,E_M+1))
    new_df = pd.DataFrame(year_values,columns=col)
    new_df['year']=y
    return new_df
```

Preprocessing steps

In [18]:
```
#READING DATA
df = pd.read_csv('sakura.csv')
```

In [19]:
```
#DELETING UNNECESSARY COLUMNS
del df['serial']
#Adding columns
df['day_count']=0
df['actual BD']=0
#NECCESSARY INPUT VARIABLES
test_years = [1966,1971,1985,1994,2008]
```

In [20]:
```
all_years = df.year.unique().tolist()
```

In [21]:
```
df = UpdateDayCount(df,all_years)
```

In [22]:
```
ActualBD_df = ActualBD(df,all_years)
print(ActualBD_df)
```

	day:actual	years
0	91	1961
1	91	1962
2	91	1963
3	93	1964
4	92	1965
5	79	1966
6	89	1967
7	89	1968
8	96	1969
9	97	1970
10	89	1971
11	88	1972
12	90	1973
13	92	1974
14	88	1975
15	82	1976
16	81	1977
17	90	1978
18	82	1979
19	91	1980
20	85	1981
21	82	1982
22	90	1983
23	102	1984
24	93	1985
25	93	1986
26	82	1987
27	93	1988
28	79	1989
29	79	1990
30	89	1991
31	84	1992
32	83	1993
33	90	1994
34	90	1995
35	91	1996
36	80	1997
37	86	1998
38	83	1999
39	90	2000
40	82	2001
41	75	2002

```
42          86   2003
43          78   2004
44          90   2005
45          80   2006
46          79   2007
47          82   2008
48          80   2009
49          81   2010
50          87   2011
51          91   2012
52          75   2013
53          84   2014
54          82   2015
55          81   2016
56          80   2017
```

In [23]:
```
train_df,test_df=SplitDataSet(df,test_years)
```

In [24]:
```
train_years=train_df.year.unique().tolist()
```

In [25]:
```
ActualBD_tr = ActualBD(train_df,train_years)
ActualBD_test = ActualBD(test_df,test_years)
print(ActualBD_test)
```

```
   day:actual  years
0          79   1966
1          89   1971
2          93   1985
3          90   1994
4          82   2008
```

Missing value checking

In [26]:
```
#Checking if any of the year has any missing values
grouped = df.groupby('year')
for i in range(len(all_years)):
        tempo = grouped.get_group(all_years[i])
        if tempo.shape[0]!= 365:
            print(all_years[i],tempo.shape[0])
```

```
1964 366
1968 366
1972 366
1976 366
1980 366
1984 366
1988 366
1992 366
1996 366
2000 366
2004 366
2008 366
2012 366
2016 366
2017 90
```

1. Prediction using the "600 Degree Rule" (15pts total)

For a rough approximaton of the bloom-date, we start with a simple "rule-based" prediction model, called the "600 Degree Rule". The rule consists of logging the maximum temperature of each day, starting on February 1st, and sum these temperatures until the sum surpasses 600°C. The day that this happens is the predicted bloom-date. This 600°C threshold is used to easily predict bloom-date in various locations varies by location. However, for more precise predictions, it should be set differently for every location. In this challenge, we verify the accuracy of the "600 Degree Rule" in the case of Tokyo.

In [27]:
```
#PREDICTION ON TRAIN DATA
pred_600_tr=ThresholdDegreeRule(train_df,600,train_years)
print(pred_600_tr)
```

	day:actual	day:predicted	sum	years
0	91	84	614.6	1961
1	91	78	609.3	1962
2	91	86	609.8	1963
3	93	90	613.9	1964
4	92	90	615.0	1965
5	89	86	612.5	1967
6	89	85	605.3	1968
7	96	88	601.8	1969
8	97	90	600.6	1970
9	88	88	610.4	1972
10	90	84	610.2	1973
11	92	89	603.7	1974
12	88	88	607.9	1975
13	82	83	611.3	1976
14	81	84	602.0	1977
15	90	88	613.4	1978
16	82	80	603.1	1979
17	91	89	601.3	1980
18	85	85	600.1	1981
19	82	84	610.7	1982
20	90	85	602.8	1983
21	102	100	612.3	1984
22	93	92	614.3	1986
23	82	83	608.3	1987
24	93	89	610.0	1988
25	79	81	608.1	1989
26	79	81	605.4	1990
27	89	83	603.0	1991
28	84	83	601.1	1992
29	83	83	612.9	1993
30	90	84	608.0	1995
31	91	87	607.8	1996
32	80	80	614.6	1997
33	86	81	603.5	1998
34	83	80	604.1	1999
35	90	85	608.2	2000
36	82	81	609.0	2001
37	75	76	612.3	2002
38	86	86	603.7	2003
39	78	77	608.4	2004
40	90	85	608.4	2005
41	80	82	604.9	2006

```
42          79            78  609.0   2007
43          80            79  601.1   2009
44          81            82  601.8   2010
45          87            84  609.8   2011
46          91            89  614.3   2012
47          75            78  612.8   2013
48          84            84  607.6   2014
49          82            81  600.9   2015
50          81            78  603.0   2016
51          80            79  606.6   2017
```

In [29]:
```
actual_tr = np.array(pred_600_tr['day:actual'])
pred_tr_600 = np.array(pred_600_tr['day:predicted'])
R2_600 = CalcR2Score(actual_tr,pred_tr_600)
print('R2 Score for 600 degree rule in TRAIN DATA is:',R2_600)
```

```
R2 Score for 600 degree rule in TRAIN DATA is: 0.5841220511371663
```

In [30]:
```
#PREDICTION ON TEST DATA
pred_600_test = ThresholdDegreeRule(test_df,600,test_years)
print(pred_600_test)
```

```
   day:actual  day:predicted    sum  years
0          79             79  607.6   1966
1          89             86  605.6   1971
2          93             88  604.4   1985
3          90             87  608.9   1994
4          82             83  614.8   2008
```

In [31]:
```
actual_test = np.array(pred_600_test['day:actual'])
pred_test_600 = np.array(pred_600_test['day:predicted'])
R2_600 = CalcR2Score(actual_test,pred_test_600)
print('R2 Score for 600 degree rule in TEST DATA is:',R2_600)
```

```
R2 Score for 600 degree rule in TEST DATA is: 0.6793002915451896
```

Result of 600 Degree Rule:

1. R2 Score on test data is: 0.6793

Problem 1-1: (5pts)

From here-on, we refer to the bloom-date in a given year j as BD_j. For each year in the training data, calculate the accumulated daily maximum temperature from February 1st to the actual bloom-date BD_j, and plot this accumulated value over the training period. Then, average this accumulated value as T_{mean}, and verify whether we should use 600°C as a rule for Tokyo.

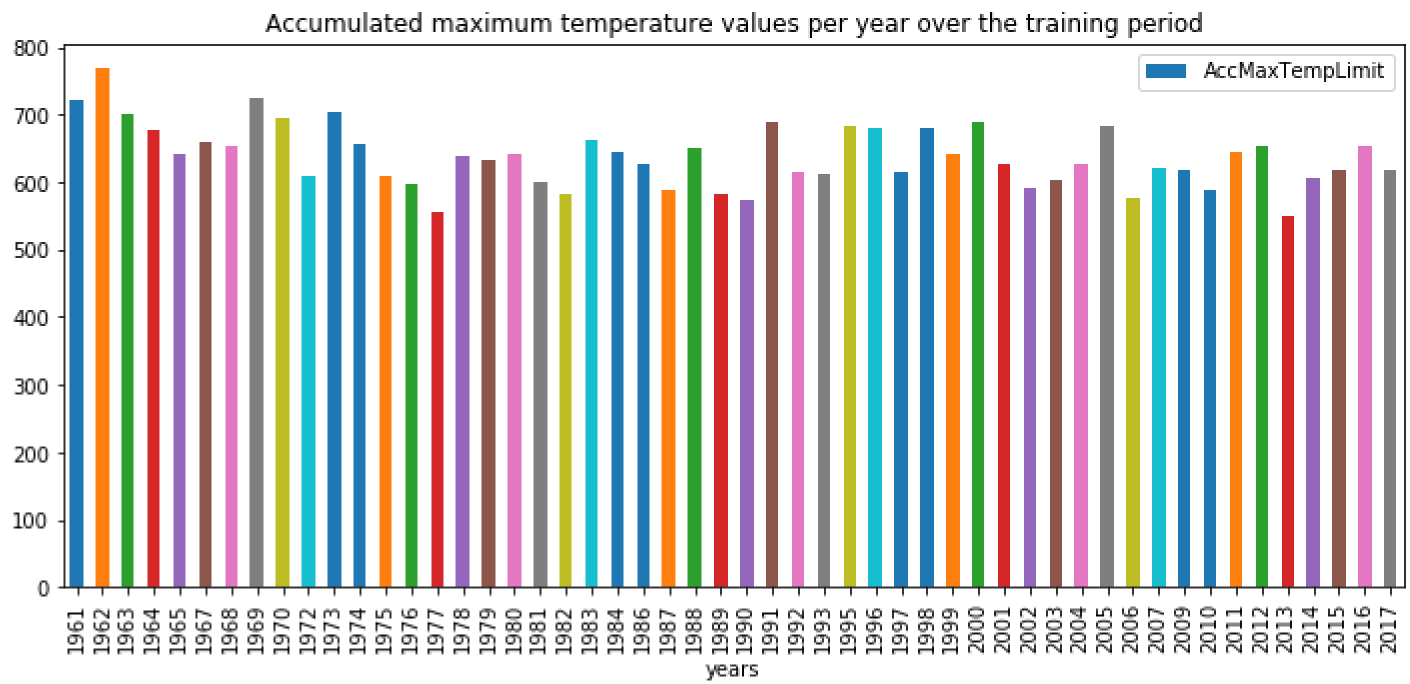
In [32]:
```
#Accumulated Max Temperature Calculation
acc_t_df_tr = AccMaxTempCounter(train_df,train_years)
print(acc_t_df_tr)
```

	AccMaxTempLimit	years
0	721.0	1961
1	768.1	1962
2	701.7	1963
3	676.6	1964
4	642.2	1965
5	660.8	1967
6	654.6	1968
7	724.0	1969
8	696.3	1970
9	610.4	1972
10	705.3	1973
11	658.0	1974
12	607.9	1975
13	596.7	1976
14	556.7	1977
15	640.1	1978
16	631.8	1979
17	640.6	1980
18	600.1	1981
19	581.5	1982
20	662.5	1983
21	643.7	1984
22	627.5	1986
23	588.7	1987
24	649.7	1988
25	583.1	1989
26	573.6	1990
27	688.2	1991
28	615.5	1992
29	612.9	1993
30	684.6	1995
31	680.1	1996
32	614.6	1997
33	679.2	1998
34	642.7	1999
35	689.7	2000
36	627.1	2001
37	591.7	2002
38	603.7	2003
39	627.9	2004
40	683.9	2005
41	577.2	2006

```
42          621.8   2007
43          616.6   2009
44          587.1   2010
45          646.1   2011
46          652.6   2012
47          550.6   2013
48          607.6   2014
49          617.9   2015
50          652.5   2016
51          619.5   2017
```

In [33]:
```
acc_t_df_tr.plot(x='years',y='AccMaxTempLimit',kind = 'bar',figsize=(12,5),title = 'Accumulated maximum tempe
rature values per year over the training period')
```

Out[33]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x121b41e1e10>
```



In [34]:
```
Tmean_tr = CalcTmean(acc_t_df_tr,train_years)
print(Tmean_tr)
```

```
638.3557692307693
```

Verification of 600 degree rule:

The calculated Tmean is 638.35 degree which is higher than 600 degree. So Different result is expected.

Problem 1-2: (10pts)

Use the average accumulated value T_{mean} calculated in 1-1 to predict BD_j for each test year, and show the error from the actual BD_j. Compare to the prediction results when 600°C is used a threshold value, and evaluate both models using the coefficient of determination (R^2 score).

In [35]:
```
pred_Tmean_tr=ThresholdDegreeRule(train_df,Tmean_tr,train_years)
print(pred_Tmean_tr)
```

	day:actual	day:predicted	sum	years
0	91	87	645.9	1961
1	91	81	641.7	1962
2	91	88	646.3	1963
3	93	92	656.5	1964
4	92	92	642.2	1965
5	89	88	647.0	1967
6	89	88	639.1	1968
7	96	91	644.4	1969
8	97	94	649.7	1970
9	88	90	646.5	1972
10	90	87	656.4	1973
11	92	91	644.5	1974
12	88	90	642.5	1975
13	82	86	645.4	1976
14	81	87	642.7	1977
15	90	90	640.1	1978
16	82	83	648.1	1979
17	91	91	640.6	1980
18	85	88	641.1	1981
19	82	87	646.7	1982
20	90	89	647.6	1983
21	102	102	643.7	1984
22	93	94	643.6	1986
23	82	85	639.2	1987
24	93	93	649.7	1988
25	79	84	649.0	1989
26	79	83	645.4	1990
27	89	86	649.7	1991
28	84	86	641.7	1992
29	83	85	646.7	1993
30	90	87	640.7	1995
31	91	89	638.6	1996
32	80	83	647.4	1997
33	86	84	645.5	1998
34	83	83	642.7	1999
35	90	88	651.2	2000
36	82	83	646.6	2001
37	75	78	649.0	2002
38	86	89	649.4	2003
39	78	79	638.4	2004
40	90	87	639.9	2005
41	80	85	651.8	2006

```
42          79              81  651.1   2007
43          80              82  650.1   2009
44          81              86  643.6   2010
45          87              87  646.1   2011
46          91              91  652.6   2012
47          75              80  649.8   2013
48          84              86  642.4   2014
49          82              84  647.3   2015
50          81              80  638.7   2016
51          80              82  650.2   2017
```

In [36]:
```
actual_tr = np.array(pred_Tmean_tr['day:actual'])
pred_tr_Tmean = np.array(pred_Tmean_tr['day:predicted'])
R2_600 = CalcR2Score(actual_tr,pred_tr_Tmean)
print('R2 Score for Tmean rule on TRAIN DATA is:',R2_600)
```

```
R2 Score for Tmean rule on TRAIN DATA is: 0.7225596835711259
```

In [37]:
```
pred_Tmean_test=ThresholdDegreeRule(test_df,Tmean_tr,test_years)
print(pred_Tmean_test)
```

```
   day:actual  day:predicted    sum  years
0          79             82  649.7   1966
1          89             88  644.8   1971
2          93             91  644.6   1985
3          90             90  656.6   1994
4          82             85  644.0   2008
```

In [37]:
```
actual_test = np.array(pred_Tmean_test['day:actual'])
pred_test_Tmean = np.array(pred_Tmean_test['day:predicted'])
R2_600 = CalcR2Score(actual_test,pred_test_Tmean)
print('R2 Score for Tmean rule on TEST DATA is:',R2_600)
```

```
R2 Score for Tmean rule on TEST DATA is: 0.8323615160349854
```

Comparison of 600 Degree Rule with Tmean:

R2 scrore on test data using calculated Tmean(638.36 degree) is 0.8323 which is much better than 600 degree rule. So it is wise to calculate Tmean from available dataset instead of using a predefined value.

2. Linear Regression Model: Transform to Standard Temperature (30pts total)

The year to year fluctuation of the bloom-date depends heavily upon the actual temperature fluctuation (not just the accumulated maximum). In order to get to a more physiologically realistic metric, Sugihara et al. (1986) considered the actual effect of temperature on biochemical activity. They introduced a method of "standardizing" the temperatures measured, according to the fluctuation relative to a standard temperature.

In order to make such a standardization, we apply two major assumptions, outlined below.

1) The Arrhenius equation:

The first assumption, also known in thermodynamics as the "Arrhenius equation", deals with chemical reaction rates and can be written as follows:

$$k = A \exp\left(-\frac{E_a}{RT}\right)$$

Basically, it says that each reaction has an activation energy, E_a and a pre-exponential factor A. Knowing these values for the particular equation, we can find the rate constant k if we know the temperature, T, and applying the universal gas constant, $R = 8.314[\mathrm{J/K \cdot mol}]$.

2) Constant output at constant temperature:

The second assumption, is simply that the output of a reaction is a simple product of the duration and the rate constant k, and that product is constant even at different temperatures.

$$tk = t'k' = t''k'' = \cdots = \mathrm{const}$$

Making the assumptions above, we can determine a "standard reaction time", t_s required for the bloom-date to occur. We can do so in the following way:

$$t_s = \exp\left(\frac{E_a(T_{i,j} - T_s)}{RT_{i,j}T_s}\right)$$

We define $T_{i,j}$ as the daily average temperature, and use a standard temperature of $T_s = 17^\circ\mathrm{C}$. For a given year j, with the last day of the hibernation phase set as D_j, we define the number of "transformed temperature days", DTS_J, needed to reach from D_j to the bloom-date BD_j with the following equation:

$$DTS_j = \sum_{i=D_j}^{BD_j} t_s = \sum_{i=D_j}^{BD_j} \exp\left(\frac{E_a(T_{i,j} - T_s)}{RT_{i,j}T_s}\right)$$

From that equation, we can find the average DTS for x number of years (DTS_{mean}) as follows:

$$\begin{aligned} DTS_{\text{mean}} &= \frac{1}{x}\sum_{j}^{x} DTS_j \\ &= \frac{1}{x}\sum_{j}^{x}\sum_{i=D_j}^{BD_j} \exp\left(\frac{E_a(T_{i,j} - T_s)}{RT_{i,j}T_s}\right) \end{aligned}$$

In this exercise, we assume that DTS_{mean} and E_a are constant values, and we use the data from the training years to fit these 2 constants. The exercise consists of 4 steps:

1. Calculate the last day of the hibernation phase D_j for every year j.
2. For every year j, calculate DTS_j as a function of E_a, then calculate the average (over training years) DTS_{mean} also as a function of E_a.
3. For every year j, and for every value of E_a, accumulate t_s from D_j and predict the bloom date BD_j^{pred} as the day the accumulated value surpasses DTS_{mean}. Calculate the bloom date prediction error as a function of E_a, and find the optimal E_a value that minimizes that error.
4. Use the previously calculated values of D_j, DTS_{mean}, and E_a to predict bloom-day on years from the test set.

In [38]:
```
#initializing constant values
Fi = 35.67 #Value of latitude [°N] for Tokyo
L = 4 #Distance of Tokyo from the nearest coastline [km]
```

In [39]:
```
#Calculating Tf
Tf_tr = CalcTf(train_df,train_years)
#print(Tf_tr)
#Calculating Dj
Dj_tr = CalcDj(Fi,L,Tf_tr,train_years)
print(Dj_tr)
```

	Dj	YEARS
0	39	1961
1	41	1962
2	38	1963
3	40	1964
4	38	1965
5	42	1967
6	43	1968
7	42	1969
8	38	1970
9	45	1972
10	44	1973
11	39	1974
12	40	1975
13	45	1976
14	40	1977
15	41	1978
16	50	1979
17	42	1980
18	41	1981
19	45	1982
20	44	1983
21	35	1984
22	39	1986
23	46	1987
24	45	1988
25	51	1989
26	48	1990
27	46	1991
28	48	1992
29	47	1993
30	45	1995
31	45	1996
32	50	1997
33	47	1998
34	48	1999
35	47	2000
36	45	2001
37	55	2002
38	44	2003
39	50	2004
40	45	2005
41	45	2006

```
42  54   2007
43  50   2009
44  47   2010
45  43   2011
46  42   2012
47  49   2013
48  47   2014
49  46   2015
50  48   2016
51  45   2017
```

Problem 2-1: (5pts)

According to Hayashi et al. (2012), the day on which the sakura will awaken from their hibernation phase, D_j, for a given location, can be approximated by the following equation:

$$D_j = 136.75 - 7.689\phi + 0.133\phi^2 - 1.307\ln L + 0.144T_F + 0.285T_F^2$$

where ϕ is the latitude [°N], L is the distance from the nearest coastline [km], and T_F is that location's average temperature [°C] over the first 3 months of a given year. In the case of Tokyo, $\phi = 35^\circ 40'$ and $L = 4\mathrm{km}$.

Find the D_j value for every year j from 1961 to 2017 (including the test years), and plot this value on a graph.

(In Problem 1, we had assumed a D_j of February 1st.)

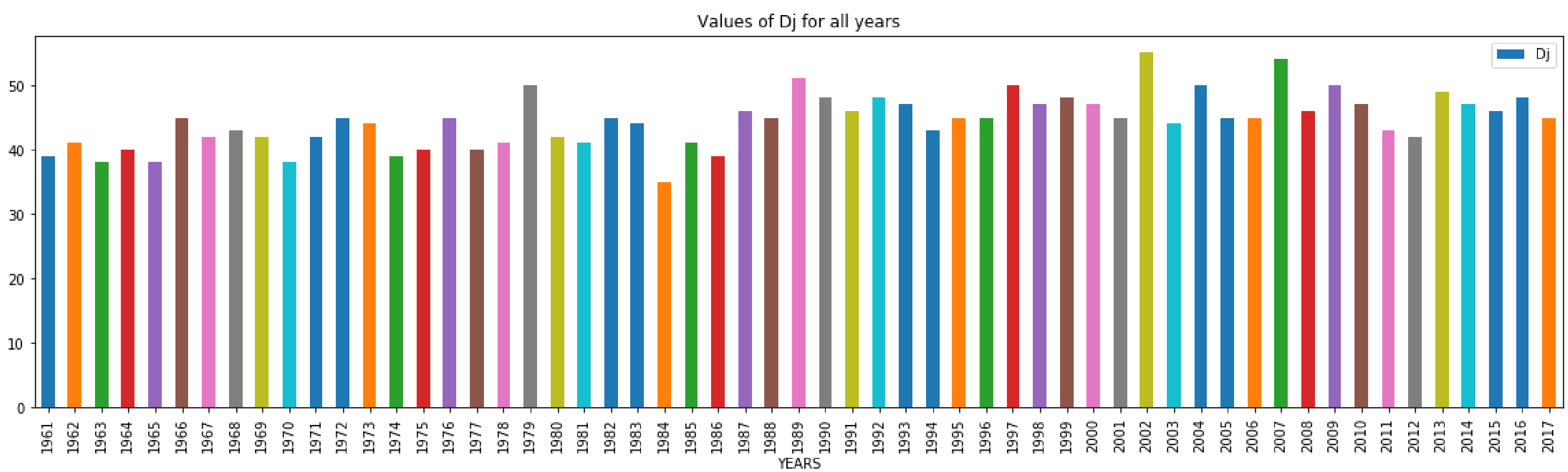
In [40]:
```
Tf_all = CalcTf(df,all_years)
Dj_all = CalcDj(Fi,L,Tf_all,all_years)
print(Dj_all)
```

	Dj	YEARS
0	39	1961
1	41	1962
2	38	1963
3	40	1964
4	38	1965
5	45	1966
6	42	1967
7	43	1968
8	42	1969
9	38	1970
10	42	1971
11	45	1972
12	44	1973
13	39	1974
14	40	1975
15	45	1976
16	40	1977
17	41	1978
18	50	1979
19	42	1980
20	41	1981
21	45	1982
22	44	1983
23	35	1984
24	41	1985
25	39	1986
26	46	1987
27	45	1988
28	51	1989
29	48	1990
30	46	1991
31	48	1992
32	47	1993
33	43	1994
34	45	1995
35	45	1996
36	50	1997
37	47	1998
38	48	1999
39	47	2000
40	45	2001
41	55	2002

```
42  44   2003
43  50   2004
44  45   2005
45  45   2006
46  54   2007
47  46   2008
48  50   2009
49  47   2010
50  43   2011
51  42   2012
52  49   2013
53  47   2014
54  46   2015
55  48   2016
56  45   2017
```

In [41]:
```
Dj_all.plot(y='Dj',x='YEARS',kind='bar',figsize=(20,5),title='Values of Dj for all years')
```

Out[41]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x121b46475f8>
```



Problem 2-2: (10pts)

Calculate DTS_j for each year j in the training set for discrete values of E_a, varying from 5 to 40kcal ($E_a = 5, 6, 7, \cdots, 40\,\mathrm{kcal}$), and plot this DTS_j against E_a. Also calculate the average of DTS_j over the training period, and indicate it on the plot as DTS_{mean}. Pay attention to the units of **every parameter** ($T_{i,j}$, E_a, ...) in the equation for t_s.

In [42]:
```
Ea_list = list(range(5,41)) #Since Ea varies from 5 Kcal to 40 Kcal
print(Ea_list)
```

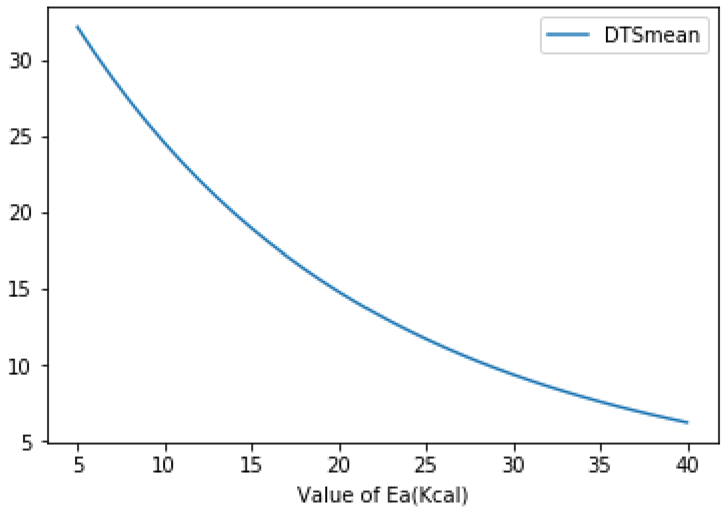
```
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 3
3, 34, 35, 36, 37, 38, 39, 40]
```

In [43]:
```
DTSj_perEa = []
DTSmean_perEa =[]
DTS_Ea_allyear =[]
for Ea in Ea_list:
    DTSj = CalcDTSj(Ea,Dj_tr,ActualBD_tr,train_df,train_years)
    #DTSj_perEa.append(DTSj)
    #print('DTSj:',DTSj)
    DTS_Ea_allyear.append(DTSj)
    DTSmean = (sum(DTSj)/len(train_years))
    DTSmean_perEa.append(DTSmean)
DTSmean_df_tr = pd.DataFrame({'DTSmean':DTSmean_perEa,'Value of Ea(Kcal)':Ea_list})
print(DTSmean_df_tr)
```

	DTSmean	Value of Ea(Kcal)
0	32.207353	5
1	30.487368	6
2	28.871963	7
3	27.354288	8
4	25.927966	9
5	24.587061	10
6	23.326045	11
7	22.139768	12
8	21.023433	13
9	19.972567	14
10	18.983002	15
11	18.050851	16
12	17.172491	17
13	16.344539	18
14	15.563839	19
15	14.827447	20
16	14.132613	21
17	13.476768	22
18	12.857516	23
19	12.272614	24
20	11.719968	25
21	11.197622	26
22	10.703743	27
23	10.236621	28
24	9.794654	29
25	9.376342	30
26	8.980284	31
27	8.605165	32
28	8.249756	33
29	7.912905	34
30	7.593532	35
31	7.290625	36
32	7.003235	37
33	6.730474	38
34	6.471507	39
35	6.225551	40

In [44]:
```
DTSmean_df_tr.plot(x='Value of Ea(Kcal)',y='DTSmean')
```

Out[44]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x121b4184978>
```



In [45]:
```
DTS_Ea_allyear=np.array(DTS_Ea_allyear).T
```

In [46]:
```
dts_df_plot = pd.DataFrame(DTS_Ea_allyear,index = train_years,columns = Ea_list)
```

In [47]:
```
dts_df_plot=dts_df_plot.transpose()
```

In [48]:
```
print(dts_df_plot)
```

	1961	1962	1963	1964	1965	1967 \
5	38.897696	38.079014	39.562942	39.134908	39.483797	36.109247
6	36.605456	35.947260	37.229751	36.747995	36.969676	34.186211
7	34.461601	33.944242	35.051627	34.524604	34.621374	32.389306
8	32.455942	32.061744	33.017677	32.453076	32.427605	30.709303
9	30.579019	30.292100	31.117802	30.522609	30.377874	29.137708
10	28.822050	28.628152	29.342636	28.723198	28.462417	27.666698
11	27.176876	27.063214	27.683487	27.045571	26.672153	26.289066
12	25.635917	25.591045	26.132296	25.481134	24.998630	24.998168
13	24.192131	24.205814	24.681582	24.021921	23.433986	23.787876
14	22.838973	22.902074	23.324403	22.660549	21.970900	22.652537
15	21.570359	21.674736	22.054315	21.390167	20.602562	21.586929
16	20.380633	20.519042	20.865336	20.204424	19.322628	20.586230
17	19.264535	19.430547	19.751911	19.097429	18.125193	19.645985
18	18.217172	18.405092	18.708883	18.063714	17.004760	18.762074
19	17.233993	17.438792	17.731461	17.098207	15.956207	17.930684
20	16.310764	16.528010	16.815192	16.196198	14.974764	17.148289
21	15.443543	15.669347	15.955943	15.353317	14.055986	16.411620
22	14.628660	14.859619	15.149870	14.565505	13.195733	15.717652
23	13.862701	14.095849	14.393401	13.828993	12.390147	15.063579
24	13.142483	13.375251	13.683215	13.140280	11.635632	14.446798
25	12.465044	12.695216	13.016224	12.496113	10.928834	13.864897
26	11.827624	12.053300	12.389558	11.893471	10.266630	13.315632
27	11.227649	11.447216	11.800545	11.329544	9.646105	12.796923
28	10.662722	10.874821	11.246701	10.801723	9.064541	12.306836
29	10.130607	10.334107	10.725714	10.307579	8.519404	11.843571
30	9.629221	9.823193	10.235432	9.844856	8.008331	11.405455
31	9.156619	9.340315	9.773852	9.411455	7.529114	10.990933
32	8.710988	8.883820	9.339110	9.005422	7.079696	10.598554
33	8.290635	8.452158	8.929470	8.624942	6.658156	10.226969
34	7.893982	8.043875	8.543313	8.268324	6.262702	9.874917
35	7.519554	7.657609	8.179133	7.933993	5.891658	9.541227
36	7.165974	7.292078	7.835527	7.620485	5.543463	9.224804
37	6.831957	6.946084	7.511185	7.326436	5.216657	8.924626
38	6.516302	6.618499	7.204889	7.050576	4.909878	8.639738
39	6.217887	6.308264	6.915501	6.791722	4.621852	8.369250
40	5.935662	6.014388	6.641960	6.548771	4.351390	8.112330

	1968	1969	1970	1972	...	2007 \
5	35.377918	41.134654	43.351901	33.263209	...	20.343955
6	33.480384	38.888086	40.673474	31.495383	...	19.387851
7	31.702390	36.788025	38.176369	29.835122	...	18.482623
8	30.035621	34.823932	35.847538	28.275414	...	17.625427

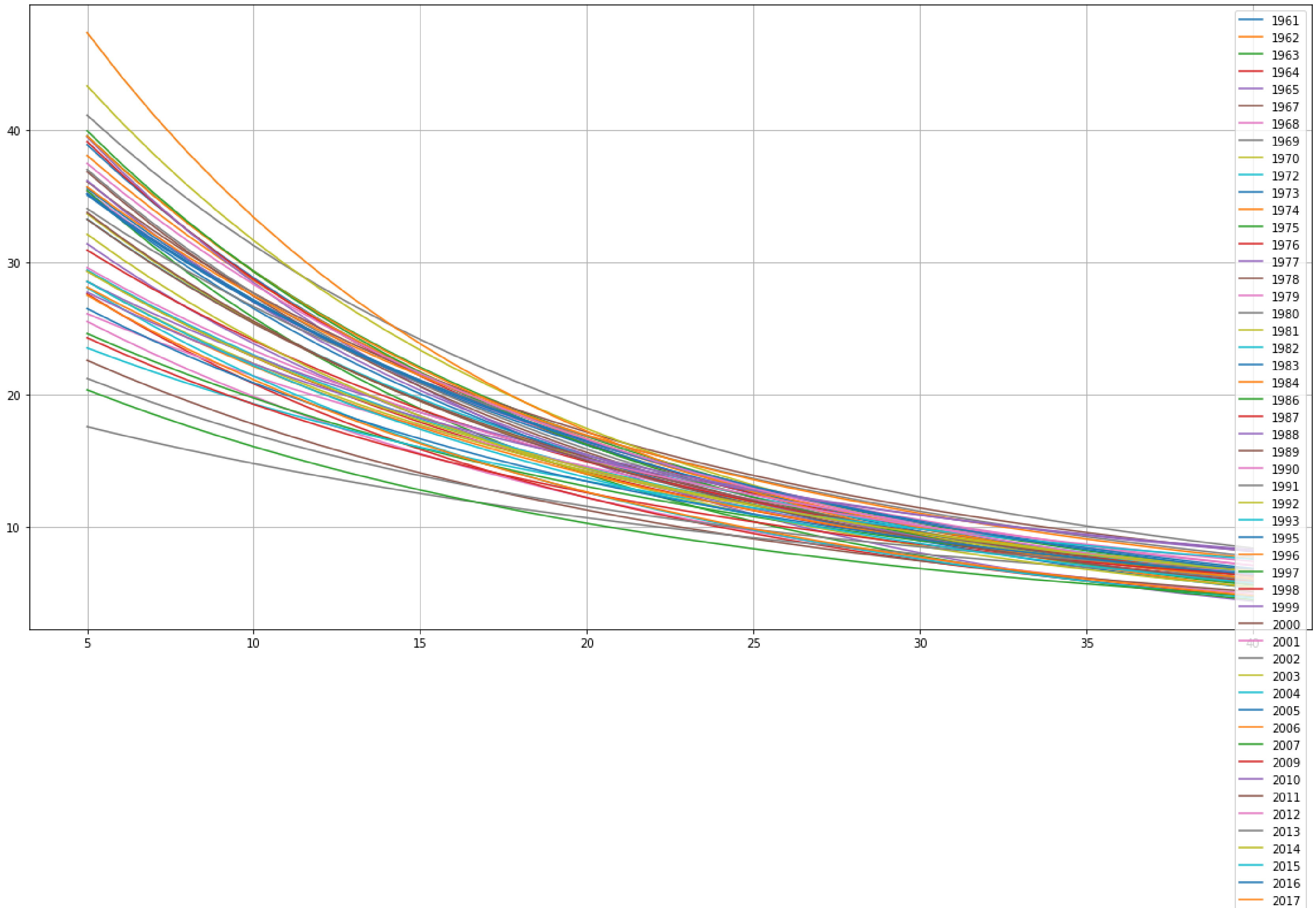
```
9   28.472383  32.986075  33.674915  26.809727    ...      16.813589
10  27.005550  31.265466  31.647336  25.431977    ...      16.044585
11  25.628525  29.653801  29.754470  24.136496    ...      15.316041
12  24.335193  28.143405  27.986757  22.918001    ...      14.625717
13  23.119889  26.727183  26.335346  21.771566    ...      13.971504
14  21.977361  25.398573  24.792041  20.692599    ...      13.351413
15  20.902738  24.151504  23.349253  19.676817    ...      12.763570
16  19.891501  22.980359  21.999951  18.720225    ...      12.206207
17  18.939458  21.879936  20.737622  17.819094    ...      11.677658
18  18.042714  20.845420  19.556229  16.969944    ...      11.176352
19  17.197656  19.872349  18.450175  16.169527    ...      10.700805
20  16.400925  18.956587  17.414272  15.414808    ...      10.249620
21  15.649401  18.094304  16.443709  14.702955    ...       9.821478
22  14.940181  17.281943  15.534023  14.031320    ...       9.415133
23  14.270568  16.516207  14.681073  13.397430    ...       9.029413
24  13.638053  15.794037  13.881015  12.798971    ...       8.663207
25  13.040298  15.112591  13.130281  12.233784    ...       8.315470
26  12.475130  14.469230  12.425561  11.699845    ...       7.985215
27  11.940522  13.861503  11.763776  11.195264    ...       7.671508
28  11.434587  13.287132  11.142070  10.718272    ...       7.373469
29  10.955567  12.743998  10.557786  10.267214    ...       7.090267
30  10.501821  12.230132  10.008457   9.840540    ...       6.821116
31  10.071819  11.743698   9.491787   9.436798    ...       6.565273
32   9.664134  11.282992   9.005643   9.054631    ...       6.322039
33   9.277434  10.846423   8.548039   8.692766    ...       6.090751
34   8.910472  10.432511   8.117129   8.350009    ...       5.870783
35   8.562088  10.039876   7.711194   8.025244    ...       5.661543
36   8.231193   9.667232   7.328633   7.717424    ...       5.462472
37   7.916773   9.313379   6.967957   7.425565    ...       5.273042
38   7.617876   8.977196   6.627776   7.148747    ...       5.092754
39   7.333614   8.657640   6.306799   6.886106    ...       4.921133
40   7.063154   8.353734   6.003818   6.636832    ...       4.757735

         2009       2010       2011       2012       2013       2014  \
5   24.285988  27.736887  33.767207  37.481296  21.205958  29.306835
6   23.161089  26.524076  31.912487  35.416652  20.250222  27.858905
7   22.098828  25.379704  30.169335  33.476808  19.352112  26.494176
8   21.095454  24.299494  28.530656  31.653751  18.507897  25.207525
9   20.147453  23.279450  26.989824  29.940008  17.714096  23.994155
10  19.251531  22.315845  25.540653  28.328608  16.967466  22.849580
11  18.404601  21.405195  24.177362  26.813045  16.264981  21.769598
12  17.603768  20.544248  22.894550  25.387246  15.603822  20.750278
13  16.846318  19.729963  21.687170  24.045545  14.981358  19.787938
```

```
14  16.129705  18.959500  20.550505  22.782647  14.395133  18.879131
15  15.451539  18.230202  19.480146  21.593611  13.842860  18.020628
16  14.809581  17.539588  18.471970  20.473819  13.322399  17.209404
17  14.201727  16.885334  17.522122  19.418958  12.831758  16.442627
18  13.626004  16.265268  16.626997  18.424996  12.369072  15.717643
19  13.080557  15.677358  15.783223  17.488166  11.932604  15.031964
20  12.563648  15.119702  14.987646  16.604946  11.520729  14.383258
21  12.073641  14.590517  14.237312  15.772043  11.131931  13.769340
22  11.609002  14.088137  13.529459  14.986374  10.764791  13.188162
23  11.168289  13.610999  12.861502  14.245060  10.417986  12.637802
24  10.750147  13.157641  12.231020  13.545404  10.090276  12.116458
25  10.353302  12.726690  11.635745  12.884883   9.780506  11.622441
26   9.976557  12.316862  11.073556  12.261134   9.487591  11.154164
27   9.618787  11.926952  10.542464  11.671947   9.210521  10.710140
28   9.278933  11.555828  10.040606  11.115251   8.948348  10.288973
29   8.955999  11.202431   9.566237  10.589106   8.700186   9.889351
30   8.649049  10.865766   9.117722  10.091695   8.465206   9.510044
31   8.357202  10.544900   8.693528   9.621317   8.242631   9.149898
32   8.079629  10.238954   8.292217   9.176374   8.031734   8.807826
33   7.815548   9.947107   7.912444   8.755371   7.831835   8.482809
34   7.564226   9.668585   7.552943   8.356906   7.642297   8.173890
35   7.324970   9.402661   7.212529   7.979663   7.462521   7.880168
36   7.097128   9.148653   6.890092   7.622408   7.291948   7.600798
37   6.880086   8.905918   6.584588   7.283983   7.130052   7.334985
38   6.673267   8.673852   6.295037   6.963303   6.976343   7.081980
39   6.476124   8.451888   6.020521   6.659347   6.830357   6.841080
40   6.288145   8.239491   5.760177   6.371157   6.691662   6.611625

         2015       2016       2017
5   28.550943  26.496934  27.509010
6   27.131151  25.240513  26.081565
7   25.789274  24.054264  24.732542
8   24.520757  22.933972  23.457460
9   23.321326  21.875685  22.252104
10  22.186969  20.875699  21.112504
11  21.113916  19.930541  20.034923
12  20.098628  19.036953  19.015842
13  19.137778  18.191881  18.051949
14  18.228244  17.392457  17.140125
15  17.367089  16.635993  16.277431
16  16.551553  15.919967  15.461102
17  15.779043  15.242011  14.688531
18  15.047121  14.599904  13.957263
```

```
19  14.353493  13.991558  13.264987
20  13.696002  13.415018  12.609522
21  13.072620  12.868442  11.988816
22  12.481439  12.350107  11.400934
23  11.920662  11.858389  10.844053
24  11.388599  11.391765  10.316455
25  10.883659  10.948804   9.816520
26  10.404345  10.528162   9.342721
27   9.949246  10.128575   8.893618
28   9.517034   9.748855   8.467854
29   9.106457   9.387886   8.064149
30   8.716338   9.044617   7.681296
31   8.345565   8.718061   7.318156
32   7.993091   8.407290   6.973656
33   7.657931   8.111430   6.646783
34   7.339154   7.829660   6.336580
35   7.035883   7.561207   6.042147
36   6.747291   7.305341   5.762631
37   6.472598   7.061380   5.497231
38   6.211068   6.828677   5.245187
39   5.962006   6.606626   5.005785
40   5.724756   6.394654   4.778349

[36 rows x 52 columns]
```

In [49]:
```
dts_df_plot.plot(figsize=(20,10),grid = True)
```

Out[49]: <matplotlib.axes._subplots.AxesSubplot at 0x121b4b95048>



Problem 2-3: (11pts)

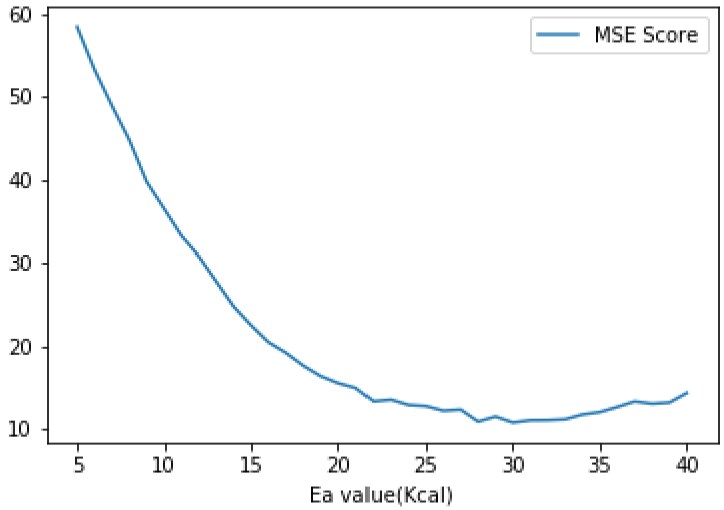
Using the same E_a values and calculated DTS_{mean} from 2-2, predict the bloom date BD_j for each of the training years. Find the mean squared error relative to the actual BD and plot it against E_a. Find the optimal E_a^* that minimizes that error on the training data.

In [50]:
```
MSE_DF= FindingMSE(train_df,Dj_tr,ActualBD_tr,DTSmean_df_tr,train_years)
print(MSE_DF)
```

```
    Ea value(Kcal)  MSE Score
0                5  58.403846
1                6  53.250000
2                7  48.903846
3                8  44.769231
4                9  39.711538
5               10  36.480769
6               11  33.211538
7               12  30.769231
8               13  27.750000
9               14  24.750000
10              15  22.480769
11              16  20.442308
12              17  19.173077
13              18  17.615385
14              19  16.346154
15              20  15.500000
16              21  14.923077
17              22  13.307692
18              23  13.500000
19              24  12.865385
20              25  12.730769
21              26  12.173077
22              27  12.307692
23              28  10.865385
24              29  11.461538
25              30  10.750000
26              31  11.019231
27              32  11.038462
28              33  11.134615
29              34  11.711538
30              35  12.000000
31              36  12.615385
32              37  13.288462
33              38  13.019231
34              39  13.153846
35              40  14.307692
```

In [51]:
```
MSE_DF.plot(x='Ea value(Kcal)',y='MSE Score')
```

Out[51]: <matplotlib.axes._subplots.AxesSubplot at 0x121b50d16d8>



Problem 2-4: (4pts)

Using the D_j dates from problem 2-1, the average DTS_{mean} from 2-2, and the best-fit E_a^* from 2-3, predict the bloom-dates BD_j for the years in the test set. Determine the error between your predicted BD_j values and the actual values, and evaluate this model using the coefficient of determination (R^2 score).

In [60]:
```
DTSmean = 9.376342
Ea = 30
Tf_test = CalcTf(test_df,test_years)
Dj_test = CalcDj(Fi,L,Tf_test,test_years)
pred_test_optEa=PREDByBestEaDTSmeas(test_df,Dj_test,ActualBD_test,Ea,DTSmean,test_years)
print(pred_test_optEa)
```

	day:actual	day:predicted	years
0	79	79	1966
1	89	88	1971
2	93	92	1985
3	90	91	1994
4	82	83	2008

In [56]:
```
DTSmean = 9.376342
Ea = 30
Tf_test = CalcTf(test_df,test_years)
Dj_test = CalcDj(Fi,L,Tf_test,test_years)
pred_test_optEa=PREDByBestEaDTSmeas(test_df,Dj_test,ActualBD_test,Ea,DTSmean,test_years)
print(pred_test_optEa)
DTSmean
```

```
   day:actual  day:predicted  years
0          79             80   1966
1          89             89   1971
2          93             93   1985
3          90             92   1994
4          82             84   2008
```

Out[56]:
```
9.376342
```

In [61]:
```
PRED_optEa=np.array(pred_test_optEa['day:predicted'])
ActualBD_test = ActualBD(test_df,test_years)
ACT_BD_optEa = np.array(ActualBD_test['day:actual'])
```

In [62]:
```
R2_Score_optEa = CalcR2Score(ACT_BD_optEa,PRED_optEa)
print("R2 Score for best DTS is: ",R2_Score_optEa)
```

```
R2 Score for best DTS is:  0.9708454810495627
```

Problem 2-5: (extra 10pts)

Discuss any improvements you could make to the model outlined above. If you have a suggestion in particular, describe it. How much do you think the accuracy would be improved?

1.According to the mentioned model and datasets, there is an assumption that the average temperature of a single day is sustained throughout the day. This is not the case in real life as the temperature can flactuate even from hour to hour. So if the fluctuations of the temperature could be added, the accuracy and could possibly be improved.

2.There are different species of cherry trees from the Prunus genus. Different species of trees might have different type if implecations to different changes of variables. We did not take that into account. If we can take those things into consideration, accuracy might be improved.

Suggested improvements:

1. Sakura blooming depends on a lot of variables such as maximum and minimum temperature of the day, precipitation, available sun hours etc. These variables are no less important that features included in the model outlined above. So inclusion of these variables in the above model can improve the model's accuracy.
2. Besides air composition can be an important factor for Sakura bloomimg because percentage of Carbon Di Oxide in air directly affects photosynthesis and respiration of trees. The model should also take this into account.

---

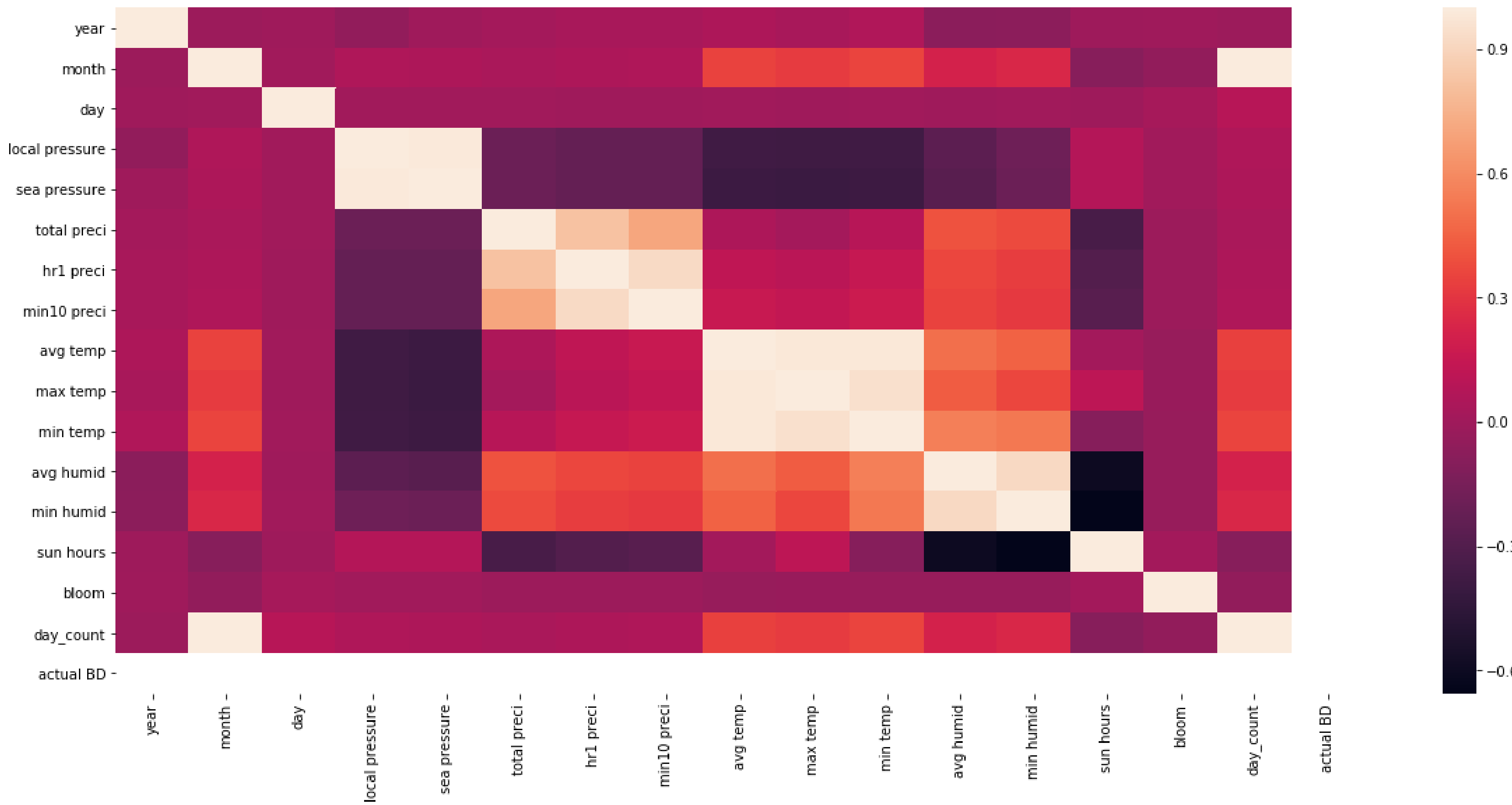
3. Predicting Bloom-date via Neural Network (30pts total)

Data analysis

In [63]:
```
c_df = df.copy()
c_train_df = train_df.copy()
c_test_df = test_df.copy()
```

In [64]:
```
#OBESERVING HEAT MAP
corr_df = c_df.corr()
f, ax = plt.subplots(figsize=(20, 9))
sns.heatmap(corr_df)
```

Out[64]: <matplotlib.axes._subplots.AxesSubplot at 0x121b4cb58d0>



Correlation analysis among attributes:

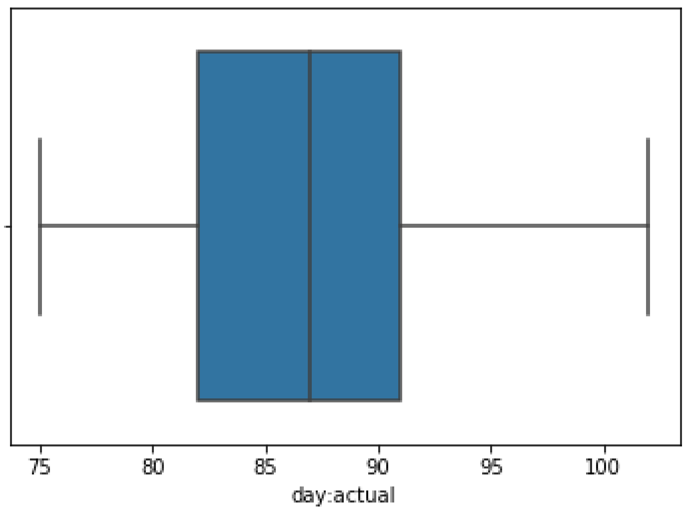
Highly correlated features:

1. Temperature and precipitation
2. Temperature and humidity
3. Humidity and precipitation
4. Pressure and temperrature(negative)
5. Average humidity and sun hours

We will analyze these correlations while discussing environmental blooming factors for sakura.

In [65]:
```
C_ACTULABD_DF = ActualBD_df.copy()
```

In [66]:
```
sns.boxplot(C_ACTULABD_DF['day:actual'])
```

Out[66]: ```
<matplotlib.axes._subplots.AxesSubplot at 0x121b4caeac8>
```


```

Blooming factors for Sakura:

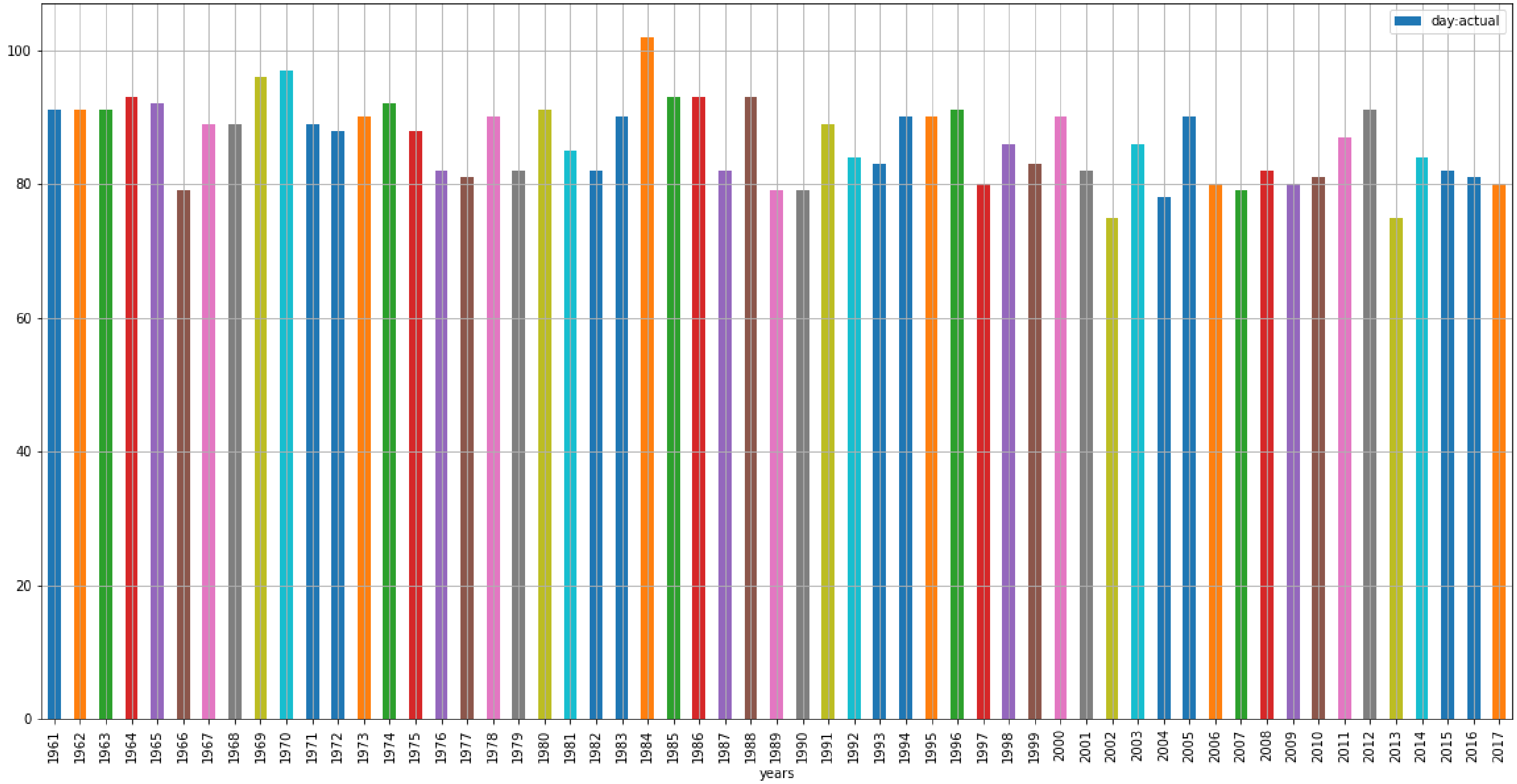
To predict the blooming day of Sakura accurately, we need to know aout the factors that affect Sakura blooming and their correlations.Various environmental factors affect when plants open their flowers, including day length, air temperature, and soil moisture.[1]
The most important factor in blooming of Sakura is temperature.
-Chilly days during the winter and warm or mild days during the spring generally accelerate the maturation of flower buds.
-By contrast, unseasonable winter warmth and unseasonable springtime chills can slow the process.[3]

Feature engineering decisions:

1. From above mentioned points obviously minimum temperatures, maximimum temperatures and averarge temperatures all play signifant role in sakura blooming. But our interest should be the winter and spring season just before the season of sakura. Hence, here we have considered the first three months of each year for each feature.
2. Soil moisture highly depends on ammount of precipitaion. So total precipitation of first three months of each year were also considered. Sun hours was also considiered as a feature.
3. Even though humidity is highly correlated to temperature, precipitaion and sun hours, this feature was not included because it degrades the performance on the NN model. This may be because we have limited number of training data and unnecessary number of features can hamper the performance of the model.
4. Since the place was fixed so presssure remained almost same during our interest period over the years. So pressure values were not included as feature. If we need to predict the blooming dateof Sakura for different places, we need to consider air pressure also.
5. Year was considered a feature to take account of the gradual change in overall environment.

In [67]:
```
C_ACTULABD_DF.plot(x='years',y='day:actual',kind='bar',grid=1,figsize=(20,10),xticks=list(range(1961,2018,2
)))
```

Out[67]: <matplotlib.axes._subplots.AxesSubplot at 0x121b42f8c88>



Evidence behind decisions:

From the above plot, we can see blooming occured later than usual times in 1969,1970 and 1984. If we analyze our data, we can easily observe that winters were warmer than usual(~6 degree celcius) and springs were colder(~7.5) than usual.
In 1966,1989,1997,2002,2013 winters were colder(~3) than usual and springs were warmer(>9 degree celcius) that usual. So as mentioned above, blooming occured pretty early. This proves that our model is expected to give satisfactory result since we have also emphasised on various temperature features of the first three months

Feature engineering

In [68]:
```
new_df = CreateFeat(df,all_years,1,3,'avg temp')
new_df2 = CreateFeat(df,all_years,1,3,'min temp')
new_df3 = CreateFeat(df,all_years,1,3,'total preci')
new_df5 = CreateFeat(df,all_years,1,3,'sun hours')
```

In [69]:
```
jan_mintemp = new_df2[1].values.tolist()
feb_mintemp = new_df2[2].values.tolist()
march_mintemp = new_df2[3].values.tolist()
jan_p = new_df3[1].values.tolist()
feb_p = new_df3[2].values.tolist()
march_p = new_df3[3].values.tolist()
jan_sun = new_df5[1].values.tolist()
feb_sun = new_df5[2].values.tolist()
march_sun = new_df5[3].values.tolist()
```

In [70]:
```
act_bd_list = ActualBD_df['day:actual'].values.tolist()
```

In [71]:
```
new_df['day:actual']=act_bd_list
new_df['min_temp_jan']= jan_mintemp
new_df['min_temp_feb']= feb_mintemp
new_df['min_temp_mar']= march_mintemp
new_df['preci_jan']= jan_p
new_df['preci_feb']= feb_p
new_df['preci_mar']= march_p
new_df['sun_jan']= jan_sun
new_df['sun_feb']= feb_sun
new_df['sun_mar']= march_sun
new_df.set_index('year',inplace=True,drop=False)
```

In [72]:
```
print(new_df)
```

```
             1         2          3  year  day:actual  min_temp_jan  \
year                                                                  
1961  3.570968  4.528571   8.222581  1961          91     -0.680645   
1962  4.545161  5.939286   8.248387  1962          91     -0.500000   
1963  3.170968  4.800000   7.635484  1963          91     -2.625806   
1964  5.400000  4.186207   7.570968  1964          93      1.570968   
1965  4.400000  4.671429   6.925806  1965          92      0.570968   
1966  4.648387  7.217857   9.558065  1966          79      0.135484   
1967  4.445161  4.932143   9.490323  1967          89      0.148387   
1968  5.703226  4.337931  10.025806  1968          89      0.903226   
1969  5.719355  5.664286   7.858065  1969          96      1.377419   
1970  4.529032  5.992857   5.548387  1970          97      0.409677   
1971  5.122581  5.853571   8.303226  1971          89      1.674194   
1972  6.587097  5.113793   9.661290  1972          88      2.822581   
1973  6.267742  6.882143   7.796774  1973          90      2.832258   
1974  4.358065  5.060714   7.332258  1974          92      0.558065   
1975  4.661290  5.107143   7.880645  1975          88      0.793548   
1976  5.374194  6.751724   8.967742  1976          82      0.700000   
1977  3.448387  4.910714   9.300000  1977          81      0.029032   
1978  5.616129  4.228571   8.651613  1978          90      1.725806   
1979  6.574194  8.439286   9.922581  1979          82      2.448387   
1980  5.554839  5.227586   8.238710  1980          91      1.683871   
1981  4.435484  5.296429   8.961290  1981          85      0.567742   
1982  5.761290  5.450000   9.877419  1982          82      2.241935   
1983  6.238710  6.053571   8.567742  1983          90      2.493548   
1984  3.735484  3.037931   5.887097  1984         102      0.158065   
1985  4.125806  6.475000   7.764516  1985          93      0.451613   
1986  4.516129  4.253571   7.751613  1986          93      0.967742   
1987  5.812903  6.832143   9.293548  1987          82      1.948387   
1988  7.683871  4.934483   8.351613  1988          93      3.729032   
1989  8.070968  7.500000   9.580645  1989          79      4.819355   
1990  5.003226  7.764286  10.622581  1990          79      1.729032   
1991  6.280645  6.546429   9.545161  1991          89      2.841935   
1992  6.793548  6.879310   9.700000  1992          84      3.174194   
1993  6.190323  7.685714   8.700000  1993          83      3.293548   
1994  5.532258  6.575000   8.061290  1994          90      2.106452   
1995  6.283871  6.460714   8.903226  1995          90      2.525806   
1996  6.590323  5.427586   9.154839  1996          91      2.935484   
1997  6.790323  7.028571  10.532258  1997          80      2.706452   
1998  5.348387  7.014286  10.132258  1998          86      1.867742   
1999  6.622581  6.721429  10.051613  1999          83      2.512903   
2000  7.583871  5.951724   9.406452  2000          90      4.203226   
2001  4.900000  6.628571   9.819355  2001          82      1.716129   
```

```
2002  7.441935  7.932143  12.203226  2002          75      3.409677
2003  5.454839  6.442857   8.674194  2003          86      1.964516
2004  6.341935  8.510345   9.822581  2004          78      3.064516
2005  6.129032  6.150000   8.964516  2005          90      2.551613
2006  5.061290  6.657143   9.845161  2006          80      2.038710
2007  7.638710  8.646429  10.825806  2007          79      4.629032
2008  5.893548  5.513793  10.677419  2008          82      2.693548
2009  6.758065  7.821429   9.964516  2009          80      3.538710
2010  7.022581  6.500000   9.096774  2010          81      2.967742
2011  5.054839  7.007143   8.080645  2011          87      1.532258
2012  4.796774  5.448276   8.825806  2012          91      1.796774
2013  5.541935  6.160714  12.096774  2013          75      1.845161
2014  6.329032  5.942857  10.380645  2014          84      2.480645
2015  5.783871  5.717857  10.251613  2015          82      1.832258
2016  6.080645  7.227586  10.141935  2016          81      1.838710
2017  5.832258  6.928571   8.493548  2017          80      1.677419

      min_temp_feb  min_temp_mar  preci_jan  preci_feb  preci_mar   sun_jan  \
year
1961     -0.457143      3.919355   1.258065   1.553571   3.438710  5.922581
1962      0.832143      3.335484   1.306452   0.482143   2.112903  6.796774
1963      0.214286      3.161290   0.006452   0.760714   2.796774  7.454839
1964      0.682759      3.077419   4.664516   2.206897   3.216129  4.019355
1965      0.289286      2.590323   1.548387   0.375000   1.435484  5.658065
1966      2.589286      5.487097   0.761290   4.357143   3.222581  6.316129
1967      1.060714      4.745161   1.038710   1.564286   2.248387  6.270968
1968     -0.137931      5.822581   0.306452   1.655172   3.032258  7.070968
1969      2.089286      3.703226   1.758065   3.571429   3.903226  4.787097
1970      1.703571      1.329032   1.887097   1.232143   1.500000  5.870968
1971      1.910714      4.109677   1.032258   1.357143   2.177419  5.858065
1972      2.310345      5.506452   3.661290   4.896552   1.419355  5.045161
1973      2.789286      3.725806   4.387097   1.607143   0.322581  5.729032
1974      1.439286      3.590323   0.935484   2.089286   3.677419  7.403226
1975      1.250000      3.993548   2.177419   2.678571   3.483871  5.893548
1976      2.875862      5.109677   0.016129   4.413793   2.806452  7.264516
1977      0.521429      5.261290   0.629032   0.964286   5.387097  5.209677
1978      0.564286      4.841935   0.903226   1.142857   3.725806  5.161290
1979      4.889286      5.777419   1.903226   3.428571   2.967742  5.790323
1980      1.306897      4.851613   2.854839   0.896552   5.596774  5.870968
1981      1.892857      5.180645   0.112903   1.357143   3.645161  7.529032
1982      1.846429      5.903226   1.048387   1.839286   2.354839  6.258065
1983      2.392857      5.187097   0.951613   1.803571   3.193548  6.612903
1984     -0.037931      1.954839   1.612903   1.758621   2.225806  6.735484
```

```
1985      3.139286      4.841935   0.145161   5.071429   4.338710  6.480645
1986      0.810714      3.900000   0.483871   0.982143   6.064516  5.990323
1987      2.953571      5.141935   1.225806   1.053571   3.032258  5.851613
1988      1.131034      4.561290   0.903226   0.603448   5.967742  6.354839
1989      4.317857      5.793548   3.064516   3.821429   3.806452  4.922581
1990      5.121429      7.019355   1.048387   4.160714   3.145161  5.783871
1991      2.532143      6.161290   1.774194   2.464286   5.290323  6.793548
1992      3.100000      6.480645   1.612903   1.241379   6.612903  5.909677
1993      3.750000      5.022581   3.709677   2.035714   1.951613  4.193548
1994      3.032143      4.819355   1.612903   3.160714   3.693548  5.600000
1995      2.939286      5.296774   1.161290   0.964286   5.806452  7.232258
1996      1.668966      5.509677   0.419355   1.551724   3.806452  6.380645
1997      3.100000      6.796774   0.935484   0.767857   3.419355  7.632258
1998      3.585714      5.929032   3.935484   3.946429   3.483871  5.241935
1999      2.535714      6.332258   0.629032   1.267857   4.661290  7.000000
2000      2.372414      5.229032   2.145161   0.137931   2.758065  4.961290
2001      2.832143      5.654839   4.080645   0.821429   3.467742  5.767742
2002      4.414286      8.080645   3.177419   0.892857   2.645161  6.512903
2003      3.235714      5.135484   3.258065   1.910714   5.145161  6.709677
2004      4.279310      5.716129   0.112903   0.689655   4.177419  6.567742
2005      2.478571      4.980645   2.483871   1.714286   2.290323  6.451613
2006      3.264286      5.929032   2.161290   4.035714   2.564516  5.480645
2007      4.967857      6.822581   1.354839   2.035714   2.483871  5.664516
2008      1.872414      7.170968   0.564516   1.965517   3.854839  5.316129
2009      4.446429      6.322581   4.580645   1.660714   3.177419  5.635484
2010      3.032143      5.093548   0.290323   4.107143   4.629032  7.158065
2011      3.217857      4.048387   0.112903   5.392857   2.387097  7.867742
2012      2.231034      5.267742   1.612903   3.241379   4.661290  5.903226
2013      2.689286      7.932258   2.258065   1.071429   1.435484  6.854839
2014      2.803571      6.680645   0.790323   5.625000   3.661290  6.583871
2015      1.910714      5.783871   2.983871   2.214286   3.032258  5.870968
2016      3.113793      6.109677   2.741935   1.965517   3.322581  6.500000
2017      2.592857      4.151613   0.838710   0.553571   2.758065  7.312903

       sun_feb   sun_mar
year
1961  7.067857  5.841935
1962  6.964286  6.122581
1963  7.139286  6.412903
1964  5.206897  6.419355
1965  6.907143  8.141935
1966  5.342857  4.851613
1967  5.807143  5.832258
```

```
1968  7.096552  5.093548
1969  3.467857  5.783871
1970  5.717857  6.316129
1971  5.317857  6.558065
1972  3.882759  6.700000
1973  5.817857  6.693548
1974  4.814286  5.925806
1975  6.300000  5.880645
1976  4.734483  4.780645
1977  6.735714  4.577419
1978  5.632143  6.229032
1979  5.878571  6.135484
1980  7.106897  5.080645
1981  5.617857  6.212903
1982  6.000000  5.735484
1983  7.246429  5.677419
1984  6.293103  6.635484
1985  5.471429  2.387097
1986  6.485714  5.209677
1987  5.685714  4.809677
1988  6.148276  4.187097
1989  4.839286  5.690323
1990  2.914286  6.464516
1991  7.117857  3.735484
1992  5.737931  3.187097
1993  6.989286  6.054839
1994  7.364286  5.100000
1995  6.403571  4.335484
1996  6.441379  5.241935
1997  6.842857  5.867742
1998  5.371429  6.177419
1999  7.175000  3.838710
2000  7.162069  6.706452
2001  5.464286  5.809677
2002  5.839286  6.170968
2003  5.500000  6.574194
2004  7.527586  5.512903
2005  5.317857  5.648387
2006  4.589286  5.683871
2007  6.914286  6.290323
2008  7.406897  6.038710
2009  4.685714  5.254839
2010  4.225000  4.509677
```

```
2011  5.317857  6.929032
2012  5.124138  4.829032
2013  6.203571  6.132258
2014  4.992857  6.612903
2015  5.960714  6.264516
2016  5.520690  5.222581
2017  6.917857  6.138710
```

Preparing labels for NN model

In [73]:
```
test_labels = new_df.loc[test_years,'day:actual'].values
train_labels = new_df.loc[train_years,'day:actual'].values
del new_df['day:actual']
```

Scaling feature values

Among various scalers(Standard,MinMax,Robust) Robust scaler performed best. Hence Robust Scaler is used here to scale the features.

Why scaling is needed:

Most machine learning algorithms take into account only the magnitude of the measurements, not the units of those measurements. That’s why one feature, which is expressed in a very high magnitude (number), may affect the prediction a lot more than an equally important feature.

In [74]:
```
col_names = new_df.columns.values.tolist()
robust_scaler = RobustScaler()
new_df[col_names] = robust_scaler.fit_transform(new_df[col_names])
```

In [75]:
```
new_df = pd.DataFrame(new_df)
new_df.head()
```

Out[75]:

	1	2	3	year	min_temp_jan	min_temp_feb	min_temp_mar	preci_jan	preci_feb	preci_mar	sun_j
year											
1961	-1.391579	-0.979978	-0.533465	-1.000000	-1.400716	-1.800000	-0.723757	-0.032967	-0.076271	0.1675	-0.0632
1962	-0.755789	-0.127354	-0.517717	-0.964286	-1.300537	-1.023656	-1.057090	0.000000	-0.584746	-0.8600	0.7530
1963	-1.652632	-0.815929	-0.891732	-0.928571	-2.479428	-1.395699	-1.156538	-0.885714	-0.452542	-0.3300	1.3674
1964	-0.197895	-1.186900	-0.931102	-0.892857	-0.152057	-1.113608	-1.204420	2.287912	0.233781	-0.0050	-1.8403
1965	-0.850526	-0.893636	-1.324803	-0.857143	-0.706619	-1.350538	-1.482505	0.164835	-0.635593	-1.3850	-0.3102

Preparing train and test data for NN model

In [76]:
```
new_test_df =new_df.loc[ test_years , : ]
new_train_df =new_df.loc[ train_years , : ]
```

In [77]:
```
train = new_train_df.copy()
test = new_test_df.copy()
```

In [78]:
```
train =np.array(train)
test =np.array(test)
train_labels =np.array(train_labels)
test_labels =np.array(test_labels)
train_labels =train_labels.reshape(-1,1)
```

Problem 3-1: (20pts)

Build a neural network and train it on the data from the training years. Use this model to predict the bloom-dates for each year in the test set. Evaluate the error between predicted dates and actual dates using the coefficient of determination (R2 score). Only use the weather data given in `tokyo.csv` and the sakura data acquired in problem 0-1.</br> You may use whichever framework or strategy that you like to construct the network.

Tuning hyperparameters and decision of best model:

a) No of input features:

1. Avg temp of January and March:

r2 score: 0.672

2. Average temp of February and March:

r2 score: 0.7303

3. Average temp of January-March and min temp of January:

r2 score: 0.7594

4. Average and min temp of January-March:

r2 score: 0.82507

5. Average, min and max temp of January-March:

r2score: 0.77405

6. Average and min temp and total precipitation of January-March:

r2 score: 0.839

7. Average, min temp,total pricipitation, avg humidity and sun hours of January-March:

r2 score: 0.665

8. Average,min temp, total precipitation and sun hours of January-March:

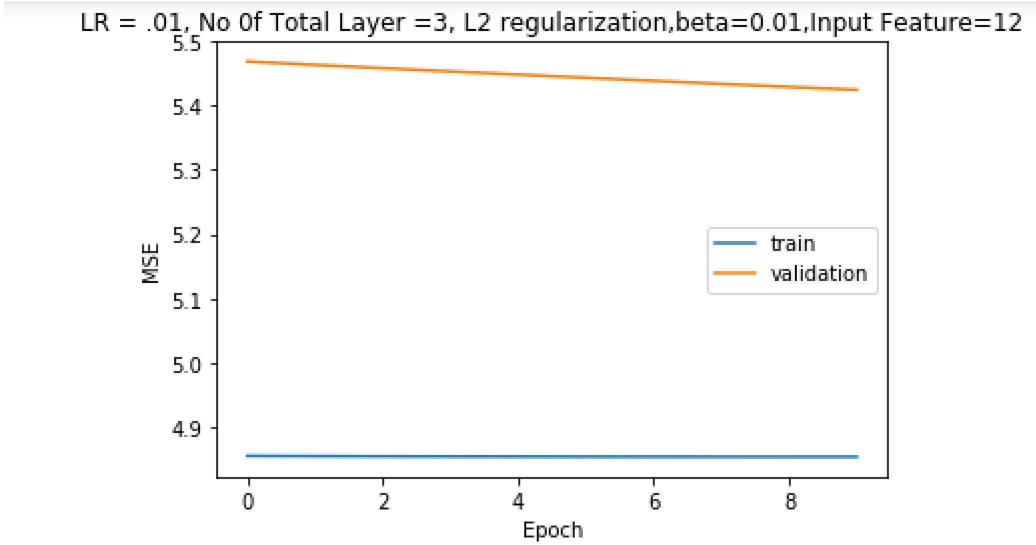
r2 score: 0.87609 All of the above model included 'year' as feature values.

9. Best model without including 'year' as feature:

r2 score: 0.81

b) Tuning number of layers:

ANN model with 1 hidden layer gave an r2 score of 0.87609 on test data and 0.81014 on train data.

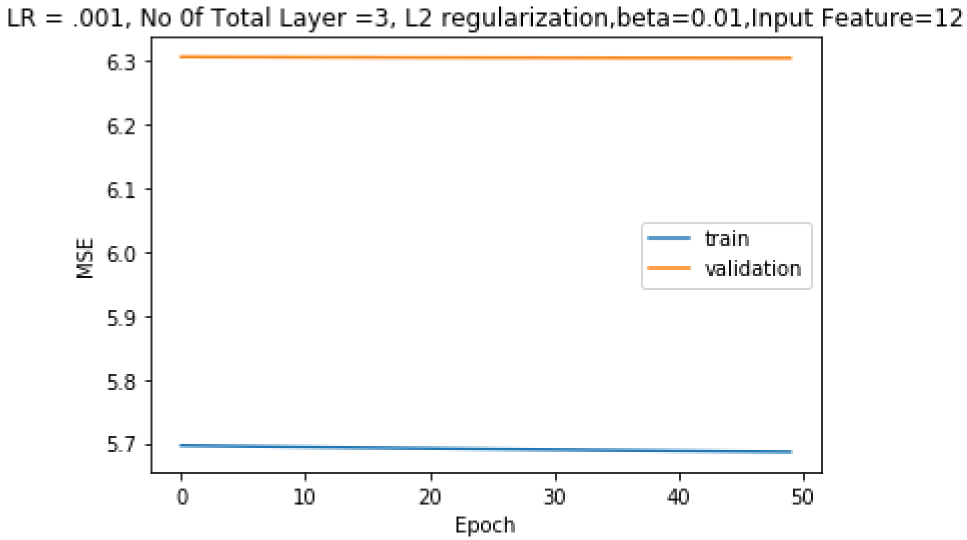



```
#Prediction on test data
in_test={input_feat:np.array(test)}
pred = sess.run((out),feed_dict=in_test)
pred = pred.reshape(-1)
for i in range(len(pred)):
    pred[i]=int(pred[i])
```

```
r2_test_NN = r2_score(test_labels,pred)
print('r2 score by NN:',r2_test_NN)
```

```
r2 score by NN: 0.8760932944606414
```

ANN model with 2 hidden layer gave an r2 score of 0.81778 on test data and 0.6709 on train data.

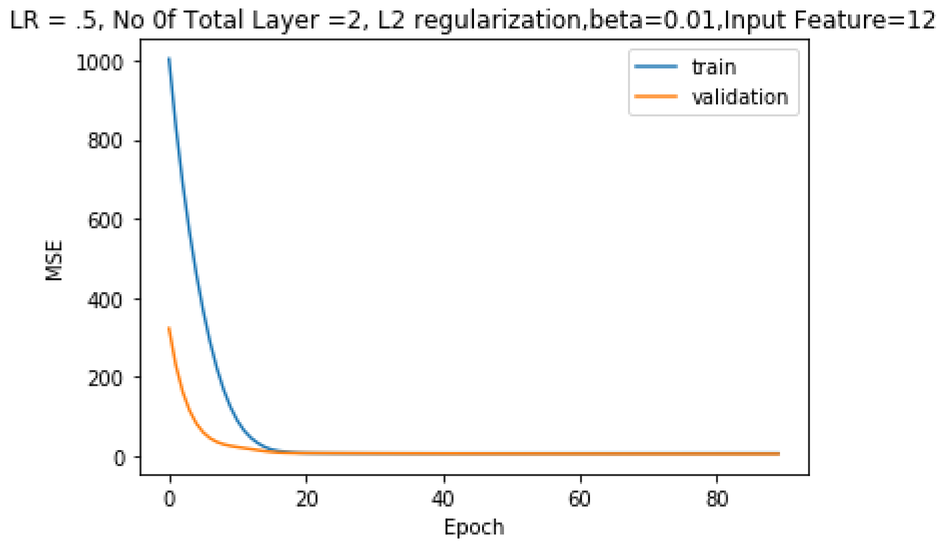



```
#Prediction on test data
in_test={input_feat:np.array(test)}
pred = sess.run((out),feed_dict=in_test)
pred = pred.reshape(-1)
for i in range(len(pred)):
    pred[i]=int(pred[i])
```

```
r2_test_NN = r2_score(test_labels,pred)
print('r2 score by NN:',r2_test_NN)
```

```
r2 score by NN: 0.8177842565597667
```

ANN model with no hidden layer gave an r2 score of 0.91 on test data and 0.77 on train data.




```
#Prediction on test data
in_test={input_feat:np.array(test)}
pred = sess.run((out),feed_dict=in_test)
pred = pred.reshape(-1)
for i in range(len(pred)):
    pred[i]=int(pred[i])
```

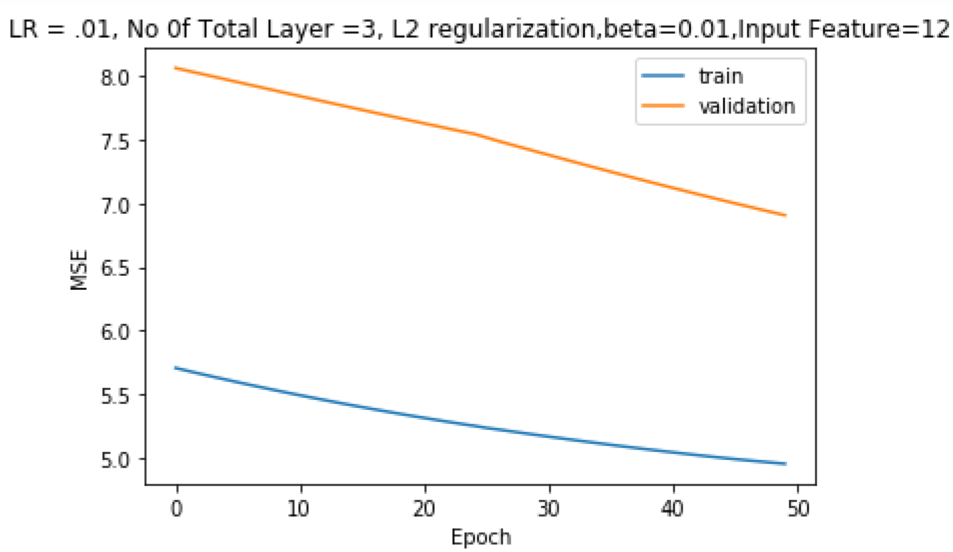
```
r2_test_NN = r2_score(test_labels,pred)
print('r2 score by NN:',r2_test_NN)
```

```
r2 score by NN: 0.9198250728862973
```

Since the ANN model with 1 hidden layer gave satisfactory values in both test and train data, this model was chosen.

c) Activation function:

With activation: 0.6428

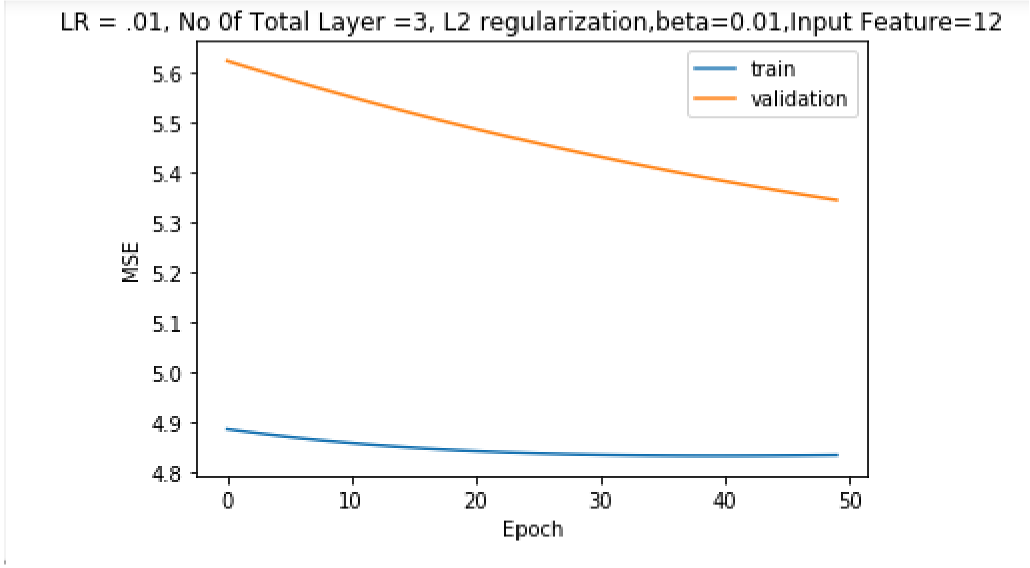



```
#Prediction on test data
in_test={input_feat:np.array(test)}
pred = sess.run((out),feed_dict=in_test)
pred = pred.reshape(-1)
for i in range(len(pred)):
    pred[i]=int(pred[i])
```

```
r2_test_NN = r2_score(test_labels,pred)
print('r2 score by NN:',r2_test_NN)
```

```
r2 score by NN: 0.6428571428571428
```

Without activation: 0.876



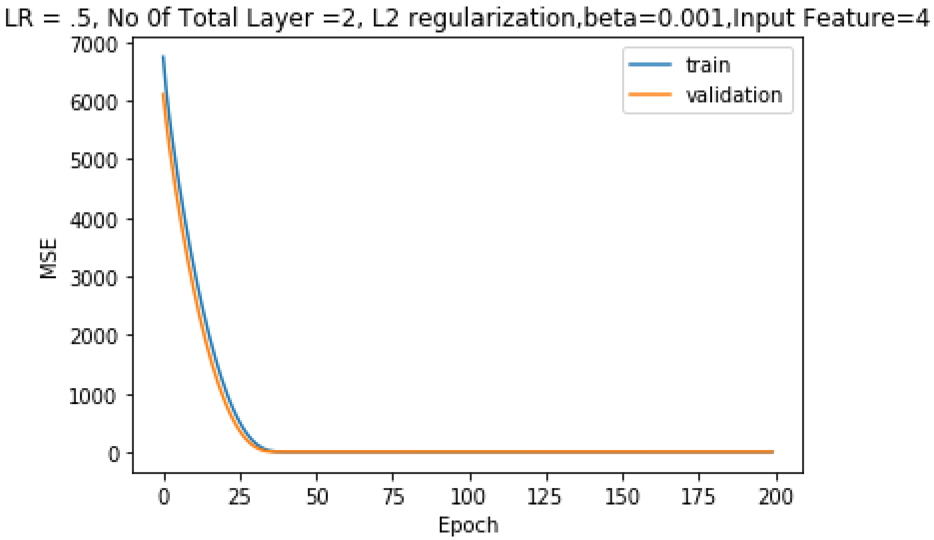

```
#Prediction on test data
in_test={input_feat:np.array(test)}
pred = sess.run((out),feed_dict=in_test)
pred = pred.reshape(-1)
for i in range(len(pred)):
    pred[i]=int(pred[i])
```

```
r2_test_NN = r2_score(test_labels,pred)
print('r2 score by NN:',r2_test_NN)
```

```
r2 score by NN: 0.8760932944606414
```

d) Batch size:

no of batch: 5, r2 score = 0.6356

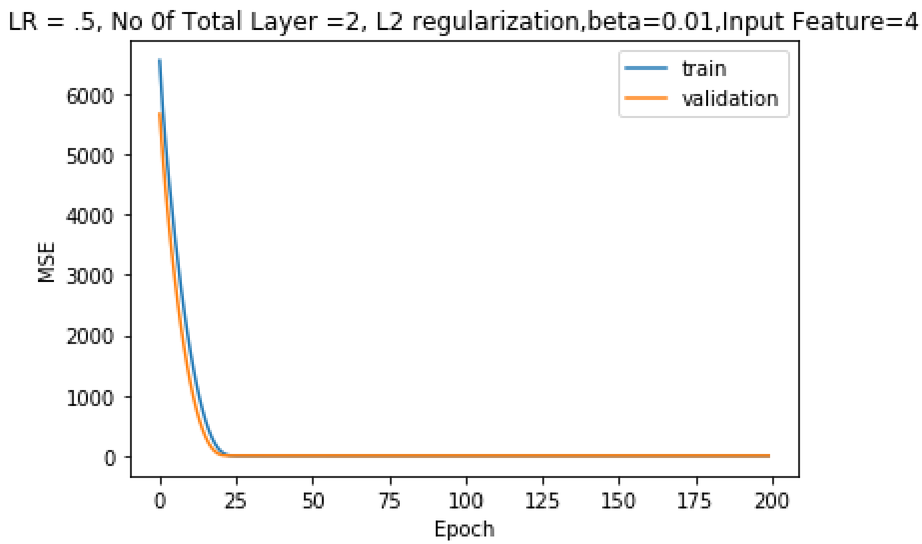



```
#Prediction on test data
in_test={input_feat:np.array(test)}
pred = sess.run((out),feed_dict=in_test)
pred = pred.reshape(-1)
for i in range(len(pred)):
    pred[i]=int(pred[i])
```

```
r2_test_NN = r2_score(test_labels,pred)
print('r2 score by NN:',r2_test_NN)
```

```
r2 score by NN: 0.6355685131195334
```

no of batch: 8, r2_score = 0.7594

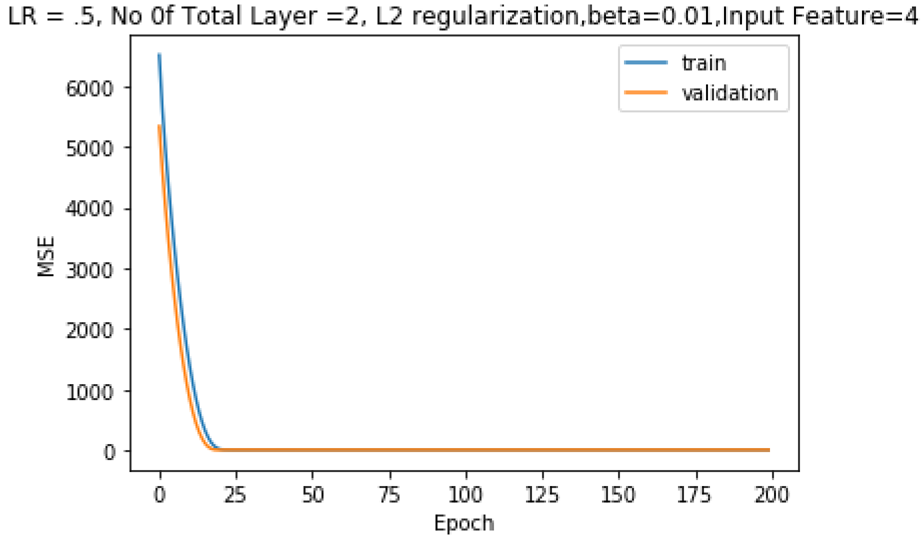



```
#Prediction on test data
in_test={input_feat:np.array(test)}
pred = sess.run((out),feed_dict=in_test)
pred = pred.reshape(-1)
for i in range(len(pred)):
    pred[i]=int(pred[i])
```

```
r2_test_NN = r2_score(test_labels,pred)
print('r2 score by NN:',r2_test_NN)
```

```
r2 score by NN: 0.7594752186588921
```

no of batch: 9, r2_score = 0.68



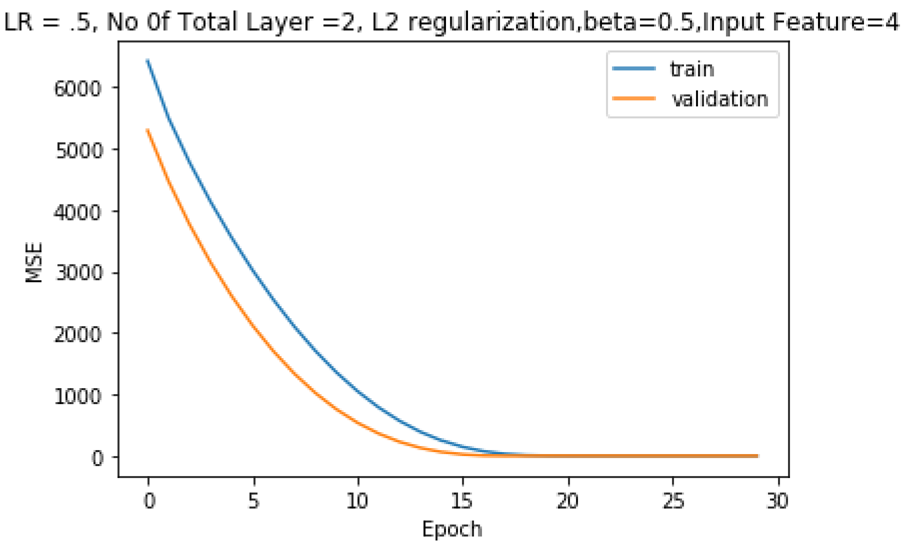

```
#Prediction on test data
in_test={input_feat:np.array(test)}
pred = sess.run((out),feed_dict=in_test)
pred = pred.reshape(-1)
for i in range(len(pred)):
    pred[i]=int(pred[i])
```

```
r2_test_NN = r2_score(test_labels,pred)
print('r2 score by NN:',r2_test_NN)
```

```
r2 score by NN: 0.6793002915451896
```

e) Regularization constant:

value: 0.5, score: 0.5116

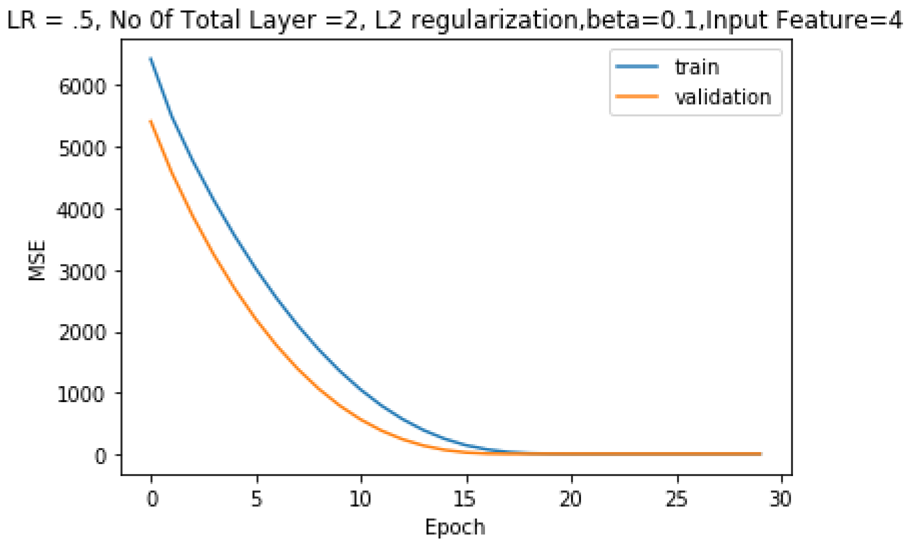



```
#Prediction on test data
in_test={input_feat:np.array(test)}
pred = sess.run((out),feed_dict=in_test)
pred = pred.reshape(-1)
for i in range(len(pred)):
    pred[i]=int(pred[i])
```

```
r2_test_NN = r2_score(test_labels,pred)
print('r2 score by NN:',r2_test_NN)
```

```
r2 score by NN: 0.5116618075801749
```

value: 0.1, score = 0.6793

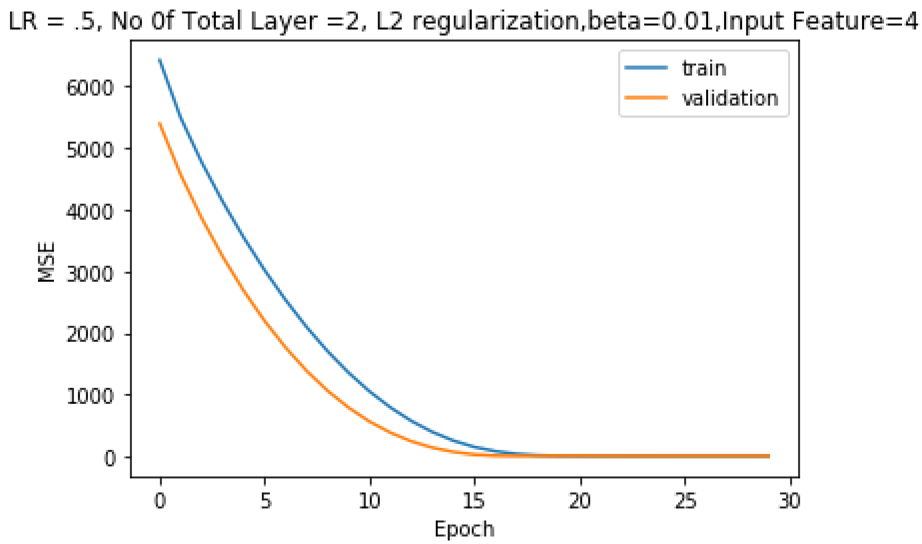



```
#Prediction on test data
in_test={input_feat:np.array(test)}
pred = sess.run((out),feed_dict=in_test)
pred = pred.reshape(-1)
for i in range(len(pred)):
    pred[i]=int(pred[i])
```

```
r2_test_NN = r2_score(test_labels,pred)
print('r2 score by NN:',r2_test_NN)
```

```
r2 score by NN: 0.6793002915451896
```

value: .01, score = 0.7157




```
#Prediction on test data
in_test={input_feat:np.array(test)}
pred = sess.run((out),feed_dict=in_test)
pred = pred.reshape(-1)
for i in range(len(pred)):
    pred[i]=int(pred[i])
```

```
r2_test_NN = r2_score(test_labels,pred)
print('r2 score by NN:',r2_test_NN)
```

f) Optimizer:

Among various optimizer, Adadelta, Gradient descent were really slow and but could not converge that much even with high values of epoch and low learning rate. Adagrad was slightly better. RMSPropOptimizer performed best among these with moderate epoch and learning rate.

g) Epochs and Learning rate:

Various combination of epochs and learning rates for various optimizer has been tried. Among them the best values are chosen based on training and validation error(MSE) and r2 score of test ant training data. Initially 70% data were used for training and 30% for cross validation. Final training are done for all training data and r2 scores are reported for both training and test set.

In [79]:
```
LearningRate = .01
epoch = 190
reg_const = .01
```

In [80]:
```
#Variable declarartions
input_node = train.shape[1]
hid_1 = int(input_node/2)
#hid_2 = int(input_node/2)
output_node = 1
with tf.variable_scope("WB", reuse=tf.AUTO_REUSE):
    w1= tf.Variable(tf.get_variable('w1',[input_node, hid_1],dtype=tf.float64))
    w2= tf.Variable(tf.get_variable('w2',[hid_1, output_node],dtype=tf.float64))
    #w3= tf.Variable(tf.get_variable('w3',[hid_2, output_node],dtype=tf.float64))
b1= tf.Variable(tf.zeros([hid_1],dtype=tf.float64))
#b2= tf.Variable(tf.zeros([hid_2],dtype=tf.float64))
b2= tf.Variable(tf.zeros([output_node],dtype=tf.float64))
weights= [w1,w2]
biases = [b1,b2]
```

In [83]:
```
input_feat = tf.placeholder(tf.float64, shape=[None, train.shape[1]])
output_labels=tf.placeholder(tf.float64, shape=[None, 1])
hlayer_1 = tf.add(tf.matmul(input_feat, weights[0]),biases[0])
#hlayer_2 = tf.add(tf.matmul(hlayer_1, weights[1]),biases[1])
#hlayer_1 =tf.nn.relu(hlayer_1)
out = tf.add(tf.matmul(hlayer_1, weights[1]),biases[1])
mse_loss = tf.losses.mean_squared_error(labels=output_labels, predictions=out)
mse_loss=tf.cast(mse_loss,tf.float64)



reg_const=np.float64(reg_const)
_reg=0.5*tf.reduce_sum(tf.square(weights[0]))   #Applying L2 regularization formula
_reg+=0.5*tf.reduce_sum(tf.square(weights[1]))
reg=tf.cast(_reg,tf.float64)
reg_loss =tf.multiply(np.float64(0.5),tf.add(mse_loss,tf.multiply(reg_const,reg)))
optimizer = tf.train.RMSPropOptimizer(LearningRate)
func_to_opt = optimizer.minimize(reg_loss)

#Running sessions
sess = tf.Session()
init = tf.global_variables_initializer()
sess.run(init)

#Setting BatchSize and splitting value for validation
no_of_batch=8
no_of_train_batch=int(0.99*no_of_batch)
train_batch=np.array_split(train,no_of_batch)
labels_of_train_batch=np.array_split(train_labels,no_of_batch)

#Training and validation phase
loss_tr=[]
loss_va=[]

for i in range(epoch):

    train_loss=0
    for j in range(0,len(train_batch)):
        single_train_batch={input_feat:train_batch[j],output_labels:labels_of_train_batch[j]}
        _= sess.run((func_to_opt),feed_dict=single_train_batch)
        trainbatch_loss = sess.run((mse_loss),feed_dict=single_train_batch)
        train_loss+=trainbatch_loss
```

```
    train_loss=train_loss/(j+1)

    val_loss=0
    for k in range(no_of_train_batch,no_of_batch):
        single_cv_batch={input_feat:train_batch[k],output_labels:labels_of_train_batch[k]}
        valid_loss = sess.run((mse_loss),feed_dict=single_cv_batch)
        val_loss+=valid_loss

    val_loss=val_loss/(no_of_batch-no_of_train_batch)
    loss_tr.append(train_loss)
    loss_va.append(val_loss)

    print('epoc:',i,'train_loss',train_loss,'valid_loss',val_loss)
```

```
epoc: 0 train_loss 7293.5238037109375 valid_loss 6509.58837890625
epoc: 1 train_loss 7160.042541503906 valid_loss 6363.44677734375
epoc: 2 train_loss 7018.608642578125 valid_loss 6197.16064453125
epoc: 3 train_loss 6858.920227050781 valid_loss 6008.31640625
epoc: 4 train_loss 6680.4144287109375 valid_loss 5798.244140625
epoc: 5 train_loss 6484.1453857421875 valid_loss 5569.18310546875
epoc: 6 train_loss 6271.630554199219 valid_loss 5323.56787109375
epoc: 7 train_loss 6044.532470703125 valid_loss 5063.82373046875
epoc: 8 train_loss 5804.561340332031 valid_loss 4792.32080078125
epoc: 9 train_loss 5553.44873046875 valid_loss 4511.38916015625
epoc: 10 train_loss 5292.947998046875 valid_loss 4223.33544921875
epoc: 11 train_loss 5024.838073730469 valid_loss 3930.458984375
epoc: 12 train_loss 4750.927062988281 valid_loss 3635.062744140625
epoc: 13 train_loss 4473.054504394531 valid_loss 3339.451904296875
epoc: 14 train_loss 4193.087921142578 valid_loss 3045.9296875
epoc: 15 train_loss 3912.9186096191406 valid_loss 2756.790283203125
epoc: 16 train_loss 3634.4530029296875 valid_loss 2474.303466796875
epoc: 17 train_loss 3359.601104736328 valid_loss 2200.6953125
epoc: 18 train_loss 3090.262725830078 valid_loss 1938.1279296875
epoc: 19 train_loss 2828.309295654297 valid_loss 1688.66845703125
epoc: 20 train_loss 2575.5592498779297 valid_loss 1454.25439453125
epoc: 21 train_loss 2333.7511596679688 valid_loss 1236.648681640625
epoc: 22 train_loss 2104.5075073242188 valid_loss 1037.3870849609375
epoc: 23 train_loss 1889.2919082641602 valid_loss 857.7140502929688
epoc: 24 train_loss 1689.3552932739258 valid_loss 698.5084838867188
epoc: 25 train_loss 1505.6687088012695 valid_loss 560.2006225585938
epoc: 26 train_loss 1338.8386039733887 valid_loss 442.6829528808594
epoc: 27 train_loss 1189.005844116211 valid_loss 345.227783203125
epoc: 28 train_loss 1055.7462005615234 valid_loss 266.4417724609375
epoc: 29 train_loss 938.0337924957275 valid_loss 204.3233184814453
epoc: 30 train_loss 834.3533382415771 valid_loss 156.48695373535156
epoc: 31 train_loss 742.982307434082 valid_loss 120.49945068359375
epoc: 32 train_loss 662.2971839904785 valid_loss 94.1455078125
epoc: 33 train_loss 590.9215469360352 valid_loss 75.50607299804688
epoc: 34 train_loss 527.70490026474 valid_loss 62.89957809448242
epoc: 35 train_loss 471.6435794830322 valid_loss 54.7967529296875
epoc: 36 train_loss 421.8201560974121 valid_loss 49.78431701660156
epoc: 37 train_loss 377.37729120254517 valid_loss 46.60800552368164
epoc: 38 train_loss 337.52454900741577 valid_loss 44.27692794799805
epoc: 39 train_loss 301.56993103027344 valid_loss 42.15919494628906
epoc: 40 train_loss 268.95394134521484 valid_loss 39.982940673828125
epoc: 41 train_loss 239.2594747543335 valid_loss 37.727806091308594
epoc: 42 train_loss 212.18968391418457 valid_loss 35.47979736328125
```

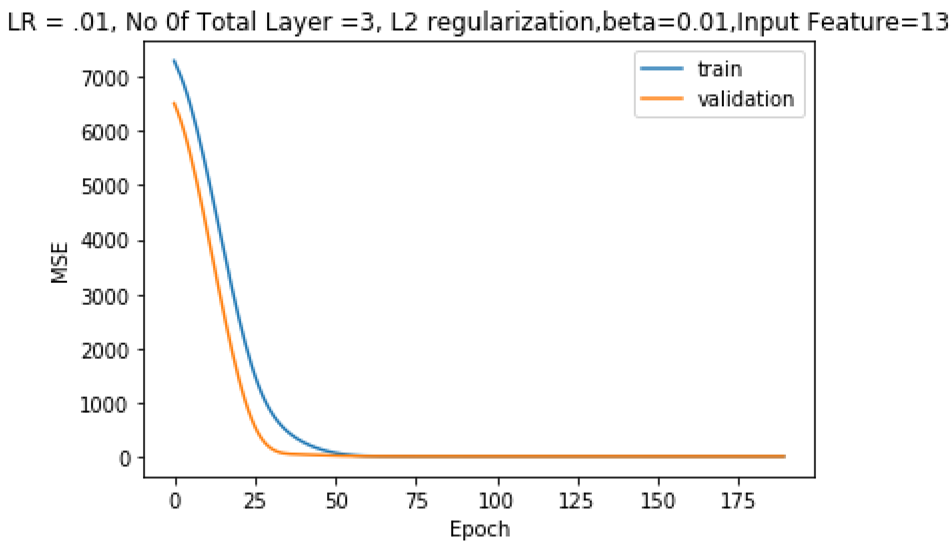
```
epoc: 43 train_loss 187.52966785430908 valid_loss 33.325462341308594
epoc: 44 train_loss 165.11074042320251 valid_loss 31.303892135620117
epoc: 45 train_loss 144.78569769859314 valid_loss 29.39923667907715
epoc: 46 train_loss 126.41471791267395 valid_loss 27.55634880065918
epoc: 47 train_loss 109.85985541343689 valid_loss 25.712064743041992
epoc: 48 train_loss 94.98648190498352 valid_loss 23.831579208374023
epoc: 49 train_loss 81.6700177192688 valid_loss 21.926712036132812
epoc: 50 train_loss 69.80389881134033 valid_loss 20.04254150390625
epoc: 51 train_loss 59.299668073654175 valid_loss 18.229778289794922
epoc: 52 train_loss 50.0783908367157 valid_loss 16.52713966369629
epoc: 53 train_loss 42.061134457588196 valid_loss 14.957253456115723
epoc: 54 train_loss 35.16475212574005 valid_loss 13.530200958251953
epoc: 55 train_loss 29.30135142803192 valid_loss 12.24791431427002
epoc: 56 train_loss 24.380082726478577 valid_loss 11.1078519821167
epoc: 57 train_loss 20.308936715126038 valid_loss 10.105491638183594
epoc: 58 train_loss 16.995645821094513 valid_loss 9.236480712890625
epoc: 59 train_loss 14.347766697406769 valid_loss 8.498221397399902
epoc: 60 train_loss 12.272662699222565 valid_loss 7.890121936798096
epoc: 61 train_loss 10.678402930498123 valid_loss 7.4113945960998535
epoc: 62 train_loss 9.47589522600174 valid_loss 7.056637287139893
epoc: 63 train_loss 8.58216443657875 valid_loss 6.812231540679932
epoc: 64 train_loss 7.9237717390060425 valid_loss 6.65646505355835
epoc: 65 train_loss 7.439444035291672 valid_loss 6.564224720001221
epoc: 66 train_loss 7.0810607969760895 valid_loss 6.512794017791748
epoc: 67 train_loss 6.812735706567764 valid_loss 6.485171794891357
epoc: 68 train_loss 6.608626276254654 valid_loss 6.470211505889893
epoc: 69 train_loss 6.450453609228134 valid_loss 6.461246013641357
epoc: 70 train_loss 6.325333505868912 valid_loss 6.454488754272461
epoc: 71 train_loss 6.2241829335689545 valid_loss 6.447904109954834
epoc: 72 train_loss 6.140556156635284 valid_loss 6.440446376800537
epoc: 73 train_loss 6.069846212863922 valid_loss 6.431612014770508
epoc: 74 train_loss 6.008780211210251 valid_loss 6.421260833740234
epoc: 75 train_loss 5.954988241195679 valid_loss 6.409404277801514
epoc: 76 train_loss 5.906755119562149 valid_loss 6.396173000335693
epoc: 77 train_loss 5.862841367721558 valid_loss 6.381738662719727
epoc: 78 train_loss 5.822345465421677 valid_loss 6.3662800788879395
epoc: 79 train_loss 5.784592181444168 valid_loss 6.349974155426025
epoc: 80 train_loss 5.749099761247635 valid_loss 6.333030700683594
epoc: 81 train_loss 5.715492755174637 valid_loss 6.315587997436523
epoc: 82 train_loss 5.6835010051727295 valid_loss 6.297827243804932
epoc: 83 train_loss 5.652906388044357 valid_loss 6.2798309326171875
epoc: 84 train_loss 5.6235582530498505 valid_loss 6.261743068695068
epoc: 85 train_loss 5.595324903726578 valid_loss 6.243649959564209
```

```
epoc: 86 train_loss 5.568108767271042 valid_loss 6.2256317138671875
epoc: 87 train_loss 5.541827440261841 valid_loss 6.207754135131836
epoc: 88 train_loss 5.516421854496002 valid_loss 6.1900482177734375
epoc: 89 train_loss 5.491834461688995 valid_loss 6.1725850105285645
epoc: 90 train_loss 5.468025267124176 valid_loss 6.155374050140381
epoc: 91 train_loss 5.444953352212906 valid_loss 6.138452053070068
epoc: 92 train_loss 5.422582983970642 valid_loss 6.12183141708374
epoc: 93 train_loss 5.400891721248627 valid_loss 6.105523586273193
epoc: 94 train_loss 5.379852592945099 valid_loss 6.089547634124756
epoc: 95 train_loss 5.359440743923187 valid_loss 6.073886394500732
epoc: 96 train_loss 5.339642375707626 valid_loss 6.058574676513672
epoc: 97 train_loss 5.320427775382996 valid_loss 6.043565273284912
epoc: 98 train_loss 5.301790118217468 valid_loss 6.028879165649414
epoc: 99 train_loss 5.283708482980728 valid_loss 6.0145134925842285
epoc: 100 train_loss 5.266165137290955 valid_loss 6.000447750091553
epoc: 101 train_loss 5.249149113893509 valid_loss 5.9866790771484375
epoc: 102 train_loss 5.232652068138123 valid_loss 5.973211288452148
epoc: 103 train_loss 5.216654539108276 valid_loss 5.960031032562256
epoc: 104 train_loss 5.201140642166138 valid_loss 5.947103500366211
epoc: 105 train_loss 5.18610817193985 valid_loss 5.934450626373291
epoc: 106 train_loss 5.17153537273407 valid_loss 5.9220428466796875
epoc: 107 train_loss 5.157418519258499 valid_loss 5.909882068634033
epoc: 108 train_loss 5.143744736909866 valid_loss 5.897955417633057
epoc: 109 train_loss 5.130502939224243 valid_loss 5.8862433433532715
epoc: 110 train_loss 5.1176818907260895 valid_loss 5.874758243560791
epoc: 111 train_loss 5.105273485183716 valid_loss 5.8634772300720215
epoc: 112 train_loss 5.0932663679122925 valid_loss 5.852396011352539
epoc: 113 train_loss 5.081654340028763 valid_loss 5.841506481170654
epoc: 114 train_loss 5.070422917604446 valid_loss 5.8308024406433105
epoc: 115 train_loss 5.059564679861069 valid_loss 5.820271015167236
epoc: 116 train_loss 5.049068599939346 valid_loss 5.809896945953369
epoc: 117 train_loss 5.038933366537094 valid_loss 5.799707889556885
epoc: 118 train_loss 5.029140323400497 valid_loss 5.789649486541748
epoc: 119 train_loss 5.019687354564667 valid_loss 5.779750823974609
epoc: 120 train_loss 5.010562092065811 valid_loss 5.769991397857666
epoc: 121 train_loss 5.001759350299835 valid_loss 5.760385036468506
epoc: 122 train_loss 4.993270128965378 valid_loss 5.7509026527404785
epoc: 123 train_loss 4.9850857853889465 valid_loss 5.7415547370910645
epoc: 124 train_loss 4.977194905281067 valid_loss 5.732325077056885
epoc: 125 train_loss 4.969598293304443 valid_loss 5.723221302032471
epoc: 126 train_loss 4.96228152513504 valid_loss 5.714224338531494
epoc: 127 train_loss 4.955239802598953 valid_loss 5.705348491668701
epoc: 128 train_loss 4.948465526103973 valid_loss 5.696584224700928
```

```
epoc: 129 train_loss 4.941949516534805 valid_loss 5.687927722930908
epoc: 130 train_loss 4.935685694217682 valid_loss 5.679368495941162
epoc: 131 train_loss 4.929670214653015 valid_loss 5.6709136962890625
epoc: 132 train_loss 4.923889368772507 valid_loss 5.662558078765869
epoc: 133 train_loss 4.9183478355407715 valid_loss 5.654300689697266
epoc: 134 train_loss 4.913029342889786 valid_loss 5.646145343780518
epoc: 135 train_loss 4.907924145460129 valid_loss 5.6380839347839355
epoc: 136 train_loss 4.90303772687912 valid_loss 5.6301045417785645
epoc: 137 train_loss 4.898356109857559 valid_loss 5.622209072113037
epoc: 138 train_loss 4.893877029418945 valid_loss 5.614409923553467
epoc: 139 train_loss 4.889594078063965 valid_loss 5.606703281402588
epoc: 140 train_loss 4.885498255491257 valid_loss 5.599067211151123
epoc: 141 train_loss 4.881581783294678 valid_loss 5.591522693634033
epoc: 142 train_loss 4.877849251031876 valid_loss 5.584065914154053
epoc: 143 train_loss 4.874290108680725 valid_loss 5.576694965362549
epoc: 144 train_loss 4.870893090963364 valid_loss 5.569387912750244
epoc: 145 train_loss 4.867658644914627 valid_loss 5.562183380126953
epoc: 146 train_loss 4.86458221077919 valid_loss 5.55504035949707
epoc: 147 train_loss 4.861660420894623 valid_loss 5.547998428344727
epoc: 148 train_loss 4.8588807284832 valid_loss 5.541007995605469
epoc: 149 train_loss 4.85624586045742 valid_loss 5.53411865234375
epoc: 150 train_loss 4.853750303387642 valid_loss 5.527305603027344
epoc: 151 train_loss 4.851383522152901 valid_loss 5.520563125610352
epoc: 152 train_loss 4.849145278334618 valid_loss 5.513898849487305
epoc: 153 train_loss 4.847033992409706 valid_loss 5.5073161125183105
epoc: 154 train_loss 4.845040380954742 valid_loss 5.500809192657471
epoc: 155 train_loss 4.843167200684547 valid_loss 5.494377613067627
epoc: 156 train_loss 4.841401174664497 valid_loss 5.488021373748779
epoc: 157 train_loss 4.8397476226091385 valid_loss 5.481753826141357
epoc: 158 train_loss 4.838194563984871 valid_loss 5.475555419921875
epoc: 159 train_loss 4.8367423713207245 valid_loss 5.469430446624756
epoc: 160 train_loss 4.835385218262672 valid_loss 5.463369369506836
epoc: 161 train_loss 4.834127306938171 valid_loss 5.457394123077393
epoc: 162 train_loss 4.832954943180084 valid_loss 5.4515061378479
epoc: 163 train_loss 4.831870466470718 valid_loss 5.445685863494873
epoc: 164 train_loss 4.830869436264038 valid_loss 5.439932346343994
epoc: 165 train_loss 4.8299490958452225 valid_loss 5.43426513671875
epoc: 166 train_loss 4.829103007912636 valid_loss 5.42865514755249
epoc: 167 train_loss 4.828333184123039 valid_loss 5.423145294189453
epoc: 168 train_loss 4.827634647488594 valid_loss 5.417701721191406
epoc: 169 train_loss 4.8270028829574585 valid_loss 5.412308216094971
epoc: 170 train_loss 4.826435565948486 valid_loss 5.407015323638916
epoc: 171 train_loss 4.8259323835372925 valid_loss 5.40179443359375
```

```
epoc: 172 train_loss 4.825492277741432 valid_loss 5.396631717681885
epoc: 173 train_loss 4.825107082724571 valid_loss 5.391557693481445
epoc: 174 train_loss 4.8247784078121185 valid_loss 5.386549472808838
epoc: 175 train_loss 4.824501991271973 valid_loss 5.381622314453125
epoc: 176 train_loss 4.824275970458984 valid_loss 5.376763820648193
epoc: 177 train_loss 4.824097260832787 valid_loss 5.371983051300049
epoc: 178 train_loss 4.823964610695839 valid_loss 5.367246150970459
epoc: 179 train_loss 4.823875680565834 valid_loss 5.362612247467041
epoc: 180 train_loss 4.823832035064697 valid_loss 5.358036041259766
epoc: 181 train_loss 4.823827281594276 valid_loss 5.353527545928955
epoc: 182 train_loss 4.823859557509422 valid_loss 5.349094867706299
epoc: 183 train_loss 4.823930606245995 valid_loss 5.344730854034424
epoc: 184 train_loss 4.82403028011322 valid_loss 5.3404388427734375
epoc: 185 train_loss 4.824165374040604 valid_loss 5.336204528808594
epoc: 186 train_loss 4.8243376314640045 valid_loss 5.332062244415283
epoc: 187 train_loss 4.824533686041832 valid_loss 5.327972888946533
epoc: 188 train_loss 4.824758887290955 valid_loss 5.323946952819824
epoc: 189 train_loss 4.82501120865345 valid_loss 5.319984436035156
```

In [84]:
```
#Plotting
plt.title('LR = .01, No Of Total Layer =3, L2 regularization,beta=0.01,Input Feature=13')
plt.plot(loss_tr[:], label = 'train')
plt.plot(loss_va[:],label='validation')
plt.xlabel('Epoch')
plt.ylabel('MSE')
plt.legend()
plt.show()
```



In [85]:
```
#Prediction on test data
in_test={input_feat:np.array(test)}
pred_test = sess.run((out),feed_dict=in_test)
pred_test = pred_test.reshape(-1)
for i in range(len(pred_test)):
    pred_test[i]=int(pred_test[i])
```

In [88]:
```
r2_test_NN = r2_score(test_labels,pred_test)
print('r2 score by NN:',r2_test_NN)
```

```
r2 score by NN: 0.8760932944606414
```

In [89]:
```
in_train={input_feat:np.array(train)}
pred = sess.run((out),feed_dict=in_train)
pred = pred.reshape(-1)
for i in range(len(pred)):
    pred[i]=int(pred[i])
```

In [90]:
```
r2_train_NN = r2_score(train_labels,pred)
print('r2 score by NN:',r2_train_NN)
```

```
r2 score by NN: 0.8101426755191411
```

Problem 3-2: (10pts)

Compare the performance (via R^2 score) of the 3 implementations above: the 600 Degree Rule, the DTS method, and the neural network approach. For all methods, and each test year, plot the predicted date vs. the actual date. Discuss the accuracy and differences of these 3 models.

Comparison between 3 implementations by R2 Score(On test data):

1. 600 Degree rule: 0.67930
2. DTS method: 0.91253
3. Neural network approach: 0.87609

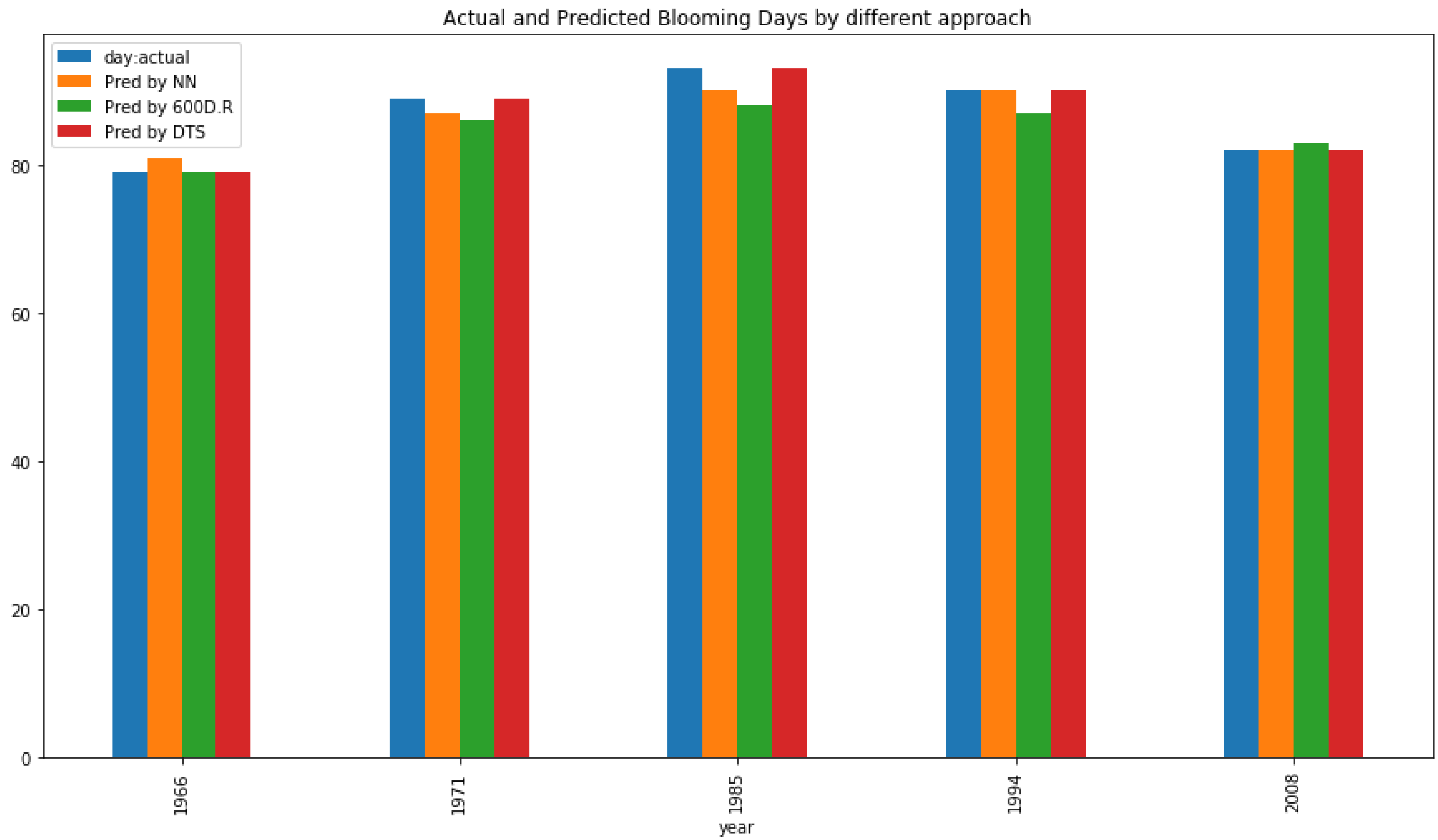
In [91]:
```
comparison_df = pd.DataFrame({'day:actual':test_labels,'year':test_years,'Pred by NN':pred_test,'Pred by 600
D.R':pred_test_600,'Pred by DTS':ACT_BD_optEa})
comparison_df.head()
```

Out[91]:

	Pred by 600D.R	Pred by DTS	Pred by NN	day:actual	year
0	79	79	81.0	79	1966
1	86	89	87.0	89	1971
2	88	93	90.0	93	1985
3	87	90	90.0	90	1994
4	83	82	82.0	82	2008

In [92]:
```
comparison_df.plot(x='year',y=['day:actual','Pred by NN','Pred by 600D.R','Pred by DTS'],kind='bar',figsize=(
15,8),title='Actual and Predicted Blooming Days by different approach')
```

Out[92]: <matplotlib.axes._subplots.AxesSubplot at 0x121ba38d518>



Discussion:

As we have seen from the test scores, Tmean = 638 degree performed better tha 600 degree rule. DTS method performed best among all the approaches. NN approach could have performed even better if there were more data available for training. More precise domain analysis for sakura blooming factors could have resulted in better predictions. As we have seen in the NN approach, data manupulation and proper feature engineering improves the R2 scores significantly. So there is obvious scopes for better result using ANN. One more thing to note here is the difference of test scores and train scores. In any approach, only a good test score can not ensure that the claimed model is good enough because of very low number of test data. In this case it would be wise to choose a model that gives optimum good scores in both train and test dataset rather than choosing a model which gives good score in a small model but performs porrly in a large model.

---

4. Trends of the Sakura blooming phenomenon (20pts total)

Problem 4-1: (20pts)

Based on the data from the past 60 years, investigate and discuss trends in the sakura hibernation (D_j) and blooming (BD_j) phenomena in Tokyo.

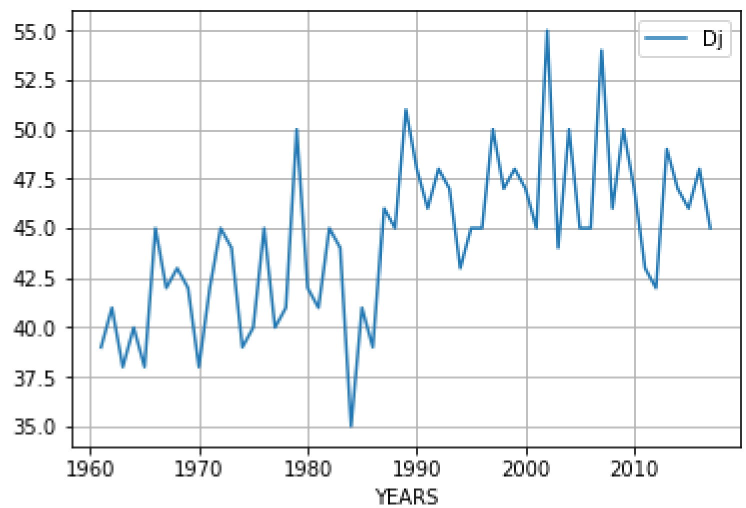
In [93]:
```
Dj_all['day:actual'] = ActualBD_df['day:actual']
print(Dj_all)
```

	Dj	YEARS	day:actual
0	39	1961	91
1	41	1962	91
2	38	1963	91
3	40	1964	93
4	38	1965	92
5	45	1966	79
6	42	1967	89
7	43	1968	89
8	42	1969	96
9	38	1970	97
10	42	1971	89
11	45	1972	88
12	44	1973	90
13	39	1974	92
14	40	1975	88
15	45	1976	82
16	40	1977	81
17	41	1978	90
18	50	1979	82
19	42	1980	91
20	41	1981	85
21	45	1982	82
22	44	1983	90
23	35	1984	102
24	41	1985	93
25	39	1986	93
26	46	1987	82
27	45	1988	93
28	51	1989	79
29	48	1990	79
30	46	1991	89
31	48	1992	84
32	47	1993	83
33	43	1994	90
34	45	1995	90
35	45	1996	91
36	50	1997	80
37	47	1998	86
38	48	1999	83
39	47	2000	90
40	45	2001	82
41	55	2002	75

42	44	2003	86
43	50	2004	78
44	45	2005	90
45	45	2006	80
46	54	2007	79
47	46	2008	82
48	50	2009	80
49	47	2010	81
50	43	2011	87
51	42	2012	91
52	49	2013	75
53	47	2014	84
54	46	2015	82
55	48	2016	81
56	45	2017	80

In [94]:
```
Dj_all.plot(x='YEARS',y='Dj',grid=True)
```

Out[94]: <matplotlib.axes._subplots.AxesSubplot at 0x121b5a4deb8>



Dj_all.plot(x='YEARS',y='day:actual',grid=True)

Comments on trend of blooming phenomena in Tokyo

As we can see from the graph plotted, blooming is occuring earlier than it used to 60 years ago. As we have mentioned earlier, cold winter temperature and warm spring temperature accelerates blooming date. We know one of the effects of global warming is colder winter and warmer summer. This can be the reason behind earlier blooming of sakura.[1]

Note

This challenge was given as an assignement in the AI training program of Hiperdyne Corporation. Please use the content only for learning purposes.Plagiarizing is strictly prohibited. -Sumaiya Saima,AI engineer,Hiperdyne.

References:

1. https://slate.com/news-and-politics/2007/03/how-do-horticulturists-know-when-the-cherry-blossoms-will-bloom.html (https://slate.com/news-and-politics/2007/03/how-do-horticulturists-know-when-the-cherry-blossoms-will-bloom.html)
2. https://www.jma.go.jp/jma/en/menu.html (https://www.jma.go.jp/jma/en/menu.html)
3. American Horticultural Society research

In [ ]: