Entrance Challenge: When Will the Sakura Bloom?

0. Basics of the Sakura Bloom-cycle (5pts total)

ID : A1803
Date of Submission : 03/03/2019

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. The 3 models to be applied to the data are described below.

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).

In [ ]:
```

```

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.

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.

Importing Libraries

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

Getting Data Ready

```
1. Function : get_yearly_dataframes
-input : data frame- read from csv, years to groupby
-output : yearly grouped dataframes , A dictionary where keys is the year an
d value is the dataframe of the corresponding year's data.

2. Functtion: add_day_counts

-input : yearly grouped dataframes as dictionary.
-output : yearly grouped dataframes with daycount addes as 'days' column

3. Functtion: add_day_counts

 -input : yearly grouped dataframes as dictionary.
 -output : yearly grouped dataframes with accumulated temps (February 1 to e
d of year) addes as 'acc_temp' column
```

In [2]:
```
data_df=pd.read_csv('sakura.csv')
test_years= [1966, 1971, 1985, 1994,2008]
years=data_df['year'].unique()
train_years=[]
for year in years:
    if year not in test_years:
        train_years.append(year)

def get_yearly_dfs(df,year):
    year_gp=data_df.groupby('year')
    year_dfs={}
    for year in years:
        df=year_gp.get_group(year)
        df=df.reset_index()
        year_dfs[year]=df
    return year_dfs

def add_day_counts(yearly_dfs):
    yr_dfs=yearly_dfs.copy()
    for year in years:
        df=yr_dfs[year]
        ind=df.index.values.copy()
        ind+=1
        df['days']=ind
        yr_dfs[year]=df
    return yr_dfs
def add_acc_temps(yearly_dfs):
    yr_dfs=yearly_dfs.copy()
    for year in years:
        df=yr_dfs[year]
        acc_temp=0
        df['acc_temp']=0
        for row in range(len(df)):
            df_row=df.loc[row].copy()
            if df_row['month']>=2 and df_row['month']<=5:
                acc_temp+=df_row['max temp']
                df_row['acc_temp']=acc_temp
                df.loc[row]=df_row
        yr_dfs[year]=df
    return yr_dfs
```

In [3]:
```
yearly_dfs=get_yearly_dfs(data_df,years)
yearly_dfs=add_day_counts(yearly_dfs)
yearly_dfs=add_acc_temps(yearly_dfs)
```

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).

```
1. Function 'get_true_label'

  - input yearly grouped data frames
  - output : true labels

2. Function : 'get_acc_temp'

  - input yearly grouped data frames
  - output : accumulated temperature data frame
```

In [4]:
```
def get_true_label(year_dfs,yrs):
    true_lb=[]
    for year in yrs:
        df=year_dfs[year].copy()
        ind= (df['bloom']==1)
        ind=np.argmax(np.array(ind))
        true_lb.append(df.loc[ind,'days'])

    true_lb=pd.DataFrame(true_lb,index=yrs,columns=['true_days'])
    return true_lb
def get_acc_temp(year_dfs,yrs):
    acc_df=[]
    for year in yrs:
        df=year_dfs[year].copy()
        ind= (df['bloom']==1)
        ind=np.argmax(np.array(ind))
        acc_df.append(df.loc[ind,'acc_temp'])

    acc_df =pd.DataFrame(acc_df,index=yrs,columns=['temp'])
    return acc_df


acc_df=get_acc_temp(yearly_dfs,train_years)
print('Mean accumulated temperature\n',acc_df.mean(),'\n')
thr_temp=acc_df.mean()
```

```
Mean accumulated temperature
 temp    638.355769
dtype: float64
```

```
Function predict :
-input : yearly dataframes, years , thr_temp
-output : pred_days as dataframe
```

In [5]:
```
def predict(yearly_dfs,yrs,thr_temp):

    pred_lb=[]
    for year in yrs:
        _df=yearly_dfs[year]
        for i in range(len(_df)):
            if (_df.loc[i,'acc_temp']>thr_temp):
                pred_lb.append(_df.loc[i,'days'])
                break
    pred_lb=pd.DataFrame(pred_lb,index=yrs,columns=['pred_days'])
    return pred_lb
true_lb=get_true_label(yearly_dfs,test_years)
pred_lb_mean=predict(yearly_dfs,test_years,float(thr_temp))
pred_lb_600=predict(yearly_dfs,test_years,float(600))
```

In [6]:
```
import numpy as np
from sklearn.metrics import r2_score, mean_squared_error
#lbs=pd.concat([true_lb,pred_lb],axis=1)
true_lb=np.array(true_lb)
pred_lb_mean=np.array(pred_lb_mean)
pred_lb_600=np.array(pred_lb_600)

err_mean=np.square(np.array(true_lb)-np.array(pred_lb_mean))
err_600=np.square(np.array(true_lb)-np.array(pred_lb_600))
r2_1=r2_score(true_lb,pred_lb_mean)
r2_2=r2_score(true_lb,pred_lb_600)

print('mean_squared_error\nerror_thr_temp',np.mean(err_mean),'\nerror
_600',np.mean(err_600))
print('r2_score\n',r2_1,'\n',r2_2)
```

```
mean_squared_error
error_thr_temp 4.6
error_600 8.8
r2_score
 0.8323615160349854
 0.6793002915451896
```

Results Mehod 600 degree rule :

```
R2 Score for mean accumulated temp found : 638.355769
R2 score for 600 deg        : 0.6793002915451896
R2 Score for mean acc. temp :  0.8323615160349854
```

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.

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:

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

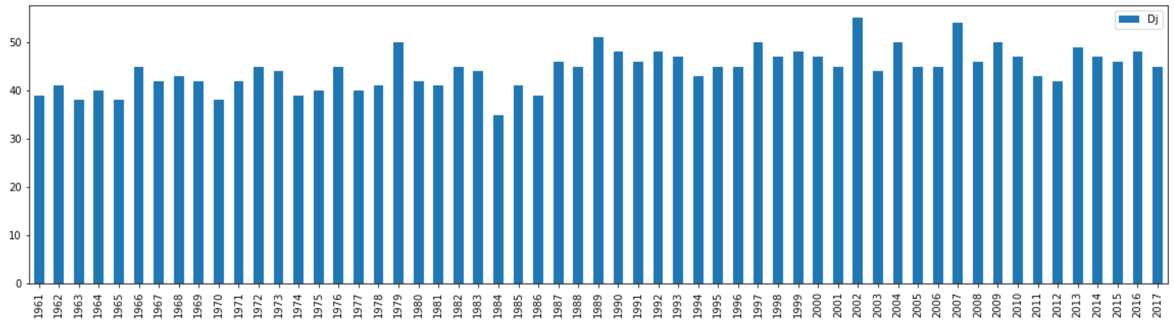
where ϕ is the latitude [°N], L is the distance from the nearest coastline [km], and T_F is that location's average temperature [°C] over the first 3 months of a given year. In the case of Tokyo, $\phi = 35^\circ 40'$ and $L = 4\text{km}$.

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.)

```
Function : get_djs
    -input : yearly  grouped dataframes
    -ouput : dataframe of Djs year as index
```

In [7]:
```
def get_Djs(yerly_dfs,years):
    Fi=35.67
    L=4
    Djs=[]
    for year in years:
        _df=yearly_dfs[year]
        Tf=_df[_df['month']<=3]['avg temp'].mean()
        Dj=136.75- 7.689*Fi+ 0.133*(Fi**2)- 1.307*np.log(L)+ 0.144*Tf
+ 0.285*(Tf**2)
        Djs.append(int(Dj))

    Dj_df=pd.DataFrame(Djs,index=years,columns=['Dj'])
    return Dj_df
fig, axes = plt.subplots(nrows=1, ncols=1,figsize=(20,5))
Dj_df=get_Djs(yearly_dfs,years)
Dj_df.plot.bar(ax=axes)
#Dj_df.plot(ax=axes)
plt.show()
```



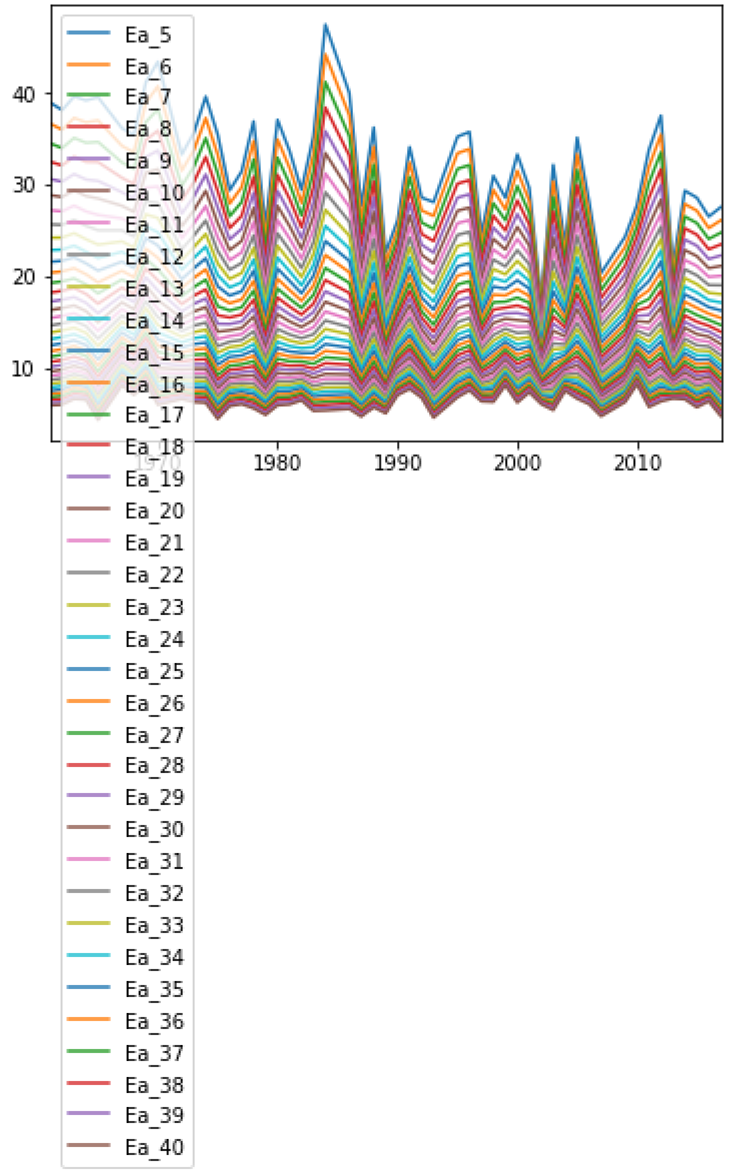
Problem 2-2: (10pts)

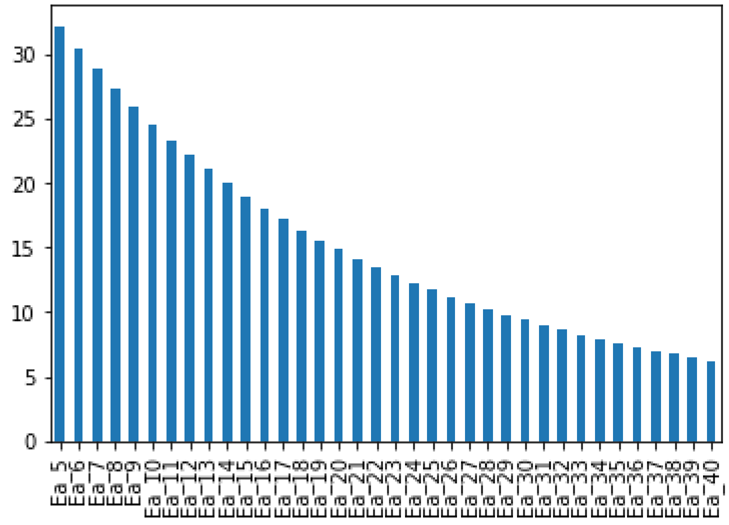
Calculate DTS_j for each year j in the training set for discrete values of E_a, varying from 5 to 40kcal ($E_a = 5, 6, 7, \cdots, 40\,\mathrm{kcal}$), and plot this DTS_j against E_a. Also calculate the average of DTS_j over the training period, and indicate it on the plot as DTS_{mean}. Pay attention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

```
Function : get_ts
-input : Ea, Avg temperature (Tj)
-output: ts
Function:
-input: yearly grouped dataframes
-ouput: data frame of DTS- years as index
```

In [8]:
```
def get_ts(Ea,Tj):
        R=8.314
        Ts=17
        A=Ea*(Tj-Ts)*4184
        B=R*(Tj+273)*(Ts+273)
        C=A/B
        return np.exp(C)

Bdj_df=get_true_label(yearly_dfs,years)

def get_dts(yearly_dfs,Ea,yrs):
    DTS_df=[]
    for year in yrs:
        _df=yearly_dfs[year]

        Dj=Dj_df.loc[year,'Dj']
        Bdj=Bdj_df.loc[year,'true_days']

        indx=_df['days']>=int(Dj)
        indx&=_df['days']<=int(Bdj)
        Tjs=_df[indx]['avg temp']

        DTS=0
        for Tj in Tjs:
            DTS+=get_ts(Ea,Tj)
        DTS_df.append(DTS)
    DTS_df=pd.DataFrame(DTS_df,index=yrs,columns=['Ea_'+str(Ea)])
    return DTS_df

DTS_df=get_dts(yearly_dfs,5,train_years)
#Eas=list([0.001,0.002,0.003])
Eas=list(range(6,41))
for Ea in Eas:
    DTS_df=pd.concat([DTS_df,get_dts(yearly_dfs,Ea,train_years)],axis
=1)
```

In [9]:
```
DTS_df.plot()
plt.show()
DTS_df.mean().plot.bar()
plt.show()
#DTS_df.mean()
```


Ea_5
Ea_6
Ea_7
Ea_8
Ea_9
Ea_10
Ea_11
Ea_12
Ea_13
Ea_14
Ea_15
Ea_16
Ea_17
Ea_18
Ea_19
Ea_20
Ea_21
Ea_22
Ea_23
Ea_24
Ea_25
Ea_26
Ea_27
Ea_28
Ea_29
Ea_30
Ea_31
Ea_32
Ea_33
Ea_34
Ea_35
Ea_36
Ea_37
Ea_38
Ea_39
Ea_40
40
30
20
10
1970
1980
1990
2000
2010


30
25
20
15
10
5
0
Ea_5
Ea_6
Ea_7
Ea_8
Ea_9
Ea_10
Ea_11
Ea_12
Ea_13
Ea_14
Ea_15
Ea_16
Ea_17
Ea_18
Ea_19
Ea_20
Ea_21
Ea_22
Ea_23
Ea_24
Ea_25
Ea_26
Ea_27
Ea_28
Ea_29
Ea_30
Ea_31
Ea_32
Ea_33
Ea_34
Ea_35
Ea_36
Ea_37
Ea_38
Ea_39
Ea_40

```
Function : predict_ts
-input :yearly grouped dataaframes
-output : prediced days dataframe - year as index
```

In [10]:
```
def predict_ts(yearly_dfs,Ea,yrs,DTS_mean):
    pred_df=[]
    for year in yrs:
        _df=yearly_dfs[year]

        Dj=Dj_df.loc[year,'Dj']
        indx=_df['days']>=int(Dj)
        Tjs=_df[indx]['avg temp']

        DTS=0
        ind=Dj-1
        for Tj in Tjs:
            DTS+=get_ts(Ea,Tj)
            if DTS>=DTS_mean['Ea_'+str(Ea)] or ind>=(len(_df)-1):
                pred_Bdj=_df.loc[ind,'days']
                break
            ind+=1


        pred_df.append(pred_Bdj)
    pred_df=pd.DataFrame(pred_df,index=yrs,columns=['Ea_'+str(Ea)])
    return pred_df
```

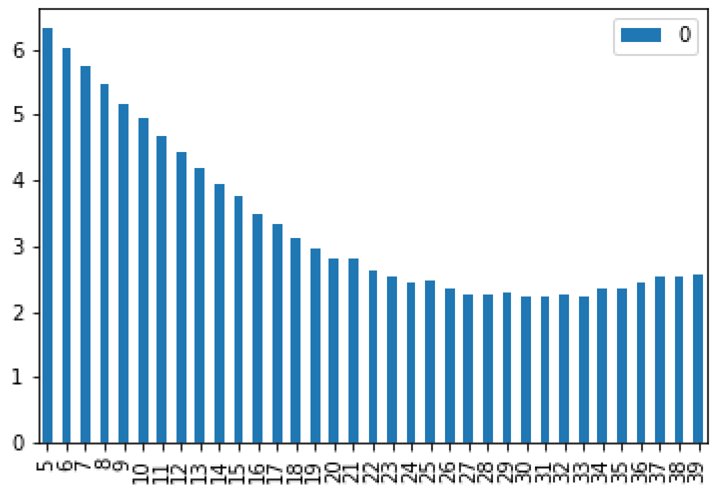
In [11]:
```
DTS_mean_df=DTS_df.mean()
pred_df=predict_ts(yearly_dfs,5,train_years,DTS_mean_df)
#Eas=list([0.001,0.002,0.003])
Eas=list(range(5,40))
for Ea in Eas[1:]:
    pred_df=pd.concat([pred_df,predict_ts(yearly_dfs,Ea,train_years,D
TS_df.mean())],axis=1)
```

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 [12]:
```
Bdj_df=get_true_label(yearly_dfs,train_years)
err_df=[]

for Ea in Eas:
    err=0
    DTS=DTS_df.mean()['Ea_'+str(Ea)]
    for year in train_years:

        Dj=int(Dj_df.loc[year,'Dj'])
        pred_Bdj=pred_df.loc[year,'Ea_'+str(Ea)]
        true_Bdj=Bdj_df.loc[year,'true_days']

        err+=np.abs(true_Bdj-pred_Bdj)
    err/=len(train_years)
    #print(Ea,err)
    err_df.append(err)

err_df=pd.DataFrame(err_df,index=Eas)
err_df.plot.bar()
Opt_Ea=np.argmin(np.array(err_df))+5
Opt_DTS_mean=DTS_mean_df['Ea_'+str(Opt_Ea)]
print('Optimum Ea : ',Opt_Ea)
print('Optimum DTS mean :',Opt_DTS_mean)
```

```
Optimum Ea :  30
Optimum DTS mean : 9.37634242025077
```



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 [13]:
```
pred_df=predict_ts(yearly_dfs,Opt_Ea,test_years,DTS_df.mean())
Bdj_df=get_true_label(yearly_dfs,test_years)
err_df=[]
Ea=30
err=0

Bdj_df=np.array(Bdj_df)
pred_df=np.array(pred_df)
err=np.square(Bdj_df-pred_df)
err=np.mean(err)
r2_err=r2_score(Bdj_df,pred_df)

#err/=len(train_years)
print('Ea :',Ea,'\nmean_squared_error :',err,'\nr2 score :',r2_err)
#30 0.8 0.9708454810495627
```

```
Ea : 30
mean_squared_error : 0.8
r2 score : 0.9708454810495627
```

Results for DTS mean rule :

```
Optimum Ea :  30
Optimum DTS mean : 9.37634242025077
R2 Score     : 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.

Improvements :

1. Precipiation :

```
 -Precipitaion can complexly effect DTS , as during rain or cloud day-light
 and temperature decreases which can influence bloom.
 -Again certain amount of precipiation may also need for bloom to occur so c
an also be a parameter for DTS gain, importance of precipitation can also be
 observed by analysing it correlation with bloom day which is shown in NN se
ction, NN performance was imporved significantly by taking precipiation in a
ccount.<br>
```

2. Sun hours :

```
-Sun hours can be considered for getting dts as day light hours changes from
 day to day. Sun hours improved a the r2 score slightly in Neural network pr
ediction.
```

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

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.

Data Preprocessing

1. Converting features of a whole year(365/366 samples) to a single vector for adding feature vector.
2. Feature Selection
3. Highly Corelated feaure specially anayzing and processing.
4. Finally building the train , validation and test data for further analysis.

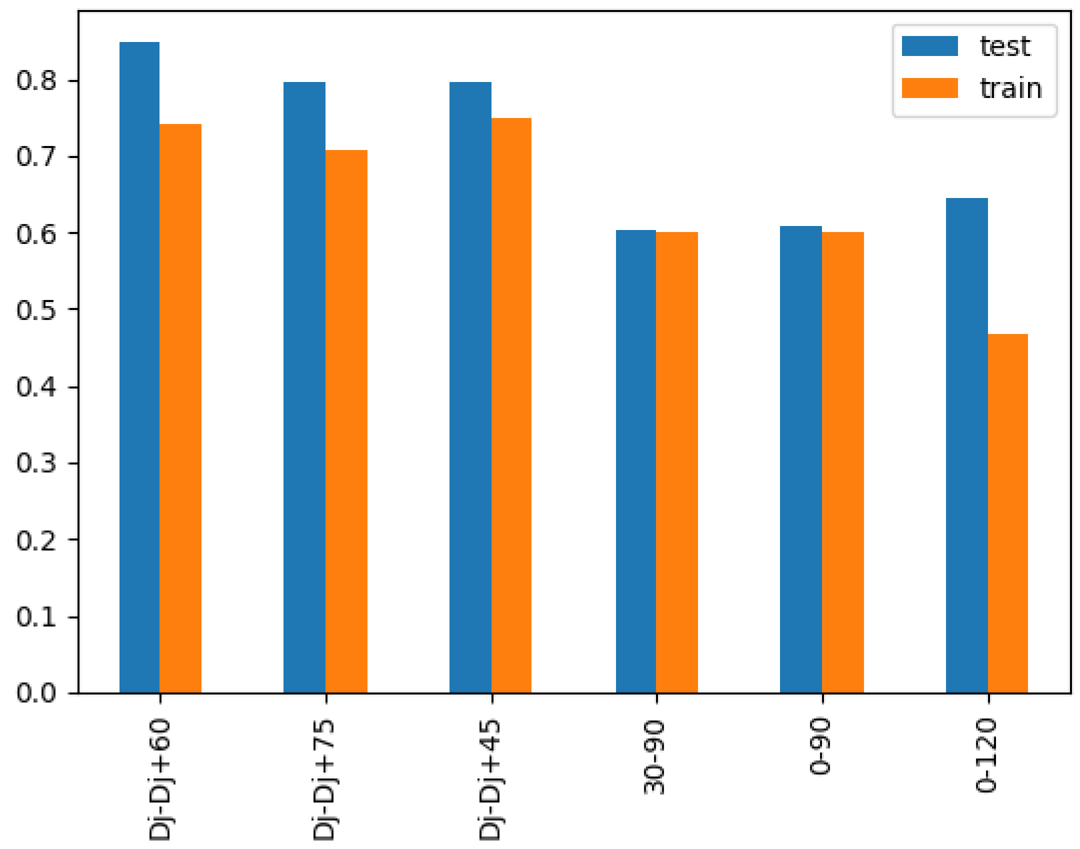
Converting each years data to a feature vector

Here some matters should be considered-

```
1. Days after bloom can be ignored. As weather before bloom can effect blooo
m date.
2. Days days before Dj- 'Last day of hibernation' is less significant
3. Feature with high correlation i.e max temp, hr1 preci can be specially co
nsidered for processing.
```

Considering this three issues several methods tried- As bloom typically occurs within march or 1st week of april so weather of the first 90 days of the years is significant.

```
1. Mean of first 90 days
2. Mean of 30-90 th days
3. mean of Dj-Dj+45 days
4. Mean Dj-Dj+60 days
5. Mean of Dj-Dj+75
```

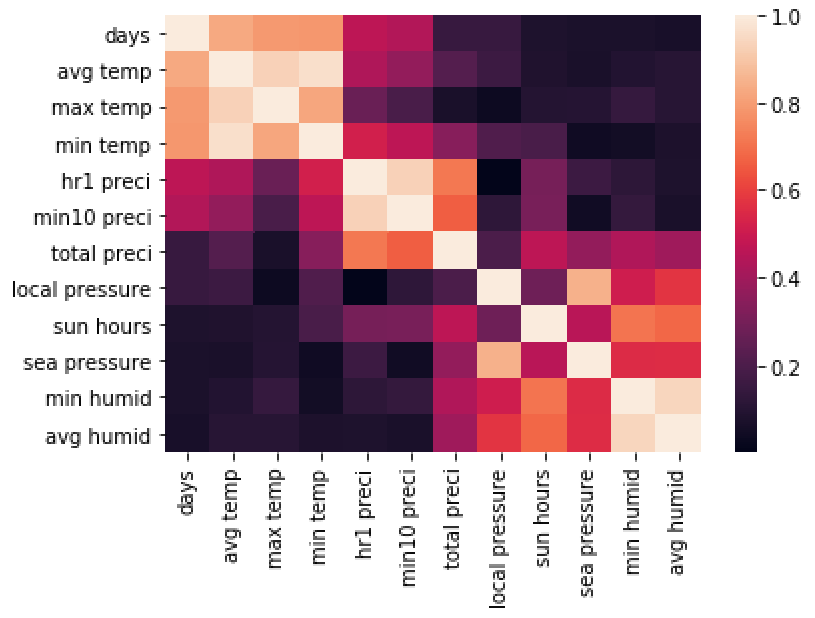


```
Function : get_vectors

Converts each years data to a single vector from a Dictionary of yearly data
frames where keys are
the years.

The mean is taken for Dj to Dj+60 days of a year. The reason is explained in
 the previous section.

1. Input- Dictionary of dataframes, years, features.
2. Output- features array,labels array,features dataframe
```

In [14]:
```
feats= [
'avg temp','max temp','min temp',
'total preci','hr1 preci','min10 preci',
'sun hours',
'local pressure','sea pressure',
'avg humid','min humid']

def get_vectors(dfs,_feats,yrs,scl=1):
    Bdj_df=get_true_label(yearly_dfs,yrs)
    _X=[]
    _Y=[]
    for year in yrs:
        _df=dfs[year]
        ind1=int(Dj-1)
        ind2=ind1+60
        df=_df.loc[ind1:ind2,:].copy()
        _X.append(df[_feats].mean())
        _Y.append(Bdj_df.loc[year])
    _X=np.array(_X)
    _Y=np.array(_Y)#-Dj_df.loc[year,'Dj'])

    n_df=pd.DataFrame(_X,columns=_feats)
    n_df['days']=_Y
    return _X,_Y,n_df

def plot_corr_sorted(df):
    corrmat = df.corr().abs()
    _corrmat=corrmat.sort_values( by=['days'],ascending=False)
    cm=(_corrmat['days'])
    cm=pd.DataFrame(cm)
    corr_cols=list(cm.index)
    sns.heatmap(df[corr_cols].corr().abs())
    return cm,corr_cols
```

Feature Selection

In [15]:
```
trainX,trainY,train_df=get_vectors(yearly_dfs,feats,train_years)
cm,_=plot_corr_sorted(train_df)
```



Obeservations

Following groups can be formed as they have same physical relation with bloom days. From the heatmap graph's high mutual correlation within these group members also reconfirms this issue. Based on Correlation and preformance on test and validation data one feautre is selected from one group.

```
1. 'avg temp','max temp','min temp', ------------------ max temp (performed
 best)
2. 'total preci','hr1 preci','min10 preci',------------ hr1 preci (performed
 best)
3. 'sun hours',---------------------------------------- sun hours (performed
 best)
4.'local pressure','sea pressure',-------------------- local pressure (perfo
rmed best)
5. 'avg humid','min humid'----------------------------- avg humid (performed
 best)
6. 'year' --------------------------------------------- year (performed bes
t)
```

In [16]:
```
cm
```

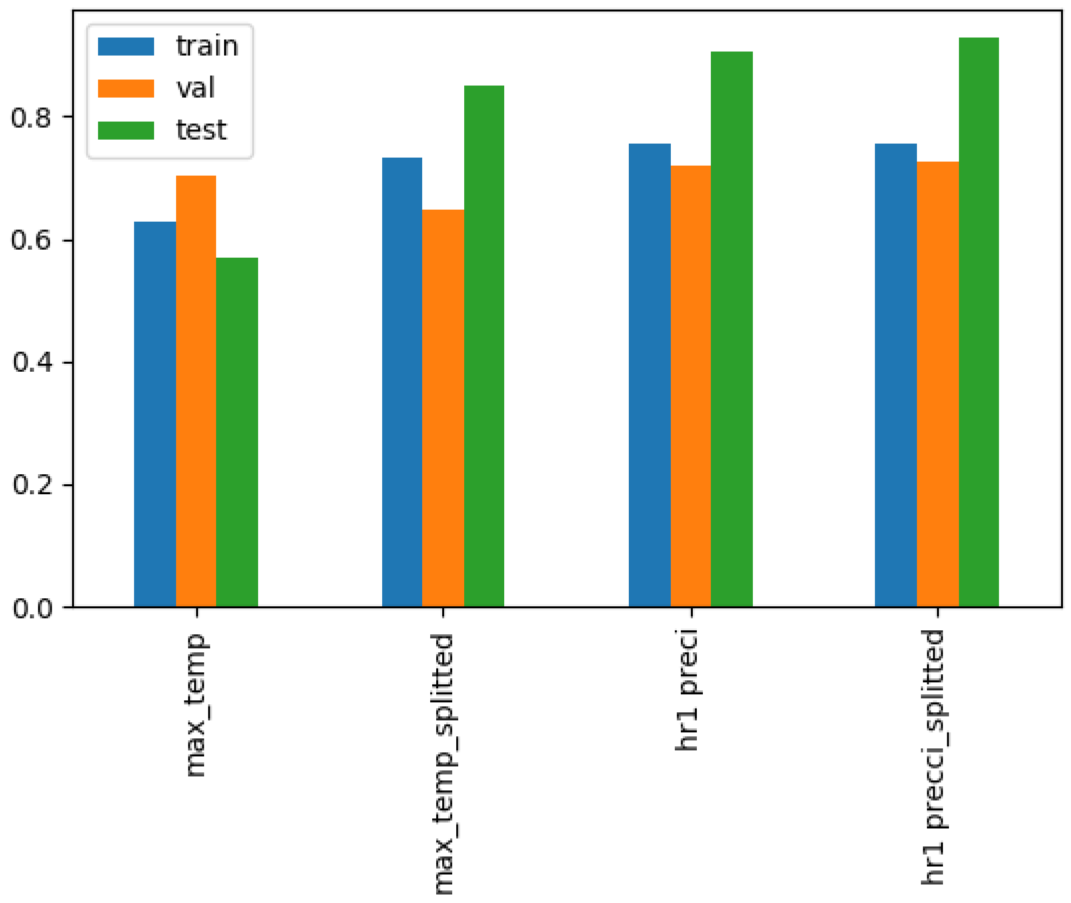
Out[16]:

	days
days	1.000000
avg temp	0.829122
max temp	0.790536
min temp	0.786602
hr1 preci	0.469541
min10 preci	0.442183
total preci	0.148526
local pressure	0.148449
sun hours	0.082137
sea pressure	0.075733
min humid	0.075391
avg humid	0.065836

Highly correlated feauture : (max temp, hr1 preci) - Analysing

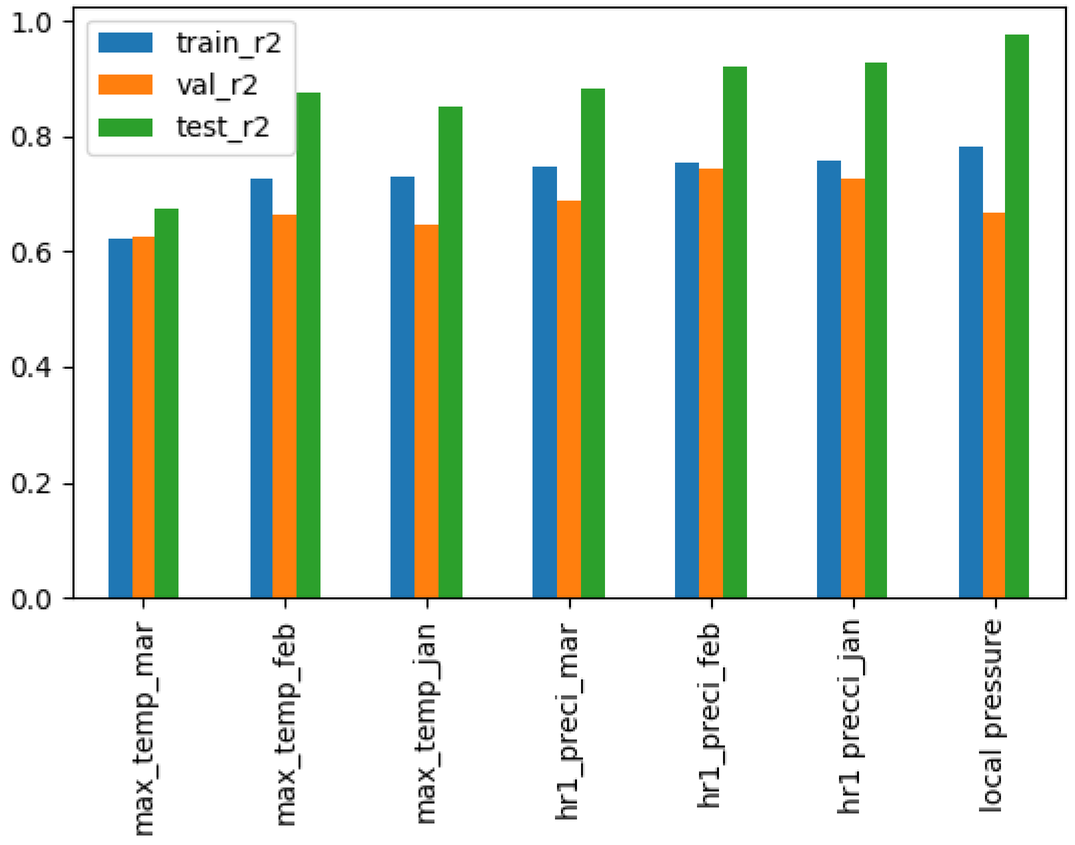
Spliting by month

```
    Temperarture mostly controlls the the bloom day as daily temperature hel
ps the tree to gain the DTS for the reaction needed for bloom to occur. So t
emperature is considered sepately. Again precipitaion is the second most imp
ortant feature found from the correlation chart.

     As the first quarter is a  season transition time differen months tempe
rature can  effect differently effect the bloom date ,So taking mean of  lik
e january , february , march in previous section can nullify their diversit
y. i.e as a part of winter - february is typically cooler than march which i
s part of sprig.

 Approaches:

 i. Taking mean of Dj-Dj+60, here one sample for each feature can be found i
n a year.
 ii. Spliting each feaure by its mean value of each of the months of the yea
r and will result 12 sample per year for each feature.
 iii. Spliting each feature on 15 day basis resulting 24 features in a year.

 Results:

     i.First method resulted moderate.
     ii. Monthly grouping one resulted best.(Taking the first three months o
nly)
     iii. 15 days groups resulted poor.

Intution behind monthly grouing :

1. In the previous section all the feaures were as the mean of Dj-Dj+60 day
s. That resulted satisfactorily.
2. Now Grouping each feature by months and by taking mean of each months dat
a can produce 12 derived features from a single feature.
3. Weather before the bloom day can only effect bloom, so weather data after
 april is ignored. As typically bloom occurs wihin march or 1st week of apri
l.
4. Grouping by 15 days of a month like - jan(1-15)'mean,jan(16-31)'mean was
 resulted poor.
    Bellow R2 score before and monthly spitting is shown.
    Where a simple nn with one layer lr-0.1 and epoc -500 used for quick tes
ting.
```


train
val
test
0.0
0.2
0.4
0.6
0.8
max_temp
max_temp_splitted
hr1 preci
hr1 precci_splitted

Each Added feature vs R2 Score improvements :

```
So finally each months mean temperature is considered as separate feature, a
nd january, february and march have shown high correlation while april and l
ater months was showing low correlation and adding them also degraded the pe
rformance of nn.

Below is the Plot of change of R2 Score after adding each additional feature
s from left to right ---
1. 'max temp march'
2. 'max temp march' , 'max temp feb'
3. 'max temp march' , 'max temp feb' , 'max temp jan'
4. 'max temp march' , 'max temp feb' , 'max temp jan', 'hr1 preci march'
5. ----
6. ---
7. ---
```



The df with corrsponing r2 score values can give a better understanding of precise changes ---

	train_r2	val_r2	test_r2
max_temp_mar	0.623061	0.626160	0.672611
max_temp_feb	0.727311	0.665169	0.877237
max_temp_jan	0.731410	0.646815	0.850886
hr1_preci_mar	0.748869	0.689130	0.882846
hr1_preci_feb	0.755149	0.744986	0.922191
hr1_preci_jan	0.756080	0.726036	0.928514

```
Function : split monthly
-input : yearly grouped dataframes
-ouput : yearly data frames with splited features added as new columns.
```

In [17]:
```
def split_monthly(dfs,yrs):
    for year in yrs[:]:
        Dj=Dj_df.loc[year,'Dj']-1
        _df=dfs[year].copy()
        mn_gp=_df.groupby('month')
        jan=mn_gp.get_group(1)
        feb=mn_gp.get_group(2)
        mrch=mn_gp.get_group(3)
        _df['max_temp_1']=jan['avg temp'].mean()
        _df['max_temp_2']=feb['max temp'].mean()#feb['max temp'].mean
()
        _df['max_temp_3']=mrch['max temp'].mean()

        _df['hr1_preci_1']=jan['hr1 preci'].mean()
        _df['hr1_preci_2']=feb['hr1 preci'].mean()#feb['max temp'].me
an()
        _df['hr1_preci_3']=mrch['hr1 preci'].mean()
        dfs[year]=_df
    return dfs
yearly_dfs=split_monthly(yearly_dfs,years)
```

Train , Validation , Test spliting and Scaling the data:

```
Function : get_splitted_years
-input : years, split
-output : train_years, val_years
```

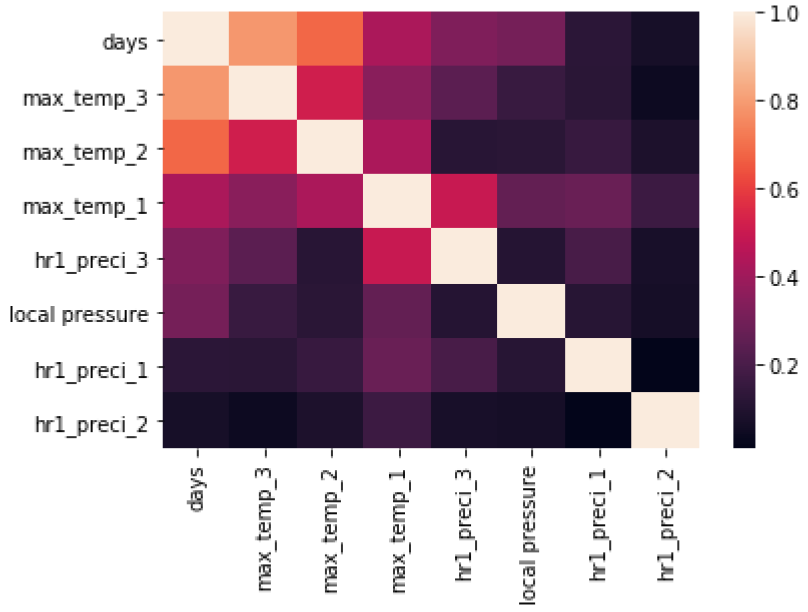
In [18]:
```
def get_splitted_years(yrs,splt):
    tr_years=[]
    val_years=[]
    for year in yrs:
        if year%splt==0:
            val_years.append(year)
        else :
            tr_years.append(year)
    tr_years=tr_years[:-1]
    return tr_years,val_years

split=5
tr_years,val_years = get_splitted_years(train_years,split)
corr_feats=['max_temp_3','max_temp_2','max_temp_1','hr1_preci_3','hr1
_preci_2','hr1_preci_1','local pressure']#,'sun hours','year','avg hu
mid']#,'year']

allX,allY,all_df=get_vectors(yearly_dfs,corr_feats,years)
trainX,trainY,train_df=get_vectors(yearly_dfs,corr_feats,tr_years)
testX,testY,test_df=get_vectors(yearly_dfs,corr_feats,test_years)
valX,valY,val_df=get_vectors(yearly_dfs,corr_feats,val_years)

print('Train shapes     :',trainX.shape,trainY.shape)
print('Validation shape :',valX.shape,valY.shape)
print('Test shape       :',testX.shape,testY.shape)
```

```
Train shapes     : (41, 7) (41, 1)
Validation shape : (10, 7) (10, 1)
Test shape       : (5, 7) (5, 1)
```

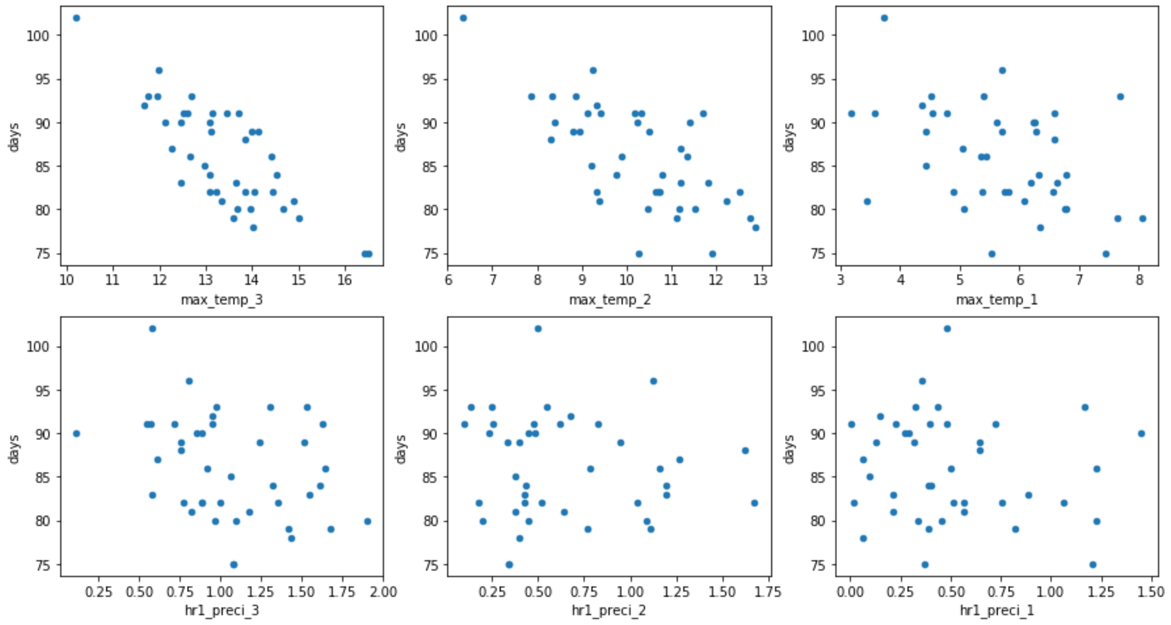
In [19]:
```
cm,_=plot_corr_sorted(train_df)
cm
```

Out[19]:

	days
days	1.000000
max_temp_3	0.790167
max_temp_2	0.686040
max_temp_1	0.429634
hr1_preci_3	0.332387
local pressure	0.309691
hr1_preci_1	0.124977
hr1_preci_2	0.072343



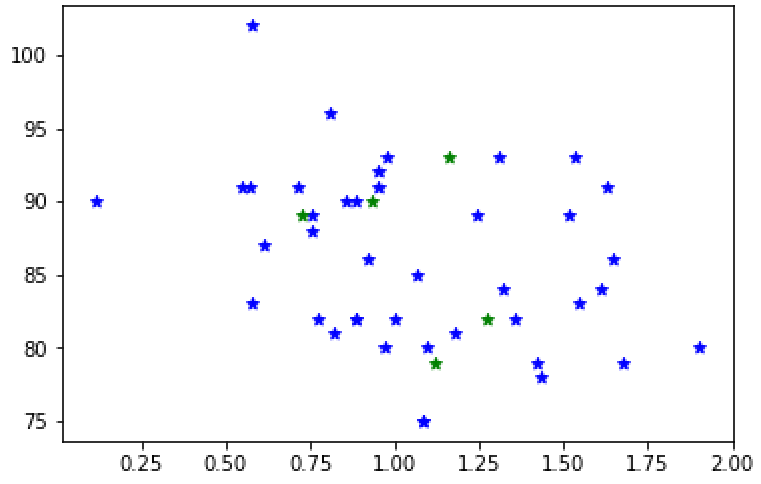
In [20]:
```
n=int(len(corr_feats)/2)
fig, axes = plt.subplots(nrows=2, ncols=n,figsize=(15,8))
for i in range(0,n):
    train_df.plot.scatter(x=corr_feats[i],y='days',ax=axes[0,i])#;axes[0,i].set_title(feats[i]);

for i in range(0,n):
    train_df.plot.scatter(x=corr_feats[i+n],y='days',ax=axes[1,i])#;axes[1,i].set_title(feats[i+3]);

plt.show()
```



Handling Outliers :

Significant outliers is not found till , still this section kept further analysis.

In [21]:
```
feat='hr1_preci_3'
#feat= 'max_temp_3'
outliers2=(train_df[feat]<0.75) & (train_df['days']<78)
plt.scatter(train_df.loc[:,feat],train_df.loc[:,'days'],marker='*',co
lor='b')
plt.scatter(test_df.loc[:,feat],test_df.loc[:,'days'],marker='*',colo
r='g')
plt.scatter(train_df.loc[outliers2,feat],train_df.loc[outliers2,'day
s'],marker='*',color='r')
plt.show()
```



In [22]:
```
ot_ind=np.array(np.where(outliers2))
#trainX=np.delete(trainX,ot_ind,axis=0)
#trainY=np.delete(trainY,ot_ind,axis=0)
ot_ind
```

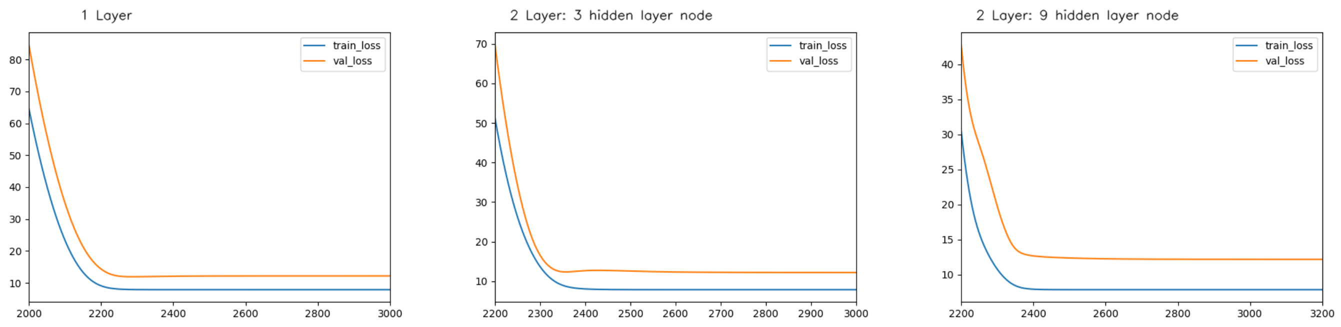
Out[22]:
```
array([], shape=(1, 0), dtype=int64)
```

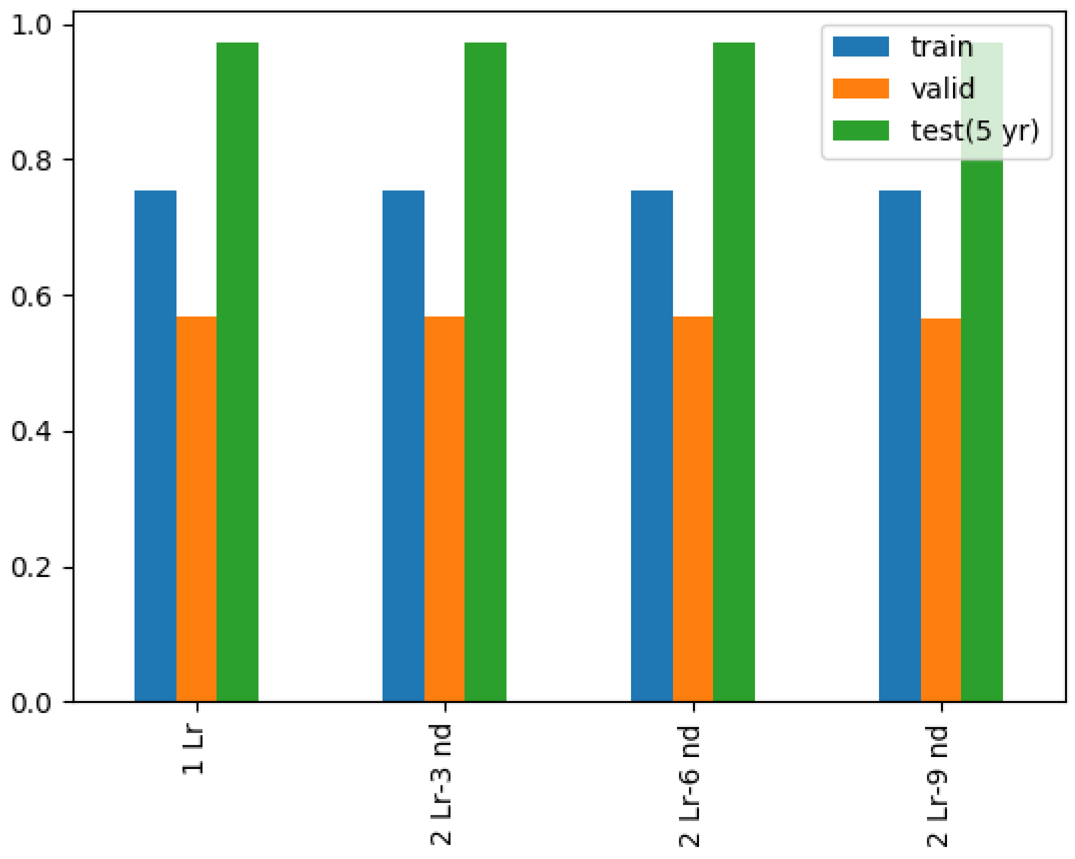
In [23]:
```
scalar= RobustScaler()
#scalar=StandardScaler(with_std=False)
#scalar= MinMaxScaler()

scl=scalar.fit(trainX)
scl_trainX=scl.transform(trainX)
#scl=scalar.fit(valX)
scl_valX=scl.transform(valX)
#scl=scalar.fit(testX)
scl_testX=scl.transform(testX)
#lm
```

Hiperparameter tuning

Layers : 1 Layer

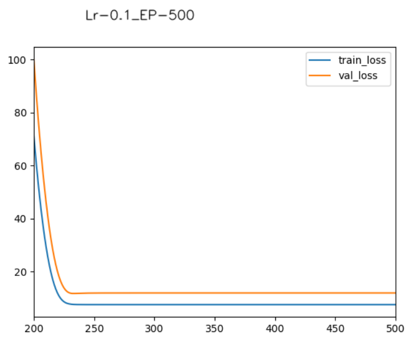
```
1. 1 Layer was performing good
2. 2 Layer was resulting little less (with 3,6,9 hidden nodes) r2 score and
 was taking more time to train.
So simple model with one layer was preferred.
```

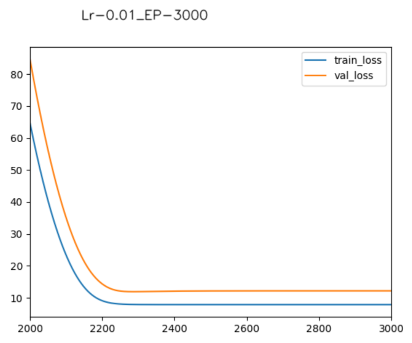


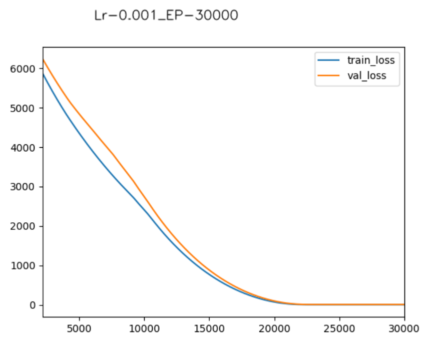


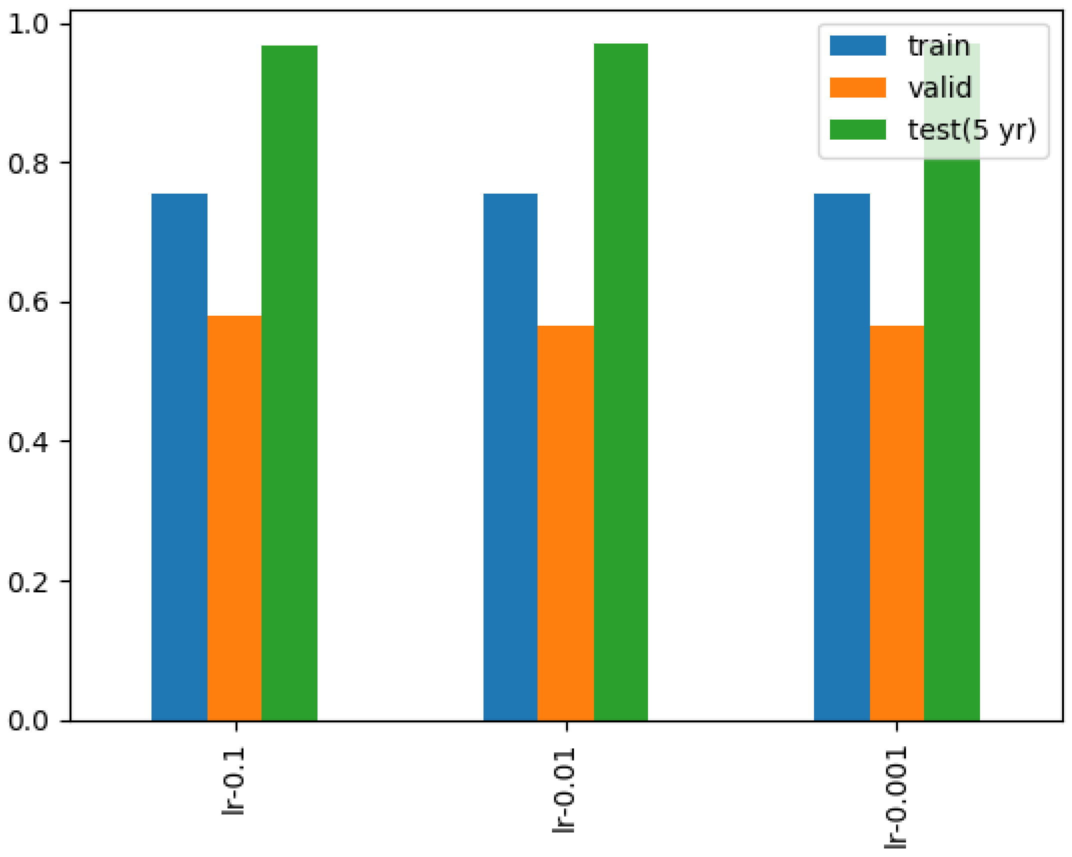
Learning Rate : 0.01

1. lr-0.1 was learning very fast with a little less r2_score than 0.01
2. lr-0.01 was perforing good with maximum r2 score
3. lr-0.001 was very very slow needed 30000 epocs to train.



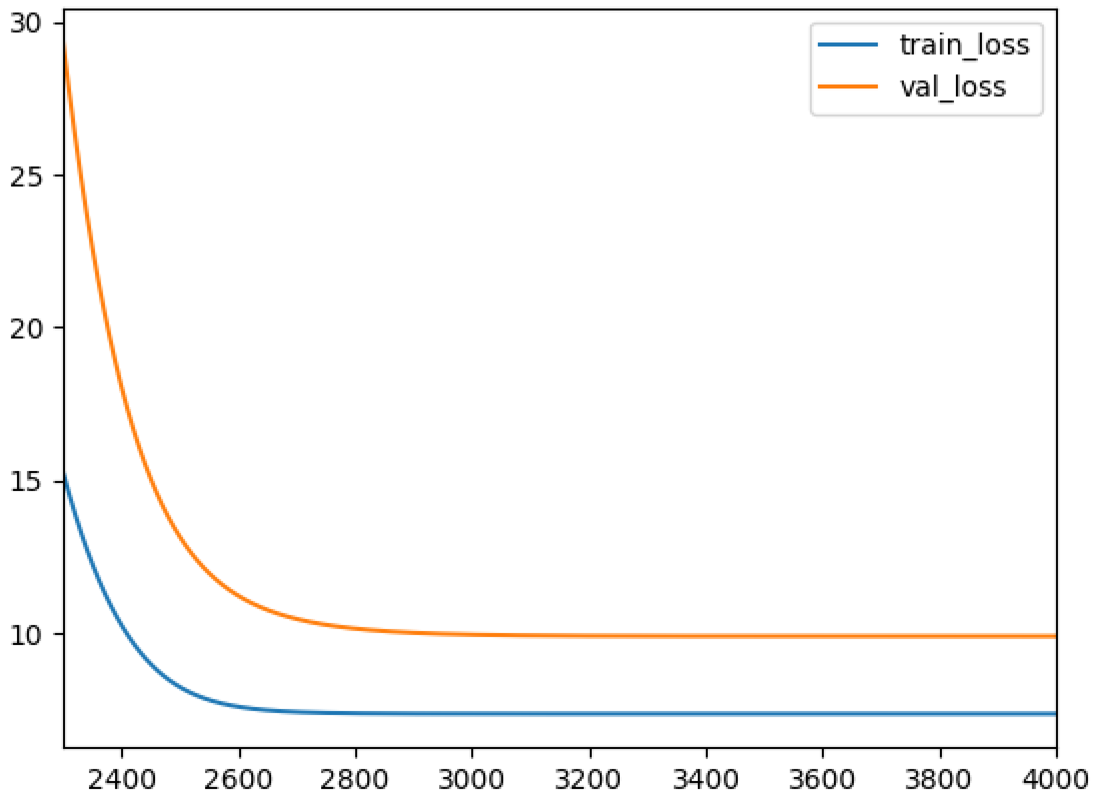






1. Afer 2400 epocs validation error was increasing. 2500 epoc was optimum.

Epoc : 3200



Batch size : 10

```
1. Bach size 10 (4 batches in  40 train data) was prforming best. lower or h
igher batch size was giveng higher error.
```

Optimizer :

```
1. SGD was giving more error.
2. Adadelta was very long to train.
1. Adam was taking about double epocs.
2. RMSprop was learning fast and error was minimum. (Best one)
```

ANN Class :

```
1. Initializing weights and biases : Get variable is preffered as it is used
 by tf.layers.Dense
Class according to official doccumentation and Get_variable uses 'glorot uni
form initializer' for
weights which is used by Keras also and there is a paper on this initialize
r.
```

Methods :

```
2. model_predict
- input : input features
- output : prediction
3. reagularized_loss_calc
- input : weights,biases,loss,beta
- output : l2 loss
4. loss_func
- input : predictions
- ouput : mean square loss
5. opt_func
- input : loss , trainable variables
- output : Network with updated weights
6. get_weights
- input : None
- output : weights, biases
7. set_weights
- input : weights, biases
- output : Network with updated weights, biases
```

In [24]:
```
layer=1
if layer==1:
    n_hidden_1 = 1
n_input = trainX.shape[1]
n_classes = 1#_train_y.shape[1]
if layer==2:
    n_hidden=9

class ANN():
    def __init__(self):
            init_zero=tf.zeros_initializer()
            init_glorot=tf.glorot_uniform_initializer()
            init_trunc_normal=tf.truncated_normal_initializer(stddev=
1e-3)
            init_rand_norm=tf.random_normal_initializer(stddev=1e-3)
            with tf.variable_scope("weights", reuse=tf.AUTO_REUSE):
                    self.w1= tf.get_variable('w1',[n_input, n_hidden_
1],dtype=tf.float64,initializer=init_glorot)
                    self.w2= tf.get_variable('w2',[n_hidden_1, n_clas
ses],dtype=tf.float64,initializer=init_glorot)
            with tf.variable_scope("biases", reuse=tf.AUTO_REUSE):
                    self.b1= tf.get_variable('b1',[n_hidden_1],dtype=
tf.float64,initializer=init_zero)
                    self.b2= tf.get_variable('b2',[n_classes],dtype=t
f.float64,initializer=init_zero)
            self.weights=[self.w1,self.w2]
            self.biases = [self.b1,self.b2]
    def model_predict(self,x):
        out = tf.add(tf.matmul(x, self.weights[0]),self.biases[0])
        out = tf.nn.relu(out)
        if layer==2:
            out = tf.add(tf.matmul(out,self.weights[1]),self.biases[1
])
            out = tf.nn.relu(out)
        return out

    def loss_func(self,y,out):
        _loss = tf.losses.mean_squared_error(labels=y, predictions=ou
t)
        loss=tf.cast(_loss,tf.float64)
        return loss
    def reg_loss_func(self,loss,beta):
        beta=np.float64(beta)

        reg_w1=0.5*tf.reduce_sum(tf.square(self.weights[0]))
        reg_w2=0.5*tf.reduce_sum(tf.square(self.weights[1]))
        if layer==1:
            _reg=reg_w1
        if layer==2:
            _reg=tf.add(reg_w1,reg_w2)

        reg=tf.cast(_reg,tf.float64)
        reg=tf.multiply(beta,reg)
        reg_loss=0.5*tf.add(loss,reg)

        return reg_loss
    def opt_func(self,lr,loss):
        if layer==1:
            trainable_vars=[self.w1,self.b1]
        if layer==2:
            trainable_vars=[self.w1,self.w2,self.b1,self.b2]
        #tf.GraphKeys.TRAINABLE_VARIABLE
        optimizer = tf.train.RMSPropOptimizer(lr)
        grads_and_vars=optimizer.compute_gradients(loss,trainable_var
s)
        train_op=optimizer.apply_gradients(grads_and_vars)
        #train_op = optimizer.minimize(loss)
        return train_op

    def get_weights(self):
        return self.weights,self.biases

    def set_weights(self,wts,bias):
        self.weights[0]=tf.convert_to_tensor(wts[0])
        self.biases[0]=tf.convert_to_tensor(bias[0])
        self.weights[1]=tf.convert_to_tensor(wts[1])
        self.biases[1]=tf.convert_to_tensor(bias[1])
```

Optimized Parameters :

```
- Layers - 1 Layer
- Learning Rate : 0.01
- Epocs : 2500
- Batch Size : 10 (number of batch 4 for total 41)
- Optimizer : RMSprop
```

In [25]:

```
epoc=4000
#epoc=1000
lr=0.01
beta=5
no_of_batch=4 #(For Batch size of about 14-15)
xx=np.array_split(scl_trainX,no_of_batch)
yy=np.array_split(trainY,no_of_batch)
xx_val=np.array_split(scl_valX,no_of_batch)
yy_val=np.array_split(valY,no_of_batch)
```

In [26]:
```
import tensorflow as tf
tf.reset_default_graph()
x = tf.placeholder(tf.float64, shape=[None, trainX.shape[1]])
y=tf.placeholder(tf.float64, shape=[None, 1])

M=ANN()
out=M.model_predict(x)
loss=M.loss_func(y,out)

reg_loss=M.reg_loss_func(loss,beta)
train_op=M.opt_func(lr,loss)
```

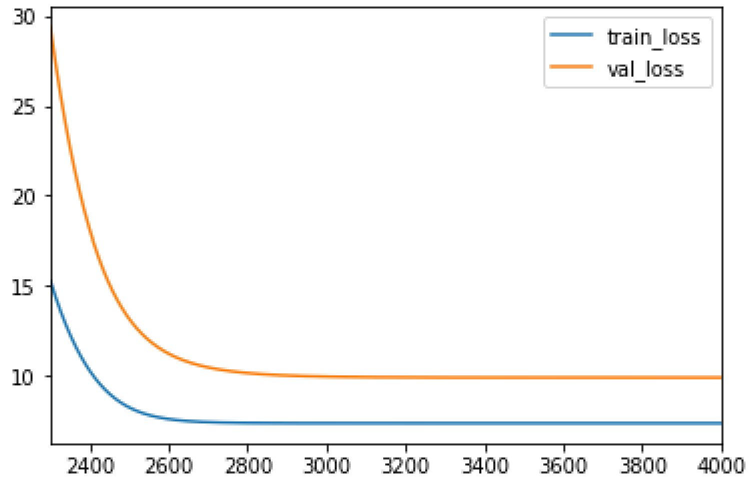
In [27]:
```
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np

#%matplotlib notebook
from IPython.display import clear_output


sess = tf.Session()
init = tf.global_variables_initializer()
sess.run(init)

t_loss=[]
v_loss=[]

#fig = plt.figure()
#ax = fig.add_subplot(111)
#plt.ion()
#fig.show()

for i in range(epoc+1):
    train_loss=0

    for j in range(0,len(xx)):
        inputs_train={x:xx[j],y:yy[j]}
        _= sess.run((train_op),feed_dict=inputs_train)
        loss_train = sess.run((loss),feed_dict=inputs_train)
        train_loss+=loss_train
    train_loss/=(j+1)

    val_loss=0
    for k in range(0,len(xx_val)):
        inputs_valid={x:xx_val[k],y:yy_val[k]}
        loss_valid = sess.run((loss),feed_dict=inputs_valid)
        val_loss+=loss_valid

    val_loss/=(k+1)
    t_loss.append(train_loss)
    v_loss.append(val_loss)

    disp_log=50
    if i>disp_log and i%disp_log==0:
        #clear_output()
        mt_loss=np.mean(np.array(t_loss[i-disp_log:i]))
        mv_loss=np.mean(np.array(v_loss[i-disp_log:i]))
        print('epoc:',i-disp_log,'-',i,'mean_train_loss:',mt_loss,'me
an_valid_loss:',mv_loss)
    if i>30000:
        ax.clear()
        ax.plot(t_loss[-300:])
        ax.plot(v_loss[-300:])
        fig.canvas.draw()
```

```
epoc: 50 - 100 mean_train_loss: 6624.976342773438 mean_valid_loss: 67
84.649230957031
epoc: 100 - 150 mean_train_loss: 6155.727155761719 mean_valid_loss: 6
286.420070800781
epoc: 150 - 200 mean_train_loss: 5711.0740234375 mean_valid_loss: 583
6.379699707031
epoc: 200 - 250 mean_train_loss: 5292.469538574219 mean_valid_loss: 5
410.975494384766
epoc: 250 - 300 mean_train_loss: 4899.732836914062 mean_valid_loss: 5
009.535589599609
epoc: 300 - 350 mean_train_loss: 4532.447163085937 mean_valid_loss: 4
632.14580078125
epoc: 350 - 400 mean_train_loss: 4190.114282226563 mean_valid_loss: 4
278.923914794922
epoc: 400 - 450 mean_train_loss: 3872.121541748047 mean_valid_loss: 3
950.011243286133
epoc: 450 - 500 mean_train_loss: 3577.699445800781 mean_valid_loss: 3
645.5384045410156
epoc: 500 - 550 mean_train_loss: 3305.878331298828 mean_valid_loss: 3
365.5442700195313
epoc: 550 - 600 mean_train_loss: 3055.4625787353516 mean_valid_loss:
3109.8281549072267
epoc: 600 - 650 mean_train_loss: 2825.05662109375 mean_valid_loss: 28
77.75409576416
epoc: 650 - 700 mean_train_loss: 2612.3249176025392 mean_valid_loss:
2670.824100036621
epoc: 700 - 750 mean_train_loss: 2406.52201171875 mean_valid_loss: 25
03.406610717773
epoc: 750 - 800 mean_train_loss: 2211.4853875732424 mean_valid_loss:
2354.802761383057
epoc: 800 - 850 mean_train_loss: 2029.341538696289 mean_valid_loss: 2
216.2479915618896
epoc: 850 - 900 mean_train_loss: 1859.3235870361327 mean_valid_loss:
2086.4224911499023
epoc: 900 - 950 mean_train_loss: 1700.6421487426758 mean_valid_loss:
1964.0467393112183
epoc: 950 - 1000 mean_train_loss: 1552.5177645874023 mean_valid_loss:
1847.930458164215
epoc: 1000 - 1050 mean_train_loss: 1414.2136352539062 mean_valid_los
s: 1737.016907081604
epoc: 1050 - 1100 mean_train_loss: 1285.0600860595703 mean_valid_los
s: 1630.4098725128174
epoc: 1100 - 1150 mean_train_loss: 1164.4656735229491 mean_valid_los
s: 1527.3788703083992
epoc: 1150 - 1200 mean_train_loss: 1051.9158044433593 mean_valid_los
s: 1427.3475958156587
epoc: 1200 - 1250 mean_train_loss: 946.9631591796875 mean_valid_loss:
1329.874296090603
epoc: 1250 - 1300 mean_train_loss: 849.2153164672851 mean_valid_loss:
1234.6332439208031
epoc: 1300 - 1350 mean_train_loss: 758.3232490539551 mean_valid_loss:
1141.4025958156585
epoc: 1350 - 1400 mean_train_loss: 673.9726467895508 mean_valid_loss:
1050.0597172164917
epoc: 1400 - 1450 mean_train_loss: 595.8781170654297 mean_valid_loss:
960.5828663444519
epoc: 1450 - 1500 mean_train_loss: 523.7797373199463 mean_valid_loss:
```

```
873.0558930206299
epoc: 1500 - 1550 mean_train_loss: 457.4404359436035 mean_valid_loss:
787.6716730880737
epoc: 1550 - 1600 mean_train_loss: 396.6432781982422 mean_valid_loss:
704.7299214363098
epoc: 1600 - 1650 mean_train_loss: 341.1874767303467 mean_valid_loss:
624.6261603164672
epoc: 1650 - 1700 mean_train_loss: 290.88313041687013 mean_valid_los
s: 547.8297885131836
epoc: 1700 - 1750 mean_train_loss: 245.54526752471924 mean_valid_los
s: 474.8525311279297
epoc: 1750 - 1800 mean_train_loss: 204.98867685317992 mean_valid_los
s: 406.21165464401247
epoc: 1800 - 1850 mean_train_loss: 169.02525102615357 mean_valid_los
s: 342.3981039047241
epoc: 1850 - 1900 mean_train_loss: 137.46508615493775 mean_valid_los
s: 283.8649846458435
epoc: 1900 - 1950 mean_train_loss: 110.12010453224183 mean_valid_los
s: 231.04243696212768
epoc: 1950 - 2000 mean_train_loss: 86.8057917690277 mean_valid_loss:
184.336942653656
epoc: 2000 - 2050 mean_train_loss: 67.33534799575806 mean_valid_loss:
144.04985016822815
epoc: 2050 - 2100 mean_train_loss: 51.50012740135193 mean_valid_loss:
110.2803867149353
epoc: 2100 - 2150 mean_train_loss: 39.02607018232346 mean_valid_loss:
82.91713160514831
epoc: 2150 - 2200 mean_train_loss: 29.519877588748933 mean_valid_los
s: 61.599146146774295
epoc: 2200 - 2250 mean_train_loss: 22.473025646209717 mean_valid_los
s: 45.622752075195315
epoc: 2250 - 2300 mean_train_loss: 17.356680989265442 mean_valid_los
s: 34.03186012268066
epoc: 2300 - 2350 mean_train_loss: 13.723579258918763 mean_valid_los
s: 25.861301996707915
epoc: 2350 - 2400 mean_train_loss: 11.229820964336396 mean_valid_los
s: 20.256608529090883
epoc: 2400 - 2450 mean_train_loss: 9.59866268157959 mean_valid_loss:
16.497466063499452
epoc: 2450 - 2500 mean_train_loss: 8.594365162849426 mean_valid_loss:
14.064518365561963
epoc: 2500 - 2550 mean_train_loss: 8.015170640945435 mean_valid_loss:
12.535788538008928
epoc: 2550 - 2600 mean_train_loss: 7.6999626326560975 mean_valid_los
s: 11.583163592219353
epoc: 2600 - 2650 mean_train_loss: 7.535600430965424 mean_valid_loss:
10.987857378423215
epoc: 2650 - 2700 mean_train_loss: 7.452103847265244 mean_valid_loss:
10.61220949947834
epoc: 2700 - 2750 mean_train_loss: 7.410175856351852 mean_valid_loss:
10.371983504295349
epoc: 2750 - 2800 mean_train_loss: 7.389116080999375 mean_valid_loss:
10.216232381165028
epoc: 2800 - 2850 mean_train_loss: 7.37842858672142 mean_valid_loss:
10.114005264639854
epoc: 2850 - 2900 mean_train_loss: 7.3729024398326874 mean_valid_los
s: 10.046235791742802
```

```
epoc: 2900 - 2950 mean_train_loss: 7.3699697756767275 mean_valid_los
s: 10.000958324968815
epoc: 2950 - 3000 mean_train_loss: 7.36836304306984 mean_valid_loss:
9.970533206164838
epoc: 3000 - 3050 mean_train_loss: 7.367451266050339 mean_valid_loss:
9.95000584244728
epoc: 3050 - 3100 mean_train_loss: 7.366916056871414 mean_valid_loss:
9.936112913787365
epoc: 3100 - 3150 mean_train_loss: 7.366591620445251 mean_valid_loss:
9.926690164804459
epoc: 3150 - 3200 mean_train_loss: 7.366389402151108 mean_valid_loss:
9.92029109954834
epoc: 3200 - 3250 mean_train_loss: 7.366259440183639 mean_valid_loss:
9.915939694344997
epoc: 3250 - 3300 mean_train_loss: 7.366174751520157 mean_valid_loss:
9.912979259192944
epoc: 3300 - 3350 mean_train_loss: 7.366119432449341 mean_valid_loss:
9.910965328216554
epoc: 3350 - 3400 mean_train_loss: 7.366082954406738 mean_valid_loss:
9.90959229171276
epoc: 3400 - 3450 mean_train_loss: 7.3660581684112545 mean_valid_los
s: 9.908657480478286
epoc: 3450 - 3500 mean_train_loss: 7.3660419964790345 mean_valid_los
s: 9.908020487427711
epoc: 3500 - 3550 mean_train_loss: 7.366030000448227 mean_valid_loss:
9.907588074803352
epoc: 3550 - 3600 mean_train_loss: 7.36602259516716 mean_valid_loss:
9.907292118668556
epoc: 3600 - 3650 mean_train_loss: 7.366017374992371 mean_valid_loss:
9.90709162503481
epoc: 3650 - 3700 mean_train_loss: 7.3660130536556245 mean_valid_los
s: 9.906953382492066
epoc: 3700 - 3750 mean_train_loss: 7.366011266708374 mean_valid_loss:
9.906860089302063
epoc: 3750 - 3800 mean_train_loss: 7.366010125875473 mean_valid_loss:
9.906796767115592
epoc: 3800 - 3850 mean_train_loss: 7.366008222103119 mean_valid_loss:
9.906754268109799
epoc: 3850 - 3900 mean_train_loss: 7.366007845401764 mean_valid_loss:
9.90672228038311
epoc: 3900 - 3950 mean_train_loss: 7.36600608587265 mean_valid_loss:
9.906704975664615
epoc: 3950 - 4000 mean_train_loss: 7.3660067474842075 mean_valid_los
s: 9.906694089472294
```

In [44]:
```
%matplotlib inline
loss_df=pd.DataFrame(np.array([t_loss,v_loss]).T,columns=['train_los
s','val_loss'])
loss_df.loc[2300:].plot()
plt.show()
```



In [38]:
```
def predict_runtime(test_data):
    inputs_test={x:np.array(test_data)}
    res = sess.run((out),feed_dict=inputs_test)
    res = res.reshape(-1)
    return res
pred_trainY=predict_runtime(scl_trainX)
pred_valY=predict_runtime(scl_valX)
pred_testY=predict_runtime(scl_testX)
```

In [39]:
```
tr_err=mean_squared_error(trainY,pred_trainY)
val_er=mean_squared_error(valY,pred_valY)
test_er=mean_squared_error(testY,pred_testY)
tr_r2=r2_score(trainY,pred_trainY)
val_r2=r2_score(valY,pred_valY)
test_r2=r2_score(testY,pred_testY)

print('Mean squared error :',tr_err,val_er,test_er)
print('R2 Score :',tr_r2,val_r2,test_r2)
```

```
Mean squared error : 7.433637045688144 9.031633070798986 0.6418759605
467601
R2 Score : 0.7842998019436276 0.6819847510282048 0.9766080189305116
```

Results Neural Network :

```
-----------  R2 Score   -----------
train score      : 0.7832225685989149,
validation score : 0.6670342151375969,
*test score      : 0.977666516001531
```

```
In [40]: import pickle
         def save_weights(model):
             wts,bias=model.get_weights()
             wt,bs=sess.run((wts,bias))

             f=open("trained_params_final","wb")
             pickle.dump([wt,bs],f)
             f.close()
         def load_weights():
             f=open("trained_params_final","rb")
             [wt,bs]=pickle.load(f)
             f.close()
             return wt,bs
         def build_and_predict(test_data,wt,bs):
             tf.reset_default_graph()

             N=ANN()
             N.set_weights(wt,bs)
             tx = tf.placeholder(tf.float64, shape=[None, trainX.shape[1]])
             t_out=N.model_predict(tx)

             t_sess = tf.Session()
             init = tf.global_variables_initializer()
             t_sess.run(init)

             inputs_test={tx:np.array(test_data)}
             res = t_sess.run((t_out),feed_dict=inputs_test)
             res = res.reshape(-1)
             return res
```

```
In [41]: #save_weights(model=M)
         wt,bs=load_weights()
         res=build_and_predict(scl_testX,wt,bs)
         r2_score(testY,res)

Out[41]: 0.9776665233403259
```

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.

In [42]:
```
Days=get_true_label(yearly_dfs,test_years)
true_days=Days['true_days']
pred_600=np.array(predict(yearly_dfs,test_years,float(thr_temp)))
pred_DTS=np.array(predict_ts(yearly_dfs,30,test_years,DTS_df.mean()))
pred_NN=np.array(build_and_predict(scl_testX,wt,bs))


_600_err=r2_score(true_days,pred_600)
DTS_err=r2_score(true_days,pred_DTS)
NN_err=r2_score(true_days,pred_NN)
errs=pd.DataFrame({'600_score':_600_err,'DTS_score':DTS_err,'NN_scor
e':NN_err},index=['r2_score'])
Days['600_pred']=pred_600
Days['DTS_pred']=pred_DTS
Days['NN_pred']=pred_NN
Days.plot()
Days.plot.bar()
errs.plot.bar()
```

Out[42]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f785245c198>
```

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 [43]:
```
Dj_df=get_Djs(yearly_dfs,years)
Bdj=get_true_label(yearly_dfs,years)
comp_df=Dj_df
comp_df['Bdj']=Bdj['true_days']
comp_df['Diff']=(comp_df['Bdj']-comp_df['Dj'])
#comp_df['diff']=comp_df['Bdj']-comp_df['Dj']
fig, axes = plt.subplots(nrows=2, ncols=2,figsize=(12,8))
comp_df['Dj'].plot(ax=axes[0,0]);axes[0,0].set_title('Dj');
comp_df['Bdj'].plot(ax=axes[0,1]);axes[0,1].set_title('Bdj');
comp_df['Diff'].plot(ax=axes[1,0]);axes[1,0].set_title('Interval');
_,_,tdf=get_vectors(yearly_dfs,corr_feats,years[:-1],scl=0);axes[1,1]
.set_title('Temp february, march');
tdf[['max_temp_2','max_temp_3']].plot(ax=axes[1,1])
plt.show()
```

Observations:

```
During these 60 years :
- Dj is increasing
- Bdj is dereasing.
- Dj-Bdj interval is decreasing.
- Temperature is increasing.

1. - 1961 Hibrnation used to end at 38-40 th day >> February 7-9
   - 2016 Hibrnation used to end at 45-48 th day >> February 14-17
   >>about 1 week later.

2. - 1961 First Bloom used to be at 91-93 th day >> April 1-3
   - 2016 First Bloom occurs at 80-82 th day  >> March 21-23
   >>about 10 days earlier.

3. - 1961 Hibernation to bloom intterval was about 50-53 days
   - 2016 Hibernation to bloom intterval was about 33-36 days
   >>about 17 days less is required for reaching bloom from hibernation
   to occur.
4. Reason is The temperature rise in february and march ,this is also
the reason for though Dj is 1 week delayed but bloom occurs 10 days
earlier in now compared to 1961.

5. In the year 1984 there was a pick delay  to bloom day and was 102 th day
 >> April 12
    Reason was sudden fall in february and march temperature.
6. In 2002 and 2013 there were sudden falls in Bdj cause Bloom day was 75 th
 day >> march 16
    Reason was March's high temperature on those years.
```

```
In [ ]:
```