29-Apr-21 11:31 PM
INM Data SVR from sklearn Point 1 vars: huss, tas, psl, uas, vas and we are predicting from imd data.
Code:

```
# In this one I will be performing statistical analysis and linear regression using statsmodels api on INM-CM4-8 point 1 data.
# %%
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import statsmodels.api as sm
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline
from sklearn.svm import SVR
# X is INM data
# y is IMD data
x_path = r'D:\Mimisbrunnr\Data\Civil_Data\INMCM4-8_historical_predictors_Point_1.csv'
y_path = r'D:\Mimisbrunnr\Data\Civil_Data\IMD_Rainfall_0.25x0.25\rain_Point_1.csv'
X = pd.read_csv(x_path)
y = pd.read_csv(y_path)
# dropping longitude and latitude since they are fixed
X.drop(columns=['lon', 'lat', 'nos'], inplace=True)
y.drop(columns=['LONGITUDE', 'LATITUDE'], inplace=True)
# removing duplicates
X.drop_duplicates(keep='first', inplace=True)
# removing everything beyond 2015-01-01 from IMD data
y.drop([i for i in range(23376, len(y))], inplace=True)
# removing everything before 1951-01-01 from INM data
X.drop([i for i in range(365)], inplace=True)
# IMD data has leap years, INM data does not. removing them
y.drop(y[y['TIME'].map(lambda s: s.split()[0][-5:]) == '02-29'].index, inplace = True)
# set indices so that they start from 0
X.reset_index(inplace=True, drop=True)
y.reset_index(inplace=True, drop=True)
# cannot have a string column, so split it into 3
X_dtCol = pd.to_datetime(X['time'], format='%Y-%m-%d %H:%M:%S')
X['time'] = X_dtCol
```

```
# convert datetime to day, month and year
# for X
X['day'] = X['time'].dt.day
X['month'] = X['time'].dt.month
X['year'] = X['time'].dt.year
X.drop(columns=['time'], inplace=True)
# for y - commented out since y must have just rainfall
# y_dtCol = pd.to_datetime(y['TIME'], format='%Y-%m-%d %H:%M:%S')
# y['TIME'] = y_dtCol
# y['day'] = y['TIME'].dt.day
# y['month'] = y['TIME'].dt.month
# y['year'] = y['TIME'].dt.year
y.drop(columns=['TIME'], inplace=True)
# print(X.head())
# print(y.head())
y['RAINFALL'].fillna(value=y['RAINFALL'].mean(), inplace=True)
# %%
# Standardising
# if we scale then Condition Number is good in statmodels
scaler = StandardScaler()
X = scaler.fit_transform(X[X.columns])
y = scaler.fit_transform(y)
# %%
# printing
"""
print('X.head()')
print(X.head())
print('y.head()')
print(y.head())
print('X.tail()')
print(X.tail())
print('y.tail()')
print(y.tail())
print()
print('X.columns')
print(X.columns)
"""
# %%
# generating test and train sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state=42)
# %%
# to compare and check whether X and y have the same set of dates
"""
```

```
with open('inm-cm4-8.txt', 'w') as f:
    for ind in range(len(X)):
        f.write(str(X['time'].iloc[ind]).split(' ')[0] + '\n')
with open('imd.txt', 'w') as f:
    for ind in range(len(y)):
        f.write(str(y['TIME'].iloc[ind]).split(' ')[0] + '\n')
"""
# %%
# Linear Regression
"""
model = LinearRegression()
model.fit(X_train, y_train)
predictions = model.predict(X_test)
predictions = np.where(predictions<=0, 0, predictions)
print(f'The R2 Score is: {r2_score(y_test, predictions)}')
"""
# %%
# Polynomial Regression with variable degree
# for degree = 2 -> 0.05325821822687249
# for degree = 3 -> 0.05630877341597906
# for degree = 4 -> 0.049656805504937784
# for degree = 5 -> 0.02727273524532825
# for degree = 6 -> 0.053098448690447775
# for degree = 7 -> 0.05035674414188229
# for degree = 8 -> 0.03664462816877734
"""
for degree in range(7, 9):
    model = make_pipeline(PolynomialFeatures(degree=degree),LinearRegression())
    model.fit(X_train, y_train)
    predictions = model.predict(X_test)
    predictions = np.where(predictions<=0, 0, predictions)
    print(f'The R2 Score is: {r2_score(y_test, predictions)}')
"""
# %%
# Perform Linear Regression using Ordinary Least Squares and generate a hell of a lot of statistics
# Link: https://medium.com/swlh/interpreting-linear-regression-through-statsmodels-summary-4796d359035a
"""
X = sm.add_constant(X)
model = sm.OLS(y, X)
res = model.fit()
print(res.summary())
"""
# %%
```

```
# generate the autocorrelation matrix of the rainfall time series data
# autocorrelation is the correlation of a given sequence with itself as a function of time lag

# %%
# SVR
# Hyperparameters:
# kernel = {‘linear’, ‘poly’, ‘rbf’, ‘sigmoid’}, default=’rbf’
# degree: int, default=3 (only for poly)
# gamma{‘scale’, ‘auto’} or float, default=’scale’
#     Kernel coefficient for ‘rbf’, ‘poly’ and ‘sigmoid’.
#     if gamma='scale' (default) is passed then it uses 1 / (n_features * X.var()) as value of gamma,
#     if ‘auto’, uses 1 / n_features.
# coef0: float, default=0.0 (only in poly, sigmoid)
# tol: float, default=1e-3
# C: float, default=1.0 (regularisation) always +ve
# epsilon: float, default=0.1
# max_iter: int, default=-1
"""
Results:
kernel      max_iter        degree      gamma       coef0       tol     C       epsilon     R2                          time-taken (s)
linear      150000          -           scale       0.0         1e-3    1.0     0.1         -7.297947472117983e-06
linear      150000          -           auto        0.0         1e-3    1.0     0.1         -7.297947472117983e-06
rbf         150000          -           scale       0.0         1e-3    1.0     0.1         -7.297947472117983e-06
linear      150000          -           auto        0.0         1e-3    0.1     0.1         -7.297947472117983e-06
linear      150000          -           auto        0.0         1e-3    10      0.1         -7.297947472117983e-06
poly        1000000         5           scale       0.0         1e-3    1.0     0.1         -1.839877979259441e-05
poly        1000000\        7           scale       0.0         1e-3    1.0     0.1         -0.0016505121393148858
rbf         -               -           scale       0.0         1e-3    0.001   0.1         -7.297947472117983e-06
rbf         -               -           scale       0.0         1e-3    0.01    0.1         -7.297947472117983e-06
rbf         -               -           scale       0.0         1e-3    0.1     0.1         -7.297947472117983e-06
rbf         -               -           scale       0.0         1e-3    1.0     0.1         -7.297947472117983e-06
rbf         -               -           scale       0.0         1e-3    10.0    0.1         5.346235345804473e-05
rbf         -               -           scale       0.0         1e-3    100     0.1         0.0023060365121116977
rbf         -               -           scale       0.0         1e-3    1000    0.1         0.013924728123650532
rbf         -               -           scale       0.0         1e-3    1000    10          -21.417639281476077         0.6166067123413086
rbf         -               -           scale       0.0         1e-3    1500    10          -16.368526014747765         0.8817434310913086
rbf         -               -           scale       0.0         1e-3    2000    10          -14.06508737925242          1.18355131149292
rbf         -               -           scale       0.0         1e-3    2500    10          -14.045774460318626         1.4827511310577393
rbf         -               -           scale       0.0         1e-3    3000    10          -14.50064409277358          1.785775899887085
rbf         -               -           scale       0.0         1e-3    3000    5           -5.039082037067492          1.8954386711120605
rbf         -               -           scale       0.0         1e-3    5000    5           -4.726379833329805          2.569713830947876
rbf         -               -           scale       0.0         1e-3    7000    5           -4.621419027024011          3.373181104660034
rbf         -               -           scale       0.0         1e-3    9000    5           -4.586154773157784          4.131232500076294
```

```
rbf         -               -           scale       0.0         1e-3    1100    5           -4.639946380185331          4.7823326587677
rbf         -               -           scale       0.0         1e-3    1300    5           -4.650582388344436          5.349377393722534
rbf         -               -           scale       0.0         1e-3    1500    5           -4.629035513170665          5.722083806991577
rbf         -               -           scale       0.0         1e-3    1700    5           -4.604327649887012          7.407163143157959
"""
"""
import winsound
from rich_console import console
from rich.table import Table
from tqdm import tqdm, trange
import time
table = Table(title="SVR for INM")
table.add_column('C', style='magenta')
table.add_column('epsilon', style='cyan')
table.add_column('R2', style='green')
table.add_column('time (sec)', style='white')
for c in [3000, 5000, 7000, 9000, 11000, 13000, 15000, 17000]:
    for eps in [5]:
        start_time = time.time()
        console.log('another loop has begun')
        regr = SVR(kernel='rbf', C=c, epsilon=eps)
        console.log('regression model instantiated')
        regr.fit(X_train, np.ravel(y_train))
        console.log('fitting complete')
        predictions = regr.predict(X_test)
        console.log('predictions made')
        predictions = np.where(predictions<=0, 0, predictions)
        console.log('predictions set')
        table.add_row(str(c), str(eps), str(r2_score(y_test, predictions)), str(time.time() - start_time))
        console.log('another iteration complete')
console.print(table)
winsound.Beep(2000, 1000)
"""
# %%
# file logging for SVM hyperparameter optimisation
import winsound
from rich_console import console
from rich.table import Table
from tqdm import tqdm, trange
from rich.live import Live
import time
table = Table(title="SVR for INM")
table.add_column('kernel', style='bright_yellow')
```

```
table.add_column('C', style='magenta')
table.add_column('epsilon', style='cyan')
table.add_column('R2', style='green')
table.add_column('time for run (sec)', style='white')
with open('inm_data_svm_hpp_search.txt', 'w') as f:
    try:
        f.write('Max iterations is fixed at 10000000. All other parameters take their default values.\n')
        f.write('kernel,C,epsilon,r2,time\n')
        with Live(table, refresh_per_second=0.1):
            for kernel in ['rbf', 'linear', 'sigmoid', 'poly']:
                for c in [1, 10, 40, 80, 160, 640, 1280, 3000, 6000, 10000, 15000, 20000]:
                    for eps in [13, 8, 2.5, 1]:
                        start_time = time.time()
                        regr = SVR(kernel=kernel, C=c, epsilon=eps, max_iter=10000000)
                        regr.fit(X_train, np.ravel(y_train))
                        predictions = regr.predict(X_test)
                        predictions = np.where(predictions<=0, 0, predictions)
                        time_taken = time.time() - start_time
                        score = r2_score(y_test, predictions)
                        table.add_row(str(kernel), str(c), str(eps), str(score), str(time_taken))
                        f.write(str(kernel) + ',' + str(c) + ',' + str(eps) + ',' + str(score) + ',' + str(time_taken) + '\n')
    except Exception as e:
        print('Error occured')
winsound.Beep(2000, 1000)
```

Results:

SVR for INM

kernel	C	epsilon	R2	time for run (sec)
rbf	1	13	-156.9984304635573	0.018948078155517578
rbf	1	8	-57.90322270445204	0.09779572486877441
rbf	1	2.5	-4.7763766496144155	1.8500502109527588
rbf	1	1	-0.4964225925764758	5.7316553592681885
rbf	10	13	-148.37074786443952	0.020950794219970703
rbf	10	8	-55.54879781734588	0.13619256019592285
rbf	10	2.5	-4.590509461630916	2.725139617919922
rbf	10	1	-0.447328900662359 53	12.58907699584961
rbf	40	13	-121.44632587499332	0.0249326229095459
rbf	40	8	-51.40558151334943	0.1895596981048584
rbf	40	2.5	-4.051830298084965	3.3855702877044678

rbf	40	13	-121.446325874993332	0.0249326229095459
rbf	40	8	-51.40558151334943	0.1895596981048584
rbf	40	2.5	-4.051830298084965	3.3855702877044678
rbf	40	1	-0.3509502582378974	20.408634424209595
rbf	80	13	-89.85600746971852	0.022919178009033203
rbf	80	8	-45.625350821164446	0.1741645336151123
rbf	80	2.5	-3.6132460485032505	4.220396995544434
rbf	80	1	-0.3036175569748054	28.24918246269226
rbf	160	13	-53.36657074957312	0.04098248481750488
rbf	160	8	-36.06648397503902	0.20338726043701172
rbf	160	2.5	-3.0914511082639384	5.143671989440918
rbf	160	1	-0.25113264255277334	40.99723243713379
rbf	640	13	-19.374500987946444	0.07480096817016602
rbf	640	8	-16.019336855360205	0.27250099182128906
rbf	640	2.5	-2.19498673088728637	10.523862361907959
rbf	640	1	-0.17520453335464148	107.3512008190155
rbf	1280	13	-19.374500987946444	0.07679390907287598
rbf	1280	8	-9.498138756071313	0.3335998058319092
rbf	1280	2.5	-1.9493551566087848	20.175148010253906
rbf	1280	1	-0.14328440014661603	188.75263094902039
rbf	3000	13	-19.374500987946444	0.07480001449584961
rbf	3000	8	-7.41685503798063	0.35256195068359375
rbf	3000	2.5	-1.6890232833293353	43.695119857788086
rbf	3000	1	-0.12222993733999976	395.0705215930939
rbf	6000	13	-19.374500987946444	0.07876372337341309
rbf	6000	8	-6.55453229300799	0.41947340965270996
rbf	6000	2.5	-1.5151825086765895	81.4121904373169
rbf	6000	1	-0.11243559753018885	449.63949251174927
rbf	10000	13	-19.374500987946444	0.07475042343139648
rbf	10000	8	-6.1965415165215925	0.5694773197174072
rbf	10000	2.5	-1.42696422902068	148.33211994171143
rbf	10000	1	-0.11921600779373112	506.8137638568878
rbf	15000	13	-19.374500987946444	0.07579255104064941
rbf	15000	8	-6.386863059514676	0.5231819152832031
rbf	15000	2.5	-1.37929529760101113	197.84984016418457
rbf	15000	1	-0.13536534961958324	592.3844263553619
rbf	20000	13	-19.374500987946444	0.07674741744995117
rbf	20000	8	-7.028081214274479	0.5722389221191406
rbf	20000	2.5	-1.3393063275596556	255.21826815605164
rbf	20000	1	-0.1422584698478606	657.3405091762543

rbf	20000	8	-7.028081214274479	0.5722389221191406
rbf	20000	2.5	-1.339306327559656	255.21826815605164
rbf	20000	1	-0.1422584698478606	657.3405091762543
linear	1	13	-157.9049111287263	0.029957056045532227
linear	1	8	-58.16286452173919	0.024995803833007812
linear	1	2.5	-4.802510057378358	0.5926938056945801
linear	1	1	-0.5028264281850732	2.3205113410949707
linear	10	13	-157.88996266164148	0.03889656066894531
linear	10	8	-58.163618646835275	0.06482601165771484
linear	10	2.5	-4.802229491065942	1.8715713024139404
linear	10	1	-0.5027901225203988	9.555987358093262
linear	40	13	-157.92279042948144	0.04787206649780273
linear	40	8	-58.16438078231083	0.17054367065429688
linear	40	2.5	-4.802244482126891	5.8260438442230225
linear	40	1	-0.5027075040105293	230.77042174339294
linear	80	13	-157.91530122178972	0.058807373046875
linear	80	8	-58.164847610070396	0.4213826656341553
linear	80	2.5	-4.801962248878247	27.648104190826416

As you can see increasing the value of C and decreasing the value of epsilon we are getting consistently better results.
So I think we can safely try with epsilon as 0.1 (default) and vary C between 3k and 20k. But even now, it is evident that the higher values of epsilon are not taking a lot of time. Thus, doing what I have proposed should not make too much of a difference in time, so I think we can continue doing what we are currently doing. maybe we can check for higher values of C.
We have stopped learning here 30-Apr-21 12:46 AM since there was found to be no utility of evaluating so much when even at C=100 and epsilon=0.1 we are getting much better results.
I have tried just C=100 and epsilon = 0.1 to see which kernel is giving best results.
Code:

```
# In this one I will be performing statistical analysis and linear regression using statsmodels api on INM-CM4-8 point 1 data.
# %%
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import statsmodels.api as sm
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline
from sklearn.svm import SVR
# X is INM data
# y is IMD data
```

```
from sklearn.svm import SVR
# X is INM data
# y is IMD data
x_path = r'D:\Mimisbrunnr\Data\Civil_Data\INMCM4-8_historical_predictors_Point_1.csv'
y_path = r'D:\Mimisbrunnr\Data\Civil_Data\IMD_Rainfall_0.25x0.25\rain_Point_1.csv'
X = pd.read_csv(x_path)
y = pd.read_csv(y_path)
# dropping longitude and latitude since they are fixed
X.drop(columns=['lon', 'lat', 'nos'], inplace=True)
y.drop(columns=['LONGITUDE', 'LATITUDE'], inplace=True)
# removing duplicates
X.drop_duplicates(keep='first', inplace=True)
# removing everything beyond 2015-01-01 from IMD data
y.drop([i for i in range(23376, len(y))], inplace=True)
# removing everything before 1951-01-01 from INM data
X.drop([i for i in range(365)], inplace=True)
# IMD data has leap years, INM data does not. removing them
y.drop(y[y['TIME'].map(lambda s: s.split()[0][-5:]) == '02-29'].index, inplace = True)
# set indices so that they start from 0
X.reset_index(inplace=True, drop=True)
y.reset_index(inplace=True, drop=True)
# cannot have a string column, so split it into 3
X_dtCol = pd.to_datetime(X['time'], format='%Y-%m-%d %H:%M:%S')
X['time'] = X_dtCol
# convert datetime to day, month and year
# for X
X['day'] = X['time'].dt.day
X['month'] = X['time'].dt.month
X['year'] = X['time'].dt.year
X.drop(columns=['time'], inplace=True)
# for y - commented out since y must have just rainfall
# y_dtCol = pd.to_datetime(y['TIME'], format='%Y-%m-%d %H:%M:%S')
# y['TIME'] = y_dtCol
# y['day'] = y['TIME'].dt.day
# y['month'] = y['TIME'].dt.month
# y['year'] = y['TIME'].dt.year
y.drop(columns=['TIME'], inplace=True)
# print(X.head())
# print(y.head())
y['RAINFALL'].fillna(value=y['RAINFALL'].mean(), inplace=True)
# %%
# Standardising
# if we scale then Condition Number is good in statmodels
scaler = StandardScaler()
```

```
# Standardising
# if we scale then Condition Number is good in statmodels
scaler = StandardScaler()
X = scaler.fit_transform(X[X.columns])
y = scaler.fit_transform(y)
# %%
# printing
"""
print('X.head()')
print(X.head())
print('y.head()')
print(y.head())
print('X.tail()')
print(X.tail())
print('y.tail()')
print(y.tail())
print()
print('X.columns')
print(X.columns)
"""
# %%
# generating test and train sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state=42)
# %%
# to compare and check whether X and y have the same set of dates
"""
with open('inm-cm4-8.txt', 'w') as f:
    for ind in range(len(X)):
        f.write(str(X['time'].iloc[ind]).split(' ')[0] + '\n')
with open('imd.txt', 'w') as f:
    for ind in range(len(y)):
        f.write(str(y['TIME'].iloc[ind]).split(' ')[0] + '\n')
"""
# %%
# Linear Regression
"""
model = LinearRegression()
model.fit(X_train, y_train)
predictions = model.predict(X_test)
predictions = np.where(predictions<=0, 0, predictions)
print(f'The R2 Score is: {r2_score(y_test, predictions)}')
"""
# %%
# Polynomial Regression with variable degree
```

```
"""
# %%
# Polynomial Regression with variable degree
# for degree = 2 -> 0.05325821822687249
# for degree = 3 -> 0.05630877341597906
# for degree = 4 -> 0.049656805504937784
# for degree = 5 -> 0.02727273524532825
# for degree = 6 -> 0.053098448690447775
# for degree = 7 -> 0.05035674414188229
# for degree = 8 -> 0.03664462816877734
"""
for degree in range(7, 9):
    model = make_pipeline(PolynomialFeatures(degree=degree),LinearRegression())
    model.fit(X_train, y_train)
    predictions = model.predict(X_test)
    predictions = np.where(predictions<=0, 0, predictions)
    print(f'The R2 Score is: {r2_score(y_test, predictions)}')
"""
# %%
# Perform Linear Regression using Ordinary Least Squares and generate a hell of a lot of statistics
# Link: https://medium.com/swlh/interpreting-linear-regression-through-statsmodels-summary-4796d359035a
"""
X = sm.add_constant(X)
model = sm.OLS(y, X)
res = model.fit()
print(res.summary())
"""
# %%
# generate the autocorrelation matrix of the rainfall time series data
# autocorrelation is the correlation of a given sequence with itself as a function of time lag

# %%
# SVR
# Hyperparameters:
# kernel = {'linear', 'poly', 'rbf', 'sigmoid'}, default='rbf'
# degree: int, default=3 (only for poly)
# gamma{'scale', 'auto'} or float, default='scale'
#     Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.
#     if gamma='scale' (default) is passed then it uses 1 / (n_features * X.var()) as value of gamma,
#     if 'auto', uses 1 / n_features.
# coef0: float, default=0.0 (only in poly, sigmoid)
# tol: float, default=1e-3
# C: float, default=1.0 (regularisation) always +ve
# epsilon: float, default=0.1
```

```
# tol: float, default=1e-3
# C: float, default=1.0 (regularisation) always +ve
# epsilon: float, default=0.1
# max_iter: int, default=-1
"""
Results:
kernel      max_iter        degree      gamma       coef0       tol     C       epsilon     R2                          time-taken (s)
linear      150000          -           scale       0.0         1e-3    1.0     0.1         -7.297947472117983e-06
linear      150000          -           auto        0.0         1e-3    1.0     0.1         -7.297947472117983e-06
rbf         150000          -           scale       0.0         1e-3    1.0     0.1         -7.297947472117983e-06
linear      150000          -           auto        0.0         1e-3    0.1     0.1         -7.297947472117983e-06
linear      150000          -           auto        0.0         1e-3    10      0.1         -7.297947472117983e-06
poly        1000000         5           scale       0.0         1e-3    1.0     0.1         -1.839877979259441e-05
poly        1000000\        7           scale       0.0         1e-3    1.0     0.1         -0.0016505121393148858
rbf         -               -           scale       0.0         1e-3    0.001   0.1         -7.297947472117983e-06
rbf         -               -           scale       0.0         1e-3    0.01    0.1         -7.297947472117983e-06
rbf         -               -           scale       0.0         1e-3    0.1     0.1         -7.297947472117983e-06
rbf         -               -           scale       0.0         1e-3    1.0     0.1         -7.297947472117983e-06
rbf         -               -           scale       0.0         1e-3    10.0    0.1         5.346235345804473e-05
rbf         -               -           scale       0.0         1e-3    100     0.1         0.0023060365121116977
rbf         -               -           scale       0.0         1e-3    1000    0.1         0.013924728123650532
rbf         -               -           scale       0.0         1e-3    1000    10          -21.417639281476077         0.6166067123413086
rbf         -               -           scale       0.0         1e-3    1500    10          -16.368526014747765         0.8817434310913086
rbf         -               -           scale       0.0         1e-3    2000    10          -14.06508737925242          1.18355131149292
rbf         -               -           scale       0.0         1e-3    2500    10          -14.045774460318626         1.4827511310577393
rbf         -               -           scale       0.0         1e-3    3000    10          -14.50064409277358          1.785775899887085
rbf         -               -           scale       0.0         1e-3    3000    5           -5.039082037067492          1.8954386711120605
rbf         -               -           scale       0.0         1e-3    5000    5           -4.726379833329805          2.569713830947876
rbf         -               -           scale       0.0         1e-3    7000    5           -4.621419027024011          3.373181104660034
rbf         -               -           scale       0.0         1e-3    9000    5           -4.586154773157784          4.131232500076294
rbf         -               -           scale       0.0         1e-3    1100    5           -4.639946380185331          4.7823326587677
rbf         -               -           scale       0.0         1e-3    1300    5           -4.650582388344436          5.349377393722534
rbf         -               -           scale       0.0         1e-3    1500    5           -4.629035513170665          5.722083806991577
rbf         -               -           scale       0.0         1e-3    1700    5           -4.604327649887012          7.407163143157959
"""
"""
import winsound
from rich_console import console
from rich.table import Table
from tqdm import tqdm, trange
import time
table = Table(title="SVR for INM")
table.add_column('C', style='magenta')
table.add_column('epsilon', style='cyan')
```

```
import time
table = Table(title="SVR for INM")
table.add_column('C', style='magenta')
table.add_column('epsilon', style='cyan')
table.add_column('R2', style='green')
table.add_column('time (sec)', style='white')
for c in [3000, 5000, 7000, 9000, 11000, 13000, 15000, 17000]:
    for eps in [5]:
        start_time = time.time()
        console.log('another loop has begun')
        regr = SVR(kernel='rbf', C=c, epsilon=eps)
        console.log('regression model instantiated')
        regr.fit(X_train, np.ravel(y_train))
        console.log('fitting complete')
        predictions = regr.predict(X_test)
        console.log('predictions made')
        predictions = np.where(predictions<=0, 0, predictions)
        console.log('predictions set')
        table.add_row(str(c), str(eps), str(r2_score(y_test, predictions)), str(time.time() - start_time))
        console.log('another iteration complete')
console.print(table)
winsound.Beep(2000, 1000)
"""
# %%
# file logging for SVM hyperparameter optimisation
import winsound
from rich_console import console
from rich.table import Table
from tqdm import tqdm, trange
from rich.live import Live
import time
table = Table(title="SVR for INM")
table.add_column('kernel', style='bright_yellow')
table.add_column('C', style='magenta')
table.add_column('epsilon', style='cyan')
table.add_column('R2', style='green')
table.add_column('time for run (sec)', style='white')
# if below line has file opened as 'w' and file already has useful data, then change open type to 'a'
with open('inm_data_svm_hpp_search.txt', 'w') as f:
    try:
        f.write('Max iterations is fixed at 10000000. All other parameters take their default values.\n')
        f.write('kernel,C,epsilon,r2,time\n')
        with Live(table, refresh_per_second=0.1):
            for kernel in ['rbf',]:
                for c in [100]:
```

```
        f.write('kernel,C,epsilon,r2,time\n')
        with Live(table, refresh_per_second=0.1):
            for kernel in ['rbf',]:
                for c in [100]:
                    for eps in [0.01]:
                        start_time = time.time()
                        regr = SVR(kernel=kernel, C=c, epsilon=eps, max_iter=10000000)
                        regr.fit(X_train, np.ravel(y_train))
                        predictions = regr.predict(X_test)
                        predictions = np.where(predictions<=0, 0, predictions)
                        time_taken = time.time() - start_time
                        score = r2_score(y_test, predictions)
                        table.add_row(str(kernel), str(c), str(eps), str(score), str(time_taken))
                        f.write(str(kernel) + ',' + str(c) + ',' + str(eps) + ',' + str(score) + ',' + str(time_taken) + '\n')
    except Exception as e:
        print('Error occured')
winsound.Beep(2000, 1000)
```

Results

SVR for INM

kernel	C	epsilon	R2	time for run (sec)
rbf	100	0.1	0.0023060365121116977	112.35501980781555
linear	100	0.1	-7.297947472117983e-06	1138.1326684951782
sigmoid	100	0.1	-104239652.04005142	19.789786100387573
poly	100	0.1	-7.297947472117983e-06	897.8196425437927

As we can see rbf is giving best results. So I will stick to it and try and improve it. 30-Apr-21 01:50 AM