```
#sd12 sd13  sd14  sd15  sd12.5     sd12.5adj  sd22  sd23  sd24  sd25
YY='sd12'
print(YY)
```

```
sd12
```

Value to be predicted is in the above cell

##Importing required libraries

```
import sys
sys.path.insert(1, '../input/algorithmwave/Wave/Program')
import myanfis
import pandas as pd
import sys
sys.maxsize
from sklearn.model_selection import KFold
from sklearn.preprocessing import MinMaxScaler
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
import warnings
warnings.filterwarnings("ignore")
epval=1000
# ==========================================================
```

##Read the data

```
df = pd.read_csv("../input/algorithmwave/Wave/Program/odnew.csv.xls")
```

```
df.head(10)
```

```
      R   d  Hi/gt2   d/gT2       S      hs/d      d/hs  Hi    T
sd12  \
0  55.0  45  2.1237  0.0319  0.0389  1.444444  0.692308   3  1.2
0.143440
1  55.0  45  4.2474  0.0319  0.0389  1.444444  0.692308   6  1.2
0.182160
2  55.0  45  6.3710  0.0319  0.0389  1.444444  0.692308   9  1.2
0.205304
3  55.0  45  1.5603  0.0234  0.0286  1.444444  0.692308   3  1.4
0.146960
4  55.0  45  3.1205  0.0234  0.0286  1.444444  0.692308   6  1.4
0.174240
5  55.0  45  4.6808  0.0234  0.0286  1.444444  0.692308   9  1.4
0.178640
6  55.0  45  6.2410  0.0234  0.0286  1.444444  0.692308  12  1.4
0.206360
7  55.0  45  1.1946  0.0179  0.0219  1.444444  0.692308   3  1.6
0.101562
```

```
8  55.0  45  2.3891  0.0179  0.0219  1.444444  0.692308   6  1.6
0.155926
9  55.0  45  3.5837  0.0179  0.0219  1.444444  0.692308   9  1.6
0.178552

      sd13    sd14   sd15  sd12.5  sd12.5adj      sd22      sd23
sd24  \
0  0.21550  0.2680  0.286  0.1630    0.13040  0.989659  0.976504
0.963419
1  0.24300  0.2790  0.329  0.2070    0.16560  0.983269  0.970026
0.960291
2  0.26565  0.2980  0.338  0.2333    0.18664  0.978698  0.964070
0.954566
3  0.21150  0.2560  0.277  0.1670    0.13360  0.989142  0.977378
0.966677
4  0.23100  0.2640  0.297  0.1980    0.15840  0.984703  0.972954
0.964523
5  0.24600  0.2889  0.326  0.2030    0.16240  0.983915  0.969283
0.957359
6  0.26660  0.2987  0.333  0.2345    0.18760  0.978476  0.963807
0.954347
7  0.18520  0.2550  0.277  0.1154    0.09230  0.994829  0.982700
0.966941
8  0.22160  0.2660  0.286  0.1772    0.14180  0.987769  0.975139
0.963973
9  0.24550  0.2880  0.299  0.2029    0.16230  0.983930  0.969409
0.957630

       sd25
0  0.958230
1  0.944330
2  0.941146
3  0.960870
4  0.954877
5  0.945370
6  0.942927
7  0.960870
8  0.958230
9  0.954253
```

```
df.info()
```

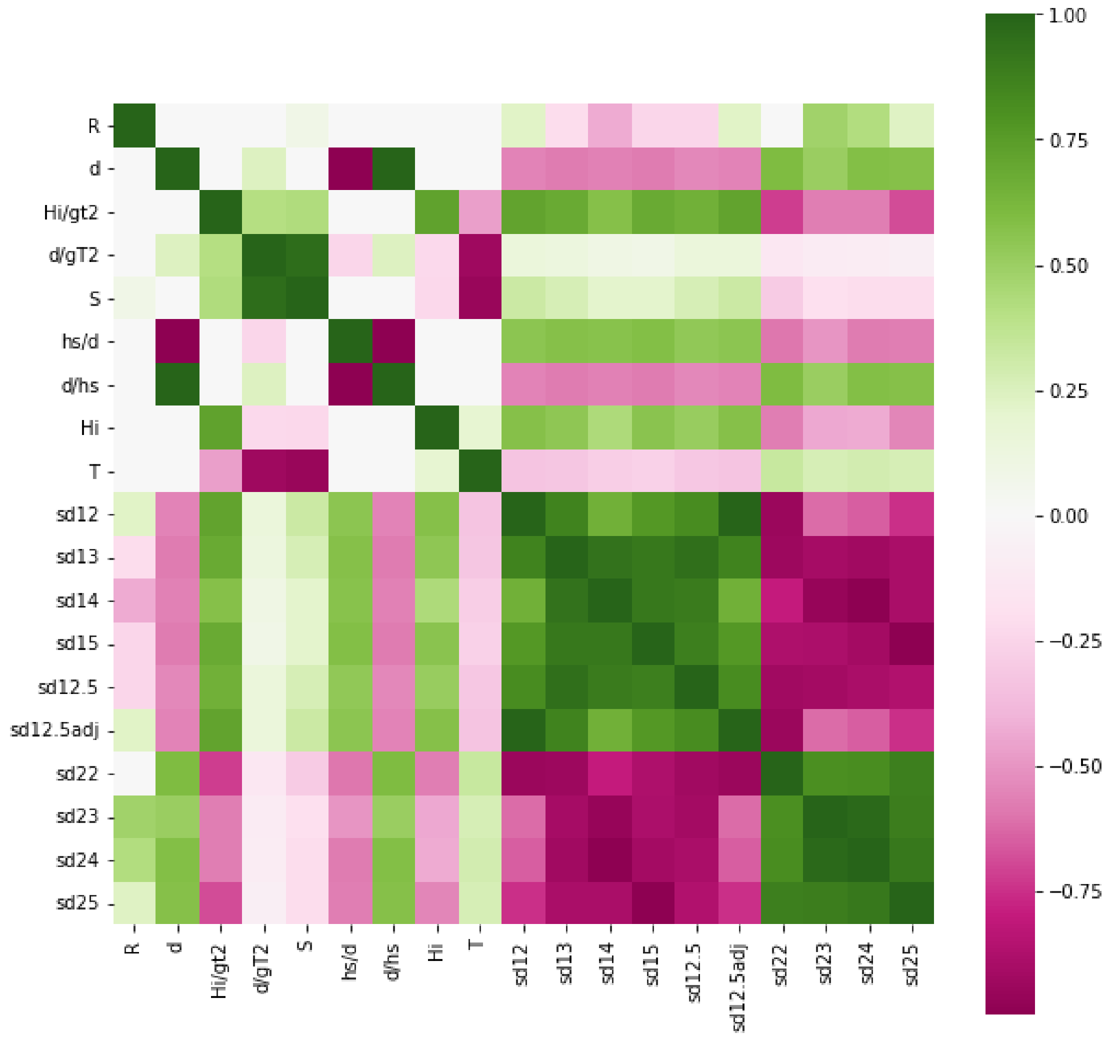
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 252 entries, 0 to 251
Data columns (total 19 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   R          252 non-null    float64
 1   d          252 non-null    int64
 2   Hi/gt2     252 non-null    float64
```

```
 3   d/gT2       252 non-null    float64
 4   S           252 non-null    float64
 5   hs/d        252 non-null    float64
 6   d/hs        252 non-null    float64
 7   Hi          252 non-null    int64
 8   T           252 non-null    float64
 9   sd12        252 non-null    float64
 10  sd13        252 non-null    float64
 11  sd14        252 non-null    float64
 12  sd15        252 non-null    float64
 13  sd12.5      252 non-null    float64
 14  sd12.5adj   252 non-null    float64
 15  sd22        252 non-null    float64
 16  sd23        252 non-null    float64
 17  sd24        252 non-null    float64
 18  sd25        252 non-null    float64
dtypes: float64(17), int64(2)
memory usage: 37.5 KB
```

```
d=df.copy()
```

Heatmap

```
import seaborn as sns
import matplotlib.pyplot as plt
plt.figure(figsize=(10,10))
hm = sns.heatmap(df.corr(), vmax=1, square=True,annot=False,
cmap="PiYG")
plt.show()
```



##PCA

```
d=df.copy()
d.columns
```

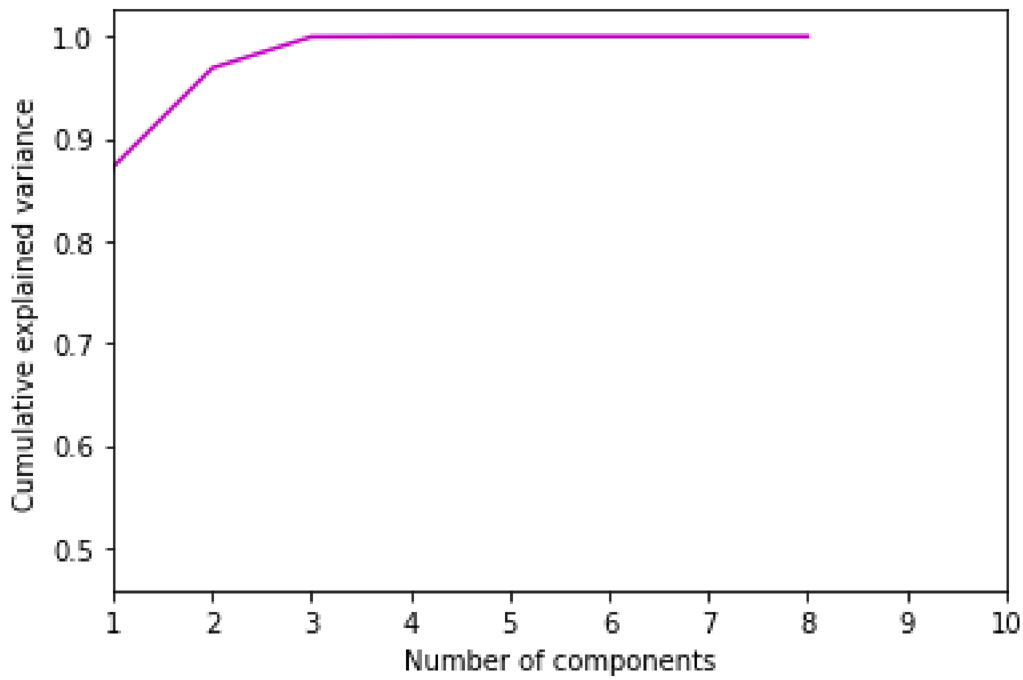
```
Index(['R', 'd', 'Hi/gt2', 'd/gT2', 'S', 'hs/d', 'd/hs', 'Hi', 'T',
'sd12',
       'sd13', 'sd14', 'sd15', 'sd12.5', 'sd12.5adj', 'sd22', 'sd23',
'sd24',
       'sd25'],
      dtype='object')
```

```
x=df[['R', 'd', 'Hi/gt2', 'd/gT2', 'S', 'hs/d', 'd/hs', 'Hi', 'T']]
y=np.array(df[YY])
```

```
from turtle import color
from sklearn.decomposition import PCA
pca = PCA().fit(x)
plt.plot(np.cumsum(pca.explained_variance_ratio_),color='m')
plt.xlim(1,10)
```

```
plt.xlabel('Number of components')
plt.ylabel('Cumulative explained variance')
```

```
Text(0, 0.5, 'Cumulative explained variance')
```



Select 5 Number of Components

```
nc=5
```

```
pca = PCA(n_components=nc)
fit = pca.fit(x)
print("Explained Variance:",fit.explained_variance_ratio_)
```

```
Explained Variance: [4.84176308e-01 3.87983090e-01 9.68079745e-02
3.06220671e-02
 4.06947824e-04]
```

```
from sklearn.decomposition import PCA

model = PCA(n_components=nc).fit(x)
X_pc = model.transform(x)
n_pcs= model.components_.shape[0]
most_important = [np.abs(model.components_[i]).argmax() for i in
range(n_pcs)]
initial_feature_names = x.columns

ldf=[]
for i in most_important:
    if initial_feature_names[i] not in ldf:
        ldf.append(initial_feature_names[i])
```

```
ldf
#PCA Results (Columns to be selected for training)
```

```
['Hi', 'd', 'R', 'Hi/gt2', 'T']
```

```
df=d.copy()
```

```
x=df[ldf]
y=np.array(df[YY])
```

```
from sklearn.preprocessing import StandardScaler
scale = StandardScaler().fit(x)
x = scale.transform(x)
#print(x)
#print(y)
```

```
minmaxScaler = MinMaxScaler().fit(x)
x = minmaxScaler.transform(x)
#print(x)
#print(y)
```

X,Y,TT-Split

```
y1 = [[i] for i in y]
from sklearn.preprocessing import StandardScaler
data = y1
scaler = StandardScaler()
scaler.fit(data)
y1 = scaler.transform(data)
#print(y1)
```

```
X_train,X_test,y_train,y_test=train_test_split(x,y1, test_size=0.2,
random_state=47)
dop=pd.DataFrame()
dop["Actual"+YY]=list(y)
```

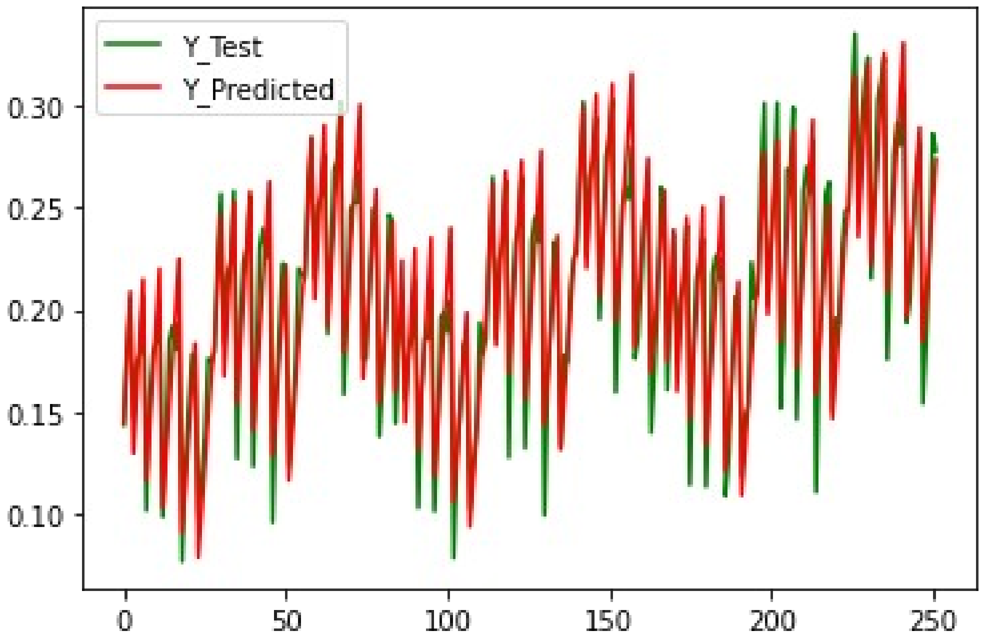
##**Linear Regression**

```
from sklearn.linear_model import LinearRegression
model=LinearRegression().fit(X_train,y_train)
MTP=model.predict(X_train)
mtp=model.predict(X_test)

from sklearn.metrics import mean_squared_error
errorstr = mean_squared_error(y_train,MTP)
errorste = mean_squared_error(y_test, mtp)
print('\nrmse(On train,On test)=',(errorstr**0.5,errorste**0.5))

from sklearn.metrics import mean_absolute_error
errorstr = mean_absolute_error(y_train,MTP)
errorste = mean_absolute_error(y_test, mtp)
print('\nmae(On train,On test)=',(errorstr,errorste))
```

```python
from sklearn.metrics import r2_score
r2tr = r2_score(y_train,MTP)
r2te = r2_score(y_test, mtp)
print('\nR2(On train,On test)=',(r2tr,r2te))

print("\nCOEF",model.coef_,"\nINTERCEPT",model.intercept_)

pred=model.predict(x)
```

```
rmse(On train,On test)= (0.32798846194322545, 0.31772423828146346)

mae(On train,On test)= (0.2647833184304276, 0.24741751710588583)

R2(On train,On test)= (0.89440033658004, 0.8887241530070377)

COEF [[ 1.67793472 -1.39630129  0.56035397  0.69874654 -1.03883116]]
INTERCEPT [0.0356928]
```

```python
# for inverse transformation
y2 = [i for i in pred]
#print(y2)
pred = scaler.inverse_transform(y2)
pred = [i[0] for i in pred]
#print(pred)

line_1 = y
line_2 = pred
fig, ax = plt.subplots()

ax.plot(line_1, color = 'green', label = 'Y_Test')
ax.plot(line_2, color = 'red', label = 'Y_Predicted')
ax.legend(loc = 'upper left')
plt.show()
```




```
dop['LR']= line_2;
```

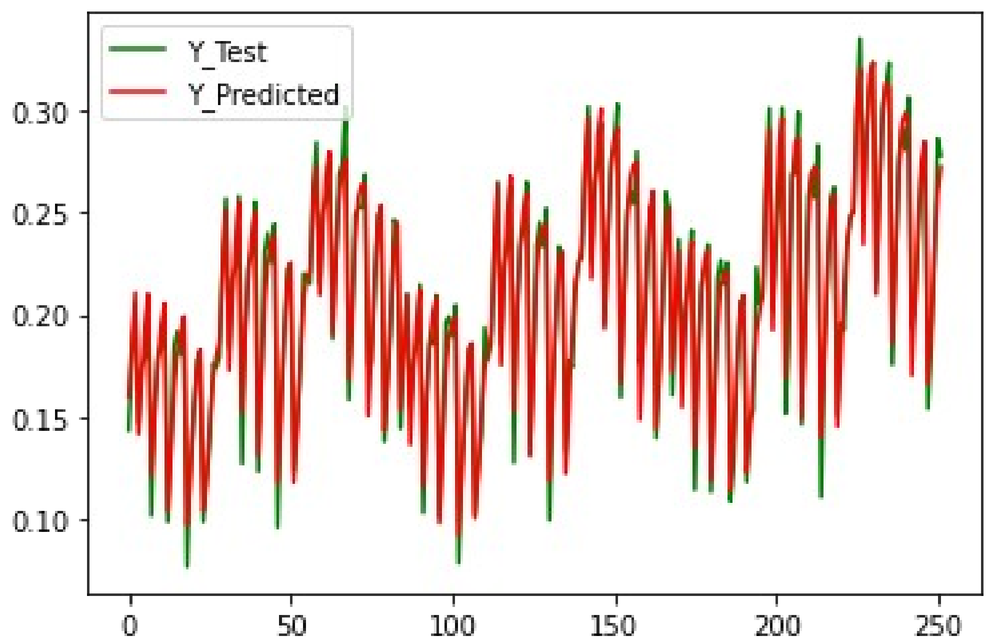
svm

```
from sklearn.svm import SVR

model=SVR().fit(X_train,y_train)
MTP=model.predict(X_train)
mtp=model.predict(X_test)

from sklearn.metrics import mean_squared_error
errorstr = mean_squared_error(y_train,MTP)
errorste = mean_squared_error(y_test, mtp)
print('\nrmse(On train,On test)=',(errorstr**0.5,errorste**0.5))

from sklearn.metrics import mean_absolute_error
errorstr = mean_absolute_error(y_train,MTP)
errorste = mean_absolute_error(y_test, mtp)
print('\nmae(On train,On test)=',(errorstr,errorste))

from sklearn.metrics import r2_score
r2tr = r2_score(y_train,MTP)
r2te = r2_score(y_test, mtp)
print('\nR2(On train,On test)=',(r2tr,r2te))

pred=model.predict(x)
```

```
rmse(On train,On test)= (0.18645308957313303, 0.2505143124232388)
```

```
mae(On train,On test)= (0.1454867807966416, 0.20090211383896409)

R2(On train,On test)= (0.9658740641794606, 0.9308224303197782)
```

```
# for inverse transformation
y2 = [[i] for i in pred]
pred = scaler.inverse_transform(y2)
pred = [i[0] for i in pred]
#print(pred)

line_1 = y
line_2 = pred
fig, ax = plt.subplots()

ax.plot(line_1, color = 'green', label = 'Y_Test')
ax.plot(line_2, color = 'red', label = 'Y_Predicted')
ax.legend(loc = 'upper left')
plt.show()
```




```
dop['SVMR']= line_2;
```

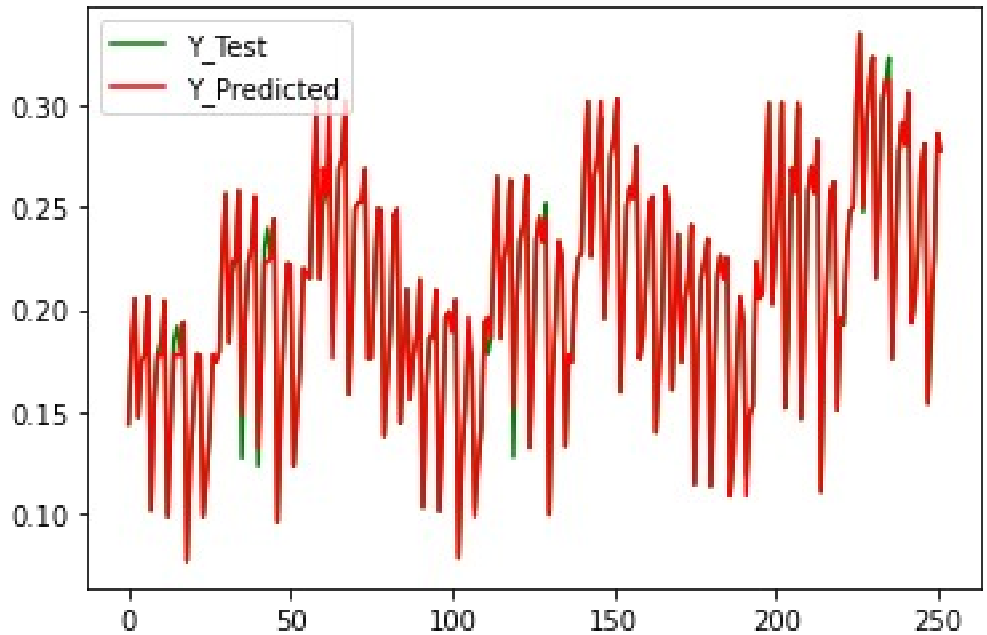
##**Decision Tree Regressor**

```
from sklearn.tree import DecisionTreeRegressor
model=DecisionTreeRegressor().fit(X_train,y_train)
MTP=model.predict(X_train)
mtp=model.predict(X_test)

from sklearn.metrics import mean_squared_error
errorstr = mean_squared_error(y_train,MTP)
errorste = mean_squared_error(y_test, mtp)
```

```python
print('\nrmse(On train,On test)=',(errorstr**0.5,errorste**0.5))

from sklearn.metrics import mean_absolute_error
errorstr = mean_absolute_error(y_train,MTP)
errorste = mean_absolute_error(y_test, mtp)
print('\nmae(On train,On test)=',(errorstr,errorste))

from sklearn.metrics import r2_score
r2tr = r2_score(y_train,MTP)
r2te = r2_score(y_test, mtp)
print('\nR2(On train,On test)=',(r2tr,r2te))

pred=model.predict(x)
```

```
rmse(On train,On test)= (0.0, 0.20831169457678395)

mae(On train,On test)= (0.0, 0.15899471525651174)

R2(On train,On test)= (1.0, 0.9521670112264253)
```

```python
# for inverse transformation
y2 = [[i] for i in pred]
pred = scaler.inverse_transform(y2)
pred = [i[0] for i in pred]
#print(pred)

line_1 = y
line_2 = pred
fig, ax = plt.subplots()

ax.plot(line_1, color = 'green', label = 'Y_Test')
ax.plot(line_2, color = 'red', label = 'Y_Predicted')
ax.legend(loc = 'upper left')
plt.show()
```




```
dop['DT']= line_2;
```

##**Random Forest Regressor**

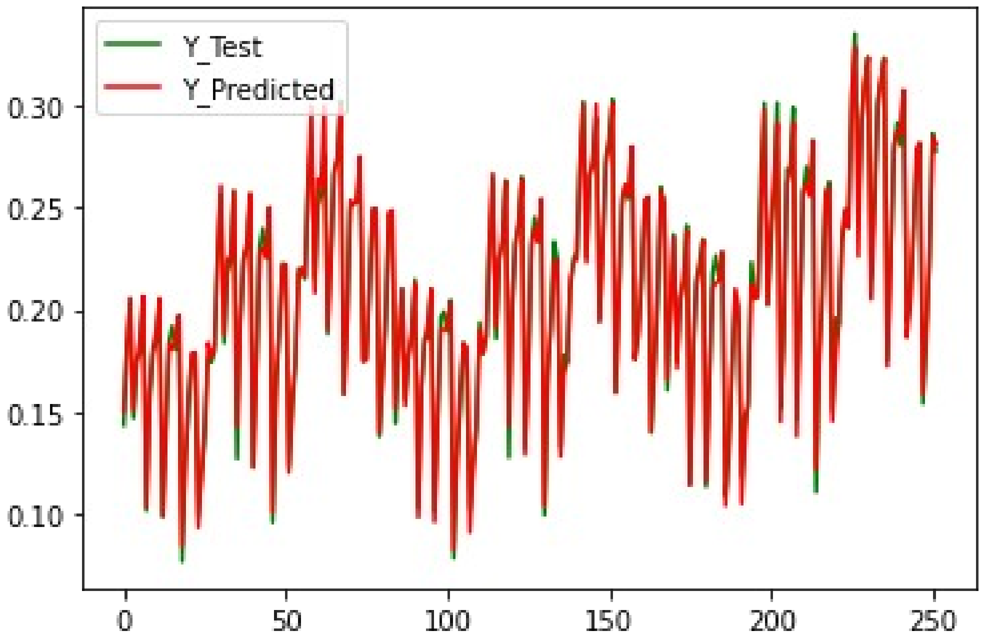
```
X_train,X_test,y_train,y_test=train_test_split(x,y1, test_size=0.2,
random_state=47)
```

```
from sklearn.ensemble import RandomForestRegressor
r2val=0
for i in [17,47,73]:
#for i in [0]:
    print(i,end=',')
    X_train,X_test,y_train,y_test=train_test_split(x,y, test_size=0.2,
random_state=i)
    model=RandomForestRegressor(n_estimators=12).fit(X_train,y_train)
    model_Train_pred=model.predict(X_train)
    model_pred=model.predict(X_test)

    errorstr = r2_score(y_train,model_Train_pred)
    errorste = r2_score(y_test, model_pred)

    if r2val<errorste:
        r2val=errorste
        I=i
        modelfinal=model
        xtr,xte,ytr,yte=X_train,X_test,y_train,y_test
X_train,X_test,y_train,y_test=xtr,xte,ytr,yte
print("\n",I,"\n")
model=modelfinal
```

```
MTP=model.predict(X_train)
mtp=model.predict(X_test)

from sklearn.metrics import mean_squared_error
errorstr = mean_squared_error(y_train,MTP)
errorste = mean_squared_error(y_test, mtp)
print('\nrmse(On train,On test)=',(errorstr**0.5,errorste**0.5))

from sklearn.metrics import mean_absolute_error
errorstr = mean_absolute_error(y_train,MTP)
errorste = mean_absolute_error(y_test, mtp)
print('\nmae(On train,On test)=',(errorstr,errorste))

from sklearn.metrics import r2_score
r2tr = r2_score(y_train,MTP)
r2te = r2_score(y_test, mtp)
print('\nR2(On train,On test)=',(r2tr,r2te))

pred=model.predict(x)
```

```
17,47,73,
 47


rmse(On train,On test)= (0.004630038680181768, 0.009123450927145269)

mae(On train,On test)= (0.003347281721393033, 0.007118404549019605)

R2(On train,On test)= (0.9927928547890694, 0.9685755903451668)
```

```
line_1 = y
line_2 = pred
fig, ax = plt.subplots()

ax.plot(line_1, color = 'green', label = 'Y_Test')
ax.plot(line_2, color = 'red', label = 'Y_Predicted')
ax.legend(loc = 'upper left')
plt.show()
```

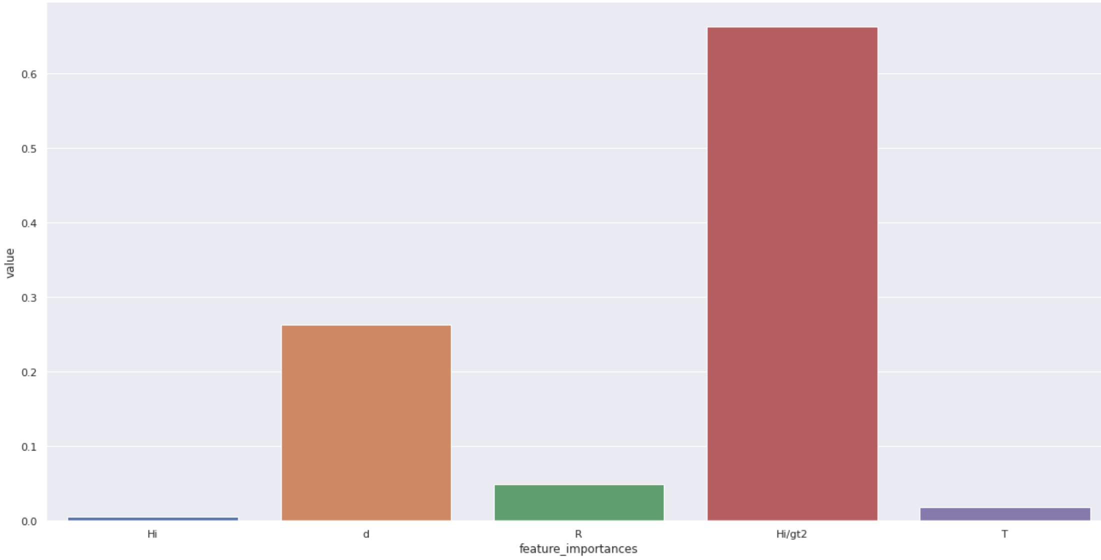


Feature Importance Graph

```
print(model.feature_importances_)
print(ldf)
sns.set(rc={'figure.figsize':(20,10)})
sns.barplot(ldf,model.feature_importances_)
plt.xlabel('feature_importances')
plt.ylabel('value')
#sns.figure(figsize=(30,10))
```

```
[0.00550265 0.26291764 0.04877673 0.66389423 0.01890875]
['Hi', 'd', 'R', 'Hi/gt2', 'T']
```

```
Text(0, 0.5, 'value')
```




```
dop['RFpred']= line_2;
```

##**ANNCG**

```
X_train,X_test,y_train,y_test=train_test_split(x,y1, test_size=0.2,
random_state=47)
from keras.models import Sequential
from keras.layers import Dense

model = Sequential()
model.add(Dense(units=5, activation='sigmoid'))
model.add(Dense(units=5, activation='relu'))
model.add(Dense(units=5, activation='sigmoid'))
model.add(Dense(1, kernel_initializer='normal'))

# Compiling the model
model.compile(loss='mean_squared_error', optimizer='adam')

# Fitting the ANN to the Training set
model.fit(X_train, y_train ,batch_size = 5, epochs = epval, verbose=0)
```

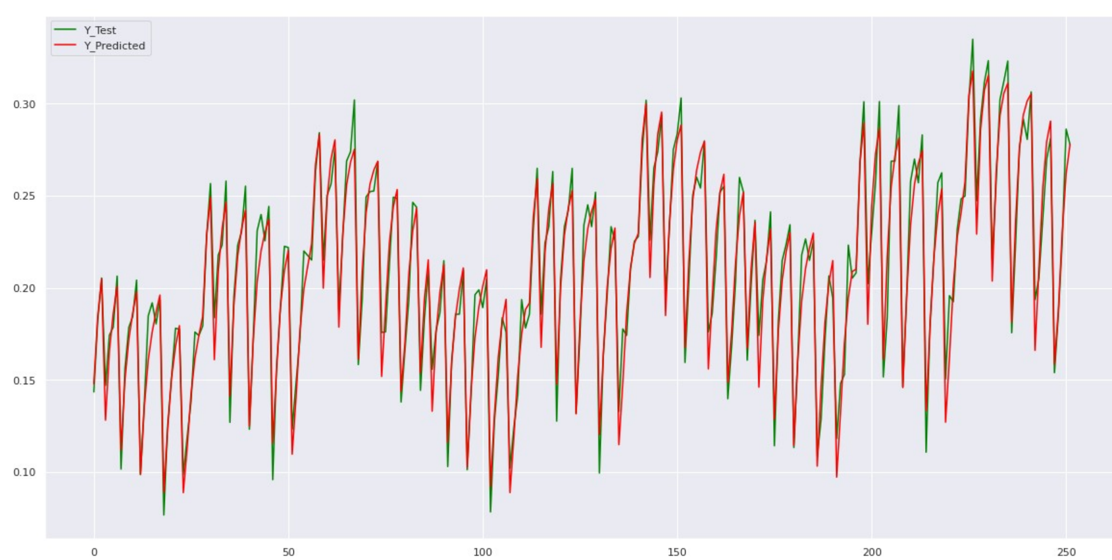
```
<keras.callbacks.History at 0x7f64bc456f90>
```

```
MTP=model.predict(X_train)
mtp=model.predict(X_test)

from sklearn.metrics import mean_squared_error
errorstr = mean_squared_error(y_train,MTP)
errorste = mean_squared_error(y_test, mtp)
print('\nrmse(On train,On test)=',(errorstr**0.5,errorste**0.5))

from sklearn.metrics import mean_absolute_error
```

```
errorstr = mean_absolute_error(y_train,MTP)
errorste = mean_absolute_error(y_test, mtp)
print('\nmae(On train,On test)=',(errorstr,errorste))

from sklearn.metrics import r2_score
r2tr = r2_score(y_train,MTP)
r2te = r2_score(y_test, mtp)
print('\nR2(On train,On test)=',(r2tr,r2te))

pred=model.predict(x)
```

```
rmse(On train,On test)= (0.21440030697883364, 0.2389190629048324)

mae(On train,On test)= (0.16788292605091332, 0.18659815218873138)

R2(On train,On test)= (0.9548771827437837, 0.9370781012359362)
```

```
# for inverse transformation
y2 = [i for i in pred]
pred = scaler.inverse_transform(y2)
pred = [i[0] for i in pred]
#print(pred)

line_1 = y
line_2 = pred
fig, ax = plt.subplots()

ax.plot(line_1, color = 'green', label = 'Y_Test')
ax.plot(line_2, color = 'red', label = 'Y_Predicted')
ax.legend(loc = 'upper left')
plt.show()
```



```
dop['ANNCG']=line_2
```

##Function for LevenBerg

```
# Copyright (c) 2020 Fabio Di Marco
#
# Permission is hereby granted, free of charge, to any person
obtaining a copy
# of this software and associated documentation files (the
"Software"), to deal
# in the Software without restriction, including without limitation
the rights
# to use.
#
=======================================================================
========

import tensorflow as tf
from tensorflow.python.keras.engine import data_adapter

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


class MeanSquaredError(tf.keras.losses.MeanSquaredError):
    """Provides mean squared error metrics: loss / residuals.

    Use mean squared error for regression problems with one or more
outputs.
    """

    def residuals(self, y_true, y_pred):
        y_pred = tf.convert_to_tensor(y_pred)
        y_true = tf.cast(y_true, y_pred.dtype)
        return y_true - y_pred


class ReducedOutputsMeanSquaredError(tf.keras.losses.Loss):
    """Provides mean squared error metrics: loss / residuals.

    Consider using this reduced outputs mean squared error loss for
regression
    problems with a large number of outputs or at least more then one
output.
    This loss function reduces the number of outputs from N to 1,
reducing both
    the size of the jacobian matrix and backpropagation complexity.
    Tensorflow, in fact, uses backward differentiation which
computational
```

```
    complexity is  proportional to the number of outputs.
    """

    def __init__(self,
                 reduction=tf.keras.losses.Reduction.AUTO,
                 name='reduced_outputs_mean_squared_error'):
        super(ReducedOutputsMeanSquaredError, self).__init__(
            reduction=reduction,
            name=name)

    def call(self, y_true, y_pred):
        y_pred = tf.convert_to_tensor(y_pred)
        y_true = tf.cast(y_true, y_pred.dtype)
        sq_diff = tf.math.squared_difference(y_true, y_pred)
        return tf.math.reduce_mean(sq_diff, axis=1)

    def residuals(self, y_true, y_pred):
        y_pred = tf.convert_to_tensor(y_pred)
        y_true = tf.cast(y_true, y_pred.dtype)
        sq_diff = tf.math.squared_difference(y_true, y_pred)
        eps = tf.keras.backend.epsilon()
        return tf.math.sqrt(eps + tf.math.reduce_mean(sq_diff,
axis=1))


"""
    The gauss-newthon algorithm is obtained from the linear
approximation of the
    squared residuals and it is used solve least square problems.
    A way to use cross-entropy instead of mean squared error is to
compute
    residuals as the square root of the cross-entropy.
"""


class
CategoricalCrossentropy(tf.keras.losses.CategoricalCrossentropy):
    """Provides cross-entropy metrics: loss / residuals.

    Use this cross-entropy loss for classification problems with two
or more
    label classes. The labels are expected to be provided in a
`one_hot`
    representation.
    """

    def residuals(self, y_true, y_pred):
        eps = tf.keras.backend.epsilon()
        return tf.math.sqrt(eps + self.fn(y_true, y_pred,
```

```
**self._fn_kwargs))


class SparseCategoricalCrossentropy(
        tf.keras.losses.SparseCategoricalCrossentropy):
    """Provides cross-entropy metrics: loss / residuals.

    Use this cross-entropy loss for classification problems with two
or more
    label classes. The labels are expected to be provided as integers.
    """

    def residuals(self, y_true, y_pred):
        eps = tf.keras.backend.epsilon()
        return tf.math.sqrt(eps + self.fn(y_true, y_pred,
**self._fn_kwargs))


class BinaryCrossentropy(tf.keras.losses.BinaryCrossentropy):
    """Provides cross-entropy metrics: loss / residuals.

    Use this cross-entropy loss for classification problems with only
two label
    classes (assumed to be 0 and 1). For each example, there should be
a single
    floating-point value per prediction.
    """

    def residuals(self, y_true, y_pred):
        eps = tf.keras.backend.epsilon()
        return tf.math.sqrt(eps + self.fn(y_true, y_pred,
**self._fn_kwargs))


"""
    Other experimental losses for classification problems.
"""


class SquaredCategoricalCrossentropy(tf.keras.losses.Loss):
    """Provides squared cross-entropy metrics: loss / residuals.

    Use this cross-entropy loss for classification problems with two
or more
    label classes. The labels are expected to be provided in a
`one_hot`
    representation.
    """
```

```python
    def __init__(self,
                 from_logits=False,
                 label_smoothing=0,
                 reduction=tf.keras.losses.Reduction.AUTO,
                 name='squared_categorical_crossentropy'):
        super(SquaredCategoricalCrossentropy, self).__init__(
            reduction=reduction,
            name=name)
        self.from_logits = from_logits
        self.label_smoothing = label_smoothing

    def call(self, y_true, y_pred):
        return
tf.math.square(tf.keras.losses.categorical_crossentropy(
            y_true,
            y_pred,
            self.from_logits,
            self.label_smoothing))

    def residuals(self, y_true, y_pred):
        return tf.keras.losses.categorical_crossentropy(
            y_true,
            y_pred,
            self.from_logits,
            self.label_smoothing)

    def get_config(self):
        config = {'from_logits': self.from_logits,
                  'label_smoothing': self.label_smoothing}
        base_config = super(SquaredCategoricalCrossentropy,
self).get_config()
        return dict(base_config + config)


class CategoricalMeanSquaredError(tf.keras.losses.Loss):
    """Provides mean squared error metrics: loss / residuals.

    Use this categorical mean squared error loss for classification
problems
    with two or more label classes. The labels are expected to be
provided in a
    `one_hot` representation and the output activation to be softmax.
    """

    def __init__(self,
                 reduction=tf.keras.losses.Reduction.AUTO,
                 name='categorical_mean_squared_error'):
        super(CategoricalMeanSquaredError, self).__init__(
            reduction=reduction,
```

```
            name=name)

    def call(self, y_true, y_pred):
        y_pred = tf.convert_to_tensor(y_pred)
        y_true = tf.cast(y_true, y_pred.dtype)
        # Selects the y_pred which corresponds to y_true equal to 1.
        prediction = tf.reduce_sum(tf.math.multiply(y_true, y_pred),
axis=1)
        return tf.math.squared_difference(1.0, prediction)

    def residuals(self, y_true, y_pred):
        y_pred = tf.convert_to_tensor(y_pred)
        y_true = tf.cast(y_true, y_pred.dtype)
        # Selects the y_pred which corresponds to y_true equal to 1.
        prediction = tf.reduce_sum(tf.math.multiply(y_true, y_pred),
axis=1)
        return 1.0 - prediction


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


class DampingAlgorithm:
    """Default Levenberg–Marquardt damping algorithm.

    This is used inside the Trainer as a generic class. Many damping
algorithms
    can be implemented using the same interface.
    """

    def __init__(self,
                 starting_value=1e-3,
                 dec_factor=0.1,
                 inc_factor=10.0,
                 min_value=1e-10,
                 max_value=1e+10,
                 adaptive_scaling=False,
                 fletcher=False):
        """Initializes `DampingAlgorithm` instance.

        Args:
          starting_value: (Optional) Used to initialize the Trainer
internal
            damping_factor.
          dec_factor: (Optional) Used in the train_step decrease the
            damping_factor when new_loss < loss.
          inc_factor: (Optional) Used in the train_step increase the
```

```
            damping_factor when new_loss >= loss.
          min_value: (Optional) Used as a lower bound for the
damping_factor.
            Higher values improve numerical stability in the
resolution of the
            linear system, at the cost of slower convergence.
          max_value: (Optional) Used as an upper bound for the
damping_factor,
            and as condition to stop the Training process.
          adaptive_scaling: Bool (Optional) Scales the damping_factor
adaptively
            multiplying it with max(diagonal(JJ)).
          fletcher: Bool (Optional) Replace the identity matrix with
            diagonal of the gauss-newton hessian approximation, so
that there is
            larger movement along the directions where the gradient is
smaller.
            This avoids slow convergence in the direction of small
gradient.
        """
        self.starting_value = starting_value
        self.dec_factor = dec_factor
        self.inc_factor = inc_factor
        self.min_value = min_value
        self.max_value = max_value
        self.adaptive_scaling = adaptive_scaling
        self.fletcher = fletcher

    def init_step(self, damping_factor, loss):
        return damping_factor

    def decrease(self, damping_factor, loss):
        return tf.math.maximum(
            damping_factor * self.dec_factor,
            self.min_value)

    def increase(self, damping_factor, loss):
        return tf.math.minimum(
            damping_factor * self.inc_factor,
            self.max_value)

    def stop_training(self, damping_factor, loss):
        return damping_factor >= self.max_value

    def apply(self, damping_factor, JJ):
        if self.fletcher:
            damping = tf.linalg.tensor_diag(tf.linalg.diag_part(JJ))
        else:
            damping = tf.eye(tf.shape(JJ)[0], dtype=JJ.dtype)
```

```
        scaler = 1.0
        if self.adaptive_scaling:
            scaler = tf.math.reduce_max(tf.linalg.diag_part(JJ))

        damping = tf.scalar_mul(scaler * damping_factor, damping)
        return tf.add(JJ, damping)


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

class Trainer:
    """Levenberg–Marquardt training algorithm.
    """

    def __init__(self,
                 model,
                 optimizer=tf.keras.optimizers.SGD(learning_rate=1.0),
                 loss=MeanSquaredError(),
                 damping_algorithm=DampingAlgorithm(),
                 attempts_per_step=10,
                 solve_method='qr',
                 jacobian_max_num_rows=100,
                 experimental_use_pfor=True):
        """Initializes `Trainer` instance.

        Args:
          model: It is the Model to be trained, it is expected to
inherit
            from tf.keras.Model and to be already built.
          optimizer: (Optional) Performs the update of the model
trainable
            variables. When tf.keras.optimizers.SGD is used it is
equivalent
            to the operation `w = w - learning_rate * updates`, where
updates is
            the step computed using the Levenberg-Marquardt algorithm.
          loss: (Optional) An object which inherits from
tf.keras.losses.Loss
          and have an additional function to compute residuals.
          damping_algorithm: (Optional) Class implementing the damping
            algorithm to use during training.
          attempts_per_step: Integer (Optional) During the train step
when new
            model variables are computed, the new loss is evaluated
and compared
            with the old loss value. If new_loss < loss, then the new
variables
            are accepted, otherwise the old variables are restored and
```

```
            new ones are computed using a different damping-factor.
            This argument represents the maximum number of attempts,
after which
            the step is taken.
          solve_method: (Optional) Possible values are:
            'qr': Uses QR decomposition which is robust but slower.
            'cholesky': Uses Cholesky decomposition which is fast but
may fail
                when the hessian approximation is ill-conditioned.
            'solve': Uses tf.linalg.solve. I don't know what algorithm
it
                implements. But it seems a compromise in terms of
speed and
                robustness.
          jacobian_max_num_rows: Integer (Optional) When the number of
residuals
            is greater then the number of variables (overdetermined),
the
            hessian approximation is computed by slicing the input and
            accumulate the result of each computation. In this way it
is
            possible to drastically reduce the memory usage and
increase the
            speed as well. The input is sliced into blocks of size
less than or
            equal to the jacobian_max_num_rows.
          experimental_use_pfor: (Optional) If true, vectorizes the
jacobian
            computation. Else falls back to a sequential while_loop.
            Vectorization can sometimes fail or lead to excessive
memory usage.
            This option can be used to disable vectorization in such
cases.
        """
        if not model.built:
            raise ValueError('Trainer model has not yet been built. '
                             'Build the model first by calling
`build()` or '
                             'calling `fit()` with some data, or
specify an '
                             '`input_shape` argument in the first
layer(s) for '
                             'automatic build.')

        self.model = model
        self.loss = loss
        self.optimizer = optimizer
        self.damping_algorithm = damping_algorithm
        self.attempts_per_step = attempts_per_step
        self.jacobian_max_num_rows = jacobian_max_num_rows
```

```
        self.experimental_use_pfor = experimental_use_pfor

        # Define and select linear system equation solver.
        def qr(matrix, rhs):
            q, r = tf.linalg.qr(matrix, full_matrices=True)
            y = tf.linalg.matmul(q, rhs, transpose_a=True)
            return tf.linalg.triangular_solve(r, y, lower=False)

        def cholesky(matrix, rhs):
            chol = tf.linalg.cholesky(matrix)
            return tf.linalg.cholesky_solve(chol, rhs)

        def solve(matrix, rhs):
            return tf.linalg.solve(matrix, rhs)

        if solve_method == 'qr':
            self.solve_function = qr
        elif solve_method == 'cholesky':
            self.solve_function = cholesky
        elif solve_method == 'solve':
            self.solve_function = solve
        else:
            raise ValueError('Invalid solve_method.')

        # Keep track of the current damping_factor.
        self.damping_factor = tf.Variable(
            self.damping_algorithm.starting_value,
            trainable=False,
            dtype=self.model.dtype)

        # Used to backup and restore model variables.
        self._backup_variables = []

        # Since training updates are computed with shape
(num_variables, 1),
        # self._splits and self._shapes are needed to split and
reshape the
        # updates so that they can be applied to the model
trainable_variables.
        self._splits = []
        self._shapes = []

        for variable in self.model.trainable_variables:
            variable_shape = tf.shape(variable)
            variable_size = tf.reduce_prod(variable_shape)
            backup_variable = tf.Variable(
                tf.zeros_like(variable),
                trainable=False)
```

```
            self._backup_variables.append(backup_variable)
            self._splits.append(variable_size)
            self._shapes.append(variable_shape)

        self._num_variables =
tf.reduce_sum(self._splits).numpy().item()
        self._num_outputs = None

    @tf.function
    def _compute_jacobian(self, inputs, targets):
        with tf.GradientTape(persistent=True) as tape:
            outputs = self.model(inputs, training=True)
            residuals = self.loss.residuals(targets, outputs)

        jacobians = tape.jacobian(
            residuals,
            self.model.trainable_variables,
            experimental_use_pfor=self.experimental_use_pfor,
            unconnected_gradients=tf.UnconnectedGradients.ZERO)

        del tape

        num_residuals = tf.reduce_prod(tf.shape(residuals))
        jacobians = [tf.reshape(j, (num_residuals, -1)) for j in
jacobians]
        jacobian = tf.concat(jacobians, axis=1)
        residuals = tf.reshape(residuals, (num_residuals, -1))

        return jacobian, residuals, outputs

    def _init_gauss_newton_overdetermined(self, inputs, targets):
        # Perform the following computation:
        # J, residuals, outputs = self._compute_jacobian(inputs,
targets)
        # JJ = tf.linalg.matmul(J, J, transpose_a=True)
        # rhs = tf.linalg.matmul(J, residuals, transpose_a=True)
        #
        # But reduce memory usage by slicing the inputs so that the
jacobian
        # matrix will have maximum shape (jacobian_max_num_rows,
num_variables)
        # instead of (batch_size, num_variables).
        slice_size = self.jacobian_max_num_rows // self._num_outputs
        batch_size = tf.shape(inputs)[0]
        num_slices = batch_size // slice_size
        remainder = batch_size % slice_size

        JJ = tf.zeros(
            [self._num_variables, self._num_variables],
```

```
            dtype=self.model.dtype)

        rhs = tf.zeros(
            [self._num_variables, 1],
            dtype=self.model.dtype)

        outputs_array = tf.TensorArray(
            self.model.dtype, size=0, dynamic_size=True)

        for i in tf.range(num_slices):
            tf.autograph.experimental.set_loop_options(
                shape_invariants=[
                    (rhs, tf.TensorShape((self._num_variables,
None)))])

            _inputs = inputs[i * slice_size:(i + 1) * slice_size]
            _targets = targets[i * slice_size:(i + 1) * slice_size]

            J, residuals, _outputs = self._compute_jacobian(_inputs,
_targets)

            outputs_array = outputs_array.write(i, _outputs)

            JJ += tf.linalg.matmul(J, J, transpose_a=True)
            rhs += tf.linalg.matmul(J, residuals, transpose_a=True)

        if remainder > 0:
            _inputs = inputs[num_slices * slice_size::]
            _targets = targets[num_slices * slice_size::]

            J, residuals, _outputs = self._compute_jacobian(_inputs,
_targets)

            if num_slices > 0:
                outputs = tf.concat([outputs_array.concat(),
_outputs], axis=0)
            else:
                outputs = _outputs

            JJ += tf.linalg.matmul(J, J, transpose_a=True)
            rhs += tf.linalg.matmul(J, residuals, transpose_a=True)
        else:
            outputs = outputs_array.concat()

        return 0.0, JJ, rhs, outputs

    def _init_gauss_newton_underdetermined(self, inputs, targets):
        J, residuals, outputs = self._compute_jacobian(inputs,
targets)
```

```
        JJ = tf.linalg.matmul(J, J, transpose_b=True)
        rhs = residuals
        return J, JJ, rhs, outputs

    def _compute_gauss_newton_overdetermined(self, J, JJ, rhs):
        updates = self.solve_function(JJ, rhs)
        return updates

    def _compute_gauss_newton_underdetermined(self, J, JJ, rhs):
        updates = self.solve_function(JJ, rhs)
        updates = tf.linalg.matmul(J, updates, transpose_a=True)
        return updates

    def _train_step(self, inputs, targets,
                    init_gauss_newton, compute_gauss_newton):
        # J: jacobian matrix not used in the overdetermined case.
        # JJ: gauss-newton hessian approximation
        # rhs: gradient when overdetermined, residuals when
underdetermined.
        # outputs: prediction of the model for the current inputs.
        J, JJ, rhs, outputs = init_gauss_newton(inputs, targets)

        # Perform normalization for numerical stability.
        batch_size = tf.shape(inputs)[0]
        normalization_factor = 1.0 / tf.dtypes.cast(
            batch_size,
            dtype=self.model.dtype)

        JJ *= normalization_factor
        rhs *= normalization_factor

        # Compute the current loss value.
        loss = self.loss(targets, outputs)

        stop_training = False
        attempt = 0
        damping_factor = self.damping_algorithm.init_step(
            self.damping_factor, loss)

        attempts = tf.constant(self.attempts_per_step, dtype=tf.int32)

        while tf.constant(True, dtype=tf.bool):
            update_computed = False
            try:
                # Apply the damping to the gauss-newton hessian
approximation.
                JJ_damped =
self.damping_algorithm.apply(damping_factor, JJ)
```

```
                # Compute the updates:
                # overdetermined: updates = (J'*J + damping)^-
1*J'*residuals
                # underdetermined: updates = J'*(J*J' + damping)^-
1*residuals
                updates = compute_gauss_newton(J, JJ_damped, rhs)
            except Exception as e:
                del e
            else:
                if tf.reduce_all(tf.math.is_finite(updates)):
                    update_computed = True
                    # Split and Reshape the updates
                    updates = tf.split(tf.squeeze(updates, axis=-1),
self._splits)
                    updates = [tf.reshape(update, shape)
                               for update, shape in zip(updates,
self._shapes)]

                    # Apply the updates to the model
trainable_variables.
                    self.optimizer.apply_gradients(
                        zip(updates, self.model.trainable_variables))

            if attempt < attempts:
                attempt += 1

                if update_computed:
                    # Compute the new loss value.
                    outputs = self.model(inputs, training=False)
                    new_loss = self.loss(targets, outputs)

                    if new_loss < loss:
                        # Accept the new model variables and backup
them.
                        loss = new_loss
                        damping_factor =
self.damping_algorithm.decrease(
                            damping_factor, loss)
                        self.backup_variables()
                        break

                    # Restore the old variables and try a new
damping_factor.
                    self.restore_variables()

                damping_factor = self.damping_algorithm.increase(
                    damping_factor, loss)

                stop_training = self.damping_algorithm.stop_training(
```

```
                    damping_factor, loss)
                if stop_training:
                    break
            else:
                break

        # Update the damping_factor which will be used in the next
train_step.
        self.damping_factor.assign(damping_factor)
        return loss, outputs, attempt, stop_training

    def _compute_num_outputs(self, inputs, targets):
        input_shape = inputs.shape[1::]
        target_shape = targets.shape[1::]
        _inputs = tf.keras.Input(shape=input_shape,
                                 dtype=inputs.dtype)
        _targets = tf.keras.Input(shape=target_shape,
                                  dtype=targets.dtype)
        outputs = self.model(_inputs)
        residuals = self.loss.residuals(_targets, outputs)
        return tf.reduce_prod(residuals.shape[1::])

    def reset_damping_factor(self):

self.damping_factor.assign(self.damping_algorithm.starting_value)

    def backup_variables(self):
        zip_args = (self.model.trainable_variables,
self._backup_variables)
        for variable, backup in zip(*zip_args):
            backup.assign(variable)

    def restore_variables(self):
        zip_args = (self.model.trainable_variables,
self._backup_variables)
        for variable, backup in zip(*zip_args):
            variable.assign(backup)

    def train_step(self, inputs, targets):
        if self._num_outputs is None:
            self._num_outputs = self._compute_num_outputs(inputs,
targets)

        batch_size = tf.shape(inputs)[0]
        num_residuals = batch_size * self._num_outputs
        overdetermined = num_residuals >= self._num_variables

        if overdetermined:
            loss, outputs, attempts, stop_training = self._train_step(
```

```
                inputs,
                targets,
                self._init_gauss_newton_overdetermined,
                self._compute_gauss_newton_overdetermined)
        else:
            loss, outputs, attempts, stop_training = self._train_step(
                inputs,
                targets,
                self._init_gauss_newton_underdetermined,
                self._compute_gauss_newton_underdetermined)

        return loss, outputs, attempts, stop_training

    def fit(self, dataset, epochs=1, metrics=None):
        """Trains self.model on the dataset for a fixed number of
epochs.

        Arguments:
            dataset: A `tf.data` dataset, must return a tuple (inputs,
targets).
            epochs: Integer. Number of epochs to train the model.
            metrics: List of metrics to be evaluated during training.
        """
        self.backup_variables()
        steps = dataset.cardinality().numpy().item()
        stop_training = False

        if metrics is None:
            metrics = []

        pl = tf.keras.callbacks.ProgbarLogger(
            count_mode='steps',
            stateful_metrics=["damping_factor", "attempts"])

        pl.set_params(
            {"verbose": 1, "epochs": epochs, "steps": steps})

        pl.on_train_begin()

        for epoch in range(epochs):
            if stop_training:
                break

            # Reset metrics.
            for m in metrics:
                m.reset_states()

            pl.on_epoch_begin(epoch)
```

```
            iterator = iter(dataset)

            for step in range(steps):
                if stop_training:
                    break

                pl.on_train_batch_begin(step)

                data = next(iterator)

                data = data_adapter.expand_1d(data)
                inputs, targets, sample_weight = \
                    data_adapter.unpack_x_y_sample_weight(data)

                loss, outputs, attempts, stop_training = \
                    self.train_step(inputs, targets)

                # Update metrics.
                for m in metrics:
                    m.update_state(targets, outputs)

                logs = {"damping_factor": self.damping_factor,
                        "attempts": attempts,
                        "loss": loss}
                logs.update({m.name: m.result() for m in metrics})

                pl.on_train_batch_end(step, logs)

            pl.on_epoch_end(epoch)

        pl.on_train_end()


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


class ModelWrapper(tf.keras.Sequential):
    """Wraps a keras model.

    When fit is called, the wrapped model is trained using Levenberg–
Marquardt.
    """

    def __init__(self, model):
        if not model.built:
            raise ValueError('This model has not yet been built. '
                             'Build the model first by calling
```

```
`build()` or '
                              'calling `fit()` with some data, or
specify an '
                              '`input_shape` argument in the first
layer(s) for '
                              'automatic build.')

        super(ModelWrapper, self).__init__([model])
        self.model = model
        self.trainer = None

    def compile(self,
                optimizer=tf.keras.optimizers.SGD(learning_rate=1.0),
                loss=MeanSquaredError(),
                damping_algorithm=DampingAlgorithm(),
                attempts_per_step=10,
                solve_method='qr',
                jacobian_max_num_rows=100,
                experimental_use_pfor=True,
                metrics=None,
                loss_weights=None,
                weighted_metrics=None,
                **kwargs):
        super(ModelWrapper, self).compile(
            optimizer=optimizer,
            loss=loss,
            metrics=metrics,
            loss_weights=loss_weights,
            weighted_metrics=weighted_metrics,
            run_eagerly=True)

        self.trainer = Trainer(
            model=self.model,
            optimizer=optimizer,
            loss=loss,
            damping_algorithm=damping_algorithm,
            attempts_per_step=attempts_per_step,
            solve_method=solve_method,
            jacobian_max_num_rows=jacobian_max_num_rows,
            experimental_use_pfor=experimental_use_pfor)

    def train_step(self, data):
        data = data_adapter.expand_1d(data)
        inputs, targets, sample_weight = \
            data_adapter.unpack_x_y_sample_weight(data)

        loss, outputs, attempts, stop_training = \
            self.trainer.train_step(inputs, targets)

        self.compiled_metrics.update_state(targets, outputs)
```

```
        logs = {"damping_factor": self.trainer.damping_factor,
                "attempts": attempts,
                "loss": loss}
        logs.update({m.name: m.result() for m in self.metrics})

        # BUG: In tensorflow v2.2.0 and v2.3.0 setting
model.stop_training=True
        # does not stop training immediately, but only at the end of
the epoch.
        # https://github.com/tensorflow/tensorflow/issues/41174
        self.stop_training = stop_training

        return logs

    def fit(self,
            x=None,
            y=None,
            batch_size=None,
            epochs=1,
            verbose=1,
            callbacks=None,
            **kwargs):
        if verbose > 0:
            if callbacks is None:
                callbacks = []

            callbacks.append(tf.keras.callbacks.ProgbarLogger(
                count_mode='steps',
                stateful_metrics=["damping_factor", "attempts"]))

        return super(ModelWrapper, self).fit(
            x=x,
            y=y,
            batch_size=batch_size,
            epochs=epochs,
            verbose=verbose,
            callbacks=callbacks,
            **kwargs)
```

Levenberg✅

```
import tensorflow as tf
X_train,X_test,y_train,y_test=train_test_split(x,y1, test_size=0.2,
random_state=47)
xt,xte,yt,yte=X_train,X_test,y_train,y_test

from keras.models import Sequential
from keras.layers import Dense
model = Sequential()
```

```
model.add(Dense(units=5, activation='sigmoid'))
model.add(Dense(units=5, activation='relu'))
model.add(Dense(units=5, activation='sigmoid'))
model.add(Dense(units=5, activation='sigmoid'))
model.add(Dense(1, kernel_initializer='normal'))
model.compile(loss='mean_squared_error', optimizer='adam')
model.fit(xt, yt ,batch_size = 20, epochs = epval, verbose=0)
model_wrapper = ModelWrapper(tf.keras.models.clone_model(model))
model_wrapper.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=
0.1),loss=MeanSquaredError(),solve_method='solve')
print("\
n_________________________________________________________________")
print("Train using Levenberg-Marquardt")
model=model_wrapper.fit(xt, yt, epochs=epval, verbose=0)
```

```
_________________________________________________________________
Train using Levenberg-Marquardt
```

```
v=model.model.predict(x)
lp=[]
for i in v:
  lp.append(i[0])
# for inverse transformation

# for inverse transformation
y2 = [[i] for i in lp]
pred = scaler.inverse_transform(y2)
lp = [i[0] for i in pred]

line_1 = y
line_2 = lp
fig, ax = plt.subplots()
ax.plot(line_1, color = 'green', label = 'Y_Test')
ax.plot(line_2, color = 'red', label = 'Y_Predicted')
ax.legend(loc = 'upper left')
plt.show()
```


```
MTP=model.model.predict(X_train)
mtp=model.model.predict(X_test)

from sklearn.metrics import mean_squared_error
errorstr = mean_squared_error(y_train,MTP)
errorste = mean_squared_error(y_test, mtp)
print('\nrmse(On train,On test)=',(errorstr**0.5,errorste**0.5))

from sklearn.metrics import mean_absolute_error
errorstr = mean_absolute_error(y_train,MTP)
errorste = mean_absolute_error(y_test, mtp)
print('\nmae(On train,On test)=',(errorstr,errorste))

from sklearn.metrics import r2_score
r2tr = r2_score(y_train,MTP)
r2te = r2_score(y_test, mtp)
print('\nR2(On train,On test)=',(r2tr,r2te))
```

```
rmse(On train,On test)= (0.20103558189335954, 0.23418458989233176)

mae(On train,On test)= (0.15373536524305179, 0.17909228347804967)

R2(On train,On test)= (0.9603273451400975, 0.9395471414349268)
```

```
dop['LB']=line_2
```

Anfis✅

Conversion for GBELL and Gaussian

```
ix=list(df.columns).index(YY)
ix
```

9

gbellmf

```
X_train,X_test,y_train,y_test=train_test_split(x,y1, test_size=0.3,
random_state=47)
param = myanfis.fis_parameters(
    n_input=5,                # no. of Regressors
    n_memb=2,                 # no. of fuzzy memberships
    batch_size=4,            # 16 / 32 / 64 / ...
    memb_func='gbellmf',      # 'gaussian' / 'gbellmf' / 'sigmoid'
    optimizer='sgd',          # sgd / adam / ...
    # mse / mae / huber_loss / mean_absolute_percentage_error / ...
    loss='mse',
    n_epochs=epval             # 10 / 25 / 50 / 100 / ...
)

fis = myanfis.ANFIS(n_input=param.n_input,
                    n_memb=param.n_memb,
                    batch_size=param.batch_size,
                    memb_func=param.memb_func,
                    name='myanfis'
                    )

# compile model
fis.model.compile(optimizer=param.optimizer,
                  loss=param.loss
                  # ,metrics=['mse']  # ['mae', 'mse']
                  )

# fit model
history = fis.fit(X_train, y_train,
                  epochs=param.n_epochs,
                  batch_size=param.batch_size, verbose=0
                  # callbacks = [tensorboard_callback]  # for
tensorboard
                  )
print("\nSuccessful")
```

```
Successful
```

```
Ly=[]
for i in range(0,252,4):
  for j in history.model.predict(x[i:i+4]):
    Ly.append(j[0])

# for inverse transformation
y2 = [[i] for i in Ly]
pred = scaler.inverse_transform(y2)
```

```
Ly = [i[0] for i in pred]
#print(Ly)

line_1 = y
line_2 = Ly
fig, ax = plt.subplots()
ax.plot(line_1, color = 'green', label = 'Y_Test')
ax.plot(line_2, color = 'red', label = 'Y_Predicted')
ax.legend(loc = 'upper left')
plt.show()
```


```
y=line_1
MTP=line_2

from sklearn.metrics import mean_squared_error
errors = mean_squared_error(y,MTP)
print('\nrmse=',(errors**0.5))

from sklearn.metrics import mean_absolute_error
errors = mean_absolute_error(y,MTP)
print('\nmae=',(errors))

from sklearn.metrics import r2_score
r2 = r2_score(y,MTP)
print('\nR2=',(r2))
```

```
rmse= 0.013636118345323836

mae= 0.011063351057648592

R2= 0.9363160260522254
```

```
dop['AnfisGBell']=line_2
```

Gaussian

```
X_train,X_test,y_train,y_test=train_test_split(x,y1, test_size=0.3,
random_state=47)
param = myanfis.fis_parameters(
    n_input=5,                # no. of Regressors
    n_memb=2,                 # no. of fuzzy memberships
    batch_size=4,            # 16 / 32 / 64 / ...
    memb_func='gaussian',      # 'gaussian' / 'gbellmf' / 'sigmoid'
    optimizer='sgd',          # sgd / adam / ...
    # mse / mae / huber_loss / mean_absolute_percentage_error / ...
    loss='mse',
    n_epochs=epval             # 10 / 25 / 50 / 100 / ...
)

fis = myanfis.ANFIS(n_input=param.n_input,
                    n_memb=param.n_memb,
                    batch_size=param.batch_size,
                    memb_func=param.memb_func,
                    name='myanfis'
                    )

# compile model
fis.model.compile(optimizer=param.optimizer,
                  loss=param.loss
                  # ,metrics=['mse']  # ['mae', 'mse']
                  )

# fit model
history = fis.fit(X_train, y_train,
                  epochs=param.n_epochs,
                  batch_size=param.batch_size, verbose=0
                  # callbacks = [tensorboard_callback]  # for
tensorboard
                  )
print("\nSuccessful")
```

```
Successful
```

```
Ly=[]
for i in range(0,252,4):
  for j in history.model.predict(x[i:i+4]):
    Ly.append(j[0])

# for inverse transformation
y2 = [[i] for i in Ly]
pred = scaler.inverse_transform(y2)
```

```python
Ly = [i[0] for i in pred]
#print(Ly)

line_1 = y
line_2 = Ly
fig, ax = plt.subplots()
ax.plot(line_1, color = 'green', label = 'Y_Test')
ax.plot(line_2, color = 'red', label = 'Y_Predicted')
ax.legend(loc = 'upper left')
plt.show()
```


```python
y=line_1
MTP=line_2

from sklearn.metrics import mean_squared_error
errors = mean_squared_error(y,MTP)
print('\nrmse=',(errors**0.5))

from sklearn.metrics import mean_absolute_error
errors = mean_absolute_error(y,MTP)
print('\nmae=',(errors))

from sklearn.metrics import r2_score
r2 = r2_score(y,MTP)
print('\nR2=',(r2))
```

```
rmse= 0.012797564628313442

mae= 0.010228494278740374

R2= 0.9439076938910574
```

```
dop['AnfisGaussian']=line_2
```

Sigmoid

```
X_train,X_test,y_train,y_test=train_test_split(x,y1, test_size=0.3,
random_state=47)
param = myanfis.fis_parameters(
    n_input=5,                # no. of Regressors
    n_memb=2,                 # no. of fuzzy memberships
    batch_size=4,            # 16 / 32 / 64 / ...
    memb_func='sigmoid',      # 'gaussian' / 'gbellmf' / 'sigmoid'
    optimizer='sgd',          # sgd / adam / ...
    # mse / mae / huber_loss / mean_absolute_percentage_error / ...
    loss='mse',
    n_epochs=epval             # 10 / 25 / 50 / 100 / ...
)

fis = myanfis.ANFIS(n_input=param.n_input,
                    n_memb=param.n_memb,
                    batch_size=param.batch_size,
                    memb_func=param.memb_func,
                    name='myanfis'
                    )

# compile model
fis.model.compile(optimizer=param.optimizer,
                  loss=param.loss
                  # ,metrics=['mse']  # ['mae', 'mse']
                  )

# fit model
history = fis.fit(X_train, y_train,
                  epochs=param.n_epochs,
                  batch_size=param.batch_size, verbose=0
                  # callbacks = [tensorboard_callback]  # for
tensorboard
                  )
print("\nSuccessful")
```

```
Successful
```

```
Ly=[]
for i in range(0,252,4):
  for j in history.model.predict(x[i:i+4]):
    Ly.append(j[0])

# for inverse transformation
y2 = [[i] for i in Ly]
pred = scaler.inverse_transform(y2)
```

```
Ly = [i[0] for i in pred]
#print(Ly)

line_1 = y
line_2 = Ly
fig, ax = plt.subplots()
ax.plot(line_1, color = 'green', label = 'Y_Test')
ax.plot(line_2, color = 'red', label = 'Y_Predicted')
ax.legend(loc = 'upper left')
plt.show()
```


```
y=line_1
MTP=line_2

from sklearn.metrics import mean_squared_error
errors = mean_squared_error(y,MTP)
print('\nrmse=',(errors**0.5))

from sklearn.metrics import mean_absolute_error
errors = mean_absolute_error(y,MTP)
print('\nmae=',(errors))

from sklearn.metrics import r2_score
r2 = r2_score(y,MTP)
print('\nR2=',(r2))


rmse= 0.013183358362119 42

mae= 0.010569535751656737

R2= 0.9404748158267525
```

```
dop['AnfisSigmoid']=line_2
##OP
dop.to_csv(YY+"-values.csv",index=False)
```