```
#!/usr/bin/env python
# coding: utf-8
import pandas as pd
import numpy as np
import math
import matplotlib.pyplot as plt
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline
data_path = "./Data.csv"
data_raw = pd.read_csv(data_path)
data_oxide= data_raw[[ "SiO2","K2O","Na2O","CaO","MnO","P2O5","Al2O3","TiO2","FeOT"]]
data_rare = pd.read_csv("./Zr_Nb.csv")
data_rare.apply(pd.to_numeric)
type(data_oxide.T.sum())
d= data_oxide.T.sum(numeric_only= float)
df_base = 100*data_oxide.T/d
df_base=df_base.T
df_base
SiO2 = df_base[['SiO2']].values
P2O5 = df_base[['P2O5']].values
TiO2 = df_base[['TiO2']].values
Zr      = data_rare [['Zr']].values
Nb      = data_rare [['Nb']].values
Age    = data_raw[['Age']].values
data_raw[['Zr']].values
TiO2_SiO2 = TiO2/SiO2
P2O5_SiO2 = P2O5/SiO2
Zr_SiO2 = Zr/SiO2
Nb_SiO2 = Nb/SiO2
Whole_Dict = {
        "Age": Age.flatten().tolist(),
        "SiO2": SiO2.flatten().tolist(),
        "P2O5": P2O5.flatten().tolist(),
        "TiO2": TiO2.flatten().tolist(),
        "Zr": Zr.flatten().tolist(),
        "Nb": Nb.flatten().tolist(),
        "TiO2-SiO2":TiO2_SiO2.flatten().tolist(),
        "P2O5-SiO2":P2O5_SiO2.flatten().tolist(),
        "Zr-SiO2":Zr_SiO2.flatten().tolist(),
        "Nb-SiO2":Nb_SiO2.flatten().tolist(),
        "Log_Zr-SiO2":np.log(Zr_SiO2.flatten().tolist()),
        "Log_Nb-SiO2":np.log(Nb_SiO2.flatten().tolist())
        }
```

```
Data_Result = pd.DataFrame(Whole_Dict)
Data_Result
Sort_Result = Data_Result.sort_values(by='Age',ascending=False)
Sort_Result.columns
#Sort_Result.plot(x='Age', y='TiO2/SiO2')
Targets =[ 'SiO2', 'P2O5', 'TiO2', 'Zr', 'Nb', 'TiO2-SiO2',
           'P2O5-SiO2',    'Zr-SiO2', 'Nb-SiO2', 'Log_Zr-SiO2', 'Log_Nb-SiO2']
Plot_list =[]
for i in Targets:
    tmp_plot = Sort_Result.plot.scatter(x='Age', y=i)
    fig = tmp_plot.get_figure()
    fig.savefig('./result/Raw_'+i+'.pdf')
    Plot_list.append(tmp_plot)
Sort_Result.Age.plot.kde()
n = 30
get_ipython().run_line_magic('timeit', 'Sort_Result.iloc[np.random.randint(n, size=n)]')
Index_list = np.arange(min(Sort_Result.Age), max(Sort_Result.Age),100)
Pair_list = [[x, y] for x, y in zip(Index_list[0:-1], Index_list[1:])]
Piece_list =[]
Length_list =[]
for i in Pair_list:
    pass
    condition = (Sort_Result['Age']> i[0])&                                    (Sort_Result['Age']<= i[1])
    piece_tmp = Sort_Result[condition]
    Piece_list.append(piece_tmp)
    Length_list.append(len(piece_tmp))
print(Length_list[6])
print(Pair_list[6])
def get_natural_cubic_spline_model(x, y, minval=None, maxval=None, n_knots=None, knots=None):
    """
    Get a natural cubic spline model for the data.
    For the knots, give (a) `knots` (as an array) or (b) minval, maxval and n_knots.
    If the knots are not directly specified, the resulting knots are equally
    space within the *interior* of (max, min).    That is, the endpoints are
    *not* included as knots.
    Parameters
    ----------
    x: np.array of float
        The input data
    y: np.array of float
        The outpur data
    minval: float
        Minimum of interval containing the knots.
    maxval: float
```

```
        Maximum of the interval containing the knots.
    n_knots: positive integer
        The number of knots to create.
    knots: array or list of floats
        The knots.
    Returns
    --------
    model: a model object
        The returned model will have following method:
        - predict(x):
            x is a numpy array. This will return the predicted y-values.
    """
    if knots:
        spline = NaturalCubicSpline(knots=knots)
    else:
        spline = NaturalCubicSpline(max=maxval, min=minval, n_knots=n_knots)
    p = Pipeline([
        ('nat_cubic', spline),
        ('regression', LinearRegression(fit_intercept=True))
    ])
    p.fit(x, y)
    return p
class AbstractSpline(BaseEstimator, TransformerMixin):
    """Base class for all spline basis expansions."""
    def __init__(self, max=None, min=None, n_knots=None, n_params=None, knots=None):
        if knots is None:
            if not n_knots:
                n_knots = self._compute_n_knots(n_params)
            knots = np.linspace(min, max, num=(n_knots + 2))[1:-1]
            max, min = np.max(knots), np.min(knots)
        self.knots = np.asarray(knots)
    @property
    def n_knots(self):
        return len(self.knots)
    def fit(self, *args, **kwargs):
        return self
class NaturalCubicSpline(AbstractSpline):
    """Apply a natural cubic basis expansion to an array.
    The features created with this basis expansion can be used to fit a
    piecewise cubic function under the constraint that the fitted curve is
    linear *outside* the range of the knots..  The fitted curve is continuously
    differentiable to the second order at all of the knots.
    This transformer can be created in two ways:
      - By specifying the maximum, minimum, and number of knots.
```

```
    - By specifying the cutpoints directly.
If the knots are not directly specified, the resulting knots are equally
space within the *interior* of (max, min).    That is, the endpoints are
*not* included as knots.
Parameters
----------
min: float
        Minimum of interval containing the knots.
max: float
        Maximum of the interval containing the knots.
n_knots: positive integer
        The number of knots to create.
knots: array or list of floats
        The knots.
"""
def _compute_n_knots(self, n_params):
        return n_params
@property
def n_params(self):
        return self.n_knots - 1
def transform(self, X, **transform_params):
        X_spl = self._transform_array(X)
        if isinstance(X, pd.Series):
                col_names = self._make_names(X)
                X_spl = pd.DataFrame(X_spl, columns=col_names, index=X.index)
        return X_spl
def _make_names(self, X):
        first_name = "{}_spline_linear".format(X.name)
        rest_names = ["{}_spline_{}".format(X.name, idx)
                                for idx in range(self.n_knots - 2)]
        return [first_name] + rest_names
def _transform_array(self, X, **transform_params):
        X = X.squeeze()
        try:
                X_spl = np.zeros((X.shape[0], self.n_knots - 1))
        except IndexError: # For arrays with only one element
                X_spl = np.zeros((1, self.n_knots - 1))
        X_spl[:, 0] = X.squeeze()
        def d(knot_idx, x):
                def ppart(t): return np.maximum(0, t)
                def cube(t): return t*t*t
                numerator = (cube(ppart(x - self.knots[knot_idx]))
                                        - cube(ppart(x - self.knots[self.n_knots - 1])))
                denominator = self.knots[self.n_knots - 1] - self.knots[knot_idx]
```

```
            return numerator / denominator
        for i in range(0, self.n_knots - 2):
            X_spl[:, i+1] = (d(i, X) - d(self.n_knots - 2, X)).squeeze()
        return X_spl

n_knots=15
for i in Targets:
    x=[]
    y=[]
    data_pair_list =    [[x, y] for x, y in zip(Sort_Result['Age'], Sort_Result[i])]

    for j in data_pair_list:#if type(j[0])!= float or type(j[1])!= float or type(j[0])== np.NaN or type(j[1])== np.NaN :
        if np.isnan(j[0]) or np.isnan(j[1]) :
            data_pair_list.remove(j)
        else:
            x.append(j[0])
            y.append(j[1])

    print(len(data_pair_list))

    x = np.array(x)
    y = np.array(y)



    model = get_natural_cubic_spline_model(x, y, minval=min(x), maxval=max(x), n_knots=n_knots)
    y_est = model.predict(x)
    # plt.plot(x, y, ls='', marker='.', label='originals')
    plt.plot(x, y_est, color='red',label='n_knots = '+ str(n_knots))
    plt.legend()
    plt.xlabel('Age (Ma)')
    plt.ylabel(i)
    plt.savefig('./result/Fitted_'+i+'.pdf')
    plt.show()
# print(Pair_list)
Age_label_list=[]
for i in Pair_list:
    Age_label_list.append(0.5*(i[0]+i[1]))
# print(Age_label_list)
# print(Targets)
SiO2_list =[]
P2O5_list =[]
TiO2_list =[]
Zr_list =[]
Nb_list =[]
```

```
TiO2_SiO2_list =[]
P2O5_SiO2_list =[]
Zr_SiO2_list =[]
Nb_SiO2_list =[]
Log_Zr_SiO2_list =[]
Log_Nb_SiO2_list =[]
def replace_nan(list=[1,2,np.NaN,3,4,5]):
    tmp_list=[]
    for i in list:
        if np.isnan(i):
            pass
        else:
            tmp_list.append(i)
    for i in range(len(list)):
        if np.isnan(list[i]):
            if i ==0:
                list[i]= np.sqrt(list[i+1]**2)
            elif   i < len(list)-1:
                list[i]= np.sqrt(list[i-1]*list[i+1])
            else:
                list[i] = np.array(tmp_list).mean()
    return(list)
for i in range(10000):#  蒙特卡洛取样循环次数
    tmp_SiO2_list =[]
    tmp_P2O5_list =[]
    tmp_TiO2_list =[]
    tmp_Zr_list =[]
    tmp_Nb_list =[]
    tmp_TiO2_SiO2_list =[]
    tmp_P2O5_SiO2_list =[]
    tmp_Zr_SiO2_list =[]
    tmp_Nb_SiO2_list =[]
    tmp_Log_Zr_SiO2_list =[]
    tmp_Log_Nb_SiO2_list =[]
    Sample_list =[]
    n = 30
    np.random.randint(n, size=n)
    for df in Piece_list:
        # Bootstrap
        # print(len(df))
        # tmp_df = df.iloc[np.random.randint(len(df), size= int(len(df)/len(Targets)))]
        tmp_df = df.iloc[np.random.randint(min(len(df),n), size= min(len(df),n))]
        # print(tmp_df)
        Sample_list.append(tmp_df)
```

```
    #Sample_list
    for df in Sample_list:
        tmp_SiO2_list.append(df["SiO2"].mean())
        tmp_P2O5_list.append(df["P2O5"].mean())
        tmp_TiO2_list.append(df["TiO2"].mean())
        tmp_Zr_list.append(df["Zr"].mean())
        tmp_Nb_list.append(df["Nb"].mean())
        tmp_TiO2_SiO2_list.append(df["TiO2-SiO2"].mean())
        tmp_P2O5_SiO2_list.append(df["P2O5-SiO2"].mean())
        tmp_Zr_SiO2_list.append(df["Zr-SiO2"].mean())
        tmp_Nb_SiO2_list.append(df["Nb-SiO2"].mean())
        tmp_Log_Zr_SiO2_list.append(df["Log_Zr-SiO2"].mean())
        tmp_Log_Nb_SiO2_list.append(df["Log_Nb-SiO2"].mean())

    SiO2_list.append(replace_nan(tmp_SiO2_list))
    P2O5_list.append(replace_nan(tmp_P2O5_list))
    TiO2_list.append(replace_nan(tmp_TiO2_list))
    Zr_list.append(replace_nan(tmp_Zr_list))
    Nb_list.append(replace_nan(tmp_Nb_list))
    TiO2_SiO2_list.append(replace_nan(tmp_TiO2_SiO2_list))
    P2O5_SiO2_list.append(replace_nan(tmp_P2O5_SiO2_list))
    Zr_SiO2_list.append(replace_nan(tmp_Zr_SiO2_list))
    Nb_SiO2_list.append(replace_nan(tmp_Nb_SiO2_list))
    Log_Zr_SiO2_list.append(replace_nan(tmp_Log_Zr_SiO2_list))
    Log_Nb_SiO2_list.append(replace_nan(tmp_Log_Nb_SiO2_list))

SiO2_values = np.array(SiO2_list).T
P2O5_values = np.array(P2O5_list).T
TiO2_values = np.array(TiO2_list).T
Zr_values = np.array(Zr_list).T
Nb_values = np.array(Nb_list).T
TiO2_SiO2_values = np.array(TiO2_SiO2_list).T
P2O5_SiO2_values = np.array(P2O5_SiO2_list).T
Zr_SiO2_values = np.array(Zr_SiO2_list).T
Nb_SiO2_values = np.array(Nb_SiO2_list).T
Log_Zr_SiO2_values = np.array(Log_Zr_SiO2_list).T
Log_Nb_SiO2_values = np.array(Log_Nb_SiO2_list).T
SiO2_mean_std = []
P2O5_mean_std = []
TiO2_mean_std = []
Zr_mean_std = []
Nb_mean_std = []
TiO2_SiO2_mean_std = []
P2O5_SiO2_mean_std = []
```

```
Zr_SiO2_mean_std = []
Nb_SiO2_mean_std = []
Log_Zr_SiO2_mean_std = []
Log_Nb_SiO2_mean_std = []
for i in SiO2_values: SiO2_mean_std.append([    i.mean(),i.std() ] )
for i in P2O5_values: P2O5_mean_std.append([    i.mean(),i.std() ] )
for i in TiO2_values: TiO2_mean_std.append([    i.mean(),i.std() ] )
for i in Zr_values: Zr_mean_std.append([    i.mean(),i.std() ] )
for i in Nb_values: Nb_mean_std.append([    i.mean(),i.std() ] )
for i in TiO2_SiO2_values: TiO2_SiO2_mean_std.append([    i.mean(),i.std() ] )
for i in P2O5_SiO2_values: P2O5_SiO2_mean_std.append([    i.mean(),i.std() ] )
for i in Zr_SiO2_values: Zr_SiO2_mean_std.append([    i.mean(),i.std() ] )
for i in Nb_SiO2_values: Nb_SiO2_mean_std.append([    i.mean(),i.std() ] )
for i in Log_Zr_SiO2_values: Log_Zr_SiO2_mean_std.append([    i.mean(),i.std() ] )
for i in Log_Nb_SiO2_values: Log_Nb_SiO2_mean_std.append([    i.mean(),i.std() ] )
result_dict = {"SiO2":SiO2_mean_std,
"P2O5":P2O5_mean_std,
"TiO2":TiO2_mean_std,
"Zr":Zr_mean_std,
"Nb":Nb_mean_std,
"TiO2_SiO2":TiO2_SiO2_mean_std,
"P2O5_SiO2":P2O5_SiO2_mean_std,
"Zr_SiO2":Zr_SiO2_mean_std,
"Nb_SiO2":Nb_SiO2_mean_std,
"Log_Zr_SiO2":Log_Zr_SiO2_mean_std,
"Log_Nb_SiO2":Log_Nb_SiO2_mean_std}

for i in result_dict:
    data = np.array(result_dict[i])
    print(i,data[:,0],data[:,1])
    plt.errorbar(Age_label_list,                    y=data[:,0],                    yerr=data[:,1],
fmt='o-',ecolor='blue',elinewidth=1,ms=5,mfc='grey',mec='black',capsize=3)
    plt.xlabel('Age (Ma)')
    plt.ylabel(i)
    plt.savefig('./result/ErrorBar_'+i+'.pdf')
    plt.show()
result_df = pd.DataFrame(result_dict)
result_df.to_csv('./result/result_df.csv')
```