

Clustering_electric_meters

July 17, 2019

1 Cluster electric meters based on interval average voltages

This is a prototype of electric meters clustering using their voltage data. Here I will present snippets of code and plots with some explanation. I cannot expose datasets due to privacy concerns.

Data analysis process: 1. Preprocess input data 2. Perform PCA to reduce dimensionality 3. I have GIS information available, so I am computing Equirectangular distance to find nearest neighbors. 4. Run fuzzy c-means with custom distance measure (Voltage correlation) 5. Plot the clusters

In addition, I am adding analysis using different data set with DBSCAN algorithm. 1. Find eps value using nearest neighbor search 2. Apply DBSCAN

```
In [640]: import pandas as pd
import numpy as np
from sklearn.cluster import KMeans
from sklearn.cluster import DBSCAN
import skfuzzy as fuzz
import matplotlib.pyplot as plt
import seaborn as sns
from scipy.stats.stats import pearsonr
from sklearn.cluster import DBSCAN
from sklearn.cluster import AffinityPropagation
from sklearn.decomposition import PCA
import datetime
import skfuzzy as fuzz
from sklearn.neighbors import NearestNeighbors
from mpl_toolkits.mplot3d import Axes3D
from scipy.spatial.distance import pdist, squareform
import math
from scipy import fftpack
from statsmodels.tsa.seasonal import seasonal_decompose
from sklearn.cluster import AffinityPropagation

from sklearn.preprocessing import MinMaxScaler
%matplotlib inline

In [ ]: ### Preprocess data
```

```

In [ ]: df_latlong, df_voltPoints, df_trMapping = processInputData()
        df_voltPoints.shape
        df_voltPoints = normalizeVoltages(df_voltPoints)
        df_voltPoints = df_voltPoints[df_voltPoints>0.95]
        df_voltPoints = averagepoints(numAvg, df_voltPoints)

In [642]: def processIds(df):
           df['_id'] = df['_id'].apply(lambda x: x.split('_', 1)[0])
           return df

In [643]: def createPoints(df, nDays):
           dfvp = pd.DataFrame()
           for i in range(0, len(df), nDays):
               df1 = df.iloc[i:i+nDays, 1:]
               dfvp[str(df.iloc[i, 0])] = df1.stack().values
           return dfvp

In [644]: def readVoltageData(vFile):
           df_voltages = pd.read_csv(path+vFile)
           return df_voltages

In [645]: def processInputData():
           #Read meter coordinates
           df_latlong = pd.read_csv(path+MeterCoordinatesFile)
           df_latlong = processIds(df_latlong)
           #create volt points
           vp1 = createPoints(readVoltageData(VoltageFile1), 3)
           print(vp1.shape)
           vp2 = createPoints(readVoltageData(VoltageFile2), 4)
           vp2.columns = list(map(lambda x: x.split('_', 1)[0], vp2.columns))
           print(vp2.shape)
           df_voltPoints = vp1.append(vp2, ignore_index = True)
           print(df_voltPoints.shape)
           return df_latlong, df_voltPoints, df_trMapping

In [646]: #select sparse points
           df_voltPoints = df_voltPoints[df_voltPoints.index%4==0].reset_index(drop=True)

In [647]: #compute deltas
           df_voltPoints = (df_voltPoints.shift(-1)-df_voltPoints).dropna()

In [648]: def imputeVals(dfx):
           dfx = dfx.dropna().reset_index(drop=True)
           for col in dfx.columns:
               dfx[col] = dfx[col].replace(to_replace=0, method='ffill')
           return dfx

In [649]: #Compute average of specified number of points
           def averagepoints(numOfAvg, dfx):

```

```

    dfx = imputeVals(dfx)
    dfxav = pd.DataFrame()
    for col in dfx.columns:
        dfxav[col] = np.mean(dfx[col][:].values.reshape(-1, numOfAvg), axis=1)
    return dfxav
#df_voltPoints = averagepoints(numAvg, df_voltPoints)
#df_voltPoints.shape

```

```

In [652]: def computeDelta(dfvp):
            return (dfvp.shift(-1)-dfvp).dropna()
            #df_voltPoints = computeDelta(df_voltPoints)

```

1.0.1 PCA

```

In [653]: #Perform Principal Component Analysis. n--> number of components
def performPca(dfp, n, plot=False):
    pca = PCA(n_components=n)
    pc = pca.fit_transform(dfp.values)
    cols = []
    print("Performing PCA")
    print("Explained variance ratio with PCA:",pca.explained_variance_ratio_)
    for i in range(n):
        cols.append("pc"+str(i+1))
    dfpc = pd.DataFrame(data =pc, columns=cols)
    # print("PCA:", dfpc)
    if plot==True:
        plt.figure(figsize=(8, 6))
        plt.bar(dfpc.columns, pca.explained_variance_ratio_)
    return dfpc

```

```

In [655]: #Visualize 3d
            # Visualize the test data
def plotClusters3D(dfvp, labels, *args):
    pcInp = performPca(dfvp.T, 3, False)
    pcInp['labels'] = labels
    plt.figure(figsize=(14, 10))
    x = pcInp.iloc[:,0]
    y = pcInp.iloc[:,1]
    z = pcInp.iloc[:,2]
    fig = plt.figure(figsize=(10,8))
    ax = fig.add_subplot(111, projection='3d')
    ax.scatter(x, y, z)
    ax.set_xlabel('X Label')
    ax.set_ylabel('Y Label')
    ax.set_zlabel('Z Label')
    if args:
        for ar in args:
            pcntr = ar

```

```

        break
    plt.scatter(pcctr[:, 0], pcctr[:, 1], c='red', s=200, alpha=0.5);
    x = pcInp['pc1']
    y = pcInp['pc2']
    z = pcInp['pc3']
plt.show()

```

1.1 Clustering

1.2 Fuzzy c-means clustering

```

In [656]: ### Run fuzzy C means
def runFuzzyCMeans(dfvp, c, fuzzy):
    #Running fuzzy with 2 pc fails. Use min of 3.
    pcInp = performPca(dfvp.T, 3, True)

    #print("Input data points",pcInp)
    cntr, u, u0, d, jm, p, fpc = fuzz.cluster.cmeans(
        pcInp.T, c, fuzzy, error=0.005,metric=voltCorr, maxiter=1000, init=None, seed = 1)
    labels_fuzzy = np.argmax(u, axis=0)
    #print(cntr)
    print("Cluster labels:",labels_fuzzy)
    print("strength:", fpc)
    return labels_fuzzy, cntr
#labels_fuzzy, cntr_fuzzy = runFuzzyCMeans(df_voltPoints, 3, 2)

```

1.2.1 Verify clusters with result set

```

In [658]: def printClusters(dfvp, labels):
    pcInp = performPca(dfvp.T, 2, False)
    # for i in np.unique(labels):
        # print("cluster:",i)
        # print(dfvp.columns[labels==i].values)
    #printClusters(df_voltPoints, labels_fuzzy)

In [659]: # Visualize the test data
def plotClusters(dfvp, labels, annotate, *args):
    pcInp = performPca(dfvp.T, 2, False)
    pcInp['labels'] = labels
    plt.figure(figsize=(14, 10))
    sns.scatterplot(x='pc1', y='pc2', hue='labels', data=pcInp)
    if args:
        for ar in args:
            pcctr = ar
            break
    plt.scatter(pcctr[:, 0], pcctr[:, 1], c='red', s=200, alpha=0.5);
    x = pcInp['pc1']
    y = pcInp['pc2']
    if annotate:

```

```

        for i, txt in enumerate(dfvp.T.index):
            plt.annotate(txt, (x[i], y[i]))
    #plotClusters(df_voltPoints, labels_fuzzy, cntr_fuzzy)

```

In [660]: *# Visualize the test data*

```

def showClustersFuzzy(dfvp, pc, labels, cntr):
    pc_copy = pc.copy()
    pc_copy['labels'] = labels

    pcntr = pc.fit_transform(cntr)
    print("Center after pca", pcntr)
    dfvp['labels'] = labels
    #for i in np.unique(labels):
        # print("cluster:", i)
        #print(dfvp.columns[labels==i].values)
    plt.figure(figsize=(10, 8))
    sns.scatterplot(x='pc1', y='pc2', hue='labels', data=dfvp)
    plt.scatter(pcntr[:, 0], pcntr[:, 1], c='red', s=200, alpha=0.5);

```

1.3 Compute distances from meter coordinates - Euclidean/Equirectangular

In [661]: *## Equirectangular distance calculation*

```

def equirectangular(p1, p2):
    lat1 = math.radians(p1[0] )
    lat2 = math.radians(p2[0])
    lon1 = math.radians(p1[1])
    lon2 = math.radians(p2[1])
    R = 6371 * 1000
    x = (lat2 - lat1) * math.cos(0.5*(lon2+lon1))
    y = lon2 - lon1
    d = R*math.sqrt(x*x + y*y)
    return d

```

In [662]: `def computePairWiseDistMat(dflatlong):`

```

    dflatlongR = dflatlong.iloc[:,1:3]
    distances = pdist(dflatlongR.values, metric=equirectangular)
    dist_matrix = squareform(distances)
    return dist_matrix

```

In [663]: *#Get nearest meters based on radius*

```

def getNearestMeters(meterId, radius, df_coord):
    #Select a meter and get neighbors within the specified radius
    idx = df_coord.index[df_coord['_id']==meterId][0]
    print("Index:", idx)
    dist_matrix = computePairWiseDistMat(df_coord)
    df_meterDist = pd.DataFrame(list(zip(df_coord['_id'], dist_matrix[idx])), columns=
    df_meterDistSorted = df_meterDist.sort_values(by='distances').reset_index(drop=True)

```

```

        df_filtered = df_meterDistSorted[df_meterDistSorted['distances']<=radius]
        return df_filtered

    #ref_meterId = '301048685'
    #df_filteredMeters = getNearestMeters(ref_meterId, radius, df_latlong)

In [664]: def getVoltagePointsFiltered(df_filteredMeters, dfvp):
    df_filteredVp = dfvp.T.merge(df_filteredMeters, left_index=True, right_on='_id').d
    df_filteredVp.head()
    del df_filteredVp.index.name
    return df_filteredVp.T

In [665]: def getExpectedClusters(dfMeters, dfTrMap):
    dfTrMap['Name'] = dfTrMap['Name'].astype(str)
    #print(dfTrMap.head()), print(dfMeters)
    dfr = dfTrMap.merge(dfMeters, left_on='Name', right_on='_id',how='inner' )
    return len(dfr.groupby(by='Current Phase'))
    #getExpectedClusters(df_filteredMeters, df_trMapping)

In [666]: def getFftAbs(dfvp):
    df_fft = pd.DataFrame()
    for i in dfvp.columns:
        f=fftpack.fft(dfvp[i])
        df_fft[i] = abs(f)
    return df_fft

In [669]: def getProcessedData(dfvp, analysisType):
    if analysisType is AnalysisType.TIME_SERIES:
        print("performing time series decomposition")
        return getTimeSeriesTrend(dfvp, 8, False, 1).dropna().reset_index(drop=True)
    elif analysisType is AnalysisType.FOURIER:
        print("performing fourier transformation")
        return getFFT(dfvp)

In [670]: from enum import Enum, auto
    class AnalysisType(Enum):
        TIME_SERIES = auto()
        FOURIER = auto()

In [671]: ##plot percentage of variance vs pcs
    def plotPercentVariance(dfvp):
        pcInpt = performPca(dfvp.T, 10, False)
        pcInpt.apply(perVar, axis=1)

    def perVar(df_var):
        df_var = abs(df_var)
        print("abs",df_var)
        sum = df_var.sum()
        res = [x*100/sum for x in df_var]

```

```

dfPlot = pd.DataFrame(data=pd.DataFrame(list(zip(df_var.index, res))), columns = ['
sns.barplot(x='pc', y = 'perVariance', data=dfPlot)
plt.show()
#plotPercentVariance(df_filteredVp)

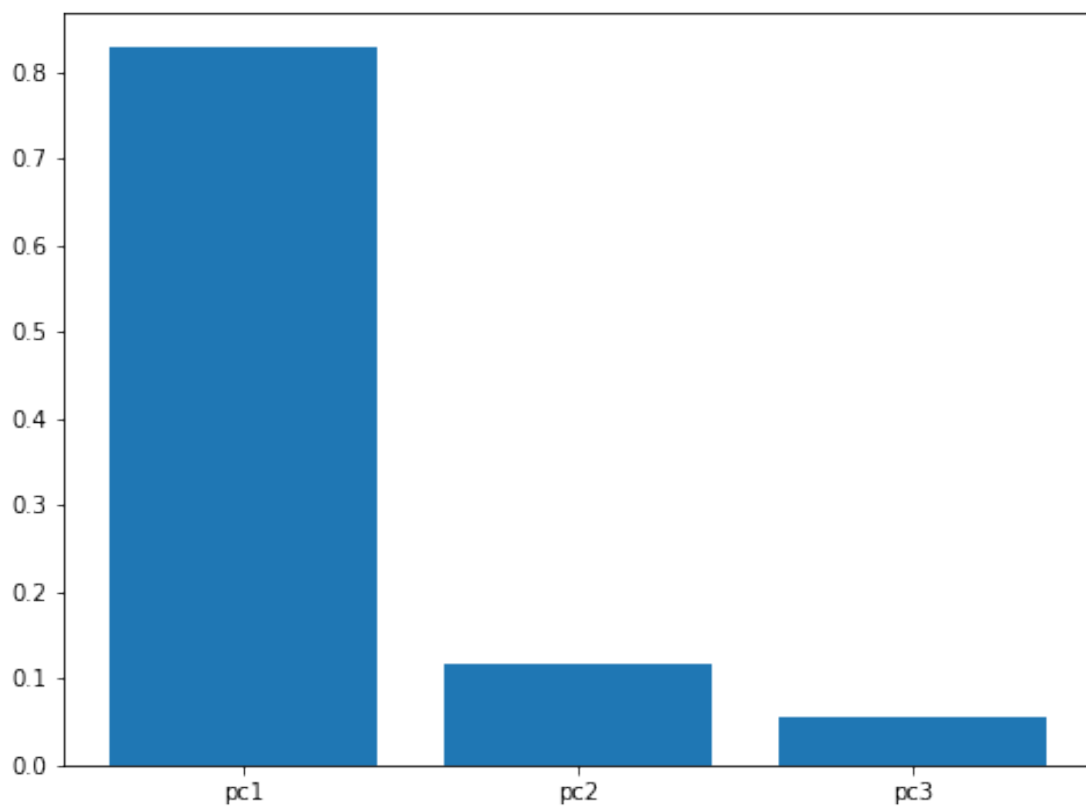
```

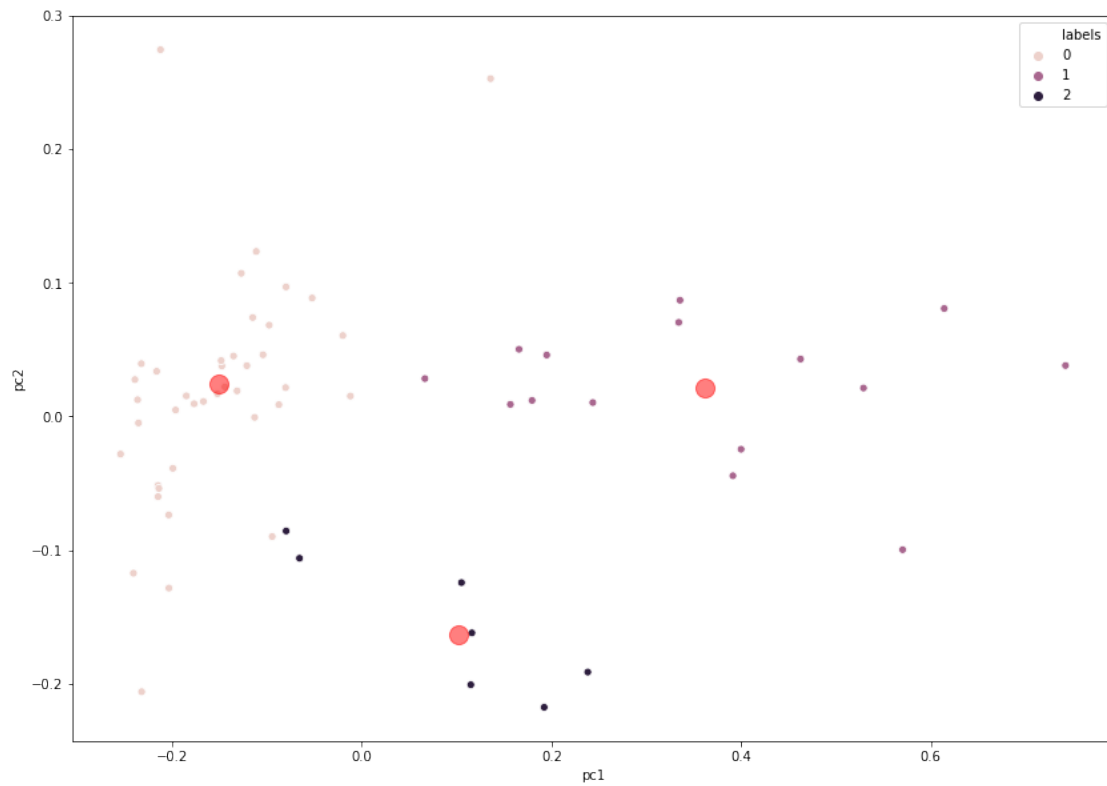
In [673]:

```

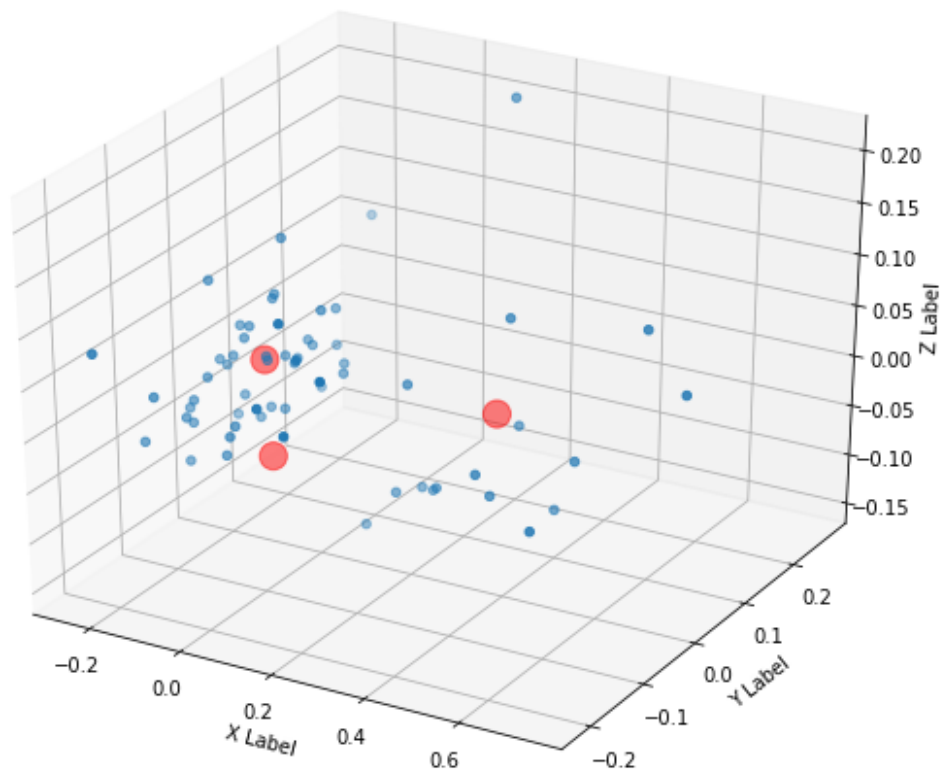
Radius in m 600
(288, 204)
(384, 204)
(672, 204)
Index: 49
Dimension points (629, 62)
performing fourier transformation
Performing PCA
Explained variance ratio with PCA: [ 0.82766017  0.11786175  0.05447807]
Cluster labels: [1 0 0 0 0 1 1 1 0 0 0 0 0 0 1 0 1 1 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
 2 2 0 0 0 0 0 2 0 0 0 2 0 1 2 1 1 2 0 1 0 2 0 0 0]
strength: 0.956076071291
Performing PCA
Explained variance ratio with PCA: [ 0.82766017  0.11786175]
Performing PCA
Explained variance ratio with PCA: [ 0.82766017  0.11786175]
cluster: 0
-----
Number of meters in each cluster: A:1 B:30 C:9
-----
cluster: 1
-----
Number of meters in each cluster: A:0 B:0 C:15
-----
cluster: 2
-----
Number of meters in each cluster: A:0 B:5 C:2
-----
Overall accuracy: 0.806451612903
-----
Performing PCA
Explained variance ratio with PCA: [ 0.82766017  0.11786175  0.05447807]

```





<matplotlib.figure.Figure at 0x7f0e518e7ef0>



```
In [674]: from scipy.spatial.distance import pdist, squareform
          df_latlong, df_voltPoints, df_trMapping = processInputData()

          distances = pdist(df_latlong.iloc[:,1:3].values, metric='euclidean')
          #print(len(distances))
          dist_matrix = squareform(distances)
          print(dist_matrix)

(288, 204)
(384, 204)
(672, 204)
[[ 0.          0.00234874  0.00234874 ...,  0.00234874  0.00234874
   0.01240292]
 [ 0.00234874  0.          0.          ...,  0.          0.          0.0104154 ]
 [ 0.00234874  0.          0.          ...,  0.          0.          0.0104154 ]
 ...,
 [ 0.00234874  0.          0.          ...,  0.          0.          0.0104154 ]
 [ 0.00234874  0.          0.          ...,  0.          0.          0.0104154 ]
 [ 0.01240292  0.0104154  0.0104154 ...,  0.0104154  0.0104154  0.          ]]
```

```
In [676]: df_meterDist = pd.DataFrame(list(zip(df_latlong['_id'], dist_matrix[idx])), columns=['_id', 'distances'])
df_meterDistSorted = df_meterDist.sort_values(by='distances').reset_index(drop=True)
```

1.4 DBSCAN analysis

```
In [ ]: nbrs = NearestNeighbors(n_neighbors=3).fit(dfxt.values)
distances, indices = nbrs.kneighbors(dfxt)
dists = np.sort(distances.flatten())
plt.plot(dists)
```

```
In [ ]: #DBSCAN Clustering
```

```
dfInp = performPca(dfxt)
clustering_dbscan = DBSCAN(eps=0.005, min_samples=2, metric=voltCorr).fit(dfxt)
labels_dbscan = clustering_dbscan.labels_
showClusters(dfInp, labels_dbscan)
```

2 Results:

```
cluster: 0 ['Meter 2' 'Meter 14' 'Meter 15' 'Meter 24'] cluster: 1 ['Meter 6' 'Meter 7' 'Meter 10' 'Meter 17' 'Meter 20' 'Meter 27'] cluster: 2 ['Meter 9' 'Meter 11']
```