In [12]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

import psycopg2
from psycopg2 import extras
from pandas import DataFrame
```

Импортируем подготовленные данные из файла datasets_66163_130012_Churn_Modelling (отток клиентов из бизнеса):

In [13]:

```
# Importing the dataset
df = pd.read_csv('dataset.csv')
```

In [14]:

```
df
```

Out[14]:

	Exited	Tenure	NumOfProducts	Age	EstimatedSalary
0	1	2	1	42	101348.88
1	0	1	1	41	112542.58
2	1	8	3	42	113931.57
3	0	1	2	39	93826.63
4	0	2	1	43	79084.10
...	...	...	...	...	...
9995	0	5	2	39	96270.64
9996	0	10	1	35	101699.77
9997	1	7	1	36	42085.58
9998	1	3	2	42	92888.52
9999	0	4	1	28	38190.78

10000 rows × 5 columns

In [15]:

```
# Logistic Regression
```

Для логистической регрессии в качестве переменной будет использовать Exited, так как она принимает значение только 0 и 1.

In [16]:

```
# Splitting the dataset into the Training set and Test set
X = df.iloc[:, 1:].values
y = df.iloc[:, 0].values
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=13)
```

In [17]:

```
# Feature Scaling
from sklearn.preprocessing import MinMaxScaler
sc_X = MinMaxScaler().fit(X_train)
X_train = sc_X.transform(X_train)
X_test = sc_X.transform(X_test)
```

In [18]:

```
# Определим значимые переменные для наших будущих моделей
```

In [19]:

```
# Baseline model
import statsmodels.api as sm
lr = sm.Logit(y_train, X_train).fit()
print(lr.summary2())
```

```
Optimization terminated successfully.
         Current function value: 0.488783
         Iterations 6
                         Results: Logit
=================================================================
Model:              Logit            Pseudo R-squared: 0.034
Dependent Variable: y                AIC:              7828.5348
Date:               2020-11-06 13:45 BIC:              7856.4835
No. Observations:   8000             Log-Likelihood:   -3910.3
Df Model:           3                LL-Null:          -4047.2
Df Residuals:       7996             LLR p-value:      4.5778e-59
Converged:          1.0000           Scale:            1.0000
No. Iterations:     6.0000
-------------------------------------------------------------------
       Coef.     Std.Err.       z       P>|z|     [0.025     0.975]
-------------------------------------------------------------------
x1     -0.6282     0.0952    -6.6017   0.0000    -0.8147    -0.4417
x2     -0.7076     0.1459    -4.8493   0.0000    -0.9936    -0.4216
x3      3.4962     0.1814    19.2702   0.0000     3.1406     3.8518
x4    -16.1256     0.6515   -24.7499   0.0000   -17.4026   -14.8486
=================================================================
```

In [20]:

```
# Будем считать значимыми те переменные, p-value которых не превышает 1%. Как мы видим, та
ковыми являются абсолютно все переменные.
```

In [21]:

```
# Fitting Logistic Regression to the Training set
from sklearn.linear_model import LogisticRegression
slr = LogisticRegression(random_state = 13).fit(X_train, y_train)
```

In [22]:

```
# Predicting the Test set results
y_pred = slr.predict(X_test)
slr.score(X_test,y_test)
```

Out[22]:

```
0.7715
```

In [23]:

```
# Making the Confusion Matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
print(cm)
```

```
[[1537   58]
 [ 399    6]]
```

In [24]:

```
# Вывод: точность данной модели = 0,77, что является неплохим показателем. Так же исходя и
з матрицы, имеем 457 ложных предсказаний от нашей модели.
```

K-Nearest Neighbors

In [25]:

```
# Cheking correlations
correlation = df.corr()
correlation.style.background_gradient(cmap='coolwarm')
```

Out[25]:

	Exited	Tenure	NumOfProducts	Age	EstimatedSalary
Exited	1.000000	-0.014001	-0.047820	0.285323	-0.001415
Tenure	-0.014001	1.000000	0.013444	-0.009997	-0.017662
NumOfProducts	-0.047820	0.013444	1.000000	-0.030680	0.011089
Age	0.285323	-0.009997	-0.030680	1.000000	-0.010690
EstimatedSalary	-0.001415	-0.017662	0.011089	-0.010690	1.000000

Только лишь переменная Age имеет видимую корреляцию с Exited. Но для построения модели попробуем использовать все переменные.

Для построения этой модели нам первостепенно важно определить k.

In [26]:

```
# Optimal K
from sklearn.neighbors import KNeighborsClassifier
error_rate = []
for i in range(1,50):
     knn = KNeighborsClassifier(n_neighbors=i)
     knn.fit(X_train,y_train)
     pred_i = knn.predict(X_test)
     error_rate.append(np.mean(pred_i != y_test))

plt.figure(figsize=(10,6))
plt.plot(range(1,50),error_rate,color='blue', linestyle='dashed',
         marker='o',markerfacecolor='red', markersize=10)
plt.title('Error Rate vs. K Value')
plt.xlabel('K')
plt.ylabel('Error Rate')
print("Minimum error:-",min(error_rate),"at K =",error_rate.index(min(error_rate)))
```

```
Minimum error:- 0.155 at K = 42
```

Выберем значение k = 42

In [27]:

```
# Fitting K-NN to the Training set
knn = KNeighborsClassifier(n_neighbors = 42, metric = 'minkowski', p = 2).fit(X_train, y_t
rain)

y_pred = knn.predict(X_test)
knn.score(X_test,y_test)
```

Out[27]:

```
0.8425
```

In [28]:

```
# Making the Confusion Matrix
from pprint import pprint
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
cm
```

Out[28]:

```
array([[1526,   69],
       [ 246,  159]], dtype=int64)
```

In [29]:

```
# Вывод: точность данной модели = 0,8425, что выше, чем у модели Логистической регрессии.
```

Support Vector Machine

In [30]:

```
from sklearn.svm import SVC
svm = SVC(kernel = 'rbf', random_state = 10).fit(X_train, y_train)


# Predicting the Test set results
y_pred = svm.predict(X_test)
svm.score(X_test,y_test)
```

Out[30]:

```
0.826
```

In [31]:

```
# Making the Confusion Matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
cm
```

Out[31]:

```
array([[1556,   39],
       [ 309,   96]], dtype=int64)
```

Вывод: точность данной модели = 0.826, что является отличным показателем.

Naive Bayes

In [32]:

```
# Fitting Naive Bayes to the Training set (2 variables)
from sklearn.naive_bayes import GaussianNB
nb = GaussianNB().fit(X_train, y_train)

y_pred = nb.predict(X_test)
nb.score(X_test,y_test)
```

Out[32]:

```
0.7905
```

In [33]:

```
# Making the Confusion Matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
cm
```

Out[33]:

```
array([[1528,   67],
       [ 352,   53]], dtype=int64)
```

Вывод: точность данной модели = 0.79, что является отличным показателем.

Classification Tree

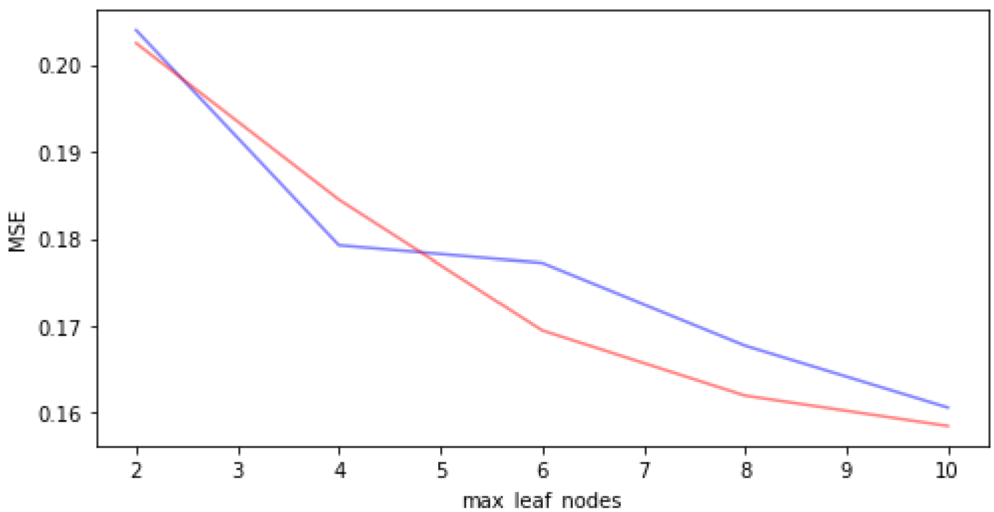
Деревья очень часто имеют эффект переобучения, поэтому нам необходимо определить размер дерева.

In [34]:

```
def max_leaf_nodes(X_train, X_test, y_train, y_test, n):
    mse_train = []
    mse_test = []
    for i in n:
        rf = DecisionTreeClassifier(max_leaf_nodes = i, random_state=10).fit(X_train, y_tr
ain)
        mse_train.append(mean_squared_error(y_train, rf.predict(X_train)))
        mse_test.append(mean_squared_error(y_test, rf.predict(X_test)))
    fig, ax = plt.subplots(figsize=(8, 4))
    ax.plot(n, mse_train, alpha=0.5, color='blue', label='train')
    ax.plot(n, mse_test, alpha=0.5, color='red', label='test')
    ax.set_ylabel("MSE")
    ax.set_xlabel("max_leaf_nodes")
```

In [35]:

```
# The optimal number of max_leaf_nodes
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import mean_squared_error
max_leaf_nodes(X_train, X_test, y_train, y_test, [2, 4, 6, 8, 10])
```



Выбор оптимальной точки: выбираем размер дерева = 5.

In [36]:

```
# Fitting Classification Tree to the Training set
ct = DecisionTreeClassifier(max_leaf_nodes = 5, criterion = 'entropy', random_state = 10).
fit(X_train, y_train)

# Predicting the Test set results
y_pred = ct.predict(X_test)
ct.score(X_test,y_test)
```

Out[36]:

```
0.8155
```

In [37]:

```
# Making the Confusion Matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
cm
```

Out[37]:

```
array([[1584,   11],
       [ 358,   47]], dtype=int64)
```

Вывод: точность данной модели = 0.8155, что является хорошим показателем, но есть модели, показавшие себя незначительно лучше.

NN Classification

In [38]:

```
import keras
from keras.models import Sequential
from keras.layers import Dense
```

In [39]:

```
# Initialising the ANN 5-4-1
cnn = Sequential()

# Adding the input layer and the first hidden layer
cnn.add(Dense(units = 5, kernel_initializer = 'uniform', activation = 'relu', input_dim =
4))

# Adding the output layer
cnn.add(Dense(units = 1, kernel_initializer = 'uniform', activation = 'sigmoid'))

# Compiling the ANN
cnn.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics = ['accuracy'])
```

In [40]:

```
# Fitting the ANN to the Training set
cnn.fit(X_train, y_train, batch_size = 10, epochs = 100)
```

```
Epoch 1/100
800/800 [==============================] - 1s 917us/step - loss: 0.5483 - acc
uracy: 0.7956
Epoch 2/100
800/800 [==============================] - 1s 879us/step - loss: 0.5043 - acc
uracy: 0.7960
Epoch 3/100
800/800 [==============================] - 1s 953us/step - loss: 0.4941 - acc
uracy: 0.7960
Epoch 4/100
800/800 [==============================] - 1s 1ms/step - loss: 0.4832 - accur
acy: 0.7960
Epoch 5/100
800/800 [==============================] - 1s 1ms/step - loss: 0.4711 - accur
acy: 0.7960
Epoch 6/100
800/800 [==============================] - 1s 962us/step - loss: 0.4589 - acc
uracy: 0.7979
Epoch 7/100
800/800 [==============================] - 1s 955us/step - loss: 0.4460 - acc
uracy: 0.8016
Epoch 8/100
800/800 [==============================] - 1s 913us/step - loss: 0.4331 - acc
uracy: 0.8058
Epoch 9/100
800/800 [==============================] - 1s 871us/step - loss: 0.4227 - acc
uracy: 0.8096
Epoch 10/100
800/800 [==============================] - 1s 945us/step - loss: 0.4149 - acc
uracy: 0.8129
Epoch 11/100
800/800 [==============================] - 1s 1ms/step - loss: 0.4090 - accur
acy: 0.8151
Epoch 12/100
800/800 [==============================] - 1s 1ms/step - loss: 0.4050 - accur
acy: 0.8174
Epoch 13/100
800/800 [==============================] - 1s 1ms/step - loss: 0.4019 - accur
acy: 0.8184
Epoch 14/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3997 - accur
acy: 0.8191
Epoch 15/100
800/800 [==============================] - 1s 982us/step - loss: 0.3979 - acc
uracy: 0.8195
Epoch 16/100
800/800 [==============================] - 1s 2ms/step - loss: 0.3964 - accur
acy: 0.8200
Epoch 17/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3955 - accur
acy: 0.8201
Epoch 18/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3944 - accur
acy: 0.8204
Epoch 19/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3939 - accur
acy: 0.8204
```

```
Epoch 20/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3934 - accur
acy: 0.8202
Epoch 21/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3932 - accur
acy: 0.8204
Epoch 22/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3926 - accur
acy: 0.8204
Epoch 23/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3926 - accur
acy: 0.8202
Epoch 24/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3923 - accur
acy: 0.8202
Epoch 25/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3921 - accur
acy: 0.8205
Epoch 26/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3919 - accur
acy: 0.8205
Epoch 27/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3917 - accur
acy: 0.8206
Epoch 28/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3916 - accur
acy: 0.8204
Epoch 29/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3915 - accur
acy: 0.8205
Epoch 30/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3914 - accur
acy: 0.8205
Epoch 31/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3914 - accur
acy: 0.8204
Epoch 32/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3911 - accur
acy: 0.8209
Epoch 33/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3910 - accur
acy: 0.8204
Epoch 34/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3910 - accur
acy: 0.8206
Epoch 35/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3908 - accur
acy: 0.8202
Epoch 36/100
800/800 [==============================] - 1s 984us/step - loss: 0.3907 - acc
uracy: 0.8205
Epoch 37/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3905 - accur
acy: 0.8206
Epoch 38/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3905 - accur
acy: 0.8205
```

```
Epoch 39/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3902 - accur
acy: 0.8204
Epoch 40/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3903 - accur
acy: 0.8205
Epoch 41/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3900 - accur
acy: 0.8205
Epoch 42/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3897 - accur
acy: 0.8206
Epoch 43/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3898 - accur
acy: 0.8206
Epoch 44/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3894 - accur
acy: 0.8206
Epoch 45/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3892 - accur
acy: 0.8206
Epoch 46/100
800/800 [==============================] - 1s 969us/step - loss: 0.3892 - acc
uracy: 0.8205
Epoch 47/100
800/800 [==============================] - 1s 960us/step - loss: 0.3889 - acc
uracy: 0.8205
Epoch 48/100
800/800 [==============================] - 1s 914us/step - loss: 0.3890 - acc
uracy: 0.8207
Epoch 49/100
800/800 [==============================] - 1s 848us/step - loss: 0.3884 - acc
uracy: 0.8206
Epoch 50/100
800/800 [==============================] - 1s 974us/step - loss: 0.3885 - acc
uracy: 0.8211
Epoch 51/100
800/800 [==============================] - 1s 975us/step - loss: 0.3883 - acc
uracy: 0.8180
Epoch 52/100
800/800 [==============================] - 1s 931us/step - loss: 0.3884 - acc
uracy: 0.8181
Epoch 53/100
800/800 [==============================] - 1s 918us/step - loss: 0.3883 - acc
uracy: 0.8225
Epoch 54/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3882 - accur
acy: 0.8213
Epoch 55/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3881 - accur
acy: 0.8220
Epoch 56/100
800/800 [==============================] - 1s 935us/step - loss: 0.3877 - acc
uracy: 0.8219
Epoch 57/100
800/800 [==============================] - 1s 951us/step - loss: 0.3877 - acc
uracy: 0.8235
```

```
Epoch 58/100
800/800 [==============================] - 1s 977us/step - loss: 0.3877 - acc
uracy: 0.8230
Epoch 59/100
800/800 [==============================] - 1s 926us/step - loss: 0.3876 - acc
uracy: 0.8223
Epoch 60/100
800/800 [==============================] - 1s 964us/step - loss: 0.3878 - acc
uracy: 0.8255
Epoch 61/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3875 - accur
acy: 0.8245
Epoch 62/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3873 - accur
acy: 0.8249
Epoch 63/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3873 - accur
acy: 0.8256
Epoch 64/100
800/800 [==============================] - 1s 931us/step - loss: 0.3874 - acc
uracy: 0.8275
Epoch 65/100
800/800 [==============================] - 1s 966us/step - loss: 0.3871 - acc
uracy: 0.8244
Epoch 66/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3868 - accur
acy: 0.8273
Epoch 67/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3873 - accur
acy: 0.8261
Epoch 68/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3872 - accur
acy: 0.8278
Epoch 69/100
800/800 [==============================] - 1s 965us/step - loss: 0.3870 - acc
uracy: 0.8260
Epoch 70/100
800/800 [==============================] - 1s 968us/step - loss: 0.3870 - acc
uracy: 0.8264
Epoch 71/100
800/800 [==============================] - 1s 952us/step - loss: 0.3870 - acc
uracy: 0.8273
Epoch 72/100
800/800 [==============================] - 1s 946us/step - loss: 0.3869 - acc
uracy: 0.8270
Epoch 73/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3868 - accur
acy: 0.8266
Epoch 74/100
800/800 [==============================] - 1s 920us/step - loss: 0.3868 - acc
uracy: 0.8269
Epoch 75/100
800/800 [==============================] - 1s 986us/step - loss: 0.3865 - acc
uracy: 0.8270
Epoch 76/100
800/800 [==============================] - 1s 998us/step - loss: 0.3869 - acc
uracy: 0.8267
```

```
Epoch 77/100
800/800 [==============================] - 1s 895us/step - loss: 0.3867 - acc
uracy: 0.8279
Epoch 78/100
800/800 [==============================] - 1s 909us/step - loss: 0.3866 - acc
uracy: 0.8274
Epoch 79/100
800/800 [==============================] - 1s 960us/step - loss: 0.3869 - acc
uracy: 0.8279
Epoch 80/100
800/800 [==============================] - 1s 875us/step - loss: 0.3865 - acc
uracy: 0.8279
Epoch 81/100
800/800 [==============================] - 1s 933us/step - loss: 0.3869 - acc
uracy: 0.8259
Epoch 82/100
800/800 [==============================] - 1s 959us/step - loss: 0.3867 - acc
uracy: 0.8294
Epoch 83/100
800/800 [==============================] - 1s 921us/step - loss: 0.3866 - acc
uracy: 0.8278
Epoch 84/100
800/800 [==============================] - 1s 989us/step - loss: 0.3868 - acc
uracy: 0.8282
Epoch 85/100
800/800 [==============================] - 1s 979us/step - loss: 0.3863 - acc
uracy: 0.8288
Epoch 86/100
800/800 [==============================] - 1s 977us/step - loss: 0.3864 - acc
uracy: 0.8266
Epoch 87/100
800/800 [==============================] - 1s 1ms/step - loss: 0.3865 - accur
acy: 0.8278
Epoch 88/100
800/800 [==============================] - 1s 987us/step - loss: 0.3866 - acc
uracy: 0.8284
Epoch 89/100
800/800 [==============================] - 1s 976us/step - loss: 0.3865 - acc
uracy: 0.8280
Epoch 90/100
800/800 [==============================] - 1s 929us/step - loss: 0.3860 - acc
uracy: 0.8276
Epoch 91/100
800/800 [==============================] - 1s 910us/step - loss: 0.3868 - acc
uracy: 0.8274
Epoch 92/100
800/800 [==============================] - 1s 975us/step - loss: 0.3864 - acc
uracy: 0.8278
Epoch 93/100
800/800 [==============================] - 1s 923us/step - loss: 0.3864 - acc
uracy: 0.8273
Epoch 94/100
800/800 [==============================] - 1s 975us/step - loss: 0.3864 - acc
uracy: 0.8271
Epoch 95/100
800/800 [==============================] - 1s 925us/step - loss: 0.3864 - acc
uracy: 0.8281
```

```
Epoch 96/100
800/800 [==============================] - 1s 976us/step - loss: 0.3866 - acc
uracy: 0.8274
Epoch 97/100
800/800 [==============================] - 1s 986us/step - loss: 0.3865 - acc
uracy: 0.8290
Epoch 98/100
800/800 [==============================] - 1s 995us/step - loss: 0.3866 - acc
uracy: 0.8274
Epoch 99/100
800/800 [==============================] - 1s 974us/step - loss: 0.3864 - acc
uracy: 0.8274
Epoch 100/100
800/800 [==============================] - 1s 905us/step - loss: 0.3865 - acc
uracy: 0.8275
```

Out[40]:

```
<tensorflow.python.keras.callbacks.History at 0x1d728ff52b0>
```

In [41]:

```
# Predicting the Test set results
y_pred = cnn.predict(X_test)
y_pred = (y_pred > 0.5)
```

In [42]:

```
# Making the Confusion Matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
cm
```

Out[42]:

```
array([[1512,   83],
       [ 242,  163]], dtype=int64)
```

Hierarchical Clustering

Для того что бы уровнять наши переменные и ни один из параметров не перетягивал все на себя, проведет шкалирование.

In [43]:

```
# Feature Scaling
from sklearn.preprocessing import MinMaxScaler
sc = MinMaxScaler().fit(df)
df = sc.transform(df)
df = pd.DataFrame(df, columns = ['Exited','Tenure','NumOfProducts','Age','EstimatedSalary'
]).round(2)
```

In [44]:

```
df
```

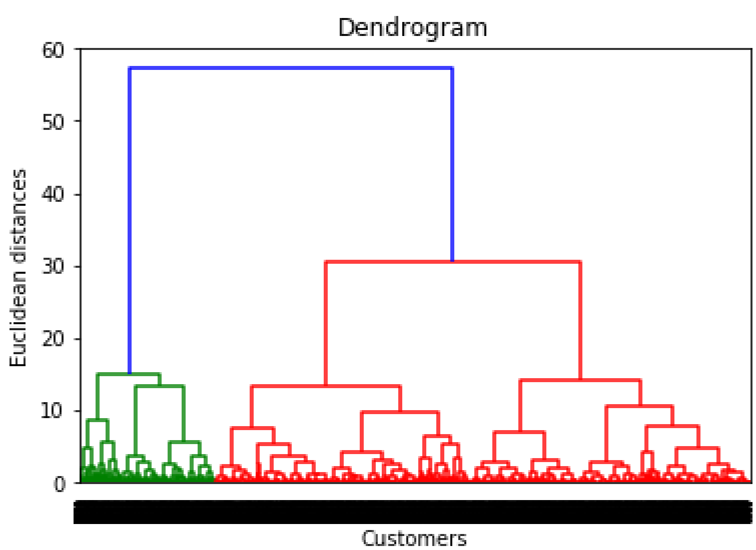
Out[44]:

	Exited	Tenure	NumOfProducts	Age	EstimatedSalary
0	1.0	0.2	0.00	0.32	0.12
1	0.0	0.1	0.00	0.31	0.12
2	1.0	0.8	0.67	0.32	0.12
3	0.0	0.1	0.33	0.28	0.12
4	0.0	0.2	0.00	0.34	0.11
...	...	...	...	...	...
9995	0.0	0.5	0.33	0.28	0.12
9996	0.0	1.0	0.00	0.23	0.12
9997	1.0	0.7	0.00	0.24	0.10
9998	1.0	0.3	0.33	0.32	0.12
9999	0.0	0.4	0.00	0.14	0.10

10000 rows × 5 columns

In [53]:

```
# Using the dendrogram to find the optimal number of clusters
import scipy.cluster.hierarchy as sch
dendrogram = sch.dendrogram(sch.linkage(df, method = 'ward'))
plt.title('Dendrogram')
plt.xlabel('Customers')
plt.ylabel('Euclidean distances')
plt.show()
```



На дендограме сверху мы изобразили абсолютно все наблюдения. Исходя из длины полученных веток, имеем: оптимальнее всего рассматривать 3 кластера.

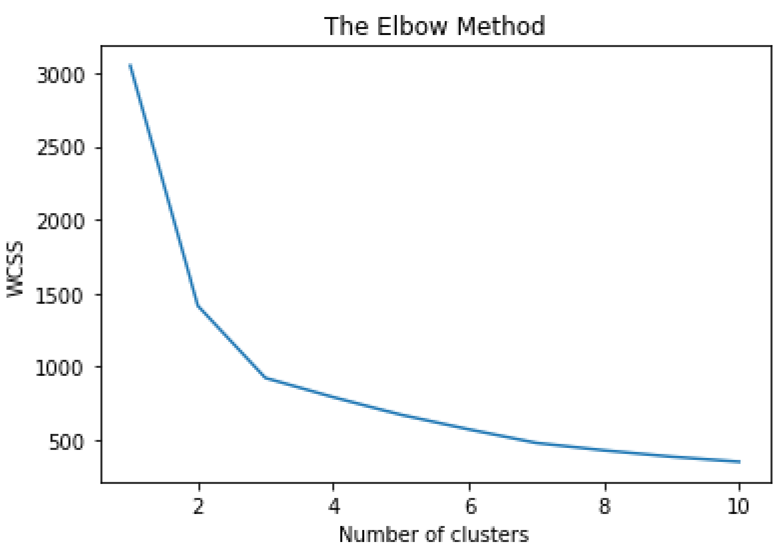
In [54]:

```
# Fitting Hierarchical Clustering to the dataset
from sklearn.cluster import AgglomerativeClustering
hc = AgglomerativeClustering(n_clusters = 3, affinity = 'euclidean', linkage = 'ward').fit
_predict(df)
```

K-Means Clustering

In [55]:

```
# Using the elbow method to find the optimal number of clusters
from sklearn.cluster import KMeans
wcss = []
for i in range(1, 11):
    kmeans = KMeans(n_clusters = i, init = 'k-means++', random_state = 0)
    kmeans.fit(df)
    wcss.append(kmeans.inertia_)
plt.plot(range(1, 11), wcss)
plt.title('The Elbow Method')
plt.xlabel('Number of clusters')
plt.ylabel('WCSS')
plt.show()
```



Имеем: данная прямая резко уходит вниз с 2.

In [56]:

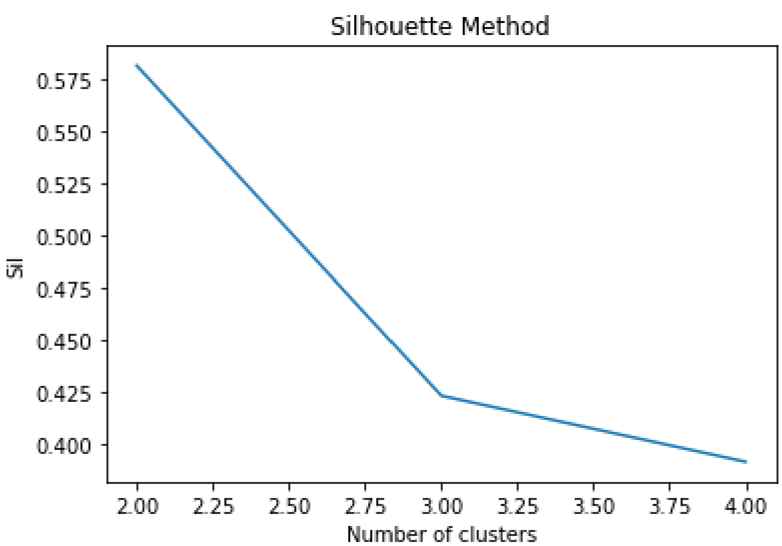
```
# Using the silhouette method to find the optimal number of clusters
from sklearn.metrics import silhouette_score

sil = []

for k in range(2, 5):
  kmeans = KMeans(n_clusters = k).fit(df)
  preds = kmeans.fit_predict(df)
  sil.append(silhouette_score(df, preds, metric = 'euclidean'))


plt.plot(range(2, 5), sil)
plt.title('Silhouette Method')
plt.xlabel('Number of clusters')
plt.ylabel('Sil')
plt.show()

for i in range(len(sil)):
 print(str(i+2) +":"+ str(sil[i]))
```




```
2:0.5815358991498022
3:0.4230930349693686
4:0.39136753018468323
```

Остановимся на значении 4.

In [57]:

```
# Fitting K-Means to the dataset
km = KMeans(n_clusters = 4, init = 'k-means++', random_state = 0).fit_predict(df)
kms = KMeans(n_clusters = 4, random_state = 0).fit(df)
pd.DataFrame(kms.cluster_centers_, columns = ['Exited','Tenure','NumOfProducts','Age','Est
imatedSalary']).round(decimals=1)
```

Out[57]:

	Exited	Tenure	NumOfProducts	Age	EstimatedSalary
0	1.0	0.3	0.1	0.4	0.1
1	0.0	0.3	0.2	0.3	0.1
2	0.0	0.8	0.2	0.3	0.1
3	1.0	0.8	0.2	0.4	0.1

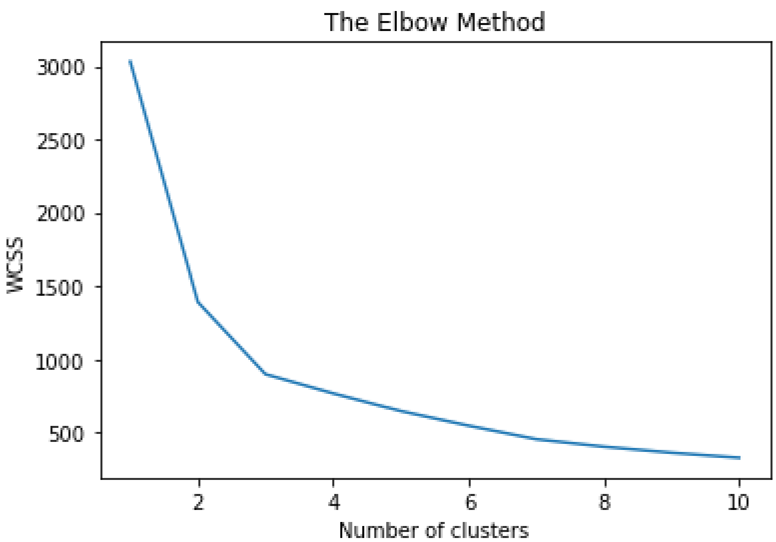
Как мы видим, EstimatedSalary совершенно не влияет на результаты кластеризации - избавимся от этой переменной.

In [58]:

```
# Less features
X = df.iloc[:, [0, 1, 2, 3]]
```

In [59]:

```
# Using the elbow method to find the optimal number of clusters
from sklearn.cluster import KMeans
wcss = []
for i in range(1, 11):
    kmeans = KMeans(n_clusters = i, init = 'k-means++', random_state = 0)
    kmeans.fit(X)
    wcss.append(kmeans.inertia_)
plt.plot(range(1, 11), wcss)
plt.title('The Elbow Method')
plt.xlabel('Number of clusters')
plt.ylabel('WCSS')
plt.show()
```



In [60]:

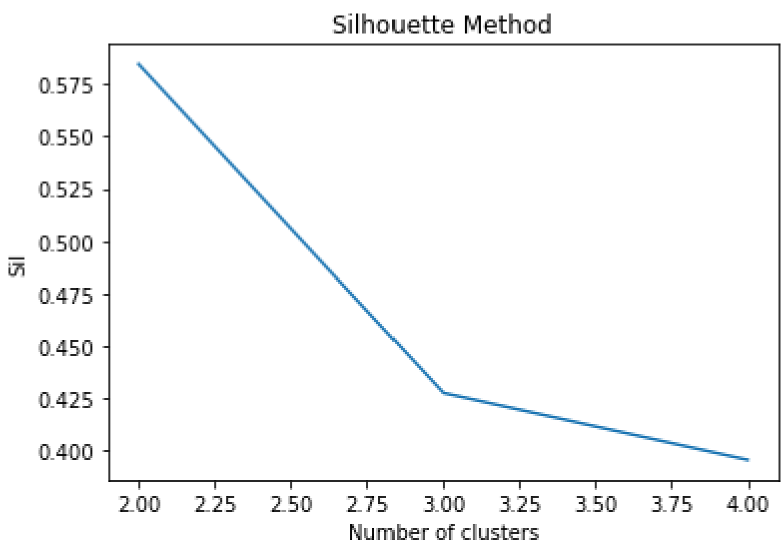
```
# Using the silhouette method to find the optimal number of clusters
from sklearn.metrics import silhouette_score

sil = []

for k in range(2, 5):
  kmeans = KMeans(n_clusters = k).fit(X)
  preds = kmeans.fit_predict(X)
  sil.append(silhouette_score(X, preds, metric = 'euclidean'))


plt.plot(range(2, 5), sil)
plt.title('Silhouette Method')
plt.xlabel('Number of clusters')
plt.ylabel('Sil')
plt.show()

for i in range(len(sil)):
 print(str(i+2) +":"+ str(sil[i]))
```




```
2:0.5844593933014209
3:0.42762032406129274
4:0.3957162178156961
```

Остановимся на значении 4.

In [61]:

```
# Fitting K-Means to the dataset
km = KMeans(n_clusters = 4, init = 'k-means++', random_state = 0).fit_predict(X)
kms = KMeans(n_clusters = 4, random_state = 0).fit(X)
pd.DataFrame(kms.cluster_centers_, columns = ['Exited','Tenure','NumOfProducts','Age']).round(decimals=1)
```

Out[61]:

	Exited	Tenure	NumOfProducts	Age
0	0.0	0.8	0.2	0.3
1	1.0	0.3	0.1	0.4
2	1.0	0.8	0.2	0.4
3	0.0	0.3	0.2	0.3

Вывод: Люди постарше, чаще всего уходят из сайта ничего не купив.

SOM

In [62]:

```
# Importing the dataset
df = pd.read_csv('dataset.csv')
```

Для начала мы перезаписали заново файл с данными и приняли все подготовительные меры.

In [63]:

```
# Feature Scaling
import SimpSOM as sps
from sklearn.preprocessing import StandardScaler
sc = StandardScaler().fit(df)
df = sc.transform(df)
```

In [64]:

```
X = df[:, [0, 1, 2, 3]]
```

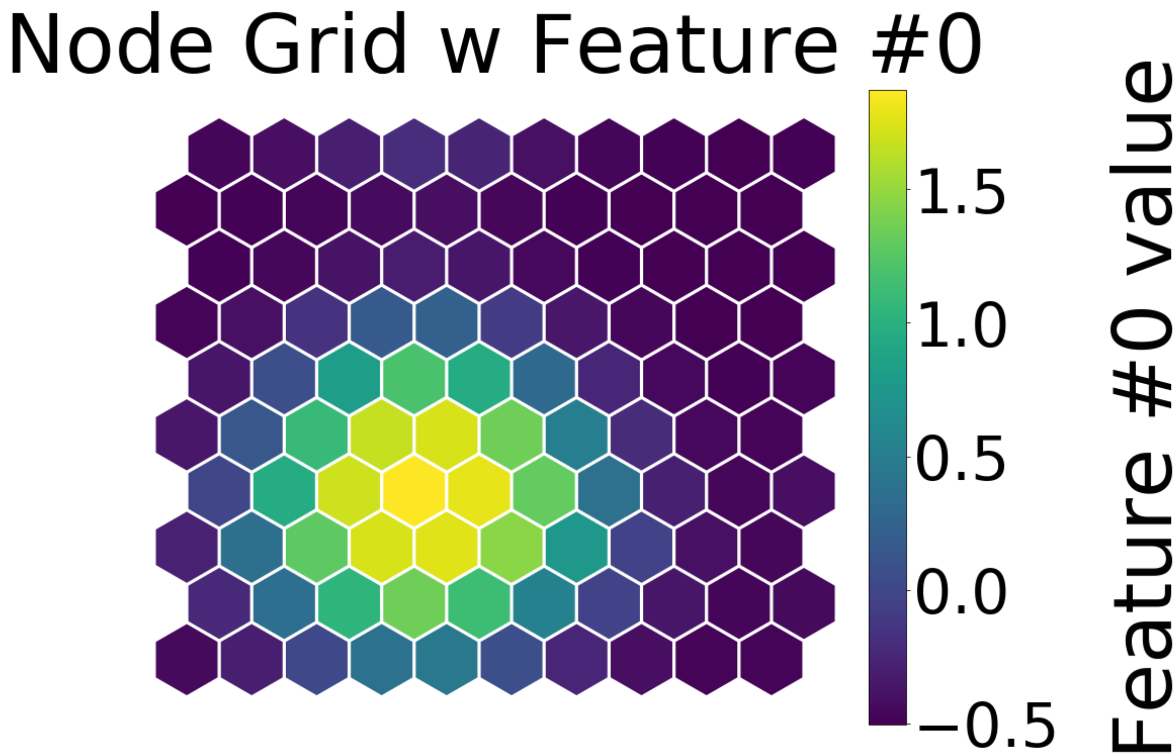
In [65]:

```
np.random.seed(605891282)
net = sps.somNet(10, 10, X, PBC=True)
net.train(0.01, 20000)
#net.save('filename_weights')
```

```
Periodic Boundary Conditions active.
The weights will be initialised randomly.
Training SOM... done!
```

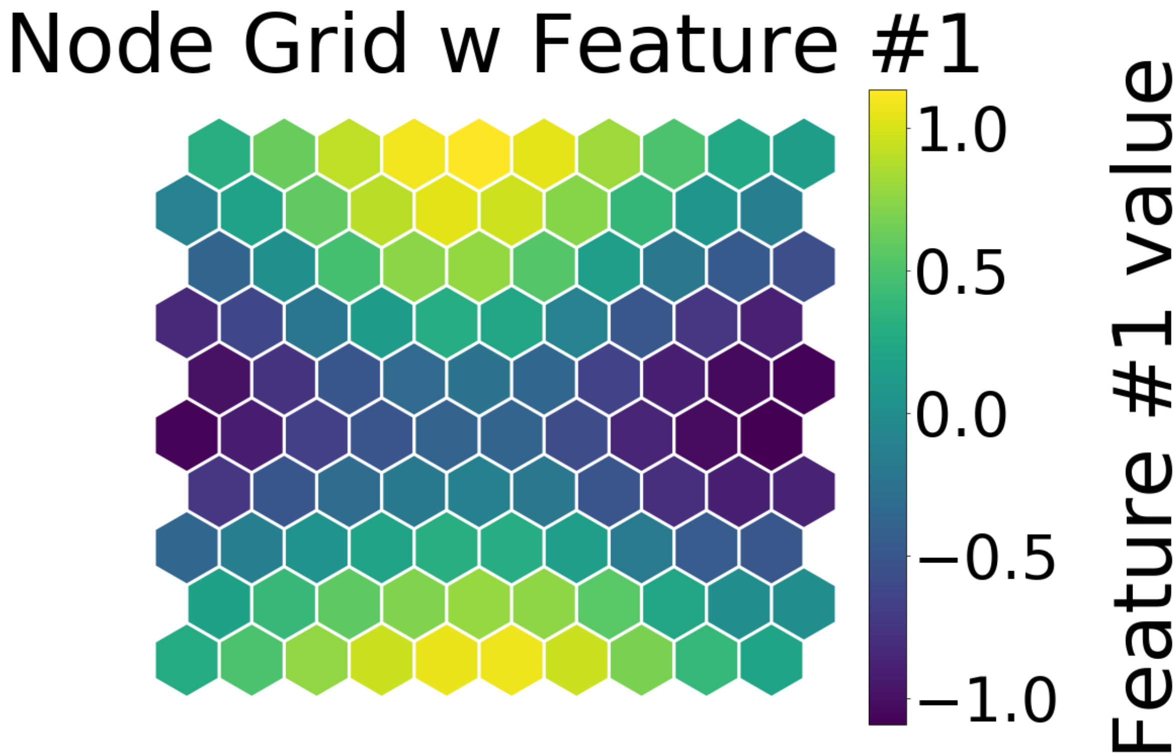
In [66]:

```
net.nodes_graph(colnum=0)
```



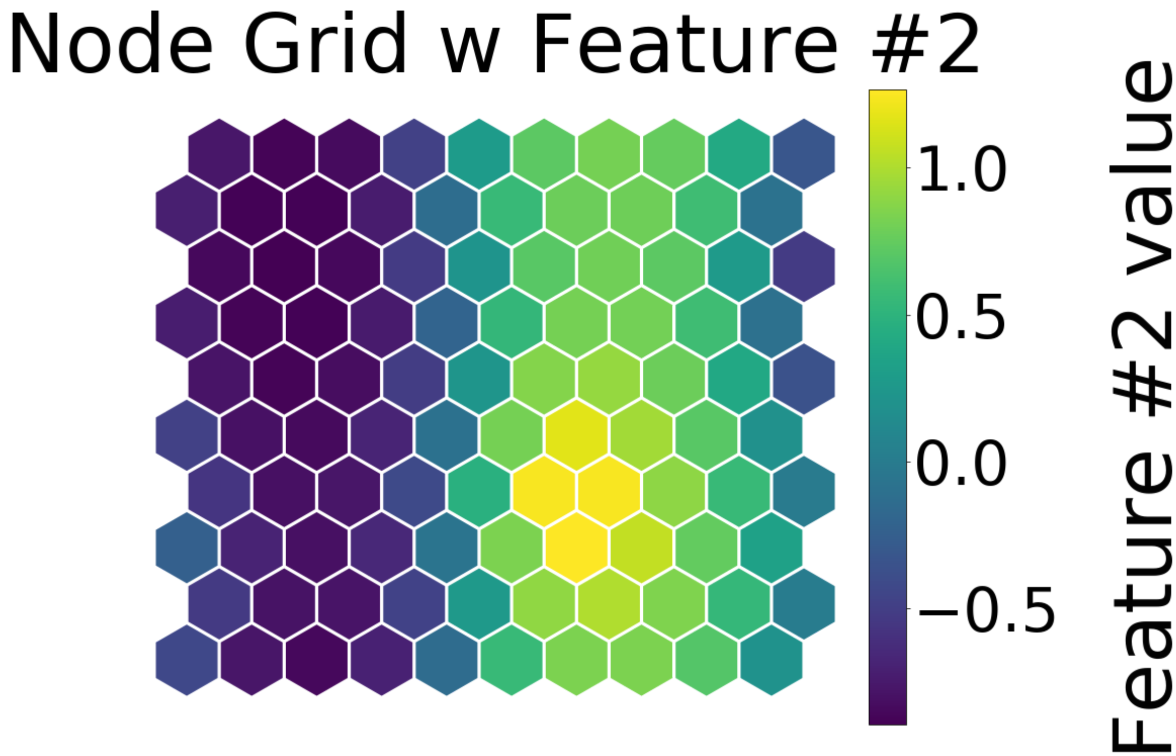
In [67]:

```
net.nodes_graph(colnum=1)
```



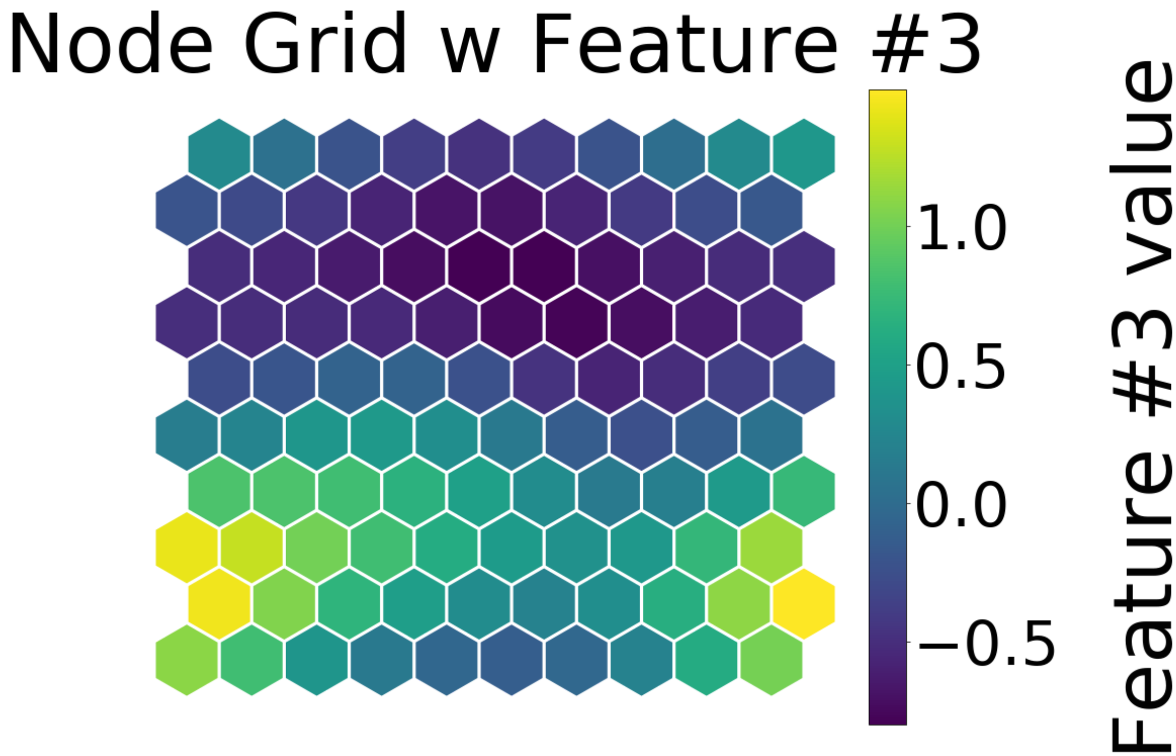
In [68]:

```
net.nodes_graph(colnum=2)
```



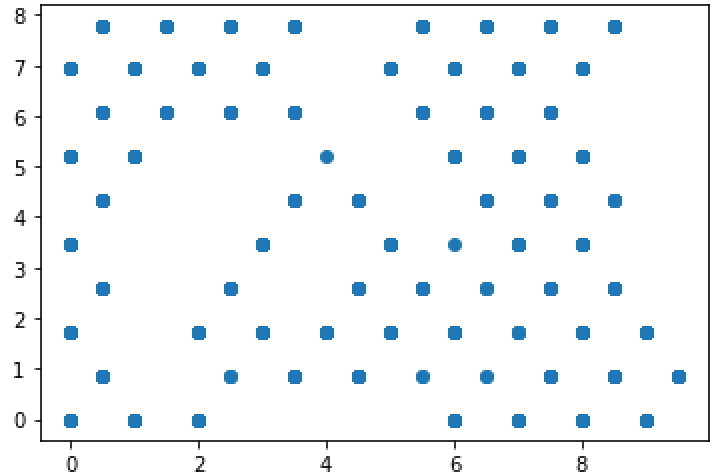
In [69]:

```
net.nodes_graph(colnum=3)
```



In [70]:

```
prj=np.array(net.project(X))
plt.scatter(prj.T[0],prj.T[1])
plt.show()
```



In [71]:

```
# Fitting kmeans to SOM
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=4, random_state=0).fit(prj)
```