Optimization Final Project Analysis and Modeling

Authors:

Tal Waitzenberg 305578189
Aviram Hadad (add ID Number)

Problem

In this project we are facing with the real estate invesments problem in Israel. In israel today there is a variety of people who looking for invest their money in real estate assets. each one of them as a different budget and interests, some of them looking for exists deals (buy in a lower price then market price and immediately sale in market price or higher) or long term invesments and earn from renting the asset. The question that arises is where is the best location and which kind of asset to buy (asset characteristics)?

Our Approach

After examining several existing academic studies, we developed an approach that believes that Israel has more than one market. Each region/city is divided into several different markets that determined according to the apartment price.

- how a market is defined? each market is defined according to apratment characteristics, that can be devided to 2 main categories:
  1) numeric features - like apartment size, number of rooms, price
  2) categorical features - like if the apratment includes

After spliting the data into markets we want to understand how a price is determined. The price of apartment is determned according to 2 categories:

- **house quality characteristics** - Variables characterizing the quality of the house, including number of rooms, net house size, house type and house age (The price ration of 2 different apratments will be the change of the quality characteristics between the apratments).
- **Environmental characteristics** – Variables characterizing the environmental quality of the house, including the socioeconomic level of the area's residents, the proximity to employment centers etc.

We wants to estimate the apartment price according to these categoties and use hedonic regression method which enables estimating the effect of each unit of a characteristic on the apartment total price.

In this notebook we are going to implements this 2 approaches we descrive above (markets, hedonic regression) in offer at the end a good markets for future investment.

In [306]:

```
import numpy as np
import pandas as pd
from sklearn.cluster import KMeans
from scipy.spatial import distance
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
from scipy.spatial.distance import euclidean
from sklearn.svm import SVR
import random, sys
from sklearn.model_selection import train_test_split
from mpl_toolkits.mplot3d import Axes3D
import itertools as iter
import locale
```

Reads Kiryat-Ono apartments history selling data

In [242]:

```
data = pd.read_csv("./kiryat_ono_data.csv")
data.shape
```

Out[242]:

```
(1730, 11)
```

In [243]:

```
data.dtypes
```

Out[243]:

```
geo_id                 object
sale_date              object
acutal_sale_price      object
sale_value             object
essence                object
sale_percentage       float64
city                   object
construction_year       int64
square_meters           int64
n_rooms               float64
se                    float64
dtype: object
```

```
In [244]:
```

```
data.head(5)
```

```
Out[244]:
```

	geo_id	sale_date	acutal_sale_price	sale_value	essence	sale_percentage	city	constr
0	006495-0231-000-00	23/01/2018	3,650,000	3,650,000	cottage tori	1.000	kiryat ono	
1	006495-0335-000-00	03/05/2018	3,575,000	3,575,000	cottage tori	0.374	kiryat ono	
2	006491-0106-001-00	25/10/2017	854,560	854,560	cottage single family	0.200	kiryat ono	
3	006492-0137-000-00	11/03/2019	1,500,000	948,000	cottage single family	0.250	kiryat ono	
4	006492-0196-000-00	22/07/2019	566,667	566,667	cottage single family	0.083	kiryat ono	

Checking for emptay or invalid data

```
In [245]:
```

```
# check for data sparse precentage
data.isnull().sum()
```

```
Out[245]:
```

```
geo_id                0
sale_date             0
acutal_sale_price     0
sale_value            0
essence               0
sale_percentage       0
city                  0
construction_year     0
square_meters         0
n_rooms               0
se                    0
dtype: int64
```

There is no empty cells, lets check for zeros

In [246]:

```
(data == 0).sum()
```

Out[246]:

```
geo_id                 0
sale_date              0
acutal_sale_price      0
sale_value             0
essence                0
sale_percentage        0
city                   0
construction_year     51
square_meters         50
n_rooms               51
se                     0
dtype: int64
```

We can see there is a few zeros in 'construction_year', 'square_meters' and 'n_rooms'. its not much so we will delete the rows

In [247]:

```
data = data[~(data.T == 0).any()]
data.shape
```

Out[247]:

```
(1679, 11)
```

Convert 'actual_sale_price', 'sale_value' to ints

In [248]:

```
locale.setlocale( locale.LC_ALL, 'en_US.UTF-8' )
data['acutal_sale_price'] = data['acutal_sale_price'].apply(lambda x: locale.ato
i(x))
data['sale_value'] = data['sale_value'].apply(lambda x: locale.atoi(x))
data.dtypes
```

Out[248]:

```
geo_id                 object
sale_date              object
acutal_sale_price       int64
sale_value              int64
essence                object
sale_percentage       float64
city                   object
construction_year       int64
square_meters           int64
n_rooms               float64
se                    float64
dtype: object
```

Markets Clustering

Has we described in our apporoach we want to devide the markets into segments, where each market is different from the other in their house quality characteristics. we dont know how many markets exists so we dont know the value of K (number of clusters). we going to use Kmeans algorithm but we eant to find the optimal number of cluster (the value of K). We going to use a global method to determine the number of clusters, this method called Hartigan's method.Hartigan propsed the folowing index:

- g - number of clusters
- W(g) - the mean distance of all points in a cluster with g clusters
- n - number of points

The smaller the value of W(g) the higher similarity between points in each cluster (with total of g clusters). The idea here is to start with g=1 and increase g by 1 in each iteration if Har(g+1) is significantly large. The stopping criteria is meet, where the stopping criteria is when Har(g) =< 10 (propsed by Hartigan).

We want to use our data for clustering. we need to convert all our features to numeric representation. the 'essence' feature which explain the kund of the aparatment is categorical feature and we going to use one hot encoder to represent this feature in a multiple binary features. we going to do the same for 'construction_year'.

In [225]:

```
clustering_data = data.copy()
clustering_data = clustering_data.drop('sale_date', axis=1)
clustering_data = clustering_data.drop('city', axis=1)
essence_dummies = pd.get_dummies(data['essence'])
construction_year_dummies = pd.get_dummies(data['construction_year'])
rooms_dummies = pd.get_dummies(data['n_rooms'])
clustering_data = clustering_data.drop('se', axis=1)
clustering_data = clustering_data.drop('essence', axis=1)
clustering_data = clustering_data.drop('construction_year', axis=1)
clustering_data = clustering_data.drop('n_rooms', axis=1)
clustering_data = pd.concat([clustering_data, construction_year_dummies, essence
_dummies, rooms_dummies], axis=1)
clustering_data = clustering_data.set_index('geo_id')
clustering_data.head(5)
```

Out[225]:

	acutal_sale_price	sale_value	sale_percentage	square_meters	1940	1950	1954	196
geo_id								
006495-0231-000-00	3650000	3650000	1.000	284	0	0	0	
006495-0335-000-00	3575000	3575000	0.374	240	0	0	0	
006492-0262-000-00	1360000	1360000	0.333	145	0	0	0	
006493-0113-000-00	5150000	5150000	1.000	350	0	0	0	
006493-0254-000-00	1542500	1542500	0.250	343	0	0	0	

5 rows × 82 columns

In [226]:

```
clustering_data.shape
```

Out[226]:

```
(1679, 82)
```

We got 82 features that we going to use for clustering. We are going to use Kmeans algorithm for clustering and we will use the hartigan metric in order to optimize the number of clusters K. We implemented k_hartigan function that will score every Kmeans estimator, by calculates the centroids in each cluster and the distance from the centroid in each cluster.

In [227]:

```
def calc_clusters_centroids(data, labels):
    '''
    calculates clusters centers by mean
    '''
    try:
        if isinstance(labels, pd.DataFrame):
            raise Exception('labels should be 1D array')

        if ~isinstance(data, pd.DataFrame):
            data = pd.DataFrame(data)

        labels = pd.DataFrame(labels, columns=['labels'], index=data.index)
        df = pd.concat([data, labels], axis=1)
        centers = pd.DataFrame.groupby(df, by='labels', as_index=True).agg('mea
n')
        return centers
    except Exception as e:
        print(e)
        raise e


def calc_points_distances(data, labels, centroids):
    '''
    calc W by the formula:
    W = sum((1/2nr)*sum((Xij - Centroid_i_tagj)^2))
    nr - number of obsevations in cluster
    i - observation
    j - feature in observation
    '''

    try:

        labels_df = pd.DataFrame(labels, columns=['labels'], index=data.index )
        data_copy = data.copy()
        data_copy = pd.concat([data_copy, labels_df], axis=1)
        clusters = pd.DataFrame.groupby(data_copy, by='labels', as_index=True)
        W = sum([(sum(map(lambda r: np.power(distance.euclidean(r[:-1], centroid
s.loc[label]), 2), Cr.values))) for label, Cr in clusters])
        return np.log1p(W)
    except Exception as e:
        print("Error:",e)
        raise e

def k_hartigan(Wks, n_clusters, n_observations, threshold=10):
    '''
    Return optimal K (number of clusters) according to Hartigan score
    '''

    if len(Wks) != len(n_clusters):
        raise Exception('Error: Wks length should be equal to n_clusters')

    try:
        it1, it2 = iter.tee(Wks, 2)
        next(it2, None)
        optimal_k = -1
        for i, (Wk, Wk_plus_1) in enumerate(zip(it1, it2)):
            k = n_clusters[i]
            H = np.multiply(np.true_divide(Wk, Wk_plus_1) - 1, n_observations -
k - 1)
```

```
            if H < threshold:
                optimal_k = k
                break

        return optimal_k
    except Exception as e:
        print(e)
        raise e
```

Now we going to run our algorithm and find the best Kmeans algorthm with optimal number of clusterz that will be the number of markets in Kiryat Ono:

In [228]:

```
wks = []
kmeans_estimator_prev = KMeans(n_clusters=1, random_state=0)
kmeans_estimator_prev.fit(clustering_data.values)
labels_prev = kmeans_estimator_prev.labels_
centroids_prev = calc_clusters_centroids(clustering_data, labels_prev)
wk_prev = calc_points_distances(clustering_data, labels_prev, centroids_prev)
wks.append(wk_prev)
k=2
optimal_k = -1
while True:
    kmeans_estimator_next = KMeans(n_clusters=k, random_state=0)
    kmeans_estimator_next.fit(clustering_data.values)
    labels_next = kmeans_estimator_next.labels_
    centroids_next = calc_clusters_centroids(clustering_data, labels_next)
    wk_next = calc_points_distances(clustering_data, labels_next, centroids_next
)
    wks.append(wk_next)
    is_optimal_k = k_hartigan([wk_prev, wk_next], [k-1,k], clustering_data.shape
[0])

    if is_optimal_k > -1:
        kmeans_best_estimator = kmeans_estimator_prev
        optimal_k = is_optimal_k
        break

    k+=1
    kmeans_estimator_prev = kmeans_estimator_next
    wk_prev = wk_next
print("Optimal number of markets %s" % str(optimal_k))
```

```
Optimal number of markets 12
```

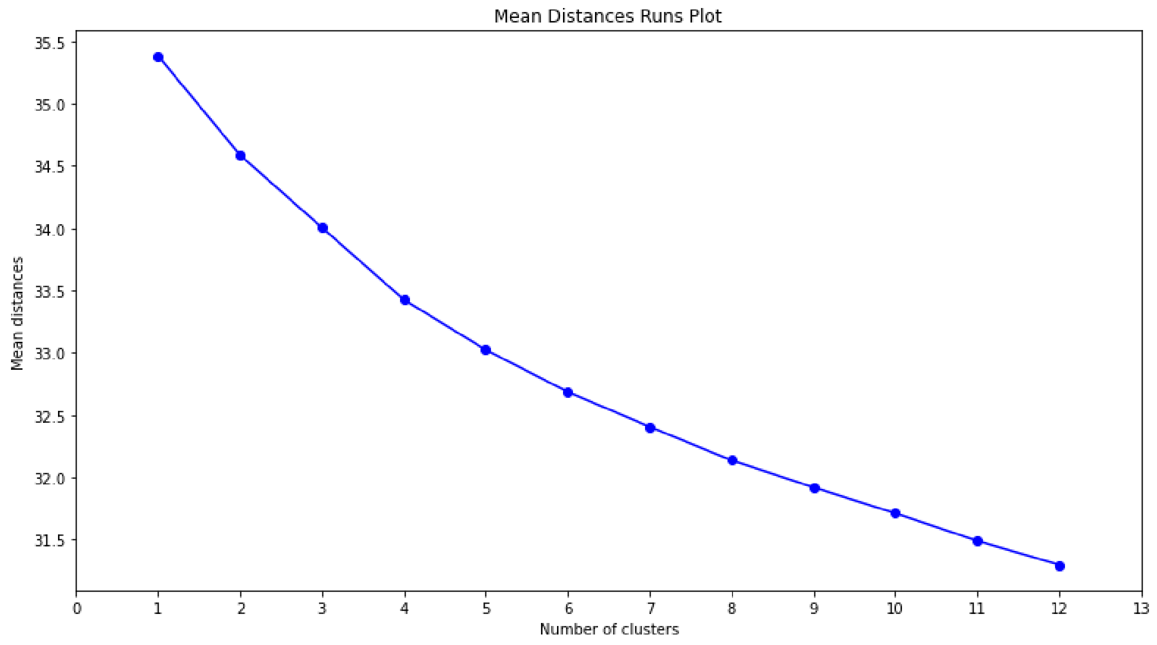
Let's plot the mean distances of each run (kmeans where K=1.....12) and to see if the distances got smaller

In [229]:

```
fig, ax = plt.subplots(figsize=(13,7))
x = list(range(1,optimal_k + 1))
y = wks[:-1]
ax.scatter(x, y, color='blue')
ax.plot(x, y, color='blue', linestyle='solid')
ax.set_xticks(np.arange(len(x) + 2))
plt.xlabel('Number of clusters')
plt.ylabel('Mean distances')
plt.title("Mean Distances Runs Plot")
```

Out[229]:

```
Text(0.5, 1.0, 'Mean Distances Runs Plot')
```



As we cann see the distance went down and the difference ration got smaller and smaller when we reached 12.

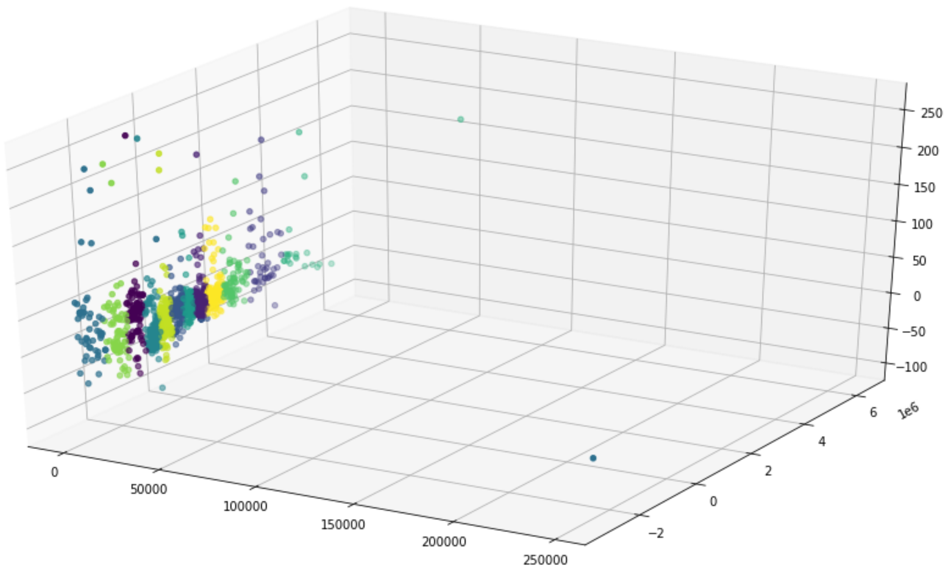
We also wants to plot the clusters, In order to plot the best algorithm clusters we need to reduce the number of dimensions. We going to use PCA to reduce the number of dimensions to 3 and plot the cluster on 3D plot.

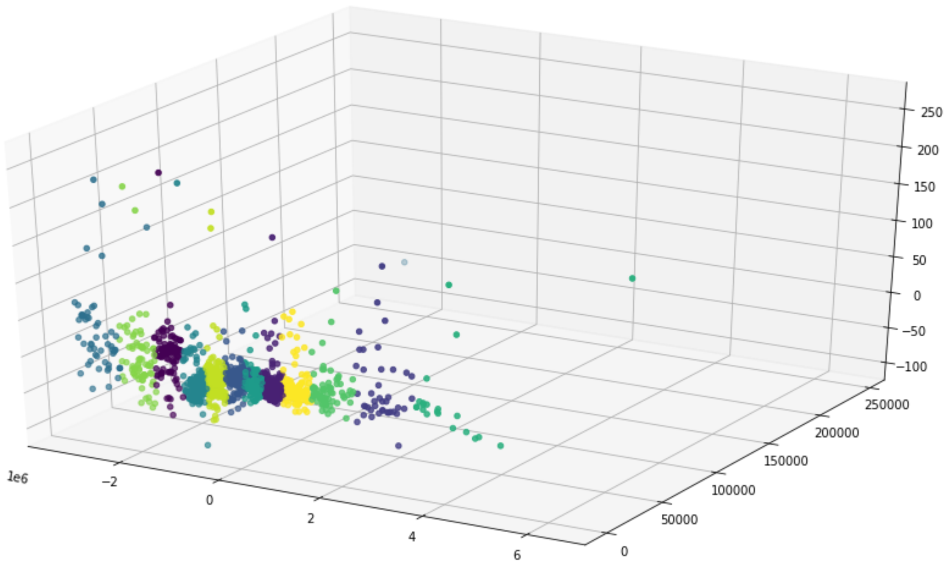
In [230]:

```
def plot_3d_scatter(data,labels):
    fig1 = plt.figure(figsize=(13,7))
    ax1  = Axes3D(fig1)
    ax1.scatter(data[:,1], data[:,0], data[:,2],c=labels)

    fig2 = plt.figure(figsize=(13,7))
    ax2  = Axes3D(fig2)
    ax2.scatter(data[:,0], data[:,1], data[:,2],c=labels)

    plt.show()
```

In [231]:

```
# perform PCA for plotting
kmeans_pca = PCA(n_components=3)
kmeans_pca_data = kmeans_pca.fit_transform(clustering_data)

# plot 3D scatter
plot_3d_scatter(kmeans_pca_data,kmeans_best_estimator.labels_)
```





As we can see we got 12 clusters, some of them really dense and some of the are less. it can point that there is some really simillar apartments in the data that are the majority in the market and there some apartments that are more rare and different than others (like penthouses, Vilas etc).

Prices Hedonic Regression

We are going to use hedonic regression for esimate house prices. We going to use SVR (Support Vector Regression) with the 'rbf' Kernel function to transform the data to linear space.

Support Vector Regression (SVR) gives us the flexibility to define how much error is acceptable in our model and will find an appropriate line (or hyperplane in higher dimensions) to fit the data. In contrast to OLS, the objective function of SVR is to minimize the coefficients — more specifically, the l2-norm of the coefficient vector — not the squared error. The error term is instead handled in the constraints, where we set the absolute error less than or equal to a specified margin, called the maximum error, ϵ (epsilon). We can tune epsilon to gain the desired accuracy of our model. Our new objective function and constraints:

Where C and ϵ are SVR hyperparameters. SVR is a powerful algorithm that allows us to choose how tolerant we are of errors, both through an acceptable error margin(ϵ) and through tuning our tolerance of falling outside that acceptable error rate.

FireFly Optimization

We want to optimizae SVR hyperparameters, we going to use Firefly (FA) optimization algorithm. Firefly algorithm was first developed by `Yang in 2007 "Firefly Algorithms for Multimodal Optimization"` which was based on the flashing patterns and behavior of fireflies. FireFly uses the following three idealized rules:

1. Fireflies are unisexual so that one firefly will be attracted to other fireflies regardless of their sex.
2. The attractiveness is proportional to the brightness and they both decrease as their distance increases. Thus, for any two flashing fireflies, the less brighter one will move toward the more brighter one. If there is no brighter one than a particular firefly, it will move randomly.
3. The brightness of a firefly is determined by the landscape of the objective function.

The movement of a firefly i is attracted to another, more attractive (brighter) firefly j is determined by:

- The first term: `X_i_t` is the position of `X_i` at iteration t.
- The second term: is the attraction where `beta_0` is the attractiveness at the distance where `r = 0`.
- The third term: is randomization with `alpha` being the randomization parameter.

We will implement the Firefly algorithm according to the `"Firefly Algorithms for Multimodal Optimization"` article by `Yang`. here is the pseudo code for the algorithm:

We will create a class called `Point` to represent each point data and values on our space.

In [267]:

```
class Point(object):

    def __init__(self, **kwargs):
        """
        :param kwargs:
        range: point boundaries, lower and upper
        dimension: point dimension
        objective_function: point objective function
        """
        self.range = kwargs.get('range', (0, 1))
        self.dimension = kwargs.get('dimension', 2)
        self.seed = kwargs.get('seed', None)

        # set random seed
        if self.seed is not None:
            np.random.seed(self.seed)

        # check range instance according to dimension
        if isinstance(self.range, list):
            self.x_min = np.array(list(map(lambda x: x[0], self.range)))
            self.x_max = np.array(list(map(lambda x: x[1], self.range)))
        else:
            self.x_min = np.array([self.range[0]] * self.dimension)
            self.x_max = np.array([self.range[1]] * self.dimension)

        # initialize position
        self.__position = np.random.uniform(self.x_min, self.x_max, self.dimensi
on)


    def get_position(self):
        return self.__position

    def set_position(self, new_position) -> None:
        self.__position = new_position
```

Now we will decide to use `Euclidean` metric to calculates the distances between points

In [268]:

```
def euclidean_distance(x1, x2):
    d = euclidean(u=x1, v=x2)
    return d
```

Now we going to create a class named `Firefly` that will represent one firefly in our swarm. each `Firefly` class will encapsulate the `Point` class. In this class we will implements the move towrds method and the update light intensity.

In [292]:

```
class Firefly(Point):

    def __init__(self, objective_function, **kwargs):
        """
        :param kwargs:
        alpha: Step size scaling factor.
        beta_0: The attractiveness at the distance r=0.
        gamma: The light absorption coefficient.
        distance_metric: the distance r_ij between two fireflies x_i and x_j, de
fault: 'euclidean'.
        range: point boundaries, lower and upper
        dimension: position dimension
        objective_function: point objective function
        """
        super().__init__(**kwargs)

        self.objective_function = objective_function
        self.alpha = kwargs.get('alpha', 0.3)
        self.beta_0 = kwargs.get('beta_0', 0.8)
        self.gamma = kwargs.get('gamma', 1e-5)
        self.distance_metric = kwargs.get('distance_metric', 'euclidean')
        self.range = kwargs.get('range', (0, 1))
        self.dimension = kwargs.get('dimension', 2)
        self.seed = kwargs.get('seed', None)

        # set random seed
        if self.seed is not None:
            np.random.seed(self.seed)

        # check range instance according to dimension
        if isinstance(self.range, list):
            self.x_min = np.array(list(map(lambda x: x[0], self.range)))
            self.x_max = np.array(list(map(lambda x: x[1], self.range)))
        else:
            self.x_min = np.array([self.range[0]] * self.dimension)
            self.x_max = np.array([self.range[1]] * self.dimension)

        self.__value = self.objective_function(self.get_position())
        self.__distance_metrics = {
            'euclidean': euclidean_distance
        }

    def __attractiveness(self, x1_pos, x2_pos) -> float:
        """
        calculates the attractiveness between Firefly x1 to Firefly x2 according
to position
        :param x1_pos: 1D vector
        :param x2_pos: 1D vector
        :return: Float
        """
        r = self.__distance_metrics[self.distance_metric](x1_pos, x2_pos)
        beta_r_ij = self.beta_0 * np.exp(-self.gamma * (r ** 2))
        return beta_r_ij

    def move_towards(self, x: Point, alpha: float = None) -> None:
        """
        Moves point of firefly towards point x
        alpha should decreases to 0
        :param x: 1D vector
        :param alpha: float (optional)
        :return: None
        """
        if alpha is not None:
            self.alpha = alpha

        x1_pos = self.get_position()
        x2_pos = x.get_position()
        attractiveness = self.__attractiveness(x1_pos, x2_pos) * (x2_pos - x1_po
s)
        new_position = self.get_position() + attractiveness + (self.alpha * (np.
random.uniform(0, 1) * -.5))
        new_position = np.clip(a=new_position, a_min=self.x_min, a_max=self.x_ma
x)
        self.set_position(new_position)

    def update_poor_firefly_position(self, mu):
        """
        This method moves poor firefly light intensity according to mu
        :param mu: float between 0 to 1
        :return: None
        """
        # x_poor(t+1) = mu(t) * (x_max - x_min) + x_min
        new_modified_position = mu * (self.x_max - self.x_min) + self.x_min
        self.set_position(new_modified_position)

    def update_light_intensity(self) -> None:
        """
        Updates Firefly light intensity
        :return: None
        """
        self.__value = self.objective_function(self.get_position())

    def update_alpha(self, t) -> None:
        """
        This method recalculates alpha according to iteration number t
        alpha gradually decreases as the iterations increases
        :param t: iteration number
        :return: None
        """
        self.alpha = (np.exp(1) - ((1 + np.true_divide(1, t)) ** t)) * self.alph
a

    def get_light_intensity(self) -> float:
        """
        returns Firefly light intensity
        :return: float
        """
        return self.__value
```

Finally we can implement the algorithm itself according to the pseudo code we described above.

In [342]:

```
class FireflyAlgorithm(object):

    def __init__(self, population_size, objective_function, **kwargs):
        """
        :param population_size: number of Fireflies.
        :param kwargs:
        alpha: Step size scaling factor.
        beta_0: The attractiveness at the distance r=0.
        gamma: The light absorption coefficient.
        distance_metric: the distance r_ij between two fireflies x_i and x_j, de
fault: 'euclidean'.
        range: point boundaries, lower and upper.
        dimension: position dimension.
        objective_function: point objective function.
        max_generation: maximal number of iterations.
        p: the probability of opposition-based FA.
        seed: set seed for testing purposes
        """
        self.population_size = population_size
        self.objective_function = objective_function
        self.alpha = kwargs.get('alpha', 0.3)
        self.max_generation = kwargs.get('max_generation', 100)
        self.p = kwargs.get('p', 0.75)
        self.diff_threshold = kwargs.get('diff_threshold', 1e-100)
        self.seed = kwargs.get('seed', None)
        self.fireflies = kwargs.get('fireflies', None)

        # set random seed
        if self.seed is not None:
            random.seed(self.seed)

        # initialize Fireflies and light intensity of each Firefly.
        if self.fireflies is None:
            self.fireflies = [Firefly(self.objective_function, **kwargs) for i i
n range(self.population_size)]

    def plot_position(self, title=""):
        x = []
        y = []
        for f in self.fireflies:
            p = f.get_position()
            x.append(p[0])
            y.append(p[1])

        plt.xlim(0, 1000)
        plt.ylim(0, 100)
        plt.scatter(x, y, alpha=0.5)
        plt.title(title)
        plt.xlabel('x')
        plt.ylabel('y')
        plt.show()

    def update_alpha(self, t) -> None:
        """
        This method recalculates alpha according to iteration number t
        alpha gradually decreases as the iterations increases
        :param t: iteration number
        :return: None
        """
        self.alpha = (np.exp(1) - ((1 + np.true_divide(1, t)) ** t)) * self.alph
a

    def iteration_message(self, iter_index: int, firefly: Firefly):
        message = "Iteration Number: %s, best objective value: %s, best positio
n: "
        for _ in range(firefly.dimension):
            message += "%s, "
        message = message[:-2]
        message_values = [iter_index, firefly.get_light_intensity()] + list(map(
lambda x: str(x), firefly.get_position()))
        print(message % tuple(message_values))

    def progress(self, count, total, status=''):
        bar_len = 60
        filled_len = int(round(bar_len * count / float(total)))

        percents = round(100.0 * count / float(total), 1)
        bar = '#' * filled_len + '-' * (bar_len - filled_len)
        sys.stdout.write('[%s] %s%s ...%s\r' % (bar, percents, '%', status))
        sys.stdout.flush()

    def solve(self, verbose=False, plot=False, **kwargs):
        """
        solving the opposition-based chaotic FA problem
        return the most brighter firefly
        :param verbose: bool, prints iterations messages
        :param plot: bool, plot point in 2D
        :param kwargs: dict, additional parameters
            * 'dynamic_alpha: bool, change alpha dynamically - default: False
        :return:
        """
        dynamic_alpha = kwargs.get('dynamic_alpha', False)
        best_firefly: Firefly = None
        prev_firefly: Firefly = None
        for t in range(1, self.max_generation + 1):

            if plot:
                self.plot_position("iteration: " + str(t))

            progress_i = 0
            # passing on brighter fireflies and update positions
            for i in range(self.population_size):
                for j in range(self.population_size):
                    x_i = self.fireflies[i]
                    x_j = self.fireflies[j]

                    # if x_i.light_intensity < x_j.light_intensity moves x_i tow
ards x_j and updates x_i light intensity
                    if x_j.get_light_intensity() < x_i.get_light_intensity():
                        x_i.move_towards(x_j, alpha=self.alpha)
                        x_i.update_light_intensity()

                    self.progress(progress_i, self.population_size**2)
                    progress_i += 1

            # finding the most brighter firefly (best firefly)
            current_best_firefly: Firefly = min(self.fireflies, key=lambda f: f.
get_light_intensity())
            if best_firefly is None or current_best_firefly.get_light_intensity
() < best_firefly.get_light_intensity():
```

```
                best_firefly = current_best_firefly

            if verbose:
                self.iteration_message(t, current_best_firefly)

            # checking if diff smaller than diff_threshold if does stop running.
            if prev_firefly is not None:
                diff = np.abs(np.true_divide(prev_firefly.get_light_intensity(),
current_best_firefly.get_light_intensity()) - 1)
                if diff < self.diff_threshold:
                    continue
            prev_firefly = current_best_firefly

            # updates alpha for next iteration
            if dynamic_alpha:
                self.update_alpha(t+1)

        return best_firefly
```

Now after we finished with the implementation we want to use the Firefly optimization on the SVR model. We want to optimize the SVR model hyperparametres:

- `C` - Regularization parameter.
- `ϵ` - It specifies the SVR epsilon-tube within which no penalty is associated in the training loss function with points predicted within a distance ϵ from the actual value.
- `gamma` - Kernel coefficient for `rbf` .

each `Firefly` point will have a vector of length 3. which represent the values of `C` , `gamma` and `ϵ` . We need to create our `objective function` that will recieve a `Firefly` and we return us the light intensity of the firefly according to the `Point` vector. The light intensiity will be the `MSE` (Mean Square Error) of the model according to the formula:

In [381]:

```
def MSE(y_true, y_hat):
    return np.mean((y_true - y_hat) ** 2)
```

And our `objective function` implementation

In [382]:

```
def objective_function(X_train, X_test, y_train, y_test):
    def inner(x):
        C, gamma, epsilon = x

        # Train SVR model on train set
        clf = SVR(C=C, epsilon=epsilon, gamma=gamma)
        clf.fit(X_train, y_train)

        # predict SVR model on test data
        y_hat = clf.predict(X_test)

        # get accuracy of fitness functions MAPE, MSE, RMSE
        mse = MSE(y_test, y_hat)

        return mse
    return inner
```

Now lets create our dataset for training

In [416]:

```
svr_data = data.copy()
svr_data = svr_data.drop('sale_date', axis=1)
svr_data = svr_data.drop('city', axis=1)
essence_dummies = pd.get_dummies(data['essence'])
construction_year_dummies = pd.get_dummies(data['construction_year'])
rooms_dummies = pd.get_dummies(data['n_rooms'])
svr_data = svr_data.drop('essence', axis=1)
svr_data = svr_data.drop('construction_year', axis=1)
svr_data = svr_data.drop('n_rooms', axis=1)
svr_data = pd.concat([svr_data, construction_year_dummies, essence_dummies, room
s_dummies], axis=1)
svr_data = svr_data.set_index('geo_id')
svr_data.head(5)
```

Out[416]:

	acutal_sale_price	sale_value	sale_percentage	square_meters	se	1940	1950
geo_id							
006495-0231-000-00	3650000	3650000	1.000	284	0.303504	0	0
006495-0335-000-00	3575000	3575000	0.374	240	5.652849	0	0
006492-0262-000-00	1360000	1360000	0.333	145	2.183057	0	0
006493-0113-000-00	5150000	5150000	1.000	350	4.153829	0	0
006493-0254-000-00	1542500	1542500	0.250	343	3.444334	0	0

5 rows × 83 columns

Finally we can run our optimization algorithm. we going to run the `Firefly` on each one of our `12 sub markets` we got with our clustering. so lets split our data according to the markets than on each market we will run the otimization with train and test data.

we will set firefly optimization argumnets `maximum iterations` to `5` and the number of fireflies will be `20` (due to time complexity issues). Then we add the following `constraints` on our `SVR` hyperparameters:

- `gamma`: `0 <= gamma <= 100`
- `ϵ`: `-log 6 <= ϵ <= log 6`
- `C`: `-log 6 <= C <= log 6`

We will plot the last market `11` iterations in order to see how the `Firefly` works.

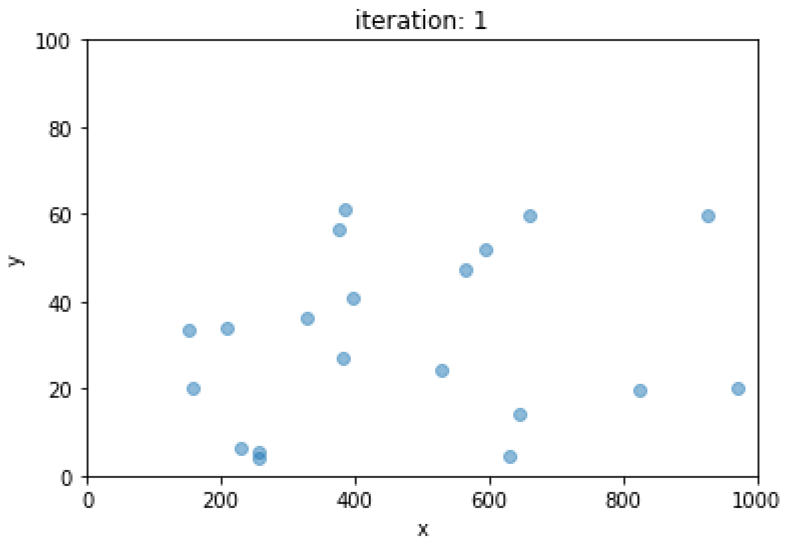
In [419]:

```
models = {}
svr_data['market'] = kmeans_estimator_prev.labels_
data_markets = pd.DataFrame.groupby(svr_data, by='market')
for market, market_df in data_markets:
    print("Optimize market number %s" % str(market))
    df = market_df.drop('market', axis=1)
    X = df.drop(['acutal_sale_price', 'sale_value'], axis=1)
    y = df['acutal_sale_price']

    # split train and test
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, ran
dom_state=42)

    ################################################################################
########
    # Fine tuning of SVR hyper parameters gamma, epsilon and C using FireFly alg
orithm #
    ################################################################################
########
    obj_func = objective_function(X_train, X_test, y_train, y_test)
    population_size = 20
    kwargs = {
        "dimension": 3,
        "max_generation": 5,
        "range": [(0, 1e3), (2 ** -6, 2 ** 6), (2 ** -6, 2 ** 6)],
        "alpha": 0.3,
        "beta_0": 0.8,
        "gamma": 1e-5,
        "p": 0.75,
        "control": 4
    }

    if market == 11:
        fa = FireflyAlgorithm(population_size=population_size, objective_functio
n=obj_func, **kwargs)
        best_fa = fa.solve(verbose=True, plot=True)
    else:
        fa = FireflyAlgorithm(population_size=population_size, objective_functio
n=obj_func, **kwargs)
        best_fa = fa.solve(verbose=True, plot=False)

        print("light intensity: %s" % str(best_fa.get_light_intensity()))
        print("position: %s, %s, %s" % (
        str(best_fa.get_position()[0]), str(best_fa.get_position()[1]), str(best
_fa.get_position()[2])))
        C, gamma, epsilon = best_fa.get_position()[0], best_fa.get_position()[1
], best_fa.get_position()[2]
        clf = SVR(C=C, epsilon=epsilon, gamma=gamma)
        clf.fit(X_train, y_train)
        y_train_hat = clf.predict(X_train)
        y_test_hat = clf.predict(X_test)
        train_mse_accuracy = MSE(y_train, y_train_hat)
        test_mse_accuracy = MSE(y_test, y_test_hat)
        print('Train MSE: %s' % str(train_mse_accuracy))
        print('Test MSE: %s' % str(test_mse_accuracy))
        models[market] = {
            "best_firefly": best_fa,
            "model": clf
        }
```

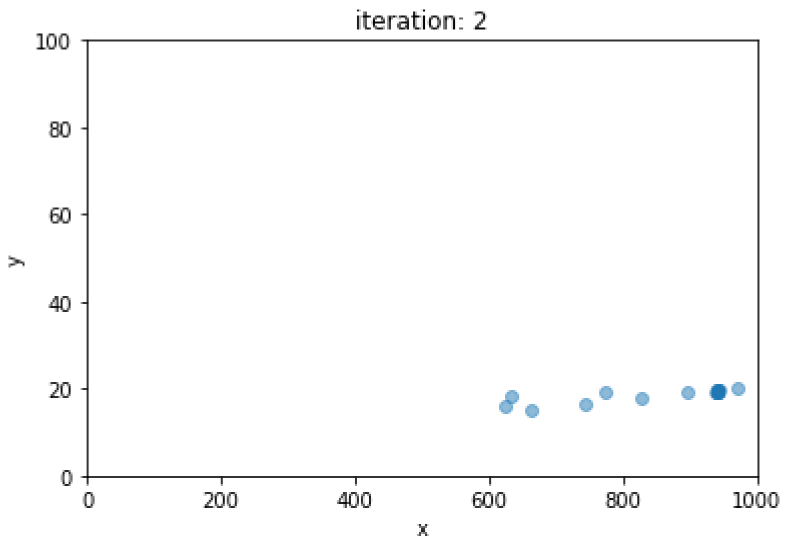
```
Optimize market number 0
Iteration Number: 1, best objective value: 9111992363.894838, best p
osition: 578.943263068268, 4.986317851884515, 8.664769244738547
Iteration Number: 2, best objective value: 9107559518.73055, best po
sition: 578.2623862109849, 3.784613304136234, 7.63679470992015
Iteration Number: 3, best objective value: 9100255972.34452, best po
sition: 576.9886942133753, 2.5118012634981115, 6.361813869129457
Iteration Number: 4, best objective value: 9091431131.732998, best p
osition: 576.1700517026472, 1.6931129118400159, 5.543135572516986
Iteration Number: 5, best objective value: 9070241421.020283, best p
osition: 575.3890060673973, 0.9121197980912846, 4.76213911708488
light intensity: 9070241421.020283
position: 575.3890060673973, 0.9121197980912846, 4.76213911708488
Train MSE: 9608041826.251116
Test MSE: 9070241421.020283
Optimize market number 1
Iteration Number: 1, best objective value: 8150655328.033969, best p
osition: 938.3179823560461, 6.25951831466461, 30.42965635587622
Iteration Number: 2, best objective value: 8149678106.853085, best p
osition: 936.9079665898378, 5.079395631347081, 29.20643770597865
Iteration Number: 3, best objective value: 8148131986.805416, best p
osition: 935.5473658637745, 3.7181135730074732, 27.84735507784103
Iteration Number: 4, best objective value: 8146920403.201378, best p
osition: 934.5986684707221, 2.7685922126177784, 26.897844497393283
Iteration Number: 5, best objective value: 8140432638.245183, best p
osition: 933.2414038676078, 1.411341487780581, 25.540592885235405
light intensity: 8140432638.245183
position: 933.2414038676078, 1.411341487780581, 25.540592885235405
Train MSE: 4575880310.113525
Test MSE: 8140432638.245183
Optimize market number 2
Iteration Number: 1, best objective value: 25223214795.692307, best
position: 743.4498805924544, 53.64487393559815, 28.308291268444542
Iteration Number: 2, best objective value: 25223214795.692307, best
position: 743.4498805924544, 53.64487393559815, 28.308291268444542
Iteration Number: 3, best objective value: 25223214795.692307, best
position: 743.4498805924544, 53.64487393559815, 28.308291268444542
Iteration Number: 4, best objective value: 25223214795.692307, best
position: 743.4498805924544, 53.64487393559815, 28.308291268444542
Iteration Number: 5, best objective value: 25223214795.692307, best
position: 743.4498805924544, 53.64487393559815, 28.308291268444542
light intensity: 25223214795.692307
position: 743.4498805924544, 53.64487393559815, 28.308291268444542
Train MSE: 48306108636.80978
Test MSE: 25223214795.692307
Optimize market number 3
Iteration Number: 1, best objective value: 5260527733.991147, best p
osition: 849.2370127818818, 6.2005758416994015, 62.606401361297856
Iteration Number: 2, best objective value: 5258804831.754906, best p
osition: 847.3322065067069, 5.342780305382138, 61.6154359613579
Iteration Number: 3, best objective value: 5256285603.523647, best p
osition: 846.4337808494971, 4.327089920395039, 60.614375716380046
Iteration Number: 4, best objective value: 5252386689.165428, best p
osition: 845.2479494315339, 3.139438445782032, 59.42689151502619
Iteration Number: 5, best objective value: 5248362043.513653, best p
osition: 844.4083334404421, 2.299813978824986, 58.58726817834409
light intensity: 5248362043.513653
position: 844.4083334404421, 2.299813978824986, 58.58726817834409
Train MSE: 5233198716.857657
Test MSE: 5248362043.513653
Optimize market number 4
```

```
Iteration Number: 1, best objective value: 28670527727.266136, best
position: 490.20003991322386, 6.136077920748617, 38.82366753478754
Iteration Number: 2, best objective value: 28669769355.13459, best p
osition: 502.4454403679142, 5.639712101861786, 38.09093826673236
Iteration Number: 3, best objective value: 28668591263.69423, best p
osition: 500.917412018318, 4.404573170496482, 36.81045550222346
Iteration Number: 4, best objective value: 28667458651.073395, best
position: 499.82861249900753, 3.314866591692189, 35.719947912785436
Iteration Number: 5, best objective value: 28666251656.417923, best
position: 498.76518980312534, 2.2514838262595953, 34.6565931322068
light intensity: 28666251656.417923
position: 498.76518980312534, 2.2514838262595953, 34.6565931322068
Train MSE: 30865852453.710583
Test MSE: 28666251656.417923
Optimize market number 5
Iteration Number: 1, best objective value: 5093885309.396536, best p
osition: 592.4286080091186, 2.5474198820182625, 40.593385616269515
Iteration Number: 2, best objective value: 5089816868.5508375, best
position: 593.1253583302948, 1.9207270312803124, 39.83167834490967
Iteration Number: 3, best objective value: 5080773661.844506, best p
osition: 592.2605195723238, 0.7521730363703953, 38.63234191239946
Iteration Number: 4, best objective value: 4963450934.240145, best p
osition: 591.5595699374222, 0.04852632074565823, 37.928397545608256
Iteration Number: 5, best objective value: 4916573866.451246, best p
osition: 591.5192462617472, 0.015625, 37.88807094926977
light intensity: 4916573866.451246
position: 591.5192462617472, 0.015625, 37.88807094926977
Train MSE: 5348648469.281878
Test MSE: 4916573866.451246
Optimize market number 6
Iteration Number: 1, best objective value: 4794837152.389636, best p
osition: 882.7921806607903, 12.839708874597473, 22.54515573574949
Iteration Number: 2, best objective value: 4794557145.245562, best p
osition: 882.1678642592901, 12.226483739574258, 21.93254047488028
Iteration Number: 3, best objective value: 4794084606.8708105, best
position: 881.0959103182972, 11.151494267317721, 20.85749150527751
Iteration Number: 4, best objective value: 4793626476.573859, best p
osition: 879.924908418249, 9.980571484196979, 19.686572172598602
Iteration Number: 5, best objective value: 4793270858.837603, best p
osition: 878.709035089386, 8.764697381967467, 18.47069803840841
light intensity: 4793270858.837603
position: 878.709035089386, 8.764697381967467, 18.47069803840841
Train MSE: 4081145995.7493553
Test MSE: 4793270858.837603
Optimize market number 7
Iteration Number: 1, best objective value: 72978554931.6068, best po
sition: 560.5366799628777, 0.015625, 22.618978119221214
Iteration Number: 2, best objective value: 72977168770.06633, best p
osition: 563.6127126116102, 0.015625, 23.462767273929583
Iteration Number: 3, best objective value: 72977168770.06633, best p
osition: 563.6127126116102, 0.015625, 23.462767273929583
Iteration Number: 4, best objective value: 72977168770.06633, best p
osition: 563.6127126116102, 0.015625, 23.462767273929583
Iteration Number: 5, best objective value: 72977168770.06633, best p
osition: 563.6127126116102, 0.015625, 23.462767273929583
light intensity: 72977168770.06633
position: 563.6127126116102, 0.015625, 23.462767273929583
Train MSE: 325794123493.91345
Test MSE: 72977168770.06633
Optimize market number 8
Iteration Number: 1, best objective value: 21829095622.11905, best p
```

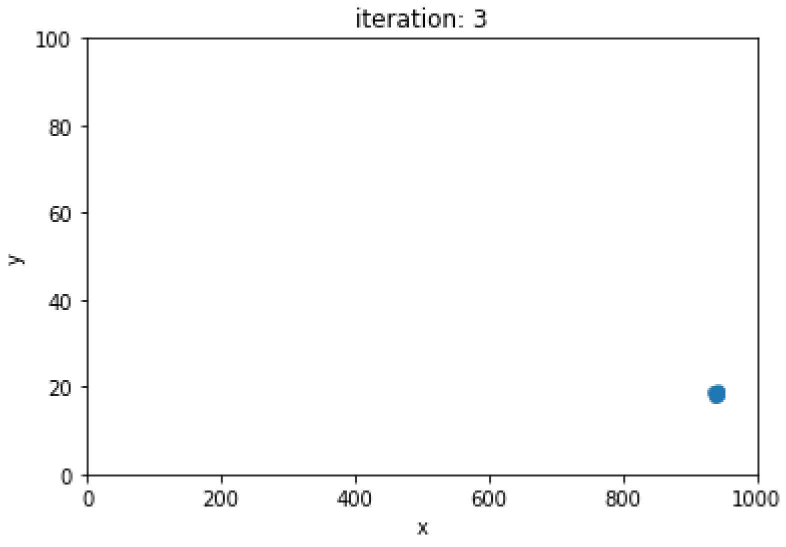
```
osition: 938.9801513636114, 54.88811475956451, 9.944707467251472
Iteration Number: 2, best objective value: 21829095622.11905, best p
osition: 938.9801513636114, 54.88811475956451, 9.944707467251472
Iteration Number: 3, best objective value: 21829095622.11905, best p
osition: 938.9801513636114, 54.88811475956451, 9.944707467251472
Iteration Number: 4, best objective value: 21829095622.11905, best p
osition: 938.9801513636114, 54.88811475956451, 9.944707467251472
Iteration Number: 5, best objective value: 21829095622.11905, best p
osition: 938.9801513636114, 54.88811475956451, 9.944707467251472
light intensity: 21829095622.11905
position: 938.9801513636114, 54.88811475956451, 9.944707467251472
Train MSE: 25978810884.387238
Test MSE: 21829095622.11905
Optimize market number 9
Iteration Number: 1, best objective value: 19640104618.836674, best
position: 915.6813977253362, 54.46497319371843, 50.32847286668365
Iteration Number: 2, best objective value: 19640104618.836674, best
position: 915.6813977253362, 54.46497319371843, 50.32847286668365
Iteration Number: 3, best objective value: 19640104618.836674, best
position: 915.6813977253362, 54.46497319371843, 50.32847286668365
Iteration Number: 4, best objective value: 19640104618.836674, best
position: 915.6813977253362, 54.46497319371843, 50.32847286668365
Iteration Number: 5, best objective value: 19640104618.836674, best
position: 915.6813977253362, 54.46497319371843, 50.32847286668365
light intensity: 19640104618.836674
position: 915.6813977253362, 54.46497319371843, 50.32847286668365
Train MSE: 13521338217.43993
Test MSE: 19640104618.836674
Optimize market number 10
Iteration Number: 1, best objective value: 6297928735.037404, best p
osition: 300.6233655896845, 3.8696434106882025, 50.991799721356855
Iteration Number: 2, best objective value: 6297776191.210303, best p
osition: 300.3769338084698, 3.387412557868389, 50.53705139262835
Iteration Number: 3, best objective value: 6297776191.210303, best p
osition: 300.3769338084698, 3.387412557868389, 50.53705139262835
Iteration Number: 4, best objective value: 6297776191.210303, best p
osition: 300.3769338084698, 3.387412557868389, 50.53705139262835
Iteration Number: 5, best objective value: 6297776191.210303, best p
osition: 300.3769338084698, 3.387412557868389, 50.53705139262835
light intensity: 6297776191.210303
position: 300.3769338084698, 3.387412557868389, 50.53705139262835
Train MSE: 6041222863.008933
Test MSE: 6297776191.210303
Optimize market number 11
```



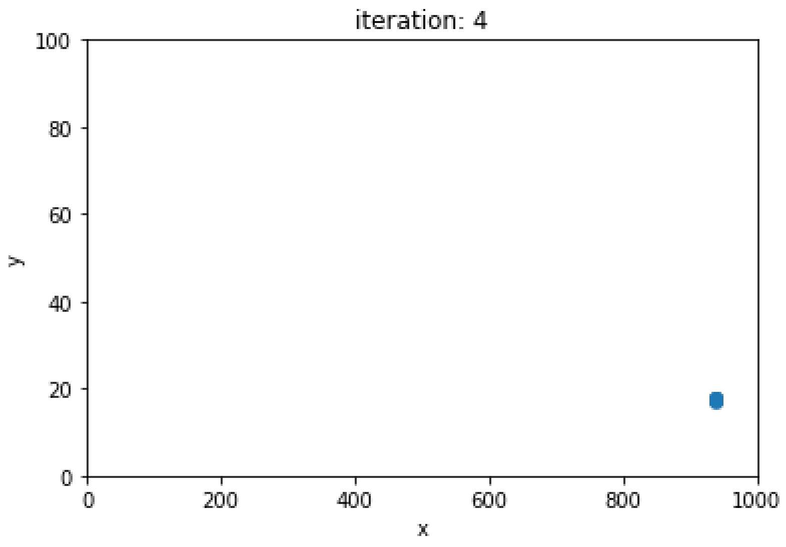
```
Iteration Number: 1, best objective value: 9211800073.633821, best p
osition: 940.5197922967208, 19.056429808952885, 29.771958860226874
```



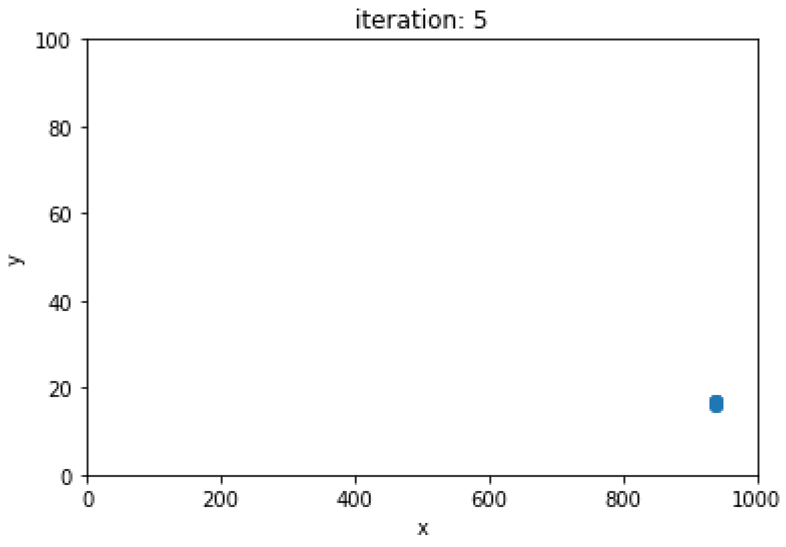
```
Iteration Number: 2, best objective value: 9211469420.424042, best p
osition: 938.8546025977195, 17.956438697944105, 28.659772340610395
```



```
Iteration Number: 3, best objective value: 9211181746.259613, best p
osition: 937.93214298572, 17.047968670182126, 27.751073183205726
```



```
Iteration Number: 4, best objective value: 9210831407.021078, best p
osition: 936.8657629882758, 15.974150863584203, 26.677411841349823
```



```
Iteration Number: 5, best objective value: 9210365125.105104, best p
osition: 935.4616174007921, 14.569950465632358, 25.273212591939558
```

We ran the optimization on each market. we plotted the last market `11` optimization process, each point representing a firefly and we can see that at iteration `3` all the fireflies got attracted to most bright firefly. when we look on the MSE accuracy we can see its hight but it is because we dont have enough data. we sure that with more data the accuracy will be much better. We can see that our optimization technique works pretty good.

Result

As we presented in the class, our final output should be a house price according to each person/investor requirements and characteristics. We going to simulate a person characteristics anf then:

1) Find the matching market he belongs

2) Calculate the predict price of a house according to his requirements

In [420]:

```
person_df = pd.read_csv("./person.csv")
person_df
```

Out[420]:

	geo_id	sale_date	acutal_sale_price	sale_value	essence	sale_percentage	city	constr
0	006495-0231-000-00	23/01/2018	2,150,000	2,150,000	cottage tori	1	kiryat ono	

Our person demands will be:

- price: 2,150,000
- essence: cottage tori
- square meters: 150

and his social-economic index is 0.303504. lets run the data engineering for finding his maching market:

In [421]:

```
locale.setlocale( locale.LC_ALL, 'en_US.UTF-8' )
person_df['acutal_sale_price'] = person_df['acutal_sale_price'].apply(lambda x:
locale.atoi(x))
person_df['sale_value'] = person_df['sale_value'].apply(lambda x: locale.atoi(x
))
data.dtypes
person_clustering_data = person_df.copy()
person_clustering_data = person_clustering_data.drop('sale_date', axis=1)
person_clustering_data = person_clustering_data.drop('city', axis=1)
person_essence_dummies = pd.get_dummies(data['essence'])
person_construction_year_dummies = pd.get_dummies(data['construction_year'])
person_rooms_dummies = pd.get_dummies(data['n_rooms'])
person_clustering_data = person_clustering_data.drop('se', axis=1)
person_clustering_data = person_clustering_data.drop('essence', axis=1)
person_clustering_data = person_clustering_data.drop('construction_year', axis=1
)
person_clustering_data = person_clustering_data.drop('n_rooms', axis=1)
person_clustering_data = pd.concat([person_clustering_data, person_construction_
year_dummies, person_essence_dummies, person_rooms_dummies], axis=1)
person_clustering_data = person_clustering_data.set_index('geo_id')
person_clustering_data = person_clustering_data.head(1)
person_clustering_data
```

Out[421]:

	acutal_sale_price	sale_value	sale_percentage	square_meters	1940	1950	1954	19
geo_id								
006495-0231-000-00	2150000.0	2150000.0	1.0	150.0	0	0	0	

1 rows × 82 columns

Now we can run the the clustering algorithm to find his most fitted market

In [422]:

```
print("Person matching market: %s" % str(kmeans_best_estimator.predict(person_cl
ustering_data)[0]))
```

```
Person matching market: 3
```

We got that his matching market number 3. Now lets run our regression to get the correct price to his requirements and characteristics.

In [430]:

```
person_svr_data = person_df.copy()
person_svr_data = person_svr_data.drop('sale_date', axis=1)
person_svr_data = person_svr_data.drop('city', axis=1)
person_essence_dummies = pd.get_dummies(data['essence'])
person_construction_year_dummies = pd.get_dummies(data['construction_year'])
person_rooms_dummies = pd.get_dummies(data['n_rooms'])
person_svr_data = person_svr_data.drop('essence', axis=1)
person_svr_data = person_svr_data.drop('construction_year', axis=1)
person_svr_data = person_svr_data.drop('n_rooms', axis=1)
person_svr_data = pd.concat([person_svr_data, person_construction_year_dummies,
person_essence_dummies, person_rooms_dummies], axis=1)
person_svr_data = person_svr_data.set_index('geo_id')
person_svr_data = person_svr_data.drop(['acutal_sale_price', 'sale_value'], axis
=1)
person_svr_data = person_svr_data.head(1)

# predict price
model = models[kmeans_best_estimator.predict(person_clustering_data)[0]]['model'
]
print("We got that according to our person and his characteristics the house pri
ce should be %s" % str(int(model.predict(person_svr_data)[0])))
```

```
We got that according to our person and his characteristics the hous
e price should be 2225001
```

Conclusions

In this notebook we presented a POC how to match people/investors a hpuse price according their requirements and characteristics. We use clustering for creating different markets according to house quality characteristics. Then we used SVR to calculate hedonic price of each house. we used house deals data from the Israeli Tax Authority and Central Bureau of Statistics. futher analysis can be made, we believe that with more data the results can be much better, we struggled with exporting data due to some limitions authentications.

References:

- "Hartigan’s Method: k-means Clustering without Voronoi" - http://proceedings.mlr.press/v9/telgarsky10a/telgarsky10a.pdf (http://proceedings.mlr.press/v9/telgarsky10a/telgarsky10a.pdf)
- "Firefly Algorithms for Multimodal Optimization" - https://arxiv.org/pdf/1003.1466.pdf (https://arxiv.org/pdf/1003.1466.pdf)
- "Measuring Local House Price Movements in Israel And Estimating Price Elasticity" - https://www.ottawagroup.org/Ottawa/ottawagroup.nsf/4a256353001af3ed4b2562bb00121564/b1ab2e63 %20Doron%20Sayag%20Measuring%20Local%20House%20Price%20Movements%20in%20Israel%2((https://www.ottawagroup.org/Ottawa/ottawagroup.nsf/4a256353001af3ed4b2562bb00121564/b1ab2e6 %20Doron%20Sayag%20Measuring%20Local%20House%20Price%20Movements%20in%20Israel%2(