Yaonan Zhong
Sep 30, 2015

Problem: Homes like this
A common query in real-estate apps is “home like this.” Given a home, a user wants to find listings that are similar in terms of location and home characteristics (e.g., living area, number of bedrooms).

Part 1: Proximity search
Write a function or class that accepts a home as input, and returns its *n* most similar listings based on 10,000 data points from `generate_data.generate_datum`. The schema of a home is

```
Home = namedtuple(Home',
                  ['num_bedrooms', 'num_bathrooms', 'living_area', 'lat', 'lon',
                   'exterior_stories', 'pool', 'dwelling_type'])
```

Solution
1. Definition of similarity measure
I use linear regression to construct the similarity model between two homes. The proposed formula here is based on the home's attributes and their weights.

$$Similarity(p,\ q)\ =\ w_1 * factor_1 + w_2 * factor_2 + ... + w_n * factor_n$$

The available factors in this program include: location_factor, bedroom_factor, bathroom_factor, dwelling_type_factor, living_area_factor, pool_factor, list_price_factor.

The definitions of the above factors are given in the module `factors.py`. This module is highly extensible and reusable. A user can easily insert new factors into it. Also, the similarity formula can be customized with combinations of different factors and weights by the user, which makes the program flexible. For example, a user may mainly consider dwelling type and location, while another may consider number of bedrooms, living area and pool.

w1~wn are the weights of corresponding factors with range 0~1, and sum(w1~wn) = 1.

The range of each factor is 0~1, and the range of similarity is 0~1. The higher the sum of weighted factors, the higher the similarity between two homes.

2. To run this program
The module `proximity_search.py` define a class that accepts a home as input, and returns its *n* most similar listings based on m randomly generated homes. The default m is 10,000.

To run this program: `python proximity_search.py`

3. Testing result

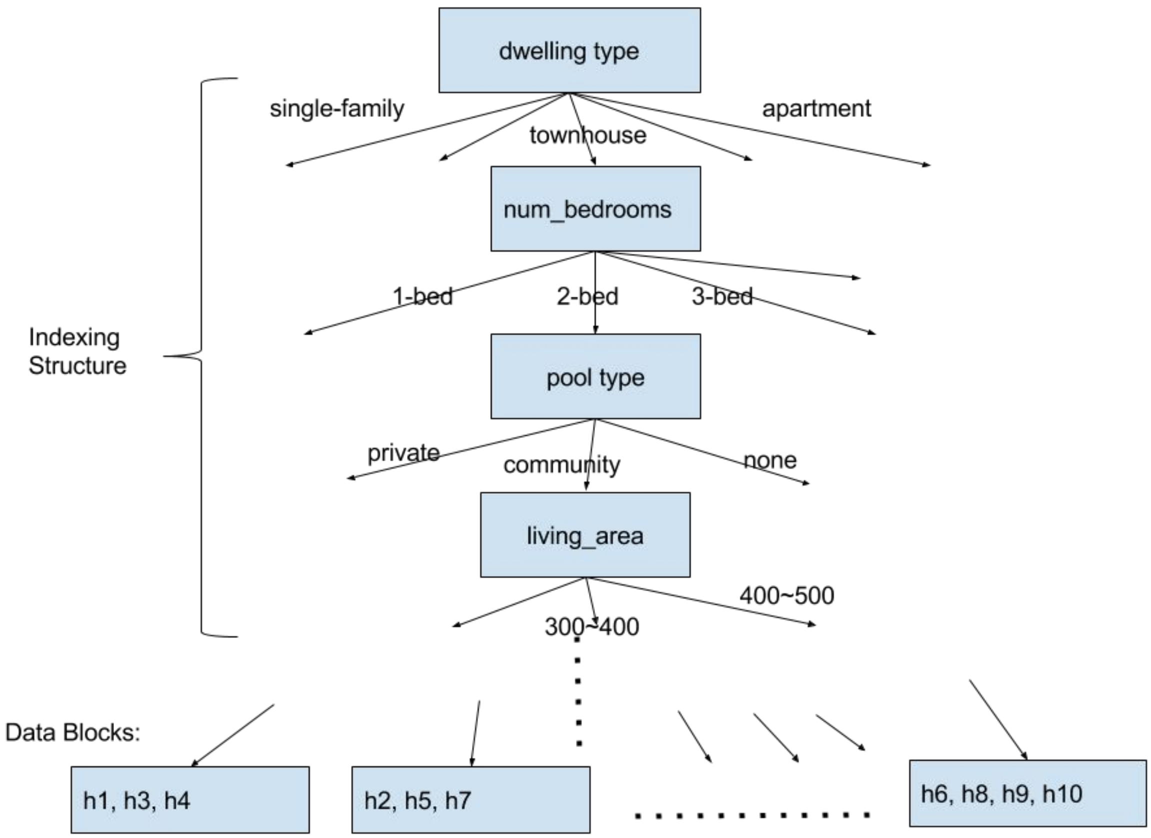
The following table shows the 10-most-similar homes testing result. The first row is the input home, and the rest are similar homes returned by the program.

num_bedrooms	num_bathrooms	living_area	lat	lon	exterior_stories	pool	dwelling_type	list_date	list_price	close_date	close_price
1	4	3480	33.133	-112.195	1	community	loft	12/24/2003	470345	1/9/2004	451836.5951
1	4	3354	33.145	-112.158	1	community	loft	7/27/2001	326423	12/5/2001	305667.4495
1	4	1199	33.151	-112.439	2	community	loft	5/29/2008	304734	8/21/2008	272971.6487
1	4	2700	33.191	-112.442	3	community	loft	10/22/2002	431592	12/29/2002	452885.7334
1	4	4847	33.262	-112.417	1	community	loft	9/11/2000	119144	12/13/2000	122364.6594
1	4	4116	33.18	-112.48	2	community	loft	2/2/2013	237608	5/31/2013	230163.1642
1	4	1061	33.103	-112.518	3	community	loft	10/12/2001	240911	3/5/2002	244877.3806
1	3	4519	33.113	-112.283	2	community	loft	6/11/2010	365170	9/28/2010	342571.1802
1	3	3684	33.099	-112.306	1	community	loft	3/28/2007	250392	7/16/2007	254720.9602
1	3	3813	33.181	-112.088	2	community	loft	8/10/2011	300062	8/18/2011	304487.491
1	4	1103	33.453	-112.392	2	community	loft	1/2/2001	232410	3/24/2001	197602.1847

Part 2: Productionizing

Suppose we are developing a production system to answer the query above, and we are constantly ingesting `Listing` data.

- How would you persist the data?
  We can store the Listing data in a decision-tree-like structure or multi-level hashtable as the following figure shows:



- What are some optimizations to make sure the query returns quickly, and how does it depend on the way data is persisted?
  In order to find out the similar homes quickly, we can pre-filter data points using the above proposed indexing structure in persisting data. It can quickly narrow down the search range of similar homes regarding of the factors a user is interested in. Thus reduce the computation time.

- How would you change your approach if the number of data points increases by 10x? 100x? 1000x?
  1. Use scientific computing libraries, such as Numpy for vectorized computation
  2. Use Python multiprocessing module
  3. Use distributed computing, such as running code at multiple machines synchronously

  Actually this code is designed with multiprocessing in mind. Since the computing of similarity between two homes is independent in the program using method `compute_factors(p, q).` We can divide the dataset into multiple batches and create multiple processes. The method `compute_similarity(self, home)` can be updated like:

```
import multiprocessing as mp

def compute_similarity_mp(self, home, num_proc):
    pool = mp.Pool(num_proc)
    unnormalized_factors = [pool.apply_async(self.compute_factors,
(home, other)) for other in self.homes]
    ...
    ...
    return similarity
```

(However, there is a pickle issue we need to solve when applying multiprocess on instance method.)