Where is the best place in NYC to run my new business ?

Marcelo Rodrigues de Oliveira

April 12, 2019

Abstract

The main objective of this project is to create a solution that get location data from New York City and find some recommendation of where are the best places to run a new business, according to density of business, visitation in the area and users rating for these categories in the neighborhoods.

Introduction

Business problem

Before we can answer the main question:

> Which is the best neighborhood in NYC to run my new business ?

we should ask more questions:

- Which neighborhoods receive more visitors/customers ?
- Which neighborhoods have the lower density of business like of yours ?
- Which neighborhoods have worst average rates from customers for this business category ?

Of course, there are lots of questions that, for example, an entrepreneur could make before it start a new business, but we will work to find the best answers, limiting our work with this scope.

Relevance and audience

This work can serve as basis for development of models that find similarities between samples of data by comparing a group of features together.
The audience for the results of this project is everyone that are analyzing neighborhoods and venues data looking for insights. The results will be presented in a straight and clear manner, so anyone that is interested in this subject like students, entrepreneurs, business analysts and others can understand them and get some value, not only number-crunchers.

Methodology

The basic ideas underlying this solution are:

1. Users must choose a business category based on Google API venues types. In our particular case, we use *cafe* as our chosen venue category.
2. We will build a dataset to be the input of machine learning model with the following columns:
    - Neighborhoods with corresponding numeric index, in order to be used on machine learning algorithms.
    - Count of business for selected category for each neighborhood and record it in the VENUE_COUNT column.
    - Count of user ratings for all business for each neighborhood for this category and record it in the USER_COUNT column. We will use this value as a proxy for visitation in the neighborhoods.

- Average rating for all business in this category for each neighborhood and record it in CATEGORY_AVG column.

Before we use this dataset as input for chosen machine learning algorithm, we need explore them and make all required preparation.

Data

The most of data needed to answer the question, we will get from Google API. In order to create a solution for that, we need information of venues based on geographical data latitude, longitude, ratings for that venues, number of visitors. For the *USER_COUNT* variable,, we will use the number of ratings posted about venues as a proxy for the number of visitors in the neighborhoods, so we can obtain it from Google API.

Data sources

We get data from the following sources:

- New York City Neighborhood Names: It is a json file and it was downloaded from:

  https://geo.nyu.edu/catalog/nyu_2451_34572

- Venues Information: I get them from Google Places API after i run out my free access to FOURSQUARE API. Both APIs supply equivalent information as json data, with some differences in structure between them. But, the information is in essence the same.

Data cleansing and transformation

In order to get a dataset with the following structure:

- Neighborhood name
- Number of venues of a chosen category (VENUE_COUNT)
- Number of users in the neighborhood (USER_COUNT)
- Average rating for venue category (CATEGORY_AVG)

We need execute the following steps:

1. Read the list of neighborhood data from "Neighborhood Names" json file
2. Search the API to get venues for each neighborhood
3. Transverse API response to select attributes of interest:
   - Venue name
   - Venue location

- Venue average rating
- Number of ratings

4. Build a dataset with all of them plus the attributes of Neighborhood names
5. Calculate the values VENUE_COUNT, USER_COUNT and CATEGORY_AVG aggregated by neighborhood. VENUE_COUNT is a count of venues by neighborhood. USER_COUNT is a sum of venue's total of ratings. CATEGORY_AVG is an average of the product of venue's average rating and total of ratings, i.e. a weighted average rating.

Exploratory data analysis

Beginning our analysis of data, we are showing below basic histograms of our variables. The first and second charts were made from venue level data and the last three were build for our aggregated variables: VENUE_COUNT, USER_COUNT and CATEGORY_AVG:

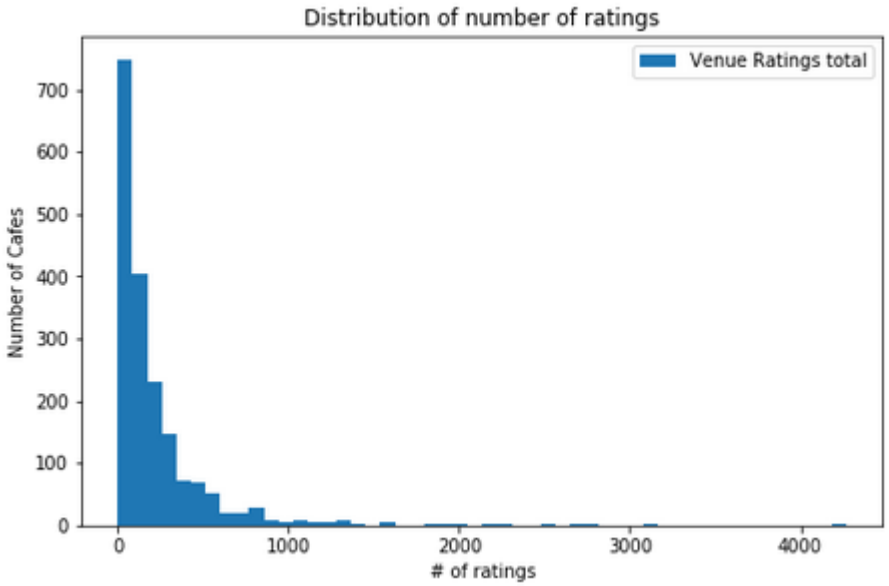

Figure 1 - Count of cafes by count of ratings

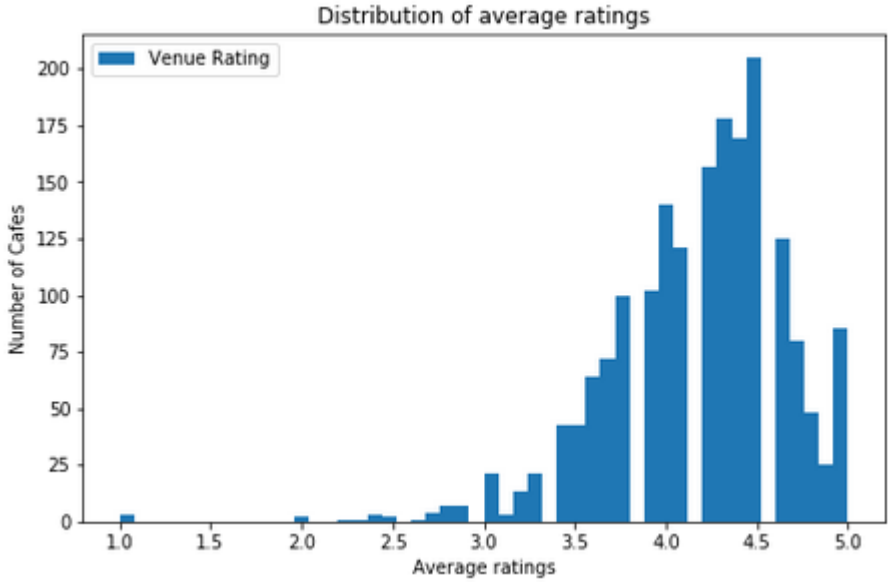

Figure 2 - Count of cafes by average ratings

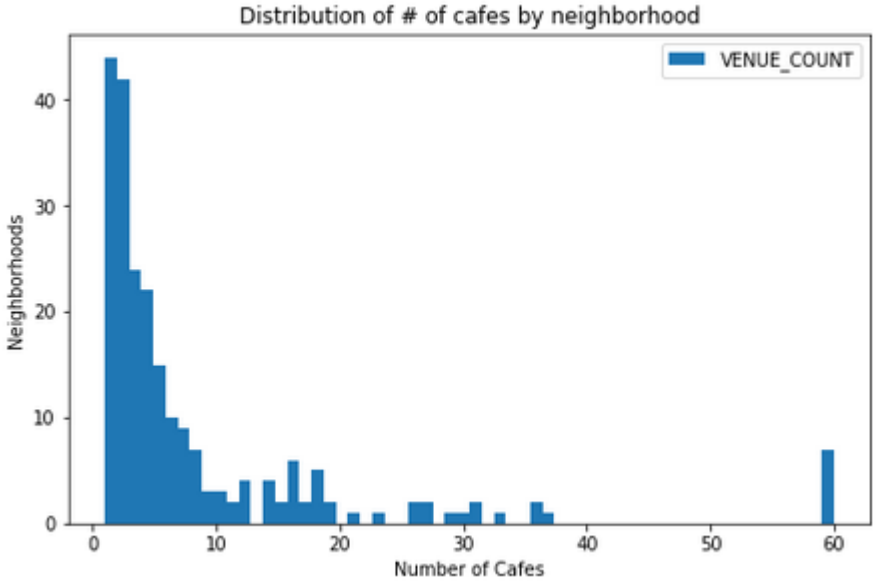

Figure 3 - Count of neighborhoods by count of cafes

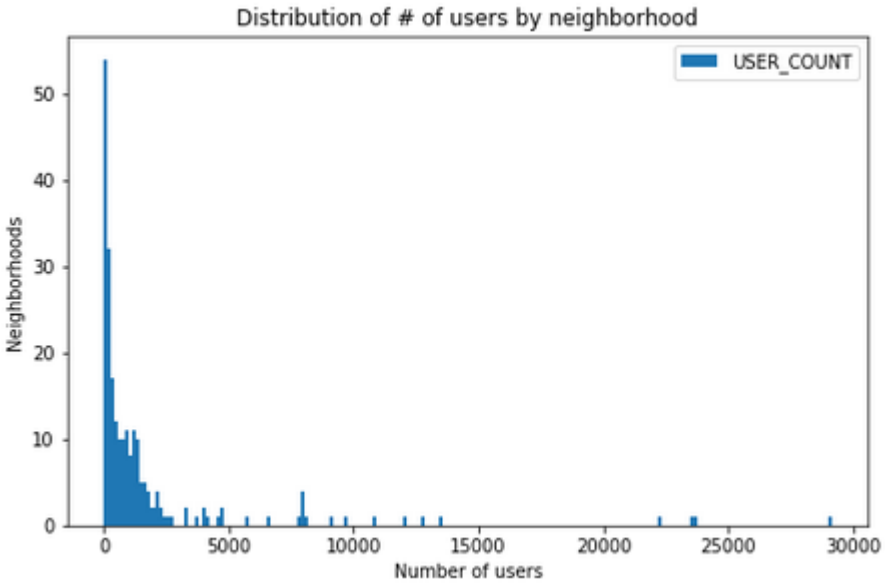

Figure 4 - Count of neighborhoods by count of users

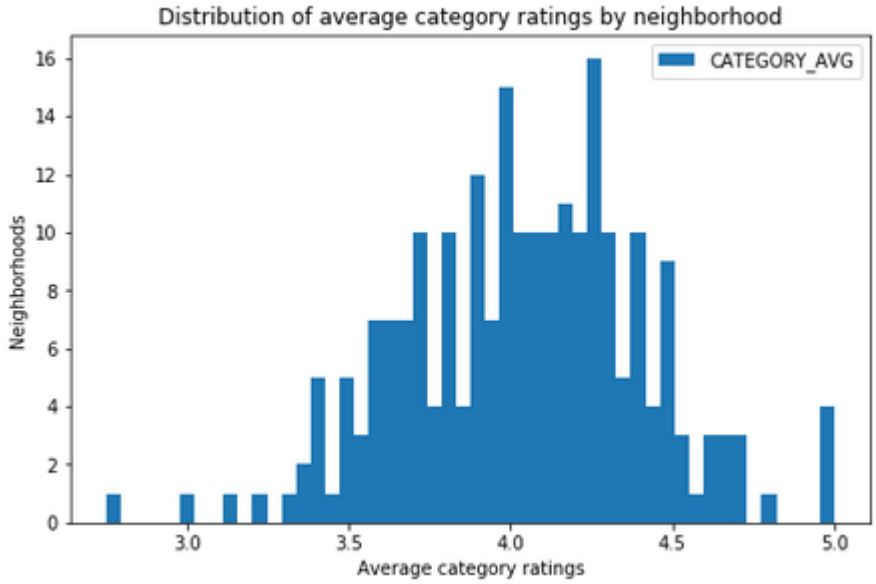

Figure 5 - Count of neighborhoods by average ratings

Below, we have summary statistics for our aggregated variables:

	VENUE_COUNT	USER_COUNT	CATEGORY_AVG
count	227	277	277
mean	8.132	1735.823	4.039
std	11.924	3833.032	0.371
min	1.000	1.000	2.754
25%	2.000	153.500	3.791
50%	4.000	563.000	4.054
75%	8.000	1381.500	4.274
max	60.000	29134.000	5.000

Data Normalization

Based on our exploration of data, we could observe that scales of variables are very different:

Variable	Minimum	Maximum
VENUE_COUNT	1	60
USER_COUNT	1	29134
CATEGORY_AVG	2.754	5.000

Considering that we will calculate differences between data points or *distances*, we need that all variables have the same scale, in order to give to all of them the same weight in the distance calculation. If data is unscaled, weights for variable with greater absolute values will be more relevant to the model. We will normalize our data dividing each variable for its maximum value. So, the maximum value will be 1 and minimum some value near 0. It is a fairly good procedure to start our model analysis.

Machine Learning algorithms selected

We will try K-Nearest Neighborhood with K = 1 (a single data point for neighborhood) and *euclidean* as metric parameter. We could try a simple distance calculation of our target point from all neighborhoods points.

Distance metrics

k-Nearest Neighborhoods is one of machine learning models based on distance. In order to find the *nearest neighborhood*, this algorithm use feature data of each data point as coordinates in a high dimension space and calculate the distance from another data points to verify if they are similar or not. There are many distance metrics, each one suitable of a particular purpose. Some of distance metric for real-valued vector spaces are showed below. The following lists the

string metric identifiers and the associated distance metric classes in scikit-learn:

Identifier	Class name	Args	Distance function
euclidean	*EuclideanDistance*	-	$\sqrt{\Sigma\left(x-y\right)^2}$
manhattan	*ManhattanDistance*	-	$\Sigma\left(\lvert x-y\rvert\right)$
chebyshev	*ChebyshevDistance*	-	$\max\left(\lvert x-y\rvert\right)$
minkowski	*MinkowskiDistance*	p	$\sqrt[p]{\Sigma\left(\lvert x-y\rvert\right)^p}$
wminkowski	*WMinkowskiDistance*	p, w	$\sqrt[p]{\Sigma\left(\lvert w*(x-y)\,\rvert\right)^p}$
seuclidean	*SEuclideanDistance*	V	$\sqrt{\frac{\Sigma(x-y)^2}{V}}$
mahalanobis	*MahalanobisDistance*	V or VI	$\sqrt{(x-y)^T * V^{-1} * (x-y)}$

Extracted from sklearn.neighbors.DistanceMetric on scikit-learn 0.20.3 documentation

Nearest Neighbor Algorithms

Extracted from 1.6.4. Nearest Neighbor Algorithms on scikit-learn 0.20.3 documentation

Brute Force

Fast computation of nearest neighbors is an active area of research in machine learning. The most simple neighbor search implementation involves the brute-force computation of distances between all pairs of points in the dataset. Efficient brute-force neighbors searches can be very competitive for small data samples. However, as the number of samples N grows, the brute-force approach quickly becomes infeasible. In the classes within sklearn.neighbors, brute-force neighbors searches are specified using the keyword algorithm = 'brute', and are computed using the routines available in sklearn.metrics.pairwise.

K-D Tree

To address the computational inefficiencies of the brute-force approach, a variety of tree-based data structures have been invented. In general, these structures attempt to reduce the required number of distance calculations by efficiently encoding aggregate distance information for the sample. The basic idea is that if point A is very distant from point B, and point B is very close to point C, then we know that points A and C are very distant, without having to explicitly calculate their distance. In this way, the computational cost of a nearest neighbors search can be reduced. This is a significant improvement over brute-force for large N. An early approach to taking advantage of this aggregate information was the KD tree data structure (short for K-dimensional tree), which generalizes two-dimensional Quad-trees and 3-dimensional Oct-trees to an arbitrary number of dimensions. The KD tree is a binary tree structure which

recursively partitions the parameter space along the data axes, dividing it into nested orthotropic regions into which data points are filed. The construction of a KD tree is very fast: because partitioning is performed only along the data axes, no D-dimensional distances need to be computed. Though the KD tree approach is very fast for low-dimensional (D less than 20) neighbors searches, it becomes inefficient as D grows very large.

Ball Tree

To address the inefficiencies of KD Trees in higher dimensions, the ball tree data structure was developed. Where KD trees partition data along Cartesian axes, ball trees partition data in a series of nesting hyper-spheres. This makes tree construction more costly than that of the KD tree, but results in a data structure which can be very efficient on highly structured data, even in very high dimensions. A ball tree recursively divides the data into nodes defined by a centroid C and radius r, such that each point in the node lies within the hyper-sphere defined by r and C. The number of candidate points for a neighbor search is reduced through use of the triangle inequality:

$$|x + y| \leq |x| + |y|$$

With this setup, a single distance calculation between a test point and the centroid is sufficient to determine a lower and upper bound on the distance to all points within the node. Because of the spherical geometry of the ball tree nodes, it can out-perform a KD-tree in high dimensions, though the actual performance is highly dependent on the structure of the training data. In scikit-learn, ball-tree-based neighbors searches are specified using the keyword algorithm = 'ball_tree', and are computed using the class sklearn.neighbors.BallTree. Alternatively, the user can work with the BallTree class directly.

Choice of Nearest Neighbors Algorithm

The optimal algorithm for a given dataset is a complicated choice, and depends on a number of factors:

1. Number of samples N (i.e. n_samples)
2. Dimensionality D (i.e. n_features).
3. Brute force query time grows as O[DN]
4. Ball tree query time grows as approximately O[D log(N)]
5. KD tree query time changes with D in a way that is difficult to precisely characterise.

For small D (less than 20 or so) the cost is approximately O[D log(N)], and the KD tree query can be very efficient. For larger

D, the cost increases to nearly O[DN], and the overhead due to the tree structure can lead to queries which are slower than brute force. For small data sets (N less than 30 or so), log(N) is comparable to N, and brute force algorithms can be more efficient than a tree-based approach. Both KDTree and BallTree address this through providing a leaf size parameter: this controls the number of samples at which a query switches to brute-force. This allows both algorithms to approach the efficiency of a brute-force computation for small N.

For more information about Nearest Neighbors Algorithms, see *1.6.4. Nearest Neighbors Algorithms on scikit-learn 0.20.3 documentation*

Results

Before we run our model, let's check how it was created on Jupiter Notebook:

```
k=1

# run k-Nearest Neighborhood classifier
neigh1 = KNeighborsClassifier(
            n_neighbors=k,
            metric='euclidean',
            algorithm='brute',
            p=2
            ).fit(X, y)

# check cluster labels generated for each row in the dataframe
neigh1
```

```
KNeighborsClassifier(algorithm='brute', leaf_size=30, metric='euclidean',
           metric_params=None, n_jobs=None, n_neighbors=1, p=2,
           weights='uniform')
```

Figure 6 - k-NN model creation on Jupyter Notebook

At first, let's give some words about our target of prediction. We want to put our new *Cafe* in a place where:

- There are the least number of competitor, i.e. VENUE_COUNT near to 0.
- There are a lot of people walking around, i.e. USER_COUNT near to 1.
- The customers in the area are not happy with this business category, i.e. CATEGORY_AVG near to 0.

Considering that we have 3 normalized real-valued variables, we can show them as 3D scatter plot, as following:

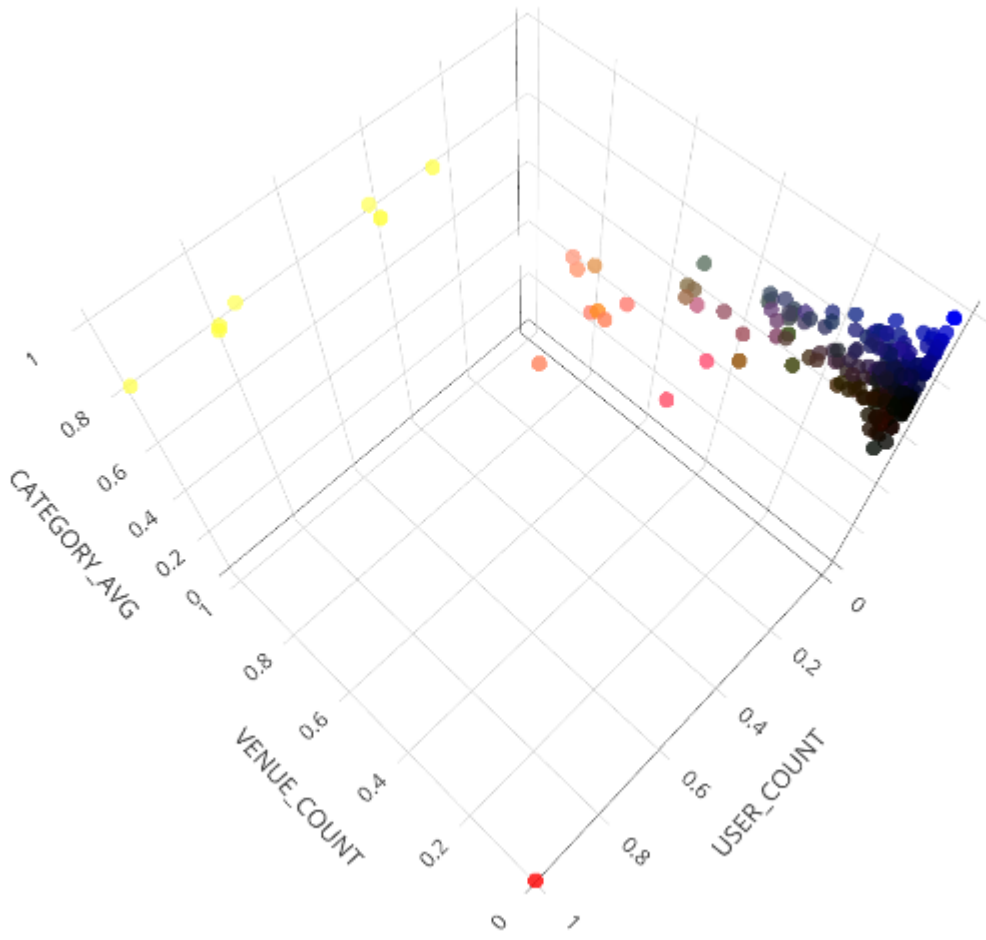

Figure 7 - 3D scatter plot of independent variables

We can see here our independent variables in 3D with our target point. In the first run of our models, we get the following outputs:

```
X_test = np.array([[0, 1, 0]])
yhat = neigh1.predict(X_test)
yhat
```

```
array([72])
```

```
neighborhood_venues.iloc[yhat[0]]
```

```
Neighborhood    Flatiron
VENUE_COUNT           18
USER_COUNT         10808
CATEGORY_AVG     4.27155
Name: 72, dtype: object
```

Flatiron has few coffee shops and lots of people walking around !!!

Figure 8 - Case 1 k-NN prediction

If we look at our 3D plot, we can see that the point with a *near red* color is the nearest neighbor. It is Flatiron. Exploring this 3D scatter plot on Jupyter notebook, it can be seen labels of all data points. In this prediction, all variables have the same importance or weight.

If we want to put more weight on USER_COUNT variable, we could do it as following:

```
X[:,1] = X[:,1]*2
```

```
k=1

# run k-Nearest Neighborhood classifier
neigh1 = KNeighborsClassifier(
            n_neighbors=k,
            metric='euclidean',
            algorithm='brute',
            p=2
            ).fit(X, y)

# check cluster labels generated for each row in the dataframe
neigh1
```

```
KNeighborsClassifier(algorithm='brute', leaf_size=30, metric='euclidean',
           metric_params=None, n_jobs=None, n_neighbors=1, p=2,
           weights='uniform')
```

```
X_test = np.array([[0, 2, 0]])
yhat = neigh1.predict(X_test)
yhat
```

```
array([185])
```

```
neighborhood_venues.iloc[yhat[0]]
```

```
Neighborhood       Soho
VENUE_COUNT          60
USER_COUNT        29134
CATEGORY_AVG    4.18217
Name: 185, dtype: object
```

Soho has much more people than Flatiron, but much more coffee shops to !!!

Figure 9 - Case 2 k-NN prediction

The new prediction was **Soho** that has the maximum number of users in the data. Looking at the new 3D plot, it will be clear:

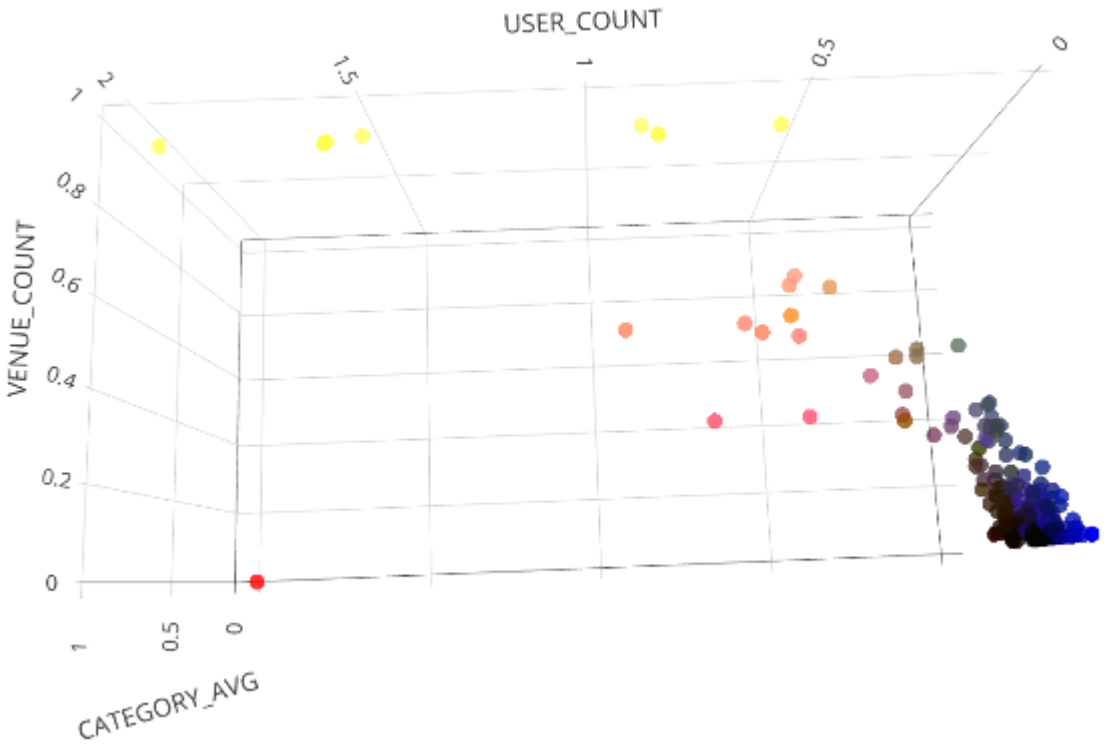

Figure 10 - Case 2 plot (Number of users with weight of 2)

Now we will make a prediction putting more weight on CATEGORY_AVG variable:

```
X[:,2] = X[:,2]*2
```

```
k=1

# run k-Nearest Neighborhood classifier
neigh1 = KNeighborsClassifier(
            n_neighbors=k,
            metric='euclidean',
            algorithm='brute',
            p=2
            ).fit(X, y)

# check cluster labels generated for each row in the dataframe
neigh1
```

```
KNeighborsClassifier(algorithm='brute', leaf_size=30, metric='euclidean',
           metric_params=None, n_jobs=None, n_neighbors=1, p=2,
           weights='uniform')
```

```
X_test = np.array([[0, 1, 0]]) # Maximum for USER_COUNT is 1 again.
yhat = neigh1.predict(X_test)  #  CATEGORY_AVG remains 0
yhat
```

```
array([201])
```

```
neighborhood_venues.iloc[yhat[0]]
```

```
Neighborhood     Travis
VENUE_COUNT           2
USER_COUNT           24
CATEGORY_AVG    2.75417
Name: 201, dtype: object
```

Travis has one of the least average ratings for coffee shops

Figure 11 - Case 3 k-NN prediction

The new prediction was **Travis** that has the minimum average ratings in the data. Looking at the plot:

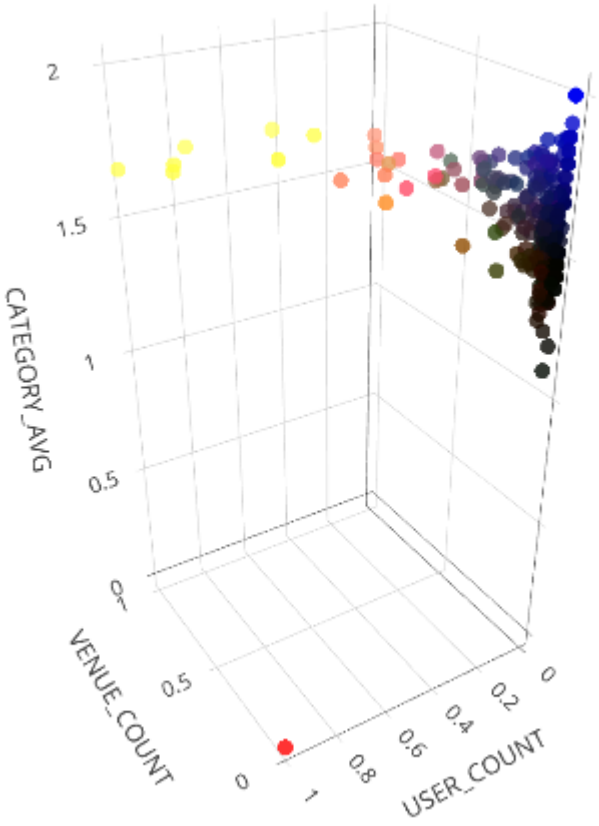

Figure 12 - Case 3 plot (Average ratings with weight of 2)

We now can see our predicted variables on the map:

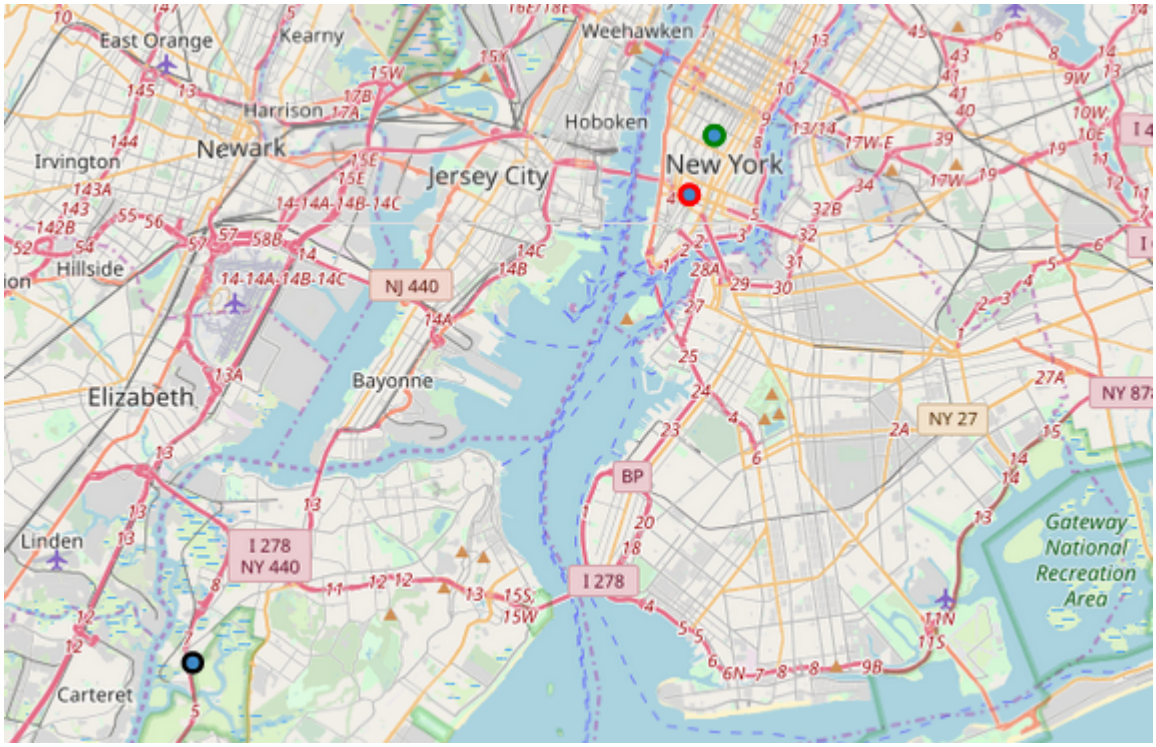

Figure 13 - New York City - Travis, Soho and Flatiron (Black, Red and Green)

Discussion

We can see as explained before, that model and its prediction are straighforward and intuitive. The predictions are consistent with data. We could talk the following about the findings:

> *If you want low competition go to Travis, but how many people there wants a cup of coffee ? If you go to Flatiron or Soho, be prepared to work hard because a lot of people need a good coffee. In Soho, you should be prepared for a "Coffee War", because there are lots of good coffee shops (average ratings in neighborhood is 4.18 out of 5.*

If we put price per square foot in our model, and tried to find the cheaper neighborhood maybe Flatiron would not be chosen at first. As it could be checked, Flatiron is not *fairly cheap* !

Conclusion

This work had some assumptions and a limited scope. So, it can be expanded in many directions. It would be good trying another machine learning algorithms. I think that get more data and create new models with additional variables are always welcome. For those that have good knowledge in scientific methods and statistics could help a lot finding flaws and proposing enhancements.