Multiple Testing Final Project

March 20, 2019

Huan Liang and Julian Waugh

1 Introduction

We are studying the restaurant data set and examining the clustering of rat violations and the probability of a restaurant getting inspected randomly given that its neighbors were inspected recently.

1.1 Vermin

The first aspect of our group project focuses on analyzing whether the vermin infestation status of neighboring restaurants influences the vermin infestation status of a given restaurant. We believe that vermin are more likely to infest nearby restaurants of an already infested restaurant rather than restaurants that are far away. A cluster of nearby restaurants could share unhygienic alleyways and trash dumpsters, so we thought that it would not be difficult for rats spread within the alley to infest neighbor restaurants. If we find a relation, then the management of nestaurants near an infested restaurant should be proactive in trying to avoid an infestation of their own.

1.2 Inspections

We next focus on determining whether patterns exist in the times of unscheduled inspections. If a restaurant could predict when it was going to be inspected, it could clean up ahead of time and have a better chance of passing the inspection. We consider the following variables: 1) month - it is likely that San Francisco issues more inspections during higher tourism months, so time is important, 2) time since last unscheduled inspection - a restaurant is probably more likely to be randomly inspected if it has been a long time since it was last inspected and 3) the number of neighboring restaurants who have been inspected in the last month - it could be that the city targets certain areas in its planning of inspections, so a restaurant might be able to pay attention to its neighbors and so determine the chances of inspection. We also believe that this last variable is the most interesting because the first two are pretty intuitive, but this one shows clear evidence of a city wide inspection policy.

1.3 Paper Outline

We outline our methods and the challenges we face on an exploratory data set, choose important threshold values for our analysis for both vermin and inspections, and then switch to a testing data set to search for the relationships we outlined above.

2 Data Cleaning

After reading in the Kaggle restaurant data, we needed to adjust the structure of the data in order to perform our analysis. First, we dropped any data points without latitude and longitude or latitude and longitude information. Then, we generated our neighborhoods. To do this, we sliced our dataset so that each unique restaurant appears only once and proceeded to use K means clustering to group our restaurants into ten clusters. We do this in order to have smaller subsets that we can work with that are generated in a way where the signal is not lost. We use these smaller data sets to make computation faster and for data splitting.

3 Analysis - Exploratory Phase

First, we read in the data (since k-Means creates slightly different clusters each time, we saved the clustered data so we can work with the same data set each time).

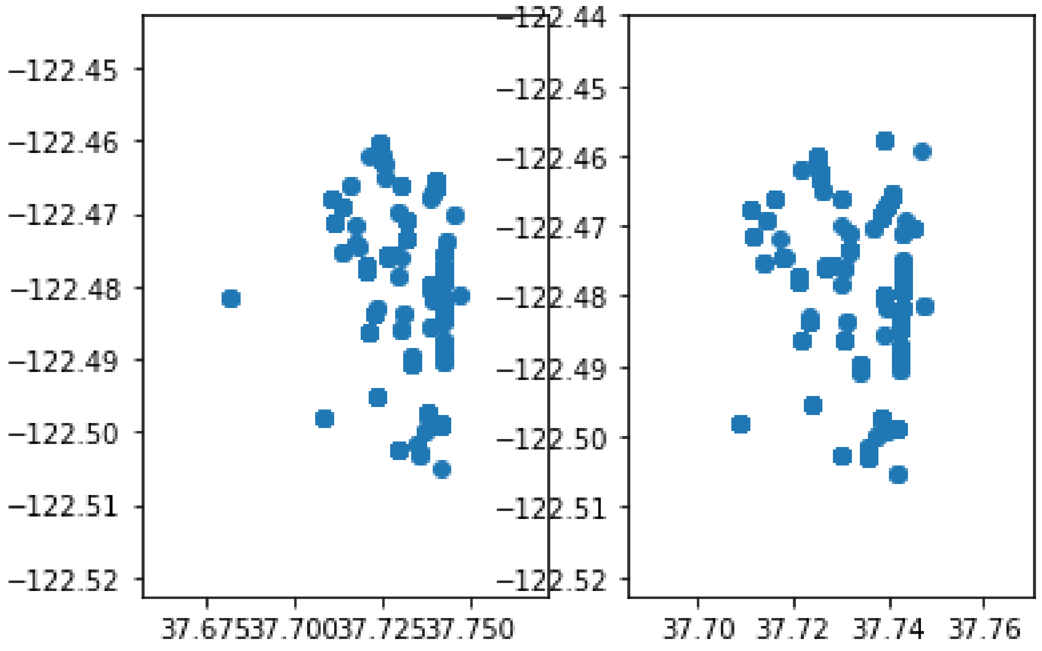
Then, we create an exploratory and a testing data set. The exploratory set is cluster 6 in 2016 and the test set is cluster 6 in 2017. We plot both data sets side by side. The overall neighborhood patterns remain the same across the years.

```
In [2]: #read in the data
        df = pd.read_csv("neighborhood_clusters.csv")
        neighborhood_2017 = get_neighborhood(2017, 6, df)
        neighborhood_2016 = get_neighborhood(2016, 6, df)

        #plotting
        x_2017 = neighborhood_2017["business_latitude"]
        y_2017 = neighborhood_2017["business_longitude"]
        x_2016 = neighborhood_2016["business_latitude"]
        y_2016 = neighborhood_2016["business_longitude"]

        plt.subplot(121)
        plt.scatter(x_2017,y_2017)
        plt.subplot(122)
        plt.scatter(x_2016,y_2016)

        plt.show()
```



3.1 Data Aggregation

To create the data set that we can use to perform our analysis, we aggregated a few important characteristics of our cleaned data set. For each unique restaurant, we created eleven instances. Each instance corresponds with a specific month. We deleted any instances with January because we were unable to keep looking back at the previous month, e.g. December of the previous year, so we chose to cut it off at January.

3.1.1 Vermin

For vermin, we created three variables. One is called rat_now, which gives us a 1 or 0 depending on whether the restaurants had a rats infestation during that current month. We counted low, medium and high risk infestations all the same. The next one is rat_last, where it is a 1 or 0 depending on whether the restaurant had an infestation the previous month. The final one is rat_count_neighbors, which given the number of neighbors, we count how many of its neighbors had a rat infestation the previous month.

3.1.2 Inspections

For inspections, we also have three variables. Inspection_now is similar to rat_now, and is a 1 if the restaurant has a unscheduled inspection that month and a 0 otherwise. The variable time_since is the time (in months) since the previous inspection and assumes that there was an inspection in December of the previous year. When there actually was an inspection, the count resets. Finally, the inspection_count_neighbors is similar to rat_count_neighbors, as it counts the number of neighbor restaurants that had an unscheduled inspection the previous month.

3.1.3 Example

We have an example of what our data looks like below.

```
In [3]: # Create our scrambled and unscrambled data
        unscrambled_2016 = generate_neighborhood_data(neighborhood_2016, 6, save = False, scram
        unscrambled_2016.head()
```

```
Out[3]:    rat_count_neighbors  rat_last  rat_now  inspection_count_neighbors  \
        0                    0         0        0                           0
        1                    0         0        0                           1
        2                    0         0        0                           1
        3                    0         0        0                           2
        4                    0         0        0                           1

           time_since  inspection_now  month
        0           2               1      2
        1           1               0      3
        2           2               0      4
        3           3               0      5
        4           4               0      6
```

3.2 Choosing K

In building the above data set, we need to come to a decision on what k we use in the k-nearest-neighbors algorithm that maps a restaurant to k neighbors. Since we believe that looking at k vs k + 1 neighbors is very similar and could lead to correlated p values, we decide to sweep over a range of ks in our analysis. Of course, that means we need to determine the minimum and maximum value of k in the range, which is a selective inference problem since we will need to look at the data to make that decision. This is why we decided to use data splitting on the year - so that we preserve the same neighborhood structure across our exploratory and our test sets.

Thus, our procedure is as follows: 1) for both the vermin and the inspection models, we choose some function that generates p values 2) using our exploratory set, we choose a large range for k: 2 - 50. 3) Then run our test at each level and return a p value. 4) We then look at the vector of p values and see if we can narrow down the range to something smaller, because for our testing we will use a permutation test and the smaller the range of k we examine the faster the permutation test runs.

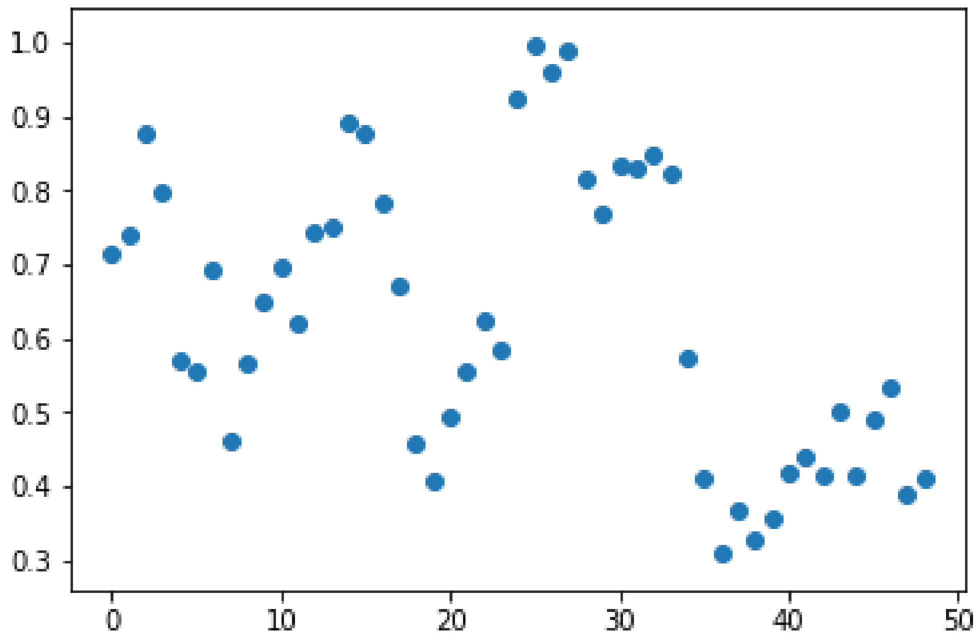
3.2.1 Vermin

Our function for generating p values is a one way ANOVA, where the two groups are months where restaurants had and did not have inspections. Then, the data associated with the groups is the number of neighbors that had inspections in the past month. If we find a signficant difference in group means, we know that a restaurant's neighbors having vermin the past month could influence whether or not the restaurant had vermin in the current month.

```
In [4]: unscrambled_pvals = get_anova(2,50,neighborhood_2016)
```

```
In [5]: #plot the p values
        x = [range(len(unscrambled_pvals))]
        y = unscrambled_pvals

        plt.close("all")
        plt.figure()
        plt.scatter(x,y)
        plt.show()
```



Taking a look at the plot of p values generated from iterating the k from 2 to 50, there appears to be a pattern. The p values decrease and then increase every 10 k's, and so, we choose the first of these oscillations. Thus, we set k_{min} to be 2 and k_{max} to be 10.

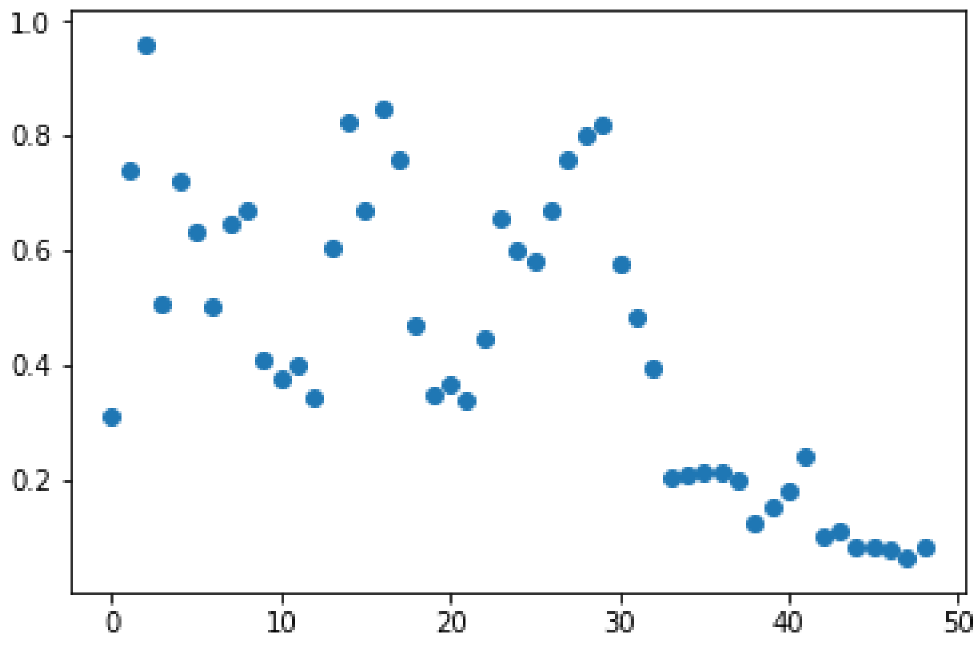
3.2.2 Inspections

We generate p values using a logistic regression model, with the months, the time since the last inspection, and the number of neighbors inspected in the past month as features. To generate p values, we use the p value associated with the neighbor count in the logistic regression model.

```
In [6]: truth = search_ks(2, 50, neighborhood_2016, False, False)

In [7]: #plot the p values
        x = [range(len(truth))]
        y = truth

        plt.close("all")
        plt.figure()
        plt.scatter(x,y)
        plt.show()
```



Above is a plot of the p values for the logistic regresion coefficient corresponding to the number of inspections seen by a restaurant's neighbors in the past month. There is a clear downward trend, although we can see oscillations as in the previous case with rats. To be as conservative as possible, we choose the largest oscialltion and fix $k_{min} = 4$ and $k_{max} = 12$.

3.3 Choosing How to Aggregate the P Values

Our method treats the above p values we find as statistics. We compute a vector of p values on the true data, then we run a permutation test where we randomly choose the nearest neighbors, as opposed to using k-n-n, and calculate p values each time. Thus, any true spatial relationships in the data are lost. We do this 100 times, each time getting a p value. Ideally, we would do this more, but the run time is pretty long just for this. Then, using our p value on the real data as a statistic and the permuted data as a null distribution, we calculate a true p value, which should give some measure of the probability of seeing the data under the null. One question that comes up is: how do we aggregate the vector of p-values produced by iterating over the range of ks into one p value. We could sum it, or we could take the inverse sum, which weights signals higher than noise. To decide, we run a few permutations with both.

3.3.1 Vermin

We begin with the vermin. We sum up the true both ways, we do 3 permutaitons of the data and sum those up both ways, then we compare. We choose 3 because this is a computationally intensive process and we are only trying to get a brief glimpse of what is going on.

```
In [8]: #This is the sum of the p-values

        truth = get_anova(2,10,neighborhood_2016, scrambled = False, random_neighbors = False)
        scrambled = []
        for i in range(3):
            l = get_anova(2, 10, neighborhood_2016, scrambled = True, random_neighbors = True)
            scrambled.append(l)

        print("..............")
        print("Regular Sum")
        print("sum of truth is " + str(np.sum(truth)))
        print("sum of scrambled data is")
        print(np.sum(scrambled, axis = 1))
        print("......................")
        print("Inverse Sum")
        print("sum of truth is " + str(np.sum(np.power(truth, -1))))
        print("sum of scrambled data is")
        print(np.sum(np.power(scrambled, -1), axis = 1))
        print("................")
```

```
...
Regular Sum
sum of truth is 5.968515127298909
sum of scrambled data is
[4.39250021 5.10987088 4.11783027]
...
Inverse Sum
sum of truth is 14.096259523667587
sum of scrambled data is
[49.23910103 24.74576933 28.6398471 ]
...
```

Since these are both pretty bad - the regular sum should be small with a signal and the inverse sum should be large with a signal, we decide to use the inverse sum for rats because in theory it should weight signals higher than noise.

3.3.2 Inspections

We repeat the same process as for vermin. Again, there does not seem to be a huge difference, so we decide to use the inverted sum.

```
In [9]: truth = search_ks(4, 12, neighborhood_2016, False, False)
        scrambled = []
        for i in range(3):
            l = search_ks(4, 12, neighborhood_2016, False, True)
            scrambled.append(l)

        print("..............")
        print("Regular Sum")
        print("sum of truth is " + str(np.sum(truth)))
        print("sum of scrambled data is")
        print(np.sum(scrambled, axis = 1))
        print("......................")
        print("Inverse Sum")
        print("sum of truth is " + str(np.sum(np.power(truth, -1))))
        print("sum of scrambled data is")
        print(np.sum(np.power(scrambled, -1), axis = 1))
        print("................")
```

```
...
Regular Sum
sum of truth is 5.418250042096478
sum of scrambled data is
[4.38691883 3.7486931  4.01632443]
...
Inverse Sum
sum of truth is 16.142427612160688
sum of scrambled data is
[ 34.86835726 114.74726475  79.15847756]
...
```

4 Analysis on Testing Data

After using the exploration data to choose k_{min} and k_{max} for both the vermin and the inspections and also to choose to sum our vector of p values as $\sum_{k=k_{min}}^{k_{max}} 1/p_k$, we ran our analysis on the testing data, which is the same cluster (6) but in 2017. We permutated the data 100 times to get a null distribution, and found p values using our get_p function. We created the final, true p value as the proportion of scrambled, inverted sums of p values that are greater than the inverted sum of the true p values.

4.1 Vermin

```
In [10]: #rats
         truth = get_anova(2, 10, neighborhood_2017, scrambled = False)
         scrambled = []
         for i in range(100):
             scrambled_pvals = get_anova(2, 10, neighborhood_2017, scrambled = True)
             scrambled.append(scrambled_pvals)

         p = get_p(truth, scrambled, True)
```

```
0.05
```

```
/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:288: RuntimeWarning: invalid value
```

We found the p value of our vermin analysis to be 0.05, which if we set α to be 0.05, makes our analysis significant. This means that in cluster 6 of 2017, there is a statistically significant differnece in the mean count of neighbors with vermin for restuarants that have vermin versus for restaurants that do not. In other words, if a restaurant's neighbor restaurants have rats in one month, then it will affect whether or not the original restaurant will have rats in the next month.

4.2 Inspections

```
In [11]: #inspection
         truth = search_ks(4, 12, neighborhood_2017, False, False)
         scrambled = []
         for i in range(100):
             l = search_ks(4, 12, neighborhood_2017, False, True)
             scrambled.append(l)

         p = get_p(truth, scrambled, True)

In [13]: p

Out[13]: 0.8
```

Our p value for our inspection analysis is not significant. Thus the number of a restaurant's neighbors that have had an unscheduled inspection one month does not affect whether the restaurant will get randomly inspected the following month.

5 Next Steps

After using 2016 data to come to a decision about how to structure our analysis, we find statistically signficant evidence of geographic clustering of vermin infestations in cluster 6 in 2017. We do not find any evidence of a relationship between neighborhood inspections and inspections for a restauarant in the month after its neighbors were inspected.

Going forward, it would be interesting to increase the geographic scope of the analysis to all of San Francisco. We avoided doing this to cut down on computation time, but it may provide insights that we are unable to find here. Additionally, it could be worthwhile to look at different years at well, or to even include multiple years in the analysis.

6 Functions

These are various functions that we used in conducting our report.

```
In [ ]: import pandas as pd
        import numpy as np
        from scipy.cluster.vq import kmeans,vq
        from scipy.spatial.distance import cdist
        import matplotlib.pyplot as plt
        import seaborn as sns
        from sklearn.cluster import KMeans
        from sklearn.neighbors import NearestNeighbors
        import statsmodels.api as sm
        from scipy import stats
```

```python
from sklearn.metrics import confusion_matrix
from scipy.stats.distributions import chi2


def get_neighborhood(year, cluster, df):
    '''
    This function takes a dataframe with all the years and restaurants
    and constrains it to just the year and cluster that are input.
    '''
    neighborhood = df[df["Year"] == year]
    neighborhood = neighborhood[neighborhood["cluster"] == cluster]
    return neighborhood

def indicator(x):
    '''
    Indicator function.
    '''
    if x > 0:
        return(1)
    else:
        return(0)

def get_restaurant_data(rest_id, neighborhood, neighbors_count,
                        rats_only = False, random_neighbors = False):
    '''
    This function takes a restaurant id, a neighborhood created by
    get_neighborhood function and number of neighbors for the restaurant id
    and returns the neighbors data for each specific id.
    '''
    rest_df = neighborhood[neighborhood["business_id"] == rest_id]

    unique_neighborhood = neighborhood.drop_duplicates(subset = ["business_id"])
    X = np.matrix(unique_neighborhood[["business_latitude", "business_longitude"]])

    #get index of restaurant
    unique_neighborhood = unique_neighborhood.reset_index(drop=True)
    rest_index = unique_neighborhood.index[unique_neighborhood
                                          ["business_id"]== rest_id].tolist()[0]

    if not random_neighbors:
        knn = NearestNeighbors(n_neighbors = neighbors_count)
        knn.fit(X)

        k_neighbors = knn.kneighbors(X[rest_index], return_distance = False)
        k_neighbors = k_neighbors[0][1:]

        neighbor_df = unique_neighborhood.iloc[k_neighbors]
        good_id = neighbor_df["business_id"]
        neighbor_df = neighborhood[neighborhood["business_id"].isin(good_id)]
    else:
        unique_neighborhood = unique_neighborhood.business_id.values
        unique_neighborhood = unique_neighborhood[unique_neighborhood != rest_index]
        good_id = np.random.choice(unique_neighborhood, neighbors_count, replace = Fals
        neighbor_df = neighborhood[neighborhood["business_id"].isin(good_id)]

    #iterate through months
    all_data = []
    counter = 0
    for month in range(2,13):
        #vermin
        neighbors_last = neighbor_df[neighbor_df["Month"] == month - 1]
        neighbors_rat = neighbors_last[(neighbors_last["violation_description"]
                                        == "High risk vermin infestation")
                                       | (neighbors_last["violation_description"]
                                          == "Moderate risk vermin infestation")
                                       | (neighbors_last["violation_description"]
                                          == "Low risk vermin infestation")]
        rat_count_neighbors = neighbors_rat.shape[0]

        rest_last = rest_df[rest_df["Month"] == month - 1]
        rest_rat_last = rest_last[(rest_last["violation_description"] ==
                                   "High risk vermin infestation")
                                  | (rest_last["violation_description"] ==
                                     "Moderate risk vermin infestation") |
                                  (rest_last["violation_description"] ==
                                   "Low risk vermin infestation")]
        rat_last = indicator(rest_rat_last.shape[0])

        rest_now = rest_df[rest_df["Month"] == month]
        rest_rat_now = rest_now[(rest_now["violation_description"]
                                 == "High risk vermin infestation")
                                | (rest_now["violation_description"]
                                   == "Moderate risk vermin infestation")
                                | (rest_now["violation_description"] ==
                                   "Low risk vermin infestation")]
        rat_now = indicator(rest_rat_now.shape[0])


        #inspections
        if not rats_only:
            neighbors_inspections = neighbors_last[neighbors_last["inspection_type"]
                                                   == "Routine - Unscheduled"]
            neighbors_inspections = neighbors_inspections.drop_duplicates(subset =
                                                                "inspection_id")
            inspection_count_neighbors = neighbors_inspections.shape[0]
```

```
            inspection_rest_now = rest_now[rest_now["inspection_type"] ==
                                                "Routine - Unscheduled"]
            inspection_now = indicator(inspection_rest_now.shape[0])

            inspection_rest_last = rest_last[rest_last["inspection_type"] ==
                                                  "Routine - Unscheduled"]
            inspection_last = indicator(inspection_rest_last.shape[0])

            if month == 2:
                if inspection_last == 1:
                    counter = 0

                else:
                    counter = 1
            counter += 1

        if not rats_only:
            data = [rat_count_neighbors, rat_last, rat_now,
                    inspection_count_neighbors, counter, inspection_now, month]
        else:
            data = [rat_count_neighbors, rat_last, rat_now, month]

        if not rats_only:
            if inspection_now == 1:
                counter = 0
        all_data.append(data)
    return(all_data)


def generate_neighborhood_data(neighborhood, neighbors_count,
                               save =True, scramble = True,
                               rats_only = False, random_neighbors = False):
    '''
    For a specific neighborhood, iterates through all of the
    restaurants and returns a dataframe with the information
    we need for our analysis.
    '''
    ids = neighborhood.business_id.unique()
    #scramble months
    if scramble:
        y = np.random.permutation(neighborhood["Month"].values)
        neighborhood["Month"] = y
        save_str = "restaurant_data_scrambled.csv"


    else:
        save_str = "restaurant_data_unscrambled.csv"
    data = []
    for id in ids:
        rest_year = get_restaurant_data(id, neighborhood, neighbors_count,
                                        rats_only, random_neighbors)
        data += rest_year

    if not rats_only:
        columns = ["rat_count_neighbors", "rat_last", "rat_now",
                   "inspection_count_neighbors", "time_since",
                   "inspection_now", "month"]
    else:
         columns = ["rat_count_neighbors", "rat_last", "rat_now", "month"]
    mydata = pd.DataFrame(data, columns = columns)
    if save:
        mydata.to_csv(save_str)
    return(mydata)


def get_anova(k_min, k_max, neighborhood, scrambled = False,
              random_neighbors = False):
    '''
    Given a min k neighbors and a max k neighbors, this iterates through the
    number of neighbors and generates the p value from the anova test by comparing
    the neighborhood data with rats and the neighborhood data without rats.
    '''
    d = []
    for i in range(k_min,k_max+1):

        unscrambled_data = generate_neighborhood_data(neighborhood, i,
                                                       save = False, scramble = False,
                                                      random_neighbors = False)
        if scrambled:

            if not random_neighbors:
                scrambled_data = generate_neighborhood_data(neighborhood,
                                                            i, save = False,
                                                            scramble = True,
                                                            random_neighbors = False)
            else:
                scrambled_data = generate_neighborhood_data(neighborhood,
                                                            i, save = False,
                                                            scramble = False,
                                                       random_neighbors = True)
            no_rats = unscrambled_data[unscrambled_data["rat_now"]
                                       == 0]["rat_count_neighbors"].values
            rats = scrambled_data[scrambled_data["rat_now"] ==
                                1]["rat_count_neighbors"].values
        else:
            no_rats = unscrambled_data[unscrambled_data["rat_now"] ==
                                     0]["rat_count_neighbors"].values
            rats = unscrambled_data[unscrambled_data["rat_now"] ==
                                  1]["rat_count_neighbors"].values

        no_rat_sum = np.mean(no_rats)
        rat_sum = np.mean(rats)


        F, p = stats.f_oneway(no_rats, rats)
        d.append(p)

    return d

def logistic_inspection_model(unscrambled, regress = True, full = True):
    '''
    This function takes an unscrambled data set and returns the fitted
    logistic model. If regress is false, then it return the a dictionary
    with keys as X and y and values as variable names.
    '''
    cat_vars = 'month'
    cat_list='var'+'_'+ cat_vars
    cat_list = pd.get_dummies(unscrambled[cat_vars], prefix = cat_vars)
    unscrambled1 = unscrambled.join(cat_list)
    unscrambled = unscrambled1
    unscrambled.columns
    nrow = unscrambled.shape[0]
    intercept = [1] * nrow
    unscrambled["intercept"] = intercept

    #index the x and y variables for inspection data
    if full:
        X_inspection = ['intercept', 'inspection_count_neighbors',
                        'time_since', 'month_3', 'month_4',
                        'month_5', 'month_6', 'month_7',
                        'month_8', 'month_9', 'month_10', 'month_11', 'month_12']
    else:
        X_inspection = ['intercept', 'month_3', 'month_4',
                        'month_5', 'month_6', 'month_7',
                        'month_8', 'month_9', 'month_10', 'month_11', 'month_12']
    X_inspection = unscrambled[X_inspection]
    y_inspection = ['inspection_now']
    y_inspection = unscrambled[y_inspection]

    if regress:
        logit_model = sm.Logit(y_inspection, X_inspection)
        result_inspection=logit_model.fit(disp = 0)
        return(result_inspection)
    else:
        rv = {}
        rv['X'] = X_inspection
        rv['y'] = y_inspection
        return(rv)

def LRT(mdl_full, mdl_small):
    '''
    The inputs are logistic regression models. Returns the
    p value from a likelihood ratio test which assumes that
    difference in the number of features between
    the models is 2.
    '''
    LL_full = mdl_full.llf
    LL_small = mdl_small.llf
    LLR = 2* (LL_full - LL_small)
    p = chi2.sf(LLR, 2)
    return(p)

def test_inspection_models(data):
    '''
    Given restaurant data, this computes a likelihood ratio test
    between a model with seasonal predictors and one without seasonal
    predictors. This functions returns the p value associated with the LRT.
    '''
    mdl_full = logistic_inspection_model(data, full = True)
    mdl_small = logistic_inspection_model(data, full = False)
    p = LRT(mdl_full, mdl_small)
    return(p)


def search_ks(k_min, k_max, neighborhood, full_model_test, random_neighbors):
    '''
    This function takes the min and max number of k neighbors
    and parameter 'full_model_test', which determines taking
    the p value from the beta of neighbors_count or the p val
    from the LRT, iterates from k_min and k_max, and returns
    a list of p values.
    '''
    #iterate through a range of k values and produce p values
    rv = []
    for i in range(k_min,k_max+1):
        data = generate_neighborhood_data(neighborhood, i,
                                save = False, scramble = False,
                                              random_neighbors = random_neighbors)
```

```
            if full_model_test:
                p = test_inspection_models(data)
            else:
                mdl = logistic_inspection_model(data)
                p = mdl.pvalues["inspection_count_neighbors"]


            rv.append(p)
        rv
        return(rv)

    def get_p(truth, all_scrambled, inverse_sum = False):
        '''
        This function takes a vector of pvalues from
        real data and a matrix of pvalues from
        scrambled data, and sums them up, with the option of summing
        1/p{i} instead (in order to give more weight to the signals).
        Then, it computes a p value by counting the number of scrambled
        sums greater than the true sum.
        '''
        if inverse_sum:
            truth = np.power(truth, -1)
            all_scrambled = np.power(all_scrambled, -1)

        p_true = np.sum(truth)
        all_scrambled = np.sum(all_scrambled, axis = 1)
        N = len(all_scrambled)

        if not inverse_sum:
            top = all_scrambled[all_scrambled < p_true]
        else:
            top = all_scrambled[all_scrambled > p_true]
        return(len(top) / N)
```

6.1 Data Cleaning Script

This is the script we used to clean the data and run the KMeans clustering. We can also provide the csv we used for the analysis, if you are interested in replicating our work.

```
In [12]: #read the data
         df = pd.read_csv("restaurant-scores-lives-standard.csv")

         #get rid of guys without latitude and longitude
         df = df.dropna(subset = ["business_latitude"])
         df = df[df["business_latitude"] != 0]

         #getting the neighborhoods
         df1 = df.drop_duplicates(subset = "business_id")
```

```
df1 = df1[["business_id", "business_latitude", "business_longitude"]]


#clustering
km = KMeans(n_clusters = 10)
X = np.matrix(df1[["business_latitude", "business_longitude"]])
km.fit(X)
df1["cluster"] = km.labels_


#merge back to the original data set
df1 = df1[["business_id", "cluster"]]
df = df.merge(df1, on = "business_id")

#get a month and a year column
df["Year"] = df.inspection_date.str[:4]
df["Month"] = df.inspection_date.str[5:7]
df["Month"]= pd.to_numeric(df["Month"])

#create a year month column
#year goes from 2016 to 2019
yr = (pd.to_numeric(df["Year"]) - 2016) * 12
YearMonth = yr + df["Month"]
df["YearMonth"] = YearMonth
```