Predictive Analytics, Fall 2016
Professor Anasse Bari, PhD
Department of Computer Science

What will the real estate market look like after the L train closure in 2019?

Team 4: Caroline Lee, Lihao Liu, Elahe Vahdani

Problem Statement and Analytics Hypothesis

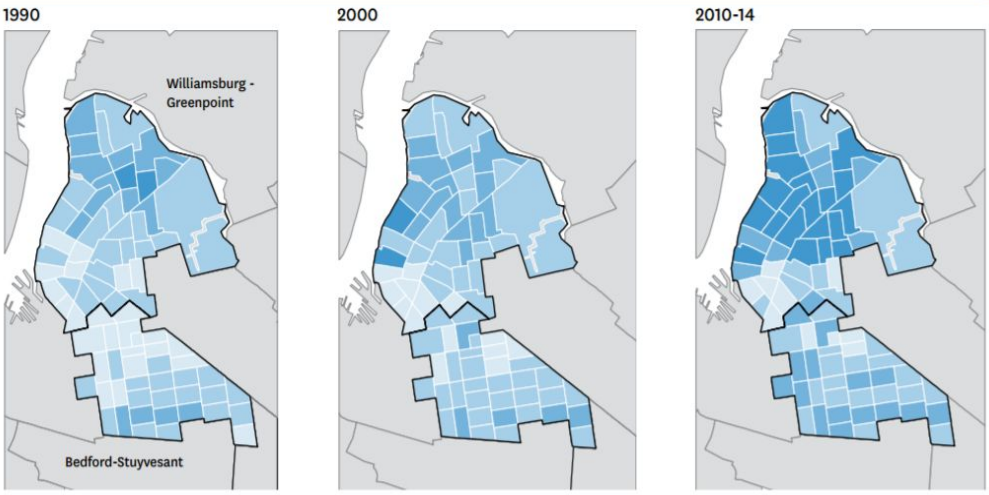
This study aims to characterize factors contributing to apartment rental prices in New York City, and to build a predictive model of rent prices following the closure of the L train line in early 2019. Suspended L train service from Brooklyn to Manhattan will likely demonstrate an increase of rental prices in areas off of other train lines that are proximal to Downtown Manhattan's commercial areas and easily accessed by public transportation. In contrast, we expect areas affected by the closure to experience a decrease in rental prices.

Business Understanding

In order to narrow down the necessary attributes used in the analysis, previous studies on New York City gentrification were examined. First, we confirmed that gentrification yields an increase in rental prices. Gentrification is a term describing cities or parts of cities, primarily in the US, that see a sharp increase in groups of stable socioeconomic status moving into areas that were not previously as expensive. Gentrification is of particular concern in New York City because it dramatically affects the cost of living. NYU Furman Center's 2016 study depicts a change in population demographics as the Williamsburg, Brooklyn area gentrifies between 1990 and 2014 the figures below. We chose 138 attributes based on their contributions to gentrification and increased rent prices: various crime indices, household income split by racial and ethnic group, culture index, transportation-related categories (average travel time to work, mode of transport, number of subway lines and stations, etc.), employment domains, educational attainment, and more.

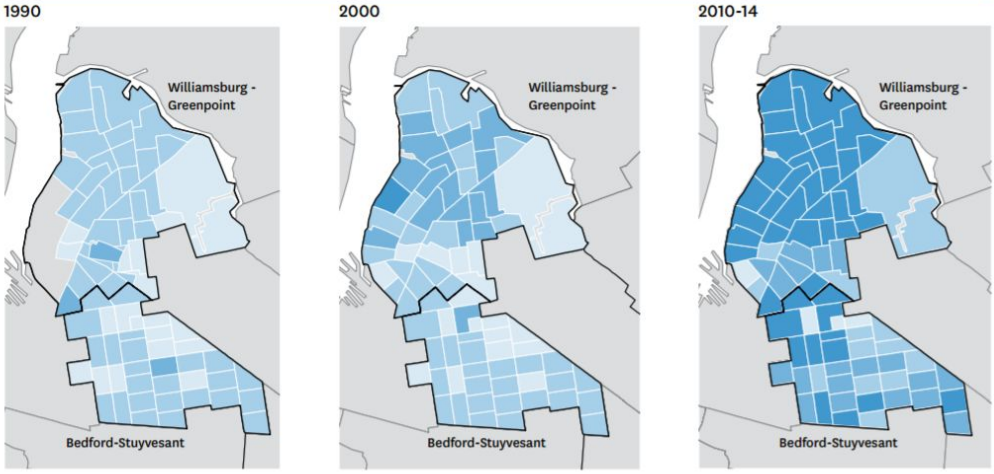
The specific aim of this project is to build a predictive model that can accurately determine median rental values and subsequent percent changes by zip code after the 2019 L train closure.

Average Household Income
- Less than $40,000
- $40,001–$60,000
- $60,001–$80,000
- Greater than $80,000

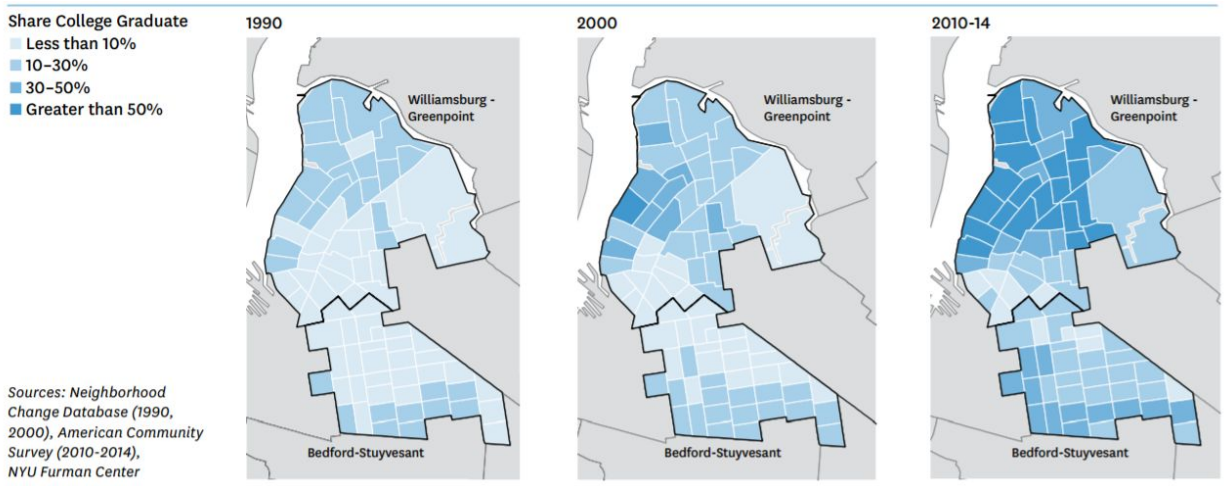


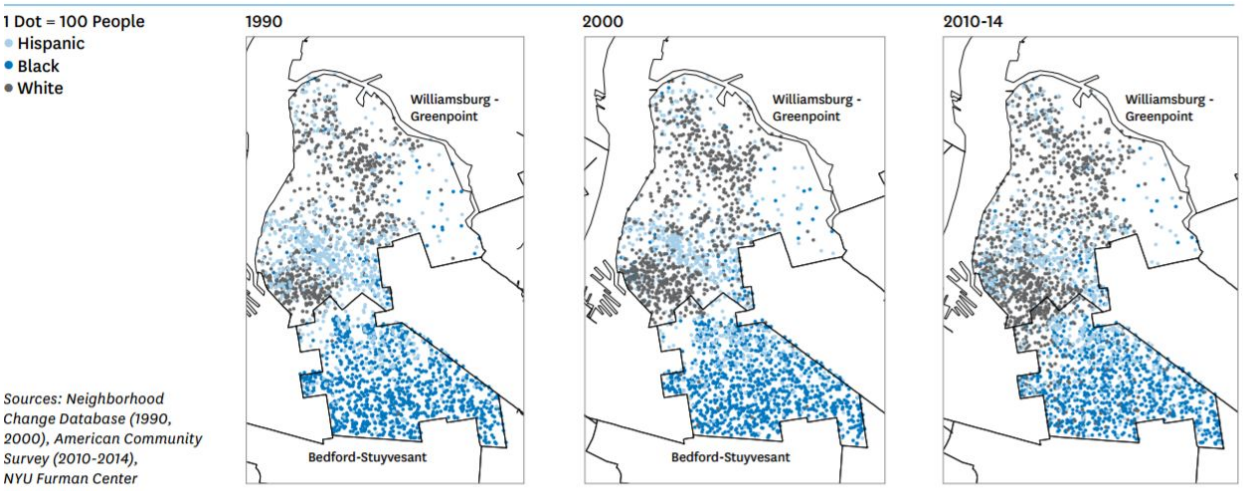
Sources: Neighborhood Change Database (1990, 2000), American Community Survey (2010-2014), NYU Furman Center

Average Rent
- Less than $750
- $751–$1,000
- $1,001–$1,250
- Greater than $1,250



Sources: Neighborhood Change Database (1990, 2000), American Community Survey (2010-2014), NYU Furman Center





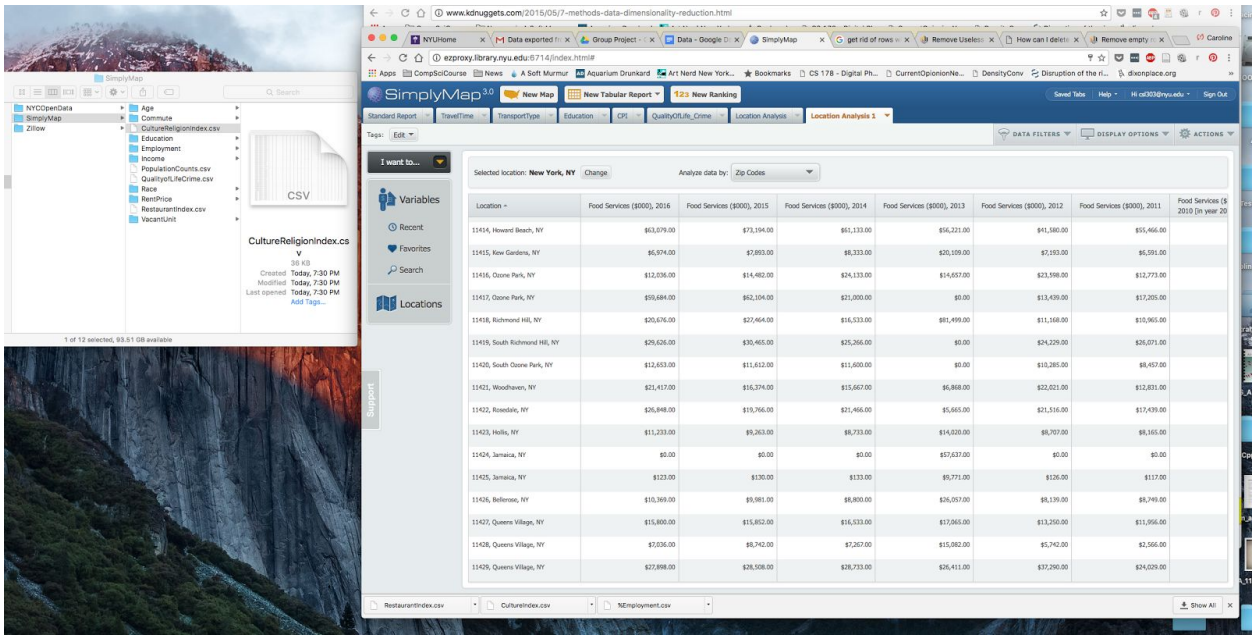
Data Understanding

Initially, four datasets were examined for the right attributes: NYC Open Data, Simply Map 3.0, Zillow, and MTA and NYC.gov data. Simply Map 3.0 and MTA/NYC.gov were chosen because both contained necessary information in a readable format. Simply Map 3.0 provided data collected annually from 2000 and 2008-2016, and compiled from various established sources: New York City government data, United States census information (for demographics information), crime, culture, and restaurant indices, etc. MTA/NYC.gov provided relevant information about the location and names of subway lines. At first glance, the data from 2000, 2008, and 2009 contained more missing values than 2010-2016 (more detail provided later).

Missing values were noted at particular zip codes as well. These zip codes are non-residential zip codes that exist for United States Postal Service (USPS) purposes only, therefore data was not collected from these areas.

Data Preparation

Attributes were downloaded in .csv format from Simply Map 3.0:



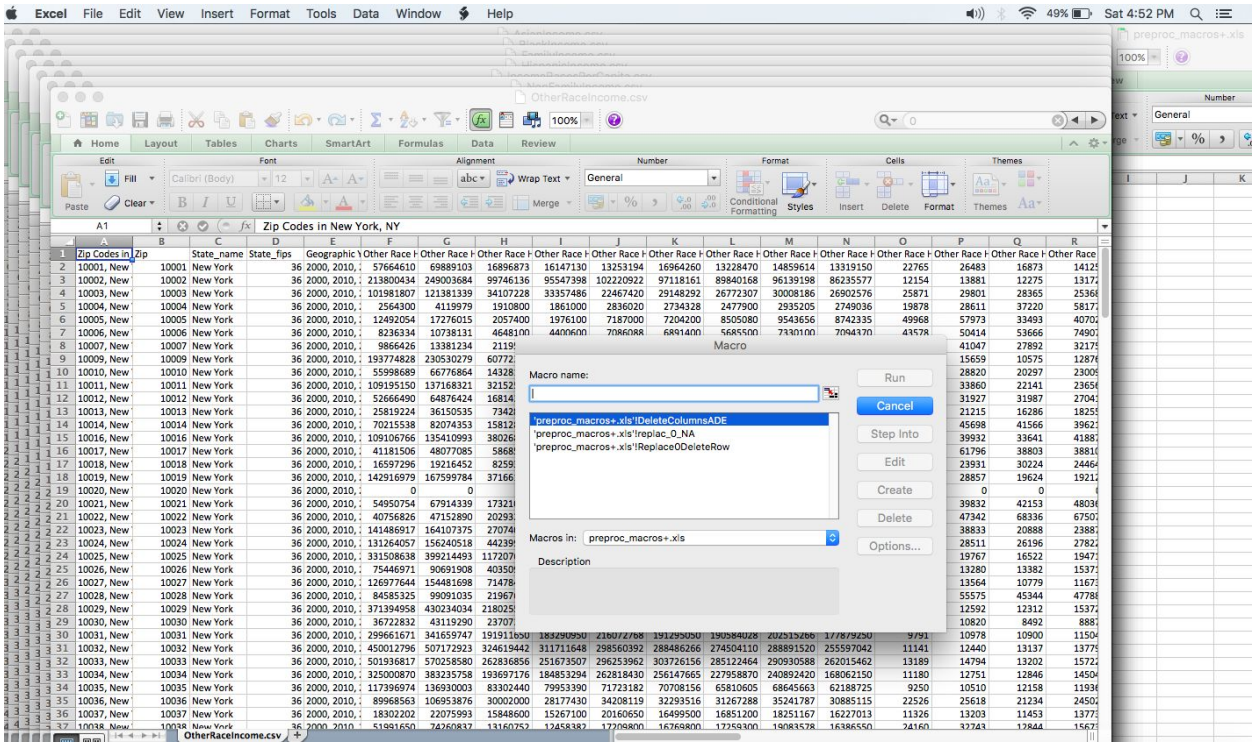
There were 140 attributes and 2120 rows, including Year and Zip Code.:

Year,Zip,TotalPopulationMedianAge,Population12to17Years,Population0to5Years,Population18to24Years,Population25to34Years,Population35to44Years,Population45to54Years,Population55to64Years,Population6to11Years,Population65to74Years,Population75to84Years,Population85YearsandOlder,EmploymentTravelTime3059Min,EmploymentTravelTime90Min,EmploymentTravelTime6089Min,EmploymentTravelTime1529Min,EmploymentTravelTimeLessthan15Min,EmploymentWorkatHome,EmploymentWalkedtoWork,EmploymentSubwayorElevatedtoWork,EmploymentCarTruckVantoWork,EmploymentFerrytoWork,EmploymentPublicTransportationtoWork,EmploymentRailroadtoWork,EmploymentTaxitoWork,EmploymentOtherTransportationtoWork,EmploymentBicycletoWork,EmploymentBusorTrolleyBustoWork,EducationCollegeMastersorDoctorateDegree,@#EducationAttainmentSomeCollege,@#EducationAttainmentBachelorsDegree,@#EducationEnrolledPublicSchool,EducationHighSchool_A,EducationAttainmentMastersDegree,EducationHighSchool,EducationAttainmentDoctorateDegree,@#EducationAttainmentAssociatesDegree,@#EducationEnrolledPrivateSchool,EmploymentCivilianTotalEmployed,EmploymentBlueColl

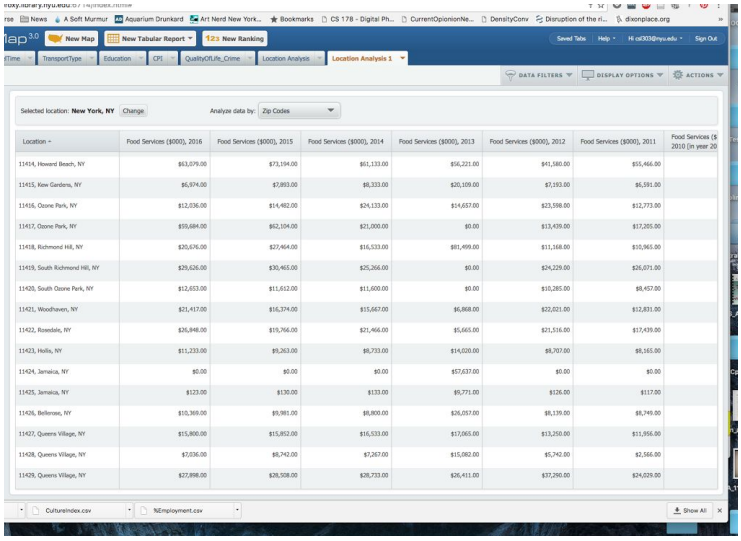
ar,EmploymentWhiteCollar,EmploymentWholesaleTrade,EmploymentArtsEntertainmentAccommodationandFoodServicesetc,EmploymentOtherServices,EmploymentFinanceandInsurance,EmploymentInformation,EmploymentManufacturing,EmploymentEducationalServices,EmploymentAgricultureForestryFishingandHunting,EmploymentConstruction,EmploymentProfessionalScientificandTechnicalServices,EmploymentRetailTrade,EmploymentTransportationandWarehousing,EmploymentRealEstateandRentalandLeasing,EmploymentManagementofCompaniesandEnterprises,EmploymentHealthCareandSocialAssistance,EmploymentArtsEntertainmentandRecreation,EmploymentAdministrativeandSupportandWasteMgt.Services,EmploymentUtilities,EmploymentPublicAdministration,EmploymentMiningQuarryingandOilandGasExtraction,AsianHouseholdIncomeMedian$,AsianHouseholdIncomeAverage$,AsianHouseholdIncomePerCapita$,AsianHouseholdIncomeTotal$,AsianHouseholdIncomeHighIncomeAverage$,BlackHouseholdIncomeMedian$,BlackHouseholdIncomeTotal$,BlackHouseholdIncomeAverage$,BlackHouseholdIncomeHighIncomeAverage$,BlackHouseholdIncomePerCapita$,FamilyInc.PerCapita$,FamilyIncomeTotal$,FamilyIncomeAverage$,FamilyIncomeHighIncomeAverage$,FamilyIncomeMedian$,HispanicHouseholdIncomeTotal$,HispanicHouseholdIncomeAverage$,HispanicHouseholdIncomeMedian$,HispanicHouseholdIncomePerCapita$,HispanicHouseholdIncomeHighIncomeAverage$,WhiteHouseholdIncomePerCapita$,OtherRaceHouseholdIncomePerCapita$,NonFamilyInc.Median$,NonFamilyInc.Average$,NonFamilyInc.PerCapita$,NonFamilyInc.HighIncomeAverage$,NonFamilyInc.Total$,OtherRaceHouseholdIncomeMedian$,OtherRaceHouseholdIncomeTotal$,OtherRaceHouseholdIncomeAverage$,OtherRaceHouseholdIncomeHighIncomeAverage$,WhiteHouseholdIncomeTotal$,EducationIndex,CultureIndex,ReligionIndex,@#PopulationPop,FamilyPopulation,NonFamilyPopulation,PopulationDensitypersq.mile,AggravatedAssaultIndex,EASIQualityofLifeIndex,MotorVehicleTheftIndex,ForcibleRapeIndex,MurderIndex,LarcenyIndex,ForcibleRobberyIndex,BurglaryIndex,EASITotalCrimeIndex,AsianPopulation,AsianHouseholds,BlackPopulation,BlackHouseholds,HispanicPopulation,HispanicHouseholds,OtherRaceHouseholds,OtherRacePopulation,WhiteHouseholds,WhitePopulation,WhiteNonHispanicHouseholds,WhiteNonHispanicPopulation,HouseholdswRent$1000$1249,HouseholdswRent$1250$1499,HouseholdswRent$1500$1999,HouseholdswRent$2000,HouseholdswRent$250$499,HouseholdswRent$500$749,HouseholdswRent$750$999,HouseholdswRentlessthan$250,HousingMedianRent$,VacantUnits,VacantUnitsForRent,VacantUnitsForSale,VacantUnitsSeasonal,VacantUnitsOther,@#Entrances,@#Lines,@#Stations

Attributes highlighted in green are from the Metropolitan Transportation Authority (MTA) while the rest were downloaded from Simply Map 3.0.

Initial preprocessing was done using Microsoft Excel. Macros were formulated to eliminate columns that held redundant or unnecessary information, such as state ID:

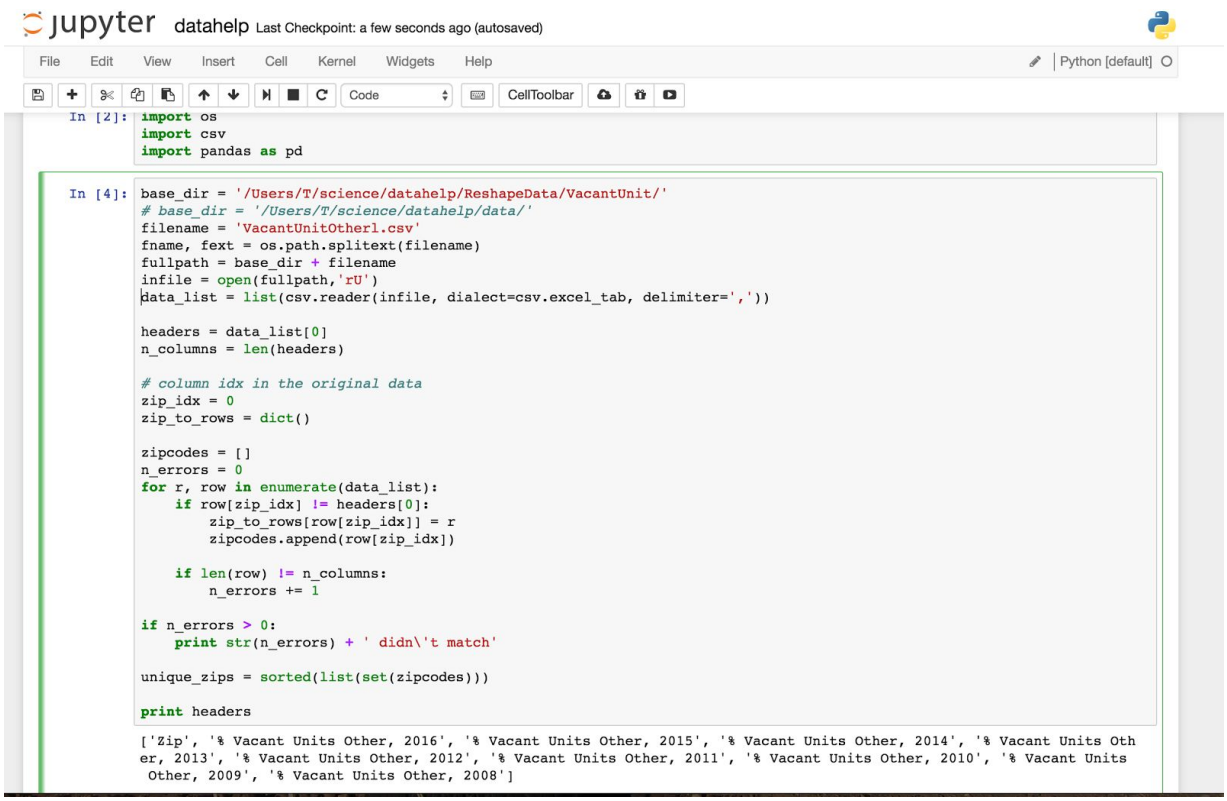


Afterward, each attribute from Simply Map 3.0 needed to be transformed from its current format to a working version (see figures below).



	Year *integer* *id*	Zip *integer*	TotalPopul... *real*	Population... *real*	Population... *real*	Population... *real*	Populati *real*
1	2010	10001	38.300	2.277	3.441	14.421	20.747
2	2010	10002	39.800	5.732	4.923	7.317	18.632
3	2010	10003	34.100	1.669	2.909	18.445	27.626
4	2010	10004	32.200	1.225	4.096	10.008	44.552
5	2010	10005	31.700	0.578	4.113	14.446	43.882
6	2010	10006	31.900	1.226	4.847	11.387	43.562
7	2010	10007	33.900	3.673	10.415	7.528	27.535
8	2010	10009	37.900	4.295	4.265	9.470	23.699
9	2010	10010	35.300	1.681	4.012	10.492	31.320
10	2010	10011	40.200	2.573	3.356	5.565	24.267
11	2010	10012	35.700	2.680	3.179	7.818	32.318
12	2010	10013	39.600	3.680	6.760	9.483	17.216

This transformation was completed using several different methods. The quickest most visually informative process was through Python and the Jupyter Notebook:



Several other methods of data preparation were experimented with and reported in our final presentation, but Python and Jupyter was the best method overall.

MTA attributes such as number of lines and stations per zip code had to be calculated from converted from latitude/longitude pairs (reverse geocoding). We tried several methods using the Google API, Texas A&M University's Geoservices, and Doogal. All methods are in more detail within the presentation.

Data Cleaning

Since some records have too many missing values, we decided to drop rows that have missing values more than 20% of number of columns. In order to do that, we used the function dropna from dataframe in Python.

```
df = df.dropna(thresh = 0.80 * len(df.columns))
# thresh means it requires that many non-NA values in that row/column
```

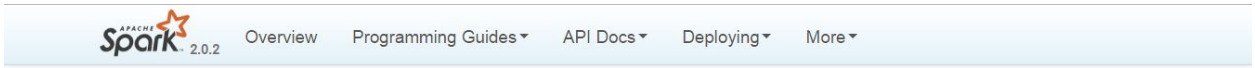
Getting Zip Code by Google API

Since we can only get longitude and latitude of subway entrances from NYC government. We need to get the zip code by using these data. To do that, we used Google API.

Some of the code for getting zipcode:

```
googlemap =
https://maps.googleapis.com/maps/api/geocode/json?key=AIzaSyBRJoEcdgq0V_gRkJiWpVSpJUD4JodOc-Q&latlng=

for address in addresslist:
  mapurl = ''
  mapresult = []
  getzip = False
  mapurl = googlemap + address[1] + ',' + address[0]
  maplink = urllib.urlopen(mapurl)
  mapdata = json.loads(maplink.read())

mapresult = mapdata['results'][0]['address_components']
for x in mapresult:
  if 'postal_code' in x['types']:
    az = address,'\t',x['long_name']
    zipcodelist.append(az)
    getzip = True
    break
```

During the data preparation phase, we are going to use StringIndexer, OneHotEncoder, VectorAssembler and ChiSqSelector from the extracting, transforming and selecting features in Spark



Extracting, transforming and selecting features

This section covers algorithms for working with features, roughly divided into these groups:

- Extraction: Extracting features from "raw" data
- Transformation: Scaling, converting, or modifying features
- Selection: Selecting a subset from a larger set of features

Table of Contents

One Hot Encoding

What is One Hot Encoding?
One Hot means there's only 1 "hot" or "on" value in this list, while the rest are "cold" or "off".
It can transform categorical features to a format that works better with classification and regression algorithms.
A common application of One Hot Encoding is in NLP, where words don't exactly turn into vectors so easily.
For example:

foo	color
1	green
2	red
3	blue

After transform, our data will be like:

foo	color
1	[1,0,0]
2	[0,1,0]
3	[0,0,1]

(vector)
or

foo	color=green	color=red	color=blue
1	1	0	0
2	0	1	0
3	0	0	1

(dummy)

Why do we need it?
Any learner based on standard distance metrics (such as k-nearest neighbors) between samples will get confused without one-hot encoding. With the naive encoding and Euclidean distance, the distance between Green and Red is 1. The distance between Green and Blue is 2. But with the one-hot encoding, the pairwise distances between [1, 0, 0], [0, 1, 0] and [0, 0, 1] are all equal to $\sqrt{2}$.
However, this is not true for all models. Decision trees and derived models such as random forests, if deep enough, can handle categorical variables without one-hot encoding.
In NLP, one hot encoding is very common, transform word to vector.
Reference: https://achyutjoshi.github.io/datascience/one-hot-encoding

How to do it?
For Python, we can use:
Pandas get_dummies
http://pandas.pydata.org/pandas-docs/stable/generated/pandas.get_dummies.html
Sklearn feature_extraction DictVectorizer
http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html
Spark MLlib OneHotEncoder
http://spark.apache.org/docs/latest/ml-features.html#onehotencoder

Is Zip code a quantitative or categorical variable?
Although zip codes are written in numbers, the numbers are simply convenient labels and don't have numeric meaning (for example, we wouldn't add together two zip codes or calculate average of two zip codes)
http://www.dummies.com/education/math/statistics/how-to-distinguish-quantitative-and-categorical-variables/

How about year?
Same as zip code, it's categorical variable. Since we are not going to calculate anything by year, and in this project, year means interval of time.
https://www.quora.com/Is-year-a-quantitative-or-categorical-variable

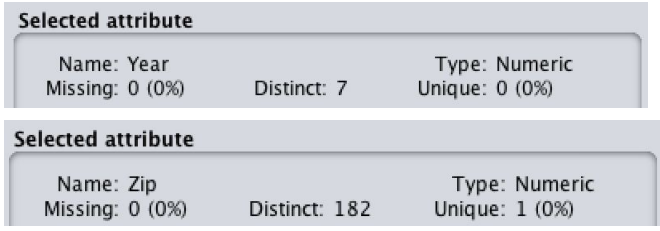
Implementation in our project:
In our project, we decided to use Spark and since we have both Year and Zip code, we need to do encoding for twice.
First, we need to use StringIndexer to index both Year and Zip code attributes.
Second, we use OneHotEncoder to transform attributes to vector, since in MLlib, most of the models requires one vector as features.

```
import pandas as pd
from pyspark import SparkContext
from pyspark.sql import SQLContext
from pyspark.ml.feature import OneHotEncoder, StringIndexer

# Creates Spark Context
sc = SparkContext()
sql_sc = SQLContext(sc)
pandas_df = pd.read_csv('data.csv', na_values=[' '])
spark_df = sql_sc.createDataFrame(pandas_df)

# Step 1 - StringIndexer
firstIndexer = StringIndexer(inputCol="Year", outputCol="YearIndex")
firstModel = firstIndexer.fit(spark_df)
firstIndexed = firstModel.transform(spark_df)
secondIndexer = StringIndexer(inputCol="Zip", outputCol="ZipIndex")
secondModel = secondIndexer.fit(firstIndexed)
secondIndexed = secondModel.transform(firstIndexed)

# Step 2 - OneHotEncoder
yearEncoder = OneHotEncoder(dropLast=False, inputCol="YearIndex", outputCol="YearVec")
zipEncoder = OneHotEncoder(dropLast=False, inputCol="ZipIndex", outputCol="ZipVec")
yearEncoded = yearEncoder.transform(secondIndexed)
zipEncoded = zipEncoder.transform(yearEncoded)
final_df = zipEncoded
final_df.select("YearIndex","YearVec","ZipIndex","ZipVec").show()
```

Year and zip attributes after encoding:

```
+---------+-------------+--------+-----------------+
|YearIndex|      YearVec|ZipIndex|           ZipVec|
+---------+-------------+--------+-----------------+
|      4.0|(7,[4],[1.0])|    47.0| (182,[47],[1.0])|
|      4.0|(7,[4],[1.0])|    20.0| (182,[20],[1.0])|
|      4.0|(7,[4],[1.0])|   153.0|(182,[153],[1.0])|
|      4.0|(7,[4],[1.0])|   115.0|(182,[115],[1.0])|
|      4.0|(7,[4],[1.0])|    87.0| (182,[87],[1.0])|
|      4.0|(7,[4],[1.0])|    30.0| (182,[30],[1.0])|
|      4.0|(7,[4],[1.0])|   163.0|(182,[163],[1.0])|
|      4.0|(7,[4],[1.0])|    21.0| (182,[21],[1.0])|
|      4.0|(7,[4],[1.0])|    25.0| (182,[25],[1.0])|
|      4.0|(7,[4],[1.0])|   102.0|(182,[102],[1.0])|
|      4.0|(7,[4],[1.0])|    53.0| (182,[53],[1.0])|
|      4.0|(7,[4],[1.0])|     8.0|  (182,[8],[1.0])|
|      4.0|(7,[4],[1.0])|   146.0|(182,[146],[1.0])|
|      4.0|(7,[4],[1.0])|    66.0| (182,[66],[1.0])|
|      4.0|(7,[4],[1.0])|    44.0| (182,[44],[1.0])|
|      4.0|(7,[4],[1.0])|   134.0|(182,[134],[1.0])|
|      4.0|(7,[4],[1.0])|   170.0|(182,[170],[1.0])|
|      4.0|(7,[4],[1.0])|    26.0| (182,[26],[1.0])|
|      4.0|(7,[4],[1.0])|   107.0|(182,[107],[1.0])|
|      4.0|(7,[4],[1.0])|    70.0| (182,[70],[1.0])|
+---------+-------------+--------+-----------------+
only showing top 20 rows
```

As we can see, the first number of YearVec and ZipVec means the distinct values in this attribute. We can compare with the result from Weka.



The One Hot Encoding result from Spark is correct.

Feature Selection

The feature selection algorithms in Spark MLlib includes VectorSlicer, RFormula and ChiSqSelector. In our project, we are going to use ChiSqSelector.

What is ChiSqSelector?

ChiSqSelector implements Chi-Squared feature selection. It operates on labeled data with categorical features. ChiSqSelector orders features based on a Chi-Squared test of independence from the class, and then filters (selects) the top features which the class label depends on the most. This is akin to yielding the features with the most predictive power.

How to use it?

First, we need to assemble the features into one vector.

```
assembler = VectorAssembler(inputCols = df_columns, outputCol = "features")
output = assembler.transform(spark_df)
regModelData = output.select("features", "HousingMedianRent$")
```

Second, we apply the data into ChiSqSelector, in this case, we keep 50 features.

```
selector = ChiSqSelector(numTopFeatures = 50, featuresCol = "features", outputCol = "selectedFeatures", labelCol = "HousingMedianRent$")
result = selector.fit(output).transform(output)
model = selector.fit(output)
```

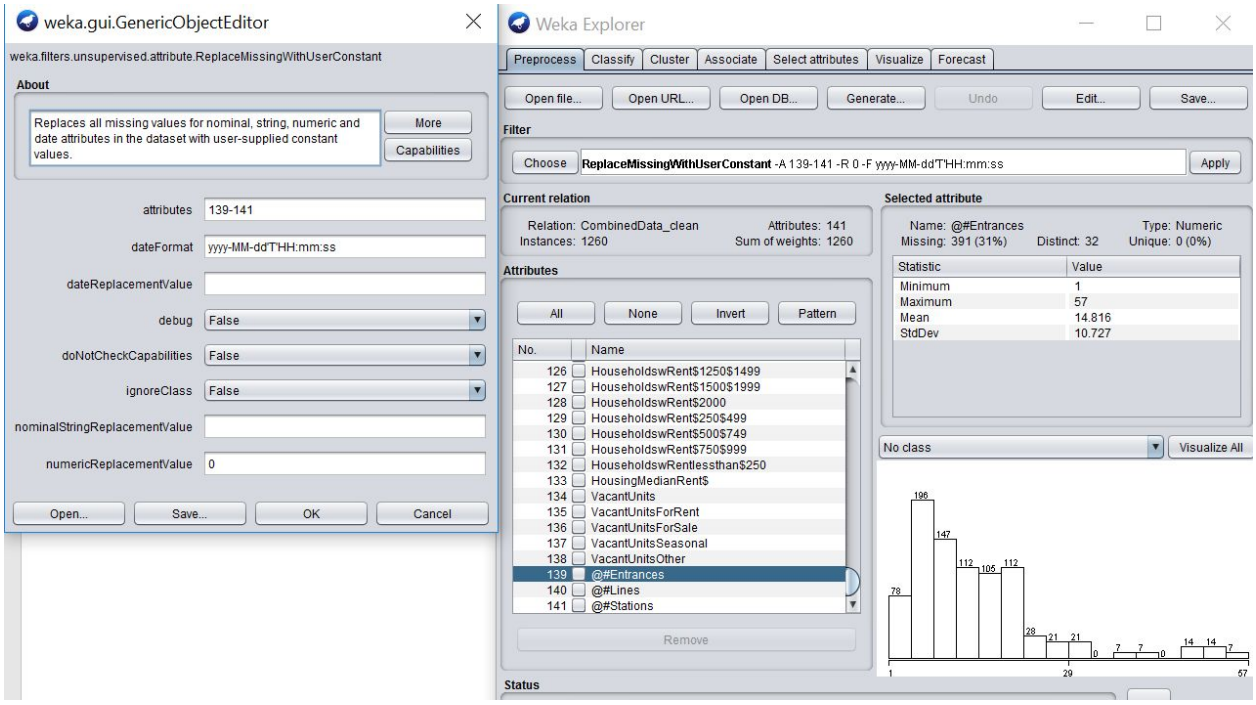
The result from ChiSqSelector

ChiSqSelector with top 50 features selected:

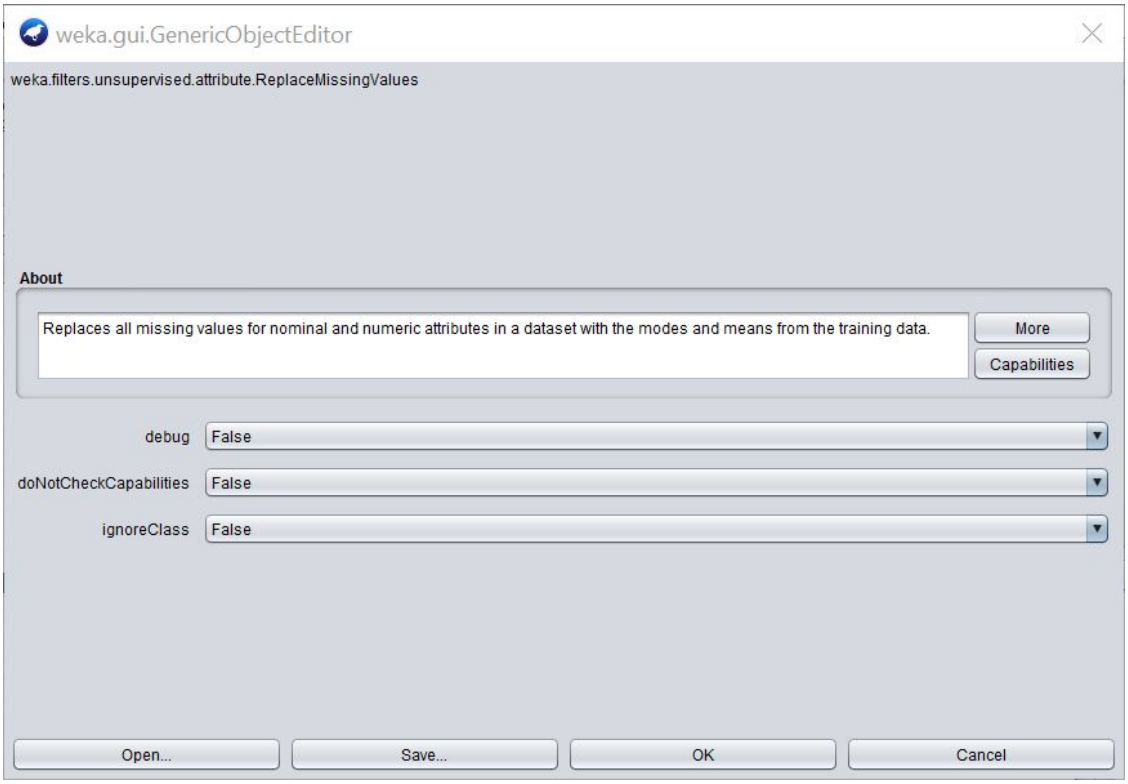
Population12to17Years
Population0to5Years
Population18to24Years
Population6to11Years
Population65to74Years
Population75to84Years
EmploymentTravelTime3059Min
EmploymentTravelTime90Min
EmploymentTravelTime6089Min
EmploymentTravelTime1529Min
EmploymentTravelTimeLessthan15Min
EmploymentWalkedtoWork
EmploymentSubwayorElevatedtoWork
EmploymentCarTruckVantoWork
EducationCollegeMastersorDoctorateDegree
EducationHighSchool_A
EducationAttainmentMastersDegree
EducationHighSchool
EmploymentWhiteCollar
EmploymentArtsEntertainmentAccommodationandFoodServices etc
EmploymentFinanceandInsurance
EmploymentEducationalServices
EmploymentProfessionalScientificandTechnicalServices
EmploymentHealthCareandSocialAssistance
EmploymentPublicAdministration
AsianHouseholdIncomeAverage$
AsianHouseholdIncomeTotal$
FamilyIncomeTotal$
FamilyIncomeAverage$
FamilyIncomeMedian$
HispanicHouseholdIncomeTotal$
HispanicHouseholdIncomeAverage$
NonFamilyIncAverage$
NonFamilyIncTotal$
WhiteHouseholdIncomeTotal$
FamilyPopulation
PopulationDensitypersqmile
AsianHouseholds
BlackPopulation
BlackHouseholds
HispanicPopulation
HispanicHouseholds
OtherRaceHouseholds
OtherRacePopulation
WhiteHouseholds
WhitePopulation
WhiteNonHispanicHouseholds
WhiteNonHispanicPopulation
HouseholdswRent$1500$1999
VacantUnits

Handling missing values with *Weka*

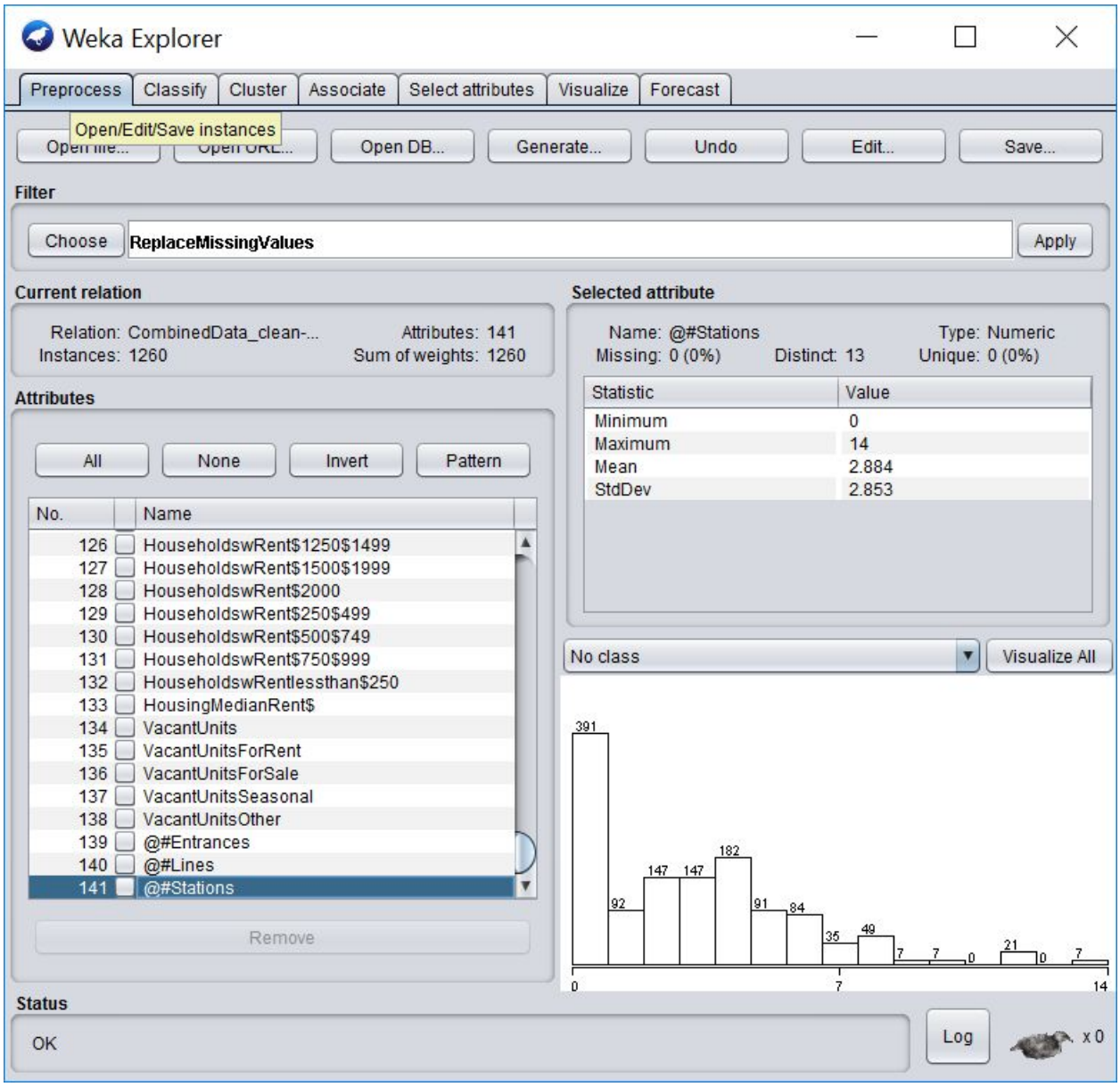
For attributes #Entrance, #Lines and #Stations we apply “ReplaceMissingWithUserConstant” filter and replace all the missing values with zero. Therefore, these three attributes have no longer missing values.



Then we apply “ReplaceMissingValues” to all attributes:

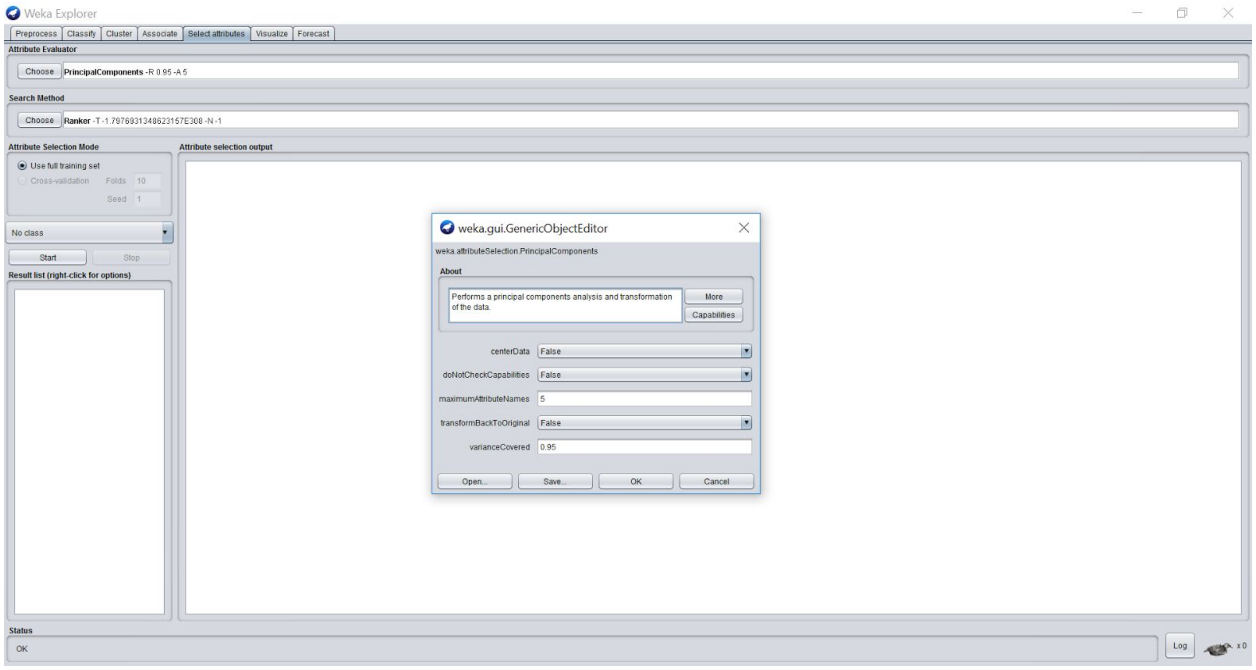


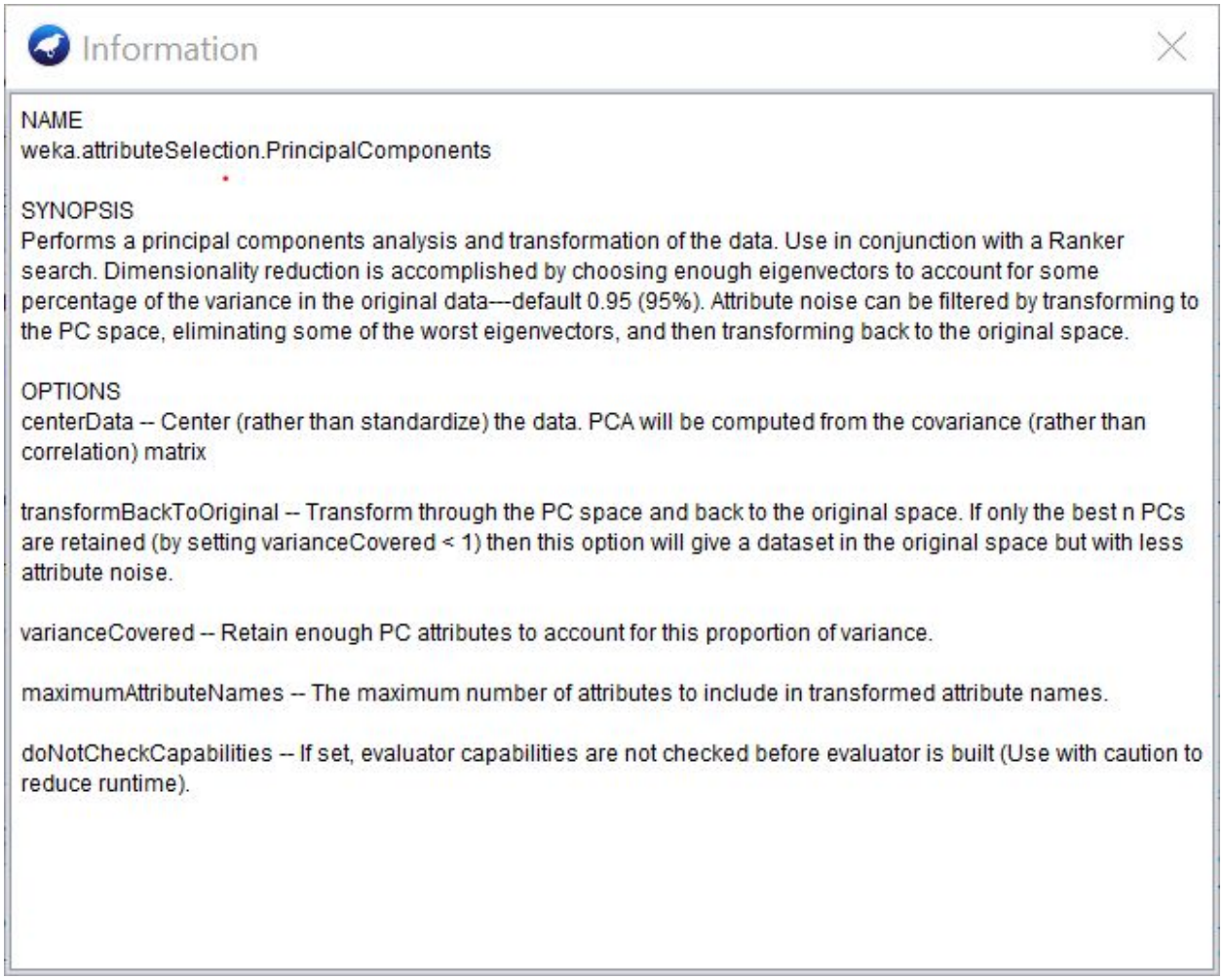
After replacing missing values:



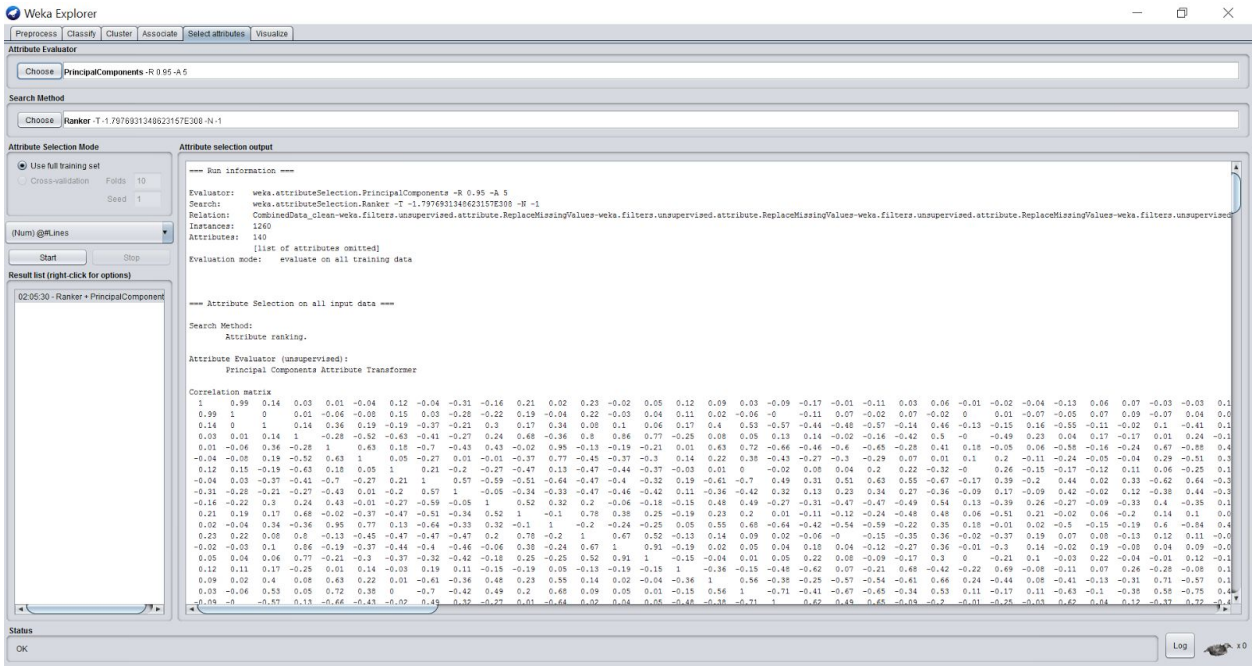
Applying PCA on the data

Apply PCA by setting “VarianceCovered” to 0.95 (default):

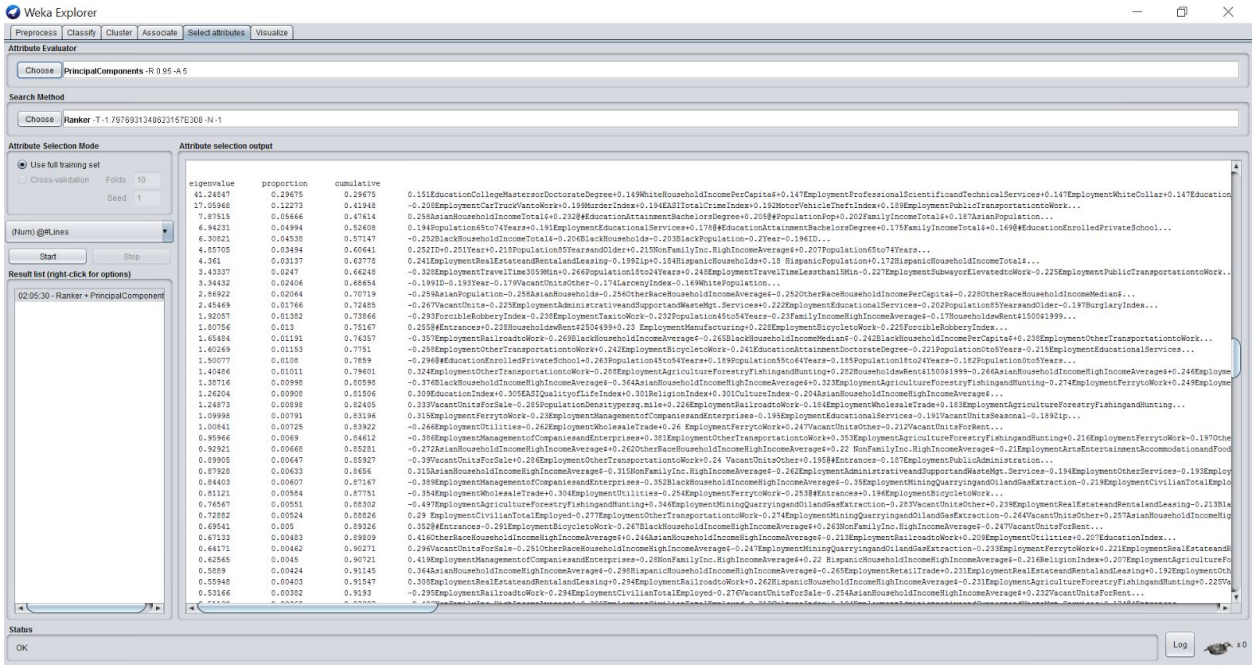

Weka Explorer
Preprocess
Classify
Cluster
Associate
Select attributes
Visualize
Forecast
Attribute Evaluator
Choose
PrincipalComponents -R 0.95 -A 5
Search Method
Choose
Ranker -T -1.7976931348623157E308 -N -1
Attribute Selection Mode
Use full training set
Cross-validation
Folds
10
Seed
1
No class
Start
Stop
Result list (right-click for options)
Attribute selection output
weka.gui.GenericObjectEditor
weka.attributeSelection.PrincipalComponents
About
Performs a principal components analysis and transformation of the data.
More
Capabilities
centerData
False
doNotCheckCapabilities
False
maximumAttributeNames
5
transformBackToOriginal
False
varianceCovered
0.95
Open...
Save...
OK
Cancel
Status
OK
Log
x 0


Information
NAME
weka.attributeSelection.PrincipalComponents
SYNOPSIS
Performs a principal components analysis and transformation of the data. Use in conjunction with a Ranker search. Dimensionality reduction is accomplished by choosing enough eigenvectors to account for some percentage of the variance in the original data---default 0.95 (95%). Attribute noise can be filtered by transforming to the PC space, eliminating some of the worst eigenvectors, and then transforming back to the original space.
OPTIONS
centerData -- Center (rather than standardize) the data. PCA will be computed from the covariance (rather than correlation) matrix
transformBackToOriginal -- Transform through the PC space and back to the original space. If only the best n PCs are retained (by setting varianceCovered < 1) then this option will give a dataset in the original space but with less attribute noise.
varianceCovered -- Retain enough PC attributes to account for this proportion of variance.
maximumAttributeNames -- The maximum number of attributes to include in transformed attribute names.
doNotCheckCapabilities -- If set, evaluator capabilities are not checked before evaluator is built (Use with caution to reduce runtime).

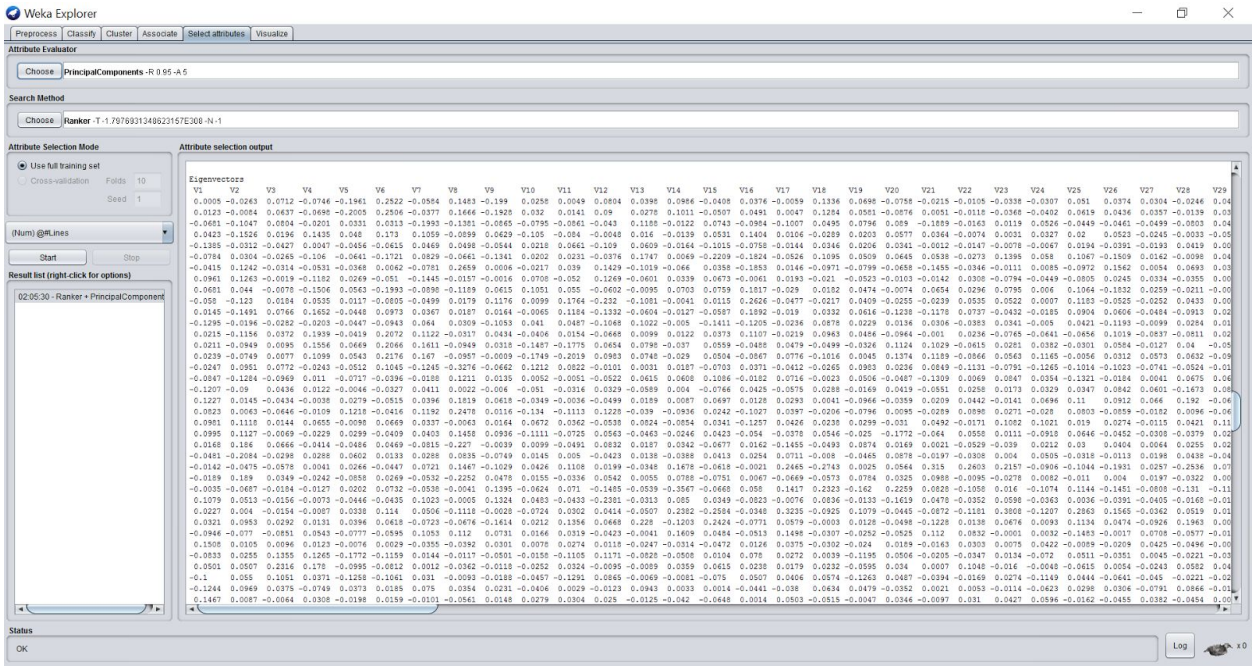
Correlation Matrix:



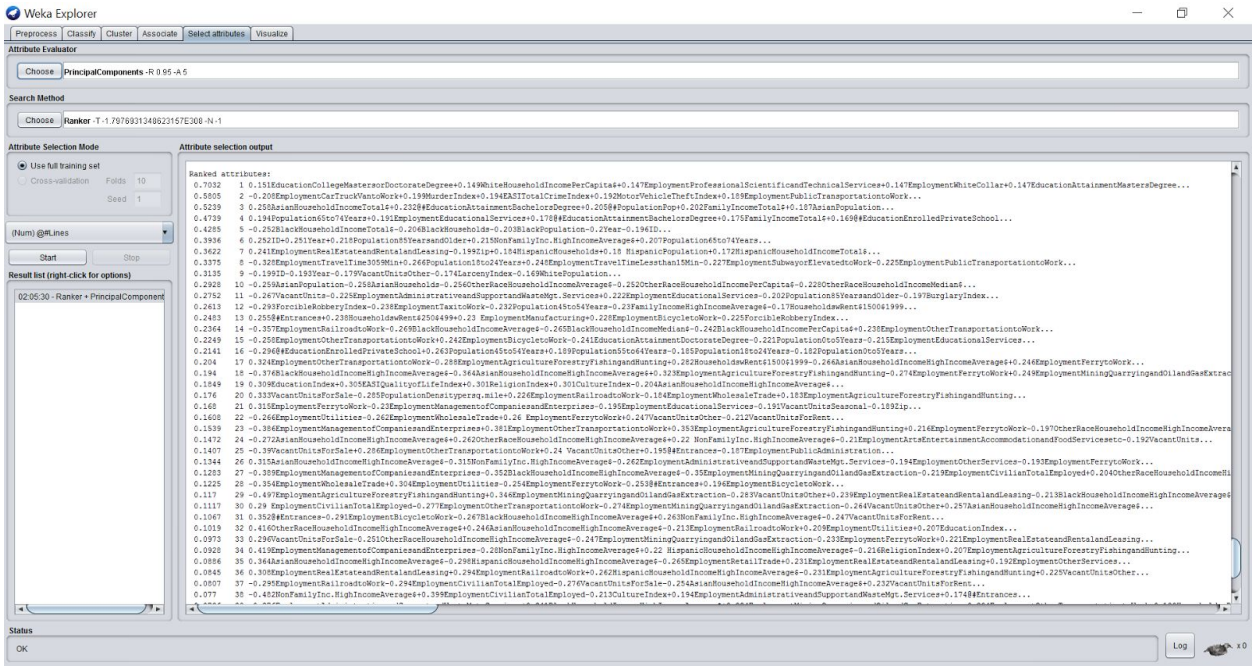
Eigen Values:



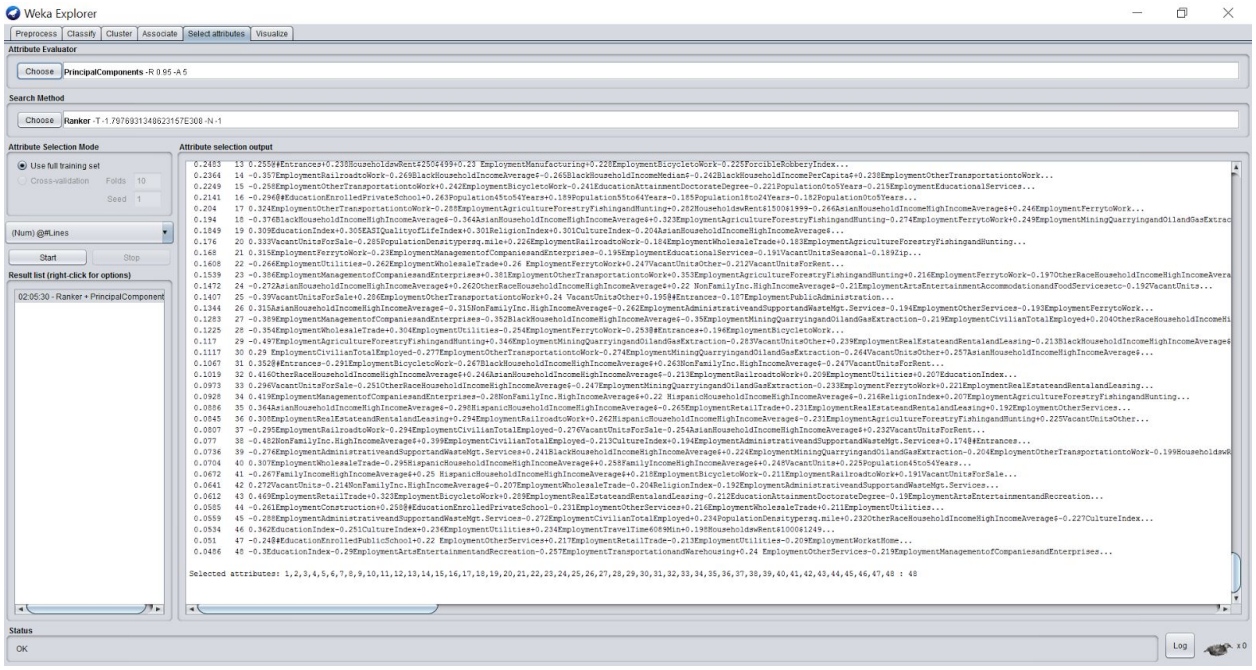
Eigen Vectors:



Ranked attributes:



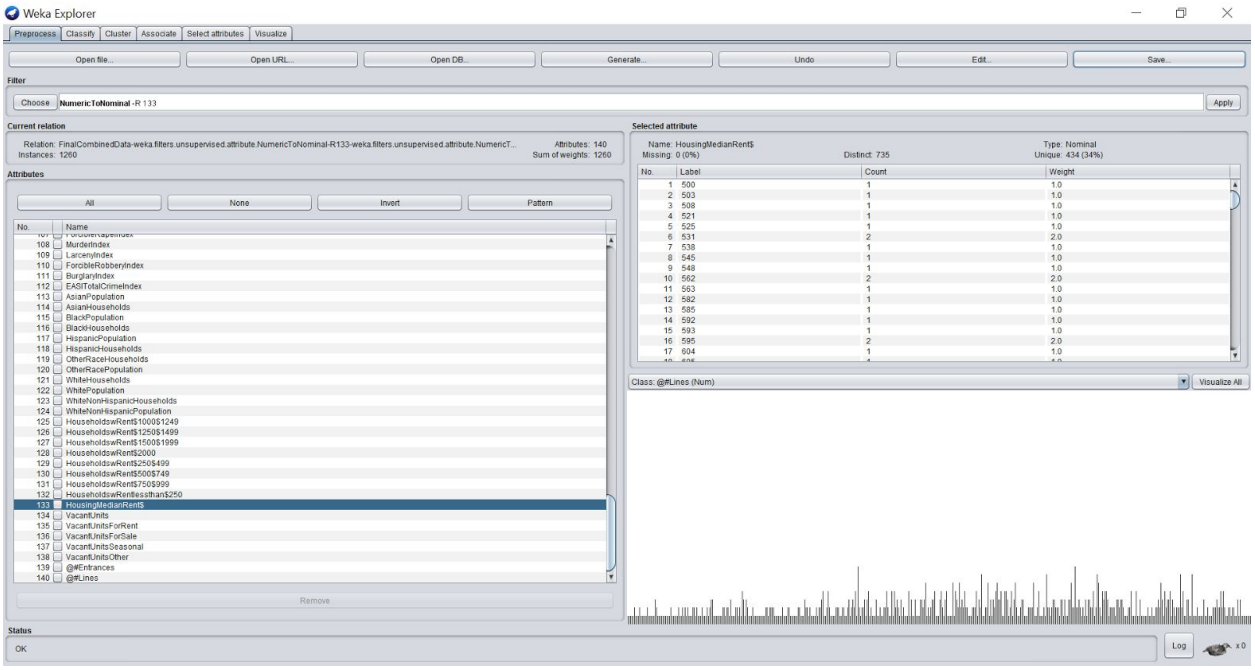
We can see only 48 attributes out of 140 attributes are selected:



After applying PCA, we lost interpretability of our data but since PCA returned 48 attributes, we decided to select “50” most important attributes in feature selection part.

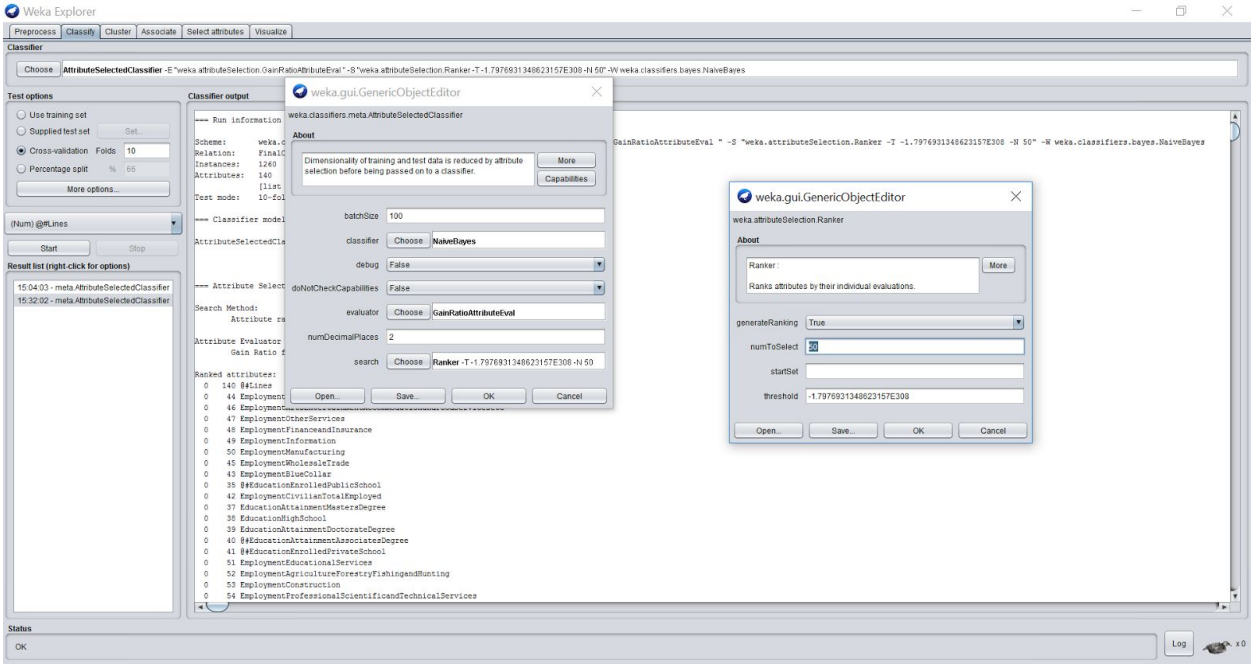
Attribute Selection with *Weka*

Attribute subset selection involves subset evaluation measure, and search method. But we have 140 attributes so searching is slow! An alternative could be using a single-attribute evaluator with ranking which evaluates each attribute individually instead of evaluating subsets of attributes. An advantage of this approach is that it can eliminate irrelevant attributes. Although, it can’t eliminate redundant attributes. We choose “HousingMedianRent$” attribute as the class. Since the data for this attribute is “Numeric”, we need to convert it to “Nominal”. We apply this transformation with “NumericToNominal” filter in Weka:

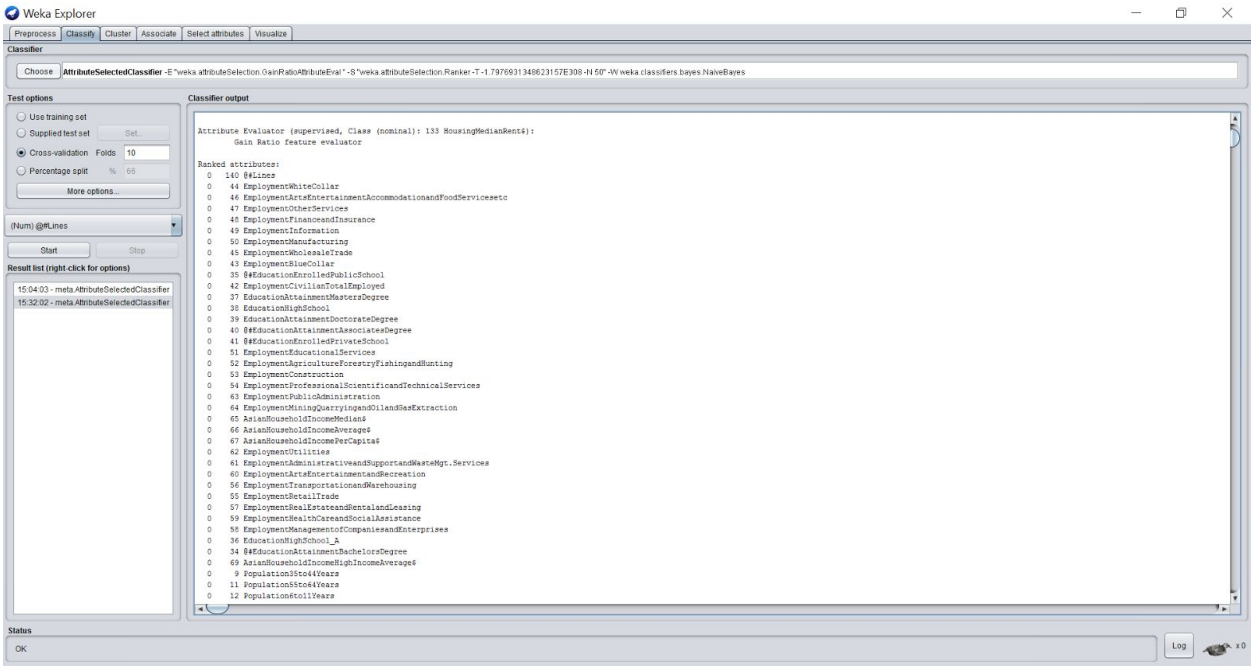


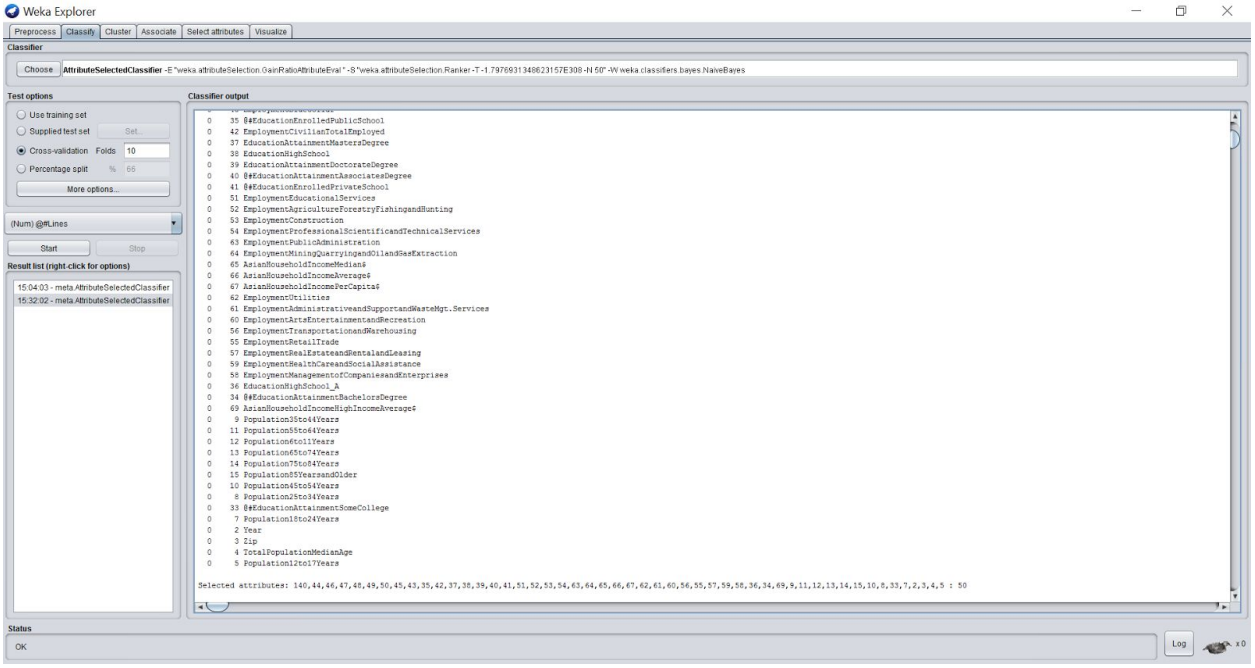
Gain Ratio Attribute Evaluator:

We use “AttributeSelectedClassifier” to get an evaluation of the attributes. We choose NaiveBayes as the classifier, Gain Ratio Attribute Evaluator for the evaluator with “Ranker” search method. For “Ranker”, we set the parameter “numToSelect” to 50 which outputs 50 best ranked attributes. We choose 50 since PCA returned back 50 attributes after applying data reduction. The default threshold parameter for “Ranker” is minus infinity (which is equal to the shown parameter in Java). We use “Cross-Validation” with 10 “Folds” for the evaluation.

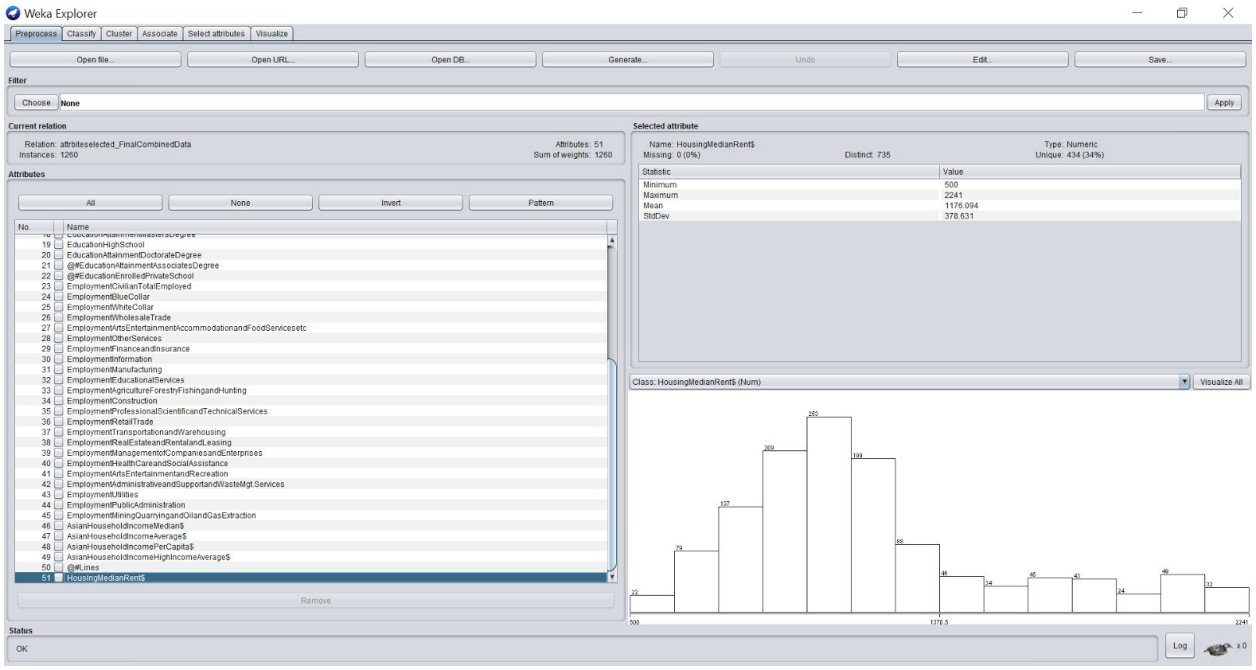


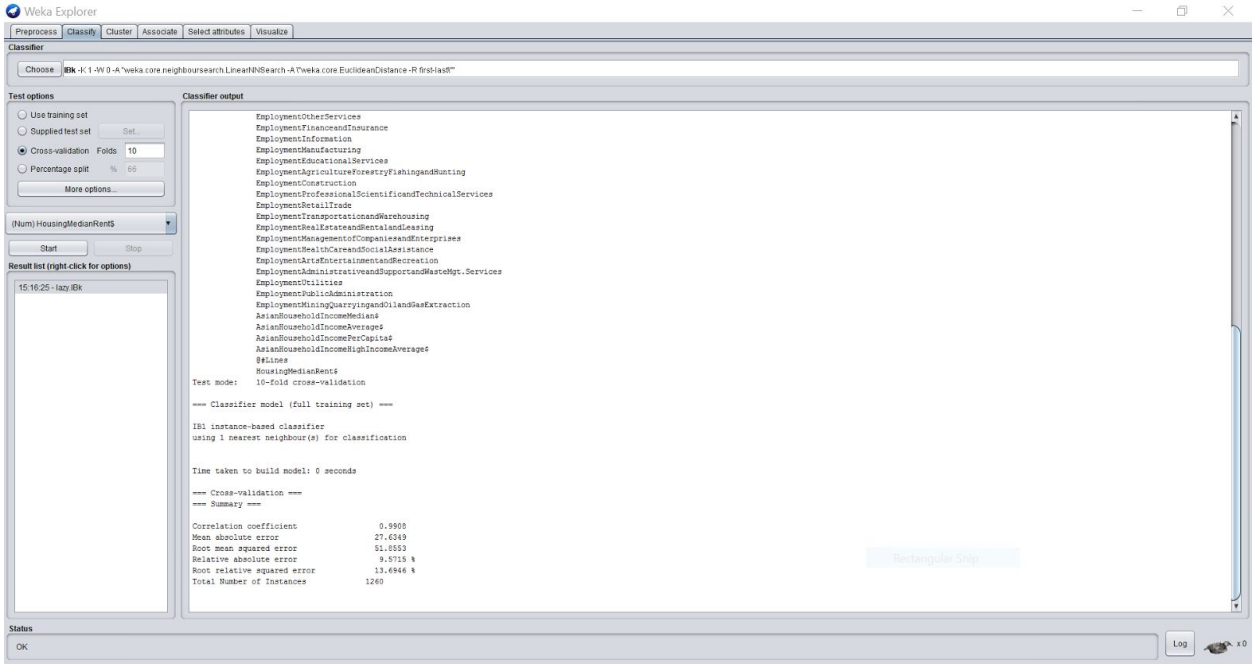
50 selected attributes are as following. As we see the most important selected attribute is “number of stations” which confirms our assumption that rental prices are directly dependent on access to the subway.





After selecting the 50 most important attributes, we create a data set with these attributes and convert the data of attribute “HousingMedianRent” to original (to avoid losing information).
We choose “HousingMedianRent “as the class and then we apply IBK-lazy classifier, and as we see:
Relative absolute error: 9.5715 %
Root relative squared error: 13.6946 %





Modeling

Modeling with Spark

In this phase, we are going to use models from MLlib, there are Linear Regression, Decision Tree Regression and Random Forest Regression from its Classification and Regression.

Classification and regression

This page covers algorithms for Classification and Regression. It also includes sections discussing specific classes of algorithms, such as linear methods, trees, and ensembles.

Table of Contents

Linear Regression in MLlib

What is Linear Regression?
Linear regression attempts to model the relationship between two variables by fitting a linear equation to observed data. One variable is considered to be an explanatory variable, and the other is considered to be a dependent variable.

Some code for Linear Regression

```
sc = SparkContext()
sql_sc = SQLContext(sc)
pandas_df = pd.read_csv('data.csv', na_values=[' '])
spark_df = sql_sc.createDataFrame(pandas_df)

df_columns = spark_df.columns
df_columns = df_columns[3:-1]
assembler = VectorAssembler(inputCols = df_columns, outputCol = "features")
vector_df = assembler.transform(spark_df)
data = vector_df.select("features", "HousingMedianRent$")
(trainingData, testData) = data.randomSplit([0.7, 0.3])

lr = LinearRegression(labelCol = "HousingMedianRent$")
model = lr.fit(trainingData, {lr.regParam:0.3})
predictions = model.transform(testData)
```

Result

```
+--------------------+------------------+-----------------+
|            features|HousingMedianRent$|       prediction|
+--------------------+------------------+-----------------+
|[29.6,6.82562,10....|               800|834.1419710054527|
|[29.8,9.00797,9.8...|               639|621.5723793103944|
|[30.0,9.33527,9.4...|               811|794.5769589562694|
|[30.0,10.4396,9.9...|               637|737.6132118800572|
|[30.2,9.37296,10....|               781|758.7192914819996|
+--------------------+------------------+-----------------+
only showing top 5 rows

Linear Regression result:
Root Mean Squared Error (RMSE): 50.3514
Mean Squared Error (MSE): 2535.26
Mean Absolute Error (MAE): 36.049
R-squared: 0.981682
```

Linear Regression result:
Root Mean Squared Error (RMSE): 50.3514
Mean Squared Error (MSE): 2535.26
Mean Absolute Error (MAE): 36.049
R-squared: 0.981682

The result is very good. RMSE is not high and R- squared is close to 1.

What is RMSE?
RMSE measures how much error there is between two datasets. RMSE usually compares a predicted value and an observed value.

What is R squared?
R-squared is a statistical measure of how close the data are to the fitted regression line. It is also known as the coefficient of determination, or the coefficient of multiple determination for multiple regression. The definition of R-squared is the percentage of the response variable variation that is explained by a linear model, therefore R-squared is always between 0 and 100%.
R-squared = Explained variation/Total variation
0% indicates that the model explains none of the variability of the response data around its mean.
100% indicates that the model explains all the variability of the response data around its mean.

Decision Tree in MLlib

What is Decision Tree?
Decision trees and their ensembles are popular methods for the machine learning tasks of classification and regression. Decision trees are widely used since they are easy to interpret, handle categorical features, extend to the multiclass classification setting, do not require feature scaling, and can capture non-linearity and feature interactions. Tree ensemble algorithms such as random forests and boosting are among the top performers for classification and regression tasks.

The spark.ml implementation supports decision trees for binary and multiclass classification and for regression, using both continuous and categorical features.

Regression vs Classification
Regression: the output variable takes continuous or numeric values.
Classification: the output variable takes class labels.
In our case, we are going to use regression since we want to predict the rent (numeric) in the future.

Some code for Linear Regression

```
sc = SparkContext()
sql_sc = SQLContext(sc)
pandas_df = pd.read_csv('data.csv', na_values=[' '])
spark_df = sql_sc.createDataFrame(pandas_df)

df_columns = spark_df.columns
df_columns = df_columns[3:-1]
assembler = VectorAssembler(inputCols = df_columns, outputCol = "features")
vector_df = assembler.transform(spark_df)
data = vector_df.select("features", "HousingMedianRent$")
featureIndexer = VectorIndexer(inputCol="features", outputCol="indexedFeatures",
maxCategories=4).fit(data)
```

```
(trainingData, testData) = data.randomSplit([0.7, 0.3])

dt = DecisionTreeRegressor(featuresCol="indexedFeatures",
labelCol="HousingMedianRent$")
pipeline = Pipeline(stages=[featureIndexer, dt])
model = pipeline.fit(trainingData)
predictions = model.transform(testData)
```

Result

```
+--------------------+------------------+-----------------+
|            features|HousingMedianRent$|       prediction|
+--------------------+------------------+-----------------+
|[29.4,9.95204,14....|               891|879.2083333333334|
|[29.9,8.8449,9.77...|               644|783.9322033898305|
|[29.9,9.36187,9.5...|               804|879.2083333333334|
|[30.0,9.33527,9.4...|               811|879.2083333333334|
|[30.0,10.4396,9.9...|               637|536.6666666666666|
+--------------------+------------------+-----------------+
only showing top 5 rows
Decision Tree Regression result:
Root Mean Squared Error (RMSE): 93.332
Mean Squared Error (MSE): 8710.86
Mean Absolute Error (MAE): 60.4492
R-squared: 0.944005
DecisionTreeRegressionModel (uid=DecisionTreeRegressor_4218b6c52f525a349c98) of depth 5 with 63 nodes
```

Decision Tree Regression result:
Root Mean Squared Error (RMSE): 93.332
Mean Squared Error (MSE): 8710.86
Mean Absolute Error (MAE): 60.4492
R-squared: 0.944005
DecisionTreeRegressionModel (uid=DecisionTreeRegressor_4218b6c52f525a349c98) of depth 5 with 63 nodes

The result is not as good as Linear Regression but still good.

How about trying Decision Tree Classifier?
In Decision Tree Classifier, we need to index label attribute (StringIndexer) into categorical variables.

```
# difference betwenn Classifier and Regression
labelIndexer = StringIndexer(inputCol="HousingMedianRent$",
outputCol="indexedLabel").fit(data)
```

The result shows

```
+--------------------+-----------+----------+
|            features|indexedLabel|prediction|
+--------------------+-----------+----------+
|[29.4,10.0699,14....|       72.0|       2.0|
|[29.6,6.82562,10....|      278.0|       1.0|
|[30.0,10.4396,9.9...|      254.0|       1.0|
|[30.2,9.37296,10....|      452.0|       2.0|
|[30.4,9.55198,9.9...|      539.0|       1.0|
+--------------------+-----------+----------+
only showing top 5 rows

Test Error = 0.97486
```

Test Error = 0.97486

Accuracy is very low, because we cannot change numeric into different classes to build the Decision Tree model.

Random Forest in MLlib

What is Random Forest?
A random forest is a meta estimator that fits several decision trees on various sub-samples of the dataset and use averaging to improve the predictive accuracy and control over-fitting.

In the code for Random Forest, we can just change the model to RandomForestRegressor in Decision Tree Regression.

```
dt = RandomForestRegressor(featuresCol="indexedFeatures", labelCol="HousingMedianRent$")
```

Result

```
+--------------------+-----------------+-----------------+
|            features|HousingMedianRent$|       prediction|
+--------------------+-----------------+-----------------+
|[29.6,6.94177,10....|              787| 765.367689330459|
|[29.8,9.00797,9.8...|              639|727.6617914602946|
|[30.0,9.33527,9.4...|              811|830.9129240992834|
|[30.0,10.4431,9.9...|              646|684.7464470317793|
|[30.1,9.40755,10....|              773|793.1010332199692|
+--------------------+-----------------+-----------------+
only showing top 5 rows

Random Forest Regression result:
Root Mean Squared Error (RMSE): 59.0543
Mean Squared Error (MSE): 3487.41
Mean Absolute Error (MAE): 39.9294
R-squared: 0.976438
RandomForestRegressionModel (uid=rfr_19429ebf3ba5) with 20 trees
```

Random Forest Regression result:

Root Mean Squared Error (RMSE): 59.0543
Mean Squared Error (MSE): 3487.41
Mean Absolute Error (MAE): 39.9294
R-squared: 0.976438
RandomForestRegressionModel (uid=rfr_19429ebf3ba5) with 20 trees

In the result, we can see when printing Random Forest, MLlib will show how many trees in this forest. As for the accuracy, Random Forest has better result than Decision Tree in every aspect.

Evaluation

Weka

	Correlation Coefficient	Mean Absolute Error	Root Mean Squared Error	Relative Absolute Error	Root Relative Squared Error
IBK-lazy	0.9908	28.0579	52.0729	9.718%	13.7521%

Figure (X) above shows the feature selection with the IBK-lazy (Weka's K-nearest neighbor) classifier. From the correlation coefficient, a sufficient positive correlation was shown between the top 50 attributes and the class label (Median Rent Price). Both RAE and RRSE were relatively low. However, MAE and RMSE are hard to evaluate without other classifiers to serve as a comparison. Future work will include at least one or two other classifiers in Weka's feature selection process.

Apache Spark

Compare results from different models

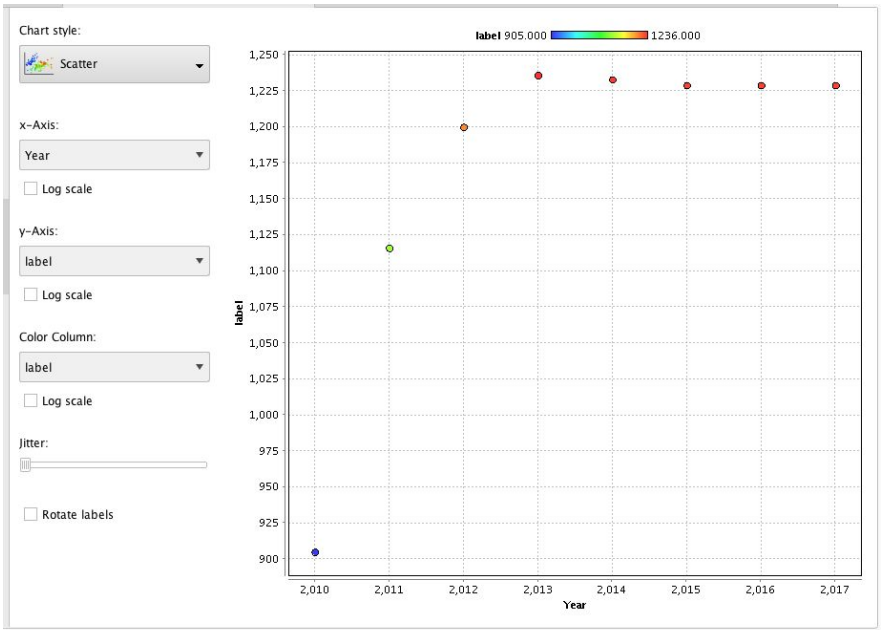
	RMSE	MSE	MAE	R-squared
Linear Regression	50.3514	2535.26	36.049	0.981682
Decision Tree	93.332	8710.86	60.4492	0.944005
Random Forest	59.0543	3487.41	39.9294	0.976438

As we can see, all three models have pretty good result, in the future, we are going to apply more models and keep testing our data.

The figure above depicts values from three different classifiers/regressions. Linear Regression outperformed both Decision Tree and Random Forest in all four measures of accuracy. Decision

Tree was notably the worst out of the three in all measures. Further investigation will be done in why a regression model outperformed a classifier.

RapidMiner



In Figure (X), the last data point is the predicted value of the Time Series analysis in RapidMIner. The analysis did not show a significant change in zip code 11211 (Williamsburg, Brooklyn) where we expected to see the most change in rent due to its reliance on the L train (Bedford Avenue is the first stop in Brooklyn on the L line). However, when we tried KNN, we believe there were not enough data points to properly evaluate or carry out the prediction. From these conclusions, we believe that Time Series, in theory, is the appropriate modeling approach but that it should be repeated using a software that can handle composite key values (year and zip code). By splitting up the zip codes, we lost a lot of critical information that rely on the interactions between multiple zip codes. Going forward, tools from the Apache ecosystem will be considered.

Deployment

Given our results, we believe our experiment should be repeated for more conclusive results using the proper tools that can handle a large dataset coupled with a predictive analytics question. Our modeling results should be considered pilot data with hypothesis generating ideas. Results from PCA and ChiSqSelector partially confirmed our hypothesis, in which we expected transportation to have a significant impact on rent decisions. Going forward, we would like to implement Time Series analysis in another software or tool, using all zip codes, years,

and attributes. These future results can be visualized as percent change and median rent prices overlaying a New York City map split by zip codes.