Questions

October 7, 2023

1 Predict House Prices

[1]:
```
# If additional packages are needed but are not installed by default, uncomment␣
↪the last two lines of this cell
# and replace <package list> with a list of additional packages.
# This will ensure the notebook has all the dependencies and works everywhere

import sys
!{sys.executable} -m pip install word2number
```

```
Requirement already satisfied: word2number in /opt/conda/lib/python3.9/site-
packages (1.1)
```

[2]:
```
# Libraries

# packages for data manipulation
import numpy as np
import pandas as pd
from word2number import w2n

# packages for data visualization
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator
import seaborn as sns

# packages for machine learning
import sklearn
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
↪f1_score, confusion_matrix
from sklearn.model_selection import train_test_split, RandomizedSearchCV
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
import xgboost as xgb

# miscellaneous packages
import warnings
warnings.filterwarnings('ignore')
```

```
pd.set_option("display.max_columns", 101)
pd.set_option('display.max_colwidth', 100)
pd.set_option('display.max_rows', 100)
```

```
/opt/conda/lib/python3.9/site-packages/xgboost/compat.py:36: FutureWarning:
pandas.Int64Index is deprecated and will be removed from pandas in a future
version. Use pandas.Index with the appropriate dtype instead.
  from pandas import MultiIndex, Int64Index
```

1.1 1. Data Description

Column	Description
id	Unique ID corresponding to the locality
income	Average monthly income in 1000s of people living in the locality
age	Average age of houses in the locality
rooms	Average number of rooms in the houses situated in the locality
bedrooms	Average number of bedrooms in the houses situated in the locality
people_per_house	Average number of people living in each house situated in the locality
location	Location of the locality represented as latitude and longitude separated by a delimiter
outcome	The predicted median price of a house in the locality (1 - High, 0 - Low)

[3]:
```
# The information dataset for the training set is already loaded below
data = pd.read_csv('train.csv')
data.head()
```

[3]:
```
   id  income          age     rooms  bedrooms  population  people_per_house  \
0   0  4.5493  forty-three  4.692308  1.026525      -639.0          2.923077
1   1  4.4391     fourteen  5.280561  1.034068      1150.0          2.304609
2   2  3.3333       eleven  6.410397  1.164159      2466.0          3.373461
3   3  3.2847         17.0  3.381720  1.188172       514.0          2.763441
4   4  1.4464         17.0  5.431034  1.534483       130.0          2.241379

         location  outcome
0  37.66;-122.43      1.0
1   33.68_-117.8      1.0
2  33.67_-116.31      0.0
3  34.24;-119.18      0.0
4  37.65,-120.46      0.0
```

1.1.1 1.1 Data Preprocessing

The first important thing to do is to understand the data structure, hence I printed the shape of the data, and observed that it has 7000 records (rows) and 9 fields (columns).

[4]:
```python
print(f'The shape of the data is {data.shape}.')
```

```
The shape of the data is (7000, 9).
```

Next, it is important to understand the quality of this dataset given that data quality is a crucial factor for successful model building.

One aspect of data quality is whether the data has missing values. By looking at the information below, we see that there is no missing values, indicating that we do not need to conduct missing value imputation to improve data quality.

[5]:
```python
# print the information (# missing values, data type) of the dataset
print(data.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7000 entries, 0 to 6999
Data columns (total 9 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   id                7000 non-null   int64
 1   income            7000 non-null   float64
 2   age               7000 non-null   object
 3   rooms             7000 non-null   float64
 4   bedrooms          7000 non-null   float64
 5   population        7000 non-null   float64
 6   people_per_house  7000 non-null   float64
 7   location          7000 non-null   object
 8   outcome           7000 non-null   float64
dtypes: float64(6), int64(1), object(2)
memory usage: 492.3+ KB
None
```

Another aspect of data quality is to inspect whether the variable types align with the information these variables actually offer. From the information table above, we see that the data types of most variables make intuitive sense (for example, `bedrooms`, i.e., average #bedrooms in house, should be a quantitative variable and hence its type should be float). However, the data types of `age` and `location` look suspicious as, intuitively, both of them should be represented in numbers, yet currently they are of type objects. Is this because they display different information as expected intuitively? Or do we need to process them before using them? We now look at these two variables one by one.

[6]:
```python
# print the age column
print(data['age'])
```

```
0       forty-three
```

```
1           fourteen
2             eleven
3               17.0
4               17.0
           …
6995       forty-six
6996            32.0
6997       fifty-two
6998            32.0
6999            22.0
Name: age, Length: 7000, dtype: object
```

It appears that the `age` column contains a mixture of ages in texts and ages in numbers, hence making it an object variable.

Since text data are usually converted into numerical values (for example, word2vec) before inputting them into a machine learning model, I decided to convert all the ages in texts to their corresponding ages in numbers. The following function does the transformation and converts `age` to a float.

[7]:
```
# for each row
for i in data.index:
    # if the first letter in the row is an alphabet
    if data['age'].str[0][i].isalpha() == True:
        # convert the value of this row to a number
        data['age'][i] = w2n.word_to_num(data['age'][i])

# change the data type of age to float
data['age'] = data['age'].astype(float)
```

[8]:
```
print(data['age'])
```

```
0       43.0
1       14.0
2       11.0
3       17.0
4       17.0
        …
6995    46.0
6996    32.0
6997    52.0
6998    32.0
6999    22.0
Name: age, Length: 7000, dtype: float64
```

From above we see that the `age` variable is now fully converted to a numerical variable.

Next, we observe that the `location` variable is a combination of longitudes and latitudes, separated by three types of delimiters `;`, `_`, and `,` - making it an object variable. For better practice of model building, I decided to split the location variable by delimiters into a longitude variable and a latitude variable, and convert them into floating points to get the precise location.

[9]:
```
print(data['location'])
```

```
0        37.66;-122.43
1         33.68_-117.8
2        33.67_-116.31
3        34.24;-119.18
4        37.65,-120.46
             …
6995      38.1,-122.23
6996     33.82;-118.37
6997     34.15;-118.15
6998     38.46_-122.69
6999     37.38,-120.72
Name: location, Length: 7000, dtype: object
```

The code below processes `location` as described. As the dataset is now prepared in its expected format, we are ready to proceed to the data exploration step.

[10]:
```
# replace the other two delimiters into ',' for convenience of handling
for delimiter in [';', '_']:
    data['location'] = data['location'].str.replace(delimiter, ',')

# split the location variable by delimiter and drop the original location␣
 ↪variable
data[['location_x', 'location_y']] = data['location'].str.split(',', expand =␣
 ↪True).astype(float)
data.drop('location', axis = 1, inplace = True)

# check that the location variable is processed correctly
print(data[['location_x', 'location_y']])
```

```
      location_x  location_y
0          37.66     -122.43
1          33.68     -117.80
2          33.67     -116.31
3          34.24     -119.18
4          37.65     -120.46
…            …           …
6995       38.10     -122.23
6996       33.82     -118.37
6997       34.15     -118.15
6998       38.46     -122.69
6999       37.38     -120.72

[7000 rows x 2 columns]
```

1.1.2 1.2 Univariate Analysis

After conducting the basic preprocessing of the data, we are now interested in the structure of each variable in the data, as they offer us useful information regarding what types of model we should use to better predict the target variable.

1.21 Numerical Variables I define numerical variables as variables that measure the quantity ('how much') of a certain entity - this definition aligns with the official definition of numerical variables.

Based on this definition, the numerical variables here are `income`, `age`, `rooms`, `bedrooms`, `population`, and `people_per_house`. For these variables, we are interested in their descriptive statistics as well as their distributions.

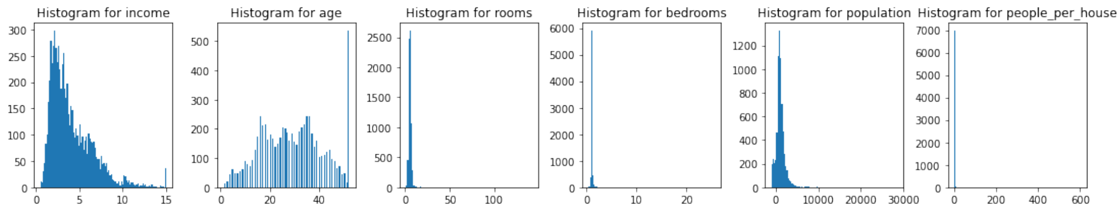
From the descriptive statistics table and the histograms below, we see that `income`, `rooms`, `bedrooms`, `population`, and `people_per_house` are all heavily skewed to the right, exhibiting rare cases of extremely large values. For example, while the median of `rooms` is approximately 5, the average number of rooms in house can be as large as 141! Similarly, while the median of `bedrooms` is approximately 1, it can be as many as 25.

The insight learned is that, during model building, it is better practice to use tree-based models (decision tree, random forest, gradient boosting, etc.) instead of models that require normal distribution of data as its statistical assumption (e.g., regression), because tree-based model is insensitive to outliers, while regression models are often very sensitive to outliers.

[11]:
```
# define numerical variables and print descriptive statistics
numeric_cols = ['income', 'age', 'rooms', 'bedrooms', 'population',␣
 ↪'people_per_house']
print(data[numeric_cols].describe())

# plot a histogram for each numerical variable
fig, axes = plt.subplots(1, 6, figsize = (15, 3))
for i, ax in enumerate(axes):
    var = numeric_cols[i]
    ax.hist(data[var], bins=100)
    ax.set_title(f'Histogram for {var}')

plt.tight_layout()
plt.show()
```

```
            income          age        rooms     bedrooms    population  \
count  7000.000000  7000.000000  7000.000000  7000.000000   7000.000000
mean      4.043755    29.374571     5.594687     1.105418   1122.459857
std       2.417744    12.592551     2.597206     0.477988   1133.401037
min       0.499900     1.000000     0.888889     0.444444  -1000.000000
25%       2.249450    19.000000     4.558694     1.009339    603.000000
50%       3.328400    29.000000     5.355294     1.055065   1005.000000
75%       5.344325    38.000000     6.276794     1.109562   1508.000000
max      15.000100    52.000000   141.909091    25.636364  28566.000000
```

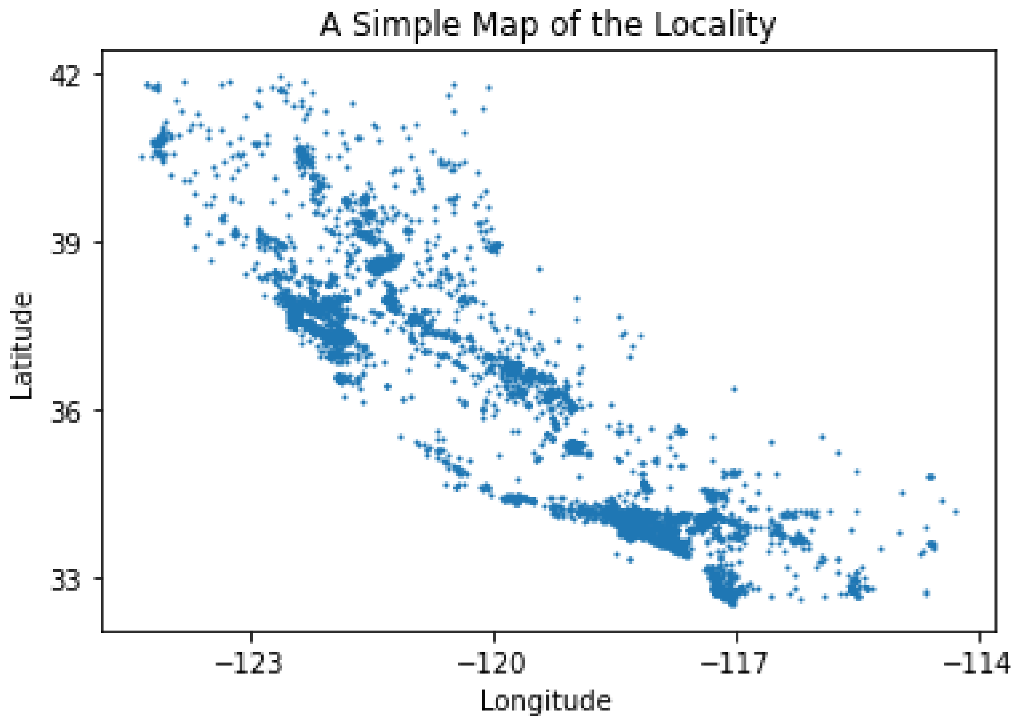
```
       people_per_house
count       7000.000000
mean           2.996824
std            9.341724
min            1.066176
25%            2.358961
50%            2.727432
75%            3.174603
max          599.714286
```



1.22 Identifier variable Besides numerical variables, there are also two identifiers in the dataset: `id`, which identifies the locality, and (`location_x`, `location_y`), which identifies the geographic location of the locality. We are interested in seeing where these localities are located at, hence I plotted a simple map below.

It can be seen that, in the map, some localities are close by, while some are more spread out. This raises a natural question: will the localities nearby each other exhibit similar median housing price? We shed some light into this question when we conduct multivariate analysis in the next section.

[12]:
```python
# plot a simple map by scattering longitude and latitude
fig, ax = plt.subplots()
ax.scatter(data['location_y'], data['location_x'], s = 1)
ax.set_xlabel('Longitude')
ax.set_ylabel('Latitude')
ax.xaxis.set_major_locator(MaxNLocator(nbins = 4))
ax.yaxis.set_major_locator(MaxNLocator(nbins = 4))
ax.set_title('A Simple Map of the Locality')
plt.show()
```



1.23 Binary variable Binary variable is defined as a variable that represents two states of an entity. In our dataset, the only binary variable is our target variable `outcome`, which is `1` when median house price in locality is high and `0` when median house price in locality is low.

For this variable, we are interested in the distribution of data across outcome groups, because if the distribution exhibits some imbalanced nature (e.g., significantly more localities with low median price than those with high median price), then we would need to use resampling techniques to deal with imbalanced data.

From the table below, we see that the data is approximately balanced, where the percentage of low-house-price localities is similar to that of high-house-price localities, hence we do not need to particularly resample the dataset during the model training process.

[13]:
```
# count of records in each outcome
counts = data['outcome'].value_counts()
# percentage of records in each outcome
percentages = data['outcome'].value_counts(normalize = True).mul(100).round(1).
 ↪astype(str)+'%'

# plot the distribution
print(pd.DataFrame({'Counts': counts, 'Percentage': percentages}))
```

```
     Counts Percentage
0.0    3581      51.2%
```

```
1.0    3419      48.8%
```

That said, we are now ready to examine the relationships between the variables.

1.1.3 1.3 Multivariate Analysis

We now look at the correlations between variables as they signal: 1. whether we need to conduct feature selection or choose appropriate models if there are multicollinearity or if there are redundant variables; 2. whether there are features that might be good predictors for our target variable.

From the correlation table and the correlation heatmap below, we see that most variables exhibit low correlations with other variables, except that `rooms` and `bedrooms` are highly correlated in a positive way - their correlation is as high as 0.83, suggesting that more average rooms in house in locality, more average bedrooms in house in locality. This further suggests that, to be cautious, we might want to use tree-based models since they are immune to multicollinearity.

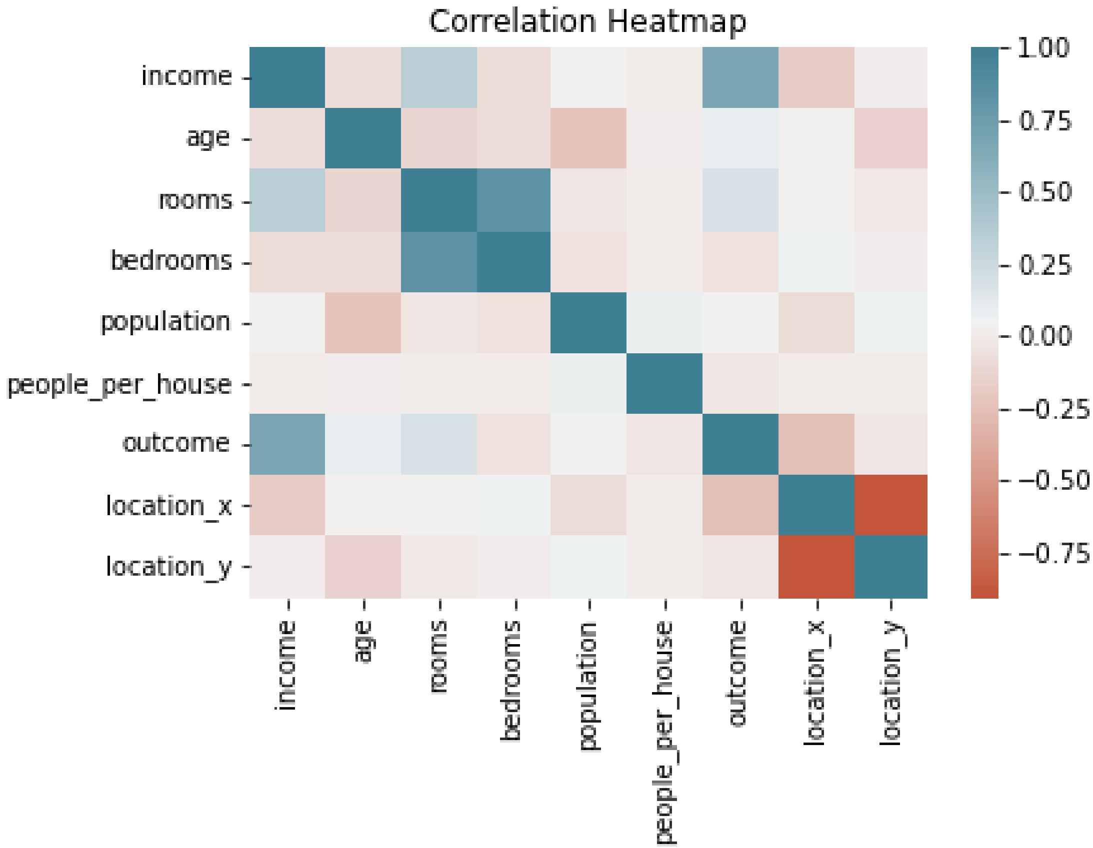
Note that it is natural for longitudes and latitudes to be highly correlated with each other as they together represent the location of a locality.

[14]:
```
print('Correlation between Variables')
print(data[data.columns[1:]].corr())
```

```
Correlation between Variables
                   income       age     rooms  bedrooms  population  \
income           1.000000 -0.069174  0.340798 -0.079305    0.028310
age             -0.069174  1.000000 -0.125984 -0.075029   -0.231015
rooms            0.340798 -0.125984  1.000000  0.834400   -0.038644
bedrooms        -0.079305 -0.075029  0.834400  1.000000   -0.058061
population       0.028310 -0.231015 -0.038644 -0.058061    1.000000
people_per_house -0.000574  0.008853  0.007153 -0.000638    0.072677
outcome          0.675644  0.086551  0.176931 -0.057833    0.028463
location_x      -0.179393  0.034098  0.036824  0.058751   -0.082463
location_y       0.013184 -0.139261 -0.004356  0.018021    0.065555

                  people_per_house   outcome  location_x  location_y
income                   -0.000574  0.675644   -0.179393    0.013184
age                       0.008853  0.086551    0.034098   -0.139261
rooms                     0.007153  0.176931    0.036824   -0.004356
bedrooms                 -0.000638 -0.057833    0.058751    0.018021
population                0.072677  0.028463   -0.082463    0.065555
people_per_house          1.000000 -0.027025    0.007750    0.004494
outcome                  -0.027025  1.000000   -0.250749   -0.024751
location_x                0.007750 -0.250749    1.000000   -0.906487
location_y                0.004494 -0.024751   -0.906487    1.000000
```

[15]:
```
# plot correlation heatmap
sns.heatmap(data[data.columns[1:]].corr(), cmap = sns.diverging_palette(20,220,␣
↪n = 200))
plt.title('Correlation Heatmap')
```

```
plt.show()
```

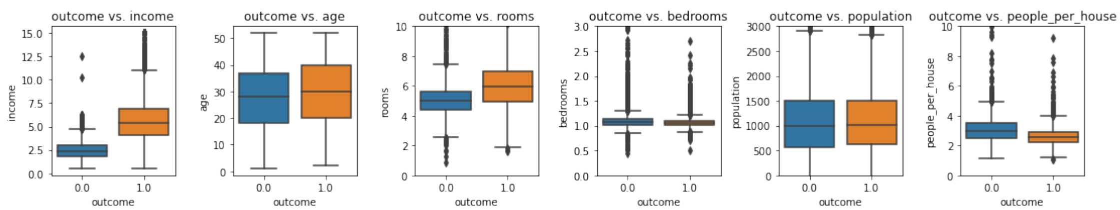


Next, we look at whether there are numerical variables in our dataset that appear as good predictors for the outcome. We do that by plotting boxplots of numerical variables across the outcome groups and examine if there are differences between groups (a more rigorous way to test this is to conduct ANOVA/t-tests).

From the boxplots below (truncated to get a better view), it appears that `income` might be a very important predictor, since the difference across groups is very salient. `rooms` might also be useful predictors. In contrast, it appears that `people_per_house`, `population`, `bedrooms`, and `age` do not exhibit salient difference across outcome groups.

[16]:
```
# create boxplot across outcome groups for each numerical variable
fig, axes = plt.subplots(1, 6, figsize = (15, 3))

for i, ax in enumerate(axes):
    var = numeric_cols[i]
    sns.boxplot(data['outcome'], data[var], ax = axes[i])
    ax.set_title(f'outcome vs. {var}')
```

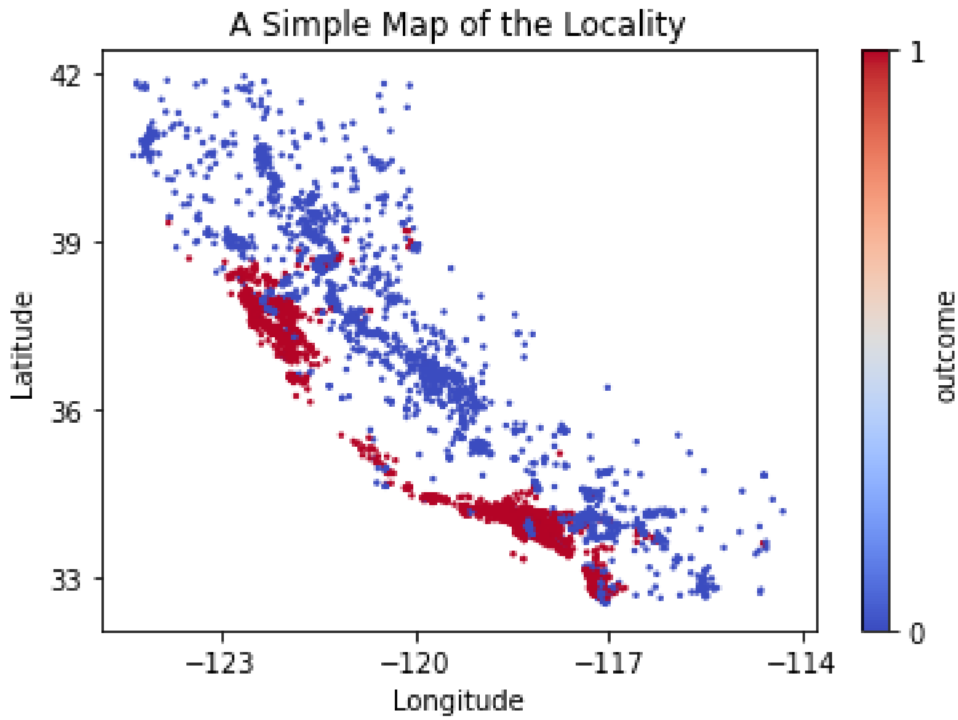
```
    # to get a better view, I truncated plots with extreme
    # outliers by limiting the upper bound as slightly
    # higher than the 75 percentile of the variable
    if i == 2:
        ax.set_ylim(0, 10)
    if i == 3:
        ax.set_ylim(0, 3)
    if i == 4:
        ax.set_ylim(0, 3000)
    if i == 5:
        ax.set_ylim(0, 10)

plt.tight_layout()
plt.show()
```



For the location-related variables, we check if they are important predictors by answering the question we had in **1.22**: do high/low-house-price localities tend to cluster together in similar locations? The map below signals that the answer is highly likely yes! High-house-price localities tend to be at the lower coast while low-price-localities tend to be at the middle and at the upper coast.

This indicates that we should definitely include location in our model given how salient the localities of two groups cluster.

[17]:
```
# plot a simple map by scattering longitude and latitude
# and adding outcome variable as an additional layer
fig, ax = plt.subplots()
sc = ax.scatter(data['location_y'], data['location_x'], s = 2, c =␣
 ↪data['outcome'], cmap = plt.cm.coolwarm)
plt.colorbar(sc, ticks = [0, 1], label = 'outcome')
ax.set_xlabel('Longitude')
ax.set_ylabel('Latitude')
ax.xaxis.set_major_locator(MaxNLocator(nbins = 4))
ax.yaxis.set_major_locator(MaxNLocator(nbins = 4))
ax.set_title('A Simple Map of the Locality')
plt.show()
```



1.2 2. Machine Learning

Build a machine learning model that can predict the outcome. - **The model's performance will be evaluated on the basis of Accuracy Score.**

1.2.1 2.1 Train-Test Preparation

To test if the model would work well, I splitted our current training set into a train set and a validation set. I then standardized the numerical variables in the train and validation sets separately using their own respective means and standard deviations.

[18]:
```
# separate features from target
X = data.drop(['outcome', 'id'], axis = 1)
y = data['outcome']

# split data to a train set and a validation set
X_train, X_valid, y_train, y_valid = train_test_split(X, y, test_size = 0.2,␣
 ↪random_state = 42)

# standardize train and validation sets separately
for col in numeric_cols:
    X_train[col] = (X_train[col] - X_train[col].mean())/X_train[col].std()
    X_valid[col] = (X_valid[col] - X_valid[col].mean())/X_valid[col].std()
```

1.2.2 2.2 Notes on Feature Engineering and Feature Selection

A traditional practice is to conduct feature engineering such as transforming numerical variables into more bell-shaped distributions via, for example, logging or Yeo-Johnson transformation. Another practice is to conduct feature selection using AIC/BIC, regularization, PCA, and so on.

The reason why I did not conduct transformation on numerical variables is because I've decided to use tree-based model which is insensitive to outliers, and the reason why I did not conduct feature selection is because I've decided to use XGBoost, which already incorporates regularization in its tree-building process.

1.2.3 2.3 XGBoost with RandomizedSearchCV

As specified previously, I decided to use XGBoost based on my understanding of the dataset so far:

1. The numerical variables of the dataset are heavily skewed with massive outliers, so I'd like to use tree-based models insensitive to outliers.
2. The dataset has features that are highly correlated, so I'd like to use tree-based models that are (generally) immune to multicollinearity.
3. I'd like a model that incorporates feature selection and prevents overfitting during its training process, and XGBoost incorporates regularization during its training.

In addition, to prevent the model from having a high variance, i.e., the model fails to generalize its predictive ability to other datasets, I've used the following techniques: 1. I used a 5-fold cross validation and selected the best model based on its average score over the 5 folds; 2. I conducted hyperparamter tuning by defining the hyperparameter space, and then using RandomizedSearchCV to examine different combinations of hyperparameters, and found the best set of hyperparameters that generated the most ideal score.

Note: Although GridSearchCV examines combinations of hyperparameters more thoroughly, it is also very expensive and very slow. In the future, a better practice is to first use Randomized-SearchCV to find a promising region of hyperparameter space, and then use GridSearchCV to thoroughly examine the combinations of hyperparameters in the promising region.

[19]:
```
# define hyperparameters
params_grid = {
    'n_estimators': [50, 100, 200],
    'learning_rate': [0.01, 0.05, 0.1],
    'booster': ['gbtree', 'gblinear'],
    'gamma': [0, 0.5, 1],
    'reg_alpha': [0, 0.5, 1],
    'reg_lambda': [0.5, 1, 5],
    'max_depth': [3, 5, 7, 10],
    'min_child_weight': [1, 3, 5],
    'subsample': [0.5, 0.7],
    'colsample_bytree': [0.5, 0.7]}
```

[20]:
```
# establish an xgb classifier and conduct RandomizedSearchCV on train set
clf = xgb.XGBClassifier()
```

```
random_search = RandomizedSearchCV(clf, param_distributions = params_grid,␣
 ↪n_iter = 50,
                                   scoring = 'accuracy', cv = 5, verbose = 0, n_jobs␣
 ↪= -1)
random_search.fit(X_train, y_train)
```

```
[09:31:35] WARNING: /home/conda/feedstock_root/build_artifacts/xgboost-
split_1645117766796/work/src/learner.cc:1115: Starting in XGBoost 1.3.0, the
default evaluation metric used with the objective 'binary:logistic' was changed
from 'error' to 'logloss'. Explicitly set eval_metric if you'd like to restore
the old behavior.
```

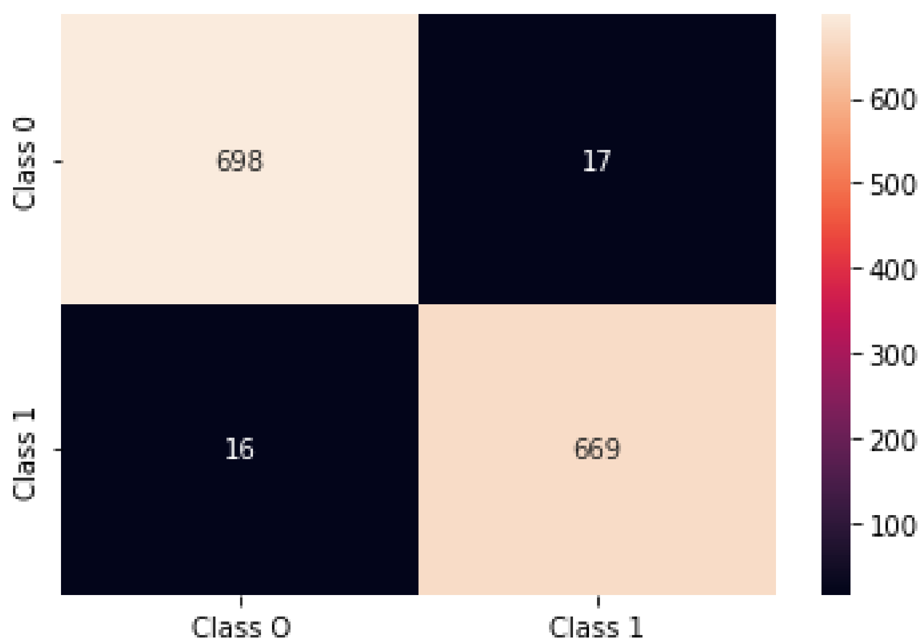
[20]:
```
RandomizedSearchCV(cv=5,
                   estimator=XGBClassifier(base_score=None, booster=None,
                                           colsample_bylevel=None,
                                           colsample_bynode=None,
                                           colsample_bytree=None,
                                           enable_categorical=False, gamma=None,
                                           gpu_id=None, importance_type=None,
                                           interaction_constraints=None,
                                           learning_rate=None,
                                           max_delta_step=None, max_depth=None,
                                           min_child_weight=None, missing=nan,
                                           monotone_constraints…
                                           validate_parameters=None,
                                           verbosity=None),
                   n_iter=50, n_jobs=-1,
                   param_distributions={'booster': ['gbtree', 'gblinear'],
                                        'colsample_bytree': [0.5, 0.7],
                                        'gamma': [0, 0.5, 1],
                                        'learning_rate': [0.01, 0.05, 0.1],
                                        'max_depth': [3, 5, 7, 10],
                                        'min_child_weight': [1, 3, 5],
                                        'n_estimators': [50, 100, 200],
                                        'reg_alpha': [0, 0.5, 1],
                                        'reg_lambda': [0.5, 1, 5],
                                        'subsample': [0.5, 0.7]},
                   scoring='accuracy')
```

Since this is a classification task, I used accuracy, precision, recall, and F1 score as the evaluation metrics. I then predicted the `outcome` variable in the validation set, and checked the evaluation metrics.

As can be seen from the numbers below, all four criteria exhibited extremely high score. The accuracy is nearly 98%, indicating that nearly 98% of instances are correctly classified in the validation set. The confusion matrix for the validation set also shows that the classification was successful: only about 30 records were misclassified.

[21]:
```python
# select the best performing model and predict the validation set
best_clf = random_search.best_estimator_
y_pred = best_clf.predict(X_valid)

# display model performance results on the validation set
print(f'Accuracy of the validation set is: {accuracy_score(y_valid, y_pred)}')
print(f'Precision of the validation set is: {precision_score(y_valid, y_pred)}')
print(f'Recall of the validation set is: {recall_score(y_valid, y_pred)}')
print(f'F1 score of the validation set is: {f1_score(y_valid, y_pred)}')

# plot confusion matrix
print('\nConfusion Matrix:')
cm = confusion_matrix(y_valid, y_pred)
sns.heatmap(cm, annot = True, fmt = 'g', xticklabels = ['Class 0', 'Class 1'],
            yticklabels = ['Class 0', 'Class 1'])
plt.show()
```

```
Accuracy of the validation set is: 0.9764285714285714
Precision of the validation set is: 0.9752186588921283
Recall of the validation set is: 0.9766423357664233
F1 score of the validation set is: 0.975929978118162

Confusion Matrix:
```



1.2.4 2.4 Training the Model using Full Dataset

As we have demonstrated that the model with hyperparameter tuning above works well, we now conduct the actual model training on the entire train dataset.

[22]:
```
# standardize the full training dataset
X_full = X.copy()
y_full = y.copy()
for col in numeric_cols:
    X_full[col] = (X_full[col] - X_full[col].mean())/X_full[col].std()

# run xgb with RandomizedSearchCV on the full dataset
clf_full = xgb.XGBClassifier()
random_search_full = RandomizedSearchCV(clf_full, param_distributions =␣
 ↪params_grid, n_iter = 50,
                                        scoring = 'accuracy', cv = 5, verbose =␣
 ↪0, n_jobs = -1)
random_search_full.fit(X_full, y_full)

# select the best performing model
best_clf_full = random_search_full.best_estimator_
```

```
[09:33:17] WARNING: /home/conda/feedstock_root/build_artifacts/xgboost-
split_1645117766796/work/src/learner.cc:1115: Starting in XGBoost 1.3.0, the
default evaluation metric used with the objective 'binary:logistic' was changed
from 'error' to 'logloss'. Explicitly set eval_metric if you'd like to restore
the old behavior.
```

We then look at the feature importance table: We see that the feature importance table aligns well with our insights extracted from the data visualization and exploration part: `income` is the most important factor when classifying a locality into high or low median housing price, while location plays another important role. As expected, `rooms` play a moderate role in separating the classes, while `people_per_house`, `bedrooms`, `age`, and `population` play fairly minor roles in predicting the classes of localities. This is another way to (sort of) showcase that our model was successful, as it captured what human eyes have captured.

[23]:
```
features_df = pd.DataFrame({'Feature:': X_full.columns,
                            'Importance': best_clf_full.feature_importances_})
print(features_df.sort_values(by = 'Importance', ascending = False))
```

```
           Feature:  Importance
0            income    0.468325
7        location_y    0.125911
2             rooms    0.112892
6        location_x    0.092509
5  people_per_house    0.088905
3          bedrooms    0.049092
1               age    0.042053
4        population    0.020312
```

Task:

- **Submit the predictions on the test dataset using your optimized model**
  Submit a CSV file with a header row plus each of the test entries, each on its own line.

The file (`submissions.csv`) should have exactly 2 columns:

Column	Description
id	Unique ID corresponding to the vehicle
outcome	The predicted median price of a house in the locality (1 - High, 0 - Low)

[24]:
```
test = pd.read_csv('test.csv')
test.head()
```

[24]:
```
     id  income        age     rooms  bedrooms  population  people_per_house  \
0  7000  4.8854       52.0  6.437186  1.030151      -915.0          2.444724
1  7001  1.7361  forty-two  3.000000  1.000000        26.0          1.857143
2  7002  5.2555       22.0  4.825199  1.217934      1581.0          1.794552
3  7003  2.3214       36.0  3.725694  0.940972      1385.0          4.809028
4  7004  5.2078       30.0  6.332317  1.042683      -864.0          2.487805

         location
0  37.33;-121.91
1   37.39_-121.9
2  34.05;-118.43
3   33.9_-118.19
4   37.0,-122.03
```

[25]:
```
# Conduct the same data manipulation as those that have been done in the
↪training dataset

# Process age
for i in test.index:
    # if the first letter in the row is an alphabet
    if test['age'].str[0][i].isalpha() == True:
        # convert the value of this row to a number
        test['age'][i] = w2n.word_to_num(test['age'][i])

# change the data type of age to float
test['age'] = test['age'].astype(float)

# Process location
# replace the other two delimiters into ',' for convenience of handling
```

```python
for delimiter in [';', '_']:
    test['location'] = test['location'].str.replace(delimiter, ',')

# split the location variable by delimiter and drop the original location␣
↪variable
test[['location_x', 'location_y']] = test['location'].str.split(',', expand =␣
↪True).astype(float)
test.drop('location', axis = 1, inplace = True)

# separate features from target
IDs = test['id']
X_test = test.set_index('id')

# standardize the test set
for col in numeric_cols:
    X_test[col] = (X_test[col] - X_test[col].mean())/X_test[col].std()

# predict the outcome of test set using the full model
y_test_pred = best_clf_full.predict(X_test)
```

[26]:
```python
# generate submission dataframe
submission_df = pd.DataFrame()
submission_df['id'] = IDs
submission_df['outcome'] = y_test_pred
```

[27]:
```python
#Submission
submission_df.to_csv('submissions.csv', index=False)
```