Part 2 — Workshop 3

TECH2: Introduction to Programming, Data, and Information Technology

Richard Foltyn
Norwegian School of Economics (NHH)

October 18, 2024

See GitHub repository for notebooks and data:
https://github.com/richardfoltyn/TECH2-H24

Contents

1 Exercise: House price levels and dispersion

For this exercise, we're using data on around 1,500 observations of house prices and house characteristics from Ames, a small city in Iowa.

1. Load the Ames housing data set from `ames_houses.csv` located in the `data/` folder.
2. Restrict the data to the columns `SalePrice` and `Neighborhood`.
3. Check that there are no observations with missing values in this data.
4. Compute the average house price (column `SalePrice`) by neighborhood (column `Neighborhood`). List the three most expensive neighborhoods, for example by using `sort_values()`.
5. You are interested to quantify the price dispersion in each neighborhood. To this end, compute the standard deviation by neighborhood using `std()`. Which are the three neighborhoods with the most dispersed prices?
6. An alternative measure of dispersion is the ratio of the 90th and 10th percentile of the house price distribution. Use the `quantile()` method to compute the P90 and P10 statistics by neighborhood, compute their ratio and print the three neighborhoods with the largest dispersion.

   Hint: The `quantile()` function takes *quantiles* as arguments, i.e., instead of the 90th percentile you need to specify the quantile as 0.9.

Solution.

Part (1)

[1]:
```
# Uncomment this to use files in the local data/ directory
DATA_PATH = '../data'

# Uncomment this to load data directly from GitHub
# DATA_PATH = 'https://raw.githubusercontent.com/richardfoltyn/TECH2-H24/main/data'
```

[2]:
```
import pandas as pd

# Path to Ames housing CSV file
fn = f'{DATA_PATH}/ames_houses.csv'

# Read in file
df = pd.read_csv(fn)
```

Part (2)

[3]:
```
# Keep only the columns SalePrice and Neighborhood
df = df[['SalePrice', 'Neighborhood']]
```

Part (3)

To check whether there are any missing values, we can for example use `info()`:

[4]:
```
N = len(df)
print(f'Total number of observations: {N:,d}\n')

# Print number of non-missing observations
df.info(show_counts=True)
```

```
Total number of observations: 1,460

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1460 entries, 0 to 1459
Data columns (total 2 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   SalePrice     1460 non-null   float64
 1   Neighborhood  1460 non-null   object
dtypes: float64(1), object(1)
memory usage: 22.9+ KB
```

Since the number of non-missing observations is the same as the number of total observations, there are no missing values in the data.

Part (4)

[5]:
```
# Group data by neighborhood
groups = df.groupby('Neighborhood')

# Compute mean house price by neighborhood
mean_price = groups['SalePrice'].mean()

# Print first 10 entries
mean_price.head(10)
```

[5]:
```
Neighborhood
Blmngtn    194870.882353
Blueste    137500.000000
BrDale     104493.750000
BrkSide    124834.051724
ClearCr    212565.428571
CollgCr    197965.773333
Crawfor    210624.725490
Edwards    128219.700000
Gilbert    192854.506329
IDOTRR     100123.783784
Name: SalePrice, dtype: float64
```

These means are not sorted, so we have to use `sort_values()` to sort them.

[6]:
```
# Sort in descending order, with highest values on top
mean_price = mean_price.sort_values(ascending=False)

# Print the 3 neighborhoods with the highest average price
mean_price.head(3)
```

[6]:
```
Neighborhood
NoRidge    335295.317073
NridgHt    316270.623377
StoneBr    310499.000000
Name: SalePrice, dtype: float64
```

If we are not interested in any of the intermediate objects, we can chain all these operations into a single line as follows:

[7]:
```
# Print the 3 neighborhoods with the most expensive average price
df.groupby('Neighborhood')['SalePrice'].mean().sort_values(ascending=False).head(3)
```

[7]:
```
Neighborhood
NoRidge    335295.317073
NridgHt    316270.623377
StoneBr    310499.000000
Name: SalePrice, dtype: float64
```

Part (5)

Computing the standard deviation and sorting in descending order is performed in exactly the same way as for the mean, so we just adapt the last line:

[8]:
```
# Print the 3 neighborhoods with the most expensive average price
df.groupby('Neighborhood')['SalePrice'].std().sort_values(ascending=False).head(3)
```

[8]:
```
Neighborhood
NoRidge    121412.658640
StoneBr    112969.676640
NridgHt     96392.544954
Name: SalePrice, dtype: float64
```

Part (6)

We first compute the P90 and P10 separately, then compute their ratio and sort the results.

[9]:
```
# Compute the 90th percentile of house prices by neighborhood
P90 = df.groupby('Neighborhood')['SalePrice'].quantile(0.9)
```

```
# Compute the 10th percentile of house prices by neighborhood
P10 = df.groupby('Neighborhood')['SalePrice'].quantile(0.1)

# Compute ratio of percentiles, P90/P10
P90_P10 = P90 / P10

# Print first 5 entries (unsorted)
P90_P10.head(5)
```

[9]:
```
Neighborhood
Blmngtn     1.453749
Blueste     1.170481
BrDale      1.395617
BrkSide     2.309796
ClearCr     1.835535
Name: SalePrice, dtype: float64
```

[10]:
```
# Sort values in descending order and print the top 3 neighborhoods
P90_P10.sort_values(ascending=False).head(3)
```

[10]:
```
Neighborhood
IDOTRR      2.546182
StoneBr     2.533834
BrkSide     2.309796
Name: SalePrice, dtype: float64
```

With the help of a lambda expression, we can also directly compute the P90/P10 ratio in a single operation as follows:

[11]:
```
# Compute P90/P10 and sort in a single line
df.groupby('Neighborhood')['SalePrice'].agg(lambda x: x.quantile(0.9) / x.quantile(0.1)).
↪sort_values(ascending=False).head(3)
```

[11]:
```
Neighborhood
IDOTRR      2.546182
StoneBr     2.533834
BrkSide     2.309796
Name: SalePrice, dtype: float64
```

---

2 Exercise: Determinants of house prices

For this exercise, we're using data on around 1,500 observations of house prices and house characteristics from Ames, a small city in Iowa.

1. Load the Ames housing data set from `ames_houses.csv` located in the `data/` folder.
2. Restrict the data to the columns `SalePrice`, `LotArea` and `Bedrooms`.
3. Restrict your data set to houses with one or more bedrooms and a lot area of at least 100m^2.
4. Compute the average lot area. Create a new column `LargeLot` which takes on the value of 1 if the lot area is above the average (*"large"*), and 0 otherwise (*"small"*).

   What is the average lot area within these two categories?
5. Create a new column `Rooms` which categorizes the number of `Bedrooms` into three groups: 1, 2, and 3 or more. You can create these categories using boolean indexing, `np.where()`, pandas's `where()`, or some other way.
6. Compute the mean `SalePrice` within each group formed by `LargeLot` and `Rooms` (for a total of 6 different categories) using `groupby()`.

7. Compute and report the average price difference between 1 and 2 bedrooms for a house with a small lot area.
8. Compute and report the average price difference between a small and a large lot for a house with 2 bedrooms.

Solution.

Part (1)

[12]:
```
# Uncomment this to use files in the local data/ directory
DATA_PATH = '../data'

# Uncomment this to load data directly from GitHub
# DATA_PATH = 'https://raw.githubusercontent.com/richardfoltyn/TECH2-H24/main/data'
```

[13]:
```
import pandas as pd

# Path to Ames housing CSV file
fn = f'{DATA_PATH}/ames_houses.csv'

# Read in file
df = pd.read_csv(fn)
```

Part (2)

[14]:
```
# Restrict DataFrame to columns used in this exercise
df = df[['SalePrice', 'LotArea', 'Bedrooms']]
```

Part (3)

[15]:
```
# Drop observations with zero bedrooms and small lot areas
df = df.query('Bedrooms > 0 & LotArea > 100').copy()
```

Part (4)

[16]:
```
# Compute mean lot area
mean_area = df['LotArea'].mean()

print(f'Average lot area: {mean_area:.1f}')
```

```
Average lot area: 974.0
```

We create the `LargeLot` indicator as the result of a logical comparison. Note that this creates a boolean data type, i.e., one with values `True` and `False`. We could additionally convert this column to type `int` to obtain 0's (`False`) and 1's (`True`) instead, but it does not change any of the computations below.

[17]:
```
# Create indicator for whether lot is above average in size ("large")
df['LargeLot'] = (df['LotArea'] > mean_area)

# Alternatively, we can force the column LargeLot to be an integer with 0/1:
# df['LargeLot'] = (df['LotArea'] > mean_area).astype(int)

# Compute and print average lot size in the large/small categories
df.groupby('LargeLot')['LotArea'].mean()
```

[17]: LargeLot
False   703.521658
True   1452.681617
Name: LotArea, dtype: float64

Part (5)

There are several ways to recode the `Bedrooms` column into the categories 1, 2, and 3 or more.

[18]:
```
# Alternative using boolean indexing
df['Rooms'] = df['Bedrooms']
three_plus = (df['Rooms'] >= 3)
# Replace all observations with 3 or more bedrooms with the value 3
df.loc[three_plus, 'Rooms'] = 3
```

[19]:
```
# Alternative using DataFrame.where()
df['Rooms'] = df['Bedrooms'].where(df['Bedrooms'] <= 2, 3)
```

[20]:
```
import numpy as np

# Alternative using np.where()
df['Rooms'] = np.where(df['Bedrooms'] <= 2, df['Bedrooms'], 3)
```

We can use `pd.crosstab()` to verify that the mapping of rooms worked as intended:

[21]:
```
# Cross-tabulate the new column Rooms vs. Bedrooms
pd.crosstab(df['Bedrooms'], df['Rooms'])
```

[21]:

Rooms	1	2	3
Bedrooms			
1	50	0	0
2	0	358	0
3	0	0	804
4	0	0	213
5	0	0	21
6	0	0	7
8	0	0	1

Part (6)

[22]:
```
# Compute mean house price within each category
mean_prices = df.groupby(['LargeLot', 'Rooms'])['SalePrice'].mean()
mean_prices
```

[22]: LargeLot Rooms
False   1   135182.388889
          2   144739.841379
          3   164034.885572
True   1   270825.357143
          2   215591.294118
          3   222596.090293
Name: SalePrice, dtype: float64

Part (7)

[23]:
```
# Difference of average sales price of homes with 2 vs 1 bedrooms for small lot area
diff = mean_prices.loc[False, 2] - mean_prices.loc[False, 1]

print(f'Price diff for 2 vs. 3 bedrooms for small lot: {diff:,.0f} USD')
```

```
Price diff for 2 vs. 3 bedrooms for small lot: 9,557 USD
```

Part (8)

[24]:
```
# Difference of average sales price of homes with 3 rooms for large vs. small lot area
diff = mean_prices.loc[True, 2] - mean_prices.loc[False, 2]

print(f'Price diff for large vs. small lot with 2 bedrooms: {diff:,.0f} USD')
```

```
Price diff for large vs. small lot with 2 bedrooms: 70,851 USD
```

---

3 Exercise: Inflation and unemployment in the US

In this exercise, you'll be working with selected macroeconomic variables for the United States reported at monthly frequency obtained from FRED. The data set starts in 1948 and contains observations for a total of 864 months.

1. Load the data from the file `FRED_monthly.csv` located in the `data/` folder. Print the first 10 observations to get an idea how the data looks like.
2. Keep only the columns `Year`, `Month`, `CPI`, and `UNRATE`. Moreover, perform this analysis only on observations prior to 1970 and drop the rest.
3. Since pandas has great support for time series data, we want to create an index based on observation dates.
   - To this end, use `to_datetime()` to convert the `Year` and `Month` columns into a date.

     Hint: `to_datetime()` requires information on Year/Month/Day, so you need to create a `Day` column first and assign it a value of 1. You can then call `to_datetime()` with the argument `df[['Year', 'Month', 'Day']]` to create the corresponding date.
   - Store the date information in the column `Date`. Delete the columns `Year`, `Month` and `Day` once you are done as these are no longer needed.
   - Set the `Date` column as the index for the `DataFrame` using `set_index()`.
4. The column `CPI` stores the consumer price index for the US. You may be more familiar with the concept of inflation, which is the percent change of the CPI relative to the previous period. Create a new column `Inflation` which contains the *annual* inflation *in percent* relative to the same month in the previous year by applying `pct_change()` to the column `CPI`.

   Hints:

   - Since this is monthly data, you need to pass the arguments `periods=12` to `pct_change()` to get annual percent changes.
   - You need to multiply the values returned by `pct_change()` by 100 to get percent values.
5. Compute the average unemployment rate (column `UNRATE`) over the whole sample period. Create a new column `UNRATE_HIGH` that contains an indicator whenever the unemployment rate is above its average value (*"high unemployment period"*).

- How many observations fall into the high- and the low-unemployment periods?
- What is the average unemployment rate in the high- and low-unemployment periods?

6. Compute the average inflation rate for high- and low-unemployment periods. Is there any difference?
7. Use `resample()` to aggregate the inflation data to annual frequency and compute the average inflation within each calendar year.

   Which are the three years with the highest inflation rates in the sample?

   Hint: Use the resampling rule `'YE'` when calling `resample()`.

Solution.

Part (1)

[25]:
```
# Uncomment this to use files in the local data/ directory
DATA_PATH = '../data'

# Uncomment this to load data directly from GitHub
# DATA_PATH = 'https://raw.githubusercontent.com/richardfoltyn/TECH2-H24/main/data'
```

[26]:
```
import pandas as pd

# Path to monthly FRED data
fn = f'{DATA_PATH}/FRED_monthly.csv'

# Read in file
df = pd.read_csv(fn)

# Print first 10 observations
df.head(10)
```

[26]:

	Year	Month	CPI	UNRATE	FEDFUNDS	REALRATE	LFPART
0	1948	1	23.7	3.4	NaN	NaN	58.6
1	1948	2	23.7	3.8	NaN	NaN	58.9
2	1948	3	23.5	4.0	NaN	NaN	58.5
3	1948	4	23.8	3.9	NaN	NaN	59.0
4	1948	5	24.0	3.5	NaN	NaN	58.3
5	1948	6	24.2	3.6	NaN	NaN	59.2
6	1948	7	24.4	3.6	NaN	NaN	59.3
7	1948	8	24.4	3.9	NaN	NaN	58.9
8	1948	9	24.4	3.8	NaN	NaN	58.9
9	1948	10	24.3	3.7	NaN	NaN	58.7

Part (2)

[27]:
```
# Keep only columns of interest for this analysis
df = df[['Year', 'Month', 'CPI', 'UNRATE']]

# Keep only periods before 1970
df = df.query('Year < 1970')
```

Part (3)

```
[28]: # Create Day information required by to_datetime(). Since this is monthly data,
      # the day does not really matter and we simply set it to 1.
      df['Day'] = 1

      # Create a date observation from the individual date components
      df['Date'] = pd.to_datetime(df[['Year', 'Month', 'Day']])

      # Delete the date component columns
      df = df.drop(columns=['Year', 'Month', 'Day'])

      # Set the Date column as the index
      df = df.set_index('Date')

      # Print first 5 obs to confirm that things look OK
      df.head(5)
```

```
[28]:              CPI  UNRATE
      Date
      1948-01-01  23.7     3.4
      1948-02-01  23.7     3.8
      1948-03-01  23.5     4.0
      1948-04-01  23.8     3.9
      1948-05-01  24.0     3.5
```

Part (4)

```
[29]: # Compute inflation as the percent change of the CPI
      df['Inflation'] = df['CPI'].pct_change(periods=12) * 100

      # Print first 15 observations
      df.head(15)
```

```
[29]:              CPI  UNRATE  Inflation
      Date
      1948-01-01  23.7     3.4        NaN
      1948-02-01  23.7     3.8        NaN
      1948-03-01  23.5     4.0        NaN
      1948-04-01  23.8     3.9        NaN
      1948-05-01  24.0     3.5        NaN
      1948-06-01  24.2     3.6        NaN
      1948-07-01  24.4     3.6        NaN
      1948-08-01  24.4     3.9        NaN
      1948-09-01  24.4     3.8        NaN
      1948-10-01  24.3     3.7        NaN
      1948-11-01  24.2     3.8        NaN
      1948-12-01  24.0     4.0        NaN
      1949-01-01  24.0     4.3   1.265823
      1949-02-01  23.9     4.7   0.843882
      1949-03-01  23.9     5.0   1.702128
```

Note that the first 12 observations of `Inflation` are missing since it is not possible to compute 12-month percent changes due to missing data.

Part (5)

[30]:
```
# Compute and report average unemployment rate
unrate_avg = df['UNRATE'].mean()
print(f'Average unemployment rate: {unrate_avg:.1f}%')
```

```
Average unemployment rate: 4.7%
```

[31]:
```
# Create indicator for above-average unemployment rate
df['UNRATE_HIGH'] = df['UNRATE'] > unrate_avg

# Tabulate number of periods with above and below-average unemployment
df['UNRATE_HIGH'].value_counts()
```

[31]:
```
UNRATE_HIGH
False    141
True     123
Name: count, dtype: int64
```

[32]:
```
# Tabulate average unemployment rate in high- and low-unemployment periods
df.groupby('UNRATE_HIGH')['UNRATE'].mean()
```

[32]:
```
UNRATE_HIGH
False    3.697872
True     5.781301
Name: UNRATE, dtype: float64
```

Part (6)

[33]:
```
# Compute average inflation in high- and low-unemployment periods
df.groupby('UNRATE_HIGH')['Inflation'].mean()
```

[33]:
```
UNRATE_HIGH
False    3.110456
True     0.942056
Name: Inflation, dtype: float64
```

Part (7)

[34]:
```
# Create groups based on calendar year
groups = df.resample('YE')

# Compute average inflation in each year
infl_avg = groups['Inflation'].mean()

# Sort in descending order and print the three years with highest average inflation
infl_avg.sort_values(ascending=False).head(3)
```

[34]:
```
Date
1951-12-31    7.987456
1969-12-31    5.432647
1968-12-31    4.241319
Name: Inflation, dtype: float64
```

Alternatively, you can perform these actions in one line:

[35]:
```
df.resample('YE')['Inflation'].mean().sort_values(ascending=False).head(3)
```

[35]:
```
Date
1951-12-31    7.987456
1969-12-31    5.432647
1968-12-31    4.241319
Name: Inflation, dtype: float64
```