Visualizing and Processing Data

Week 9

Natalie Agus

Learning Objectives

- Give **example** application of linear regression and classification
- Create a **Pandas** DataFrame and selecting data from DataFrame
- Using **Pandas** library to **read** CSV file
- **Split** data randomly into training set and testing set
- **Normalize** data using z-normalization and min-max normalization
- **Convert** Pandas Dataframe to NumPy Array
- **Selecting** data from NumPy Array
- Use **mathematical**, statistical and linear algebra functions on NumPy Array
- Creating **new** Numpy Arrays
- Create **scatter** plot and other **statistical** plots like box plot, histogram, and bar plot

Short Introduction to Machine Learning

Supervised ML

cat	not a cat
	

Reading Data

- csv data

```
import pandas as pd
```

```
df = pd.read_csv('mydata.csv')
```

- Series: 1d **labeled array**
- DataFrame: 2d **labeled** data structure
  - Access column of DataFrame: we get a **view** as a Series
  - view vs copy

```
df[column_name]
```

```
print(df['resale_price'])
print(type(df['resale_price']))
```

Selecting Data: [ ], .loc, or .iloc

```
df.loc[:, 'resale_price']
```

- Access all rows, of column 'resale_price', we get a **view** of the Series

```
# we get a Series type here
print("Get resale_price column only:")
print(df['resale_price'])
print(type(df['resale_price']))
print('---------------------')

# using .loc to access a column
print(df.loc[:, 'resale_price'])
print(type(df.loc[:, 'resale_price']))
print('---------------------')

# using .loc to access the first row
print("Get first row of df only:")
print(df.loc[0, :])
print(type(df.loc[0, :]))
print('---------------------')

# cast to DataFrame
print("Get first row of df only as Dataframe:")
df_row0 = pd.DataFrame(df.loc[0, :])
print(df_row0)
print(type(df_row0))
print('---------------------')
```

```
df[column_name]
```

```
print(df['resale_price'])
print(type(df['resale_price']))
```

[ ] access vs .loc access

- `df[col_name]` and `df.loc[:, col_name]` returns the **same** result
  - `df[colname]`: Column Access via `__getitem__`
  - `df.loc[:, colname]`: Access via `.loc` (*property*), returns an **indexer** object which you can access using [ ] notation after it
- Having both: allows for **intuitive** data manipulation
  - [ ]: **simple** access, direct to access column(s) and all rows within those column(s)
  - .loc allows more **complex** op: select multiple rows and cols simultaneously + filter

SettingWithCopy Warning

- If you attempt to modify the **result** of `df.loc[:, columns_names]`, e.g:
  - `df_new = df.loc[:, columns_names]` ---> `df_new` can be a View
  - Or are you modifying the **original** df?
  - Or are you modifying the ***new*** view `df_new`?
- To ensure you *are* working with a **copy**, explicitly **copy** the original DataFrame:
  - `df_copy = df.loc[:, columns_names].copy()`

DANGER

What is this `SettingWithCopyWarning`?

This error came about because of this line: `df_tampines = df.loc[df['town'] == 'TAMPINES',:]`. In the next line, you're attempting to change `df_tampines['resale_price_1000']` which can possibly just be a "copy" (a "view") of the original dataframe.

In this case we are not doing a chained assignments so it's okay to do it and this warning is simply a **false positive**. However it might be dangerous to do so at other times, especially if you meant to modify the original dataframe with **chained assignments** like `df_tampines.loc[917]['town'] = "NEW_TAMPINES"`, as this will not modify `df_tampines` at all since you're applying the change on a "view": `df_tampines.loc[917]`. A correct way will be to apply the change as a **single assignment**: `df_tampines.loc[917, 'town'] = "NEWTAMPINES"`.

Give this article a read, and also this article.

> You can try to silence the warning it by using `copy`: `df_tampines = df.loc[df['town'] == 'TAMPINES',:].copy()`. This way you explicitly create a new dataframe called `df_tampines`.

.loc vs .iloc

```
df_new = df.loc[0:9, :]
```

- .loc access **labels**, **not** index. 0-9 above are **labels**
- You get a DataFrame if you need 2d data structure as a result

A similar output can be obtained without `.loc`:

```
df[columns]
```

```
df.loc[0:10, columns]
```

- Use iloc if you want to index the rows and columns (0-10 below are **positions**)

```
columns = [1, 3, -1]
df.iloc[0:10, columns]
```

main.py

```python
import pandas as pd

file_url = 'https://www.dropbox.com/s/jz8ck0obu9u1rng/resale-flat-pri
df = pd.read_csv(file_url)

# select the first 10 rows only
df_out = df.loc[0:10, :]
print("First 10 rows of df:")
print(df_out)
print(type(df_out))
print('---------------------')

# select some columns only
columns = ['town', 'block', 'resale_price']
print("Selected 3 cols of df:")
df_out = df.loc[:, columns]
print(df_out)
print(type(df_out))
print('---------------------')

# similar to the above, select some columns only
df_out = df[columns]
print(df_out)
print(type(df_out))
print('---------------------')

# similar to the above, select some columns only, but only the first
df_out_1 = df.loc[0:10, columns]
print(df_out)
print(type(df_out))
print('---------------------')

# getting by column index
columns = [1, 3, -1]
df_out = df.iloc[0:10, columns]
print(df_out)
print(type(df_out))
print('---------------------')
```

Conditional Select

```
df.loc[df['resale_price'] > 500_000, columns]
```

```
df.loc[(df['resale_price'] >= 500_000) & (df['resale_price'] <= 600_000), columns]
```

```
df.loc[(df['resale_price'] >= 500_000) & (df['resale_price'] <= 600_000) &
       (df['town'] == 'ANG MO KIO') &
       (df['block'] >= '300') & (df['block'] < '500'), columns]
```

- The parenthesis separating the two **AND** conditions are **compulsory**

Series and DataFrame Functions

- Creation

```
price = df['resale_price']
print(isinstance(price, pd.Series)) # True
price_df = pd.DataFrame(price)
print(isinstance(price_df, pd.DataFrame)) # True
print(price_df)
```

```
new_series = pd.Series(list(range(2,101)))
print(isinstance(new_series, pd.Series)) # True
print(new_series)
```

- Copying

```
df_1 = df  # shallow copy of df
df_2 = df.copy # deep copy of df
print(df_1 is df) # True
print(df_2 is df) # False
```

- Statistical function: .describe(), mean(), std(), etc

```
print(df['resale_price'].mean()) # 446724.22886801313
print(df['resale_price'].std()) #  155297.43748684428
print(df['resale_price'].min()) # 140000.0
print(df['resale_price'].max()) # 1258000.0
print(df['resale_price'].quantile(q=0.75)) # 525000.0
```

Shallow vs Deep Copy

Matters when you have nested objects

```python
import copy

original: list[list[int]] = [[1, 2, 3], [4, 5, 6]]
shallow_copy: list[list[int]] = copy.copy(original)

#  A shallow copy creates a new object, but it only copies the references to the original object's elements, NOT the elements themselves.
shallow_copy[0][0] = 10
print(original)      # Output: [[10, 2, 3], [4, 5, 6]]
print(shallow_copy)  # Output: [[10, 2, 3], [4, 5, 6]]


# A deep copy creates a new object and recursively copies all objects found within the original object, creating entirely separate copies of any nested objects.
deep_copy: list[list[int]] = copy.deepcopy(original)

deep_copy[0][0] = 10
print(original)      # Output: [[1, 2, 3], [4, 5, 6]]
print(deep_copy)     # Output: [[10, 2, 3], [4, 5, 6]]

```

Axis

```
df.mean(axis=1)
```

- axis=0: over all rows (default)
  - e.g: if you compute mean with axis=0, you get a value of mean in each **column**. The mean is computed over all rows
- axis=1: over all columns
  - e.g: if you compute mean with axis=01 you get a value of mean in each **row**. The mean is computed over all columns
- Be careful: ensure all data is **numeric** before calling statistical functions

Transpose

Change column into rows, and rows into column

```
df_row0 = pd.DataFrame(df.loc[0, :])
```

```
df_row0_transposed = df_row0.T
```

Vector Operations (very useful)

```
def divide_by_1000(data):
    return data / 1000

df['resale_price_in1000'] = df['resale_price'].apply(divide_by_1000)
```

```
df['resale_price_in1000'] = df['resale_price'].apply(lambda data: data/1000)
```

- `.apply()` returns a **NEW** series/dataframe object that contains the transformed data
- it is **different** from .loc, .iloc, or using brackets ([])

Normalization

To ensure that different features contribute equally to a model's learning process

- **Z normalization**
  - Also known as standardization
  - Range: between 0 and 1
  - Used when features have different variances / outliers are present, assuming data is normally distributed
- **Min-max normalization**
  - Range: can also be between 0 and 1, or other fixed scale
  - Used when the distribution shape of data does not matter, and when things should be in **bounded range,** and outliers are not a concern

Splitting Dataset

train set, validation set, test set

- **training dataset**: used to **build** the model and compute the **parameters**
- **validation dataset:** used to evaluate the model for various parameters and to choose the optimum parameter
- **test dataset**: used to evaluate the model built with the optimum parameter found previously.
  - Used only **once** at the end once the **optimum** params are found with validation dataset
  - !! DO **NOT** use test dataset to fine-tune your parameter.

Splitting Dataset

train set, validation set, test set

- **training dataset: 50%**
- **validation dataset: 30%**
- **test dataset: 20%**
- **Using validation dataset:**
  - re-train and tweak the training parameter
  - retrain with **training** dataset
  - ***evaluate*** *with validation test,* ***repeat*** *until satisfactory*
- Once satisfactory, use **test** set for final evaluation

NumPy Array

- Convert Pandas df to NumPy

```
df = pd.read_csv('mydata.csv')
array = df.to_numpy()
```

- NumPy more **plain,** no **automatic row index**, no **column names**
  - Can represent **higher dimension** (Pandas df only represents 2D)
- array.shape
- array[0], array[0, :], array[:, 0], array[:10, 1:3]

```
zeros = np.zeros(5)
```

```
zero_matrix = np.zeros((2,3))
```

```
list_matrix = [[1, 2, 3],
               [4,5,6]]
array_matrix = np.array(list_matrix)
```

```
>>> array_matrix
array([[1, 2, 3],
       [4, 5, 6]])
>>> np.sum(array_matrix)
21
```

```
>>> np.sum(array_matrix, axis=0)
array([5, 7, 9])
```

```
>>> np.sum(array_matrix, axis=1)
array([ 6, 15])
```

NumPy

- np.mean(), np.median(), np.std()
  - Can define the axis
- np.matmul()
- Matrix operation & broadcasting
  - Can add if #columns are different (but not #row)

```
list_matrix = [[1, 2, 3],
               [4,5,6]]
array_matrix = np.array(list_matrix)
```

```
>>> np.mean(array_matrix, axis=0)
array([2.5, 3.5, 4.5])
>>> np.mean(array_matrix, axis=1)
array([2., 5.])
```

```
>>> ones = np.ones((5,1))
>>> ones + 1
array([[2.],
       [2.],
       [2.],
       [2.],
       [2.]])
```

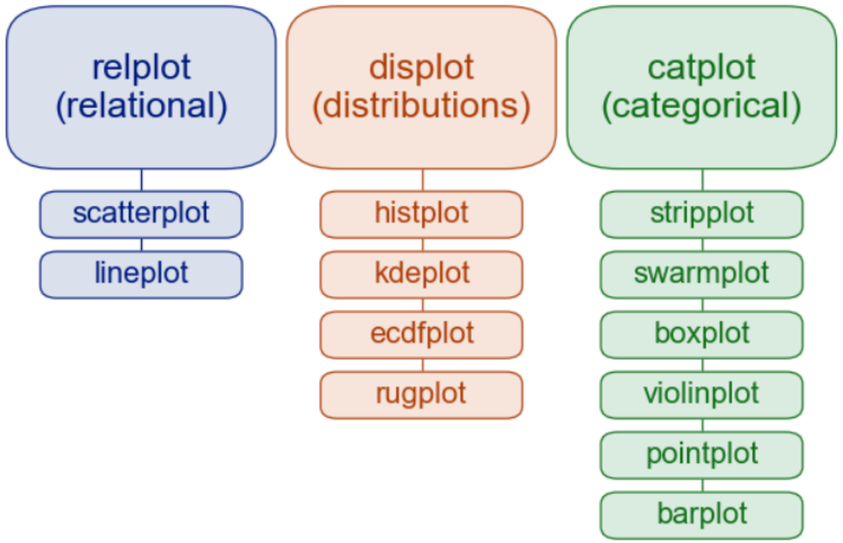
```
>>> np.ones((5,1)) + np.ones((7,1))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (5,1) (7,1)
```

```
>>> np.ones((2,1)) + array_matrix
array([[2., 3., 4.],
       [5., 6., 7.]])
```

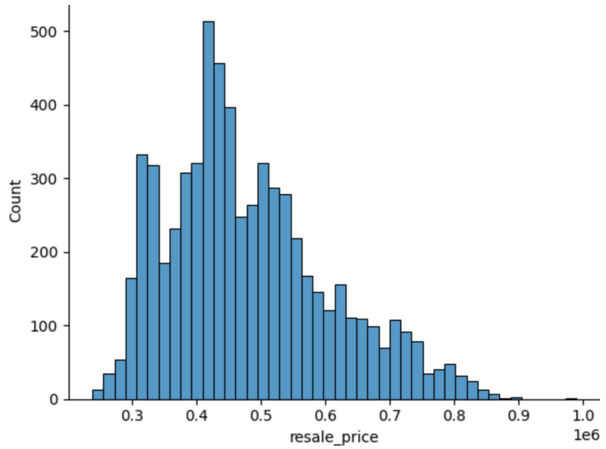
Data Visualization

Matplotlib, Seaborn

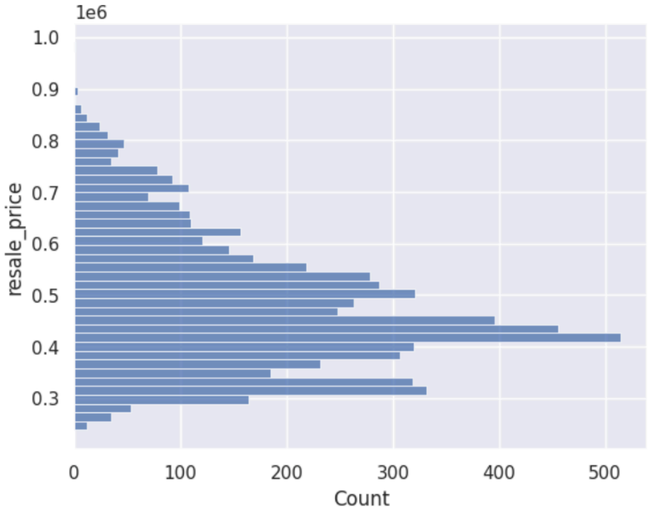
- Basically to know how to use these libraries
- Plot dataset, aids in coming up with **hypothesis**



Histogram



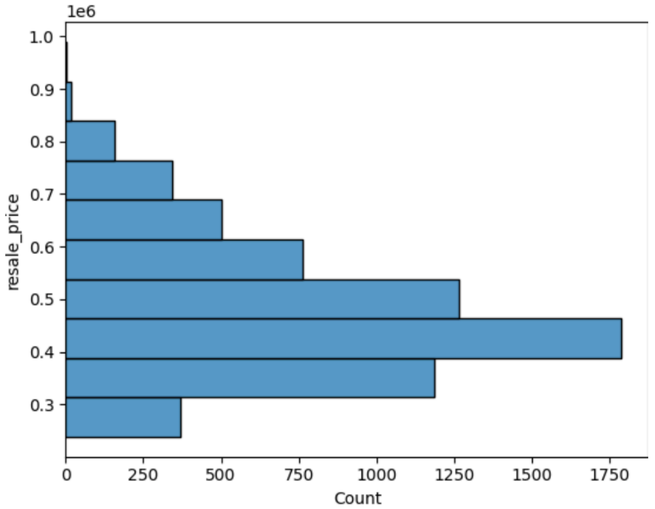

```
sns.histplot(x='resale_price', data=df_tampines)
```




```
sns.set()
sns.histplot(y='resale_price', data=df_tampines)
```

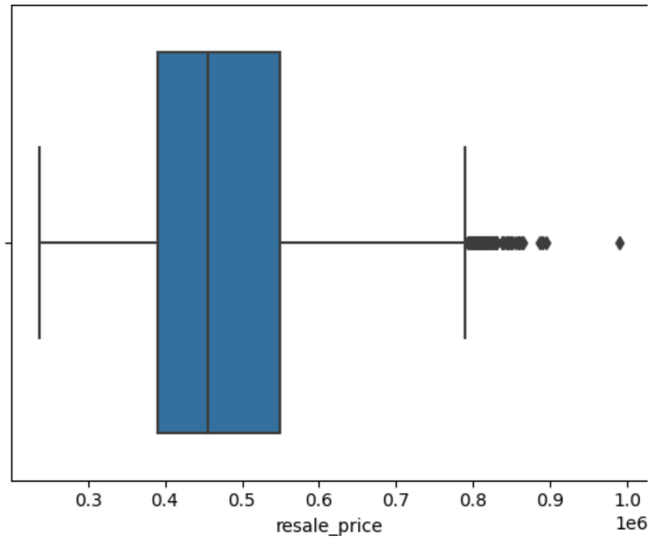
Histogram

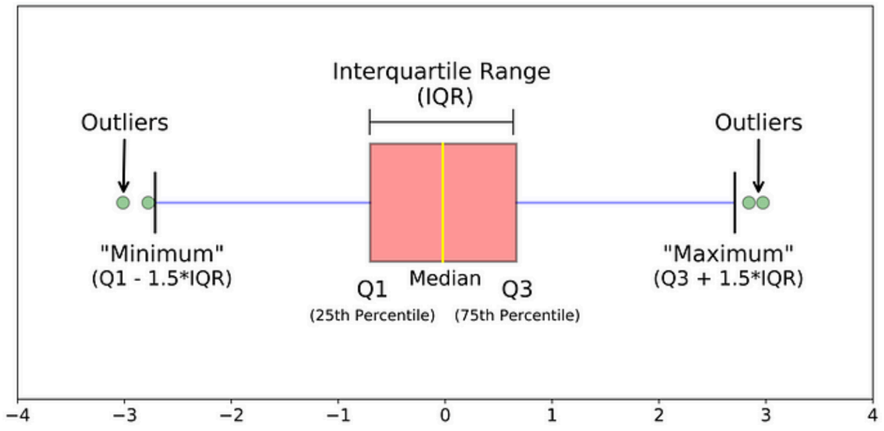
```
sns.histplot(y='resale_price', data=df_tampines, bins=10)
```



Boxplot

```
sns.boxplot(x='resale_price', data=df_tampines)
```



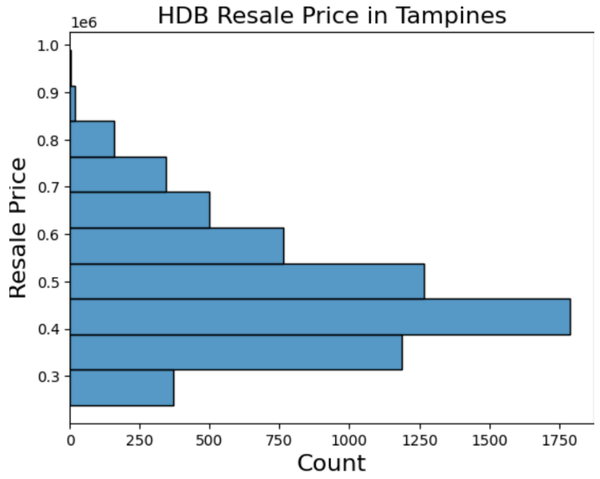


$$min = Q1 - 1.5 \times IQR$$

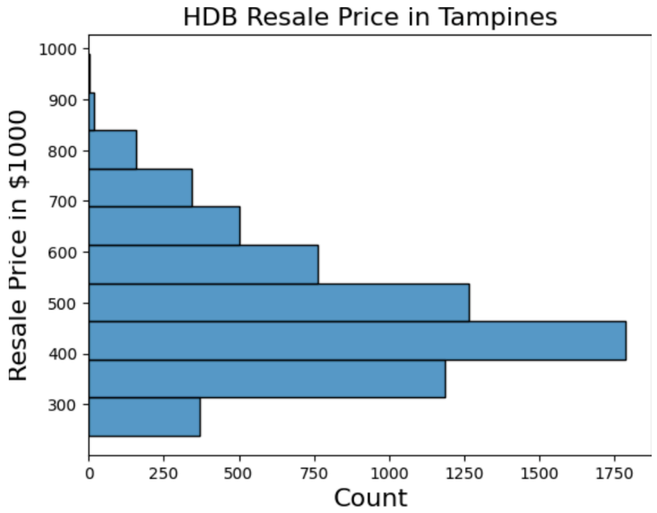
$$max = Q3 + 1.5 \times IQR$$

Setting labels & titles

```
myplot = sns.histplot(y='resale_price', data=df_tampines, bins=10)
myplot.set_xlabel('Count', fontsize=16)
myplot.set_ylabel('Resale Price', fontsize=16)
myplot.set_title('HDB Resale Price in Tampines', fontsize=16)
```

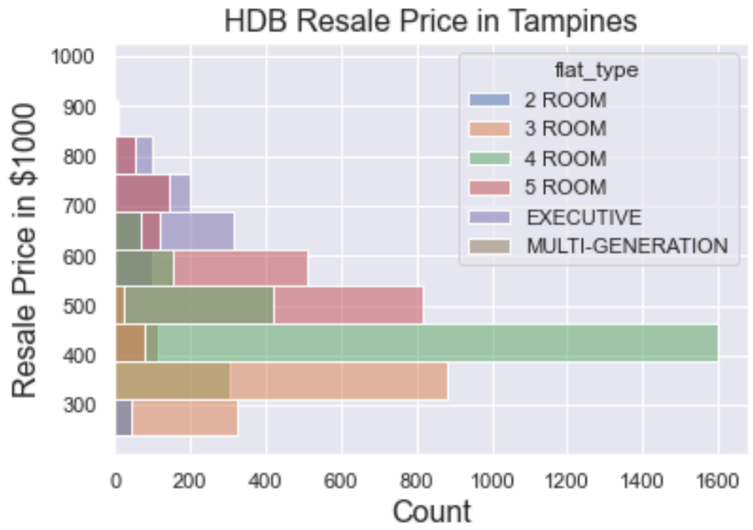



```
myplot = sns.histplot(y='resale_price_1000', data=df_tampines, bins=10)
myplot.set_xlabel('Count', fontsize=16)
myplot.set_ylabel('Resale Price in $1000', fontsize=16)
myplot.set_title('HDB Resale Price in Tampines', fontsize=16)
```



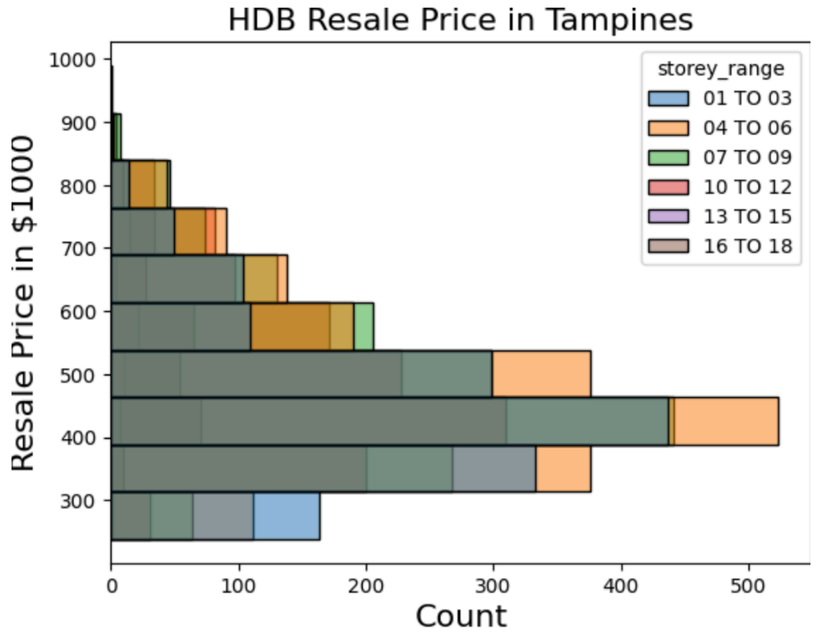
Using Hue

```
myplot = sns.histplot(y='resale_price_1000', hue='flat_type', data=df_tampines, bins=10)
myplot.set_xlabel('Count', fontsize=16)
myplot.set_ylabel('Resale Price in $1000', fontsize=16)
myplot.set_title('HDB Resale Price in Tampines', fontsize=16)
```



Bad UI

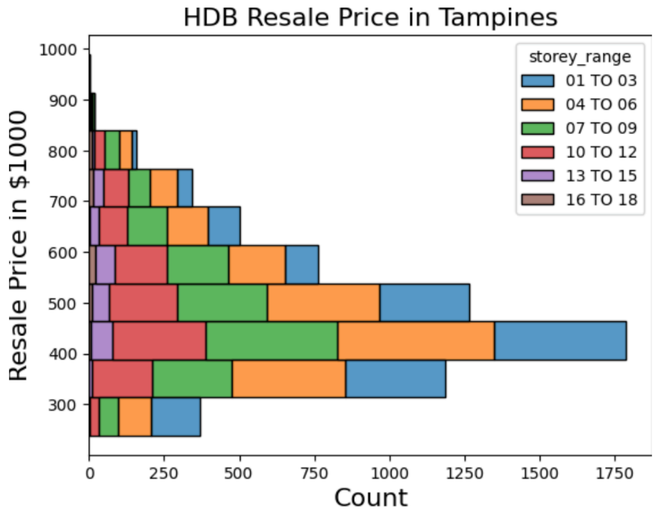
```
myplot = sns.histplot(y='resale_price_1000', hue='storey_range', data=df_tampines, bins=10)
myplot.set_xlabel('Count', fontsize=16)
myplot.set_ylabel('Resale Price in $1000', fontsize=16)
myplot.set_title('HDB Resale Price in Tampines', fontsize=16)
```



Good UI

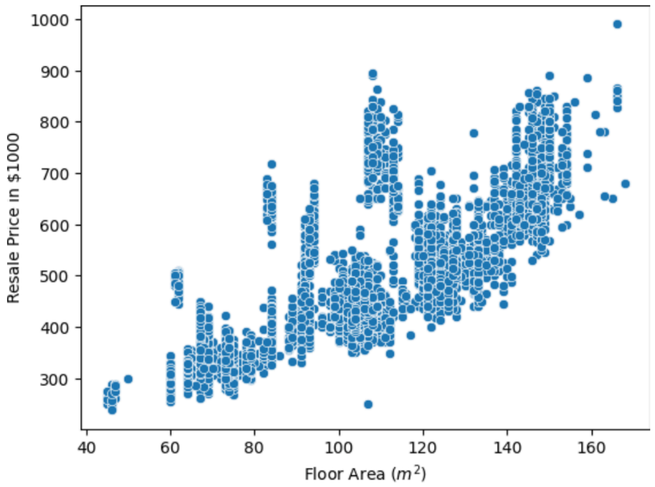
Stack the bars

```
myplot = sns.histplot(y='resale_price_1000', hue='storey_range',
                      multiple='stack',
                      data=df_tampines, bins=10)
myplot.set_xlabel('Count', fontsize=16)
myplot.set_ylabel('Resale Price in $1000', fontsize=16)
myplot.set_title('HDB Resale Price in Tampines', fontsize=16)
```



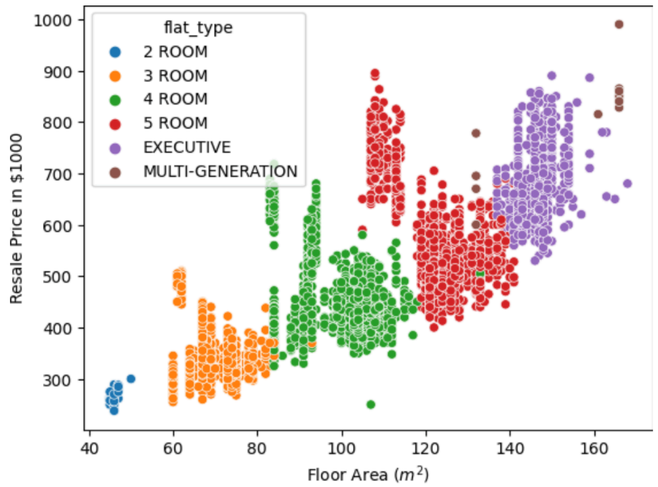
Scatter Plot

```
myplot = sns.scatterplot(x='floor_area_sqm', y='resale_price_1000', data=df_tampines)
myplot.set_xlabel('Floor Area ($m^2$)')
myplot.set_ylabel('Resale Price in $1000')
```



Scatter Plot

```
myplot = sns.scatterplot(x='floor_area_sqm', y='resale_price_1000',
                         hue='flat_type',
                         data=df_tampines)
myplot.set_xlabel('Floor Area ($m^2$)')
myplot.set_ylabel('Resale Price in $1000')
```

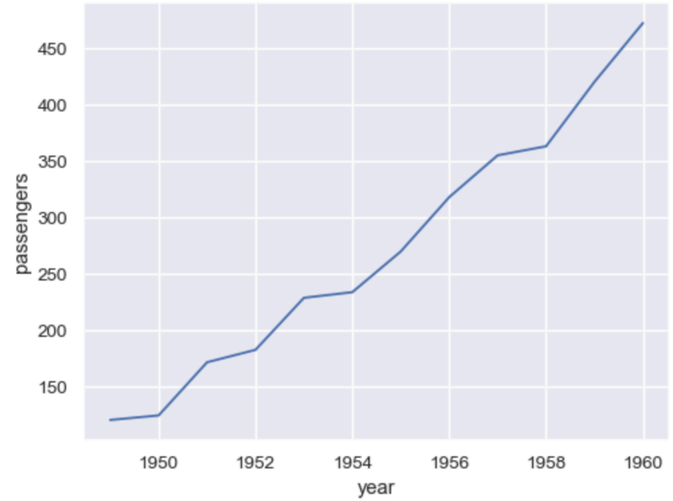


Line Plot

	year	month	passengers
0	1949	Jan	112
1	1949	Feb	118
2	1949	Mar	132
3	1949	Apr	129
4	1949	May	121

To draw a line plot using long-form data, assign the `x` and `y` variables:

```
may_flights = flights.query("month == 'May'")
sns.lineplot(data=may_flights, x="year", y="passengers")
```

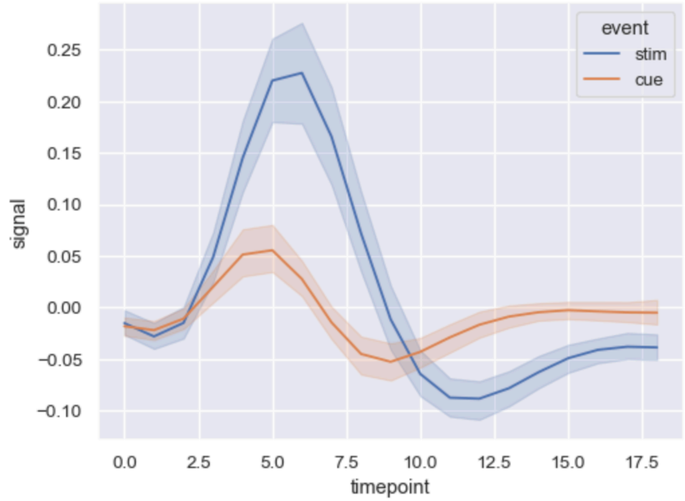



```
fmri = sns.load_dataset("fmri")
fmri.head()
```

	subject	timepoint	event	region	signal
0	s13	18	stim	parietal	-0.017552
1	s5	14	stim	parietal	-0.080883
2	s12	18	stim	parietal	-0.081033
3	s11	18	stim	parietal	-0.046134
4	s10	18	stim	parietal	-0.037970

Repeated observations are aggregated even when semantic grouping is used:

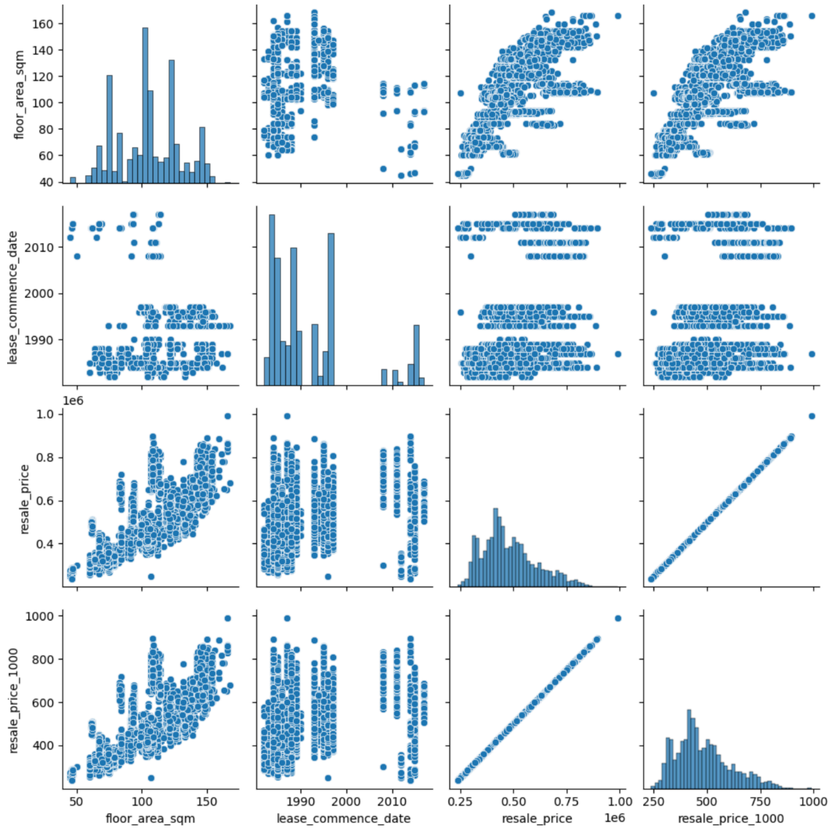
```
sns.lineplot(data=fmri, x="timepoint", y="signal", hue="event")
```



Pair Plot

Plots the relationship on multiple data columns.

```
myplot = sns.pairplot(data=df_tampines)
```



Learning Objectives

- Give **example** application of linear regression and classification
- Create a **Pandas** DataFrame and selecting data from DataFrame
- Using **Pandas** library to **read** CSV file
- **Split** data randomly into training set and testing set
- **Normalize** data using z-normalization and min-max normalization
- **Convert** Pandas Dataframe to NumPy Array
- **Selecting** data from NumPy Array
- Use **mathematical**, statistical and linear algebra functions on NumPy Array
- Creating **new** Numpy Arrays
- Create **scatter** plot and other **statistical** plots like box plot, histogram, and bar plot