Python初級數據分析員證書

（六）數據分析 及可視化專案

16. 數據分析專案
Machine Learning – Part 3

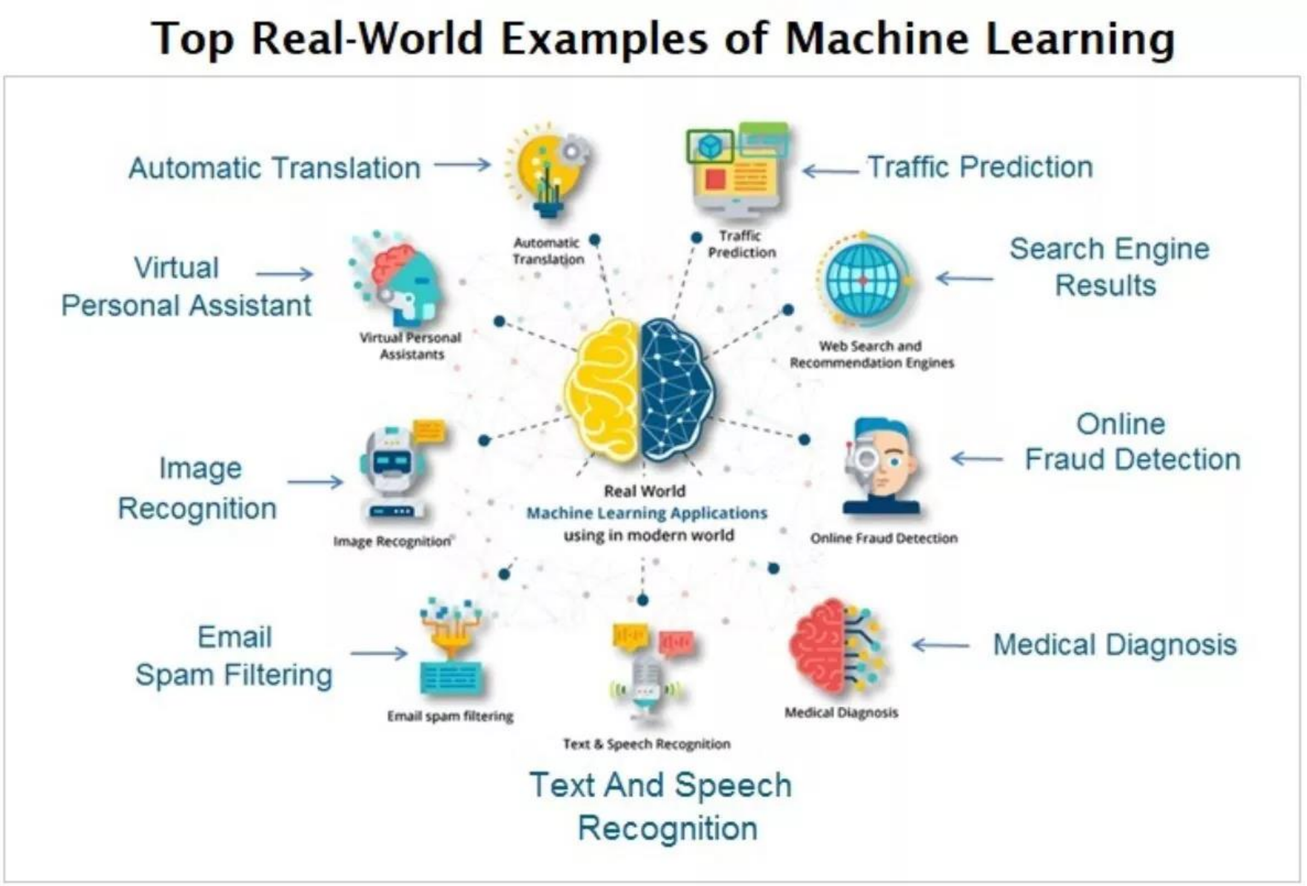
Review

- Statistics
- Hypothesis testing
- Algebra
- Linear regression
- Propositional logic
- Python
- R
- SQL

- Pandas, NumPy, SciPy
- Data Visualization, Matplotlib, Seaborn, Plotly
- Dashboard Visualization, Business Intelligence
- Storytelling
- Machine Learning

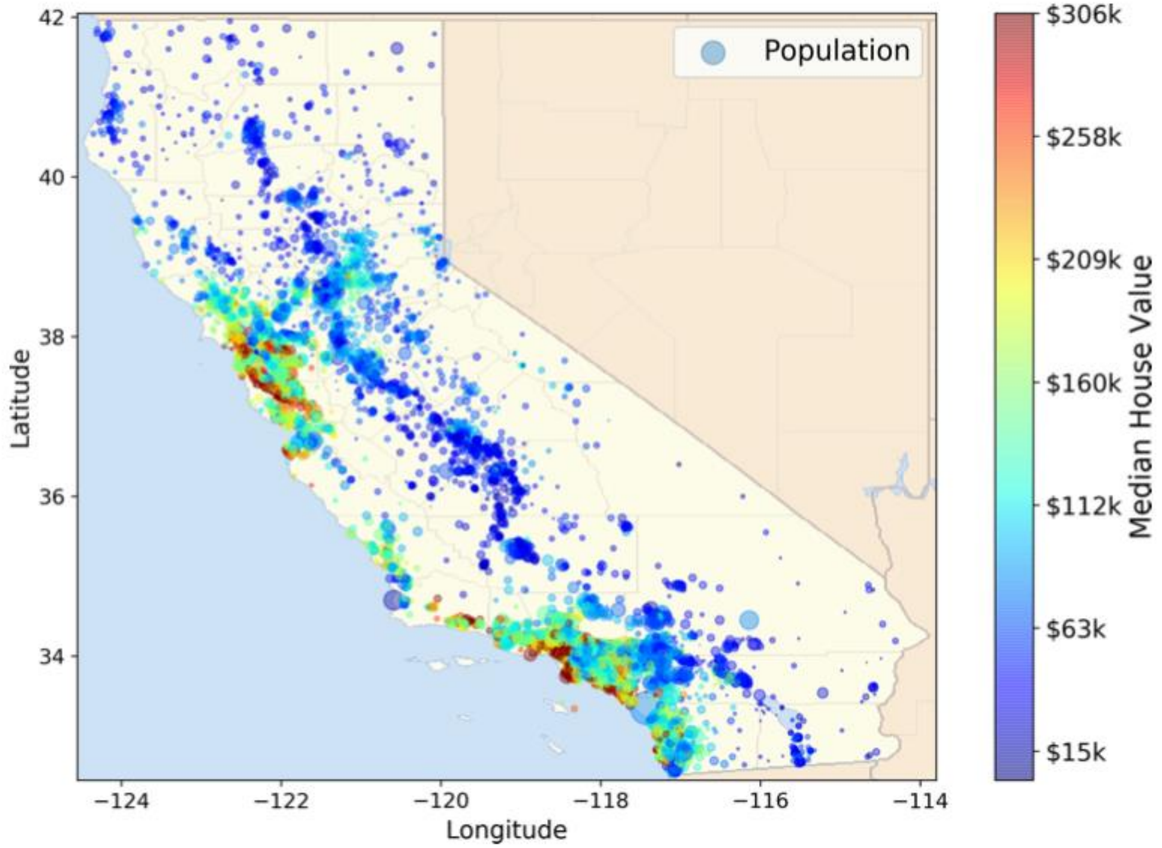
Review

- **Application**
- **Types of Machine Learning**
  - Supervised learning
  - Unsupervised learning
  - Semi-supervised learning
  - Reinforcement learning
- **Example: Iris Species Classification**
- **Generalization**
- **Underfitting & Overfitting**
- **KNN, LinearRegressor, RandomForest**



Case Study

In this chapter, we will use a dataset of California Housing Price. This data has metrics such as the population, median income, median housing price, and so on for each block group in California. Block groups are the smallest geographical unit for which the US Census Bureau.



Data import

```
1  import numpy as np
2  import pandas as pd
3  import missingno as msno
4  from scipy import stats
5  import matplotlib.pyplot as plt
6  import seaborn as sns
7  import plotly.express as px
8
9  from sklearn.impute import SimpleImputer
10 from sklearn.linear_model import LinearRegression
11 from sklearn.tree import DecisionTreeRegressor
12 from sklearn.ensemble import RandomForestRegressor
13 from sklearn.metrics import mean_squared_error, r2_score
14 from sklearn.model_selection import train_test_split, cross_val_score, GridSearchCV
15 from sklearn.preprocessing import OrdinalEncoder, OneHotEncoder, StandardScaler
16 from sklearn.pipeline import Pipeline
17 from sklearn.compose import ColumnTransformer
18 from sklearn.base import BaseEstimator, TransformerMixin
```

```
1  housing = pd.read_csv('housing.csv')
2  housing
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	ocean_proximity
0	-122.23	37.88	41.0	880.0	129.0	322.0	126.0	8.3252	452600.0	NEAR BAY
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0	1138.0	8.3014	358500.0	NEAR BAY

Data Overview

```
1 housing.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
 #   Column              Non-Null Count  Dtype  
---  ------              --------------  -----  
 0   longitude           20640 non-null  float64
 1   latitude            20640 non-null  float64
 2   housing_median_age  20640 non-null  float64
 3   total_rooms         20640 non-null  float64
 4   total_bedrooms      20433 non-null  float64
 5   population          20640 non-null  float64
 6   households          20640 non-null  float64
 7   median_income       20640 non-null  float64
 8   median_house_value  20640 non-null  float64
 9   ocean_proximity     20640 non-null  object 
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

```
1 # Categoric column
2 housing["ocean_proximity"].value_counts()
```

```
<1H OCEAN     9136
INLAND        6551
NEAR OCEAN    2658
NEAR BAY      2290
ISLAND           5
Name: ocean_proximity, dtype: int64
```

Data Overview

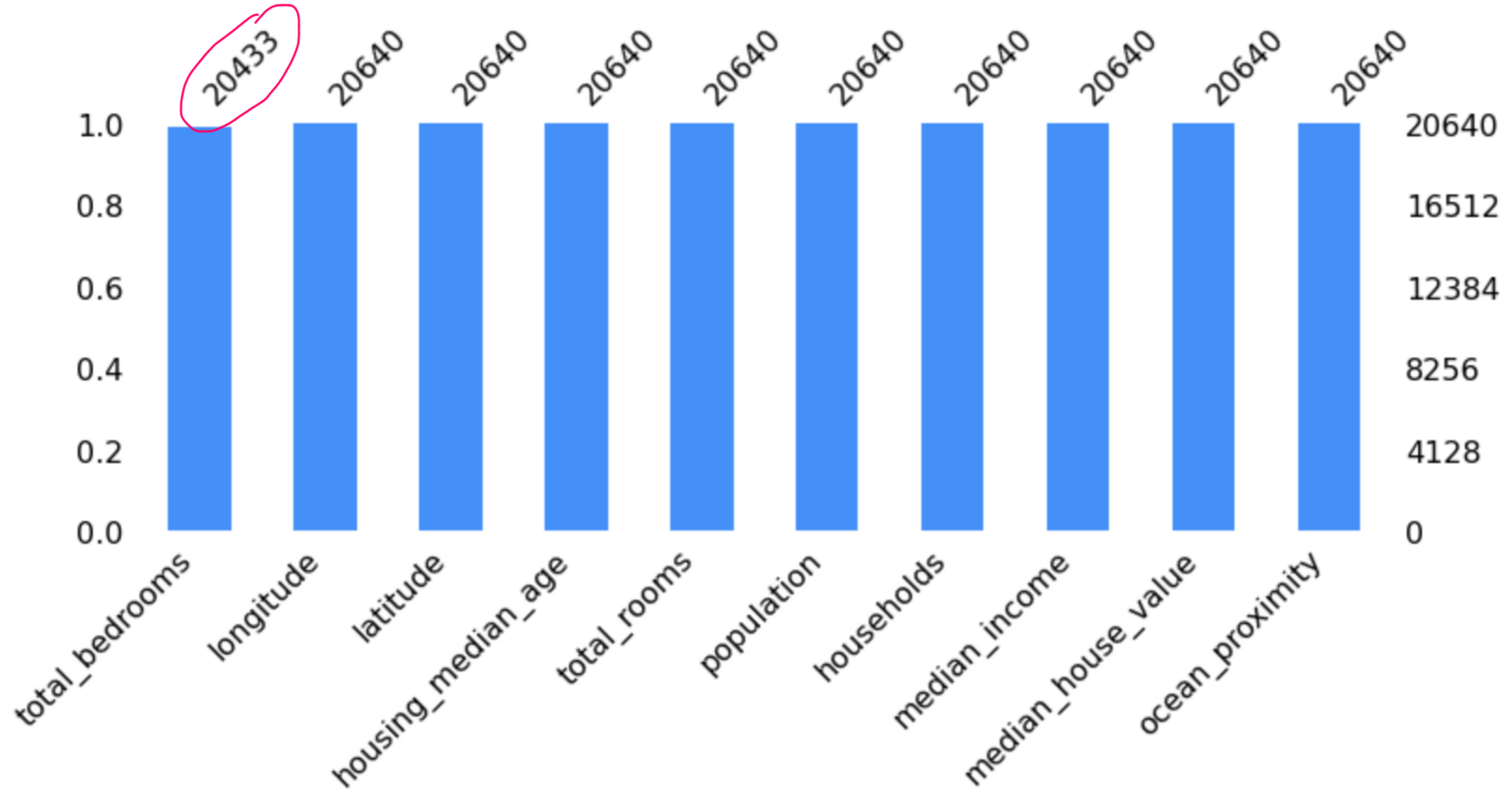
```
1 housing.describe(include='all').style.format().background_gradient(cmap='Blues')
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	oce
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000	20640.000000	20640.000000	20640.000000	20640.000000	
unique	nan	nan	nan	nan	nan	nan	nan	nan	nan	
top	nan	nan	nan	nan	nan	nan	nan	nan	nan	
freq	nan	nan	nan	nan	nan	nan	nan	nan	nan	
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553	1425.476744	499.539680	3.870671	206855.816909	
std	2.003532	2.135952	12.585558	2181.615252	421.385070	1132.462122	382.329753	1.899822	115395.615874	
min	-124.350000	32.540000	1.000000	2.000000	1.000000	3.000000	1.000000	0.499900	14999.000000	
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000	787.000000	280.000000	2.563400	119600.000000	
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000	1166.000000	409.000000	3.534800	179700.000000	
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000	1725.000000	605.000000	4.743250	264725.000000	
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000	35682.000000	6082.000000	15.000100	500001.000000	

Check Null and NaN

```
1 msno.bar(housing, figsize=(12,4), color="dodgerblue", sort="ascending")
```

```
<AxesSubplot:>
```



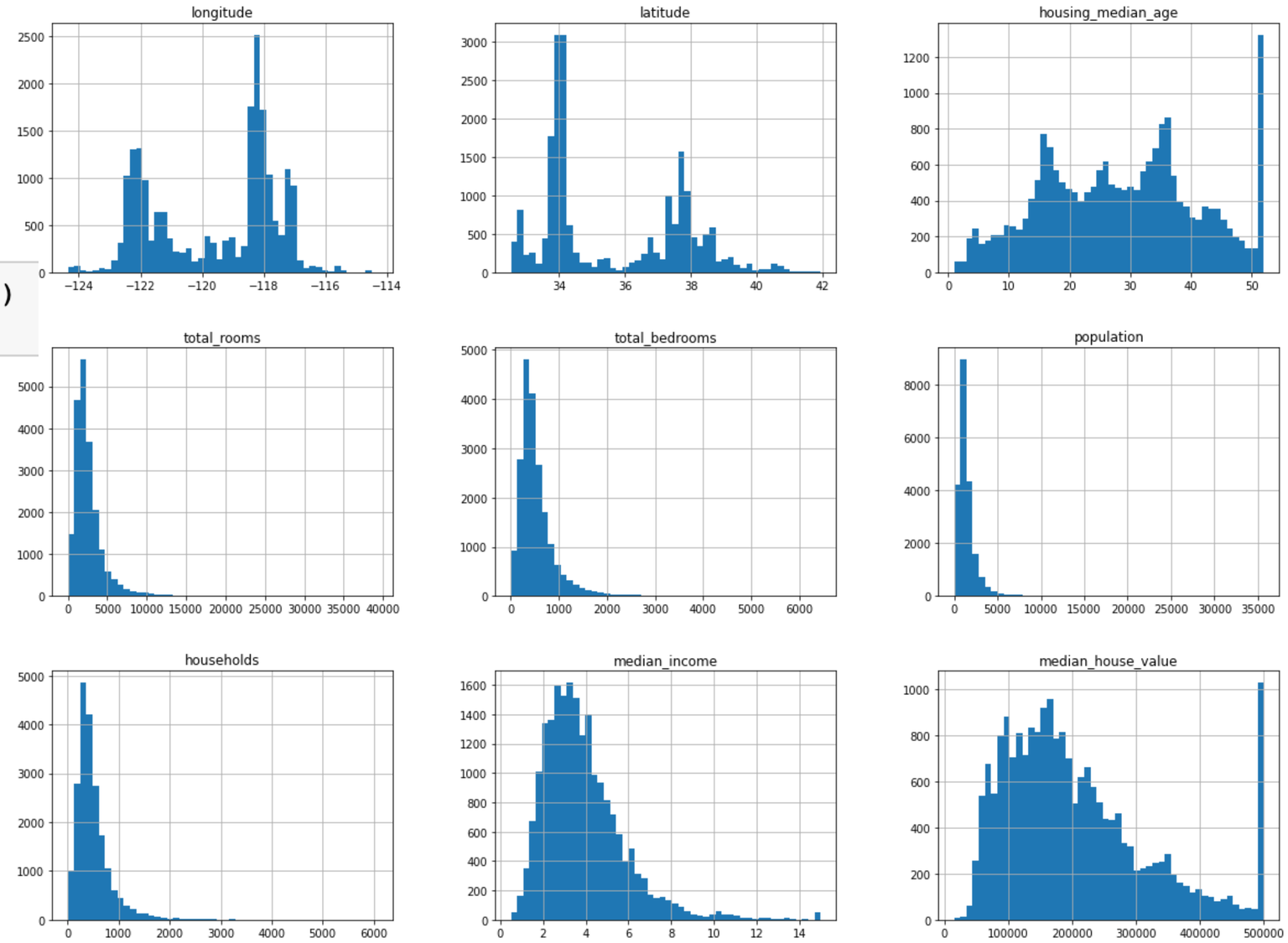
Missing data in total_bedrooms column

```
1  # check NaN column and numbers
2  housing.isnull().sum()
```

```
longitude                 0
latitude                  0
housing_median_age        0
total_rooms               0
total_bedrooms          207
population                0
households                0
median_income             0
median_house_value        0
ocean_proximity           0
dtype: int64
```

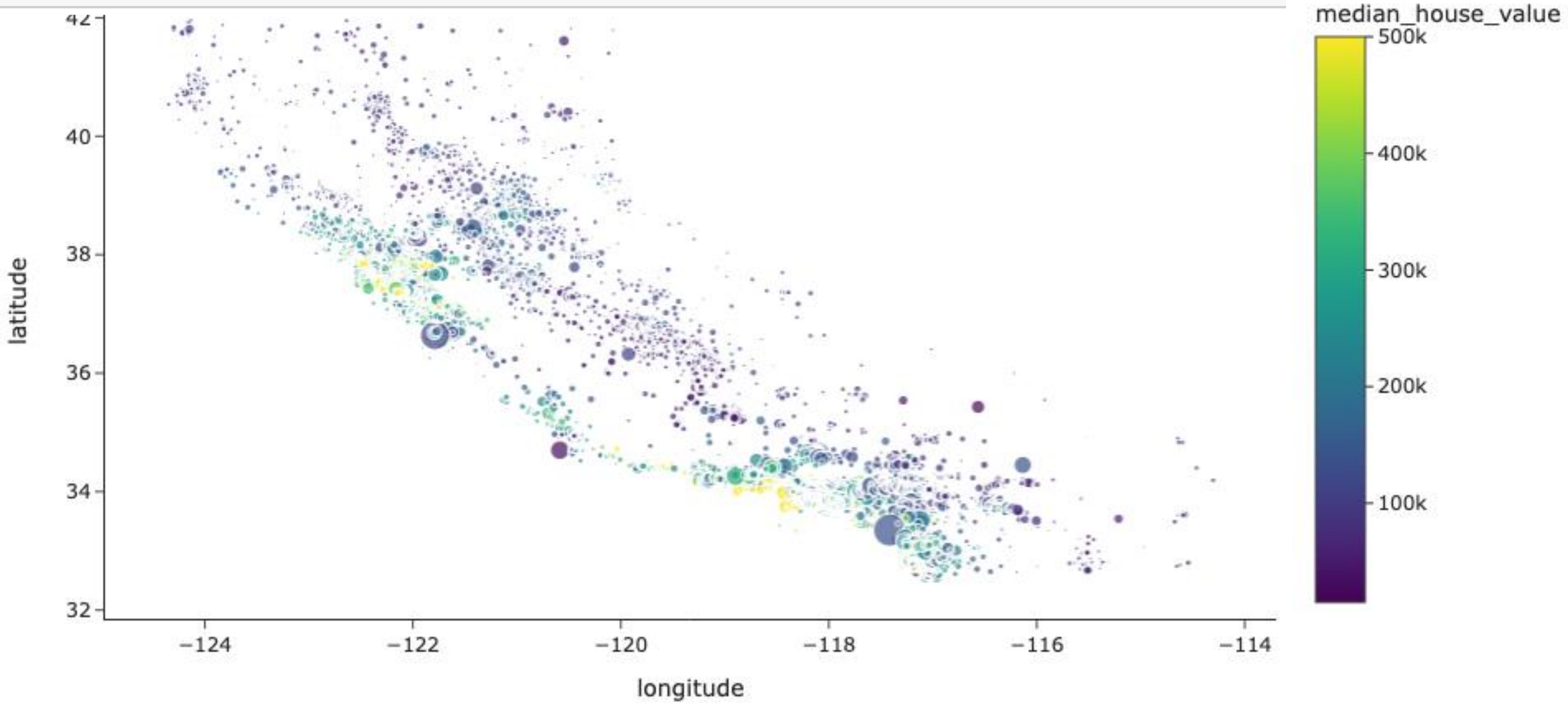
Histogram on each numeric columns

```
housing.hist(bins=50, figsize=(20,15))
plt.show()
```



Scatter plot

```
fig = px.scatter(housing, x="longitude", y="latitude",
                 color="median_house_value", size='population',
             hover_data=['ocean_proximity'], template="simple_white")
fig.show()
```



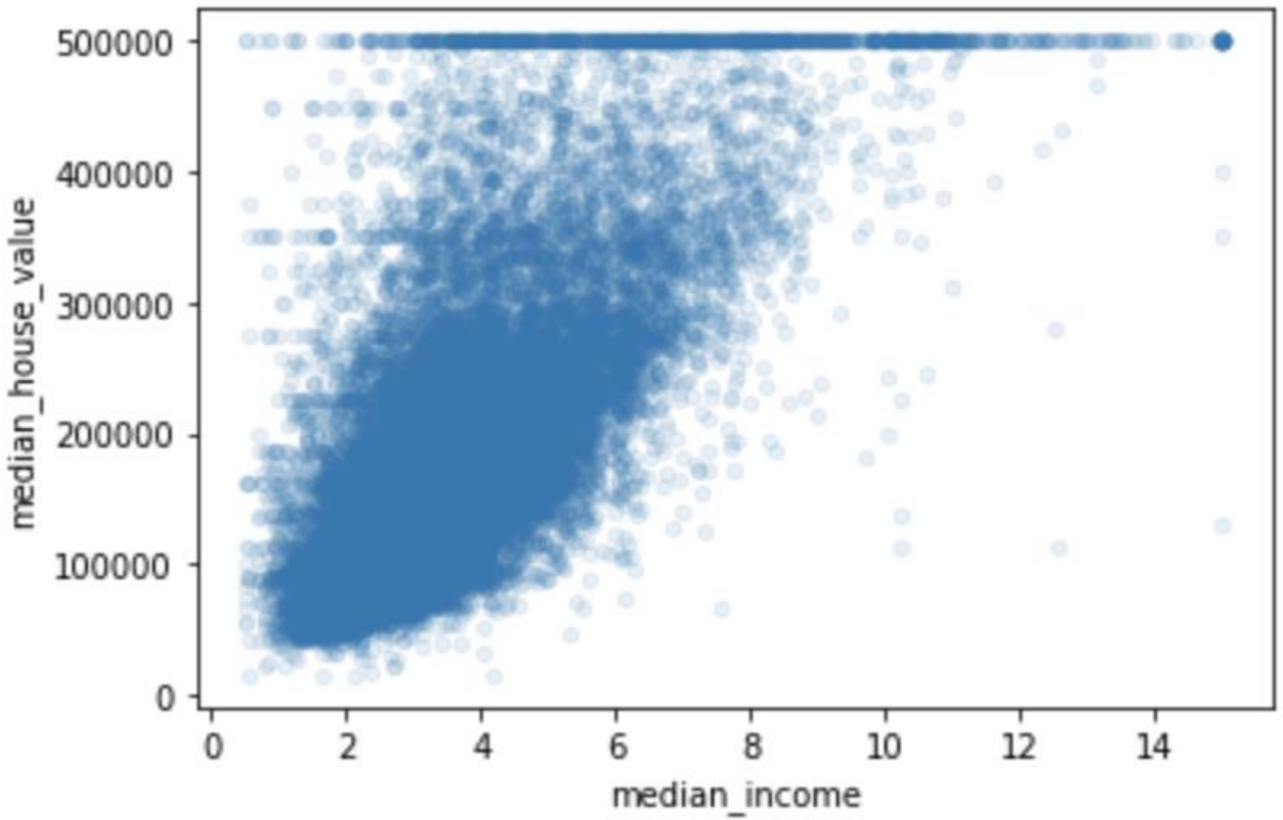
Find out possible relation of house value

```
1 corr_matrix = housing.select_dtypes(include=np.number).corr()
2 corr_matrix["median_house_value"].sort_values(ascending=False)
```

```
median_house_value    1.000000
median_income         0.688075
total_rooms           0.134153
housing_median_age    0.105623
households            0.065843
total_bedrooms        0.049686
population           -0.024650
longitude            -0.045967
latitude             -0.144160
Name: median_house_value, dtype: float64
```

```
1  housing.plot(kind="scatter", x="median_income", y="median_house_value",
2               alpha=0.1)
```

```
<AxesSubplot:xlabel='median_income', ylabel='median_house_value'>
```



Data wrangling

Since the total_rooms, total_bedrooms, and population are not a good indicator to the single house evaluation, we divide them on relevant portion.

```
# Add new features
housing["rooms_per_household"] = housing["total_rooms"]/housing["households"]
housing["bedrooms_per_room"] = housing["total_bedrooms"]/housing["total_rooms"]
housing["population_per_household"] = housing["population"]/housing["households"]
```

Correlation

```
1 corr_matrix = housing.select_dtypes(include=np.number).corr()
2 corr_matrix["median_house_value"].sort_values(ascending=False)
```

```
median_house_value          1.000000
median_income               0.688075
rooms_per_household         0.151948
total_rooms                 0.134153
housing_median_age          0.105623
households                  0.065843
total_bedrooms              0.049686
population_per_household   -0.023737
population                 -0.024650
longitude                  -0.045967
latitude                   -0.144160
bedrooms_per_room          -0.255880
Name: median_house_value, dtype: float64
```

Data cleaning

Scikit-Learn provides a handy class to take care of missing values: SimpleImputer.

Replace missing values using a descriptive statistic (e.g. mean, median, or most frequent) along each column, or using a constant value.

```
1 imputer = SimpleImputer(strategy="median")
```

Since the median can only be computed on numerical attributes, we need to create a copy of the data without the text attribute ocean_proximity:

```
1 housing_num = housing.drop("ocean_proximity", axis=1)
```

```
1 imputer.fit(housing_num)
```

```
▼ SimpleImputer
SimpleImputer(strategy='median')
```

Replace NaN with median

The imputer has simply computed the median of each attribute and stored the result in its statistics_ instance variable

```
1  imputer.statistics_
```

```
array([-1.18490000e+02,  3.42600000e+01,  2.90000000e+01,  2.12700000e+03,
        4.35000000e+02,  1.16600000e+03,  4.09000000e+02,  3.53480000e+00,
        1.79700000e+05,  5.22912879e+00,  2.03162434e-01,  2.81811565e+00])
```

```
1  # Same as these
2  housing_num.median().values
```

```
array([-1.18490000e+02,  3.42600000e+01,  2.90000000e+01,  2.12700000e+03,
        4.35000000e+02,  1.16600000e+03,  4.09000000e+02,  3.53480000e+00,
        1.79700000e+05,  5.22912879e+00,  2.03162434e-01,  2.81811565e+00])
```

```
1  housing_tr = pd.DataFrame(imputer.transform(housing_num), columns=housing_num.columns)
```

Handling Text and Categorical Attributes

preprocess the categorical input feature, ocean_proximity:

```
housing_cat = housing[["ocean_proximity"]]
ordinal_encoder = OrdinalEncoder()
housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
housing_cat_encoded
```

```
array([[3.],
       [3.],
       [3.],
       ...,
       [1.],
       [1.],
       [1.]])
```

Handling Text and Categorical Attributes

Most of the machine learning tools prefer numeric data. Therefore we change the categoric one to numbers.

```
1 ordinal_encoder.categories_
```

```
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
       dtype=object)]
```

```
1 housing_cat_encoded
```

```
array([[3.],
       [3.],
       [3.],
       ...,
       [1.],
       [1.],
       [1.]])
```

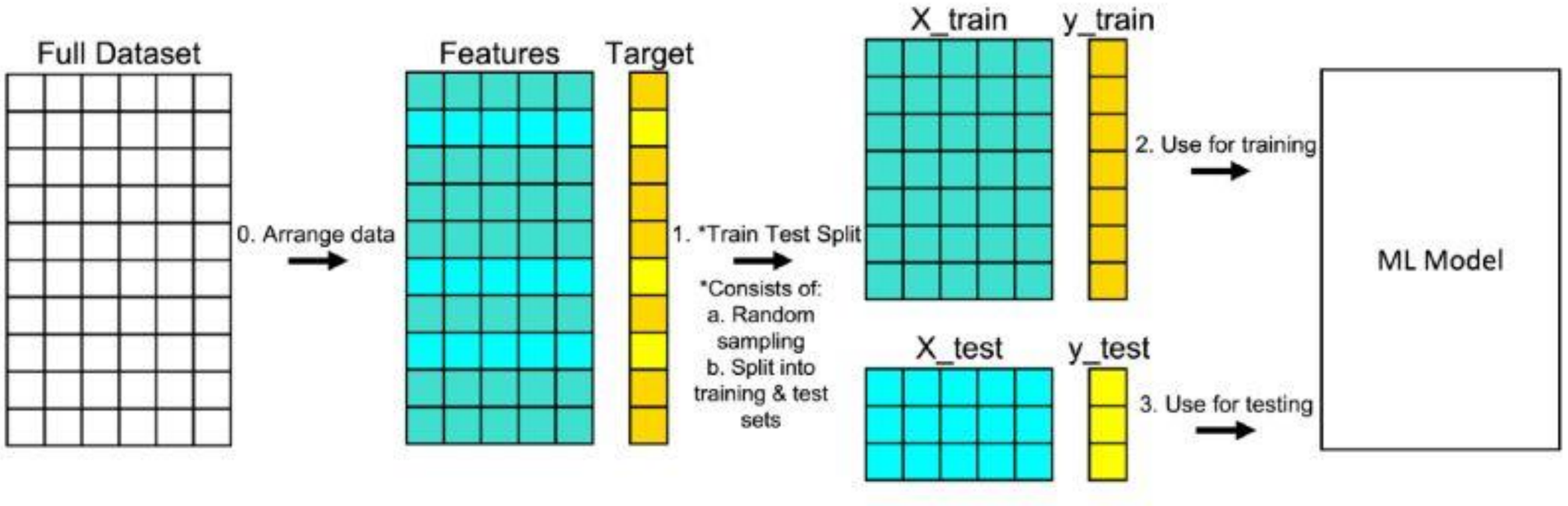
Handling Text and Categorical Attributes

```
housing_tr['housing_cat']=housing_cat_encoded
housing_tr
```

total_bedrooms	population	households	median_income	median_house_value	rooms_per_household	bedrooms_per_room	population_per_household	housing_cat
129.0	322.0	126.0	8.3252	452600.0	6.984127	0.146591	2.555556	3.0
1106.0	2401.0	1138.0	8.3014	358500.0	6.238137	0.155797	2.109842	3.0
190.0	496.0	177.0	7.2574	352100.0	8.288136	0.129516	2.802260	3.0
235.0	558.0	219.0	5.6431	341300.0	5.817352	0.184458	2.547945	3.0
280.0	565.0	259.0	3.8462	342200.0	6.281853	0.172096	2.181467	3.0
...	...	...	...	...	...	...	...	...
374.0	845.0	330.0	1.5603	78100.0	5.045455	0.224625	2.560606	1.0
150.0	356.0	114.0	2.5568	77100.0	6.114035	0.215208	3.122807	1.0
485.0	1007.0	433.0	1.7000	92300.0	5.205543	0.215173	2.325635	1.0
409.0	741.0	349.0	1.8672	84700.0	5.329513	0.219892	2.123209	1.0
616.0	1387.0	530.0	2.3886	89400.0	5.254717	0.221185	2.616981	1.0

Prepare the Data for Machine Learning Algorithms

```
1  y = housing_tr['median_house_value'].copy()
2  X = housing_tr.drop('median_house_value', axis=1)
3
4  X_train, X_test, y_train, y_test = train_test_split(
5                                                  X, y, test_size=0.2, random_state=42)
```



Testing and Validating

Split your data into two sets: the *training set* and the *test set*. As these names imply, you train your model using the training set, and you test it using the test set.

The error rate on new cases is called the generalization error (or out-of-sample error), and by evaluating your model on the test set, you get an estimate of this error.

LinearRegression model

```
1  lin_reg = LinearRegression()
2  lin_reg.fit(X_train, y_train)
```

▼ LinearRegression

LinearRegression()

```
1  y_pred = lin_reg.predict(X_test)
2  lin_mse = mean_squared_error(y_test, y_pred)
3  lin_rmse = np.sqrt(lin_mse)
4  print('MSE: ', lin_mse)
5  print('RMSE: ', lin_rmse)
```

```
MSE:  4885570662.763147
RMSE:  69896.85731678604
```

```
1  r2_score(y_test, y_pred)
```

```
0.627172070408006 2
```

The LinearRegression model is not so effective in this dataset.

Selecting Performance Measure

A typical performance measure for regression problems is the Root Mean Square Error (**RMSE**). It gives an idea of how much error the system typically makes in its predictions, with a higher weight for large errors.

$$\text{RMSE}(\mathbf{X}, h) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(h\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right)^2}$$

Selecting Performance Measure

In some contexts you may prefer to use another function. For example, suppose that there are many outlier districts. In that case, you may consider using the *Mean Absolute Error* .

$$\text{MAE}(\mathbf{X}, h) = \frac{1}{m}\sum_{i=1}^{m}\left|h\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right|$$

RandomForestRegressor model

```
%%time
from sklearn.ensemble import RandomForestRegressor

forest_reg = RandomForestRegressor()
forest_reg.fit(X_train, y_train)

y_pred = forest_reg.predict(X_test)
lin_mse = mean_squared_error(y_test, y_pred)
lin_rmse = np.sqrt(lin_mse)

print('MSE: ', lin_mse)
print('RMSE: ', lin_rmse)
print('R2 Score: ', r2_score(y_test, y_pred))
```

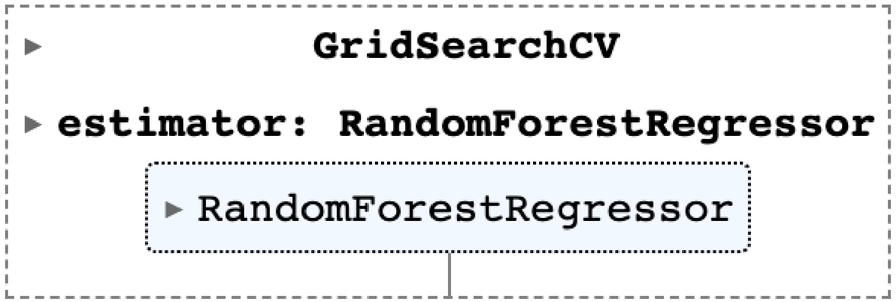
```
MSE:  2519785162.6918917
RMSE:  50197.461715627534
R2 Score:  0.807710019960756
CPU times: user 11.2 s, sys: 144 ms, total: 11.3 s
Wall time: 11.3 s
```

RandomForestRegressor looks much better. Let’s try to fine tune it.

Fine tune model by Grid Search

```
%%time
from sklearn.model_selection import GridSearchCV
param_grid = [
        {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
        {'bootstrap': [False], 'n_estimators': [3, 10],
                                        'max_features': [2, 3, 4]},
        ]
forest_reg = RandomForestRegressor()
grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
                           scoring='neg_mean_squared_error',
                           return_train_score=True)
grid_search.fit(X_train, y_train)
```

```
CPU times: user 47.4 s, sys: 714 ms, total: 48.2 s
Wall time: 48.6 s
```



Fine tune model by Grid Search

```
1 grid_search.best_params_
```

```
{'max_features': 4, 'n_estimators': 30}
```

```
1 grid_search.best_estimator_
```

```
RandomForestRegressor(max_features=4, n_estimators=30)
```

Fine tune model by Grid Search

You may keep changing the model parameter until you find a satisfied result.

Alternatively, you should get Scikit-Learn's **GridSearchCV** to search for you.

All you need to do is tell it which hyperparameters you want it to experiment with, and what values to try out, and it will evaluate all the possible combinations of hyperparameter values, using cross-validation.

Fine tune model by Grid Search

```
cvres = grid_search.cv_results_
for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]): print(np.sqrt(-mean_score), params)
```

```
63453.59616283231 {'max_features': 2, 'n_estimators': 3}
54516.48301387678 {'max_features': 2, 'n_estimators': 10}
52051.46938681462 {'max_features': 2, 'n_estimators': 30}
58589.066939071105 {'max_features': 4, 'n_estimators': 3}
51735.79258319917 {'max_features': 4, 'n_estimators': 10}
49607.14617702199 {'max_features': 4, 'n_estimators': 30}
59054.63861015314 {'max_features': 6, 'n_estimators': 3}
52342.87773841201 {'max_features': 6, 'n_estimators': 10}
50346.961599956856 {'max_features': 6, 'n_estimators': 30}
58973.83239625586 {'max_features': 8, 'n_estimators': 3}
53104.256944208675 {'max_features': 8, 'n_estimators': 10}
50627.53664560542 {'max_features': 8, 'n_estimators': 30}
60881.3218805105 {'bootstrap': False, 'max_features': 2, 'n_estimators': 3}
53207.9300298594 {'bootstrap': False, 'max_features': 2, 'n_estimators': 10}
58706.241142890496 {'bootstrap': False, 'max_features': 3, 'n_estimators': 3}
51620.22949018809 {'bootstrap': False, 'max_features': 3, 'n_estimators': 10}
58626.178107233034 {'bootstrap': False, 'max_features': 4, 'n_estimators': 3}
50822.375171282096 {'bootstrap': False, 'max_features': 4, 'n_estimators': 10}
```

Fine tune the model

For example, the following code searches for the best combination of hyperparameter values for the RandomForestRegressor:

```
from sklearn.model_selection import GridSearchCV

param_grid = [
    {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
    {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},
  ]

forest_reg = RandomForestRegressor()

grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
                           scoring='neg_mean_squared_error',
                           return_train_score=True)

grid_search.fit(housing_prepared, housing_labels)
```

Fine tune the model - GridSearchCV

Get the best estimator as follow:

```
>>> grid_search.best_estimator_
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
           max_features=8, max_leaf_nodes=None, min_impurity_decrease=0.0,
           min_impurity_split=None, min_samples_leaf=1,
           min_samples_split=2, min_weight_fraction_leaf=0.0,
           n_estimators=30, n_jobs=None, oob_score=False, random_state=None,
           verbose=0, warm_start=False)
```

Analyse the Best Models and Their Errors

```
>>> feature_importances = grid_search.best_estimator_.feature_importances_
>>> feature_importances
array([7.33442355e-02, 6.29090705e-02, 4.11437985e-02, 1.46726854e-02,
       1.41064835e-02, 1.48742809e-02, 1.42575993e-02, 3.66158981e-01,
       5.64191792e-02, 1.08792957e-01, 5.33510773e-02, 1.03114883e-02,
       1.64780994e-01, 6.02803867e-05, 1.96041560e-03, 2.85647464e-03])

>>> sorted(zip(feature_importances, attributes), reverse=True)
[(0.3661589806181342, 'median_income'),
 (0.1647809935615905, 'INLAND'),
 (0.10879295677551573, 'pop_per_hhold'),
 (0.07334423551601242, 'longitude'),
 (0.062909070482620 3, 'latitude'),
```

Final testing

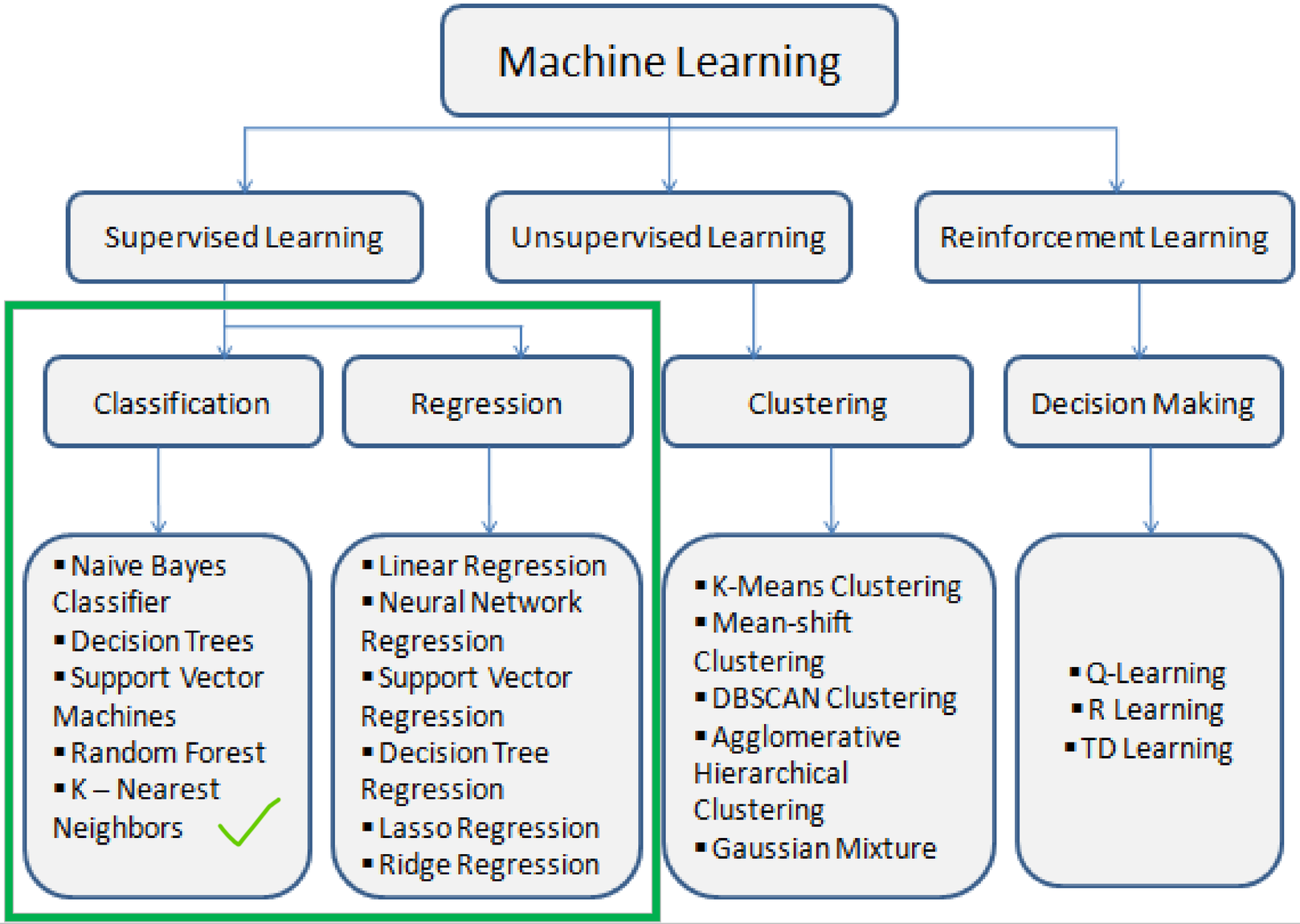
Now we had an improved model for housing valuation.

```
final_model = grid_search.best_estimator_
final_predictions = final_model.predict(X_test)
final_mse = mean_squared_error(y_test, final_predictions)
final_rmse = np.sqrt(final_mse)

print('Final MSE: ', final_mse)
print('Final RMSE: ', final_rmse)
print('Final R2 Score: ', r2_score(y_test, final_predictions))
```

```
Final MSE:  2443915014.7243557
Final RMSE:  49435.96883570054
Final R2 Score:  0.8134998267483575
```

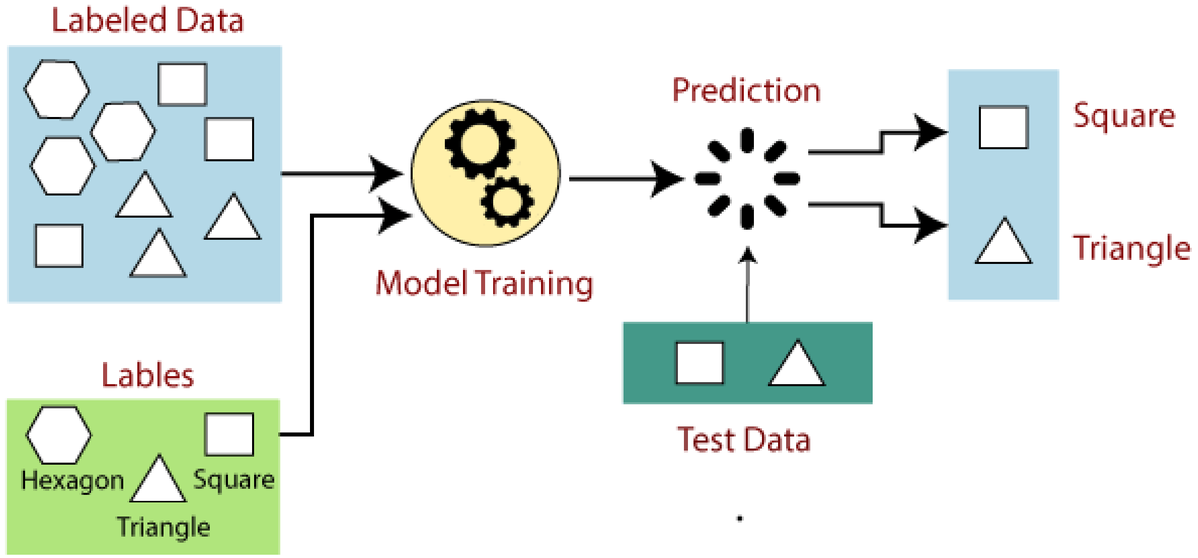
Summarise Machine Learning



Supervised Learning - Classification

In our last chapter, we learnt to use KNN to classify Iris species.

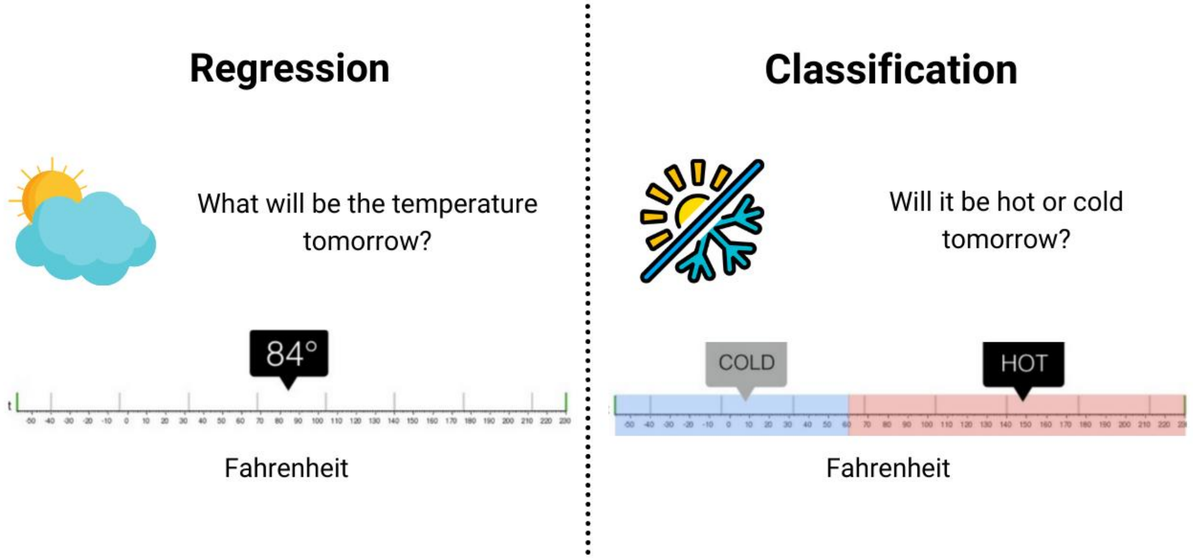
Apart from KNN, there are more ML tools to build the classification model, such as Logistic function, Naïve Bayes, Decision Tree, and Support Vector Machine.



Supervised Learning - Regression

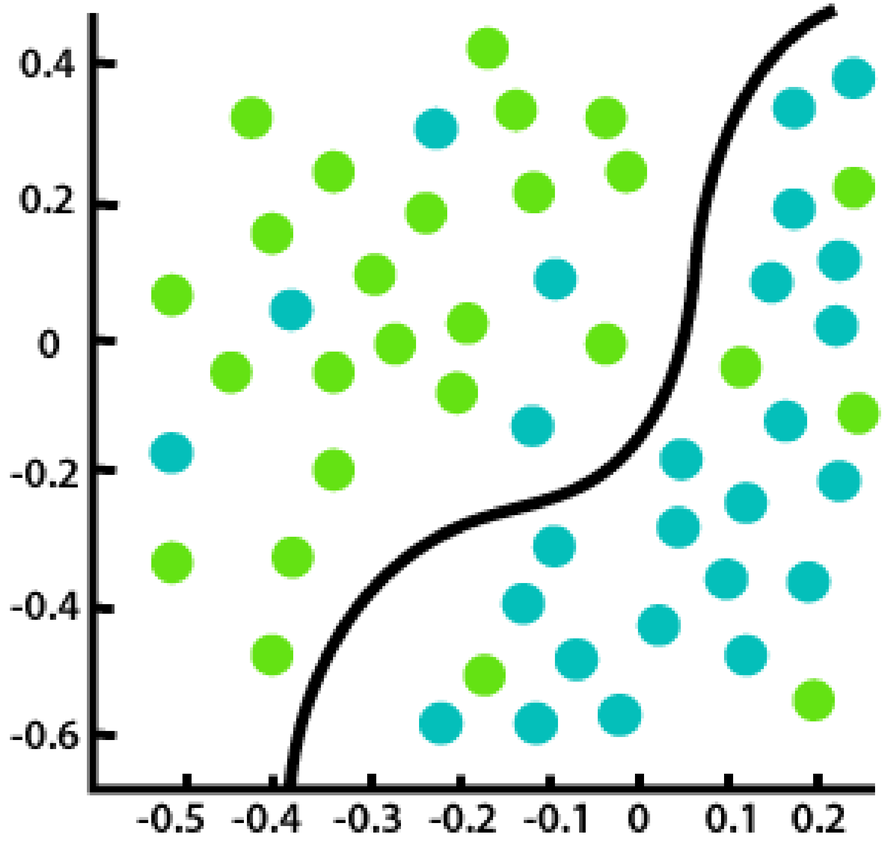
With Supervised Learning Regression, we can predict and forecast continuous or real number. Regression algorithm can be divided into: Linear and Non Linear Regression.

In Regression model, we can find the best fit line to output or predict more accurately in number. Regression model can solve the problem like weather, house price, and employee salary.

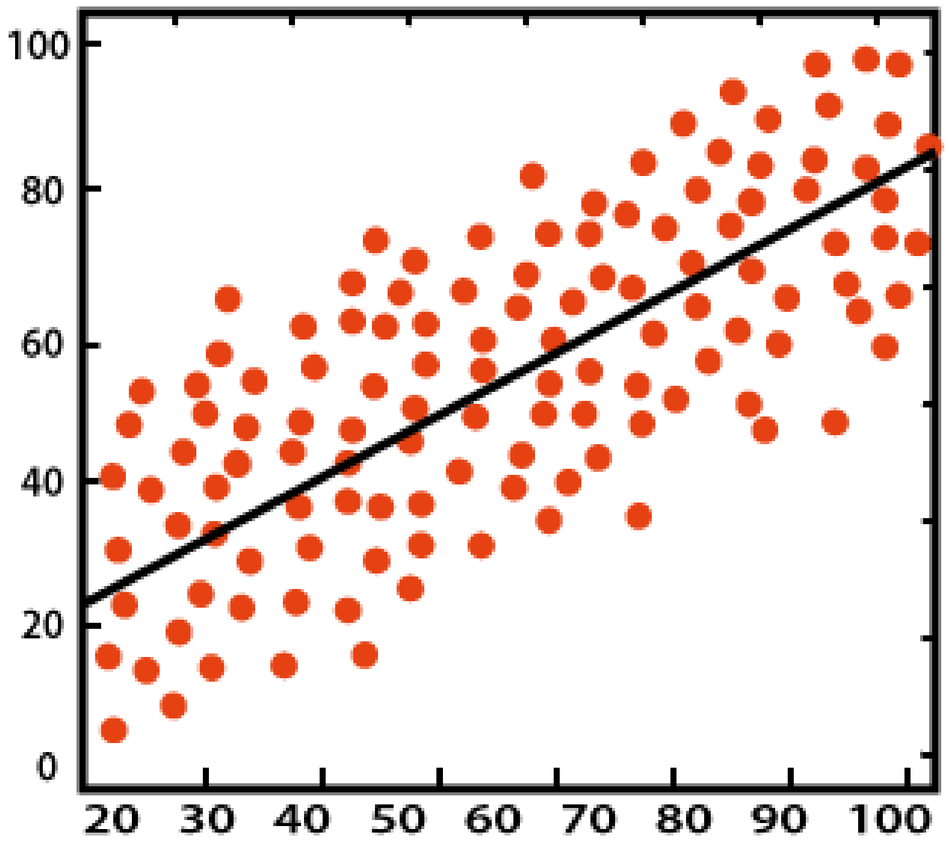


Supervised Learning – Types of Regression

- **Linear Regression**
- **Logistic Regression**
- **Polynomial Regression**
- **Support Vector Regression**
- **Decision Tree Regression**
- **Random Forest Regression**
- **Ridge Regression**
- **Lasso Regression**



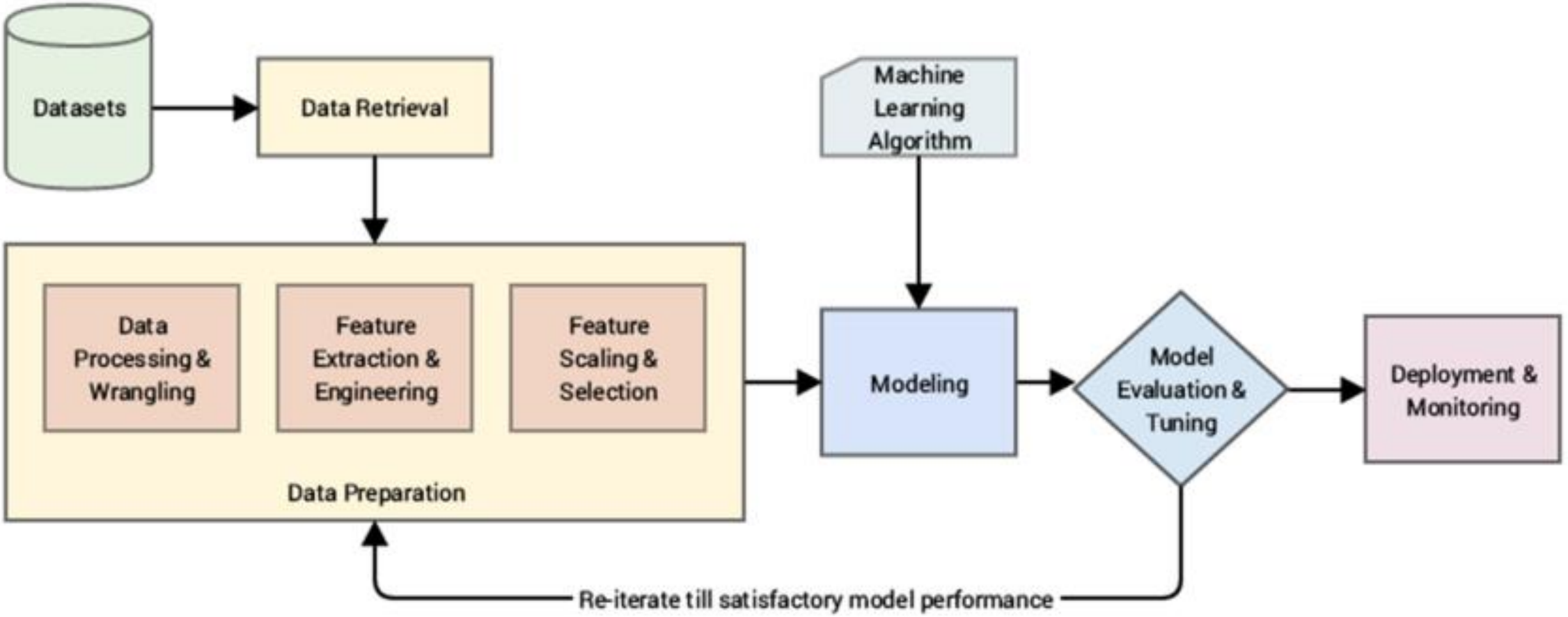
Classification



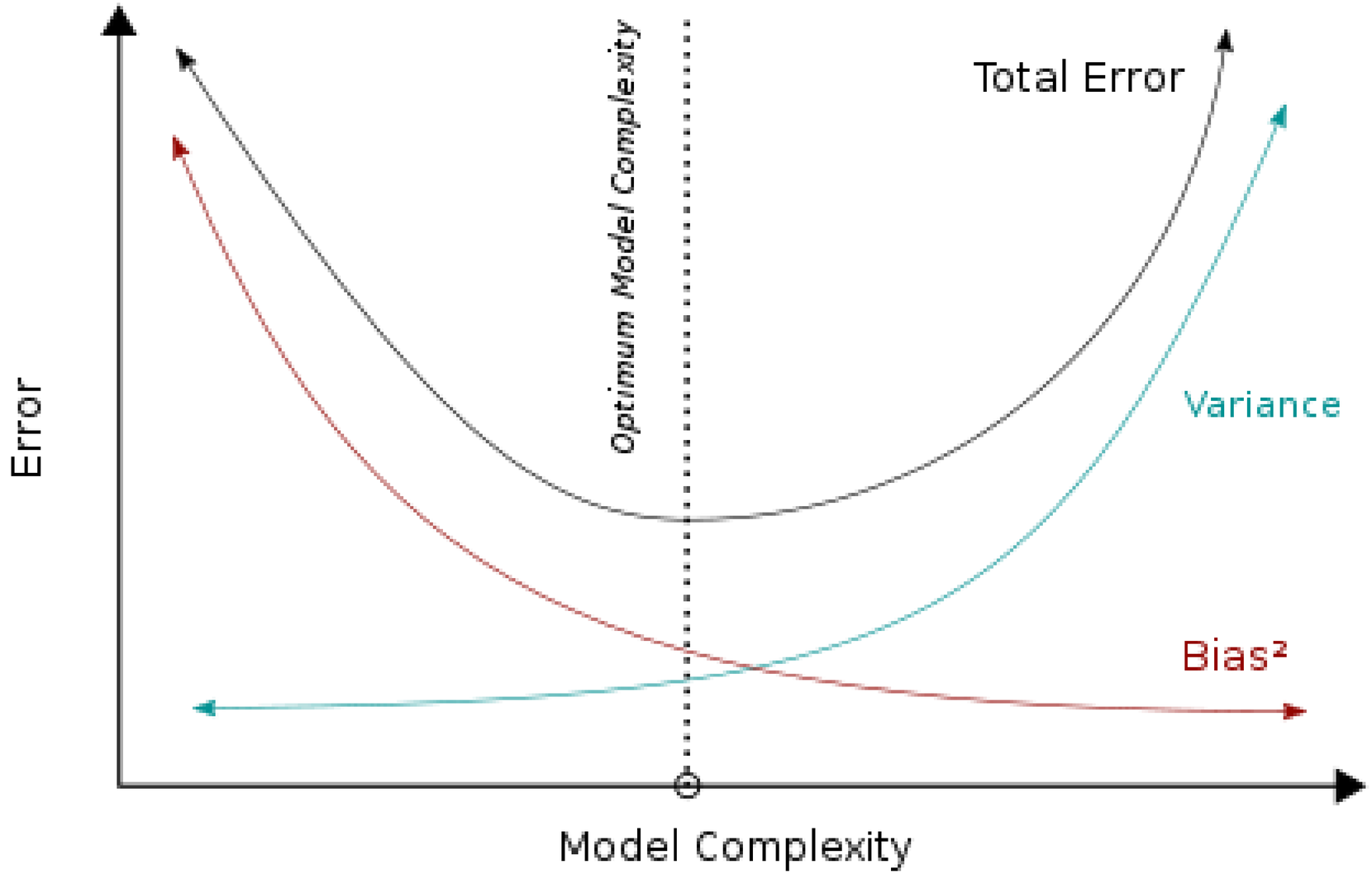
Regression

Machine Learning Workflow

In general the ML process works like as follow. Of course in actual scenario, it could be much more complicated.

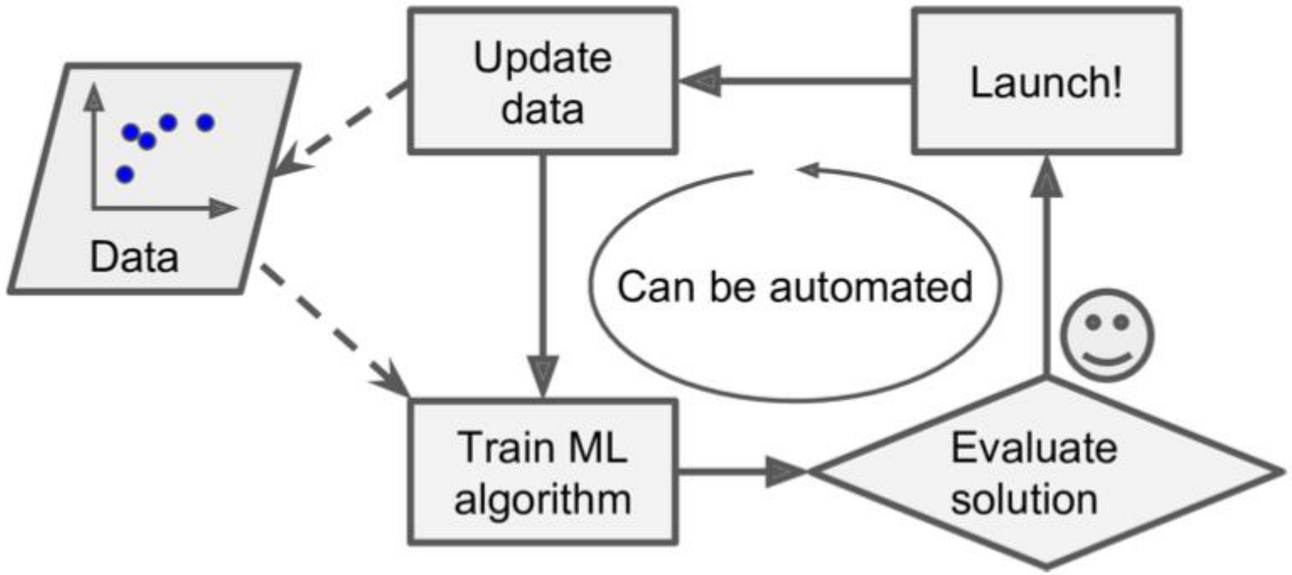


Overfitting & Underfitting



Chapter Wrap Up

Machine Learning is the science (and art) of programming computers so they can ***learn from data***. Machine learning is a subfield of artificial intelligence that involves the development of algorithms and models that can learn patterns from data and make predictions or decisions based on those patterns.



Reference & Resources

Scikit Learn:

https://scikit-learn.org/

IPython Cookbook, 2nd edition

https://ipython-books.github.io/

Plotly Graph Objects:

https://plotly.com/python/graph-objects/

Seaborn:

https://seaborn.pydata.org/examples/index.html

Matplotlib:

https://matplotlib.org/