U O W

End-to-End Projects in Big Data

CSCI316: Big Data Mining Techniques and Implementation

UNIVERSITY
OF WOLLONGONG
AUSTRALIA

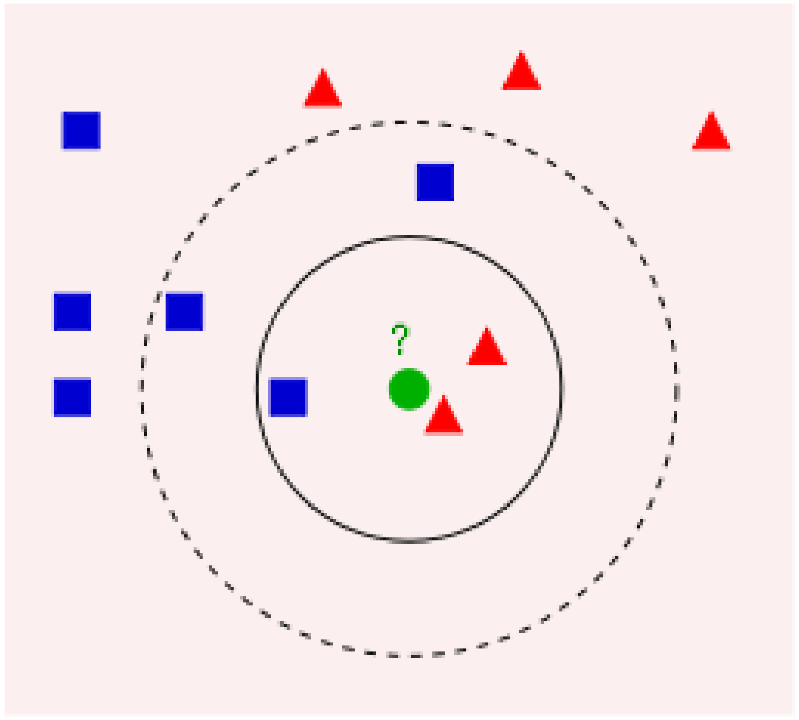
First ML Algorithm: kNN

- kNN: **k-Nearest Neighbours**
  - Predict the label of a record based on its nearest neighbours (assuming we already know the labels of those neighbours)
  - I.e., we have an existing set of example data and know which class each piece of data falls into; we want to know the class of a new piece of data by checking it against the k pieces of most similar data.
- Pseudocode of kNN classifier:

```
For every point in our dataset:
    calculate the distance between inX (input) and the current point
    sort the distances in increasing order
    take k items with lowest distances to inX
    find the majority class among these items
    return the majority class as our prediction for the class of inX
```

- Good as an introductory algorithm, but poor scalability (for large datasets)

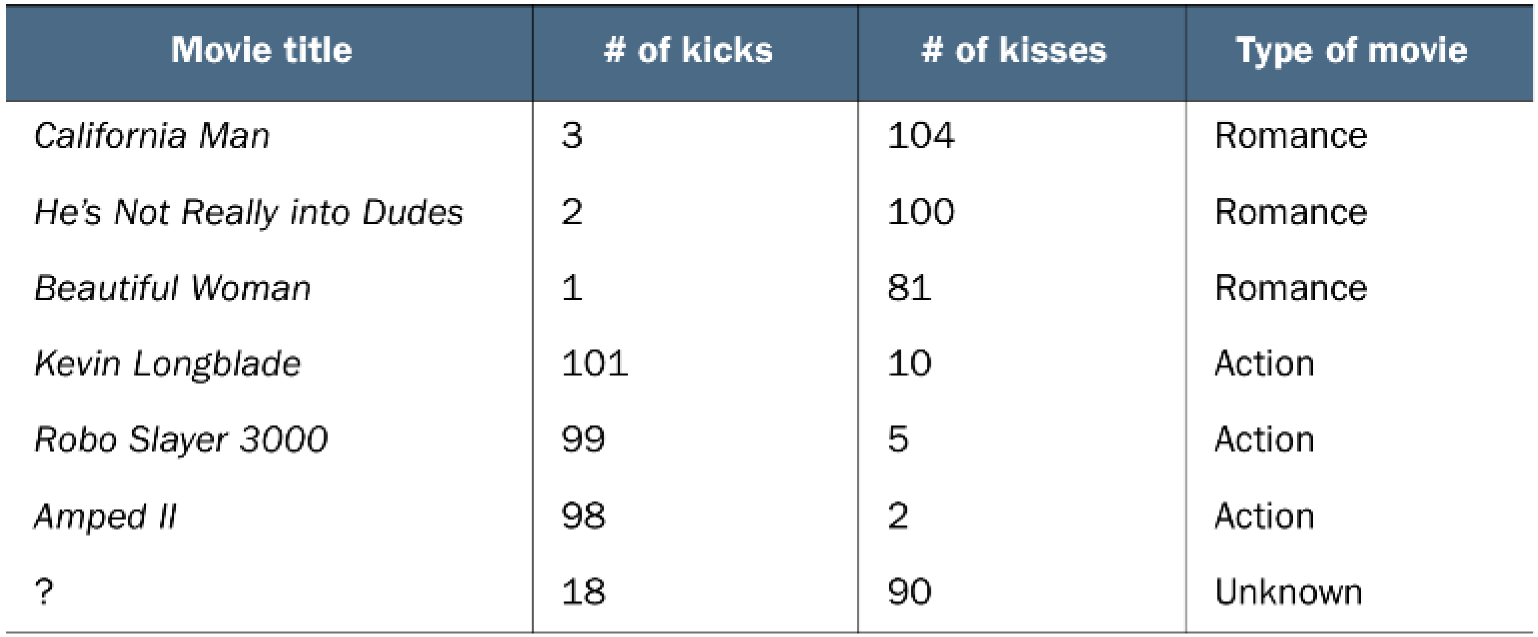
First ML Algorithm: kNN



UNIVERSITY
OF WOLLONGONG
AUSTRALIA

First ML Algorithm: kNN

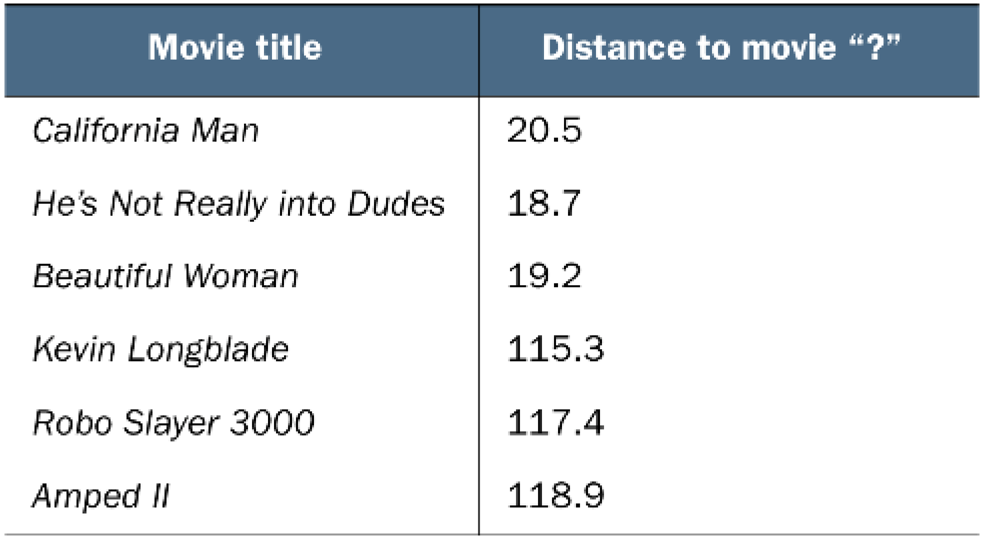
- Feature matrix of movies



Movie title	# of kicks	# of kisses	Type of movie
California Man	3	104	Romance
He's Not Really into Dudes	2	100	Romance
Beautiful Woman	1	81	Romance
Kevin Longblade	101	10	Action
Robo Slayer 3000	99	5	Action
Amped II	98	2	Action
?	18	90	Unknown

First ML Algorithm: kNN

- Distance between each movie and the unknown movie:



Movie title	Distance to movie “?”
California Man	20.5
He’s Not Really into Dudes	18.7
Beautiful Woman	19.2
Kevin Longblade	115.3
Robo Slayer 3000	117.4
Amped II	118.9

- Thus, movie “?” is a romantic film.

First ML Algorithm: kNN

- Euclidean distance for two records:
  - Let $A = (a_1, \dots, a_n)$ and $B = (b_1, \dots, b_n)$:

$$||A - B|| = \sqrt{(a_1 - b_1)^2 + \cdots + (a_n - b_n)^2}$$

  - E.g. the distance between "California Man" and "?" is:

$$||\,(3, 104)\ - (18, 90)|| = 20.5$$

- Some convenient numpy functions

```
# import numpy as np

data = np.array([3, 1, 2])
ast = np.argsort(x)  # ast: array([1, 2, 0])

np.tile(data, (2,1))  # array([[3, 1, 2],
                             # [3, 1, 2]])
np.tile(["a","b"],(2,2)) # array(['a','b','a','b'],
                                # ['a','b','a','b'])
```

First ML Algorithm: kNN

```python
from numpy import *
def classify0(inX, dataSet, labels, k):
    dataSetSize = dataSet.shape[0]
    diffMat = tile(inX, (dataSetSize, 1)) - dataSet
    sqDiffMat = diffMat ** 2    # square
    sqDistances = sqDiffMat.sum(axis=1)
    distances = sqDistances ** 0.5  # square root
    sortedDistIndicies = distances.argsort()
    classCount = {}
    for i in range(k):
        voteIlabel = labels[sortedDistIndicies[i]]
        classCount[voteIlabel] = \
            classCount.get(voteIlabel, 0) + 1
    sortedClassCount = sorted(classCount.items(),
                              key=lambda x: x[1],
                              reverse=True)
    return sortedClassCount[0][0]
```

UNIVERSITY
OF WOLLONGONG
AUSTRALIA

First ML Algorithm: kNN

- Use the kNN classifier:

```
group = array([[1.0, 1.1], [1.0, 1.0],
               [0, 0], [0, 0.1]])
labels = ['A', 'A', 'B', 'B']
# predict the class of a record [0,0] based on
# its 3-nearest neighbours in group
# (whose labels are known).
classify0([0.1, 0], group, labels, 3)  # returns ‘B’
```

- *First ML implementation done!*
  - *If I have any unlabeled record, then I can predict its label with the kNN function.*

An End-to-End Data Mining Project

Python Pandas and Scikit-Learn

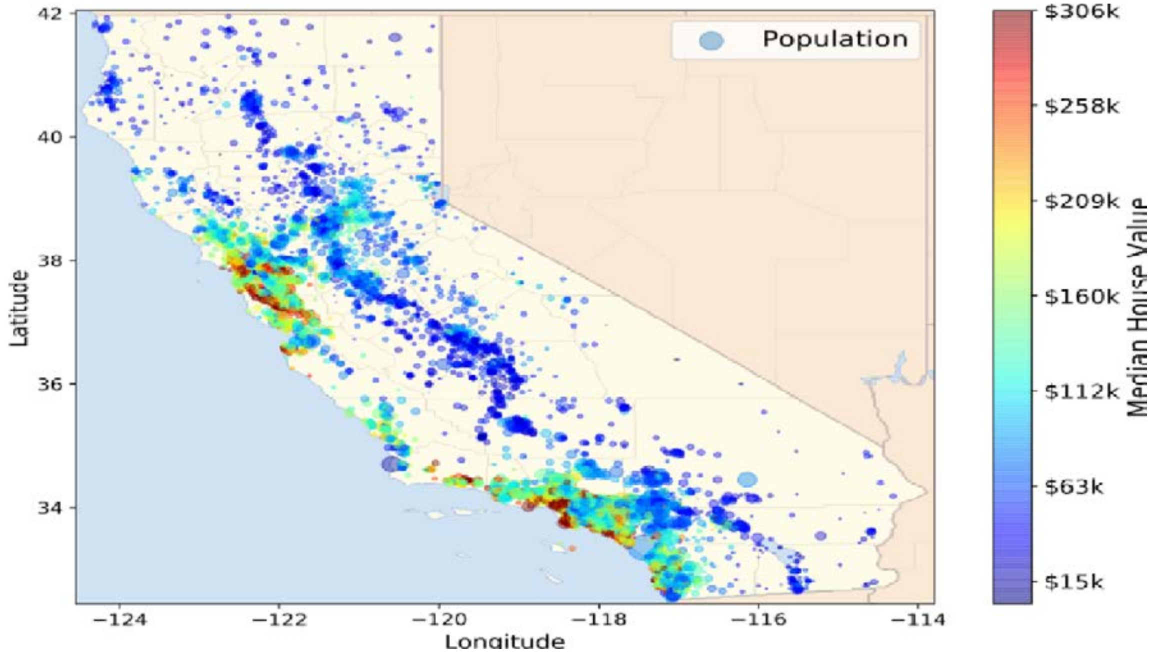
- **Pandas** is a high-performing library for Python, which provides easy-to-use data structure and data analysis tools
- **Data Frame** is a 2D data structure of Pandas, which can be conceptually view as a SQL table or spreadsheet.
- **Scikit-Learn** is a leading ML library for Python, which implements most common ML algorithms.
- The ML algorithms implemented in Scikit-Learn work on Pandas Data Frames.

Steps of A Typical Real-life Big Data Project

1. Look at the big picture
2. Get the data
3. Discover and virtualise the data to gain insight
4. Prepare the data for machine learning algorithms
5. Select a model to train it
6. Fine-tune your model
7. Present your solution
8. Launch, monitor and maintain your system

Look at the Big Picture

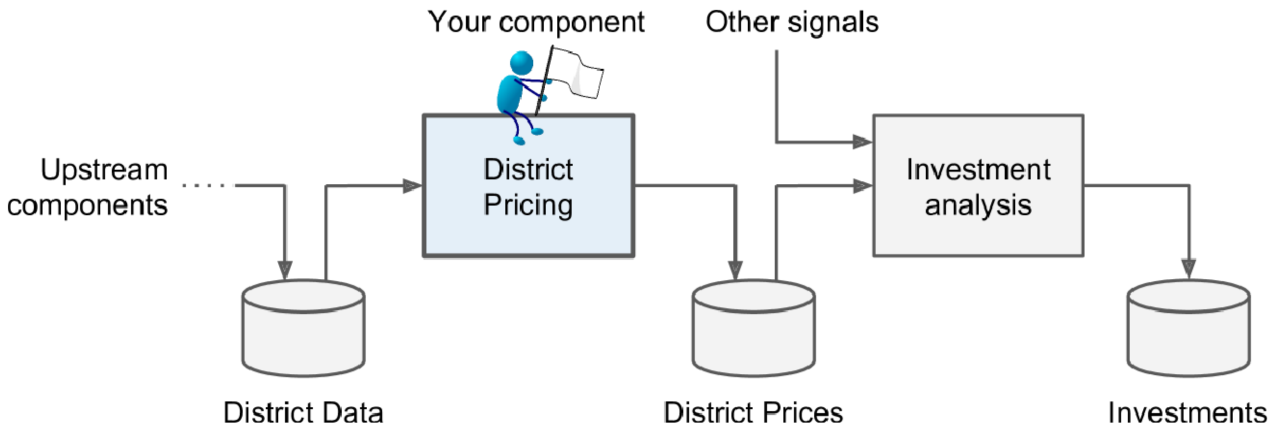
- The Task: Build a model of housing prices in California using the 1990 California census data
  - ❖ https://github.com/ageron/handson-ml/tree/master/datasets/housing
  - – Metrics includes the population, median income, median housing price, and so on for each block group in California



UNIVERSITY OF WOLLONGONG AUSTRALIA

Frame the Problem

- Building an ML model is NOT the end goal (at least model of the time)!
- In practice, the output of your model will be one input into other systems
- Say, another ML model that determines whether an house is worthy of investment.
- A data pipeline is a sequence of data processing components which is usually illustrated with a data-flow graph.



Frame the Problem

- It is a supervised learning task.
  - The training examples are labelled (i.e., each instance comes with the expected output, i.e., the district's median housing price)
  - Moreover, it is a univariate regression problem.
- Select a performance measure:
  - Root Mean Square Error (RMSE):

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

  - Mean Absolute Error:

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

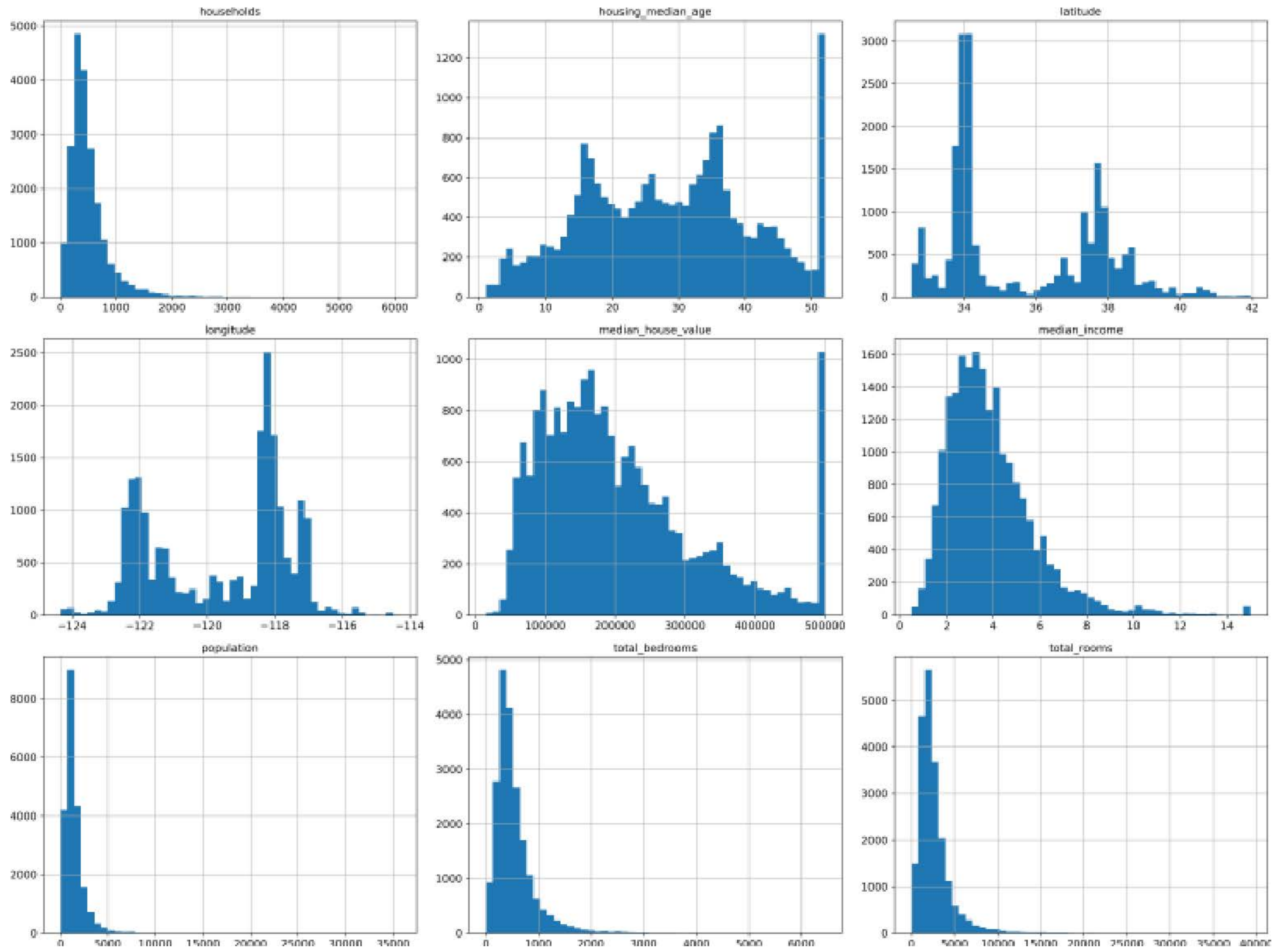
  - Other *vector norms*.

Load the Dataset

- Download data and read it using the library **Pandas**

```
> import pandas as pd
> housing = pd.read_csv("house.csv")
> housing.info
```

Explore the Data



Create Test Data

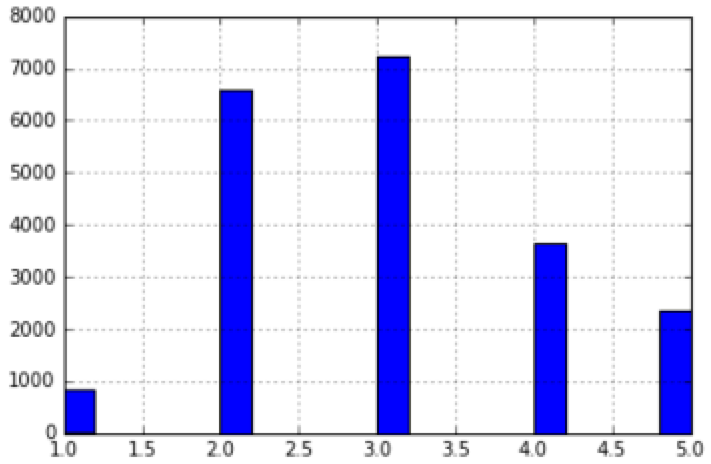
- How do we know our model can accurately predict future data or not?
- The basic idea: Use a portion of the data to test the performance of your model (based on your selected performance measure)
- Random sampling: select the test data randomly

```
> import numpy as np
> def split_train_test(data, test_ratio):
    shuffled_indices = np.random.permutation(len(data))
    test_set_size = int(len(data) * test_ratio)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]
> train_set, test_set = split_train_test(housing, 0.2)
```

Create Test Data

- *Stratified sampling*
  - the population is divided into homogeneous subgroups called *strata*, and the right number of instances is sampled from each stratum to guarantee that the test set is representative of the overall population.
  - In ML terms, we refer each "stratum" as a "bin"
- Suppose you decide to do stratified sampling based on the "income" attribute.

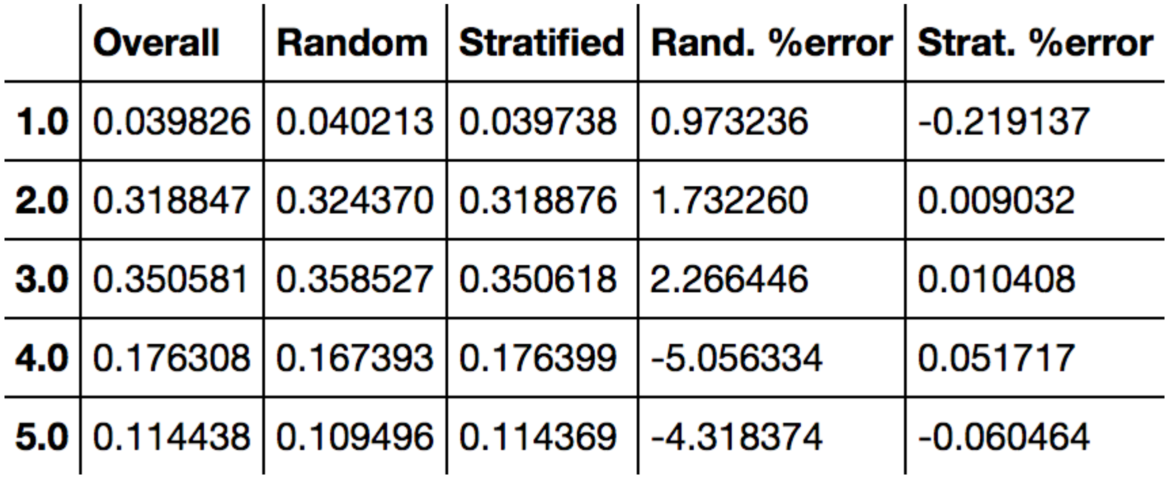
```
> housing["income_cat"] = \
    pd.cut(housing["median_income"],
    bins=[0., 1.5, 3.0, 4.5, 6.,
           np.inf],
> labels=[1, 2, 3, 4, 5])
> housing["income_cat"].hist()
```



Create Test Data

```
> from sklearn.model_selection import StratifiedShuffleSplit
> split = StratifiedShuffleSplit(n_splits=1, test_size=0.2,
    random_state=42)
> for train_index, test_index in split.split(housing,
    housing["income_cat"]):
    strat_train_set = housing.loc[train_index]
    strat_test_set = housing.loc[test_index]
```

- Compare random sampling and stratified sampling



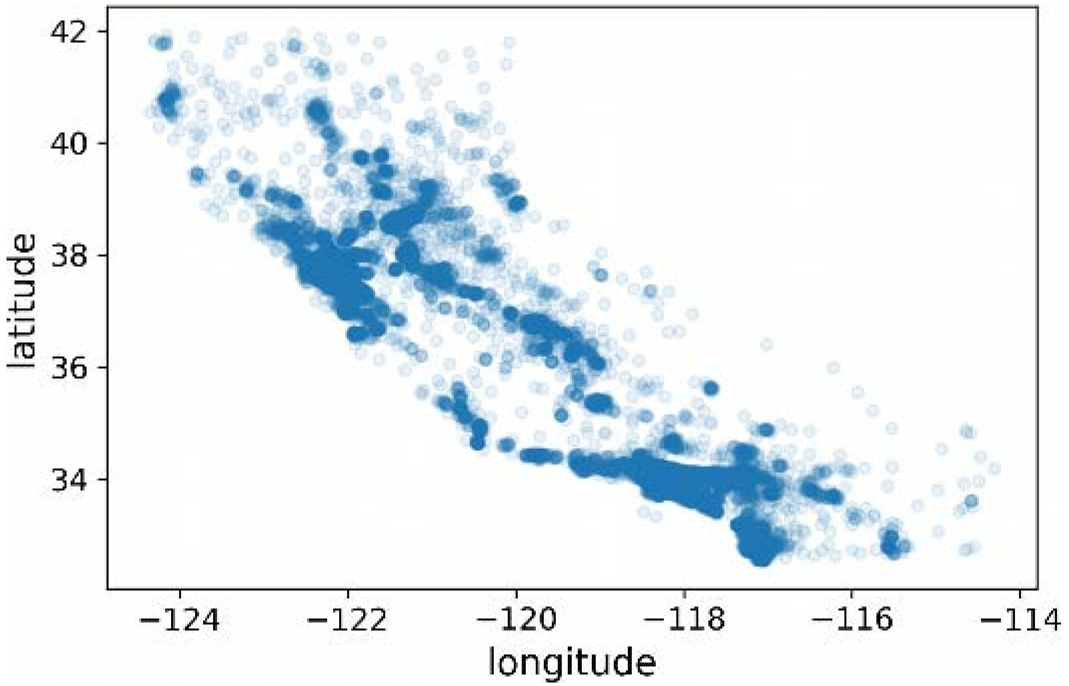
	Overall	Random	Stratified	Rand. %error	Strat. %error
1.0	0.039826	0.040213	0.039738	0.973236	-0.219137
2.0	0.318847	0.324370	0.318876	1.732260	0.009032
3.0	0.350581	0.358527	0.350618	2.266446	0.010408
4.0	0.176308	0.167393	0.176399	-5.056334	0.051717
5.0	0.114438	0.109496	0.114369	-4.318374	-0.060464

UNIVERSITY
OF WOLLONGONG
AUSTRALIA

Virtualise Data and Gain Insight

- Virtualise the geographical information

```
> housing = strat_train_set.copy()
> housing.plot(kind="scatter", x="longitude", y="latitude",
alpha=0.1)
```

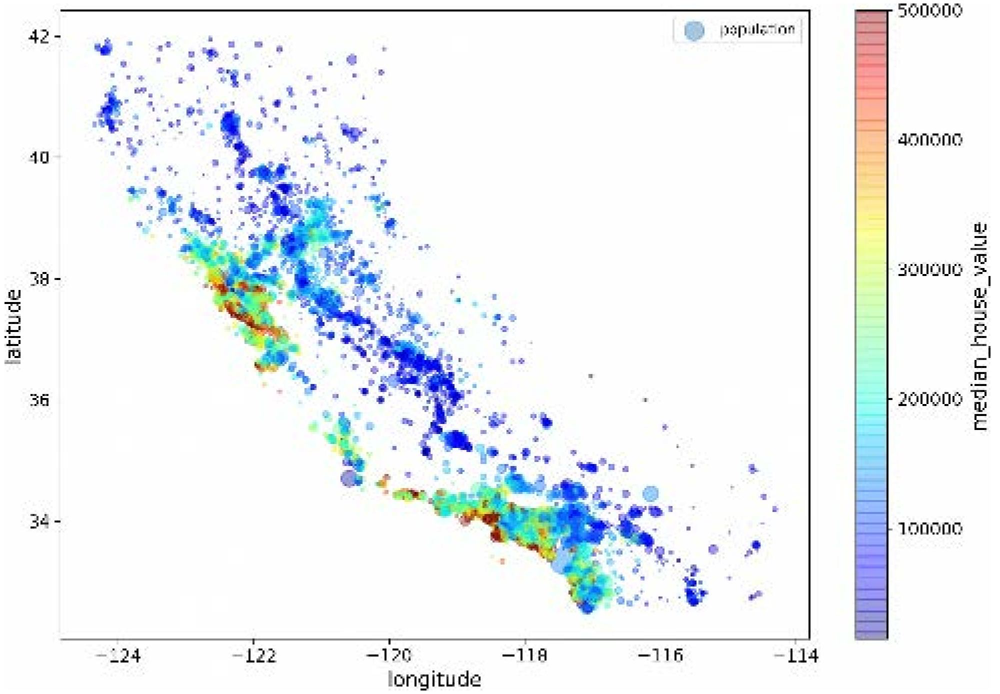


UNIVERSITY OF WOLLONGONG AUSTRALIA

Virtualise Data and Gain Insight

- Virtualise the Housing price and population:

```
> housing.plot(kind="scatter", x="longitude", y="latitude",
  alpha=0.4, s=housing["population"]/100, label="population",
  figsize=(10,7), c="median_house_value",
  cmap=plt.get_cmap("jet"), colorbar=True,)
> plt.legend()
```



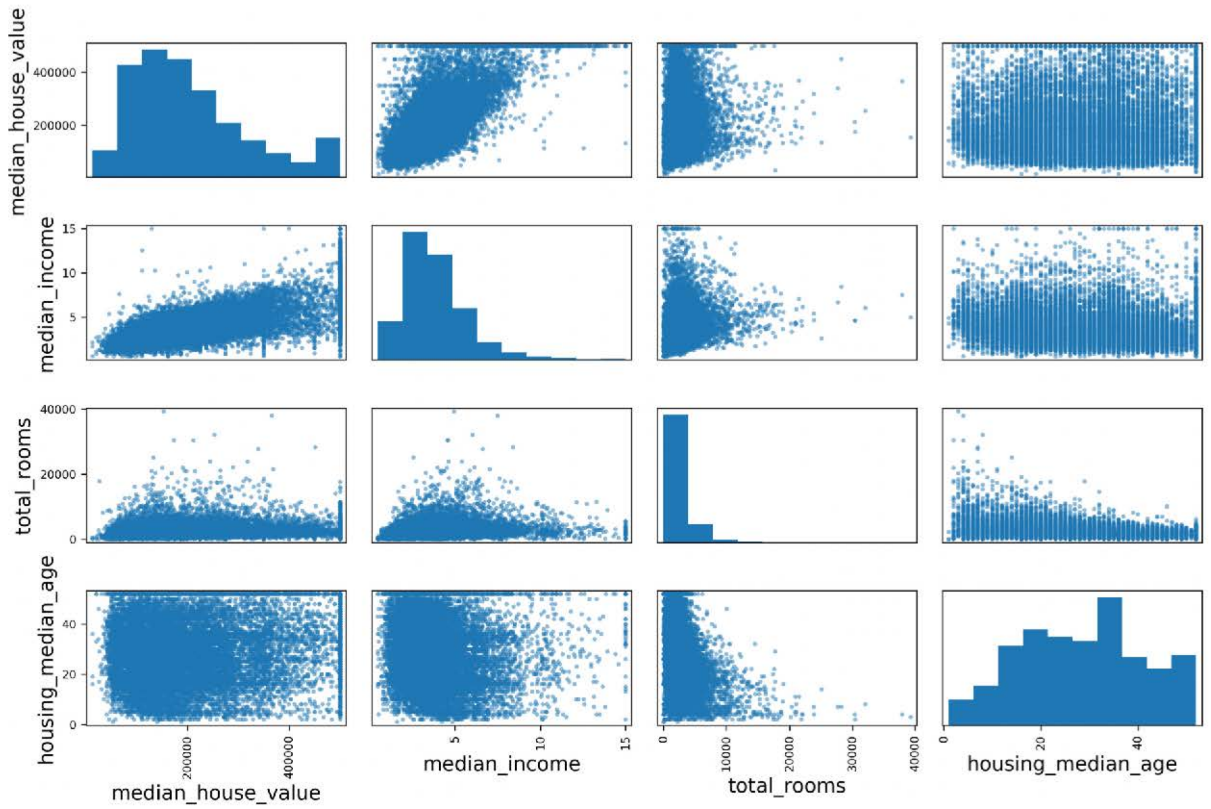
Attribute Correlation

- Intuitively, the correlation coefficient only measures linear correlations
  - E.g., “if *x* goes up, then *y* generally goes up/down”.
  - It may completely miss out on nonlinear relationships

```
> corr_matrix = housing.corr()
> corr_matrix["median_house_value"].\
    sort_values(ascending=False)
```

Attribute Correlation

```
> from pandas.plotting import scatter_matrix
> attributes = ["median_house_value", "median_income",
"total_rooms", "housing_median_age"]
> scatter_matrix(housing[attributes], figsize=(12, 8))
```



UNIVERSITY OF WOLLONGONG AUSTRALIA

Attribute Combination

- You can create additional attributes based on the existing ones:

```
> 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"]
> corr_matrix = housing.corr()
> corr_matrix["median_house_value"].sort_values(ascending=False)

median_house_value                 1.000000
median_income                      0.687160
rooms_per_household                0.146285
…
population_per_household          -0.021985
…
bedrooms_per_room                 -0.259984
```

Prepare Data for ML Algorithms

- Data Cleaning
  - Handling missing features (three options)

1. Get rid of the corresponding districts.
2. Get rid of the whole attribute.
3. Set the values to some value (zero, the mean, the median, etc.).

```
> housing = strat_train_set.drop("median_house_value", axis=1)
> housing_labels = strat_train_set["median_house_value"].copy()

> housing.dropna(subset=["total_bedrooms"]) # option 1
> housing.drop("total_bedrooms", axis=1) # option 2
> median = housing["total_bedrooms"].median() # option 3
> housing["total_bedrooms"].fillna(median, inplace=True)
```

UNIVERSITY OF WOLLONGONG AUSTRALIA

Data Cleaning

- Using SimpleImputer from Scikit-Learn

```
> from sklearn.impute import SimpleImputer
> imputer = SimpleImputer(strategy="median")
> housing_num = housing.drop("ocean_proximity", axis=1)
> imputer.fit(housing_num)
> imputer.statistics_
    array([ -118.51 , 34.26 , 29. , 2119.5 ,
             433. , 1164. , 408. , 3.5409])
```

- Use this “trained” imputer to transform the training set by replacing missing values by the learned medians
- As the returned is a Numpy array, convert it to be a DataFrame

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

Handling Categorical Features

- Recall that house has a text attribute ocean_proximity
- Most ML algorithms prefer to work with numbers, so we encode the categories by numbers.
  - Ordinary encoding: categories are encoded as scalars (fast, but the distances between encoded categories are comparable, e.g., 0 < 1 < 4)
  - One-hot encoding: categories are encoded as vectors (so that their distances are not comparable)

```
> from sklearn.preprocessing import OrdinalEncoder
> ordinal_encoder = OrdinalEncoder()
> housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
> from sklearn.preprocessing import OneHotEncoder
> cat_encoder = OneHotEncoder()
> housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
```

User-Defined Transformer

- To define a custom transformer in Scikit-Learn, need to implement three methods:
  - **fit()** (always returning self) and **transform()**
  - And fit_transformer() (which we can get just by adding TransformerMixin as a base class).
  - If also adding BaseEstimator as a base class, you get two extra methods get_params() and set_params() that will be useful for automatic hyperparameter tuning later.

User-Defined Transformer

```
> from sklearn.base import BaseEstimator, TransformerMixin
> rooms_ix, bedrooms_ix, population_ix, households_ix = 3, 4, 5, 6
> class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
    def __init__(self, add_bedrooms_per_room = True):
        self.add_bedrooms_per_room = add_bedrooms_per_room
    def fit(self, X, y=None):
        return self # nothing else to do
    def transform(self, X, y=None):
        rooms_per_household = X[:, rooms_ix] / X[:, households_ix]
        population_per_household =
            X[:, population_ix] / X[:, households_ix]
        if self.add_bedrooms_per_room:
            bedrooms_per_room = X[:, bedrooms_ix] / X[:, rooms_ix]
            return np.c_[X, rooms_per_household,
                         population_per_household, bedrooms_per_room]
        else:
            return np.c_[X, rooms_per_household,
                            population_per_household]

> attr_adder = \
    CombinedAttributesAdder(add_bedrooms_per_room=False)
> housing_extra_attribs = attr_adder.transform(housing.values)
```

Feature Scaling

- Most ML algorithms do not perform well with numerical attributes with very different scales.
- Min-max scaling (normalisation): linearly scaling to range in some interval (usually [0, 1])
  - Scikit-Learn provides MinMaxScale
- Standardisation: subtracts the mean value (so standardized values always have a zero mean), and then divides by the standard deviation (so the resulted distribution has a unit deviation)
  - Scikit-Learn provides StandardScaler

Transformation Composition

- Transformation concatenation: Scikit-Learn provides a high-level API Pipeline to build a sequence of transformers.

```
> from sklearn.pipeline import Pipeline
> from sklearn.preprocessing import StandardScaler
> num_pipeline = Pipeline([
          ('imputer', SimpleImputer(strategy="median")),
          ('attribs_adder', CombinedAttributesAdder()),
          ('std_scaler', StandardScaler()) ])
> housing_num_tr = num_pipeline.fit_transform(housing_num)
```

- Branching composition: It provides ColumnTransformer

```
> from sklearn.compose import ColumnTransformer
> num_attribs = list(housing_num)
> cat_attribs = ["ocean_proximity"]
> full_pipeline = ColumnTransformer([
           ("num", num_pipeline, num_attribs),
           ("cat", OneHotEncoder(), cat_attribs)])
> housing_prepared = full_pipeline.fit_transform(housing)
```

Train a Model and Evaluation

- Suppose you have three models in mind: Linear Regression, Decision Tree, Random Forest
  – (Will dive into those three models in future, for now see them as black boxes…)
- Build one with Linear Regression

```
> from sklearn.linear_model import LinearRegression
> lin_reg = LinearRegression()
> lin_reg.fit(housing_prepared, housing_labels)
>
> some_data = housing.iloc[:5]
> some_labels = housing_labels.iloc[:5]
> some_data_prepared = full_pipeline.transform(some_data)
> print("Predictions:", lin_reg.predict(some_data_prepared))
Predictions: [ 210644.6045 317768.8069
               210956.4333 59218.9888 189747.5584]
> print("Labels:", list(some_labels))
Labels: [286600.0, 340600.0, 196900.0, 46300.0, 254500.0]
```

Train a Model and Evaluation

- Evaluate with RSME

```
> from sklearn.metrics import mean_squared_error
> housing_predictions = lin_reg.predict(housing_prepared)
> lin_mse = mean_squared_error(housing_labels,
        housing_predictions)
> lin_rmse = np.sqrt(lin_mse)
> lin_rmse
68628.19819848922
```

- Try Decision Tree

```
> from sklearn.tree import DecisionTreeRegressor
> tree_reg = DecisionTreeRegressor()
> tree_reg.fit(housing_prepared, housing_labels)
> housing_predictions = tree_reg.predict(housing_prepared)
> tree_mse = mean_squared_error(housing_labels,
        housing_predictions)
> tree_rmse = np.sqrt(tree_mse)
> tree_rmse
0.0
```

What?! Actually, this phenomenon is called overfitting.

Evaluation with Cross-Validation

- One better way of evaluation is to split the training data set into a smaller training set and a validation set and then evaluate the trained model against the validation set.
- K-folder cross validation: randomly split the dataset into K folders, choose K-1 for training and 1 for validation; the whole process repeats K times and calculate the mean performance
- For the Decision Tree model:

```
> from sklearn.model_selection import cross_val_score
> scores = cross_val_score(tree_reg, housing_prepared,
    housing_labels, scoring="neg_mean_squared_error", cv=10)
> tree_rmse_scores = np.sqrt(-scores)
> def display_scores(scores):
    print("Mean:", scores.mean())
> display_scores(tree_rmse_scores)
Mean: 71407.68766037929
Standard deviation: 2439.4345041191004
```

UNIVERSITY OF WOLLONGONG AUSTRALIA

Evaluation with Cross-Validation

- For the Linear Regression model:

```
> lin_scores = cross_val_score(lin_reg, housing_prepared,
    housing_labels, scoring="neg_mean_squared_error", cv=10)
> lin_rmse_scores = np.sqrt(-lin_scores)
> display_scores(lin_rmse_scores)
Mean: 69052.46136345083
Standard deviation: 2731.674001798348
```

- Indeed, the Decision Tree model performs more poorly.
- Try one last model, Random Forest:

```
> from sklearn.ensemble import RandomForestRegressor
> forest_reg = RandomForestRegressor()
> forest_reg.fit(housing_prepared, housing_labels)
> display_scores(forest_rmse_scores)
Mean: 50182.303100336096
Standard deviation: 2097.0810550985693
```

Fine-Tune the Model

- There are a lot of different ML algorithms. In practice, try out some of them to shortlist several.
- The next step is to find tune the selected algorithms.
- Grid Search: Specify the range of each hyperparameter and exhaust all possible combinations

```
> 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)
> grid_search.best_params_
{'max_features': 8, 'n_estimators': 30}
```

<= What do they mean?

UNIVERSITY OF WOLLONGONG AUSTRALIA

Evaluate on Test Data

- Finally, you evaluate the performance of the best trained with the reserved test data (which is never used in the whole training process).

```
> final_model = grid_search.best_estimator_
> X_test = strat_test_set.drop("median_house_value", axis=1)
> y_test = strat_test_set["median_house_value"].copy()
> X_test_prepared = full_pipeline.transform(X_test)
> final_predictions = final_model.predict(X_test_prepared)
> final_mse = mean_squared_error(y_test, final_predictions)
> final_rmse = np.sqrt(final_mse)
```

- Important note: The ultimate goal is to train a model that generalises well for all unseen data.
  - Overfitting means the opposite.

UNIVERSITY OF WOLLONGONG AUSTRALIA

Launch, Monitor and Maintain Your System

- Put your solution system into production
  - E.g., plugging the input data sources into your system
- Write code to monitor the live performance
  - Model performance (e.g., RMSE)
  - Run time
  - Input data quality
- Update (i.e., re-train) the model regularly based on up-to-date data
  - Offline training vs. online training
  - Working version and updating version