End-to-End Machine Learning project

Submitted by Manish Bafna
Student Id: 19655
Instructor: Dr. Henry Chang
GIT: [End-to-End Machine Learning Project]

Table of Content

Look at the big picture.

The task is to use California census data to build a model of housing prices in the state. This data includes metrics such as the population, median income, and median housing price for each block group in California. Block groups are the smallest geographical unit for which the US Census Bureau publishes sample data (a block group typically has a population of 600 to 3,000 people)

The model will learn from this data and be able to predict the median housing price in any district, given all the other metrics.

The model’s output (a prediction of a district’s median housing price) will be fed to another Machine Learning system, along with many other signals.

This downstream system will determine whether it is worth investing in a given area or not. Getting this right is critical, as it directly affects revenue.

Look at the big picture.

What the current solution looks like (if any)?

- The district housing prices are currently estimated manually by experts:
  - A team gathers up-to-date information about a district, and when they cannot get the median housing price, they estimate it using complex rules.
- This is costly and time-consuming, and their estimates are not great;
  - In cases where they manage to find out the actual median housing price, they often realize that their estimates were off by more than 10%.
  - This is why the company thinks that it would be useful to train a model to predict a district’s median housing price given other data about that district.
  - The census data looks like a great dataset to exploit for this purpose, since it includes the median housing prices of thousands of districts, as well as other data.

Look at the big picture.

Frame the problem

Is it supervised, unsupervised, or Reinforcement Learning?

- It is a typical supervised learning task since we have labeled training examples (each instance comes with the expected output, i.e., the district's median housing price).

Is it a classification task, a regression task, or something else?

- It is a typical regression task, since we need to predict a value. More specifically, this is a multivariate regression problem since the system will use multiple features to make a prediction (it will use the district's population, the median income, etc.).

Should batch learning or online learning techniques be used?

- There is no continuous flow of data, there is no need to adjust to changing data rapidly, and the data is small enough to fit, hence plain batch learning should be good

Look at the big picture.

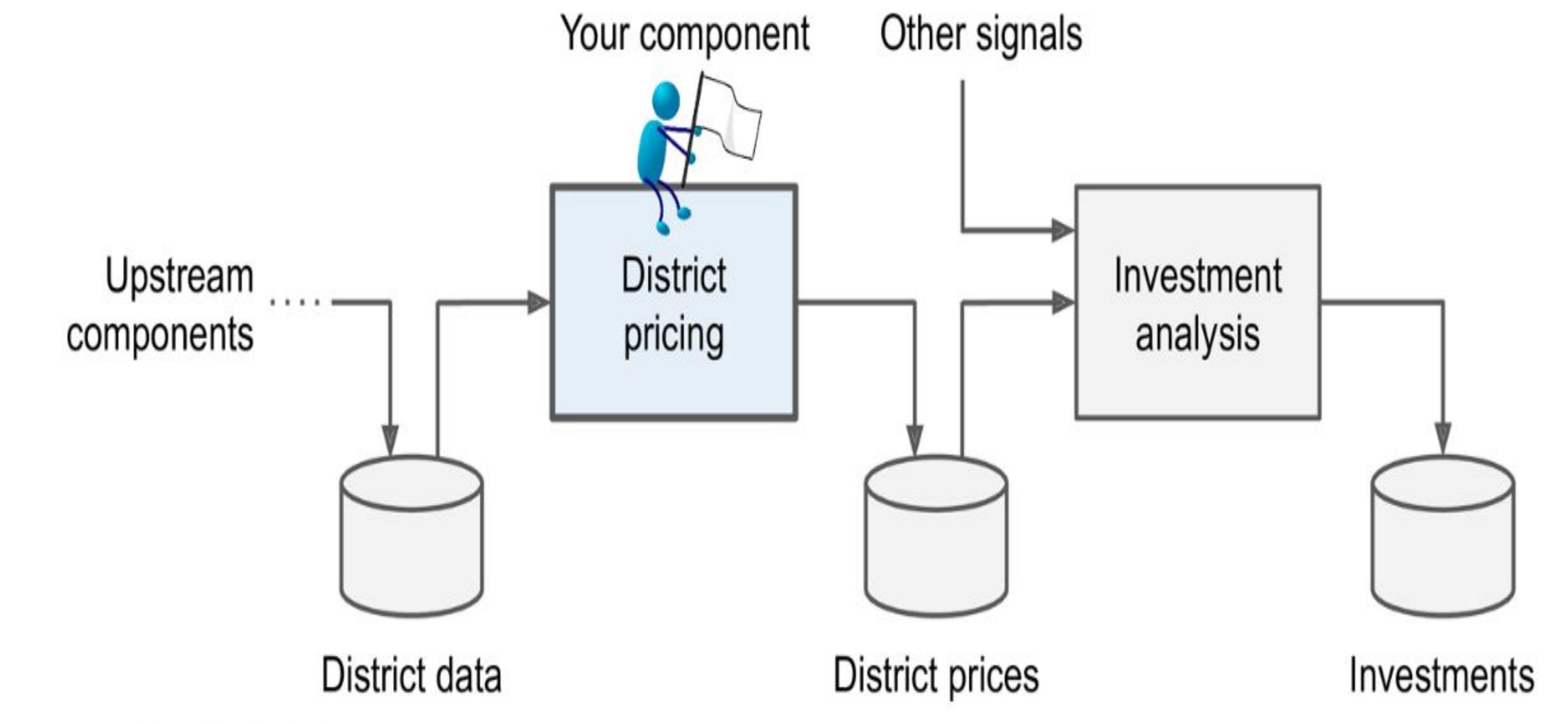


Figure 2-2. A Machine Learning pipeline for real estate investments

Look at the big picture.

Select a 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.

Equation 2-1. Root Mean Square Error (RMSE)

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

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

Look at the big picture.

Check the Assumptions

- List and verify the assumptions that were made so far
  - List the assumptions
    - The district prices that the system outputs are going to be fed and used in a downstream Machine Learning system.
  - Verify the assumptions
    - What if the downstream system actually converts the district prices into categories (e.g., “cheap,” “medium,” or “expensive”) and then uses those categories instead of the prices themselves?
      - In this case, getting the price perfectly right is not important at all; the system just needs to get the category right.
        - If that’s so, then the problem should have been framed as a *classification task*, not a *regression task*.

Look at the big picture.

Check the Assumptions

- List and verify the assumptions that were made so far
    - List the assumptions
        - The district prices that the system outputs are going to be fed and used in a downstream Machine Learning system.
    - Verify the assumptions
        - What if the downstream system actually converts the district prices into categories (e.g., “cheap,” “medium,” or “expensive”) and then uses those categories instead of the prices themselves?
            - In this case, getting the price perfectly right is not important at all; the system just needs to get the category right.
                - If that’s so, then the problem should have been framed as a *classification task*, not a *regression task*.

Get the data

For this project, we'll use the California Housing Prices dataset from the StatLib repository. This dataset is based on data from the 1990 California census. It is not exactly recent (a nice house in the Bay Area was still affordable at the time), but it has many qualities for learning, so we will pretend it is recent data.

Create the Workspace: We will use Google Colab and jupyter notebook

A notebook contains a list of cells. Each cell can contain executable code or formatted text.

Get the data

Download the data:

Download a single compressed file, *housing.tgz*, which contains a comma-separated values (CSV) file called *housing.csv* with all the data.

- Option 1: Manually
  - Download housing.tgz from Internet
  - tar xzf housing.tgz

Option 2: Programmably

Step 2: Load the data to the RAM using Pandas

Get the data

```
[ ] import os
    import tarfile
    import urllib.request

    DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-ml/master/"
    HOUSING_PATH = os.path.join("datasets", "housing")
    HOUSING_URL = DOWNLOAD_ROOT + "datasets/housing/housing.tgz"

    def fetch_housing_data(housing_url=HOUSING_URL, housing_path=HOUSING_PATH):
        os.makedirs(housing_path, exist_ok=True)
        tgz_path = os.path.join(housing_path, "housing.tgz")
        urllib.request.urlretrieve(housing_url, tgz_path)
        housing_tgz = tarfile.open(tgz_path)
        housing_tgz.extractall(path=housing_path)
        housing_tgz.close()
```

```
[ ] fetch_housing_data()
```

```
[ ] import pandas as pd

    def load_housing_data(housing_path=HOUSING_PATH):
        csv_path = os.path.join(housing_path, "housing.csv")
        return pd.read_csv(csv_path)
```

Get the data

Take a Quick Look at the Data Structure: The info() method is useful to get a quick description of the data, in particular the total number of rows, and each attribute's type and number of non-null values.

- There are 10 attributes:
  - longitude
  - latitude
  - housing_median_age
  - total_rooms
  - total_bedrooms
  - population
  - households
  - median_income
  - median_house_value
  - ocean_proximity

```
In [6]: housing.info()
        <class 'pandas.core.frame.DataFrame'>
        RangeIndex: 20640 entries, 0 to 20639
        Data columns (total 10 columns):
        longitude             20640 non-null float64
        latitude              20640 non-null float64
        housing_median_age    20640 non-null float64
        total_rooms           20640 non-null float64
        total_bedrooms        20433 non-null float64
        population            20640 non-null float64
        households            20640 non-null float64
        median_income         20640 non-null float64
        median_house_value    20640 non-null float64
        ocean_proximity       20640 non-null object
        dtypes: float64(9), object(1)
        memory usage: 1.6+ MB
```

Get the data

Let's look at the other fields. The describe() method shows a summary of the numerical attributes

In [8]:
```
housing.describe()
```

Out[8]:

	longitude	latitude	housing_median_age	total_rooms	total_bedr
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000
mean	-119.569704	35.631861	28.639486	2635.763081	537.87055
std	2.003532	2.135952	12.585558	2181.615252	421.38507
min	-124.350000	32.540000	1.000000	2.000000	1.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.00000
50%	-118.490000	34.260000	29.000000	2127.000000	435.00000
75%	-118.010000	37.710000	37.000000	3148.000000	647.00000
max	-114.310000	41.950000	52.000000	39320.000000	6445.0000

Get the data

Histogram for each numerical attribute

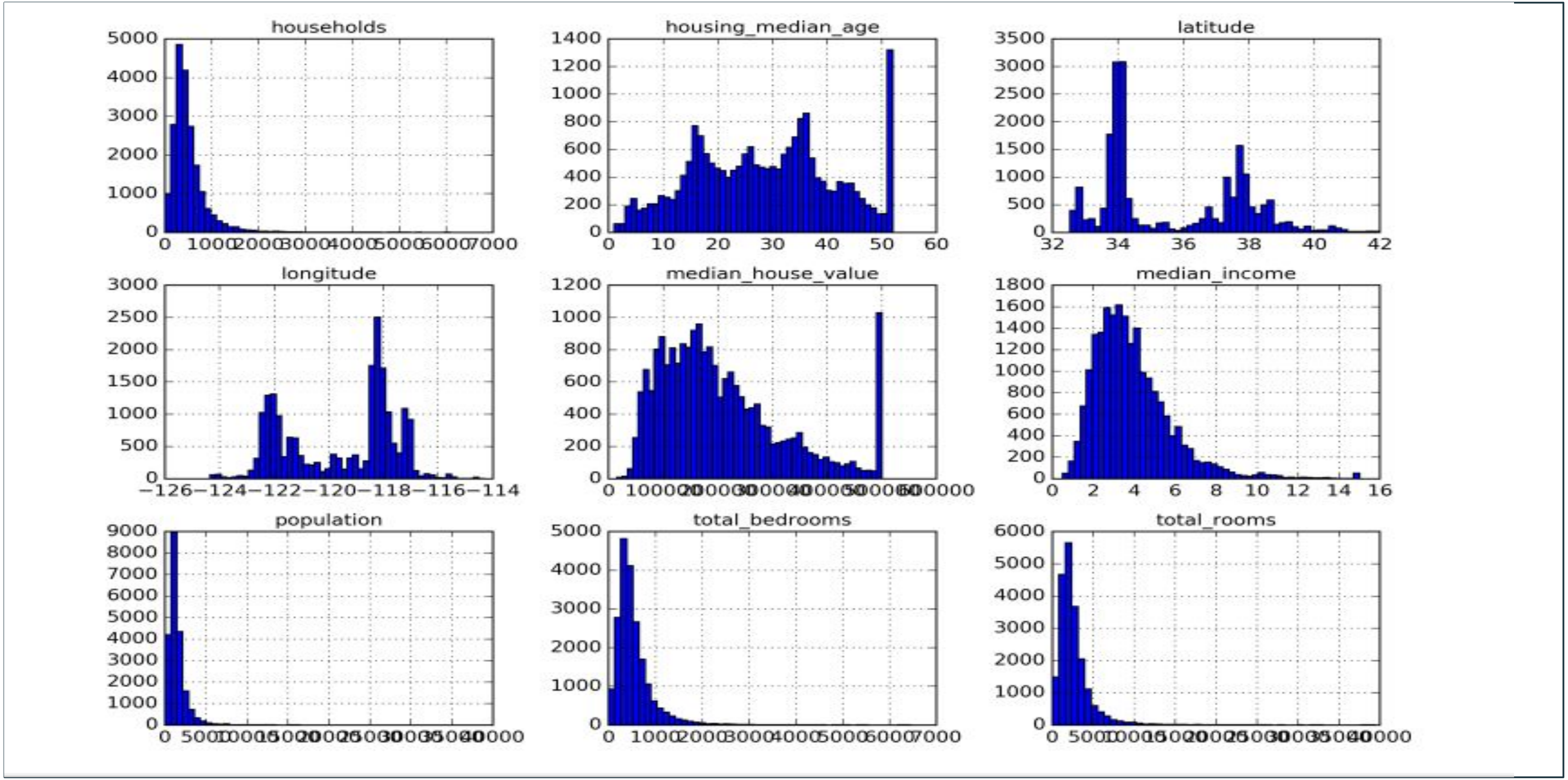
- A histogram shows the number of instances (on the vertical axis) that have a given value range (on the horizontal axis).
    - Option 1: Plot this one attribute at a time
    - Option 2: Call the hist() method on the whole dataset, and it will plot a histogram for each numerical attribute (see Figure 2-8).
        - For example, you can see that slightly over 800 districts have a median_house_value equal to about $100,000.

Get the data

The hist() method relies on Matplotlib, which in turn relies on a user-specified graphical backend Matplotlib to draw on your screen.

- The simplest option is to use Jupyter's magic command %matplotlib inline.
    - This tells Jupyter to set up Matplotlib so it uses Jupyter's own backend. Plots are then rendered within the notebook itself.
- Note that calling show() is optional in a Jupyter notebook, as Jupyter will automatically display plots when a cell is executed.

Get the data


households
housing_median_age
latitude
longitude
median_house_value
median_income
population
total_bedrooms
total_rooms

Get the data

<u>Create a Test Set</u>

Random Approach

Approach 1: Simple approach

Creating a test set is theoretically quite simple: just pick some instances randomly, typically 20% of the dataset, and set them aside:

Issue: if you run the program again, it will generate a different test set! Over time, you (or your Machine Learning algorithms) will get to see the whole dataset, which is what you want to avoid. Possible solutions

To save the test set on the first run and then load it in subsequent runs.

To set the random number generator's seed (e.g., <u>np.random.seed</u>) before calling <u>np.random.permutation()</u>, so that it always generates the same shuffled indices.

Get the data

Approach 2: To use each instance's identifier to decide whether or not it should go in the test set (assuming instances have a unique and immutable identifier)

We can compute a hash of each instance's identifier, keep only the last byte of the hash, and put the instance in the test set if this value is lower or equal to 51 (~20% of 256).

- This ensures that the test set will remain consistent across multiple runs, even if you refresh the dataset.
    - The new test set will contain 20% of the new instances, but it will not contain any instance that was previously in the training set.

Approach 3: Unfortunately, the housing dataset does not have an identifier column. The simplest solution is to use the row index as the ID:

Get the data

Approach 4: Use Scikit-Learn's train_test_split

Scikit-Learn provides a few functions to split datasets into multiple subsets in various ways. The simplest function is train_test_split, which does pretty much the same thing as the function split_train_test defined earlier, with a couple of additional features.

1. There is a random_state parameter that allows to set the random generator seed as explained previously, and
2. Wecan pass it multiple datasets with an identical number of rows, and it will split them on the same indices (this is very useful, for example, when there are separate DataFrame for labels):

Get the data

Non-random Approach: *stratified sampling*

- So far we have considered purely random sampling methods. This is generally fine if your dataset is large enough (especially relative to the number of attributes), but if it is not, you run the risk of introducing a significant sampling bias.
- For example, the US population is composed of 51.3% female and 48.7% male, so a well-conducted survey in the US would try to maintain this ratio in the sample: 513 female and 487 male.
- This is called *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.

Get the data

Non-random Approach: *stratified sampling*

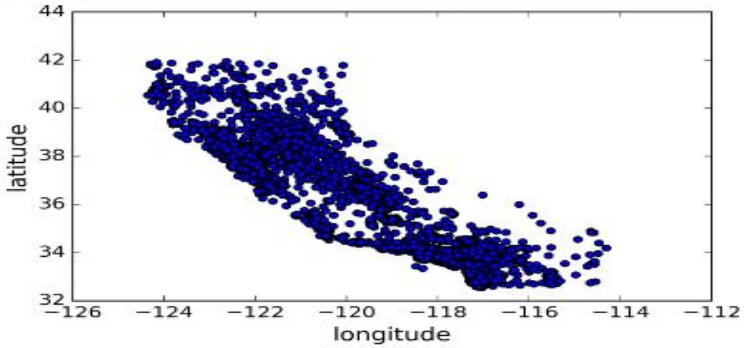
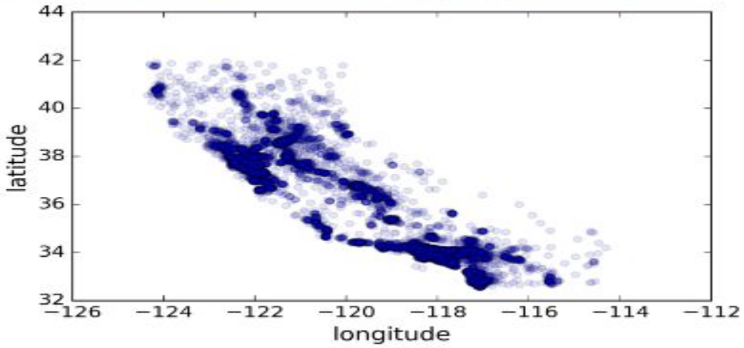
Let's look at the median income histogram more closely: most median income values are clustered around $20,000–$50,000, but some median incomes go far beyond $60,000.

- It is important to have a sufficient number of instances in your dataset for each stratum, or else the estimate of the stratum's importance may be biased.
    - This means that you should not have too many strata, and each stratum should be large enough.
    - The following code creates an income category attribute by dividing the median income by 1.5 (to limit the number of income categories), and rounding up using ceil (to have discrete categories), and then merging all the categories greater than 5 into category 5:

Discover and Visualize the Data to Gain

Visualizing Geographical Data

Population Density Information

No Patterns	Has Patterns
`housing.plot(kind="scatter", x="longitude", y="latitude")`	`housing.plot(kind="scatter", x="longitude", y="latitude", alpha=0.1)`
	
	Note: ◦ Setting the alpha option to 0.1 makes it much easier to visualize the places where there is a high density of data points ▪ the Bay Area ▪ Los Angeles ▪ San Diego ▪ A long line of fairly high density in the Central Valley, in particular around Sacramento and Fresno. ◦ You may need to play around with visualization parameters to make the patterns stand out.

Discover and Visualize the Data to Gain

Looking for Correlations

How much each attribute correlates with the Median House Value?

- Approach 1: Computing Standard Correlation Coefficient to find the correlations

Code

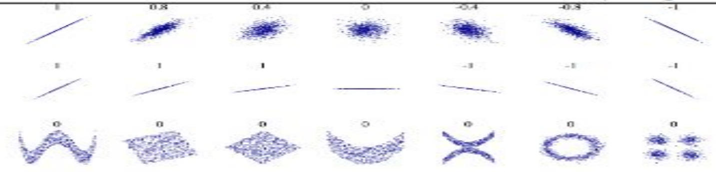
```
# Since the dataset is not too large, you can easily
# compute the standard correlation coefficient (also
# called Pearson's r) between every pair of attributes
# using the corr() method:
corr_matrix = housing.corr()

# How much each attribute correlates with the median
# house value:
>>> corr_matrix["median_house_value"].
        sort_values(ascending=False)
median_house_value      1.000000
median_income           0.687170
total_rooms             0.135231
housing_median_age      0.114220
households              0.064702
total_bedrooms          0.047865
population             -0.026699
longitude              -0.047279
latitude               -0.142826
Name: median_house_value, dtype: float64
```

Note:

- The correlation coefficient only measures linear correlations ("if *x* goes up, then *y* generally goes up/down").
  - It may completely miss out on nonlinear relationships (e.g., "if *x* is close to zero then *y* generally goes up").
  - How all the plots of the bottom row have a correlation coefficient equal to zero despite the fact that their axes are clearly not independent: these are examples of nonlinear relationships.
  - The second row shows examples where the correlation coefficient is equal to 1 or −1; notice that this has nothing to do with the slope.
    - For example, your height in inches has a correlation coefficient of 1 with your height in feet or in nanometers.

Standard correlation coefficient of various datasets (Wikipedia)



Note:

- **The correlation coefficient ranges from −1 to 1.**
  - **~ 1 ==> strong positive correlation**
    - **For example, the median house value tends to go up when the median income goes up.**
  - **~ −1 ==> strong negative correlation**
    - **For example, a small negative correlation between the latitude and the median house value (i.e., prices have a slight tendency to go down when you go north).**
  - **~0 ==> no linear correlation.**

Discover and Visualize the Data to Gain

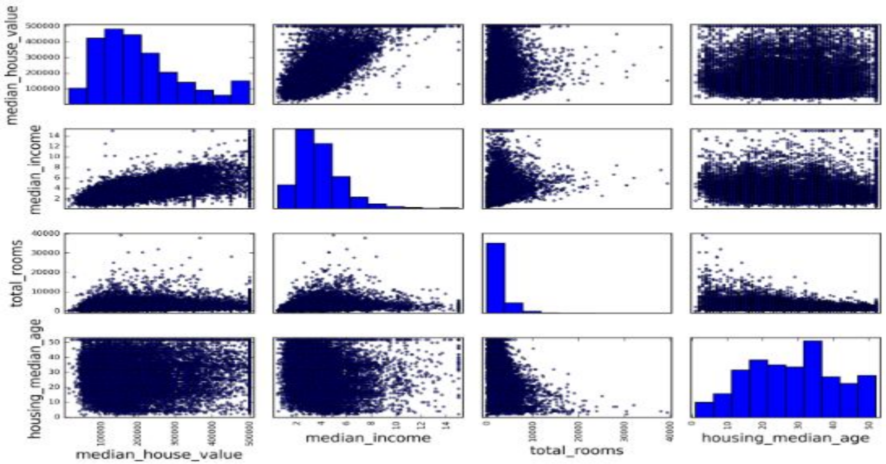
Looking for Correlations

Approach 2: Using Pandas' scatter_matrix to find the correlations visually.

- Since there are now 11 numerical attributes, you would get $11^2 = 121$ plots, which would not fit on a page, so let's just focus on a few promising attributes that seem most correlated with the median housing value

```
from pandas.tools.plotting import scatter_matrix

attributes = ["median_house_value", "median_income", "total_rooms",
              "housing_median_age"]
scatter_matrix(housing[attributes], figsize=(12, 8))
```



Discover and Visualize the Data to Gain

Experimenting with Attribute Combinations

Step 1: Review What you learned in the previous sections:

- Identified a few data quirks
  - May want to clean up before feeding the data to a Machine Learning algorithm.
- Found interesting correlations between attributes, in particular with the target attribute (i.e., median house price).
- Found that some attributes have a tail-heavy distribution
  - May want to transform them (e.g., by computing their logarithm).

Discover and Visualize the Data to Gain

Experimenting with Attribute Combinations

Step 1: Try out various attribute combinations.

- The total number of rooms in a district is not very useful if you don’t know how many households there are.
    - What you really want is the number of rooms per household.
      housing["rooms_per_household"] = housing["total_rooms"]/housing["households"]
- The total number of bedrooms by itself is not very useful:
    - you probably want to compare it to the number of rooms.
      housing["bedrooms_per_room"] = housing["total_bedrooms"]/housing["total_rooms"]
- The population per household also seems like an interesting attribute combination to look at.

housing["population_per_household"]=housing["population"]/housing["households"]

Discover and Visualize the Data to Gain

Experimenting with Attribute Combinations

Step 1: Let's look at the correlation matrix again:

Before Attribute Combinations	After Attribute Combinations

`>>> corr_matrix["median_house_value"].`
`sort_values(ascending=False)`
`median_house_value 1.000000`
`median_income 0.687170`
`total_rooms 0.135231`
`housing_median_age 0.114220`
`households 0.064702`
`total_bedrooms 0.047865`
`population -0.026699`
`longitude -0.047279`
`latitude -0.142826`
`Name: median_house_value, dtype: float64`
	`>>> 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`
	`total_rooms 0.135097`
	`housing_median_age 0.114110`
	`households 0.064506`
	`total_bedrooms 0.047689`
	`population_per_household -0.021985`
	`population -0.026920`
	`longitude -0.047432`
	`latitude -0.142724`
	`bedrooms_per_room -0.259984`
	`Name: median_house_value, dtype: float64`
	
	Note:
	◦ **The new bedrooms_per_room attribute is much more correlated with the median house value than the total number of rooms or bedrooms.**
	▪ Apparently houses with a lower bedroom/room ratio tend to be more expensive.
	▪ The number of rooms per household is also more informative than the total number of rooms in a district - obviously the larger the houses, the more expensive they are.

Discover and Visualize the Data to Gain

Experimenting with Attribute Combinations

Step 1: Step 4: Continue from Step 1 to 3

This is an iterative process:

- Once you get a prototype up and running, you can analyze its output to gain more insights and come back to this exploration step.

Prepare the Data for Machine Learning

It's time to prepare the data for the Machine Learning algorithms.

Instead of just doing this manually, we need to write functions to do that, for several good reasons:

This will allow to reproduce these transformations easily on any dataset

Gradually build a library of transformation functions that can be reused in future projects.

These functions can be used in live system to transform the new data before feeding it to algorithms.

This will make it possible to easily try various transformations and see which combination of transformations works best.

Prepare the Data for Machine Learning

Data Cleaning

Fix missing features

- Pandas' Approach to replace the missing values of only one attribute: DataFrame's dropna(), drop(), and fillna() methods

207 (= 20640 - 20433) districts are missing the value of total_bedrooms	Fix the missing value

<pre>housing.info()
 	
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
longitude   20640 non-null float64
latitude   20640 non-null float64
housing_median_age   20640 non-null float64
total_rooms   20640 non-null float64
total_bedrooms   20433 non-null float64
population   20640 non-null float64
households   20640 non-null float64
median_income   20640 non-null float64
median_house_value   20640 non-null float64
ocean_proximity   20640 non-null object
dtypes: float64(9), object(1)
memory usage: 1.6+ MB</pre>Note:
◦ **There are 20640 districts, but only 20433 districts provide total_bedrooms information.**
  ▪ **Thus 207 (= 20640 - 20433) districts are missing the value of total_bedrooms.**
	<pre># Option 1: Get rid of the corresponding districts.
	housing.dropna(subset=["total_bedrooms"])
	
	# Option 2: Get rid of the whole attribute.
	housing.drop("total_bedrooms", axis=1)
	
	# Option 3: Set the missing values to some value (zero,
	# the mean, the median, etc.) on the training
	# set
	median = housing["total_bedrooms"].median()
	housing["total_bedrooms"].fillna(median, inplace=True)</pre>

- Sciki-Learn's Approach to replace missing values of all attributes: Imputer

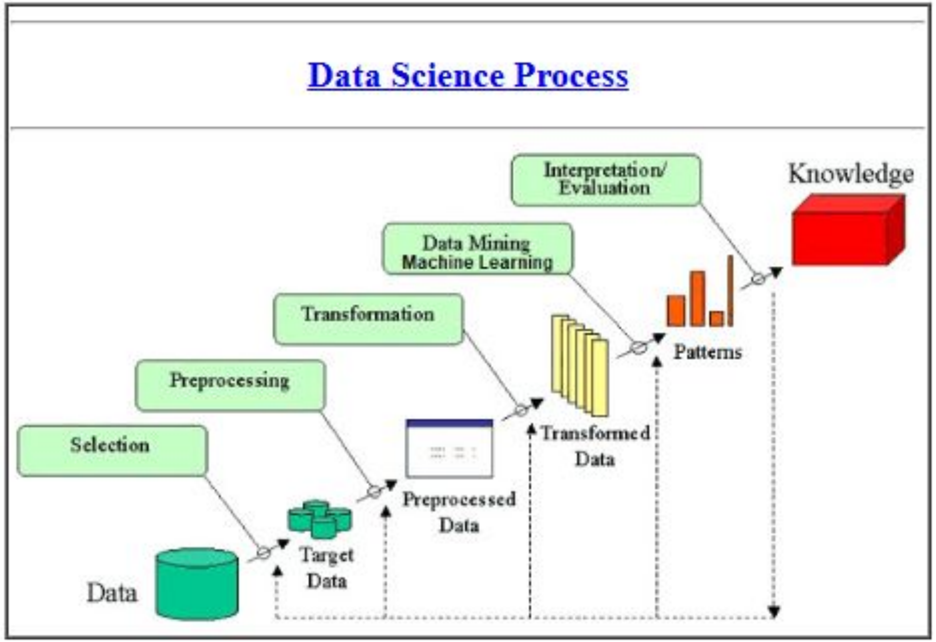
Prepare the Data for Machine Learning

Handling Text and Categorical Attributes

- Convert text categorical attribute ocean_proximity to be able to compute its median
    - Step 1: Convert from text categories to integer categories
      Note:
        - One issue with Categorical Value representation is that ML algorithms will assume that two nearby values are more similar than two distant values.
    - Step 2: Convert from integer categories to One-Hot Vectors

Prepare the Data for Machine Learning

Custom Transformers



Prepare the Data for Machine Learning

Custom Transformers

Although Scikit-Learn provides many useful transformers, we need to write code for tasks such as *custom cleanup operations* or *combining specific attributes* .

We need transformer to work seamlessly with Scikit-Learn functionalities (such as pipelines),

Since Scikit-Learn relies on duck typing (not inheritance), all we need is to create a class and implement three methods:

- fit() (returning self)
- transform()
- fit_transform()

We can get fit_transform() for free by simply adding TransformerMixin as a base class.

Prepare the Data for Machine Learning

Feature Scaling

1. Min-Max Scaling = Normalization
   - Values are rescaled to ranging from 0 to 1.
   - x_scaled = (x-min(x)) / (max(x)–min(x))
   - Sample code
   - Standardization Vs Normalization- Feature Scaling
2. Standardization.
   - Scikit-Learn provides a transformer called StandardScaler for standardization

Mean	Standard Deiviation	Standardization
$\mu = \frac{1}{N}\sum_{i=1}^{N}(x_i)$	$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i-\mu)^2}$	$z = \frac{x-\mu}{\sigma}$

Prepare the Data for Machine Learning

Transformation Pipelines

Scikit-Learn provides the Pipeline class to help with sequences of transformations.

- The Pipeline constructor takes a list of name/estimator pairs defining a sequence of steps.

```
##################################################
# A small pipeline for the numerical attributes:
##################################################
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

num_pipeline = Pipeline([
        # For Data Cleaning
        ('imputer', Imputer(strategy="median")),
        # For Custom Transformer
        ('attribs_adder', CombinedAttributesAdder()),
        # For Feature Scaling
        ('std_scaler', StandardScaler()),
    ])

housing_num_tr = num_pipeline.fit_transform(housing_num)
```

Select and Train a Model

Process

Framed the problem

Got the data and explored it

sampled a training set and a test set

Created transformation pipelines to clean up and prepare your data for Machine Learning algorithms automatically.

Select and train a Machine Learning model.

Select and Train a Model

Training and Evaluating on the Training Set -

Option 1: Linear Regression
(Regression)

- **Step 1: Train a Linear Regression model**

```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(housing_prepared, housing_labels)
```

- **Step 2: Try on a few instances from the training set:**

```
>>> 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]
```

Note:
- It works, although the predictions are not exactly accurate (e.g., the first prediction is off by close to 40%!).

- **Step 3: Measure this regression model's RMSE on the whole training set using Scikit-Learn's mean_squared_error function:**

```
>>> 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
```

Note:
- Most districts' median_housing_values range between $120,000 and $265,000, so a typical prediction error of $68,628 is not very satisfying.
  - This is an example of a model underfitting the training data. You could try to add more features (e.g., the log of the population), but first let's try a more complex model (e.g., DecisionTree Regressor) to see how it does.

Option 2: Decision Tree
(Classification)

- **Step 1: Let's train a DecisionTreeRegressor**

Note:
- This is a powerful model, capable of finding complex nonlinear relationships in the data (Decision Trees are presented in more detail in Chapter 6) .

```
from sklearn.tree import DecisionTreeRegressor

tree_reg = DecisionTreeRegressor()
tree_reg.fit(housing_prepared, housing_labels)
```

- **Step 2: Evaluate it on the training set:**

```
>>> 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
```

Note:
- Wait, No error at all? It is much more likely that the model has badly overfit the data.
  - You need to use part of the training set for training, and part for model validation.

Select and Train a Model

Better Evaluation Using Cross-Validation

Cross-Validation: K Fold vs Monte Carlo

K-Fold Cross-Validation Steps (local copy) (Example)

1. Split training data into K equal parts
2. Fit the model on k-1 parts and calculate test error using the fitted model on the kth part
3. Repeat k times, using each data subset as the test set once. (usually k= 5~20)

Select and Train a Model

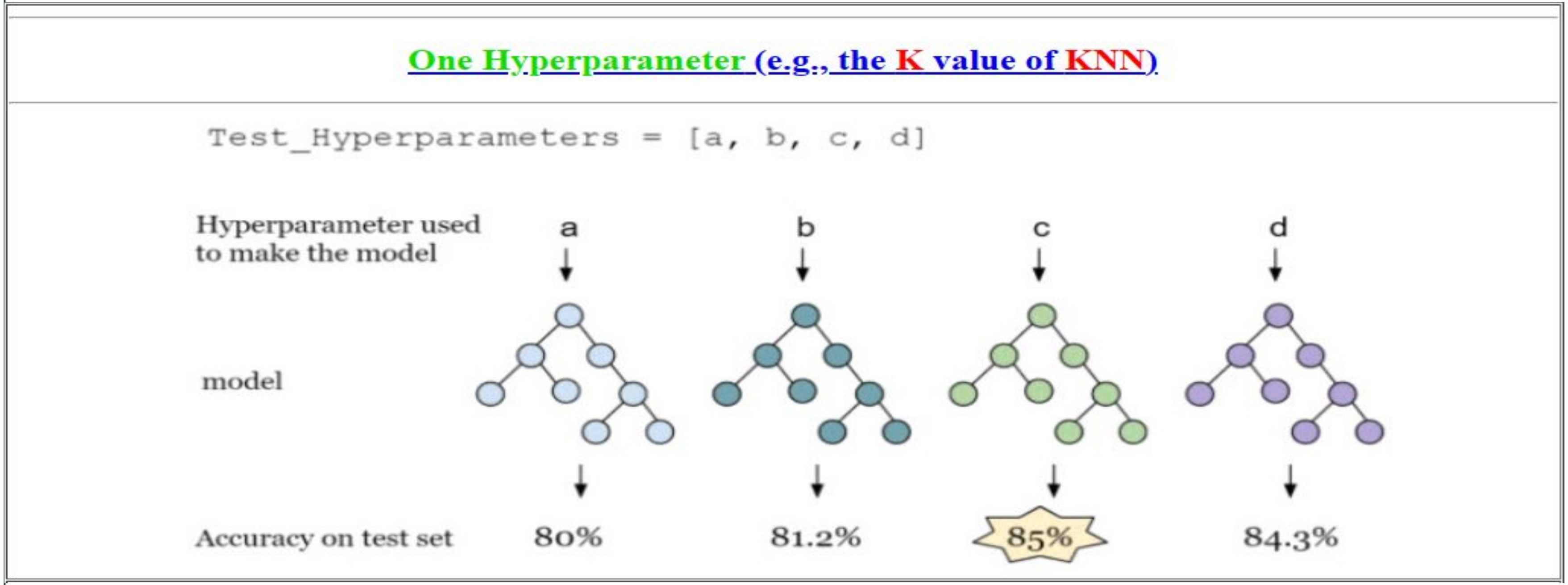
Better Evaluation Using Cross-Validation

Item	Model Parameter	Model Hyperparameter . .
Definition	**A model parameter is a configuration variable that is internal to the model and whose value can be estimated from data.** ◦ They are required by the model when making predictions. ◦ They values define the skill of the model on your problem. ◦ **They are estimated or learned from data.** ◦ They are often not set manually by the practitioner. ◦ They are often saved as part of the learned model.	A model hyperparameter is a configuration that is external to the model and whose value cannot be estimated from data. ◦ They are often used in processes to help estimate model parameters. ◦ They are often specified by the practitioner. ◦ **They can often be set using heuristics** ◦ They are often tuned for a given predictive modeling problem.
Example	◦ **The coefficients in a linear regression or logistic regression.** **Regression Equation(y) = a + bx** ◦ The weights in an artificial neural network. ◦ The support vectors in a support vector machine.	◦ **The K value of K-Nearest Neighbors (KNN).** ◦ The number and size of hidden layers for a neural network. ◦ The C and sigma hyperparameters for support vector machines.
Need to be tuned	No	Yes

Fine-Tune Your Model

Find hyperparameter values

Grid Search vs. Random Search



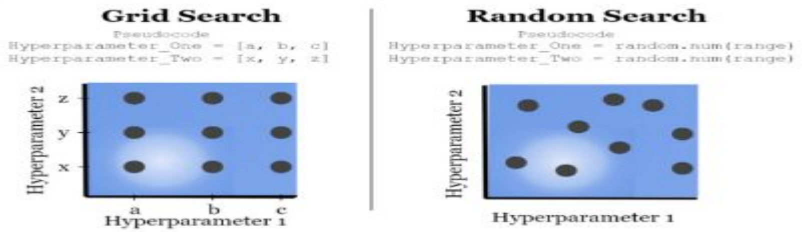
Fine-Tune Your Model

Find hyperparameter values

Grid Search vs. Random Search

Multiple Hyperparameters (e.g., SVM, Random Forest)

No dominant Hyperparameters - use Grid Search



Note:

- Assuming that white means that this certain point performs well, and blue means it doesn't.
- **The Grid Search is fine if**
  - **The hyperparameters are relatively few.**
  - **No dominant hyperparameters.**
- GridSearchCV can be used for Grid Search.

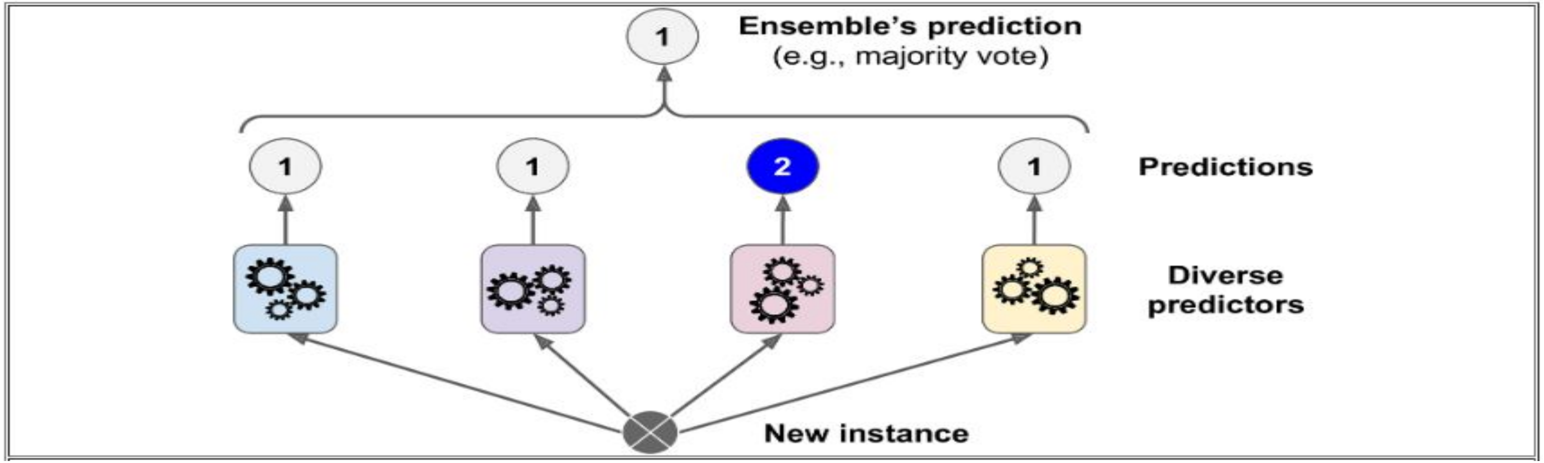
With dominant Hyperparameters (e.g., Hyperparameter 1 is more important) - use Random Search

- Conditions that Random Search is better
  1. **If one hyperparameter is more important than the others**
     - We can see the above diagrams that random search does better because of the way the values are picked.
       - In this example, grid search only tested three unique values for each hyperparameter, whereas the random search tested 9 unique values for each.
     - Think of it this way: if hyperparameter 2 doesn't really matter, then we would want 9 different hyperparameter 1 values to test instead of 3.
  2. **If the hyperparameter search space is large (i.e., higher dimensions hyperparameters).**

Fine-Tune Your Model

Find hyperparameter values

Ensemble Methods: Another way to fine-tune the system is to try to combine the models that perform best. The group (or "ensemble") will often perform better than the best individual model (just like Random Forests perform better than the individual Decision Trees they rely on), especially if the individual models make very different types of errors.



Fine-Tune Your Model

Analyze the Best Models and Their Errors

We often gain good insights on the problem by inspecting the best models.

- For example, the RandomForestRegressor can indicate the *relative importance of each attribute* for making accurate predictions:

```
>>> 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])
```

- Let's display these importance scores next to their corresponding attribute names:

```
>>> extra_attribs = ["rooms_per_hhold", "pop_per_hhold", "bedrooms_per_room"]
>>> cat_encoder = cat_pipeline.named_steps["cat_encoder"]
>>> cat_one_hot_attribs = list(cat_encoder.categories_[0])
>>> attributes = num_attribs + extra_attribs + cat_one_hot_attribs
>>> sorted(zip(feature_importances, attributes), reverse=True)
[(0.36615898061813418, 'median_income'),
 (0.16478099356159051, 'INLAND'),
 (0.10879295677551573, 'pop_per_hhold'),
 (0.073344235516012421, 'longitude'),
 (0.062909070482620302, 'latitude'),
 (0.056419179181954007, 'rooms_per_hhold'),
 (0.053351077347675809, 'bedrooms_per_room'),
 (0.041143798478729635, 'housing_median_age'),
 (0.014874280890402767, 'population'),
 (0.014672685420543237, 'total_rooms'),
 (0.014257599323407807, 'households'),
 (0.014106483453584102, 'total_bedrooms'),
 (0.010311488326303787, '<1H OCEAN'),
 (0.0028564746373201579, 'NEAR OCEAN'),
 (0.0019604155994780701, 'NEAR BAY'),
 (6.0280386727365991e-05, 'ISLAND')]
```

Fine-Tune Your Model

Analyze the Best Models and Their Errors

- With this information, we may want to try dropping some of the less useful features (e.g., apparently only one ocean_proximity category is really useful, so we can try dropping the others).
- We should also look at the specific errors that the system makes, then try to understand why it makes them and what could fix the problem (adding extra features or, on the contrary, getting rid of uninformative ones, cleaning up outliers, etc.).

Fine-Tune Your Model

Evaluate the System on the Test Set

After tweaking the models for a while, we eventually have a system that performs sufficiently well.

Now is the time to evaluate the final model on the test set.

```
final_model = grid_search.best_estimator_

# Step 1: Get the predictors and the labels from your test set
X_test = strat_test_set.drop("median_house_value", axis=1)
y_test = strat_test_set["median_house_value"].copy()

# Step 2: Run your full_pipeline to transform the data
#        (call transform(), not fit_transform()!)
X_test_prepared = full_pipeline.transform(X_test)

final_predictions = final_model.predict(X_test_prepared)

# Step 3: Evaluate the final model on the test set
final_mse = mean_squared_error(y_test, final_predictions)
final_rmse = np.sqrt(final_mse)   # => evaluates to 47,766.0
```

Fine-Tune Your Model

Evaluate the System on the Test Set

After tweaking the models for a while, we eventually have a system that performs sufficiently well.

Now is the time to evaluate the final model on the test set.

```
final_model = grid_search.best_estimator_

# Step 1: Get the predictors and the labels from your test set
X_test = strat_test_set.drop("median_house_value", axis=1)
y_test = strat_test_set["median_house_value"].copy()

# Step 2: Run your full_pipeline to transform the data
#        (call transform(), not fit_transform()!)
X_test_prepared = full_pipeline.transform(X_test)

final_predictions = final_model.predict(X_test_prepared)

# Step 3: Evaluate the final model on the test set
final_mse = mean_squared_error(y_test, final_predictions)
final_rmse = np.sqrt(final_mse)   # => evaluates to 47,766.0
```

Launch Monitor and Maintain Your System

After getting approval to launch

- Step 1: We need to get the solution ready for production.
- Step 2: Write monitoring code to check the system’s live performance at regular intervals and trigger alerts when it drops.
  - May need to get assistance from field experts.
- Step 3: We should also make sure to evaluate the system’s input data quality.

Conclusion

This machine learning project is for predicting median house values in Californian districts using the California Housing Prices dataset. The code performs the following tasks:

Loads the data and splits it into training and testing sets

Prepares the data by scaling numerical features and one-hot encoding categorical features

Trains and evaluates three regression models: a decision tree, a random forest, and a support vector machine (SVM)

Fine-tunes the hyperparameters of the best model (a random forest) using a grid search and a randomized search and evaluates the final model on the test set

Computes a 95% confidence interval for the test set error using both a t-distribution and a normal distribution

Finally, it creates a pipeline that can take in new data and output predicted median house values.

References

[2 End to End Machine Learning Project (sfbu.edu)]