Machine Learning in practice

ML in practice

- Look at the big picture.
- Get the data.
- Discover and visualize the data to gain insights.
- Prepare the data for Machine Learning algorithms.
- Select a model and train it.
- Fine-tune your model.
- Present your solution.
- Launch, monitor, and maintain your system.

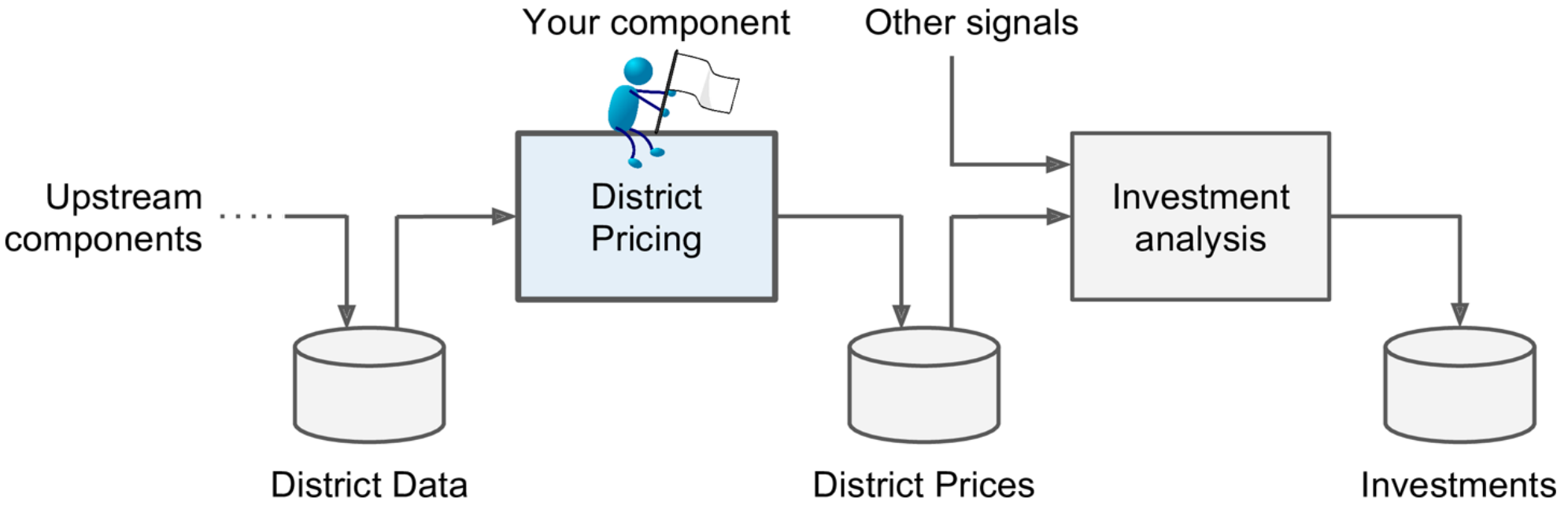
ML in practice:Real Data

- UC Irvine Machine Learning Repository
- Kaggle datasets
- Amazon’s AWS datasets
- Meta portals (they list open data repositories):
  - http://dataportals.org/
  - http://opendatamonitor.eu/
  - http://quandl.com/
- Other pages listing many popular open data repositories:
  - Wikipedia’s list of Machine Learning datasets
  - Quora.com question
  - Datasets subreddit

ML in practice:Big Picture

- Frame the problem
  - Building a model is probably not the end goal.
  - How does the company/organization expect to use and benefit from this model?
  - This is important because it will determine how you frame the problem.
    - what algorithms is selected?
    - what performance measure to evaluate the model?
    - how much effort is spent tweaking it?

ML in practice:Big Picture:ML Pipeline



- Pipelines are of common use in ML, since there is lots of data to manipulate and many data transformation to apply.
- Each component is self contained with only the data store as the interface.

ML in practice:Big Picture:Notation

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

→ Performance measure, others are also possible.

$$\mathbf{x}^{(1)} = \begin{pmatrix} -118.29 \\ 33.91 \\ 1,416 \\ 38,372 \end{pmatrix}$$

→ X sample with the first 2 denoting latitude and longitude respectively

$$y^{(1)} = 156,400$$

→ Y - sample or label.

$$\mathbf{X} = \begin{pmatrix} \left(\mathbf{x}^{(1)}\right)^T \\ \left(\mathbf{x}^{(2)}\right)^T \\ \vdots \\ \left(\mathbf{x}^{(1999)}\right)^T \\ \left(\mathbf{x}^{(2000)}\right)^T \end{pmatrix} = \begin{pmatrix} -118.29 & 33.91 & 1,416 & 38,372 \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

→ Matrix of samples

ML in practice:Big Picture:Check the Assumptions

- The district prices that your system outputs are going to be fed into a downstream Machine Learning system, and we assume that these prices are going to be used as such. But what if the downstream system actually converts the 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;
- your 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

ML in practice:Big Picture:Getting the data

```
from file_loader import fileloader

fileloader=fileloader("housing/housing.tgz")
fileloader.fetch_data()
housing=fileloader.load_data()
print(housing.head())
print()
print(housing.info())
print()
print(housing["ocean_proximity"].value_counts
print()
print(housing.describe())
print()
```

```
housing = load_housing_data()
housing.head()
```

	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
2	-122.24	37.85	52.0	1467.0	190.0	496.0	177.0	7.2574	352100.0	NEAR BAY
3	-122.25	37.85	52.0	1274.0	235.0	558.0	219.0	5.6431	341300.0	NEAR BAY
4	-122.25	37.85	52.0	1627.0	280.0	565.0	259.0	3.8462	342200.0	NEAR BAY

```
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
```

- missing data has to be handled.

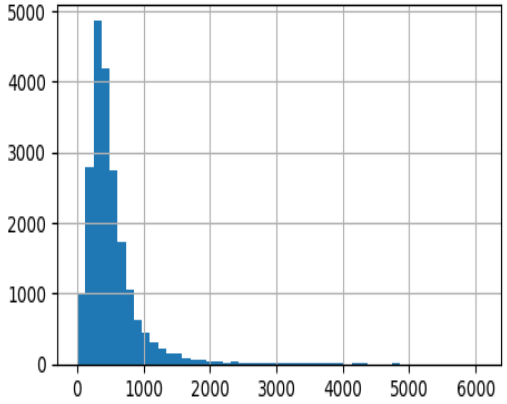
```
housing["ocean_proximity"].value_counts()
```

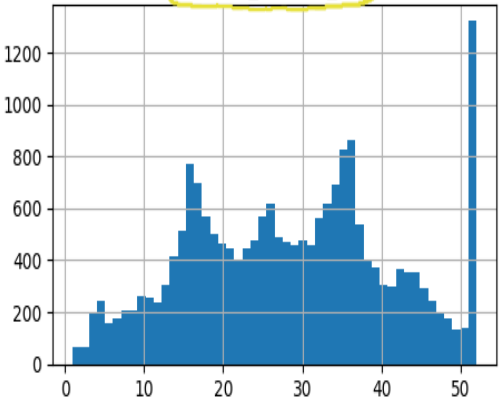
```
<1H OCEAN     9136
INLAND        6551
NEAR OCEAN    2658
NEAR BAY      2290
ISLAND           5
Name: ocean_proximity, dtype: int64
```

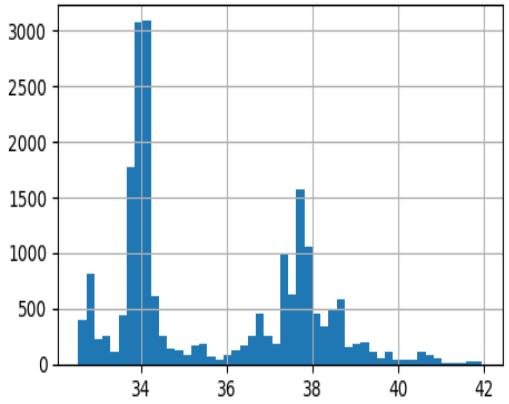
```
print(housing.describe())
```

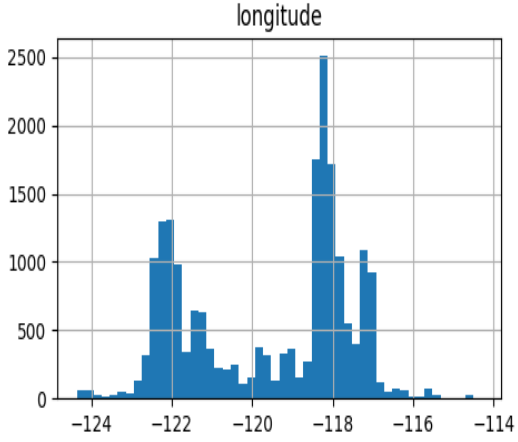
```
          longitude      latitude  housing_median_age   total_rooms  \
count  20640.000000  20640.000000        20640.000000  20640.000000
mean    -119.569704     35.631861           28.639486   2635.763081
std        2.003532      2.135952           12.585558   2181.615252
min     -124.350000  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

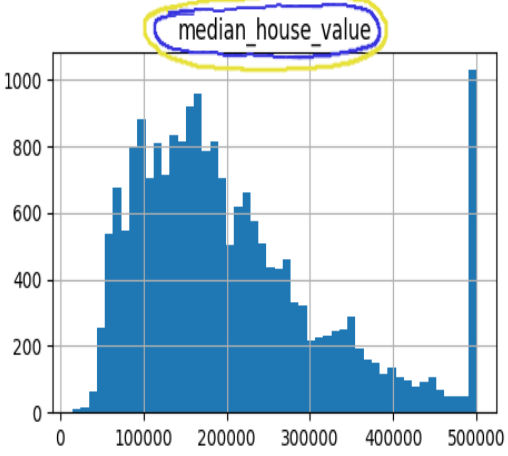
ML in practice:Big Picture:**Visualize**

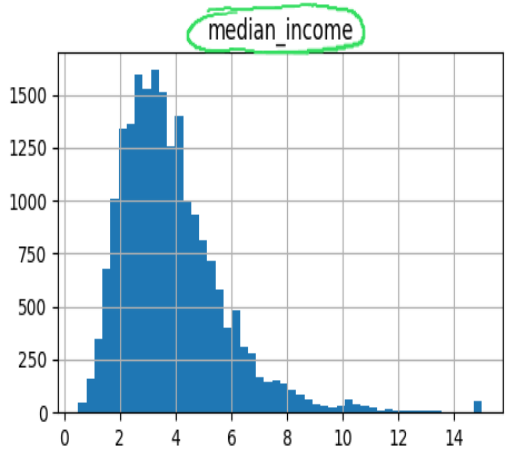


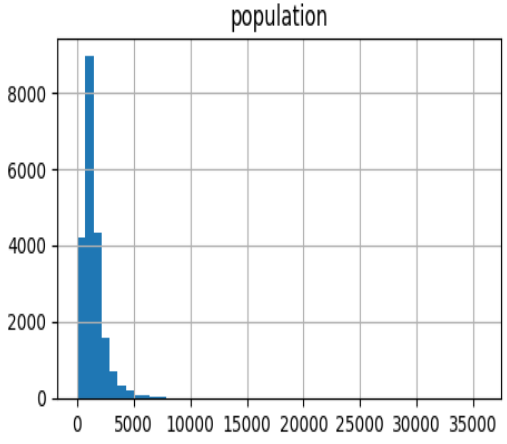


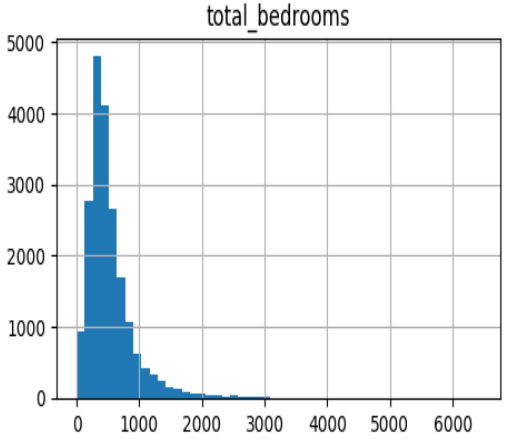


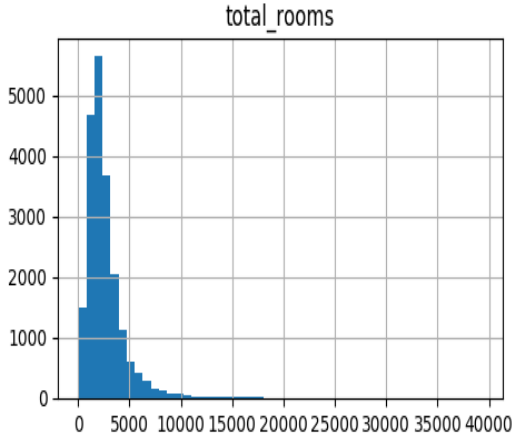












- >800 districts have value over 5,00,000.
- median income capped at 0.5 and 15 respectively.
- house median age and median house value also capped.
- MHV is the label and hence a serious problem.
- all of them have different scales
- many are tail heavy. some ML algorithms have issues with that. Tranformation will be applied to get a bell shaped curve.

ML in practice:Big Picture:Test set

```python
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]

def split_train_test_seed(data, test_ratio):
    np.random.seed(42)
    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]


fileloader=fileloader("housing/housing.tgz")
fileloader.fetch_data()
housing=fileloader.load_data()

#both will break if we fetch an updated
train_set, test_set = split_train_test(housing, 0.2)
print(len(train_set), "train +", len(test_set), "test")
#print(housing["median_income"].value_counts() / len(housing))
```

- Setting some data aside for testing.
- Will eventually look at the whole data while testing. One way around is to save the training set.
- This one will generate the same shuffled indexes all the time.

ML in practice:Big Picture:Test set:hash

```python
def test_set_check(identifier, test_ratio, hash):
    return hash(np.int64(identifier)).digest()[-1] < 256 * test_ratio

def split_train_test_by_id(data, test_ratio, id_column, hash=hashlib.md5):
    ids = data[id_column]
    in_test_set = ids.apply(lambda id_: test_set_check(id_, test_ratio, hash))
    return data.loc[~in_test_set], data.loc[in_test_set]



fileloader=fileloader("housing/housing.tgz")
fileloader.fetch_data()
housing=fileloader.load_data()

housing_with_id = housing.reset_index() # adds an `index` column
train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "index")
print(len(train_set), "train +", len(test_set), "test")

housing_with_id["id"] = housing["longitude"] * 1000 + housing["latitude"]
train_set, test_set = split_train_test_by_id(housing_with_id, 0.2, "id")
print(len(train_set), "train +", len(test_set), "test")
```

- Assumes unique and immutable identifier.
- Compute the hash and check for the last byte's value is < 51 (≈ 20% of 256)
- if ID not present, use row index as ID.

ML in practice:Big Picture:Test set:hash(scikit)

```
from file_loader import fileloader
from sklearn.model_selection import train_test_split

fileloader=fileloader("housing/housing.tgz")
fileloader.fetch_data()
housing=fileloader.load_data()

train_set, test_set = train_test_split(housing, test_size=0.2, random_state=42)
print(len(train_set), "train +", len(test_set), "test")
```

ML in practice:Big Picture:Test set:Stratified hash(scikit)

```
print("housing[median_income].head():",housing["median_income"].head())
housing["income_cat"] = np.ceil(housing["median_income"] / 1.5)
housing["income_cat"].where(housing["income_cat"] < 5, 5.0, inplace=True)
print("housing[income_cat].head():",housing["income_cat"].head())


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]
print("housing[income_cat].value_counts() / len(housing):\n",housing["income_cat"].value_counts() /
print("strat_train_set[income_cat].value_counts() / len(strat_train_set):",strat_train_set["income_c
print("strat_test_set[income_cat].value_counts() / len(strat_test_set):",strat_test_set["income_cat"


def income_cat_proportions(data):
    return data["income_cat"].value_counts() / len(data)

train_set, test_set = train_test_split(housing, test_size=0.2, random_state=42)

compare_props = pd.DataFrame({
    "Overall": income_cat_proportions(housing),
    "Stratified": income_cat_proportions(strat_test_set),
    "Random": income_cat_proportions(test_set),
}).sort_index()
compare_props["Rand. %error"] = 100 * compare_props["Random"] / compare_props["Overall"] - 100
compare_props["Strat. %error"] = 100 * compare_props["Stratified"] / compare_props["Overall"] - 100
print(compare_props)
```

```
housing[median_income].head(): 0    8.3252
1    8.3014
2    7.2574
3    5.6431
4    3.8462
Name: median_income, dtype: float64
housing[income_cat].head(): 0    5.0
1    5.0
2    5.0
3    4.0
4    3.0
Name: income_cat, dtype: float64
housing[income_cat].value_counts() / len(housing):
 3.0    0.350581
2.0    0.318847
4.0    0.176308
5.0    0.114438
1.0    0.039826
Name: income_cat, dtype: float64
strat_train_set[income_cat].value_counts() / len(strat_train_set):
2.0    0.318859
4.0    0.176296
5.0    0.114402
1.0    0.039850
Name: income_cat, dtype: float64
strat_test_set[income_cat].value_counts() / len(strat_test_set): 3
2.0    0.318798
4.0    0.176357
5.0    0.114583
1.0    0.039729
Name: income_cat, dtype: float64
      Overall    Random  Stratified  Rand. %error  Strat. %error
1.0  0.039826  0.040213    0.039729      0.973236      -0.243309
2.0  0.318847  0.324370    0.318798      1.732260      -0.015195
3.0  0.350581  0.358527    0.350533      2.266446      -0.013820
4.0  0.176308  0.167393    0.176357     -5.056334       0.027480
5.0  0.114438  0.109496    0.114583     -4.318374       0.127011
```

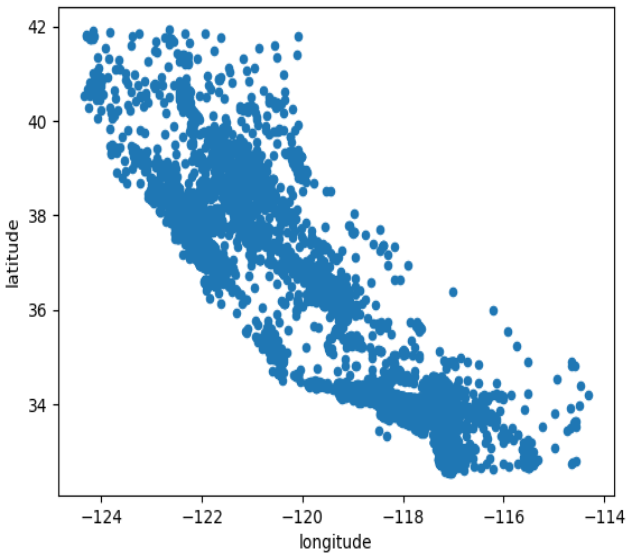
- limit the number of strata
- everything over 5 goes to 5

- Stratified sample is representative of the overall.

ML in practice:Big Picture:**Visualize**

```
#make a copy so we dont harm the original
housing = strat_train_set.copy()
housing.plot(kind="scatter", x="longitude", y="latitude")

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

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

california_img=fileloader.read_image("housing/california.png")
ax = housing.plot(kind="scatter", x="longitude", y="latitude", figsize=(10,7),
                        s=housing['population']/100, label="Population",
                        c="median_house_value", cmap=plt.get_cmap("jet"),
                        colorbar=False, alpha=0.4,
                       )
plt.imshow(california_img, extent=[-124.55, -113.80, 32.45, 42.05], alpha=0.5)
plt.ylabel("Latitude", fontsize=14)
plt.xlabel("Longitude", fontsize=14)
```

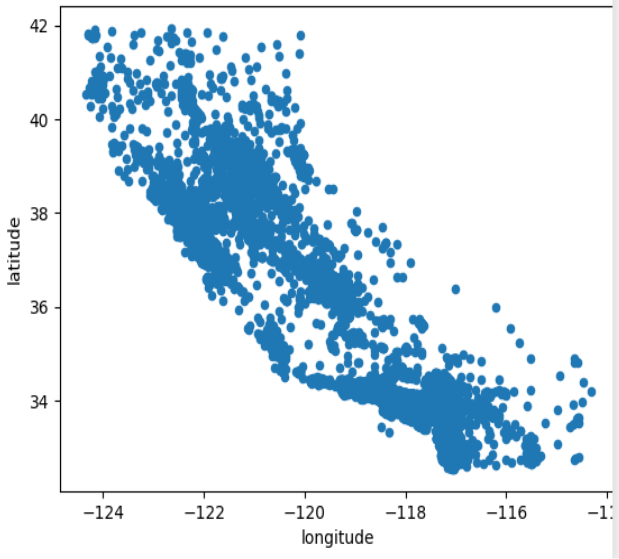
- plot for California, other than that difficult to figure out anything else

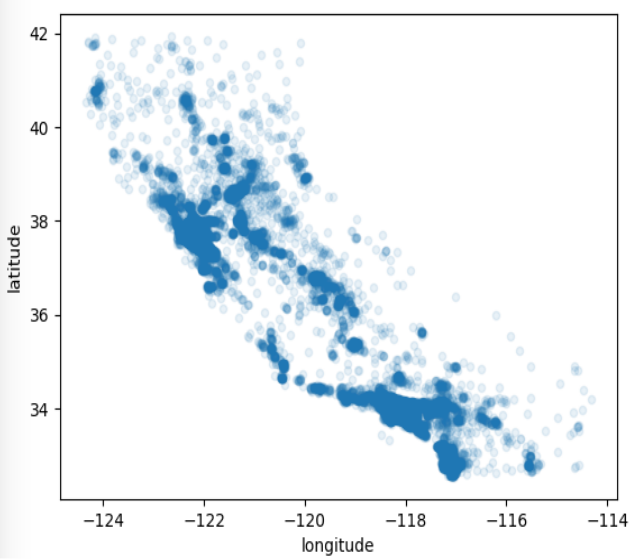


ML in practice:Big Picture:**Visualize**

```
#make a copy so we dont harm the original
housing = strat_train_set.copy()
housing.plot(kind="scatter", x="longitude", y="latitude")

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

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

california_img=fileloader.read_image("housing/california.png")
ax = housing.plot(kind="scatter", x="longitude", y="latitude", figsize=(10,7),
                       s=housing['population']/100, label="Population",
                       c="median_house_value", cmap=plt.get_cmap("jet"),
                       colorbar=False, alpha=0.4,
                      )
plt.imshow(california_img, extent=[-124.55, -113.80, 32.45, 42.05], alpha=0.5)
plt.ylabel("Latitude", fontsize=14)
plt.xlabel("Longitude", fontsize=14)
```

- Setting alpha to 0.1 makes it easier to visualize places with high density of data points.



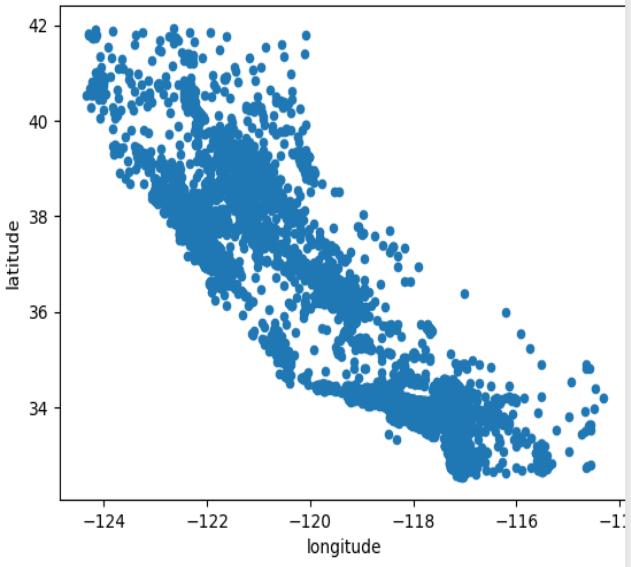


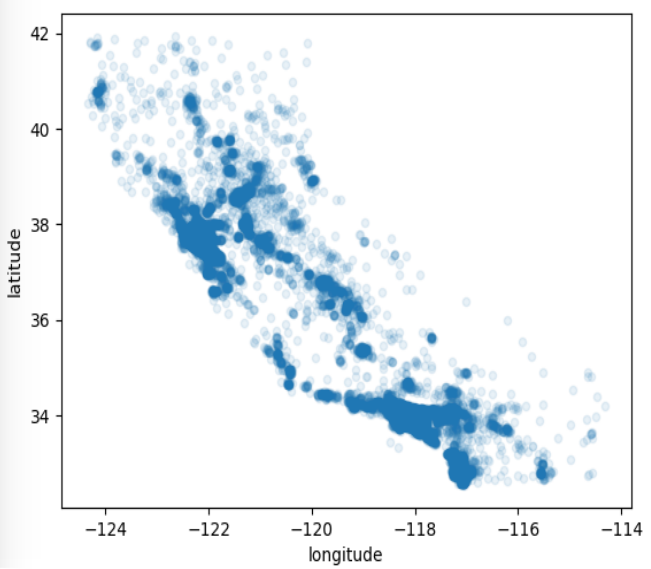
ML in practice:Big Picture:**Visualize**

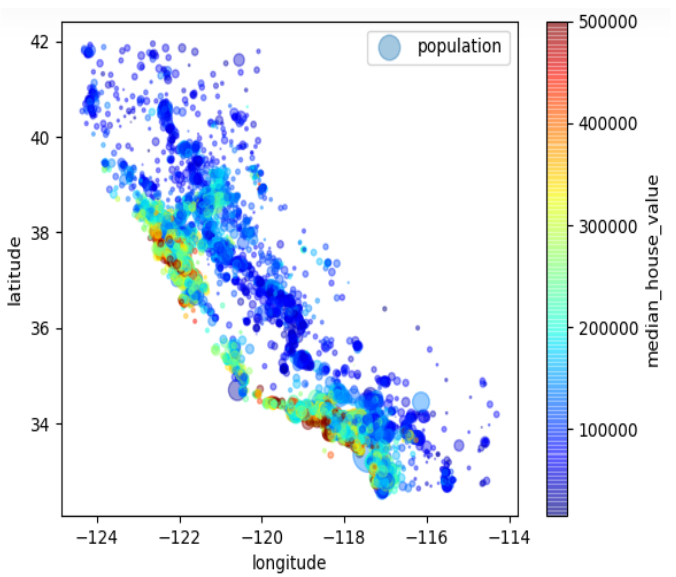
```
#make a copy so we dont harm the original
housing = strat_train_set.copy()
housing.plot(kind="scatter", x="longitude", y="latitude")

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

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

- the radius of each circle represents the population.
- the colour represents the price

```
california_img=fileloader.read_image("housing/california.png")
ax = housing.plot(kind="scatter", x="longitude", y="latitude", figsize=(10,7),
                        s=housing['population']/100, label="Population",
                        c="median_house_value", cmap=plt.get_cmap("jet"),
                        colorbar=False, alpha=0.4,
                       )
plt.imshow(california_img, extent=[-124.55, -113.80, 32.45, 42.05], alpha=0.5)
plt.ylabel("Latitude", fontsize=14)
plt.xlabel("Longitude", fontsize=14)
```



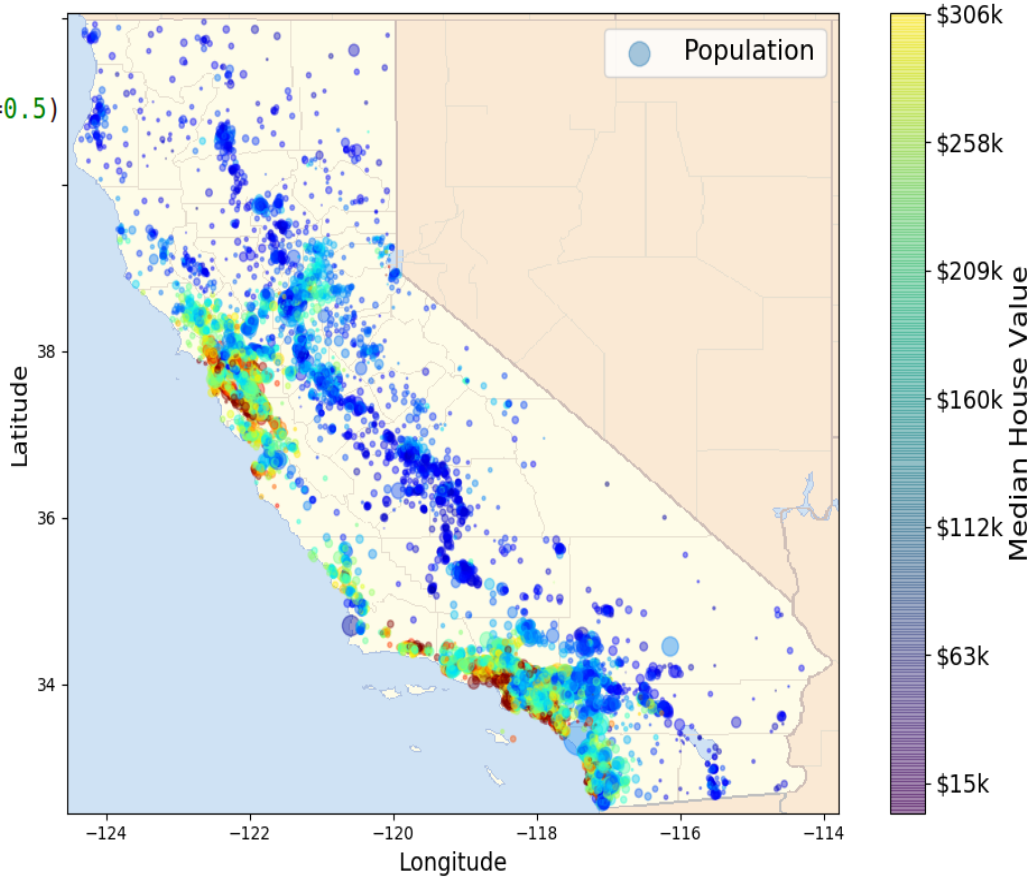




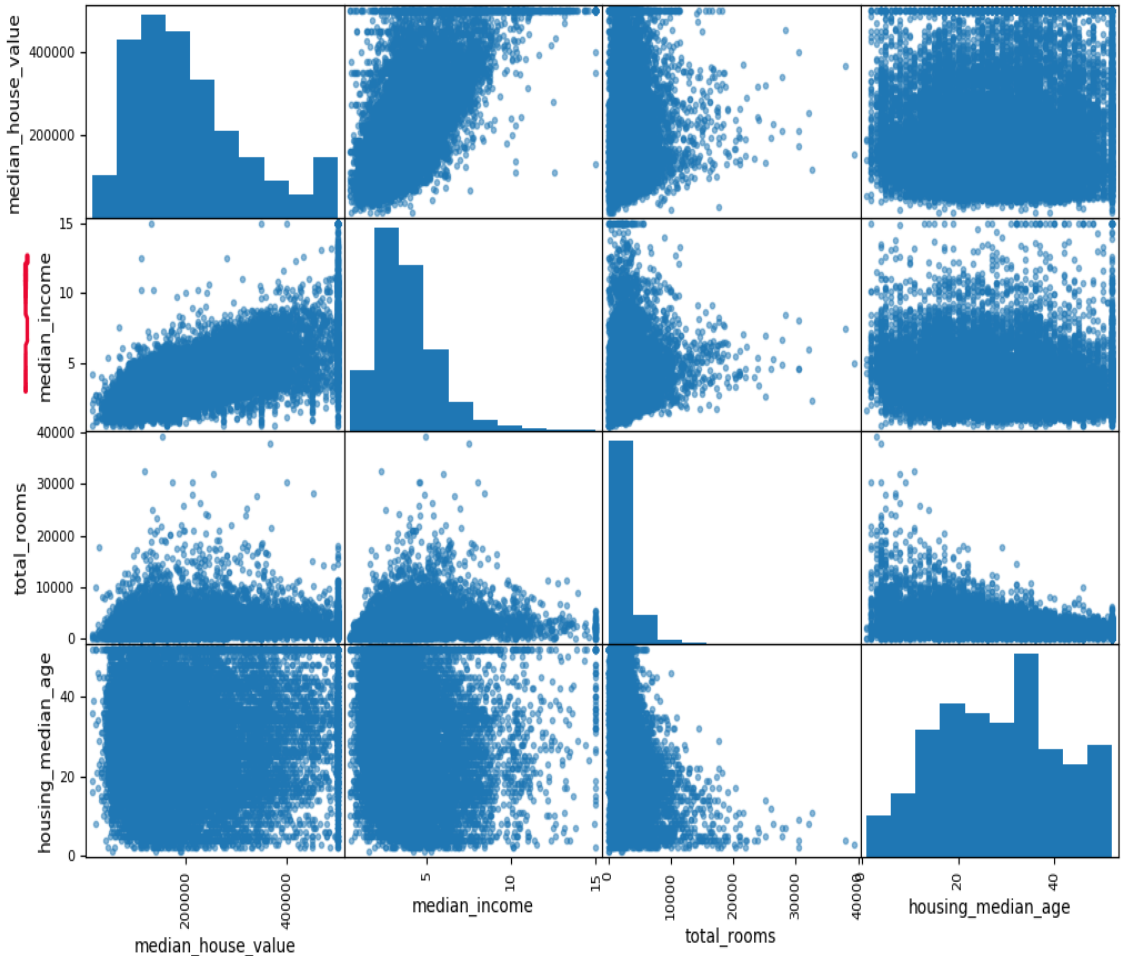
ML in practice:Big Picture:**Visualize**

```
#make a copy so we dont harm the original
housing = strat_train_set.copy()
housing.plot(kind="scatter", x="longitude", y="latitude")

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

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

california_img=fileloader.read_image("housing/california.png")
ax = housing.plot(kind="scatter", x="longitude", y="latitude", figsize=(10,7),
                       s=housing['population']/100, label="Population",
                       c="median_house_value", cmap=plt.get_cmap("jet"),
                       colorbar=False, alpha=0.4,
                      )
plt.imshow(california_img, extent=[-124.55, -113.80, 32.45, 42.05], alpha=0.5)
plt.ylabel("Latitude", fontsize=14)
plt.xlabel("Longitude", fontsize=14)
```

- superimposing it with the location map



ML in practice:Big Picture:Visualize:Correlation



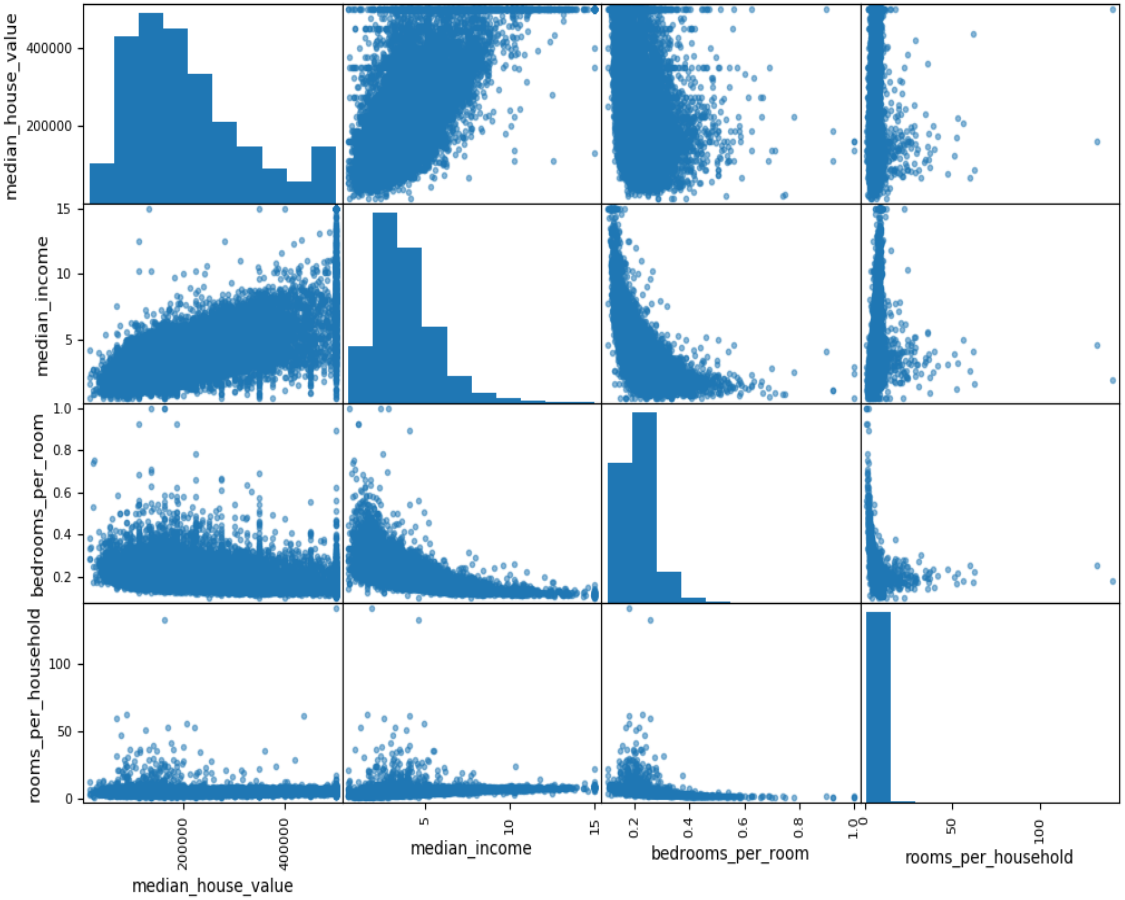
- histogram along the main diagonal.
- scatter plot of some important attributes against each other.
- This correlation is vital for attribute inclusion and exclusion.

```
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
```

range from -1 to +1

```
housing=fileloader.load_data()

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

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

ML in practice:Big Picture:Visualize:Correlation:attribute combination



- bedrooms/rooms is much more correllated than total number of rooms.
- rooms/household is also quite correllated.

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

```
#attribute combinations
housing["bedrooms_per_room"] = housing["total_bedrooms"]/housing["total_rooms"]
housing["rooms_per_household"] = housing["total_rooms"]/housing["households"]
housing["population_per_household"]=housing["population"]/housing["households"]
attributes = ["median_house_value", "median_income", "bedrooms_per_room","rooms_per_household"
scatter_matrix(housing[attributes], figsize=(12, 8))
plt.show()
```

ML in practice:Big Picture:Data Cleaning

```
fileloader=fileloader("housing/housing.tgz")
fileloader.fetch_data()
housing=fileloader.load_data()


housing["income_cat"] = np.ceil(housing["median_income"] / 1.5)

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]

#makes a copy and leaves strat_train_set untouched
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()

#this will have a different order in sampled set. because it is hash+strata sampled
housing_copy = housing.copy().iloc[21:24]
print("housing_copy:",housing_copy)
housing_copy=housing_copy.dropna(subset=["total_bedrooms"])    # option 1
print("housing_copy:option1:",housing_copy)
housing_copy = housing.copy().iloc[21:24]
housing_copy=housing_copy.drop("total_bedrooms", axis=1)   # option 2
print("housing_copy:option2:",housing_copy)
housing_copy = housing.copy().iloc[21:24]
median = housing_copy["total_bedrooms"].median()
housing_copy["total_bedrooms"].fillna(median, inplace=True) # option 3
print("housing_copy:option3:",housing_copy)
```

1. get rid of corresponding districts
2. get rid of the attribute
3. Set the missing value to some value (0, mean, median)

If it is the median then it needs to saved, it needs to be replaced in the test-set as well.

ML in practice:Big Picture:Data Cleaning(scikit)

```
fileloader=fileloader("housing/housing.tgz")
fileloader.fetch_data()
housing=fileloader.load_data()


housing["income_cat"] = np.ceil(housing["median_income"] / 1.5)

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]

#makes a copy and leaves strat_train_set untouched
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()

imputer = Imputer(strategy='median')
#imputer only works with numerical attributes
housing_num = housing.drop("ocean_proximity", axis=1)
#Only the total_bedrooms attribute had missing
#values, but we cannot be sure that there won't be any missing values in new data
imputer.fit(housing_num)
print(imputer.statistics_)
print(housing_num.median().values)
#X is plain numpy array, fit it back into Data Frame
X = imputer.transform(housing_num)
housing_tr = pd.DataFrame(X, columns=housing_num.columns)
print(housing_tr.iloc[21:24])
```

. Imputer from scikit-does exactly that.

• But it only works on numerical data so text attribute "Ocean Proximity" has to taken out in a copy.

```
imputer.statistics_:         [ -118.51    34.26    29.    2122.     434.    1166.     408.    3.5409    3.   ]
housing_num.median().values: [ -118.51    34.26    29.    2122.     434.    1166.     408.    3.5409    3.   ]
housing_tr.iloc[21:24]:     longitude  latitude  housing_median_age  total_rooms  total_bedrooms  \
21    -122.02     37.30                32.0       2134.0           328.0
22    -120.66     35.49                17.0       4422.0           945.0
23    -117.25     34.16                37.0       1709.0           278.0

    population  households  median_income  income_cat
21       903.0       322.0         6.3590         5.0
22      2307.0       885.0         2.8285         2.0
23       744.0       274.0         3.7188         3.0
```

Imputer computes median for all attributes. Should be applied on attributes, just in case.

ML in practice:scikit design

- Scikit-Learn’s API is remarkably well designed. The main design principles are:
  - Consistency. All objects share a consistent and simple interface:
  - Estimators.
    - Any object that can estimate some parameters based on a dataset is called an estimator (e.g., an imputer is an estimator).
    - The estimation itself is performed by the **fit()** method, and it takes only a dataset as a parameter (or two for supervised learning algorithms; the second dataset contains the labels).
    - Any other parameter needed to guide the estimation process is considered a hyperparameter (such as an imputer’s strategy), and it must be set as an instance variable (generally via a constructor parameter).

ML in practice:scikit design

- Scikit-Learn’s API is remarkably well designed. The main design principles are:
  - Transformers.
    - Some estimators (such as an imputer) can also transform a dataset; these are called transformers. Once again, the API is quite simple: the transformation is performed by the **transform()** method with the dataset to transform as a parameter. It returns the transformed dataset.
    - This transformation generally relies on the learned parameters, as is the case for an imputer.
    - All transformers also have a convenience method called **fit_transform()** that is equivalent to calling fit() and then transform() (but sometimes fit_transform() is optimized and runs much faster).

ML in practice:scikit design

- Scikit-Learn’s API is remarkably well designed. The main design principles are:
  - Predictors.
    - Finally, some estimators are capable of making predictions given a dataset; they are called predictors.
    - For example, the LinearRegression model is a predictor: it predicted life satisfaction given a country’s GDP per capita.
    - A predictor has a **predict()** method that takes a dataset of new instances and returns a dataset of corresponding predictions. It also has a **score()** method that measures the quality of the predictions given a test set (and the corresponding labels in the case of supervised learning algorithms).

ML in practice:scikit design

- Scikit-Learn’s API is remarkably well designed. The main design principles are:
  - Inspection. All the estimator’s hyperparameters are accessible directly via public instance variables (**e.g., imputer.strategy**), and all the estimator’s learned parameters are also accessible via public instance variables with an underscore suffix (e.g., **imputer.statistics_**).
  - Nonproliferation of classes. Datasets are represented as NumPy arrays or SciPy sparse matrices, instead of homemade classes. Hyperparameters are just regular Python strings or numbers.

ML in practice:scikit design

- Scikit-Learn’s API is remarkably well designed. The main design principles are:
  - Composition. Existing building blocks are reused as much as possible. For example, it is easy to create a Pipeline estimator from an arbitrary sequence of transformers followed by a final estimator.
  - Sensible defaults. Scikit-Learn provides reasonable default values for most parameters, making it easy to create a baseline working system quickly.

ML in practice:Big Picture:Handling Text

```
fileloader=fileloader("housing/housing.tgz")
fileloader.fetch_data()
housing=fileloader.load_data()


housing["income_cat"] = np.ceil(housing["median_income"] / 1.5)

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]

#makes a copy and leaves strat_train_set untouched
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()

#One issue with this representation is that ML algorithms will assume that two
#values are more similar than two distant values
encoder = LabelEncoder()
housing_cat = housing["ocean_proximity"]
housing_cat_encoded = encoder.fit_transform(housing_cat)
print("Labelled encoding class:",encoder.classes_,"\n")
print("Labelled encoding:",housing_cat_encoded,"\n")

#one hot encoding
encoder = OneHotEncoder()
housing_cat_1hot = encoder.fit_transform(housing_cat_encoded.reshape(-1,1))
#Notice that the output is a SciPy sparse matrix, instead of a NumPy array. This is very
#useful when you have categorical attributes with thousands of categories
print("One hot encoding:\n",housing_cat_1hot,"\n")
print("One hot encoding:\n",housing_cat_1hot.toarray(),"\n")
```

```
Labelled encoding class: ['<1H OCEAN' 'INLAND' 'ISLAND' 'NEAR BAY' 'NEAR OCEAN']

Labelled encoding: [0 3 4 ..., 1 0 3]

One hot encoding:
   (0, 0)       1.0
  (1, 3)        1.0
  (16502, 4)    1.0
  (16503, 0)    1.0
  (16504, 1)    1.0
  (16505, 1)    1.0
  (16506, 0)    1.0
  (16507, 1)    1.0
  (16508, 1)    1.0
  (16509, 1)    1.0
  (16510, 0)    1.0
  (16511, 3)    1.0

One hot encoding:
 [[ 1.  0.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  1.]
 ...,
 [ 0.  1.  0.  0.  0.]
 [ 1.  0.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.]]
```

The only problem here is ML algorithms assume 2 nearby values are similar, which may not be the case with categories.

We get a binary encoding. And it returns a Scipy sparse matrix instead of numpy array. This saves a lot of space for large number of categories.

ML in practice:Big Picture:Handling Text

```python
import numpy as np
from file_loader import fileloader
from sklearn.model_selection import StratifiedShuffleSplit
from sklearn.model_selection import train_test_split
import pandas as pd
from sklearn.preprocessing import Imputer
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import LabelBinarizer

fileloader=fileloader("housing/housing.tgz")
fileloader.fetch_data()
housing=fileloader.load_data()


housing["income_cat"] = np.ceil(housing["median_income"] / 1.5)

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]

#makes a copy and leaves strat_train_set untouched
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()

#text to integer and integer to one hot, in one shot
housing_cat = housing["ocean_proximity"]
encoder = LabelBinarizer()
housing_cat_1hot = encoder.fit_transform(housing_cat)
#returns a dense numpy array You can get a sparse matrix
#instead by passing sparse_output=True to the LabelBinarizer constructor
print("Encoder classes:\n",encoder.classes_,"\n")
print("One hot encoding:\n",housing_cat_1hot,"\n")
```

- LabelBinarizer composes the LabelEncoder and OneHotEncoder

```
Encoder classes:
 ['<1H OCEAN' 'INLAND' 'ISLAND' 'NEAR BAY' 'NEAR OCEAN']

One hot encoding:
 [[1 0 0 0 0]
 [0 0 0 1 0]
 [0 0 0 0 1]
 ...,
 [0 1 0 0 0]
 [1 0 0 0 0]
 [0 0 0 1 0]]
```

ML in practice:Big Picture:Handling Text

```
class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
    def __init__(self, add_bedrooms_per_room = True): # no *args or **kargs
        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[:, household_ix]
        population_per_household = X[:, population_ix] / X[:, household_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]

housing=fileloader.load_data()


housing["income_cat"] = np.ceil(housing["median_income"] / 1.5)

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]

#makes a copy and leaves strat_train_set untouched
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()
attr_adder = CombinedAttributesAdder()
housing_extra_attribs = attr_adder.transform(housing.values)

housing_extra_attribs = pd.DataFrame(housing_extra_attribs, columns=list(housing.columns)
    +["rooms_per_household", "population_per_household","bedrooms_per_room"])
print(housing_extra_attribs.head())
```

- Custom transformers can be used for tasks such as custom clean up, or combining specific attributes
- They need to confirm to an interface to work with Scikit

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population
0	-121.89	37.29	38	1568	351	710
1	-122.46	37.79	52	899	96	304
2	-117.2	32.77	31	1952	471	936
3	-119.61	36.31	25	1847	371	1460
4	-118.59	34.23	17	6592	1525	4459

	households	median_income	ocean_proximity	income_cat	rooms_per_household
0	339	2.7042	<1H OCEAN	2	4.62537
1	110	14.2959	NEAR BAY	10	8.17273
2	462	2.8621	NEAR OCEAN	2	4.22511
3	353	1.8839	INLAND	2	5.23229
4	1463	3.0347	<1H OCEAN	3	4.50581

	population_per_household	bedrooms_per_room
0	2.0944	0.223852
1	2.76364	0.106785
2	2.02597	0.241291
3	4.13598	0.200866
4	3.04785	0.231341

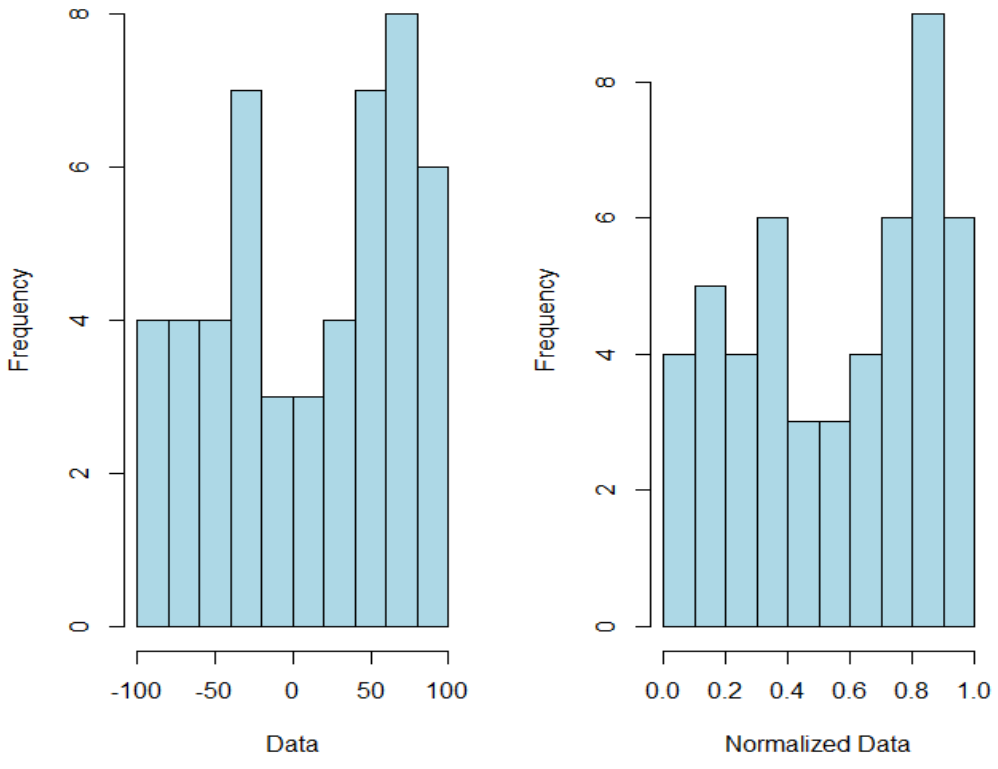
- In this case it is combining attributes

ML in practice:Big Picture:Feature Scaling

MinMaxScaler

- also called normalization

$$z_i = \frac{x_i - \min(x)}{\max(x) - \min(x)}$$



ML in practice:Big Picture:Feature Scaling

$$x_{new} = \frac{x - \mu}{\sigma}$$

- Standardization or mean normalization is quite different.
- It does not bind values to a specific range. This may be a problem for some ML algorithms
- However it is less effected by outliers

ML in practice:Big Picture:Pipeline

```
housing=fileloader.load_data()


housing["income_cat"] = np.ceil(housing["median_income"] / 1.5)

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]

#makes a copy and leaves strat_train_set untouched
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()

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

num_pipeline = Pipeline([
        ('imputer', Imputer(strategy="median")),
        ('attribs_adder', CombinedAttributesAdder()),
        #z score scaller, others like min-max are also available
        ('std_scaler', StandardScaler()),
    ])

housing_num=num_pipeline.fit_transform(housing_num)
print("num_pipeline:",num_pipeline)
print("housing_num:",housing_num)
```

```
num_pipeline: Pipeline(steps=[('imputer', Imputer(axis=0, copy=True, missing_values='NaN', strategy='median', verbose=0)), ('attribs_adder', CombinedAttributesAdder
(add_bedrooms_per_room=True)), ('std_scaler', StandardScaler(copy=True, with_mean=True, with_std=True))])
housing_num: [
 [-1.15788621  0.77388697  0.74440696 ..., -0.31200699 -0.08649926   0.1663318 ]
 [-1.44259512  1.0077626   1.85708974 ...,  1.04438561 -0.02872844  -1.72035025]
 [ 1.18471864 -1.3403487   0.18806557 ..., -0.46505337 -0.09240558   0.44738297]
 ...,
 [ 1.58431009 -0.72291704 -1.56043594 ...,  0.34606045 -0.0305544  -0.5467078 ]
 [ 0.7801323  -0.84920988  0.18806557 ...,  0.02457121  0.06150939  -0.31674597]
 [-1.43760023  0.99840757  1.85708974 ..., -0.22859871 -0.09586355   0.10997909]]
```

- This class by Scikit takes a list of names/estimators defining a sequence of steps (as usually required in an ML operations.
- All estimators must be transformers (i.e. they must have "fit_transform()" function) except the last one.

- This cascades the call to all transformers in the pipeline.
- The pipeline exposes the same function as the final estimator.

ML in practice:Big Picture:Pipeline

```
#makes a copy and leaves strat_train_set untouched
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()

#remove text attr
housing_num = housing.drop("ocean_proximity", axis=1)
num_attribs = list(housing_num)
#to be added later after "binarizing"
cat_attribs = ["ocean_proximity"]

num_pipeline = Pipeline([
        ('selector', DataFrameSelector(num_attribs)),
        ('imputer', Imputer(strategy="median")),
        ('attribs_adder', CombinedAttributesAdder()),
        ('std_scaler', StandardScaler()),
    ])


cat_pipeline = Pipeline([
        ('selector', DataFrameSelector(cat_attribs)),
        ('label_binarizer', LabelBinarizer()),
    ])
#adding the "binarized value back"
preparation_pipeline = FeatureUnion(transformer_list=[
        ("num_pipeline", num_pipeline),
        ("cat_pipeline", cat_pipeline),
    ])

housing_prepared = preparation_pipeline.fit_transform(housing)
print("preparation_pipeline:",preparation_pipeline)
print("housing_prepared:",housing_prepared)
```

```
class DataFrameSelector(BaseEstimator, TransformerMixin):
    def __init__(self, attribute_names):
        self.attribute_names = attribute_names
    def fit(self, X, y=None):
        return self
    def transform(self, X):
        return X[self.attribute_names].values
```

- Feature Union waits for the results from 2 parallel pipeline and then concatenates the results.
- Custom transformer to add remove features.

```
preparation_pipeline: FeatureUnion(n_jobs=1,
       transformer_list=[('num_pipeline', Pipeline(steps=[('selector', DataFrameSelector(attribute_names=['longitude', 'latitude', 'housing_median_age',
       'total_rooms', 'total_bedrooms', 'population', 'households', 'median_income', 'income_cat'])), ('imputer', Imputer(axis=0, copy=True,
missing_values='NaN'...roximity'])), ('label_binarizer', LabelBinarizer(neg_label=0, pos_label=1, sparse_output=False))]))],
       transformer_weights=None)

housing_prepared: [[-1.15788621  0.77388697  0.74440696 ...,  0.          0.          0.        ]
 [-1.44259512  1.0077626   1.85708974 ...,  0.          1.          0.        ]
 [ 1.18471864 -1.3403487   0.18806557 ...,  0.          0.          1.        ]
 ...,
 [ 1.58431009 -0.72291704 -1.56043594 ...,  0.          0.          0.        ]
 [ 0.7801323  -0.84920988  0.18806557 ...,  0.          0.          0.        ]
 [-1.43760023  0.99840757  1.85708974 ...,  0.          1.          0.        ]]
```

ML in practice:Big Picture:Train and predict

```
#to be added later after "binarizing"
cat_attribs = ["ocean_proximity"]

num_pipeline = Pipeline([
        ('selector', DataFrameSelector(num_attribs)),
        ('imputer', Imputer(strategy="median")),
        ('attribs_adder', CombinedAttributesAdder()),
        ('std_scaler', StandardScaler()),
    ])
cat_pipeline = Pipeline([
        ('selector', DataFrameSelector(cat_attribs)),
        ('label_binarizer', LabelBinarizer()),
    ])
#adding the "binarized value back"
preparation_pipeline = FeatureUnion(transformer_list=[
        ("num_pipeline", num_pipeline),
        ("cat_pipeline", cat_pipeline),
    ])
#prepare data
housing_prepared = preparation_pipeline.fit_transform(housing)

#select Model
lin_reg = LinearRegression()

#train
lin_reg.fit(housing_prepared, housing_labels)
some_data = housing.iloc[:5]
some_labels = housing_labels.iloc[:5]
some_data_prepared = preparation_pipeline.transform(some_data)

#predict
print("Predictions:\t", lin_reg.predict(some_data_prepared))
print("Labels:\t\t", list(some_labels))

housing_predictions = lin_reg.predict(housing_prepared)
lin_mse = mean_squared_error(housing_labels, housing_predictions)
lin_rmse = np.sqrt(lin_mse)
print("lin rmse:\t",lin rmse)
```

```
Predictions:    [ 210470.97924915  661390.78423558  212482.55030341   56554.83062456  184931.82695168]
Labels:         [ 286600.0,        500001.0,        196900.0,         46300.0,        254500.0]
lin_rmse:       68312.8688461
```

- Choose model and train
- Select and prepare some data for testing
- Predict.
- The prediction 3 not very accurate.
- There can be 2 reasons for this
  1. features not providing enough information
  2. Model not powerful enough
  3. Overfitting - (This model is not regularized so this option is ruled out.

ML in practice:Big Picture:Train and predict

```
#makes a copy and leaves strat_train_set untouched
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()

#remove text attr
housing_num = housing.drop("ocean_proximity", axis=1)
num_attribs = list(housing_num)
#to be added later after "binarizing"
cat_attribs = ["ocean_proximity"]

num_pipeline = Pipeline([
        ('selector', DataFrameSelector(num_attribs)),
        ('imputer', Imputer(strategy="median")),
        ('attribs_adder', CombinedAttributesAdder()),
        ('std_scaler', StandardScaler()),
    ])


cat_pipeline = Pipeline([
        ('selector', DataFrameSelector(cat_attribs)),
        ('label_binarizer', LabelBinarizer()),
    ])
#adding the "binarized value back"
preparation_pipeline = FeatureUnion(transformer_list=[
        ("num_pipeline", num_pipeline),
        ("cat_pipeline", cat_pipeline),
    ])

housing_prepared = preparation_pipeline.fit_transform(housing)

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)
print("tree_rmse:",tree_rmse)
```

```
housing.tgz
tree_rmse: 0.0
```

- change the model to Decision tree Regressor.
- No errors, but more likely the model is severly overfitting the data.
- Test set is not to be used now, test it using cross validation.

ML in practice:Big Picture:Train and predict:Cross Validation

```
num_pipeline = Pipeline([
        ('selector', DataFrameSelector(num_attribs)),
        ('imputer', Imputer(strategy="median")),
        ('attribs_adder', CombinedAttributesAdder()),
        ('std_scaler', StandardScaler()),
    ])


cat_pipeline = Pipeline([
        ('selector', DataFrameSelector(cat_attribs)),
        ('label_binarizer', LabelBinarizer()),
    ])
#adding the "binarized value back"
preparation_pipeline = FeatureUnion(transformer_list=[
        ("num_pipeline", num_pipeline),
        ("cat_pipeline", cat_pipeline),
    ])

housing_prepared = preparation_pipeline.fit_transform(housing)

tree_reg = DecisionTreeRegressor()
tree_scores = cross_val_score(tree_reg, housing_prepared, housing_labels,
                              scoring="neg_mean_squared_error", cv=10)
tree_rmse_scores = np.sqrt(-tree_scores)

def display_scores(scores):
    print("Scores:", scores)
    print("Mean:", scores.mean())
    print("Standard deviation:", scores.std())

display_scores(tree_rmse_scores)
```

- It randomly splits the training into 10 subsets called folds.
- It then trains and evaluate the decision tree model 10 times picking a different set to evaluate each time.
- The error goes up to 70669.

```
housing.tgz
Scores: [ 68502.38321384  68414.40925792  66382.00590591  70061.54015321
  69234.16286321  74019.15159651  69557.16377296  72869.3643456
  75811.15855228  72139.85589649]
Mean: 70699.1195559
Standard deviation: 2765.51857139
```

ML in practice:Big Picture:Train and predict:Cross Validation

```
num_pipeline = Pipeline([
        ('selector', DataFrameSelector(num_attribs)),
        ('imputer', Imputer(strategy="median")),
        ('attribs_adder', CombinedAttributesAdder()),
        ('std_scaler', StandardScaler()),
    ])


cat_pipeline = Pipeline([
        ('selector', DataFrameSelector(cat_attribs)),
        ('label_binarizer', LabelBinarizer()),
    ])
#adding the "binarized value back"
preparation_pipeline = FeatureUnion(transformer_list=[
        ("num_pipeline", num_pipeline),
        ("cat_pipeline", cat_pipeline),
    ])

housing_prepared = preparation_pipeline.fit_transform(housing)
lin_reg = LinearRegression()
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)
def display_scores(scores):
    print("Scores:", scores)
    print("Mean:", scores.mean())
    print("Standard deviation:", scores.std())

display_scores(lin_rmse_scores)
```

```
Scores: [ 66650.88765204  66113.66276732  67340.90680673  74448.79357433
  71411.93096204  69777.57757966  65363.33706061  67637.20198388
  69674.78358646  67373.40118857]
Mean:  68579.2483162
Standard deviation: 2629.19423962
```

- Cross validating linear regression

ML in practice:Big Picture:Train and predict:Cross Validation

```
num_pipeline = Pipeline([
        ('selector', DataFrameSelector(num_attribs)),
        ('imputer', Imputer(strategy="median")),
        ('attribs_adder', CombinedAttributesAdder()),
        ('std_scaler', StandardScaler()),
    ])


cat_pipeline = Pipeline([
        ('selector', DataFrameSelector(cat_attribs)),
        ('label_binarizer', LabelBinarizer()),
    ])
#adding the "binarized value back"
preparation_pipeline = FeatureUnion(transformer_list=[
        ("num_pipeline", num_pipeline),
        ("cat_pipeline", cat_pipeline),
    ])
housing_prepared = preparation_pipeline.fit_transform(housing)

forest_reg = RandomForestRegressor()
forest_reg.fit(housing_prepared, housing_labels)
housing_predictions = forest_reg.predict(housing_prepared)
forest_mse = mean_squared_error(housing_labels, housing_predictions)
forest_rmse = np.sqrt(forest_mse)
print("forest_rmse:",forest_rmse)
```

- Changed model to Random Forest Regressor

```
housing.tgz
22306.9237494
```

ML in practice:Big Picture:Train and predict:Cross Validation

```
num_pipeline = Pipeline([
        ('selector', DataFrameSelector(num_attribs)),
        ('imputer', Imputer(strategy="median")),
        ('attribs_adder', CombinedAttributesAdder()),
        ('std_scaler', StandardScaler()),
    ])


cat_pipeline = Pipeline([
        ('selector', DataFrameSelector(cat_attribs)),
        ('label_binarizer', LabelBinarizer()),
    ])
#adding the "binarized value back"
preparation_pipeline = FeatureUnion(transformer_list=[
        ("num_pipeline", num_pipeline),
        ("cat_pipeline", cat_pipeline),
    ])
housing_prepared = preparation_pipeline.fit_transform(housing)

forest_reg = RandomForestRegressor()
forest_scores = cross_val_score(forest_reg, housing_prepared, housing_labels,
                                scoring="neg_mean_squared_error", cv=10)
forest_rmse_scores = np.sqrt(-forest_scores)
def display_scores(scores):
    print("Scores:", scores)
    print("Mean:", scores.mean())
    print("Standard deviation:", scores.std())

display_scores(forest_rmse_scores)
```

```
Scores: [ 52450.30929591  50645.63652175  51359.64227392  54843.97088518
  53744.58345703  56741.17979855  51100.15014801  50987.44152127
  55425.79786729  53672.67451753]
Mean: 53097.1386286
Standard deviation: 2009.71312432
```

- Idea is to try out a few model and select the top 2-3 best performing model and fine tune them.
- This seems to be the best performing model so far but, it performs better on training set then on test set, which is an indication of overfitting.
- Before finetuning a few models should be tried out. SVM, NN.

ML in practice:Big Picture:Train and predict:Cross Validation

```
svm_rmse: 106592.721637
```

- SVM scores

```
Scores: [ 104927.26557934  107899.45549145  106654.29879924  109321.90215978
  106815.99176337  109519.57409202  104655.06074856  106450.97336327
  110285.0002465   109540.36930092]
Mean: 107606.989154
Standard deviation: 1906.79350128
```

ML in practice:Big Picture:Train and predict:Fine tune

```
housing_prepared = preparation_pipeline.fit_transform(housing)
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')
grid_search.fit(housing_prepared, housing_labels)
print("grid_search:",grid_search)
print("best params:",grid_search.best_params_)
print("best estimator:",grid_search.best_estimator_)
cvres = grid_search.cv_results_
for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
    print(np.sqrt(-mean_score), params)

feature_importances = grid_search.best_estimator_.feature_importances_
#print("feature_importance:",feature_importances)
extra_attribs = ["rooms_per_household", "population_per_household", "bedr
cat_one_hot_attribs = list(encoder.classes_)
attributes = num_attribs + extra_attribs + cat_one_hot_attribs
print("feature_importance:",sorted(zip(feature_importances, attributes),
#print("",pd.DataFrame(grid_search.cv_results_))
```

- Fine tuning often means searching the hyperparameter space for best values.
- score from CV earlier

```
Scores: [ 51503.2551048   49557.54503291  51807.43084548  53673.15840113
  54527.23033067  56890.92131457  51598.54256106  50967.33420488
  55389.10925591  53748.50014218]
Mean: 52966.3027194
Standard deviation: 2139.86821282

best params: {'max_features': 6, 'n_estimators': 30}
best estimator: RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
          max_features=6, max_leaf_nodes=None, min_impurity_split=1e-07,
          min_samples_leaf=1, min_samples_split=2,
          min_weight_fraction_leaf=0.0, n_estimators=30, n_jobs=1,
          oob_score=False, random_state=None, verbose=0, warm_start=False)
64187.1029193 {'max_features': 2, 'n_estimators': 3}
56080.912239 {'max_features': 2, 'n_estimators': 10}
53418.0530961 {'max_features': 2, 'n_estimators': 30}
60537.2574097 {'max_features': 4, 'n_estimators': 3}
54043.4856778 {'max_features': 4, 'n_estimators': 10}
51520.4051092 {'max_features': 4, 'n_estimators': 30}
60099.58749 {'max_features': 6, 'n_estimators': 3}
53436.3658709 {'max_features': 6, 'n_estimators': 10}
51133.3163999 {'max_features': 6, 'n_estimators': 30}
59914.6222925 {'max_features': 8, 'n_estimators': 3}
52801.0307249 {'max_features': 8, 'n_estimators': 10}
51222.6532761 {'max_features': 8, 'n_estimators': 30}
62588.1509939 {'bootstrap': False, 'max_features': 2, 'n_estimators': 3}
55395.2970768 {'bootstrap': False, 'max_features': 2, 'n_estimators': 10}
60580.3658574 {'bootstrap': False, 'max_features': 3, 'n_estimators': 3}
53601.6652214 {'bootstrap': False, 'max_features': 3, 'n_estimators': 10}
59284.2851113 {'bootstrap': False, 'max_features': 4, 'n_estimators': 3}
52850.5484029 {'bootstrap': False, 'max_features': 4, 'n_estimators': 10}
feature_importance: [(0.29137398596732467, 'median_income'), (0.16038554120969828, 'IN
(0.095637653493671815, 'income_cat'), (0.069041276469578541, 'longitude'), (0.06171286
(0.047212676690880806, 'rooms_per_household'), (0.041471519714720087, 'housing_median_
(total_rooms'), (0.017416441682265141, 'households'), (0.017295591933519244, 'total_be
OCEAN'), (0.0026628083119608278, 'NEAR BAY'), (5.0064433343251487e-05, 'ISLAND')]
```

- Run performance test for (4x3=12) combination of hyper parameters.
- Then try (2x3=6) more of these
- All together there would be (18x5) rounds of training
- These are algo specific param, but for these values, you get best results.

ML in practice:Big Picture:Train and predict:Fine tune

```
housing_prepared = preparation_pipeline.fit_transform(housing)
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')
grid_search.fit(housing_prepared, housing_labels)
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_transformed = preparation_pipeline.transform(X_test)
final_predictions = final_model.predict(X_test_transformed)

final_mse = mean_squared_error(y_test, final_predictions)
final_rmse = np.sqrt(final_mse)
print("final_rmse:",final_rmse)
```

```
final_rmse: 47877.7186985
```

- Find the best estimator
- Validate against the test set.

ML in practice:Big Picture:Train and predict:Fine tune

```
housing_prepared = preparation_pipeline.fit_transform(housing)
param_distribs = {
        'n_estimators': randint(low=1, high=200),
        'max_features': randint(low=1, high=8),
    }

#print(encoder.classes_)
forest_reg = RandomForestRegressor()
rnd_search = RandomizedSearchCV(forest_reg, param_distributions=param_distribs,
                                n_iter=10, cv=5, scoring='neg_mean_squared_error')
rnd_search.fit(housing_prepared, housing_labels)
print("random search:",rnd_search)
cvres = rnd_search.cv_results_
for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
    print(np.sqrt(-mean_score), params)

feature_importances = rnd_search.best_estimator_.feature_importances_
#print("feature_importances:",feature_importances)
extra_attribs = ["rooms_per_household", "population_per_household", "bedrooms_per_room"]
cat_one_hot_attribs = list(encoder.classes_)
attributes = num_attribs + extra_attribs + cat_one_hot_attribs
print("feature_importances:",sorted(zip(feature_importances, attributes), reverse=True))
```

```
random search: RandomizedSearchCV(cv=5, error_score='raise',
         estimator=RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
          max_features='auto', max_leaf_nodes=None,
          min_impurity_split=1e-07, min_samples_leaf=1,
          min_samples_split=2, min_weight_fraction_leaf=0.0,
          n_estimators=10, n_jobs=1, oob_score=False, random_state=None,
          verbose=0, warm_start=False),
         fit_params={}, iid=True, n_iter=10, n_jobs=1,
         param_distributions={'n_estimators': <scipy.stats._distn_infrastructure.rv_frozer
<scipy.stats._distn_infrastructure.rv_frozen object at 0x7f911228b630>},
         pre_dispatch='2*n_jobs', random_state=None, refit=True,
         return_train_score=True, scoring='neg_mean_squared_error',
         verbose=0)
50134.4214306 {'n_estimators': 192, 'max_features': 7}
50226.3758162 {'n_estimators': 189, 'max_features': 6}
51905.3546453 {'n_estimators': 39, 'max_features': 3}
50177.0078953 {'n_estimators': 137, 'max_features': 7}
50609.074143 {'n_estimators': 118, 'max_features': 4}
50235.4288957 {'n_estimators': 158, 'max_features': 6}
50903.2542536 {'n_estimators': 100, 'max_features': 4}
50952.1689078 {'n_estimators': 97, 'max_features': 4}
54875.9982901 {'n_estimators': 40, 'max_features': 1}
51273.8390168 {'n_estimators': 30, 'max_features': 5}
```

- It has done a better job than Grid Search.

- grid search is good when there are few combinations to try out.
- But when the hyper-param space is large, like it usually is, then Randomized search is used to search all possible combinations.
- 2 Benefits
  1. if you run 10 iterations it will explore 10 different values of each parameter.
  2. More control over compute budget.

ML in practice:Big Picture:Train and predict:Fine tune

- Ensemble Method
  - Another way to fine-tune your 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.