CAP4770 – Intro to Data Science

End-to-End Machine Learning Project – Part 2

Prof. Ye Xia

Prepare the Data for Machine Learning Algorithms

- You should write functions for this purpose: for reproducibility and reuse.
- We will revert to `strat_train_set` and separate the inputs and labels.

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

Side Note: `housing_labels` is a series not DataFrame, and it may have some limitation.

A series is not a single-column DataFrame.

Data Cleaning

- What to do with missing values:
  - Get rid of the corresponding districts.
  - Get rid of the whole attribute.
  - Set the missing values to some value (zero, the mean, the median, etc.).
- You can accomplish these easily using DataFrame's `dropna()`, `drop()`, and `fillna()` methods:

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

- If you choose option 3, you should compute the median value on the training set and use it to fill the missing values in the training set.

Save the median value that you have computed.

You will need it later to replace missing values in the test set, and also once the system goes live to replace missing values in new data.

Scikit-Learn SimpleImputer

- SimpleImputer is a class for fixing missing values.

```
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy="median")
```

- Since the median can only be computed on numerical attributes, you need to create a copy of the data without the text attribute `ocean_proximity`. Then, apply imputer to fill the missing values with the median.

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

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

  It hasn't filled the missing entries yet.

- For now, only the `total_bedrooms` attribute has missing values,

but we cannot be sure that there won't be any missing values in the other attributes in new data after the system goes live, so it is safer to apply the imputer to all the numerical attributes.

```
In [257]: imputer.statistics_
Out[257]: array([-118.51   ,   34.26   ,   29.     , 2119.     ,  433.     ,
                 1164.     ,  408.     ,    3.54155])

In [258]: housing_num.median().values
Out[258]: array([-118.51   ,   34.26   ,   29.     , 2119.     ,  433.     ,
                 1164.     ,  408.     ,    3.54155])
```

- Next, you can use this "trained" imputer to transform the training set by replacing missing values with the learned medians:

```
X = imputer.transform(housing_num)
```

The result is a plain NumPy array containing the transformed features.

If you want to put it back into a pandas DataFrame, do:

```
housing_tr = pd.DataFrame(X, columns=housing_num.columns,
                          index=housing_num.index)
```

Scikit-Learn API

- All objects within Scikit-Learn share a uniform common basic API consisting of three complementary interfaces (i.e., objects):
  - an **estimator** interface for building and fitting models,
  - a **predictor** interface for making predictions
  - a **transformer** interface for converting data.

Scikit-Learn API: Estimator

- The estimator interface is at the core of the library. It is the base object that implements a `fit()` method for learning a model from training data.

  All supervised and unsupervised learning algorithms (e.g., for classification, regression or clustering) are offered as objects implementing this interface. Example:

```
from sklearn.linear_model import LogisticRegression
clf = LogisticRegression(C=1e5)
clf.fit(X_train, y_train)
```

  `fit()` takes only one or two dataset as parameters. The latter is for supervised learning algorithms and the second dataset contains the labels.

  Any other parameter needed to guide the learning process is considered a hyperparameter (such as an SimpleImputer's

`strategy` or LogisticRegression's `C`), and it must be set as an instance variable (generally via a constructor parameter).

Machine learning tasks like feature extraction, feature selection or dimensionality reduction are also provided as estimators.

- All the estimator's hyperparameters are accessible directly via public instance variables (e.g., `imputer.strategy`), and all the estimator's learned parameters are accessible via public instance variables with an underscore suffix (e.g., `imputer.statistics_`).

Scikit-Learn API: Transformer

- Some estimators implement a transformer interface which defines a `transform()` method. It is used to modify or filter data before feeding it to a learning algorithm.

  `transform()` takes as input some dataset X and yields as output a transformed version of X.

  The transformation generally relies on the learned parameters (from `fit()`).

  All transformers also have a method called `fit_transform()` that is equivalent to calling `fit()` and then `transform()`. But, sometimes `fit_transform()` is optimized and runs much faster.

  Preprocessing, feature selection, feature extraction and dimensionality reduction algorithms are all provided as transformers within the Scikit-Learn library.

Example: SimpleImputer is an estimator and also a transformer. It learns/computes the median from the dataset with `fit()`. Then, it uses `transform()` to fill the entries whose data is missing with the median.

Scikit-Learn API: Predictor

- Some estimators are capable of making predictions and they are called predictors. A predictor has a `predict()` method that takes a dataset (usually new instances) and returns a dataset of corresponding predictions.
  For instance, in the above, LogisticRegression,

```
y_pred=clf.predict(X_test)
```

  A predictor must have a `score()` method that measures the quality of the predictions. In linear regression, for instance, this method takes as input `X_test` and `y_test` and typically computes the *coefficient of determination* based on `y_test` and `predict(X_test)`.

  Some predictors such as LogisticRegression has a `predict_proba()` method. For instance,

```
clf.predict_proba(X_test)
```

It takes in an input dataset `X_test` and returns the probability estimates for all classes (i.e., all possible labels).

A Bit of Statistics: Coefficient of Determination

- In statistics, the **coefficient of determination**, denoted R^2 or r^2, is the proportion of the variation in the dependent variable (i.e., output) that is predictable from the independent variable(s).
- For regression, for $i = 1, \ldots, N$, let y_i be observed output and let $\hat{y}_i = f(x_i)$ be predicted output. Let $\bar{y} = \frac{1}{N} \sum_{i=1}^{N} y_i$ be the sample mean of the output.
- Define the *total sum of squares*:

$$SS_{\text{tot}} = \sum_{i=1}^{N} (y_i - \bar{y})^2.$$

  SS_{tot} is the total variation of the observed data, as compared with sample mean $\bar{y}$. It is N times of the empirical or sample variance. SS_{tot} is the base line for variation comparison. If one makes no use

of the input X in predicting the output Y, then the best prediction for Y is its mean (assuming the expected square loss function).

- Define the *residual sum of squares*:

$$SS_{\text{res}} = \sum_{i=1}^{N} (y_i - \hat{y}_i)^2.$$

  SS_{res} is the remaining total variation after the prediction.

- The coefficient of determination is define as

$$R^2 \triangleq 1 - \frac{SS_{\text{res}}}{SS_{\text{tot}}} = \frac{SS_{\text{tot}} - SS_{\text{res}}}{SS_{\text{tot}}}.$$

  - $\frac{SS_{\text{res}}}{SS_{\text{tot}}}$ is known as the *fraction of variance unexplained*.
  - R^2 is *fraction of variance explained* by an estimator.
  - The maximum possible value of R^2 is 1, which happens when $SS_{\text{res}} = 0$ (your model fits the data perfectly).
  - $R^2 = 0$ when you use the constant $\bar{y}$ as the predicted output for

all input, which provides a baseline for models. Your model should aim at doing much better than that. It is possible to have worse models with $R^2 < 0$.

– Consider a linear model $Y = aX + \epsilon$ where ϵ is an independent random noise with zero mean and variance σ^2. Suppose the variance of X is σ_X^2.
  The theoretically best predictor is $E[Y|X] = aX$.

Then, given an input x_i, the predicted output is $\hat{y}_i = ax_i$.

$$
\begin{aligned}
SS_{\text{res}} &= \sum_{i=1}^{N}(y_i - \hat{y}_i)^2 = \sum_{i=1}^{N}(ax_i + \epsilon_i - ax_i)^2 \\
&= \sum_{i=1}^{N}\epsilon_i^2 \approx N\sigma^2 \quad \text{for large } N \\
SS_{\text{tot}} &= \sum_{i=1}^{N}(y_i - \bar{y}_i)^2 = \sum_{i=1}^{N}(ax_i + \epsilon_i - a(\bar{x} + \bar{\epsilon}))^2 \\
&= \sum_{i=1}^{N}a^2(x_i - \bar{x})^2 + \sum_{i=1}^{N}(\epsilon_i - \bar{\epsilon})^2 + 2\sum_{i=1}^{N}a(x_i - \bar{x})(\epsilon_i - \bar{\epsilon}) \\
&\approx Na^2\sigma_X^2 + N\sigma^2 + o(N),
\end{aligned}
$$

because $\frac{1}{N}\sum_{i=1}^{N}a(x_i - \bar{x})(\epsilon_i - \bar{\epsilon})$ approaches $\text{Cov}(X, \epsilon)$, which is equal to zero, as $N \to \infty$.

We see that

$$R^2 \approx \frac{SS_{\text{tot}} - SS_{\text{res}}}{SS_{\text{tot}}} = \frac{a^2\sigma_X^2}{a^2\sigma_X^2 + \sigma^2}.$$

We see that our best predictor can eliminate the $a^2\sigma_X^2$ part of the variance which is entirely from the input.
It can't eliminate the σ^2 part because the noise is independent of the input and it cannot be predicted from the input.

In this case, the correct model is linear and the best you can do is to explain $R^2 = \frac{a^2\sigma_X^2}{a^2\sigma_X^2+\sigma^2}$ portion of the variance.
However, if you use a higher-order polynomial to overfit the model, you can get to $R^2 = 1$. But, it will not generalize well.
We see that R^2 is not enough for judging a model. Having a test set is important.

Handling Text and Categorical Attributes

- `ocean_proximity` is a categorical attribute represented by text.
- Most machine learning algorithms prefer to work with numbers, so we need to convert from text to numbers.
- First, let try to use Scikit-Learn's OrdinalEncoder class.

```
from sklearn.preprocessing import OrdinalEncoder

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

```
array([0., 1., 2., 3., 4.])
```

Note that `ordinal_encoder` is an estimator and also a transformer.

With `[[ ]]`, `housing_cat` is a DataFrame and it is a copy.

`fit_transform()` returns a NumPy array. We listed the set of unique elements after the encoding.

- You can get the list of categories using the `categories_` instance variable. These are part of the "learning" results.

```
ordinal_encoder.categories_
```

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

- Issue: ML algorithms will assume that two nearby values are more similar than two distant values. For some categorical data, this makes sense, e.g., $\{1, 2, 3, 4, 5\}$ with 1 standing for 'strongly disagree', 5 standing for 'strongly agree'. These are known as *ordinal* numbers, each of which denotes the position of one item w.r.t others.

  That is not the case here. When encoding the `ocean_proximity`

categories into numbers $0, 1, \ldots, 4$, those numbers do not have ordering relationship but are merely IDs. These numbers are known as *nominal* numbers.

We can also say some categorical data are ordinal, and some are nominal.

We need a different encoding for nominal data.

One-Hot Encoding

- For a categorical attribute, create one new binary attribute for each category. For each instance, exactly one of the created attributes will be equal to 1 (one-hot).

  Our categories are: `<1H OCEAN`, `INLAND`, `ISLAND`, `NEAR BAY`, `NEAR OCEAN`. There are five new attributes.

instance's category	new-attr1	new-attr2	new-attr3	new-attr4	new-attr5
`<1H OCEAN`	1	0	0	0	0
`INLAND`	0	1	0	0	0
`ISLAND`	0	0	1	0	0
`NEAR BAY`	0	0	0	1	0
`NEAR OCEAN`	0	0	0	0	1

  The Scikit-Learn class OneHotEconder, which is a transformer, does the encoding for us.

```
from sklearn.preprocessing import OneHotEncoder
cat_encoder = OneHotEncoder()
housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
housing_cat_1hot
```

```
<16512x5 sparse matrix of type '<class 'numpy.float64'>'
        with 16512 stored elements in Compressed Sparse Row format>
```

Note that the result is a sparse matrix rather than a normal NumPy array.

This is useful when you have a categorical attribute with thousands of categories. After one-hot encoding, we get a matrix with thousands of columns, and the matrix is full of 0s except for a single 1 per row.

In such a case, a sparse matrix saves memory because it only stores the location of the nonzero elements.

You can use it mostly like a normal 2D array, but if you really want to convert it to a NumPy array, just call the `toarray()` method.

```
housing_cat_1hot.toarray()
```

```
array([[0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 1.],
       [0., 1., 0., 0., 0.],
       ...,
       [1., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.]])
```

- Note: One-hot encoding of a categorical attribute with a large number of categories can cause many problems. It slows down learning and degrade performance. There are other techniques to encode such an attribute.

Custom Transformer

- You will need to write your own transformers for tasks such as custom cleanup operations or combining specific attributes.
- All you need to do is create a class and implement three methods: `fit()` (returning self), `transform()`, and `fit_transform()`.
- You can get the last one for free by simply adding TransformerMixin as a base class.
- If you add BaseEstimator as a base class (do not use *args and **kargs in your constructor), you will also get two extra methods (`get_params()` and `set_params()`) that will be useful for automatic hyperparameter tuning.
- The following is an example of a transformer class, which adds the combined attributes we discussed earlier.

```
from sklearn.base import BaseEstimator, TransformerMixin
rooms_ix, bedrooms_ix, population_ix, households_ix = 3, 4, 5, 6

class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
    def __init__(self, add_bedrooms_per_room = True): # 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):
        rooms_per_household = X[:, rooms_ix] / X[:, households_ix]
        population_per_household = X[:, population_ix] / X[:, households_ix]
        if self.add_bedrooms_per_room:
            bedrooms_per_room = X[:, bedrooms_ix] / X[:, rooms_ix]
            return np.c_[X, rooms_per_household, population_per_household, bedrooms_per_room]
        else:
            return np.c_[X, rooms_per_household, population_per_household] # concatenate columns

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

```
housing_extra_attribs[1,:]
```

```
array([-117.23, 33.09, 7.0, 5320.0, 855.0, 2015.0, 768.0, 6.3373,
       'NEAR OCEAN', 6.927083333333333, 2.6236979166666665], dtype=object)
```

- The transformer has one hyperparameter, `add_bedrooms_per_room`, which is set to true by default.
- `housing.values`: Return a NumPy representation of the

DataFrame. Only the values in the DataFrame will be returned, the axes labels will be removed. It is better to use `housing.to_numpy()`.

- In general, Scikit-Learn works with NumPy array. However, many Scikit-Learn's transformer classes (e.g., `OrdinalEncoder`) can also take DataFrame as input. Our implementation of `transform()` can only take NumPy array as input.
- `transform, fit_transform`: usually return a NumPy array (sometimes, sparse matrix).
- `np.c_[ ]`: concatenate NumPy arrays along the column axis. It is a short hand for `np.r_['-1,2,0',... ]` with the string argument fixed.
- All estimators should specify all the parameters that can be set at the class level in their constructor `__init__` as explicit keyword arguments (i.e., do not use *args or **kwargs).

Feature Scaling

- One of the most important transformations you need to apply to your input data is feature scaling.
- Why? Machine Learning algorithms generally don't perform well when the input numerical attributes have very different scales.
- The housing data: the total number of rooms ranges from about 6 to 39,320, while the median incomes only range from 0 to 15.
- Two common methods to get all attributes to have the same scale: *min-max* scaling and *standardization*.
- **Min-max** scaling (aka normalization): values are shifted and rescaled so that they end up ranging from 0 to 1.

  Scikit-Learn provides a transformer called `MinMaxScaler` for this. It has a `feature_range` hyperparameter that lets you change the

range if you don't want 0–1.

- **Standardization**: First, subtracts the mean value, and then divides by the standard deviation.

  The transformed data has zero mean and unit variance.

  Standardization does not bound values to a specific range, which may be a problem for some algorithms (e.g., neural networks often expect an input value ranging from 0 to 1).

  However, standardization is much less affected by outliers. For example, suppose a district had a median income equal to 100 (by mistake). Min-max scaling would then crush all the other values from 0–15 down to 0–0.15, whereas standardization would not be much affected.

  Scikit-Learn provides a transformer called `StandardScaler` for standardization.

Transformation Pipelines

- There are many data transformation steps that need to be executed in the right order.
- Fortunately, Scikit-Learn provides the `Pipeline` class to help with such sequences of transformations.

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

num_pipeline = Pipeline([
            ('imputer', SimpleImputer(strategy="median")),
            ('attribs_adder', CombinedAttributesAdder()),
            ('std_scaler', StandardScaler()),
        ])
housing_num_tr = num_pipeline.fit_transform(housing_num)
```

- The `Pipeline` constructor takes a list of name/estimator pairs

defining a sequence of steps.

- All but the last estimator must be transformers (i.e., they must have a `fit_transform()` method).
- When you call the pipeline's `fit()` method, it calls `fit_transform()` sequentially on all transformers, passing the output of each call as the parameter to the next call until it reaches the final estimator, for which it calls the `fit()` method.
- The pipeline exposes the same methods as the final estimator.

  In this example, the last estimator is a `StandardScaler`, which is a transformer, so the pipeline has a `transform()` method that applies all the transforms to the data in sequence (and of course also a `fit_transform()` method, which is the one we used).

ColumnTransformer

- So far, the categorical columns and the numerical columns are handled separately.
- It would be convenient to have a single transformer to handle all columns.
- `ColumnTransformer` is for this purpose, and it works well with pandas DataFrames.

```
from sklearn.compose import ColumnTransformer

num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]
full_pipeline = ColumnTransformer([
                ("num", num_pipeline, num_attribs),
                ("cat", OneHotEncoder(), cat_attribs),
            ])

housing_prepared = full_pipeline.fit_transform(housing)
```

- The `ColumnTransformer` constructor requires a list of tuples, where each tuple contains a name, a transformer (or pipeline), and a list of names (or indices) of columns that the transformer should be applied to.

  The name can be anything as long as it doesn't contain double underscores.

- Recall the `OneHotEncoder` returns a sparse matrix, while the `num_pipeline` returns a regular (aka dense) matrix.

  When there is such a mix of sparse and dense matrices, the `ColumnTransformer` estimates the density of the final matrix (i.e., the ratio of nonzero cells), and it returns a sparse matrix if the density is lower than a given threshold (by default, `sparse_threshold=0.3`).

  In this example, it returns a dense matrix.

Select and Train a Model

Training and Evaluating on the Training Set

- First, train a Linear Regression model.

```
from sklearn.linear_model import LinearRegression

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

- Use the trained model for prediction on the first five rows.

```
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: [ 85657.90192014 305492.60737488 152056.46122456 186095.70946094
 244550.67966089]
```

```
print("Labels:", list(some_labels))
```

```
Labels: [72100.0, 279600.0, 82700.0, 112500.0, 238300.0]
```

It works; but how well?.

- Check the regression model's RMSE (root mean square error) on the whole training set using the `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
```

```
68627.87390018745
```

- Most districts' `median_housing_values` range between $120,000 and $265,000. A typical prediction error of $68,628 is not very satisfying.
- This is an example of a model underfitting the training data. When this happens, it may be due to:
  - the features do not provide enough information to make good predictions,
  - the model is not powerful enough, or is constrained (e.g., by regularization),
  - other reasons.
- The main ways to fix underfitting:
  - select a more powerful model,
  - feed the training algorithm with better features,
  - or reduce the constraints on the model (not applicable here).

Decision Tree

- You could try to add more features (e.g., the log of the population), but first let's try a more complex model to see how well it works.
- A decision tree is capable of finding complex nonlinear relationships in the data.
- By now, we know how to use it without knowing how it works (which will come later).

```
from sklearn.tree import DecisionTreeRegressor

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

```
▾ DecisionTreeRegressor
DecisionTreeRegressor()
```

```
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
```

- This is almost certainly an overfitting situation. We will be able to tell if we apply the trained model to the test set.
- However, you don't want to touch the test set until you are very confident about a model.

- At this point, we are far from being confident because we have hardly explored the balance between overfitting and underfitting.
- We need to explore more while having a way to evaluate the explored model.
- Solution: You need to use part of the training set for training and part of it for model validation.

Better Evaluation Using Cross-Validation

- One way: Use the `train_test_split()` function to split the training set into a smaller training set and a validation set, then train your models against the smaller training set and evaluate them against the validation set.
- Better alternative: Use Scikit-Learn's K-fold cross-validation feature.

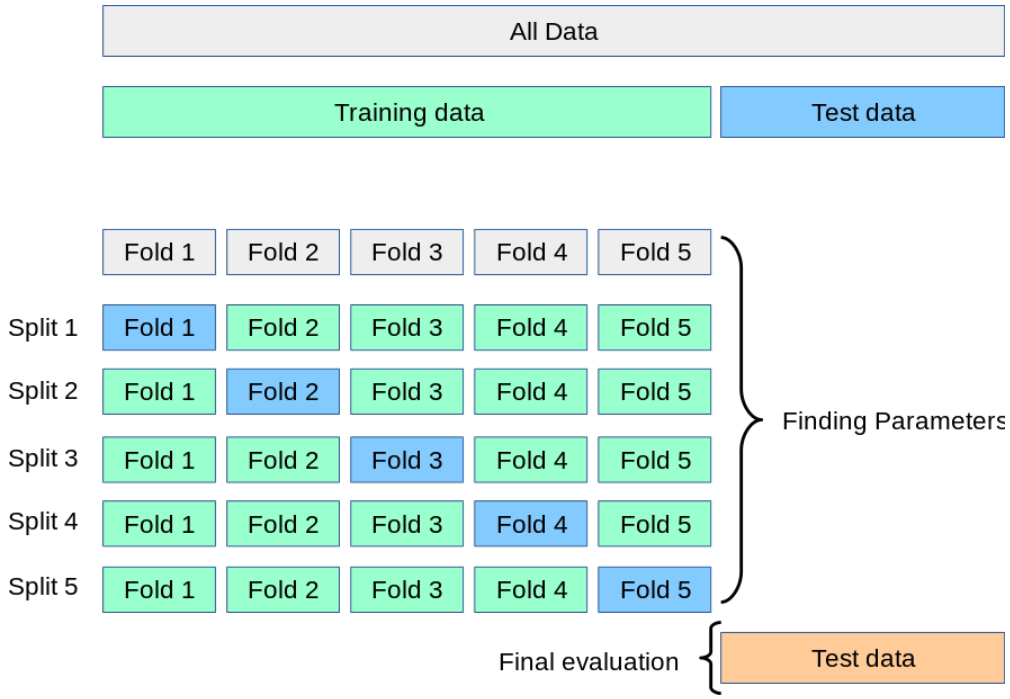


Figure 1: Cross-validation example: 5 folds

- The following code randomly splits the training set into 10 distinct subsets called **folds**. Then, it trains and evaluates the decision tree model 10 times, picking a different fold for evaluation every time and training on the other 9 folds.

The result is an array containing the 10 evaluation scores.

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(tree_reg, housing_prepared, housing_labels,
                         scoring="neg_mean_squared_error", cv=10)
tree_rmse_scores = np.sqrt(-scores)
```

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

display_scores(tree_rmse_scores)
```

```
Scores: [72251.22610062 70811.34799853 68000.08591434 71929.39104752
 69328.61392367 77963.99482744 71628.79228943 73511.78648072
 69045.58722762 69854.4073967 ]
Mean: 71432.52332065887
Standard deviation: 2696.6824522410743
```

- Scikit-Learn's cross-validation features expect a utility function (greater is better) rather than a cost function, so the scoring function is the negative of the MSE.

- `cross_val_score()` does not shuffle the data. If the original data in the training set has not been shuffled, you will need to shuffle it before calling `cross_val_score()`, for instance, using `KFold` or `StratifiedKFold` with option `shuffle=True`. In our case, we have done shuffling earlier when we split the data into the training and test sets.
- `cross_val_score()` is a way of assessing a model, not for final training. Final training should take place on all available training data.
- `cross_val_score()` doesn't modify your model/estimator (the first argument). It doesn't save the learned model or model parameters after each round of training and validation.
- There is another related method `cross_validate()`, which gives the option to save the estimator fitted on each training set (when `return_estimator=True`).

- **Importantly:** Recall that all learned parameters can be considered random. If you feed different training data to the learning algorithm, you will get different learned parameters.

  Therefore, the model performance is also random.

- Cross-validation allows you to get not only an estimate of the

performance of your model, but also a measure of how precise this estimate is (i.e., its standard deviation).

The decision tree has a score of approximately 71,407, generally $\pm 2,439$. You would not have this information if you just used one validation set.

But cross-validation comes at the cost of training the model several times.

- Let's compute the same scores for the linear regression model.

```
lin_scores = cross_val_score(lin_reg, housing_prepared, housing_labels,
    scoring="neg_mean_squared_error", cv=10)
lin_rmse_scores = np.sqrt(-lin_scores)
display_scores(lin_rmse_scores)
```

```
Scores: [71762.76364394 64114.99166359 67771.17124356 68635.19072082
 66846.14089488 72528.03725385 73997.08050233 68802.33629334
 66443.28836884 70139.79923956]
Mean: 69104.07998247063
Standard deviation: 2880.3282098180675
```

Both models have similar performance. The decision tree is slightly worse.

Random Forest

- Let's try yet another model now: the `RandomForestRegressor`.
- Random forests work by training many decision trees on random subsets of the features, then averaging out their predictions.
- Building a model on top of many other models is called **Ensemble Learning**, and it is often a great way to push ML algorithms even further.
- Running this code takes quite a bit longer.

```
from sklearn.ensemble import RandomForestRegressor

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)
display_scores(forest_rmse_scores)
```

```
Scores: [50805.9803667  48854.73952453 47065.07345482 52126.31842501
 47568.58945875 51681.12362029 52421.98850781 49965.91873683
 48836.23747416 53890.88649719]
Mean: 50321.68560660848
Standard deviation: 2125.085836974857
```

- The random forest has better performance.
- However, the random forest is still overfitting on the training set, as the training RMSE is much smaller than the validation RMSE.

```
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)
forest_rmse
```

```
18645.786147705774
```

Possible Solutions for Overfitting

- simplify the model,
- constrain it (i.e., regularize it),
- get a lot more training data,
- tweak the hyperparameters of the model,
- try out other models from various categories (e.g., several Support Vector Machines with different kernels, and possibly a neural network) before spending too much time tweaking the hyperparameters for one model. The goal is to shortlist a few (two to five) promising models.

Save Your Models

- You should save every model you experiment with so that you can come back easily to any model you want.
- Some algorithm-dataset combinations take long to train. Once you have done it, you should save the trained model.
- Make sure you save both the hyperparameters and the trained parameters.
- You should also save the cross-validation scores. This will allow you to easily compare scores across model types, and compare the types of errors they make.
- You may also want to save the actual predictions after your model is trained on the full training set.
- You can easily save Scikit-Learn models by using Python's `pickle`

module or by using the `joblib` library.

`joblib` is more efficient at serializing large NumPy arrays that are present in some learning algorithms (such as K-Nearest Neighbors).

For small models, you won't see much difference.

The following is an example with `joblib`.

```
import joblib
filename = 'joblib_forest.sav'
joblib.dump(forest_reg, filename)  # random forest estimator
```

```
['joblib_forest.sav']
```

```
# some time later
loaded_forest_reg = joblib.load(filename)
result = loaded_forest_reg.score(housing_prepared, housing_labels)
print(result)
```

```
0.9740276501805243
```

Recall regression models have a `score()` method, which gives the R^2 value, the coefficient of determination.

The next is an example with `pickle`.

```
import pickle
filename_pkl = 'forest_pickle.sav'
pickle.dump(forest_reg, open(filename_pkl, 'wb'))
```

```
# some time later
loaded_model = pickle.load(open(filename_pkl, 'rb'))
```

- You can also manually save the model parameters by accessing the estimator's variables. This has the advantage of future compatibility.

Fine-Tune Your Model

Grid Search

- Use Scikit-Learn's `GridSearchCV` to search for best hyperparameters.
- Tell it which hyperparameters you want it to experiment with and what values to try out, and it will use cross-validation to evaluate all the possible combinations of hyperparameter values. Example:

```
from sklearn.model_selection import GridSearchCV

param_grid = [
    {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
    {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]},
]
forest_reg = RandomForestRegressor()
grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
                           scoring='neg_mean_squared_error',
                           return_train_score=True)

grid_search.fit(housing_prepared, housing_labels)
```

- Here, `param_grid` is a list of two dictionaries. It tells Scikit-Learn to first evaluate all $3 \times 4 = 12$ combinations of `n_estimators` and `max_features` hyperparameter values specified in the first `dict`. Then, try all $2 \times 3 = 6$ combinations of hyperparameter values in the second `dict`, but this time with the `bootstrap` hyperparameter set to `False` instead of `True` (which is the default value for this hyperparameter).
- `n_estimators`: number of trees in a forest. Default is 100.
- The grid search will explore $12 + 6 = 18$ combinations of `RandomForestRegressor` hyperparameter values, and it will train each model 5 times (since we are using 5-fold cross validation). All in all, there will be $18 \times 5 = 90$ rounds of training, which may take some time.

  When it is done you can get the best combination of parameters like this:

```
grid_search.best_params_
```

```
{'max_features': 6, 'n_estimators': 30}
```

Since 30 is the maximum value evaluated for `n_estimators`, you should probably try searching again with higher values; the score may continue to improve.

- You can also get the best estimator directly:

```
grid_search.best_estimator_
```

```
▾ RandomForestRegressor
RandomForestRegressor(max_features=6, n_estimators=30)
```

- If `GridSearchCV` is initialized with `refit=True` (which is the default), then once it finds the best estimator using cross-validation, it

retrains it on the whole training set.

- You can get the evaluation scores for different estimators (i.e., different combinations of hyperparameter values).

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

```
63934.23520558415 {'max_features': 2, 'n_estimators': 3}
55410.916671145314 {'max_features': 2, 'n_estimators': 10}
53001.2532391972 {'max_features': 2, 'n_estimators': 30}
60415.374105929295 {'max_features': 4, 'n_estimators': 3}
52405.88873579722 {'max_features': 4, 'n_estimators': 10}
50613.85310563787 {'max_features': 4, 'n_estimators': 30}
59229.6048856397 {'max_features': 6, 'n_estimators': 3}
51889.018515685275 {'max_features': 6, 'n_estimators': 10}
50024.890006984235 {'max_features': 6, 'n_estimators': 30}
58526.99890169603 {'max_features': 8, 'n_estimators': 3}
52433.20895777424 {'max_features': 8, 'n_estimators': 10}
50345.82246876446 {'max_features': 8, 'n_estimators': 30}
62033.41921684935 {'bootstrap': False, 'max_features': 2, 'n_estimators': 3}
54389.19151624534 {'bootstrap': False, 'max_features': 2, 'n_estimators': 10}
60513.40848452883 {'bootstrap': False, 'max_features': 3, 'n_estimators': 3}
52651.85991077819 {'bootstrap': False, 'max_features': 3, 'n_estimators': 10}
58017.13627048026 {'bootstrap': False, 'max_features': 4, 'n_estimators': 3}
51676.062272793904 {'bootstrap': False, 'max_features': 4, 'n_estimators': 10}
```

In this case, the best score 50025 is only slightly better than the

estimator with the default hyperparameter values, which has a score 50322.

You can also check that the best estimator still overfits the entire training data.

`zip()`: combines one or more iterables (e.g., list, array, dictionary, tuple) into a single iterator. The result is an object of the type `zip`, which can be viewed by applying `list()`.

- Recall that some of the data preparation steps have hyperparameters. You can use grid search on those.

  For example, the grid search can automatically find out whether or not to add a feature you were not sure about (e.g., using the `add_bedrooms_per_room` hyperparameter of your `CombinedAttributesAdder` transformer).

  It may similarly be used to automatically find the best way to handle outliers, missing features, feature selection, and more.

How: Do grid search on a pipeline. There are multiple estimators in a pipeline. You can do grid search on the hyperparameters of different estimators.

Note that Pipeline and RandomForestRegressor are both estimators (so is GridSearchCV). You can do grid search on both RandomForestRegressor and Pipeline.

You should look for more information about this. For instance, see:

Tomas Beuzen, “Simultaneous feature preprocessing, feature selection, model selection, and hyperparameter tuning in Scikit-Learn with Pipeline and GridSearchCV”. https://www.tomasbeuzen.com/post/scikit-learn-gridsearch-pipelines/

Randomized Search

- Grid search doesn't work well when the hyperparameter search space is large. When that is the case, one can use `RandomizedSearchCV` instead.
- This class can be used in much the same way as the `GridSearchCV` class, but instead of trying out all possible combinations, it tries a number of iterations specified by you, and in each iteration, it selects a random value for each hyperparameter.

Ensemble Methods

- Another way to fine-tune your system is to try to combine the models that perform the 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.
- More details will come later.

Analyze the Best Models and Their Errors

- You will 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([7.60441893e-02, 7.29616960e-02, 4.37479688e-02, 1.83918790e-02,
       1.65723759e-02, 1.75940012e-02, 1.68902455e-02, 3.33334131e-01,
       6.60464490e-02, 1.05826875e-01, 7.84188168e-02, 1.13514216e-02,
       1.34042730e-01, 5.11382124e-05, 2.87827452e-03, 5.84780837e-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 = full_pipeline.named_transformers_["cat"]
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)
```

```
[(0.3333341307853294, 'median_income'),
 (0.13404272980025225, 'INLAND'),
 (0.10582687524718433, 'pop_per_hhold'),
 (0.07841881677107472, 'bedrooms_per_room'),
 (0.07604418932462566, 'longitude'),
 (0.07296169598371789, 'latitude'),
 (0.06604644897745622, 'rooms_per_hhold'),
 (0.04374796882835045, 'housing_median_age'),
 (0.018391878984044888, 'total_rooms'),
 (0.017594001223356404, 'population'),
 (0.016890245541757203, 'households'),
 (0.016572375865343374, 'total_bedrooms'),
 (0.011351421563823768, '<1H OCEAN'),
 (0.005847808372679485, 'NEAR OCEAN'),
 (0.0028782745185584164, 'NEAR BAY'),
 (5.1138212445531596e-05, 'ISLAND')]
```

With this information, you may want to try dropping some of the less useful features (e.g., apparently only one `ocean_proximity` category is really useful, so you could try dropping the others).

- You should also look at the specific errors that your system makes, then try to understand why it makes them and what could fix the problem (adding extra features or getting rid of uninformative ones, cleaning up outliers, etc.).

Evaluate Your System on the Test Set

- Get the predictors (input features) and the labels from your test set.
- Run your `full_pipeline` to transform the data (call `transform()`, not `fit_transform()` – you do not want to fit the test set!).
- Evaluate the final model on the test set.

```
final_model = grid_search.best_estimator_
X_test = strat_test_set.drop("median_house_value", axis=1)
y_test = strat_test_set["median_house_value"].copy()

X_test_prepared = full_pipeline.transform(X_test)

final_predictions = final_model.predict(X_test_prepared)
final_mse = mean_squared_error(y_test, final_predictions)
final_rmse = np.sqrt(final_mse)
```

```
final_rmse
```

```
47922.03505354278
```

- The computed RMSE measures how good the predicted outputs are from the true outputs. It is computed based on a single set of the observed data. We don't know how far it is from the true RMSE (which is defined using the underlying probability distribution).

  To have a sense where the true RMSE is, we can calculate the 95%-confidence interval (for the true RMSE).

```
from scipy import stats
confidence = 0.95
squared_errors = (final_predictions - y_test) ** 2
np.sqrt(stats.t.interval(confidence, len(squared_errors) - 1,
    loc=squared_errors.mean(),
    scale=stats.sem(squared_errors)))
```

```
array([45993.24003251, 49776.14648261])
```

In the code, the collection of data is $(y_1 - \hat{y}_1)^2, (y_2 - \hat{y}_2)^2, \ldots, (y_N - \hat{y}_N)^2$, where each y_i is the true output (label), and $\hat{y}_i$ is the predicted output.

`len(squared_errors)-1` is the degrees of freedom of the (generalized) Student's t-distribution.

To call the `t.interval()` function, you also pass the sample mean of the above data as `loc` and the computed standard error of the sample mean (sem) as `scale`. (See later slides for more details.)

- If you did a lot of hyperparameter tuning, the performance on the test set will usually be slightly worse than what you measured using cross-validation (because your system ends up fine-tuned to perform well on the validation data and will likely not perform as well on unknown datasets).

  It is not the case in this example. Our best score in the grid search is 50025.

- But when it does happen, you must resist the temptation to tweak the hyperparameters to make the numbers look good on the test set; the improvements would be unlikely to generalize to new data.

Some Basic Statistics

- Suppose we are given a random sample $Z = (Z_1, \ldots, Z_N)$. That is, the Z_i's are IID random variables. Suppose the mean is μ and the variance is σ^2.
- The **sample mean** is defined as

$$\bar{Z} = \frac{1}{N} \sum_{i=1}^{N} Z_i.$$

- The **sample variance** is defined as

$$S^2 = \frac{1}{N-1} \sum_{i=1}^{N} (Z_i - \bar{Z})^2.$$

- Since the sample mean is also random, we can ask about its mean and

its variance.

$$E[\bar{Z}] = \frac{1}{N}\sum_{i=1}^{N} E[Z_i] = \frac{N\mu}{N} = \mu$$

$$\text{var}(\bar{Z}) = \frac{1}{N^2}\sum_{i=1}^{N} \text{var}(Z_i) = \frac{\sigma^2}{N}.$$

Note the reduction of variance when averaging: in proportion to $1/N$.

- The standard deviation of $\bar{Z}$ is $\frac{\sigma}{\sqrt{N}}$, which is known as the **standard error of the (sample) mean**.
- In statistics, we often use the sample mean $\bar{Z}$ as the estimator for the true mean μ (an unknown constant). By the law of large numbers, we know that $\frac{1}{N}\sum_{i=1}^{N} Z_i \to \mu$ as $N \to \infty$. But, it is difficult to say how good that estimate is for a fixed-sized sample.
- Alternatively, one can consider a confidence interval.
- A γ**-confidence interval** for the parameter μ is an interval

$(a(Z), b(Z))$ such that $P(a(Z) \leq \mu \leq b(Z)) = \gamma$.

Such γ is called the **confidence level** or confidence coefficient.

- Meaning: Such an interval $(a(Z), b(Z))$ is random because it depends on the random sample $Z = (Z_1, \ldots, Z_N)$. If you have a large of number of realizations of the random sample (i.e., a large number of different sets of N data points), a fraction γ of the intervals will contain the parameter μ.
- Note that a confidence interval is always with respect to an underlying parameter such as μ.
- Suppose the values of a sample are $z = (z_1, \ldots, z_N)$ and suppose we get $a(z) = 4$ and $b(z) = 8$ for $\gamma = 0.95$. It is a usual practice to say the 95% confidence interval is $(4, 8)$, even though this is just one particular realization of the confidence interval $(a(Z), b(Z))$.

  Moreover, we don't mean that the probability of any single Z_i falling on $(4, 8)$ is 0.95.

We also don't mean that the probability of the sample mean $\bar{Z}$ falling on $(4, 8)$ is 0.95.

Roughly, we mean that we are 95% confident that the **unknown** constant parameter μ (usually the true mean) falls on $(4, 8)$.

- Note that it is not alway possible to compute $P(a(Z) \leq \mu \leq b(Z))$ because it depends on the constant μ, which is unknown. However, in some situations, the probability does not depend on the unknown parameter(s).

Confidence Interval and Student's t-Distribution

- Confidence interval is not uniquely define. `scipy.stats.t` computes a particular kind of confidence interval using the Student's t-distribution, which is motivated by the following.
- Suppose $Z_1, \ldots, Z_N$ are IID whose common distribution is Gaussian (i.e., a normal distribution) with unknown mean μ and unknown variance σ^2.
- Then, the random variable $T = \frac{\bar{Z}-\mu}{S/\sqrt{N}}$ has the Student's t-distribution with $N-1$ degrees of freedom.
- The probability density function of Student's t-distribution with ν degrees of freedom is:

$$p(x) = \frac{\Gamma(\frac{\nu+1}{2})}{\sqrt{\nu\pi}\Gamma(\frac{\nu}{2})}\left(1+\frac{x^2}{\nu}\right)^{-\frac{\nu+1}{2}}, \quad x \in (-\infty, \infty),$$

where $\Gamma(u)$ is the Γ-function defined by

$$\Gamma(u) = \int_0^\infty t^{u-1} e^{-t} dt.$$

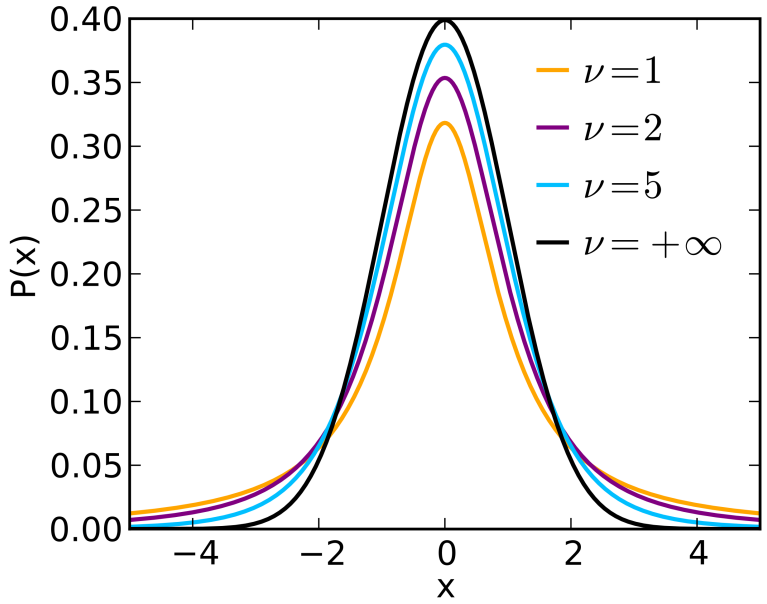


Figure 2: Probability density functions of Student's t-distribution.

- It has the mean 0 for $\nu > 1$ (otherwise, undefined).

- It has the variance $\frac{\nu}{\nu-2}$ for $\nu > 2$, ∞ for $1 < \nu \leq 2$, and undefined for other ν values.
- As $\nu \to \infty$, Student's t-distribution approaches a standard normal distribution: $p(x) \to \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$.
- Student's t-distribution has heavy tails when ν is small. For $\nu = 1$, Student's t-distribution becomes the standard Cauchy distribution, which has very "fat" tails. The tails become less and less heavy as ν increases.
- T is a quantity constructed based on the random sample and the unknown parameters of the model (here, only μ). Interestingly, the distribution of T does not depend on the values of the unobservable parameters μ and σ^2. In statistics, such a quantity is called a **pivotal quantity**, which is often very useful.
- Here, we use T to come up with a confidence interval. Suppose for a given γ, we find $c > 0$ such that $P(-c \leq T \leq c) = \gamma$.

The event $\{-c \leq T \leq c\}$ is the same as $\{-c \leq \frac{\bar{Z}-\mu}{S/\sqrt{N}} \leq c\}$, which is same as

$$\{\bar{Z} - \frac{cS}{\sqrt{N}} \leq \mu \leq \bar{Z} + \frac{cS}{\sqrt{N}}\}.$$

Then, the γ-confidence interval for μ is

$$(\bar{Z} - \frac{cS}{\sqrt{N}}, \bar{Z} + \frac{cS}{\sqrt{N}}).$$

Note: The above confidence interval is random, as it is supposed to.

- Suppose we are given a particular realization of the sample $(Z_1, \ldots, Z_N)$, i.e., a dataset. Suppose we compute the sample mean and sample variance using the dataset and get $\bar{Z} = \bar{z}$ and $S^2 = s^2$. Then, the realization of the γ-confidence interval for μ is

$$(\bar{z} - \frac{cs}{\sqrt{N}}, \bar{z} + \frac{cs}{\sqrt{N}}).$$

Note: In practice, one often call $(\bar{z} - \frac{cs}{\sqrt{N}}, \bar{z} + \frac{cs}{\sqrt{N}})$ the γ-confidence

interval for μ, which is a constant interval when the data is given.

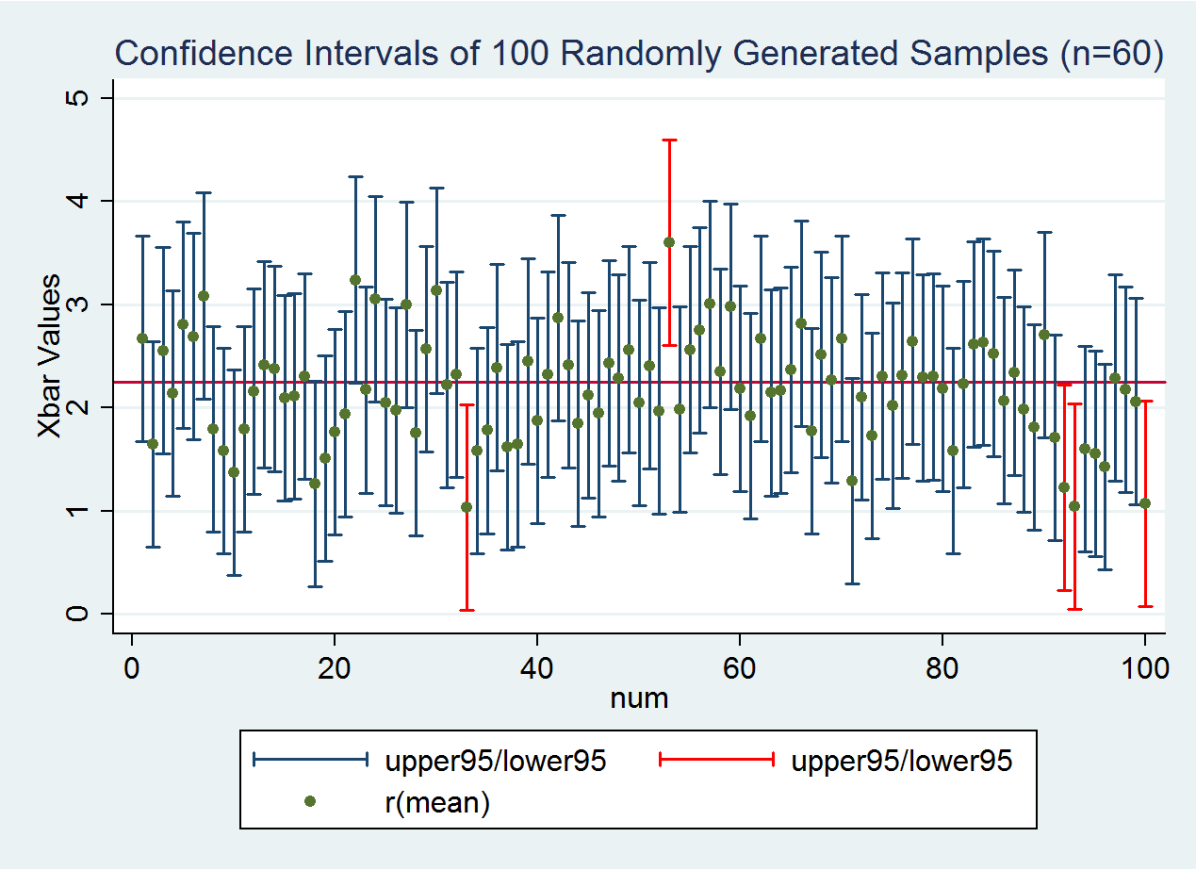
Meaning: Suppose we draw datasets many times and for each dataset, we compute the above interval. Then, a fraction γ of all such intervals will contain μ.

However, if we are given only one dataset, the interval $(\bar{z} - \frac{cs}{\sqrt{N}}, \bar{z} + \frac{cs}{\sqrt{N}})$ is fixed and there is no notion of fraction. The interval either contains μ or it doesn't. In this case, we can say that our confidence level is γ that the true mean μ lies in the interval $(\bar{z} - \frac{cs}{\sqrt{N}}, \bar{z} + \frac{cs}{\sqrt{N}})$. Such a statement only makes sense because we don't know μ.

Interestingly, this level of confidence seems to be based on the above frequentist/fractional interpretation of probability. (Think about where your confidence is from.)

Some people also say that there is a chance of γ that the true mean μ falls into $(\bar{z} - \frac{cs}{\sqrt{N}}, \bar{z} + \frac{cs}{\sqrt{N}})$; however, such a statement is quite vague and problematic. (What do we mean by chance?)

- Example: from https://sites.nicholas.duke.edu/statsreview/ci/



  - 100 randomly generated samples. The size of each sample is 60 data points.
  - The true mean is $\mu = 2.25$, shown as the horizontal red line.
  - Each vertical bar is a confidence interval, centered on a sample mean (green point).

The intervals all have the same length, but are centered on different sample means because the data are random.

- The five red confidence intervals DO NOT cover the true mean. This is what we would expect using a 95% confidence level: approximately 95% of the intervals cover the true mean.

- Suppose T has a Student's t-distribution with ν degrees of freedom. Consider another random variable $R = \hat{\mu} + \hat{\sigma}T$ or equivalently $T = \frac{R-\hat{\mu}}{\hat{\sigma}}$.

  We say R has a generalized Student's t-distribution with three parameters, ν, $\hat{\mu}$ and $\hat{\sigma}$.

  $\hat{\mu}$ is called the **location** parameter, which has to do with shift of the mean; $\hat{\sigma}$ is called the **scale** variable, which has to do with scaling the random variable and affects the variance. These terms show up in the early code.

Side Note: Confidence Interval under Known σ

- Suppose $Z_1, \ldots, Z_N$ are IID whose common distribution is Gaussian (i.e., a normal distribution) with unknown mean μ but **known** variance σ^2.
- Then, the random variable $\bar{Z} = \frac{1}{N} \sum_{i=1}^{N} Z_i$ has a normal distribution with mean μ and variance σ^2/N.
- Then, $W = \frac{\bar{Z}-\mu}{\sigma/\sqrt{N}}$ has the standard normal distribution (with mean 0 and variance 1), $\mathcal{N}(0,1)$.
- For the given confidence level γ, we find $b > 0$ such that

$$P(-b < W < b) = \gamma.$$

- The event $\{-b < W < b\}$ is the same as $\{-b < \frac{\bar{Z}-\mu}{\sigma/\sqrt{N}} < b\}$, which

is the same as

$$\{\bar{Z} - \frac{b\sigma}{\sqrt{N}} < \mu < \bar{Z} + \frac{b\sigma}{\sqrt{N}}\}.$$

We see that $(\bar{Z} - \frac{b\sigma}{\sqrt{N}}, \bar{Z} + \frac{b\sigma}{\sqrt{N}})$ is the γ-confidence interval for μ.

- For $\gamma = 0.95$, $b = 1.96$. The 95%-confidence interval for μ is

$$\{\bar{Z} - 1.96\frac{\sigma}{\sqrt{N}} < \mu < \bar{Z} + 1.96\frac{\sigma}{\sqrt{N}}\}.$$

- When the true variance σ^2 is **unknown**, we use Student's t-distribution to estimate the confidence interval. This technique can be understood as using the sample variance S^2 as a replacement of the true variance σ^2 (or equivalently, S for σ, the standard deviation). We get confidence interval of the form:

$$(\bar{Z} - \frac{cS}{\sqrt{N}}, \bar{Z} + \frac{cS}{\sqrt{N}}).$$

- In Student's t-distribution, the degrees of freedom is a measure of

how well S estimates σ. The larger the degrees of freedom, the better σ is estimated.

- For a very large sample size, using either the t-distribution or the normal distribution give almost the same result because when the sample size is large, the estimate of the standard deviation is likely to be very close to the true standard deviation.
- Up to now, we assume that the underlying data are drawn from a Gaussian distribution; that is, the sample $Z_1, \ldots, Z_N$ are IID Gaussian with mean μ and variance σ^2. If that is not the case, the confidence interval calculation is only reliable for large N. This is due to the Central Limit Theorem. The implication is that, for large N, the distribution of the sample mean $\bar{Z}$ is approximately a normal distribution with mean μ and variance σ^2/N (see later).

Back to Our Code

```
final_model = grid_search.best_estimator_
X_test = strat_test_set.drop("median_house_value", axis=1)
y_test = strat_test_set["median_house_value"].copy()

X_test_prepared = full_pipeline.transform(X_test)

final_predictions = final_model.predict(X_test_prepared)
final_mse = mean_squared_error(y_test, final_predictions)
final_rmse = np.sqrt(final_mse)
```

```
final_rmse
```

```
47922.03505354278
```

```
from scipy import stats
confidence = 0.95
squared_errors = (final_predictions - y_test) ** 2
np.sqrt(stats.t.interval(confidence, len(squared_errors) - 1,
    loc=squared_errors.mean(),
    scale=stats.sem(squared_errors)))
```

```
array([45993.24003251, 49776.14648261])
```

What Do MSE and RMSE Measure?

- Suppose the training dataset $\mathcal{D}$ is given. Some of the learning algorithm is deterministic (such as linear regression by using the formula for the solutions); others are random, such as a random forest.
- Conditional on the training data and on the realization of the learning algorithm, the learned model $\hat{f}$ can be considered as being deterministic. Let the event we are conditioning on be denoted by $\mathcal{A}$.
- The predicted output is $\hat{Y} = \hat{f}(X)$.
- Let us call $E[(Y - \hat{Y})^2|\mathcal{A}] = E[(Y - \hat{f}(X))^2|\mathcal{A}]$ the **true** (conditional) mean squared error.
- $\sqrt{E[(Y - \hat{Y})^2|\mathcal{A}]}$ is called the **true** (conditional) root mean squared error.

- Suppose $(y_1, \ldots, y_N)$ and $(\hat{y}_1, \ldots, \hat{y}_N)$ are N instances of true output and predicted output given the realized input $x_1, \ldots, x_N$. The sample (conditional) mean square error (MSE) is defined as

$$\text{MSE} = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2.$$

  The sample (conditional) root mean square error (RMSE) is:

$$\text{RMSE} = \sqrt{\text{MSE}}.$$

- MSE is an estimator for $E[(Y - \hat{Y})^2|\mathcal{A}]$. Then, RMSE is an estimator for $\sqrt{E[(Y - \hat{Y})^2|\mathcal{A}]}$.
- Reason: Conditional on $\mathcal{A}$, $(Y_1 - \hat{Y}_1)^2, \ldots, (Y_N - \hat{Y}_N)^2$ are IID. Then, conditional on $\mathcal{A}$, $\frac{1}{N} \sum_{i=1}^{N} (Y_i - \hat{Y}_i)^2$ is a unbiased and consistent estimator of $\text{E}[(Y - \hat{Y})^2|\mathcal{A}]$.
- The confidence interval in the code is for $\sqrt{E[(Y - \hat{Y})^2|\mathcal{A}]}$.

Estimate the Standard Error of the Sample Mean (SEM)

- `scipy.stats.sem()` calculates the sem. Here is what it does.
- Suppose the data is $(z_1, \ldots, z_N)$. First, calculate the sample mean $\bar{z}$:

$$\bar{z} = \frac{1}{N} \sum_{i=1}^{N} z_i.$$

- Then, calculate the sample variance s^2:

$$s^2 = \frac{1}{N-1} \sum_{i=1}^{N} (z_i - \bar{z})^2.$$

- Then, the estimated SEM is calculated as

$$\text{estimated SEM} = \frac{s}{\sqrt{N}}.$$

Confidence Intervals in the Code

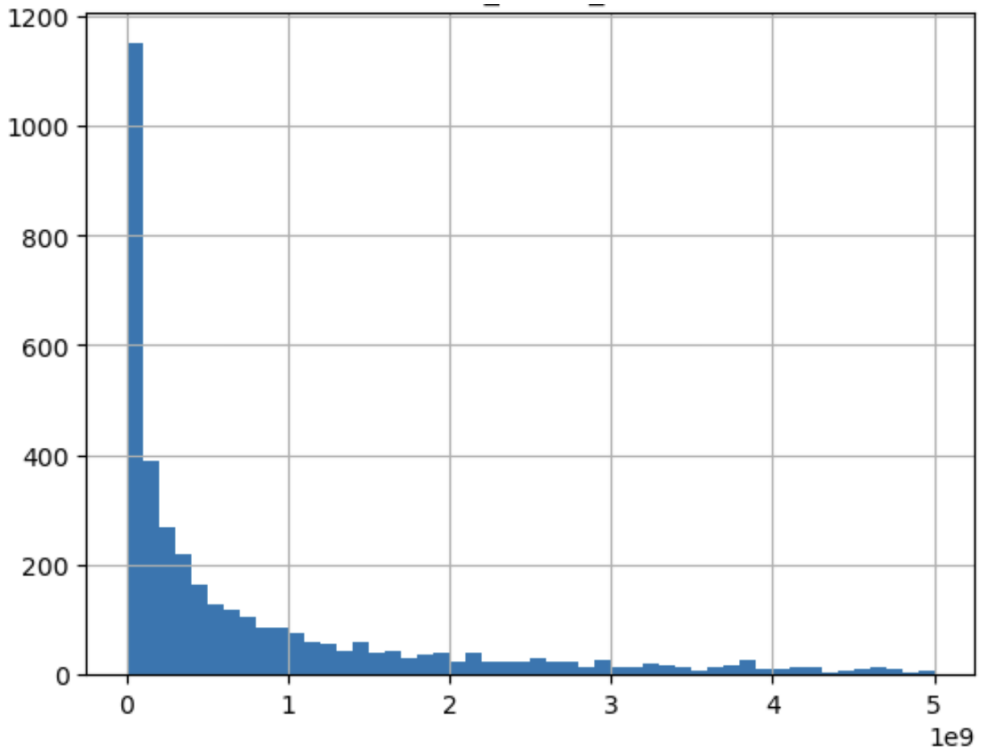
- In the code, the data is all the squared prediction errors:
  $z_1 = (y_1 - \hat{y}_1)^2, z_2 = (y_2 - \hat{y}_2)^2, \ldots, z_N = (y_N - \hat{y}_N)^2,$
  `len(squared_errors)-1` is the degrees of freedom of the (generalized) Student's t-distribution.

  To call the `t.interval()` function, you also pass the sample mean of the above data as `loc` and the computed standard error of the sample mean (estimated SEM) as `scale`.

  `t.interval()` returns the 95% confidence interval for the true mean squared error $E[(Y - \hat{Y})^2]$.

  We then take the square root of the interval end points. This gives the 95% confidence interval for the true root mean squared error $\sqrt{E[(Y - \hat{Y})^2]}$.
- Potential Issue: The data $z_1, \ldots, z_N$ are not drawn from a normal

distribution. See the histogram of the squared errors.



However, in our case $N = 4128$, which is fairly large, and the Central Limit Theorem kicks in. Also, by the law of large numbers, the sample variance s^2 becomes very close to the true variance σ^2. We are likely getting reliable estimate of the confidence interval.

Central Limit Theorem: Suppose $Z_1, Z_2, \ldots$ are IID random variables with mean μ and variance σ^2. Let $\bar{Z}_n = \frac{1}{n}\sum_{i=1}^{n} Z_i$, which is the sample mean. As $n \to \infty$, $\frac{\bar{Z}_n - \mu}{\sigma/\sqrt{n}}$ converges in distribution to the standard normal distribution $\mathcal{N}(0,1)$.

$$\frac{\bar{Z}_n - \mu}{\sigma/\sqrt{n}} \xrightarrow{d} \mathcal{N}(0,1)$$

The implication is that for a large N, the distribution of $\frac{\bar{Z}_N - \mu}{\sigma/\sqrt{N}}$ is approximately $\mathcal{N}(0,1)$.

For large N, $S \approx \sigma$ (almost surely). Then, the distribution of $\frac{\bar{Z}_N - \mu}{S/\sqrt{N}}$ is approximately $\mathcal{N}(0,1)$.

For a given confidence level γ, let b be such that $P(-b < W < b) = \gamma$, where the random variable W has the distribution $\mathcal{N}(0,1)$.

Then, $P(-b < \frac{\bar{Z}_N - \mu}{S/\sqrt{N}} < b) \approx P(-b < W < b) = \gamma$.

Since $\{-b < \frac{\bar{Z}_N - \mu}{S/\sqrt{N}} < b\}$ is the same event as $\{\bar{Z}_N - \frac{bS}{\sqrt{N}} < \mu < \bar{Z}_N + \frac{bS}{\sqrt{N}}\}$, we see that the γ-confidence interval is approximately $(\bar{Z}_N - \frac{bS}{\sqrt{N}}, \bar{Z}_N + \frac{bS}{\sqrt{N}})$.

Conclusion: For large N, even if the original data is not from a Gaussian distribution, we expect that Student's t-distribution approaches the standard normal distribution.

For large N, we expect the confidence interval computed using Student's t-distribution or the normal distribution to be about the same, regardless of whether the original data is Gaussian or not.

In our example, the 95%-confidence interval for the true root mean square error is $(45788, 49711)$ using Student's t-distribution.

It is $(45788, 49710)$ using the normal distribution.

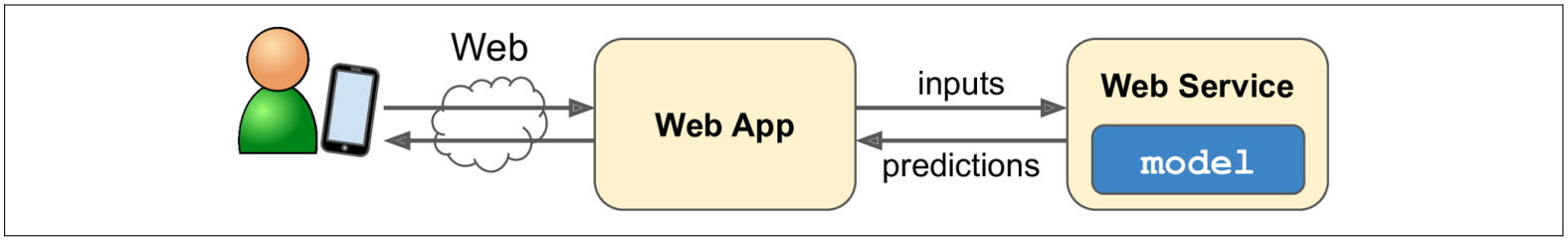
For small N, the approach of using Student's t-distribution is useful when the true variance is unknown and when data are drawn from a Gaussian distribution.

Prelaunch: Document Everything and Presentation

- At this point, you should document everything and present your solution.
- To a technical audience and yourself: high-light what you have learned, what worked and what did not, what assumptions were made, and what your system's limitations are.
- To a general audience or management: create nice presentations with clear visualizations and easy-to-remember statements (e.g., "the median income is the number one predictor of housing prices").

Launch

- You now need to get your solution ready for production (e.g., polish the code, write documentation and tests, etc.).
- Deploy your model to your production environment.
- One way to do this is to save the trained Scikit-Learn model (e.g., using joblib), including the full preprocessing and prediction pipeline, then load this trained model within your production environment and use it to make predictions by calling its `predict()` method.



- One popular strategy is to deploy your model on the cloud, for example on Google Cloud AI Platform:
  - Save your model using `joblib` and upload it to Google Cloud Storage (GCS).
  - Next, go to Google Cloud AI Platform and create a new model version, pointing it to the GCS file.
  - This gives you a simple web service that takes care of load balancing and scaling for you.
  - You can then use this web service in your website.
  - It take JSON requests containing the input data (e.g., of a district) and returns JSON responses containing the predictions.

Monitor Performance

- You need to write monitoring code to check your system's live performance at regular intervals and trigger alerts when it drops.
- Performance drop may be steep, likely due to a broken component in your infrastructure.
- More interestingly, it could also be a gentle decay that could easily go unnoticed for a long time, because models tend to "rot" over time. Why?

  The world changes. A model trained with last year's data may not work well for today's data.

  Even a model trained to classify pictures of cats and dogs may deteriorate in performance.
- If the data keeps evolving, you will need to update your datasets and retrain your model regularly. You should automate the whole process as much as possible.