Model evaluation

- Scoring
- Tools

Regression metrics

- The `sklearn.metrics` module implements several loss, score, and utility functions to measure regression performance.
- A lot of those have been enhanced to handle the `multioutput` case. e.g `.mean_squared_error`, `mean_absolute_error`, `explained_variance_score` and `r2_score`

Multioutput scoring

Needs to specify the way the scores or losses for each individual target should be averaged.

- The default is `uniform_average`, which specifies a uniformly weighted mean over outputs.
- If an ndarray of shape (`n_outputs,`) is passed, then its entries are interpreted as weights and an according weighted average is returned.
- If multioutput is `raw_values` is specified, then all unaltered individual scores or losses will be returned in an array of shape (`n_outputs,`).

The `r2_score` and `explained_variance_score` accept an additional value `variance_weighted` for the `multioutput` parameter.

- This option leads to a weighting of each individual score by the variance of the corresponding target variable.
- If the target variables are of different scale, then this score puts more importance on well explaining the higher variance variables.

Let's study these metrics one by one.

Explained variance score

The `explained_variance_score` computes the explained variance regression score.

If $\hat{\mathbf{y}}$ is the estimated target output, $\mathbf{y}$ the corresponding (correct) target output, and `Var` is Variance, the square of the standard deviation, then the explained variance is estimated as follow:

$$\text{explained\_var} = 1 - \frac{\text{Var}\left(\mathbf{y} - \hat{\mathbf{y}}\right)}{\text{Var}(\mathbf{y})}$$

Example with single output regression

```
from sklearn.metrics import explained_variance_score
y_true = [3, -0.5, 2, 7]
y_pred = [2.5, 0.0, 2, 8]
explained_variance_score(y_true, y_pred)
```

```
0.9571734475374732
```

Double-click (or enter) to edit

Example with multioutput regression

```
y_true = [[0.5, 1], [-1, 1], [7, -6]]
y_pred = [[0, 2], [-1, 2], [8, -5]]
explained_variance_score(y_true, y_pred, multioutput='raw_values')
```

```
array([0.96774194, 1.        ])
```

Weighted average by passing the weights in `multioutput` parameter.

```
explained_variance_score(y_true, y_pred, multioutput=[0.3, 0.7])
```

```
0.9903225806451612
```

Max-error

The `max_error` function computes the maximum residual error, a metric that captures the worst case error between the predicted value and the true value.

In a perfectly fitted single output regression model, max_error would be 0 on the training set and though this would be highly unlikely in the real world, this metric shows the extent of error that the model had when it was fitted.

If $\hat{y}^{(i)}$ is the predicted value of the i-th sample, and $y^{(i)}$ is the corresponding true value, then the max error is defined as

$$\text{Max error}(\mathbf{y}, \hat{\mathbf{y}}) = \max\left(|y^{(i)} - \hat{y}^{(i)}|\right)$$

Here is a small example of usage of the max_error function:

```
from sklearn.metrics import max_error
y_true = [3, 2, 7, 1]
y_pred = [9, 2, 7, 1]
max_error(y_true, y_pred)
```

```
6
```

The `max_error` does not support multioutput.

Mean absolute error (MAE)

The `mean_absolute_error` function computes mean absolute error, a risk metric corresponding to the expected value of the absolute error loss or l_1-norm loss.

$$\text{MAE}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{n} \sum_{i=1}^{n} |\mathbf{y}^{(i)} - \hat{\mathbf{y}}^{(i)}|$$

Single output

```
from sklearn.metrics import mean_absolute_error
y_true = [3, -0.5, 2, 7]
y_pred = [2.5, 0.0, 2, 8]
mean_absolute_error(y_true, y_pred)
```

```
0.5
```

Multi-output: `Uniform average` (Default)

```
y_true = [[0.5, 1], [-1, 1], [7, -6]]
y_pred = [[0, 2], [-1, 2], [8, -5]]
mean_absolute_error(y_true, y_pred)
```

```
0.75
```

Multi-output: raw

```
mean_absolute_error(y_true, y_pred, multioutput='raw_values')
```

```
array([0.5, 1. ])
```

Multi-output weighted

```
mean_absolute_error(y_true, y_pred, multioutput=[0.3, 0.7])
```

Mean squared error (MSE)

The `mean_squared_error` function computes mean square error, a risk metric corresponding to the expected value of the squared (quadratic) error or loss.

$$\text{MSE}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{n}\sum_{i=1}^{n}\left(\mathbf{y}^{(i)} - \hat{\mathbf{y}}^{(i)}\right)^2$$

Single output

```
from sklearn.metrics import mean_squared_error
y_true = [3, -0.5, 2, 7]
y_pred = [2.5, 0.0, 2, 8]
mean_squared_error(y_true, y_pred)
```

```
0.375
```

Multioutput: `uniform_average` (default)

```
y_true = [[0.5, 1],[-1, 1],[7, -6]]
y_pred = [[0, 2],[-1, 2],[8, -5]]
mean_squared_error(y_true, y_pred)
```

```
0.7083333333333334
```

Multioutput: raw output

```
mean_squared_error(y_true, y_pred, multioutput='raw_values')
```

```
array([0.41666667, 1.        ])
```

Multioutput: weighted

```
mean_squared_error(y_true, y_pred, multioutput=[0.3, 0.7])
```

```
0.825
```

Root mean squared error (RMSE)

By setting `squared` argument in `mean_squared_error` metric to `False`, we obtain root mean squared error.

Single output RMSE:

```
y_true = [3, -0.5, 2, 7]
y_pred = [2.5, 0.0, 2, 8]
mean_squared_error(y_true, y_pred, squared=False)
```

```
0.6123724356957945
```

Multioutput RMSE with default `uniform_average`:

```
y_true = [[0.5, 1],[-1, 1],[7, -6]]
y_pred = [[0, 2],[-1, 2],[8, -5]]

mean_squared_error(y_true, y_pred, squared=False)
```

```
0.8416254115301732
```

Multioutput: Raw RMSE values

```
mean_squared_error(y_true, y_pred, squared=False, multioutput='raw_values')
```

```
array([0.41666667, 1.        ])
```

Multioutput: Weighted RMSE values

```
mean_squared_error(y_true, y_pred, squared=False, multioutput=[0.3, 0.7])
```

```
0.9082951062292475
```

R² score (coefficient of determination)

The `r2_score` function computes the coefficient of determination, usually denoted as R².

- It represents the proportion of variance (of $\mathbf{y}$) that has been explained by the independent variables in the model.
- It provides an indication of goodness of fit and therefore a measure of how well unseen samples are likely to be predicted by the model, through the proportion of explained variance.
- As such variance is dataset dependent, R² may not be meaningfully comparable across different datasets.
- Best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse).

- A constant model that always predicts the expected value of y, disregarding the input features, would get a R² score of 0.0.

It is computed as

$$R^2(\mathbf{y}, \hat{\mathbf{y}}) = 1 - \frac{\sum_{i=1}^{n} \left(\mathbf{y}^{(i)} - \hat{\mathbf{y}}^{(i)} \right)^2}{\sum_{i=1}^{n} \left(\mathbf{y}^{(i)} - \bar{\mathbf{y}}^{(i)} \right)^2}$$

where, $\bar{y} = \frac{1}{n} \sum_{i=1}^{n} \mathbf{y}^{(i)}$

Example 1: Single output

```
from sklearn.metrics import r2_score
y_true = [3, -0.5, 2, 7]
y_pred = [2.5, 0.0, 2, 8]
r2_score(y_true, y_pred)
```

```
0.9486081370449679
```

Example 2: Multi output - `variance_weighted` leads to a weighting of each individual score by the variance of the corresponding target variable.

```
y_true = [[0.5, 1], [-1, 1], [7, -6]]
y_pred = [[0, 2], [-1, 2], [8, -5]]
r2_score(y_true, y_pred, multioutput='variance_weighted')
```

Example 3: Multi output - `uniform_average`

```
y_true = [[0.5, 1], [-1, 1], [7, -6]]
y_pred = [[0, 2], [-1, 2], [8, -5]]
r2_score(y_true, y_pred, multioutput='uniform_average')
```

Example 4: Multi output - `raw_values`

```
r2_score(y_true, y_pred, multioutput='raw_values')
```

Example 5: Multi output - weighted

```
r2_score(y_true, y_pred, multioutput=[0.3, 0.7])
```

There are a few other regression metrics that are included in `sklearn`. We encourage you to check them out:

- `mean_squared_logarithmic_error` is best used when targets have exponential growth, such as population counts, average sales of a commodity over a span of years etc. Note that this metric penalizes an under-predicted estimate greater than an over-predicted estimate.
- `mean_absolute_percentage_error` is sensitive to relative error.
- `median_absolute_error` is robust to outliers. It does not support multioutput case

Model quality scoring

`sklearn` provides three different APIs for evaluating the quality of model prediction:

- **`score` method** of estimators with default evaluation criteria.
- **Scoring parameter** with model-evaluation tools using cv e.g. `model_selection.cross_val_score`.
- **Metric functions** implemented in `sklearn.metrics`.

`sklearn` also provides **Dummy estimators** for creating a baseline model.

Scoring parameter

Model selection and evaluation methods like `model_selection.GridSearchCV` and `model_selection.cross_val_score` take a `scoring` parameter that specifies metric for evaluating the estimator.

Predefined values

- Uses scorer object.
- Scorer object convention: *Higher return values are better than the lower values.*
- The error metrics are available as negated version of the metric. For example, `metrics.mean_squared_error` is available as `neg_mean_squared_error`.

scoring	Function
explained_variance	metrics.explained_variance_score
max_error	metrics.max_error
neg_mean_absolute_error	metrics.mean_absolute_error
neg_mean_squared_error	metrics.mean_squared_error
neg_root_mean_squared_error	metrics.mean_squared_error
neg_mean_squared_log_error	metrics.mean_squared_log_error
neg_median_absolute_error	metrics.median_absolute_error
r2	metrics.r2_score

Scoring strategy from metric functions

`sklearn.metrics` exposes a set of simple functions measuring a prediction error given ground truth and prediction:

- functions ending with `_score` follows convention of higher the better.
- functions ending with `_error` follows convention of lower the better.
    - While converting into a scorer object using `make_scorer`, set `greater_is_better` parameter to `False`.

Many metrics are given names to be used as `scoring` value.

- Generate appropriate scoring objects using `make_scorer`.

Note on `make_scorer`

- Makes a scorer from a performance metric or loss function.
- It's a factory function that wraps scoring functions for use in model evaluation and selection.
- It takes a score function, such as `accuracy_score`, `mean_squared_error` and returns a callable that scores an estimator's output.
- The signature of the call is (`estimator, X, y`) where
    - `estimator` is the model to be evaluated,
    - `X` is the data and `y` is the ground truth labeling (or `None` in the case of unsupervised models).

Scorer from existing metrics

One typical use case is to wrap an existing metric functions with non-default value:

`[ ]` *1 cell hidden*

Custom scorer

`sklearn` provides a way to build a completely custom scorer object use `make_scorer`, that can take several parameters:

- Python function for custom scoring
- whether the python function returns a score (`greater_is_better=True`, the default) or a loss (`greater_is_better=False`).

    - In case of a loss, the output of the python function is negated by the scorer object, conforming to the cross validation convention that scorers return higher values for better models.
- *for classification metrics only*: whether the python function requires continuous decision certainties (`needs_threshold=True`). The default value is `False`.
- Any additional parameters

[ ] ↳ *1 cell hidden*

Implement own scoring object

- `sklearn` provide a way to write flexible model scorer by constructing your own scoring objects from scratch without using `make_scorer`.
- Needs to adhere to the protocols specified by the following two rules:
    - It can be called with parameters (`estimator, X, y`), where
        - `estimator` is the model for evaluation,
        - `X` is validation data, and `y` is the ground truth target for `X` (in the supervised case) or `None` (in the unsupervised case).
- It returns a floating point number that quantifies the prediction quality of `estimator` on `X`, with reference to `y`.

> Again, by convention *higher numbers are better*, so if your scorer returns loss, that value should be negated.

Using multiple metrics for evaluation

`sklearn` provides a way to use multiple metrics for model evaluation and selection.

Three ways to specify multiple scoring metrics for `scoring` parameter:

[ ] ↳ *6 cells hidden*

Dummy regressor

When doing supervised learning, a simple sanity check consists of comparing one's estimator against simple rules of thumb.

`DummyRegressor` implements four simple rules of thumb or strategies for regression:

- `mean` always predicts the mean of the training targets.

- `median` always predicts the median of the training targets.
- `quantile` always predicts a user provided quantile of the training targets.
- `constant` always predicts a constant value that is provided by the user.

In all these strategies, the predict method completely ignores the input data.

It is useful as a simple baseline to compare with other (real) regressors. **Do not use it for real problems.**

```
import numpy as np
from sklearn.dummy import DummyRegressor
X = np.array([1.0, 2.0, 3.0, 4.0])
y = np.array([2.0, 3.0, 5.0, 10.0])
dummy_regr = DummyRegressor(strategy="mean")
dummy_regr.fit(X, y)
dummy_regr.predict(X)
dummy_regr.score(X, y)
```

```
0.0
```

When the accuracy of regressor is too close to dummy regressor, it probably means that something went wrong: features are not helpful, or a hyperparameter is not correctly tuned.

CV: Model performance evaluation

As you may recall, in k-fold cross-validation scheme, we partition the training data into k partitions, train with $k-1$ of those and validate the model performance on the remaining partition.

The performance measure reported by k-fold cross-validation is then the average of the values computed in the loop.

This approach can be computationally expensive, but does not waste too much data (as is the case when fixing an arbitrary validation set).

The final evaluation is performed on the test set.

**Note to slidemaker: Add figure https://scikit-learn.org/stable/_images/grid_search_cross_validation.png in the slide deck.

Let's first check out `sklearn` utilities for splitting dataset according to different cross validation strategies.

In regression setup, we made an iid assumption about the data. (Independent and identically distributed).

It means all samples are drawn from the same generative process and that the generative process is assumed to have no memory of past generated samples.

`sklearn` provides cross validation utilities for non-iid data too. We won't be covering it in this course. However we are listing it down here for your reference:

- Time series aware cross validation for samples generated using a time dependent process.
- Group-wise cross validation for samples generated from a process with a group structure (samples collected from different subjects, experiments, measurement devices).

k-fold

- `KFold` divides all the samples in groups of samples, called folds (if $k = n$, this is equivalent to the Leave One Out strategy), of equal sizes (if possible).
- The prediction function is learned using $k - 1$ folds, and the fold left out is used for test.

```
# This is am example of 2 fold cv on a dataset with 4 samples.
import numpy as np
from sklearn.model_selection import KFold

X = ["a", "b", "c", "d"]
kf = KFold(n_splits=2)

split = 1
for train, test in kf.split(X):
    print("Split #%d, Train fold: %s | Test fold: %s" % (split, train, test))
    split += 1
```

```
 Split #1, Train fold: [2 3] | Test fold: [0 1]
 Split #2, Train fold: [0 1] | Test fold: [2 3]
```

The training and test set can be constructed with `train` and `test` indices.

```
X = np.array([[0., 0.], [1., 1.], [-1., -1.], [2., 2.]])
y = np.array([0, 1, 0, 1])

kf = KFold(n_splits=2)
for train, test in kf.split(X):
  X_train, X_test, y_train, y_test = X[train], X[test], y[train], y[test]
  print ('START OF FOLD')
  print ('X_train: %s'% X_train)
  print ('y_train: %s'% y_train)
  print ('X_test: %s'% X_test)
  print ('X_test: %s'% X_test)
  print ('END OF FOLD\n')
```

```
   START OF FOLD
   X_train: [[-1. -1.]
```

```
 [ 2.  2.]]
y_train: [0 1]
X_test: [[0. 0.]
 [1. 1.]]
X_test: [[0. 0.]
 [1. 1.]]
END OF FOLD

START OF FOLD
X_train: [[0. 0.]
 [1. 1.]]
y_train: [0 1]
X_test: [[-1. -1.]
 [ 2.  2.]]
X_test: [[-1. -1.]
 [ 2.  2.]]
END OF FOLD
```

Repeated k-fold

`RepeatedKFold` repeats K-Fold `n_repeats` times.

It can be used when one requires to run KFold `n_repeats` times, producing different splits in each repetition.

```
# Example of 2-fold K-Fold repeated 2 times:
import numpy as np
from sklearn.model_selection import RepeatedKFold

X = np.array([[1, 2], [3, 4], [1, 2], [3, 4]])
random_state = 42

rkf = RepeatedKFold(n_splits=2, n_repeats=2, random_state=random_state)

split = 1
for train, test in rkf.split(X):
    print("split #%d, %s %s" % (split, train, test))
    split += 1
```

```
 split #1, [0 2] [1 3]
 split #2, [1 3] [0 2]
 split #3, [0 2] [1 3]
 split #4, [1 3] [0 2]
```

Leave one out (LOO)

LeaveOneOut (or LOO) is a simple cross-validation.

Each learning set is created by taking all the samples except one, the test set being the sample left out. Thus, for samples, we have different training sets and different tests set.

This cross-validation procedure does not waste much data as only one sample is removed from the training set:

```
from sklearn.model_selection import LeaveOneOut

X = [1, 2, 3, 4]
loo = LeaveOneOut()

split = 1
for train, test in loo.split(X):
    print("split# %d, %s %s" % (split, train, test))
    split += 1
```

```
 split# 1, [1 2 3] [0]
 split# 2, [0 2 3] [1]
 split# 3, [0 1 3] [2]
 split# 4, [0 1 2] [3]
```

There are a few caveats that we should be aware of:

When compared with k-fold cross validation,

- In terms of the number of models:
  - We builds n models from n samples instead of k models, where $n > k$.
  - Moreover, each is trained on $(n-1)$ samples rather than $(k-1)n/k$. In both ways, assuming k is not too large and $k < n$, LOO is more computationally expensive than k-fold cross validation.
- In terms of accuracy,
  - LOO often results in high variance as an estimator for the test error.
  - Intuitively, since $n-1$ of the n samples are used to build each model, models constructed from folds are virtually identical to each other and to the model built from the entire training set.

> As a general rule, most authors, and empirical evidence, suggest that 5- or 10- fold cross validation should be preferred to LOO.

Leave P out (LPO)

`LeavePOut` is very similar to `LeaveOneOut` as it creates all the possible training/test sets by removing p samples from the complete set.

- For n samples, this produces $\binom{n}{p}$ train-test pairs.
- Unlike `LeaveOneOut` and `KFold`, the test sets will overlap for $p > 1$.

Example of Leave-2-Out on a dataset with 4 samples:

```
from sklearn.model_selection import LeavePOut

X = np.ones(4)
lpo = LeavePOut(p=2)

split = 1
for train, test in lpo.split(X):
    print("split #%d, %s %s" % (split, train, test))
    split += 1
```

```
 split #1, [2 3] [0 1]
 split #2, [1 3] [0 2]
 split #3, [1 2] [0 3]
 split #4, [0 3] [1 2]
 split #5, [0 2] [1 3]
 split #6, [0 1] [2 3]
```

Random permutation (shuffle and split)

The `ShuffleSplit` iterator will generate a user defined number of independent train/test dataset splits.

- Samples are first shuffled and then split into a pair of train and test sets.
- It is possible to control the randomness for reproducibility of the results by explicitly seeding the random_state pseudo random number generator.

```
from sklearn.model_selection import ShuffleSplit
X = np.arange(10)
ss = ShuffleSplit(n_splits=5, test_size=0.25, random_state=0)

split = 1
for train_index, test_index in ss.split(X):
    print("split #%d, %s %s" % (split, train_index, test_index))
    split += 1
```

```
 split #1, [9 1 6 7 3 0 5] [2 8 4]
 split #2, [2 9 8 0 6 7 4] [3 5 1]
 split #3, [4 5 1 0 6 9 7] [2 3 8]
 split #4, [2 7 5 8 0 3 4] [6 1 9]
 split #5, [4 1 0 6 8 9 3] [5 2 7]
```

`ShuffleSplit` is a good alternative to `KFold` cross validation that allows a finer control on the number of iterations and the proportion of samples on each side of the train / test split.

CV metrics

Scoring by CV: `cross_val_score`

`cross_val_score` is the simplest way to evaluate a score by cv.

It has the following important arguments:

1. `estimator` object implementing `fit`.
2. `X`: feature matrix of shape (n, m)
3. `y`: label vector of shape (n,) or (n, k) for multi-output
4. `scoring` parameter to specify how score should be calculated. If `None` the default scorer of estimator is used.
5. `cv`
    - `None`: uses 5-fold cv.
    - int, to specify # of folds in `KFold`.
    - CV splitter
    - Iterable yielding (train, test) splits of array indices
6. `error_score` is the value to assign to the score if an error occurs in fitting.
    - `raise` - error is raised
    - numeric - `FitFailedWarning` is used

`[ ]` *7 cells hidden*

Multiple metric evaluation: `cross_validate`

- Many parameters listed in `cross_val_score` are applicable here too.
- It however differs in two ways with `cross_val_score`:
    - Allows multiple metric specification
    - Returns a dict containing fit-times, scores-times in addition to test scores.
- The multiple metrics can be specified either as a list, tuple or set of predefined scorer names, Or as a dict mapping scorer name to a predefined or custom scoring function.
- Return dict for *single metric evaluation* contains the following keys: ['test_score', 'fit_time', 'score_time']
- And for multiple metric evaluation, the return value is a dict with the following keys - ['test_', 'test_', 'test_<scorer...>', 'fit_time', 'score_time']
- `return_train_score` is used to specify if we desire to evaluate the scores on training set.
    - `True` evaluates score on training set.
    - `False` [default] does not evaluate on training set. It leads to saving of computation time.
- `return_estimator` parameters helps us to specify if we want to retain the estimator fitted on each training set in cv.

[ ] ↳ *8 cells hidden*

Predictions by CV: `cross_val_predict`

- The function `cross_val_predict` has a similar interface to `cross_val_score`, but returns, for each element in the input, the prediction that was obtained for that element when it was in the test set.
- Only cross-validation strategies that assign all elements to a test set exactly once can be used (otherwise, an exception is raised).

↳ *2 cells hidden*