10. Common pitfalls and recommended practices

The purpose of this chapter is to illustrate some common pitfalls and anti-patterns that occur when using scikit-learn. It provides examples of what **not** to do, along with a corresponding correct example.

10.1. Inconsistent preprocessing

scikit-learn provides a library of Dataset transformations, which may clean (see Preprocessing data), reduce (see Unsupervised dimensionality reduction), expand (see Kernel Approximation) or generate (see Feature extraction) feature representations. If these data transforms are used when training a model, they also must be used on subsequent datasets, whether it's test data or data in a production system. Otherwise, the feature space will change, and the model will not be able to perform effectively.

For the following example, let's create a synthetic dataset with a single feature:

```
>>> from sklearn.datasets import make_regression
>>> from sklearn.model_selection import train_test_split

>>> random_state = 42
>>> X, y = make_regression(random_state=random_state, n_features=1, noise=1)
>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, test_size=0.4, random_state=random_state)
```

Wrong

The train dataset is scaled, but not the test dataset, so model performance on the test dataset is worse than expected:

```
>>> from sklearn.metrics import mean_squared_error
>>> from sklearn.linear_model import LinearRegression
>>> from sklearn.preprocessing import StandardScaler

>>> scaler = StandardScaler()
>>> X_train_transformed = scaler.fit_transform(X_train)
>>> model = LinearRegression().fit(X_train_transformed, y_train)
>>> mean_squared_error(y_test, model.predict(X_test))
62.80...
```

Right

Instead of passing the non-transformed `X_test` to `predict`, we should transform the test data, the same way we transformed the training data:

```
>>> X_test_transformed = scaler.transform(X_test)
>>> mean_squared_error(y_test, model.predict(X_test_transformed))
0.90...
```

Alternatively, we recommend using a `Pipeline`, which makes it easier to chain transformations with estimators, and reduces the possibility of forgetting a transformation:

```
>>> from sklearn.pipeline import make_pipeline

>>> model = make_pipeline(StandardScaler(), LinearRegression())
>>> model.fit(X_train, y_train)
Pipeline(steps=[('standardscaler', StandardScaler()),
                ('linearregression', LinearRegression())])
>>> mean_squared_error(y_test, model.predict(X_test))
0.90...
```

Pipelines also help avoiding another common pitfall: leaking the test data into the training data.

10.2. Data leakage

Data leakage occurs when information that would not be available at prediction time is used when building the model. This results in overly optimistic performance estimates, for example from cross-validation, and thus poorer performance when the model is used on actually novel data, for example during production.

A common cause is not keeping the test and train data subsets separate. Test data should never be used to make choices about the model. **The general rule is to never call `fit` on the test data**. While this may sound obvious, this is easy to miss in some cases, for example when applying Toggle Menu ocessing steps.

Although both train and test data subsets should receive the same preprocessing transformation (as described in the previous section), it is important that these transformations are only learnt from the training data. For example, if you have a normalization step where you divide by the average value, the average should be the average of the train subset, **not** the average of all the data. If the test subset is included in the average calculation, information from the test subset is influencing the model.

An example of data leakage during preprocessing is detailed below.

10.2.1. Data leakage during pre-processing

Note: We here choose to illustrate data leakage with a feature selection step. This risk of leakage is however relevant with almost all transformations in scikit-learn, including (but not limited to) `StandardScaler`, `SimpleImputer`, and `PCA`.

A number of Feature selection functions are available in scikit-learn. They can help remove irrelevant, redundant and noisy features as well as improve your model build time and performance. As with any other type of preprocessing, feature selection should **only** use the training data. Including the test data in feature selection will optimistically bias your model.

To demonstrate we will create this binary classification problem with 10,000 randomly generated features:

```
>>> import numpy as np
>>> n_samples, n_features, n_classes = 200, 10000, 2
>>> rng = np.random.RandomState(42)
>>> X = rng.standard_normal((n_samples, n_features))
>>> y = rng.choice(n_classes, n_samples)
```

Wrong

Using all the data to perform feature selection results in an accuracy score much higher than chance, even though our targets are completely random. This randomness means that our `X` and `y` are independent and we thus expect the accuracy to be around 0.5. However, since the feature selection step 'sees' the test data, the model has an unfair advantage. In the incorrect example below we first use all the data for feature selection and then split the data into training and test subsets for model fitting. The result is a much higher than expected accuracy score:

```
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.feature_selection import SelectKBest
>>> from sklearn.ensemble import GradientBoostingClassifier
>>> from sklearn.metrics import accuracy_score

>>> # Incorrect preprocessing: the entire data is transformed
>>> X_selected = SelectKBest(k=25).fit_transform(X, y)

>>> X_train, X_test, y_train, y_test = train_test_split(
...     X_selected, y, random_state=42)
>>> gbc = GradientBoostingClassifier(random_state=1)
>>> gbc.fit(X_train, y_train)
GradientBoostingClassifier(random_state=1)

>>> y_pred = gbc.predict(X_test)
>>> accuracy_score(y_test, y_pred)
0.76
```

Right

To prevent data leakage, it is good practice to split your data into train and test subsets **first**. Feature selection can then be formed using just the train dataset. Notice that whenever we use `fit` or `fit_transform`, we only use the train dataset. The score is now what we would expect for the data, close to chance:

```
>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, random_state=42)
>>> select = SelectKBest(k=25)
>>> X_train_selected = select.fit_transform(X_train, y_train)

>>> gbc = GradientBoostingClassifier(random_state=1)
>>> gbc.fit(X_train_selected, y_train)
GradientBoostingClassifier(random_state=1)

>>> X_test_selected = select.transform(X_test)
>>> y_pred = gbc.predict(X_test_selected)
>>> accuracy_score(y_test, y_pred)
0.46
```

Here again, we recommend using a `Pipeline` to chain together the feature selection and model estimators. The pipeline ensures that only the training data is used when performing `fit` and the test data is used only for calculating the accuracy score:

Toggle Menu

```
>>> from sklearn.pipeline import make_pipeline
>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, random_state=42)
>>> pipeline = make_pipeline(SelectKBest(k=25),
...                          GradientBoostingClassifier(random_state=1))
>>> pipeline.fit(X_train, y_train)
Pipeline(steps=[('selectkbest', SelectKBest(k=25)),
                ('gradientboostingclassifier',
                GradientBoostingClassifier(random_state=1))])

>>> y_pred = pipeline.predict(X_test)
>>> accuracy_score(y_test, y_pred)
0.46
```

The pipeline can also be fed into a cross-validation function such as `cross_val_score`. Again, the pipeline ensures that the correct data subset and estimator method is used during fitting and predicting:

```
>>> from sklearn.model_selection import cross_val_score
>>> scores = cross_val_score(pipeline, X, y)
>>> print(f"Mean accuracy: {scores.mean():.2f}+/-{scores.std():.2f}")
Mean accuracy: 0.45+/-0.07
```

10.2.2. How to avoid data leakage

Below are some tips on avoiding data leakage:

- Always split the data into train and test subsets first, particularly before any preprocessing steps.
- Never include test data when using the `fit` and `fit_transform` methods. Using all the data, e.g., `fit(X)`, can result in overly optimistic scores.

  Conversely, the `transform` method should be used on both train and test subsets as the same preprocessing should be applied to all the data. This can be achieved by using `fit_transform` on the train subset and `transform` on the test subset.
- The scikit-learn pipeline is a great way to prevent data leakage as it ensures that the appropriate method is performed on the correct data subset. The pipeline is ideal for use in cross-validation and hyper-parameter tuning functions.

10.3. Controlling randomness

Some scikit-learn objects are inherently random. These are usually estimators (e.g. `RandomForestClassifier`) and cross-validation splitters (e.g. `KFold`). The randomness of these objects is controlled via their `random_state` parameter, as described in the Glossary. This section expands on the glossary entry, and describes good practices and common pitfalls w.r.t. this subtle parameter.

Note: Recommendation summary
For an optimal robustness of cross-validation (CV) results, pass `RandomState` instances when creating estimators, or leave `random_state` to `None`. Passing integers to CV splitters is usually the safest option and is preferable; passing `RandomState` instances to splitters may sometimes be useful to achieve very specific use-cases. For both estimators and splitters, passing an integer vs passing an instance (or `None`) leads to subtle but significant differences, especially for CV procedures. These differences are important to understand when reporting results.

For reproducible results across executions, remove any use of `random_state=None`.

10.3.1. Using `None` or `RandomState` instances, and repeated calls to `fit` and `split`

The `random_state` parameter determines whether multiple calls to fit (for estimators) or to split (for CV splitters) will produce the same results, according to these rules:

- If an integer is passed, calling `fit` or `split` multiple times always yields the same results.
- If `None` or a `RandomState` instance is passed: `fit` and `split` will yield different results each time they are called, and the succession of calls explores all sources of entropy. `None` is the default value for all `random_state` parameters.

We here illustrate these rules for both estimators and CV splitters.

Note: Since passing `random_state=None` is equivalent to passing the global `RandomState` instance from `numpy` (`random_state=np.random.mtrand._rand`), we will not explicitly mention `None` here. Everything that applies to instances also applies to using `None`.

Estimators

Passing instances means that calling `fit` multiple times will not yield the same results, even if the estimator is fitted on the same data and with the same hyper-parameters:

Toggle Menu

```
>>> from sklearn.linear_model import SGDClassifier
>>> from sklearn.datasets import make_classification
>>> import numpy as np

>>> rng = np.random.RandomState(0)
>>> X, y = make_classification(n_features=5, random_state=rng)
>>> sgd = SGDClassifier(random_state=rng)

>>> sgd.fit(X, y).coef_
array([[ 8.85418642,  4.79084103, -3.13077794,  8.11915045, -0.56479934]])

>>> sgd.fit(X, y).coef_
array([[ 6.70814003,  5.25291366, -7.55212743,  5.18197458,  1.37845099]])
```

We can see from the snippet above that repeatedly calling `sgd.fit` has produced different models, even if the data was the same. This is because the Random Number Generator (RNG) of the estimator is consumed (i.e. mutated) when `fit` is called, and this mutated RNG will be used in the subsequent calls to `fit`. In addition, the `rng` object is shared across all objects that use it, and as a consequence, these objects become somewhat inter-dependent. For example, two estimators that share the same `RandomState` instance will influence each other, as we will see later when we discuss cloning. This point is important to keep in mind when debugging.

If we had passed an integer to the `random_state` parameter of the [`SGDClassifier`], we would have obtained the same models, and thus the same scores each time. When we pass an integer, the same RNG is used across all calls to `fit`. What internally happens is that even though the RNG is consumed when `fit` is called, it is always reset to its original state at the beginning of `fit`.

CV splitters

Randomized CV splitters have a similar behavior when a `RandomState` instance is passed; calling `split` multiple times yields different data splits:

```
>>> from sklearn.model_selection import KFold
>>> import numpy as np

>>> X = y = np.arange(10)
>>> rng = np.random.RandomState(0)
>>> cv = KFold(n_splits=2, shuffle=True, random_state=rng)

>>> for train, test in cv.split(X, y):
...     print(train, test)
[0 3 5 6 7] [1 2 4 8 9]
[1 2 4 8 9] [0 3 5 6 7]

>>> for train, test in cv.split(X, y):
...     print(train, test)
[0 4 6 7 8] [1 2 3 5 9]
[1 2 3 5 9] [0 4 6 7 8]
```

We can see that the splits are different from the second time `split` is called. This may lead to unexpected results if you compare the performance of multiple estimators by calling `split` many times, as we will see in the next section.

10.3.2. Common pitfalls and subtleties

While the rules that govern the `random_state` parameter are seemingly simple, they do however have some subtle implications. In some cases, this can even lead to wrong conclusions.

Estimators

Different `random_state` types lead to different cross-validation procedures

Depending on the type of the `random_state` parameter, estimators will behave differently, especially in cross-validation procedures. Consider the following snippet:

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> from sklearn.datasets import make_classification
>>> from sklearn.model_selection import cross_val_score
>>> import numpy as np

>>> X, y = make_classification(random_state=0)

>>> rf_123 = RandomForestClassifier(random_state=123)
>>> cross_val_score(rf_123, X, y)
array([0.85, 0.95, 0.95, 0.9 , 0.9 ])

>>> rf_inst = RandomForestClassifier(random_state=np.random.RandomState(0))
>>> cross_val_score(rf_inst, X, y)
array([0.9 , 0.95, 0.95, 0.9 , 0.9 ])
```

Toggle Menu

We see that the cross-validated scores of `rf_123` and `rf_inst` are different, as should be expected since we didn't pass the same `random_state` parameter. However, the difference between these scores is more subtle than it looks, and **the cross-validation procedures that were performed by `cross_val_score` significantly differ in each case**:

- Since `rf_123` was passed an integer, every call to `fit` uses the same RNG: this means that all random characteristics of the random forest estimator will be the same for each of the 5 folds of the CV procedure. In particular, the (randomly chosen) subset of features of the estimator will be the same across all folds.
- Since `rf_inst` was passed a `RandomState` instance, each call to `fit` starts from a different RNG. As a result, the random subset of features will be different for each folds.

While having a constant estimator RNG across folds isn't inherently wrong, we usually want CV results that are robust w.r.t. the estimator's randomness. As a result, passing an instance instead of an integer may be preferable, since it will allow the estimator RNG to vary for each fold.

Note: Here, `cross_val_score` will use a non-randomized CV splitter (as is the default), so both estimators will be evaluated on the same splits. This section is not about variability in the splits. Also, whether we pass an integer or an instance to `make_classification` isn't relevant for our illustration purpose: what matters is what we pass to the `RandomForestClassifier` estimator.

Cloning

Another subtle side effect of passing `RandomState` instances is how `clone` will work:

```
>>> from sklearn import clone
>>> from sklearn.ensemble import RandomForestClassifier
>>> import numpy as np

>>> rng = np.random.RandomState(0)
>>> a = RandomForestClassifier(random_state=rng)
>>> b = clone(a)
```

Since a `RandomState` instance was passed to `a`, `a` and `b` are not clones in the strict sense, but rather clones in the statistical sense: `a` and `b` will still be different models, even when calling `fit(X, y)` on the same data. Moreover, `a` and `b` will influence each-other since they share the same internal RNG: calling `a.fit` will consume `b`'s RNG, and calling `b.fit` will consume `a`'s RNG, since they are the same. This bit is true for any estimators that share a `random_state` parameter; it is not specific to clones.

If an integer were passed, `a` and `b` would be exact clones and they would not influence each other.

Warning: Even though `clone` is rarely used in user code, it is called pervasively throughout scikit-learn codebase: in particular, most meta-estimators that accept non-fitted estimators call `clone` internally (`GridSearchCV`, `StackingClassifier`, `CalibratedClassifierCV`, etc.).

CV splitters

When passed a `RandomState` instance, CV splitters yield different splits each time `split` is called. When comparing different estimators, this can lead to overestimating the variance of the difference in performance between the estimators:

```
>>> from sklearn.naive_bayes import GaussianNB
>>> from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
>>> from sklearn.datasets import make_classification
>>> from sklearn.model_selection import KFold
>>> from sklearn.model_selection import cross_val_score
>>> import numpy as np

>>> rng = np.random.RandomState(0)
>>> X, y = make_classification(random_state=rng)
>>> cv = KFold(shuffle=True, random_state=rng)
>>> lda = LinearDiscriminantAnalysis()
>>> nb = GaussianNB()

>>> for est in (lda, nb):
...     print(cross_val_score(est, X, y, cv=cv))
[0.8  0.75 0.75 0.7  0.85]
[0.85 0.95 0.95 0.85 0.95]
```

Directly comparing the performance of the `LinearDiscriminantAnalysis` estimator vs the `GaussianNB` estimator **on each fold** would be a mistake: **the splits on which the estimators are evaluated are different**. Indeed, `cross_val_score` will internally call `cv.split` on the same `KFold` instance, but the splits will be different each time. This is also true for any tool that performs model selection via cross-validation, e.g. `GridSearchCV` and `RandomizedSearchCV`: scores are not comparable fold-to-fold across different calls to `search.fit`, since `cv.split` would have been called multiple times. Within a single call to `search.fit`, however, fold-to-fold comparison is possible since the search estimator only calls `cv.split` once.

For comparable fold-to-fold results in all scenarios, one should pass an integer to the CV splitter: `cv = KFold(shuffle=True, random_state=0)`.

Toggle Menu

Note: While fold-to-fold comparison is not advisable with `RandomState` instances, one can however expect that average scores allow to conclude whether one estimator is better than another, as long as enough folds and data are used.

Note: What matters in this example is what was passed to `KFold`. Whether we pass a `RandomState` instance or an integer to `make_classification` is not relevant for our illustration purpose. Also, neither `LinearDiscriminantAnalysis` nor `GaussianNB` are randomized estimators.

10.3.3. General recommendations

Getting reproducible results across multiple executions

In order to obtain reproducible (i.e. constant) results across multiple *program executions*, we need to remove all uses of `random_state=None`, which is the default. The recommended way is to declare a `rng` variable at the top of the program, and pass it down to any object that accepts a `random_state` parameter:

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> from sklearn.datasets import make_classification
>>> from sklearn.model_selection import train_test_split
>>> import numpy as np

>>> rng = np.random.RandomState(0)
>>> X, y = make_classification(random_state=rng)
>>> rf = RandomForestClassifier(random_state=rng)
>>> X_train, X_test, y_train, y_test = train_test_split(X, y,
...                                                     random_state=rng)
>>> rf.fit(X_train, y_train).score(X_test, y_test)
0.84
```

We are now guaranteed that the result of this script will always be 0.84, no matter how many times we run it. Changing the global `rng` variable to a different value should affect the results, as expected.

It is also possible to declare the `rng` variable as an integer. This may however lead to less robust cross-validation results, as we will see in the next section.

Note: We do not recommend setting the global `numpy` seed by calling `np.random.seed(0)`. See here for a discussion.

Robustness of cross-validation results

When we evaluate a randomized estimator performance by cross-validation, we want to make sure that the estimator can yield accurate predictions for new data, but we also want to make sure that the estimator is robust w.r.t. its random initialization. For example, we would like the random weights initialization of a `SGDCLassifier` to be consistently good across all folds: otherwise, when we train that estimator on new data, we might get unlucky and the random initialization may lead to bad performance. Similarly, we want a random forest to be robust w.r.t the set of randomly selected features that each tree will be using.

For these reasons, it is preferable to evaluate the cross-validation performance by letting the estimator use a different RNG on each fold. This is done by passing a `RandomState` instance (or `None`) to the estimator initialization.

When we pass an integer, the estimator will use the same RNG on each fold: if the estimator performs well (or bad), as evaluated by CV, it might just be because we got lucky (or unlucky) with that specific seed. Passing instances leads to more robust CV results, and makes the comparison between various algorithms fairer. It also helps limiting the temptation to treat the estimator's RNG as a hyper-parameter that can be tuned.

Whether we pass `RandomState` instances or integers to CV splitters has no impact on robustness, as long as `split` is only called once. When `split` is called multiple times, fold-to-fold comparison isn't possible anymore. As a result, passing integer to CV splitters is usually safer and covers most use-cases.

Toggle Menu