This wrap-up quiz uses the penguins dataset, but notice that **we do not use the traditional `Species` column** as predictive target:

```
import pandas as pd


dataset = pd.read_csv("../datasets/penguins.csv")


feature_names = [
    "Culmen Length (mm)",
    "Culmen Depth (mm)",
    "Flipper Length (mm)",
]
target_name = "Body Mass (g)"


dataset = dataset[feature_names +
[target_name]].dropna(axis="rows", how="any")
dataset = dataset.sample(frac=1,
random_state=0).reset_index(drop=True)
data, target = dataset[feature_names], dataset[target_name]
```

We therefore define our problem as a regression problem: we want to predict the body mass of a penguin given its culmen and flipper measurements.

Notice that we randomly shuffled the rows of the dataset after loading it (`dataset.sample(frac=1, random_state=0)`). The reason is to break a spurious order-related statistical dependency that would otherwise cause trouble with the naive cross-validation procedure we use in this notebook. The problem of order-dependent samples will be discussed in more detail on the model evaluation module and is outside of the scope of this quiz for now. Now, evaluate the following tree-based models:

- a decision tree regressor, i.e. `sklearn.tree.DecisionTreeRegressor`

- a random forest regressor,
  i.e. `sklearn.ensemble.RandomForestRegressor`

Use the default hyper-parameter settings for both models. The only exception is to pass `random_state=0` for all models to be sure to recover the exact same performance scores as the solutions to this quiz.

Evaluate the generalization performance of these models using a 10-fold cross-validation:

- use `sklearn.model_selection.cross_validate` to run the cross-validation routine
- set the parameter `cv=10` to use a 10-fold cross-validation strategy. Store the training score of the cross-validation by setting the parameter `return_train_score=True` in the function `cross_validate` as we will use it later on.

Question 1

(1/1 point)
By comparing the cross-validation test scores fold-to-fold, count the number of times a random forest is better than a single decision tree. Select the range which this number belongs to:

a) [0, 3]: the random forest model is substantially worse than the single decision tree regressor b) [4, 6]: both models are almost equivalent c) [7, 10]: the random forest model is substantially better than the single decision tree regressor c) [7, 10]: the random forest model is substantially better than the single decision tree regressor - correct

EXPLANATION

solution: c)

The code snippet to evaluate the models and compare them is:

```
from sklearn.model_selection import cross_validate
from sklearn.tree import DecisionTreeRegressor

tree = DecisionTreeRegressor(random_state=0)
```

```
cv=10
cv_results_tree = cross_validate(tree, data, target, cv=cv,
return_train_score=True)
cv_results_tree["test_score"].mean(),
cv_results_tree["test_score"].std()
```

Notice that we are setting `random_state=0` such that the reported results below are deterministic. This step is not strictly necessary because it will not affect the choice of the expected answers. You will see a similar pattern in the following corrections.

```
from sklearn.ensemble import RandomForestRegressor

rf = RandomForestRegressor(random_state=0)
cv_results_rf = cross_validate(rf, data, target, cv=cv,
return_train_score=True)
cv_results_rf["test_score"].mean(), cv_results_rf["test_score"].std()
print(
    "Random forest is better than a single decision tree for "
    f"{sum(cv_results_rf['test_score'] > cv_results_tree['test_score'])} "
    "CV iterations out of 10"
)
```

```
Random forest is better than a single decision tree for 10 CV iterations
out of 10
```

Hide Answer

You have used 1 of 1 submissions

Now, train and evaluate with the same cross-validation strategy a random forest with 5 decision trees and another containing 100 decision trees. Once again store the training score.

Question 2

(1/1 point)

By comparing the cross-validation test scores fold-to-fold, count the number of times a random forest with 100 decision trees is better than a random forest with 5 decision trees. Select the range which this number belongs to:

a) [0, 3]: the random forest model with 100 decision trees is substantially worse than the random forest model with 5 decision trees b) [4, 6]: both models are almost equivalent c) [7, 10]: the random forest model with 100 decision trees is substantially better than the random forest model with 5 decision trees c) [7, 10]: the random forest model with 100 decision trees is substantially better than the random forest model with 5 decision trees - correct

EXPLANATION

solution: c)

We can run the cross-validation for the two models as in the above solution:

```
rf_5_trees = RandomForestRegressor(n_estimators=5, random_state=0)
rf_100_trees = RandomForestRegressor(n_estimators=100,
random_state=0)

cv_results_rf_5_trees = cross_validate(
    rf_5_trees, data, target, cv=cv, return_train_score=True
)
cv_results_rf_100_trees = cross_validate(
```

```
    rf_100_trees, data, target, cv=cv, return_train_score=True
)
print(
    "Random forest with 100 trees is better than a random forest with 5 trees for "
    f"{sum(cv_results_rf_100_trees['test_score'] > cv_results_rf_5_trees['test_score'])}
"
    "CV iterations out of 10"
)
```

```
Random forest with 100 trees is better than a random forest with 5
trees for 9 CV iterations out of 10
```

Adding trees in the forest helps improving the generalization performance of the model.

We can also get more insights by comparing the test and training scores of each model:

```
print(
    "Scores for random forest with 5 trees: "
    f"train: {cv_results_rf_5_trees['train_score'].mean():.3f} +/- "
    f"{cv_results_rf_5_trees['train_score'].std():.3f}, "
    f"test: {cv_results_rf_5_trees['test_score'].mean():.3f} +/- "
    f"{cv_results_rf_5_trees['test_score'].std():.3f}"
)
```

```
print(
    "Scores for random forest with 100 trees: "
    f"train: {cv_results_rf_100_trees['train_score'].mean():.3f} +/- "
    f"{cv_results_rf_100_trees['train_score'].std():.3f}, "
    f"test: {cv_results_rf_100_trees['test_score'].mean():.3f} +/- "
    f"{cv_results_rf_100_trees['test_score'].std():.3f}"
)
```

```
Score for random forest with 5 trees: train: 0.950 +/- 0.003, test: 0.767
+/- 0.074
Score for random forest with 100 trees: train: 0.972 +/- 0.001, test:
0.804 +/- 0.045
```

On the model with 5 trees, the average train score was already quite high but the test scores were quite low. The performance of this small random forest is therefore primarily limited by overfitting.

In the forest with 100 trees, the train score is still high (even slightly higher), and the test scores have increased. Overfitting was reduced by adding more trees to the forest.

Hide Answer

You have used 1 of 1 submissions

Plot the validation curve of the `n_estimators` parameters defined by:

```
import numpy as np
```

```
n_estimators = np.array([1, 2, 5, 10, 20, 50, 100, 200, 500,
1_000])
```

Question 3

(1 point possible)
Select the correct statements below.

a) the **train score** decreases when `n_estimators` become large (above 500 trees) b) the **train score** reaches a plateau when `n_estimators` become large (above 500 trees) c) the **train score** increases when `n_estimators` become large (above 500 trees) d) the **test score** decreases when `n_estimators` become large (above 500 trees) e) the **test score** reaches a plateau when `n_estimators` become large (above 500 trees) f) the **test score** increases when `n_estimators` become large (above 500 trees)
a) the **train score** decreases when `n_estimators` become large (above 500 trees), d) the **test score** decreases when `n_estimators` become large (above 500 trees), - incorrect

Select all answers that apply

EXPLANATION

solution: b) e)

We can build the validation curve with `sklearn.model_selection.ValidationCurveDisplay` :

```
import numpy as np
from sklearn.model_selection import ValidationCurveDisplay

n_estimators = np.array([1, 2, 5, 10, 20, 50, 100, 200, 500,
1_000])
```

```
disp = ValidationCurveDisplay.from_estimator(
    rf,
    data,
    target,
    param_name="n_estimators",
    param_range=n_estimators,
    scoring="r2",    # this is already the default for regression
    score_name="R2 score",
    std_display_style="errorbar",
    cv=cv,
    n_jobs=2,
)

_ = disp.ax_.set(
    xlabel="Number of trees",
    title="Validation curve for Random Forest",
)
```

We observe that above 500 trees, both the train and test scores become nearly constant. This diminishing returns effect is typical of Random Forests and bagging ensembles in general.

Hide Answer

You have used 2 of 2 submissions

Repeat the previous experiment but this time, instead of choosing the default parameters for the random forest, set the parameter `max_depth=5` and build the validation curve.

Question 4

(1 point possible)
Comparing the validation curve (train and test scores) of the random forest with a full depth and the random forest with a limited depth, select the correct statements.

a) the **test score** of the random forest with a full depth is (almost) always better than the **test score** of the random forest with a limited depth b) the **train score** of the random forest with a full depth is (almost) always better than the **train score** of the random forest with a limited depth c) the gap between the train and test scores decreases when reducing the depth of the trees of the random forest d) the gap between the train and test scores increases when reducing the depth of the trees of the random forest

Select all answers that apply

EXPLANATION

Solution: b) c)

To answer the question, we need to repeat the experiment and set the depth of the trees in the random forest.

```
import matplotlib.pyplot as plt

rf_shallow = RandomForestRegressor(max_depth=5, random_state=0)

fig, axs = plt.subplots(ncols=2, figsize=(12, 7),  sharey=True)
ValidationCurveDisplay.from_estimator(
    rf,
    data,
    target,
```

```
    param_name="n_estimators",
    param_range=n_estimators,
    scoring="r2",
    score_name="R2 score",
    std_display_style="errorbar",
    cv=cv,
    n_jobs=2,
    ax=axs[0]
)
ValidationCurveDisplay.from_estimator(
    rf_shallow,
    data,
    target,
    param_name="n_estimators",
    param_range=n_estimators,
    std_display_style="errorbar",
    cv=cv,
    n_jobs=2,
    ax=axs[1],
)

axs[0].set(
    xlabel="Number of trees",
    title="Random Forest with unconstrained depth",
```

```
)
axs[1].set(
    xlabel="Number of trees",
    title="Random Forest with max_depth=5",
)
_ = fig.suptitle("Validation curve for Random Forests")
```

We see that decreasing the depth reduces the gap between the train and test scores. We also see that the random forest with limited depth has a better generalization performance for a small number of trees but becomes equivalent for higher numbers of trees. We can conclude that the random forest models with a limited depth overfit less than the random forest with fully grown trees, especially when the number of trees in the ensemble is small.

One can also observe that the limiting the depth has a significant effect on limiting the training score (ability to memorize exactly the training data) and that this effect remains important, even when increasing the size of the ensemble.

Hide Answer

You have used 2 of 2 submissions

Let us now focus at the very beginning of the validation curves, and consider the training score of a random forests with a single tree while using the default `max_depth=None` parameter setting:

```
rf_1_tree = RandomForestRegressor(n_estimators=1,
random_state=0)
```

```
cv_results_tree = cross_validate(

    rf_1_tree, data, target, cv=10, return_train_score=True

)

cv_results_tree["train_score"]
```

should return:

```
array([0.83120264, 0.83309064, 0.83195043, 0.84834224,
0.85790323,

       0.86235297, 0.84791111, 0.85183089, 0.82241954,
0.85045978])
```

The fact that this single-tree Random Forest can never reach a perfect R2 score of 1.0 on the training can be surprising.

Indeed, if you we evaluate the training accuracy of the single `DecisionTreeRegressor` one gets perfect memorization of the training data:

```
tree = DecisionTreeRegressor(random_state=0)

cv_results_tree = cross_validate(

    tree, data, target, cv=10, return_train_score=True

)

cv_results_tree["train_score"]
```

which outputs the expected perfect score:

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

Question 5

(1 point possible)
From the following statements, select the one that explains that a single-tree random forest cannot achieve perfect training scores.

a) the single tree in the random forest is trained using a bootstrap of the training set and not the training set itself (because `bootstrap=True` by

default) b) for a given feature, the single tree in the random forest uses random splits while the single decision tree uses the best splitb) for a given feature, the single tree in the random forest uses random splits while the single decision tree uses the best split - incorrect c) the random forest automatically limits the depth of the single decision tree, which prevents overfitting

EXPLANATION

solution: a)

By default, random forests train their trees with a bootstrapping procedure. Since each tree is trained on a bootstrap sample, some data points from the original training set are not seen by each individual trees in the forest. As a result the single-tree RF model does not make perfect predictions on those data-points (named out-of-bag samples) which prevents the training score to reach 1.0.

We can confirm this hypothesis by deactivating the bootstrap option and checking again the train scores (leaving all the other hyper-parameters unchanged):

```
rf_1_tree = RandomForestRegressor(n_estimators=1, bootstrap=False,
random_state=0)
cross_validate(rf_1_tree, data, target, cv=cv,
return_train_score=True)["train_score"]
```

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

In this case, we fall back to the same algorithm as the single decision tree by training on the original data and thus overfitting the original training dataset.

We can also note, that when increasing the number of trees to 5 or 10, the sub-optimal training caused by bootstrapping vanishes quickly away. Random Forests are typically trained with at least 10 trees and typically much more than that, so this effect is almost never observed in practice.

Furthermore, even with a single bootstrapped tree, an R2 score larger than 0.85 is still very high and typically much larger than the test score for the same model. As a result, even if the training score of that model is not perfect, one cannot conclude that the generalization performance of a single tree-random forest is limited by underfitting. Instead single tree models are typically limited by overfitting when there depth is not bound.

Hide Answer

You have used 1 of 1 submissions

Build a validation curve for a `sklearn.ensemble.HistGradientBoostingRegressor` varying `max_iter` as follows:

```
max_iters = np.array([1, 2, 5, 10, 20, 50, 100, 200, 500])
```

We recall that `max_iter` corresponds to the number of trees in the boosted model.

Plot the average train and test score for each value of `max_iter`.

Question 6

(1/1 point)

Select the correct statements.

a) for a small number of trees (between 5 and 10 trees), the gradient boosting model behave like the random forest algorithm: the train scores are high while the test scores are not optimum b) for a small number of trees (between 5 and 10 trees), the gradient boosting model behave differently to the random forest algorithm: both the train and test scores are small c) with a large number of trees (> 100 trees) adding more trees in the ensemble causes the gradient boosting model overfit (increasing the gap between the train score and test score) d) with a large number of trees (> 100 trees) adding more trees in the ensemble does not impact the generalization performance of the gradient boosting model
b) for a small number of trees (between 5 and 10 trees), the gradient boosting model behave differently to the random forest algorithm: both the train and test scores are small, c) with a large number of trees (> 100 trees) adding more trees in the ensemble causes the gradient boosting model overfit (increasing the gap between the train score and test score), - correct

Select all answers that apply

EXPLANATION

solution: b) c)

We can build the validation with the following curve:

```
from sklearn.ensemble import HistGradientBoostingRegressor

hgbdt = HistGradientBoostingRegressor(random_state=0)
max_iters = np.array([1, 2, 5, 10, 20, 50, 100, 200, 500])

disp = ValidationCurveDisplay.from_estimator(
    hgbdt,
    data,
    target,
```

```
    param_name="max_iter",
    param_range=max_iters,
    scoring="r2",    # note: this is already the default for regression
    score_name="R2 score",
    std_display_style="errorbar",
    cv=cv,
    n_jobs=2,
)

_ = disp.ax_.set(
    xlabel="(Maximum) number of trees",
    title="Validation curve for Histogram GBDT",
)
```

In the figure, we can clearly observe the three phase behavior "underfitting / best generalization / overfitting" of gradient boosting models. Indeed, with a low number of trees, the model has a low score for both the train and test scores. We can clearly see that the test scores are bound above by the train score, which is characteristic of underfitting models (contrary to what we previously observed on the learning curve of the RF models).

Both scores then improve until a sweet spot (~50 trees) where the test score is maximum. After this, the gradient boosting algorithm starts to overfit: the train score improves towards reaching a perfect score of 1 while the test score reduces. Indeed, the model starts to memorize specific rules only true

on the training set. These rules become detrimental on the generalization performance of the model.

Here, it shows the importance of not adding too many trees to our gradient boosting ensemble. Indeed, one can use early-stopping and monitor the performance on an internal validation set to stop adding new trees when the validation score stops improving. Here is an example to show how to do this automatically:

```
hgbdt = HistGradientBoostingRegressor(early_stopping=True,
random_state=0)
cv_results_hgbdt = cross_validate(
    hgbdt, data, target, cv=cv, return_train_score=True,
return_estimator=True
)
cv_results_hgbdt["train_score"].mean(),
cv_results_hgbdt["train_score"].std()
```

```
(0.8802093174685013, 0.009772033922083758)
```

We see that the train score is not perfect meaning that our model stopped much before adding too many trees. We can check the generalization performance to ensure that an equivalent model to the previous random forest:

```
cv_results_hgbdt["test_score"].mean(),
cv_results_hgbdt["test_score"].std()
```

```
(0.8075456252855009, 0.030400979794505564)
```

So we observe that on average, the model performs as good as a large random forest.

Finally, we can check how many trees where used for each CV iteration:

```
for idx, est in enumerate(cv_results_hgbdt["estimator"]):
    print(
        f"For CV iteration {idx + 1}, {est.n_iter_} trees were built"
    )
```

```
For CV iteration 1, 60 trees were built
For CV iteration 2, 50 trees were built
For CV iteration 3, 46 trees were built
For CV iteration 4, 29 trees were built
For CV iteration 5, 33 trees were built
For CV iteration 6, 33 trees were built
For CV iteration 7, 36 trees were built
For CV iteration 8, 31 trees were built
For CV iteration 9, 24 trees were built
For CV iteration 10, 23 trees were built
```

We therefore see that we never used more than 60 trees, before entering the overfitting zone we observed on the validation curve.

Even if this model is not stronger than a large random forest, it is smaller which means that it can be much faster to predict and will use fewer memory (RAM) on the machines where it is deployed. This is a practical advantage of Gradient Boosted Trees with early stopping over Random Forests with a large number of deep trees.