Hands-on Machine Learning

7. Ensemble Learning

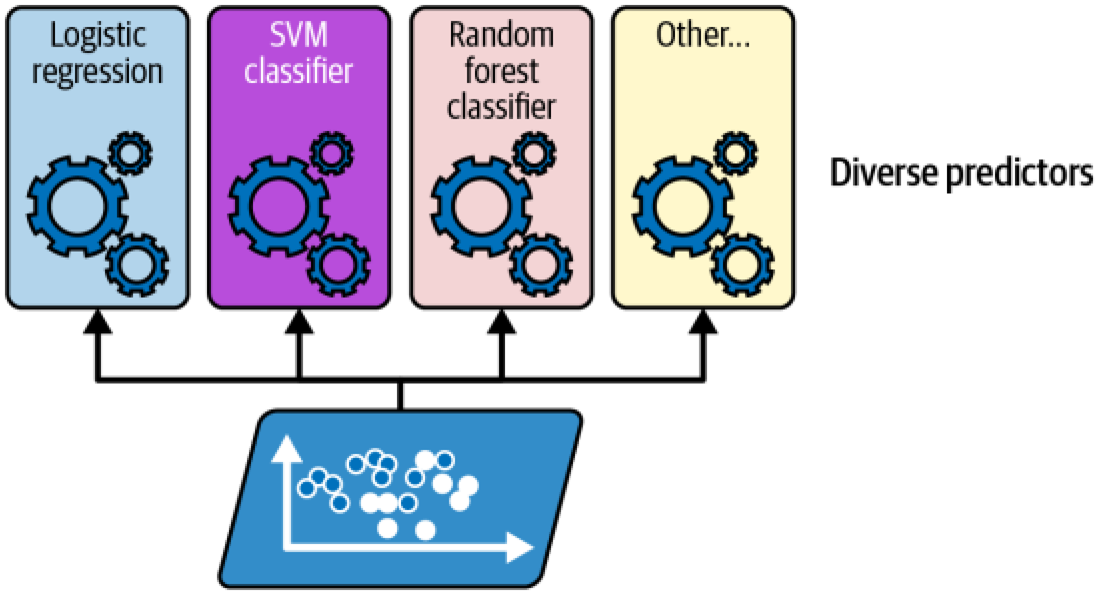
Ensemble Learning

- If you aggregate the predictions of a group of predictors, you will often get better predictions.
- A group of predictors is called an *ensemble*.
  - This technique is called *ensemble learning.*
  - An ensemble learning algorithm is called an *ensemble method.*
- Example of ensemble method:
  - Train a group of decision tree classifiers, each on a different random subset of the training set.
  - Obtain the predictions of all the individual trees, and the class that gets the most votes is the ensemble's prediction.
  - Such an ensemble of decision trees is called a *random forest*.

1.
Voting Classifers

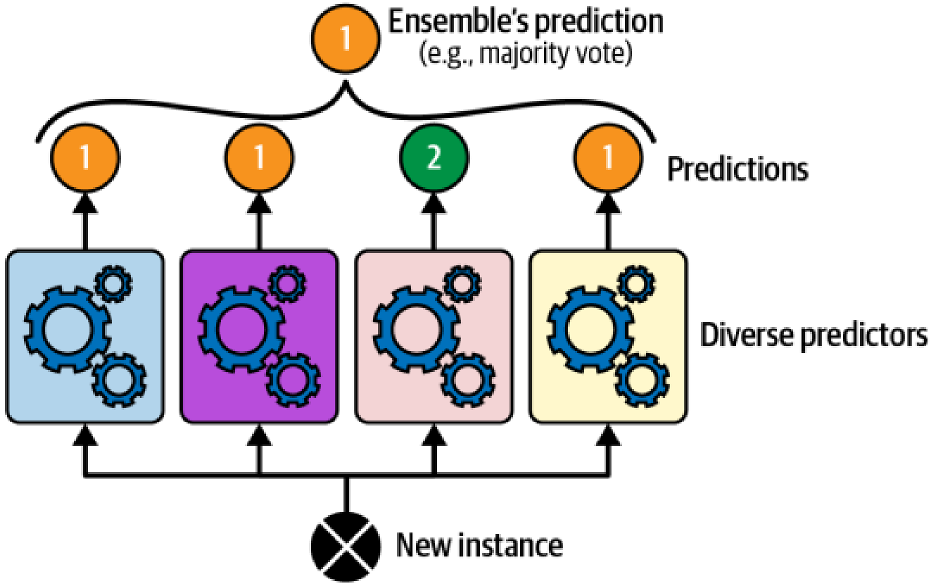
Training Diverse Classifiers

- Train diverse classifiers, and then aggregate the predictions of each classifier.



Hard Voting Classifier

- *Hard voting* classifier: a majority-vote classifier where the class that gets the most votes is the ensemble's prediction.



Why Ensemble Works?

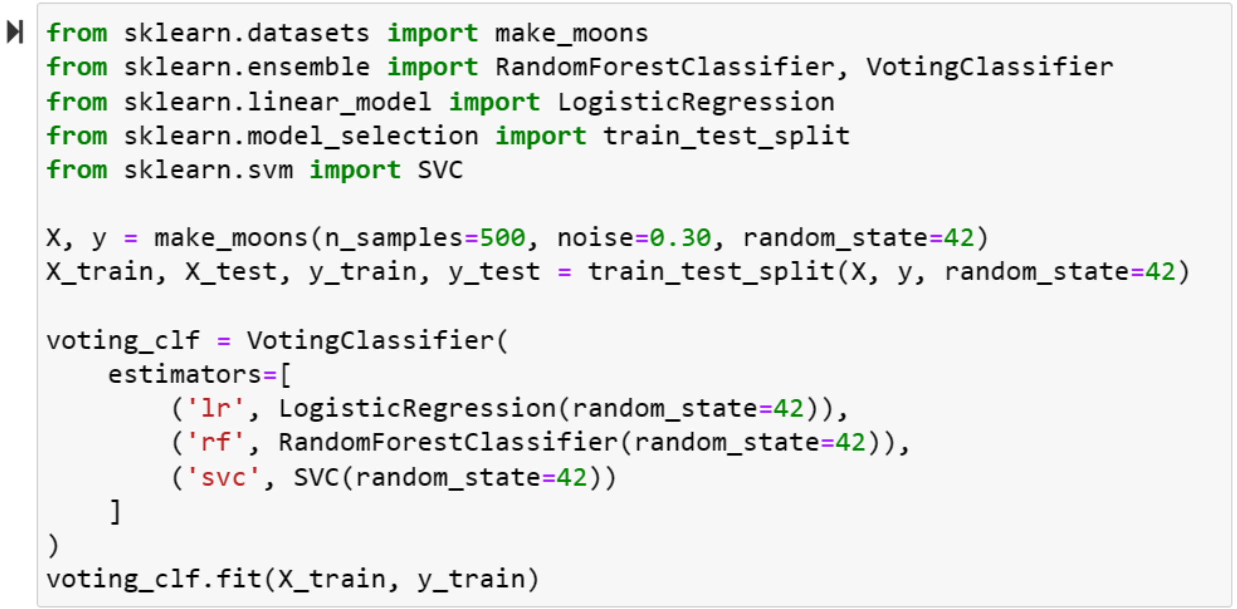
- Suppose you build an ensemble consisting of 1000 independent classifiers that are individually correct only 51% of the time (barely better than random guessing).
- If you predict the majority voted class, you can hope for close to 75% accuracy!

$$p_{correct} = \sum_{k=500}^{1000} \binom{1000}{k} (0.51)^k (0.49)^{1000-k} = 0.747$$

- Ensemble methods work best when the predictors are as independent from one another as possible.

`VotingClassifier` Class

- Scikit-Learn provides a `VotingClassifier` class: give it a list of name/predictor pairs, and use it like a normal classifier.



```python
from sklearn.datasets import make_moons
from sklearn.ensemble import RandomForestClassifier, VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC

X, y = make_moons(n_samples=500, noise=0.30, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

voting_clf = VotingClassifier(
    estimators=[
        ('lr', LogisticRegression(random_state=42)),
        ('rf', RandomForestClassifier(random_state=42)),
        ('svc', SVC(random_state=42))
    ]
)
voting_clf.fit(X_train, y_train)
```

Performance Comparison

- Each fitted classifier’s accuracy on the test set:

```
for name, clf in voting_clf.named_estimators_.items():
    print(name, "=", clf.score(X_test, y_test))
```

```
lr = 0.864
rf = 0.896
svc = 0.896
```

- The performance of the voting classifier on the test set:

```
voting_clf.score(X_test, y_test)
```

```
0.912
```

- When you call the voting classifier’s predict() method, it performs hard voting:

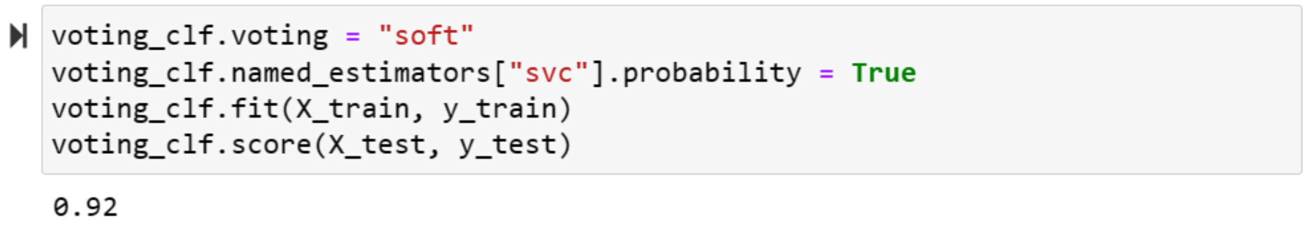
```
[clf.predict(X_test[:1]) for clf in voting_clf.estimators_]
```

```
[array([1], dtype=int64), array([1], dtype=int64), array([0], dtype=int64)]
```

Soft Voting

- *Soft voting*: if all classifiers in the ensemble are able to estimate class probabilities (i.e., they have a `predict_proba` method), then you can predict the class with the highest class probability, averaged over all the individual classifiers.

- It often achieves higher performance than hard voting because it gives more weight to highly confident votes.



```
voting_clf.voting = "soft"
voting_clf.named_estimators["svc"].probability = True
voting_clf.fit(X_train, y_train)
voting_clf.score(X_test, y_test)
```

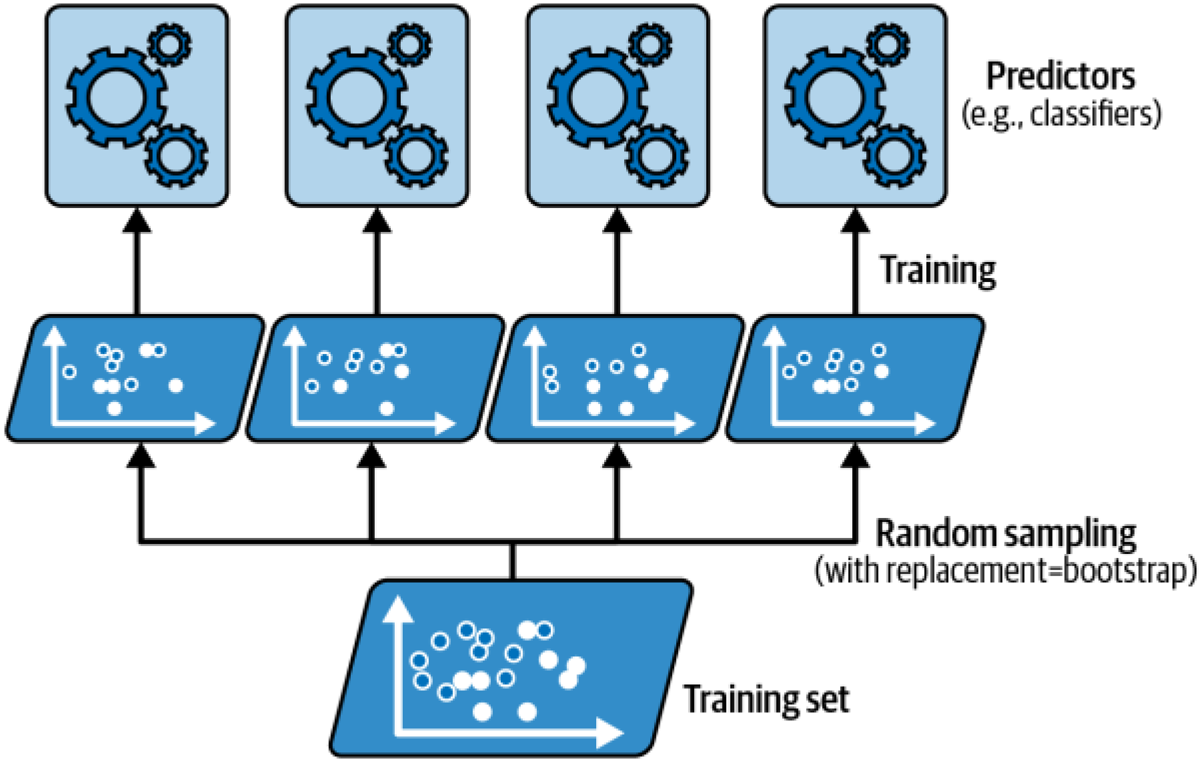
```
0.92
```

2.
Bagging and Pasting

Bagging and Pasting

- One way to get a diverse set of classifiers is to use very different training algorithms.
- Another approach is to use the same training algorithm for every predictor but train them on different random subsets of the training set.
  - When sampling is performed *with* replacement, this method is called *bagging* (short for *bootstrap aggregating*).
  - When sampling is performed *without* replacement, it is called *pasting*.

Bagging and Pasting

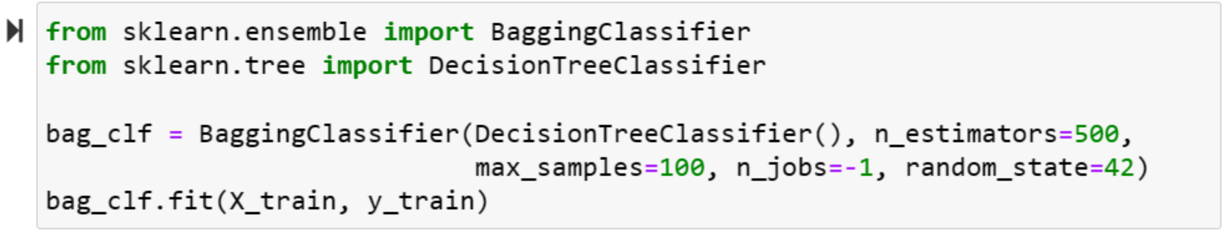


Aggregation in Bagging and Pasting

- The aggregation function is typically the *statistical mode* for classification (i.e., the most frequent prediction), or the average for regression.
- The ensemble has a similar bias but a lower variance than a single predictor trained on the original training set.
- Predictors can all be trained in parallel, via different CPU cores or even different servers.
  - Similarly, predictions can be made in parallel.
- Bagging and pasting are popular because they scale very well.

Bagging and Pasting in Scikit-Learn

- Scikit-Learn class for both bagging and pasting: `BaggingClassifier` (or `BaggingRegressor` for regression).



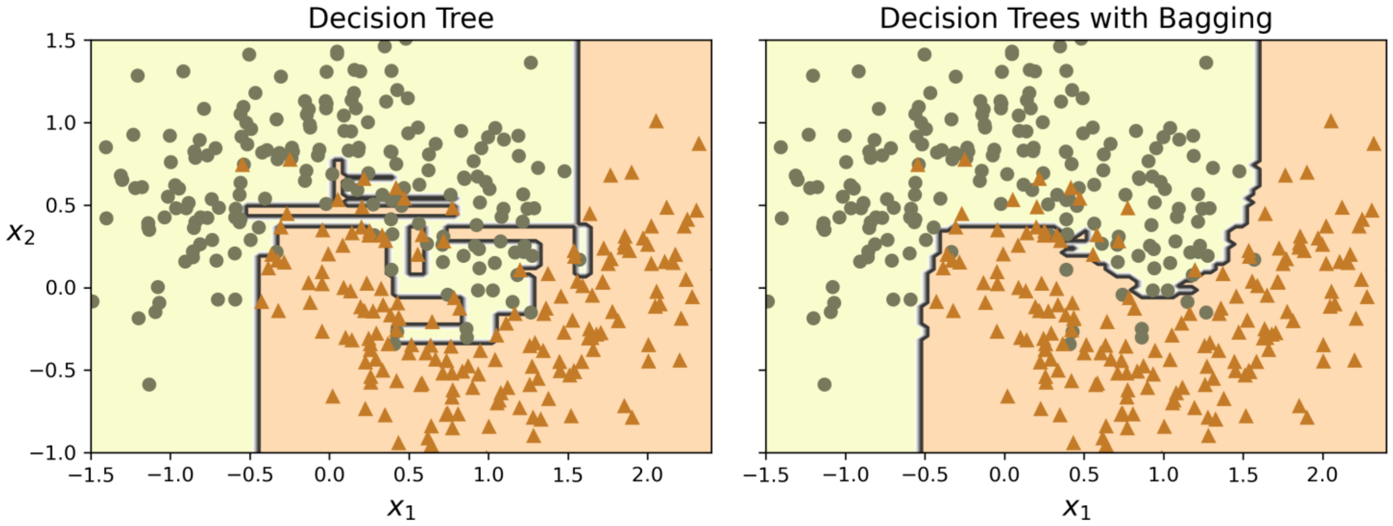
```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

bag_clf = BaggingClassifier(DecisionTreeClassifier(), n_estimators=500,
                            max_samples=100, n_jobs=-1, random_state=42)
bag_clf.fit(X_train, y_train)
```

- A `BaggingClassifier` automatically performs soft voting instead of hard voting if the base classifier can estimate class probabilities.
- To use pasting instead, set `bootstrap=False`

Ensemble reduces variance

- The ensemble has a comparable bias but a smaller variance.
  - It makes roughly the same number of errors on the training set, but the decision boundary is less irregular.



Bagging vs. Pasting

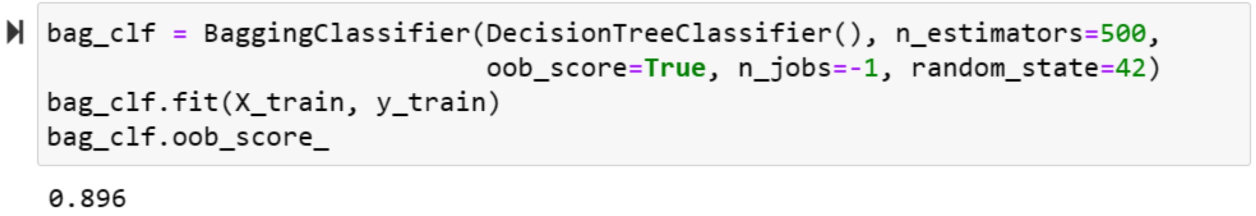
- Bagging introduces a bit more diversity in the subsets that each predictor is trained on, so bagging ends up with a slightly higher bias than pasting.
- The extra diversity also means that the predictors end up being less correlated, so the ensemble's variance is reduced.
- Overall, bagging often results in better models.
- If you have spare time and CPU power, you can use cross-validation to evaluate both bagging and pasting and select the one that works best.

Out-of-Bag Evaluation

- With bagging, some training instances may be sampled several times for any given predictor, while others may not be sampled at all.
  - Only about 63% of the training instances are sampled on average for each predictor.
  - The remaining 37% of the training instances that are not sampled are called *out-of-bag* (OOB) instances.
- A bagging ensemble can be evaluated using OOB instances, without the need for a separate validation set.

OOB in Scikit-Learn

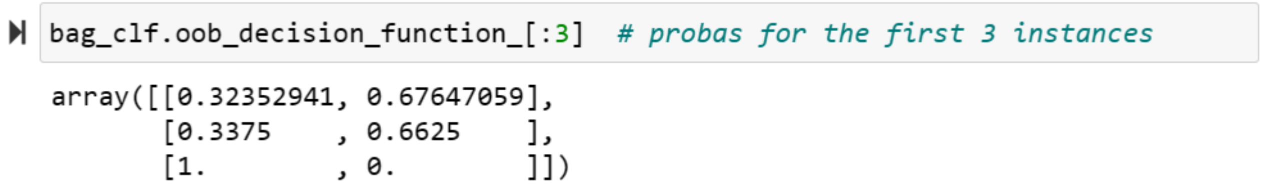
- Set `oob_score=True` when creating a `BaggingClassifier` to request an automatic OOB evaluation after training.



```
bag_clf = BaggingClassifier(DecisionTreeClassifier(), n_estimators=500,
                            oob_score=True, n_jobs=-1, random_state=42)
bag_clf.fit(X_train, y_train)
bag_clf.oob_score_
```

```
0.896
```

- The OOB decision function for each training instance is also available through the `oob_decision_function_` attribute:



```
bag_clf.oob_decision_function_[:3]  # probas for the first 3 instances
```

```
array([[0.32352941, 0.67647059],
       [0.3375    , 0.6625    ],
       [1.        , 0.        ]])
```

Random Patches and Random Subspaces

- The `BaggingClassifier` supports sampling the features as well.
  - Useful when you are dealing with high-dimensional inputs (such as images), as it can considerably speed up training.
- Sampling is controlled by two hyperparameters: `max_features` and `bootstrap_features`.
- Sampling both training instances and features is called the *random patches* method.
- Keeping all training instances (by setting `bootstrap=False` and `max_samples=1.0`) but sampling features is called the *random subspaces* method.
- Sampling features results in even more predictor diversity.

3.
Random Forest

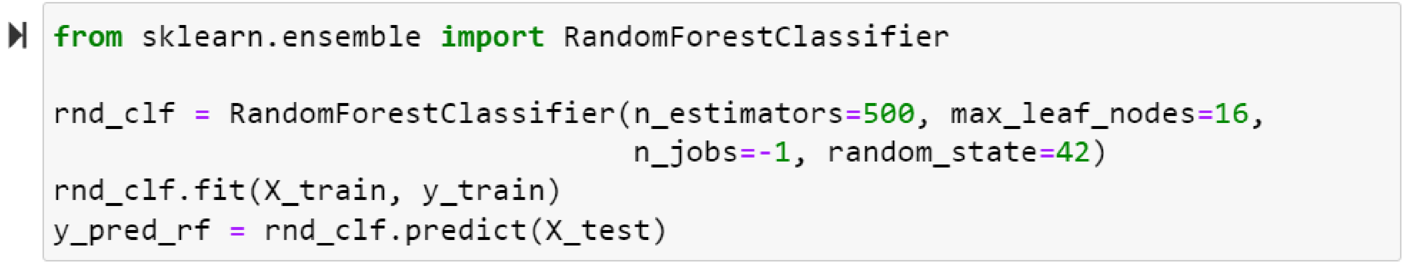
Random Forest

- A random forest is an ensemble of decision trees, usually trained via the bagging method, typically with `max_samples` set to the size of the training set.

- Instead of building a `BaggingClassifier` and passing it a `DecisionTreeClassifier`, we use the `RandomForestClassifier`, which is more convenient and optimized for decision trees.
  - Similarly, a `RandomForestRegressor` for regression tasks.

Random Forest in Scikit-Learn

- The following code trains a random forest classifier with 500 trees, each limited to maximum 16 leaf nodes, using all available CPU cores:



```
from sklearn.ensemble import RandomForestClassifier

rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16,
                                 n_jobs=-1, random_state=42)
rnd_clf.fit(X_train, y_train)
y_pred_rf = rnd_clf.predict(X_test)
```

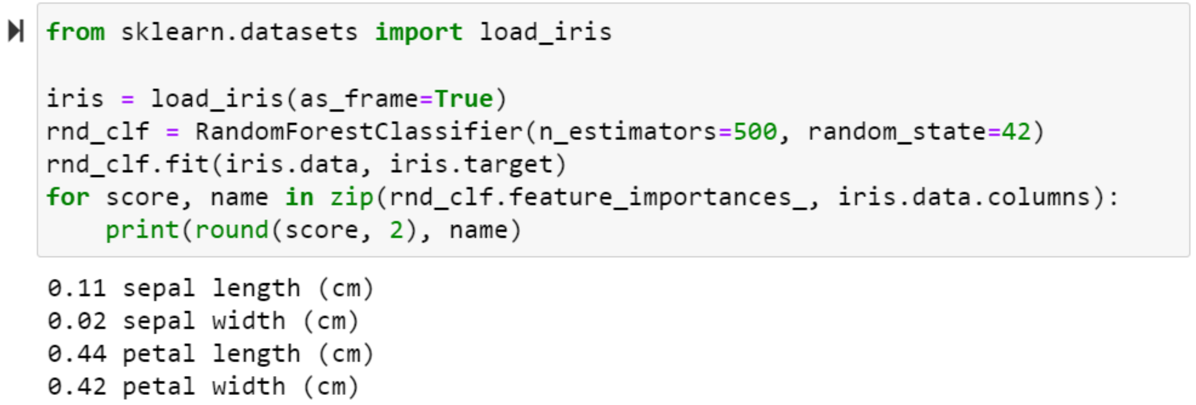
- The random forest algorithm, instead of searching for the very best feature when splitting a node, searches for the best feature among a random subset of features.
  - By default, it samples $\sqrt{n}$ features (where n is the total number of features).

Extra Trees

- When you are growing a tree in a random forest, at each node only a random subset of the features is considered for splitting.
- It is possible to make trees more random using random thresholds for each feature rather than searching for the best possible thresholds.
  - Set `splitter="random"` when creating a `DecisionTreeClassifier`.
- Such a forest is called an *extremely randomized trees* (or *extra-trees*) ensemble.
  - It achieves lower variance and much faster training, but more bias.
- You can create an extra-trees classifier using `ExtraTreesClassifier`.
  - Similarly, the `ExtraTreesRegressor` for regression tasks.

Feature Importance

- Scikit-Learn measures a feature’s importance by looking at how much the tree nodes that use the feature reduce impurity on average, across all trees in the forest.
  - It is a weighted average, where each node’s weight is equal to the number of training samples that are associated with it.
- You can access the result using the `feature_importances_` variable.

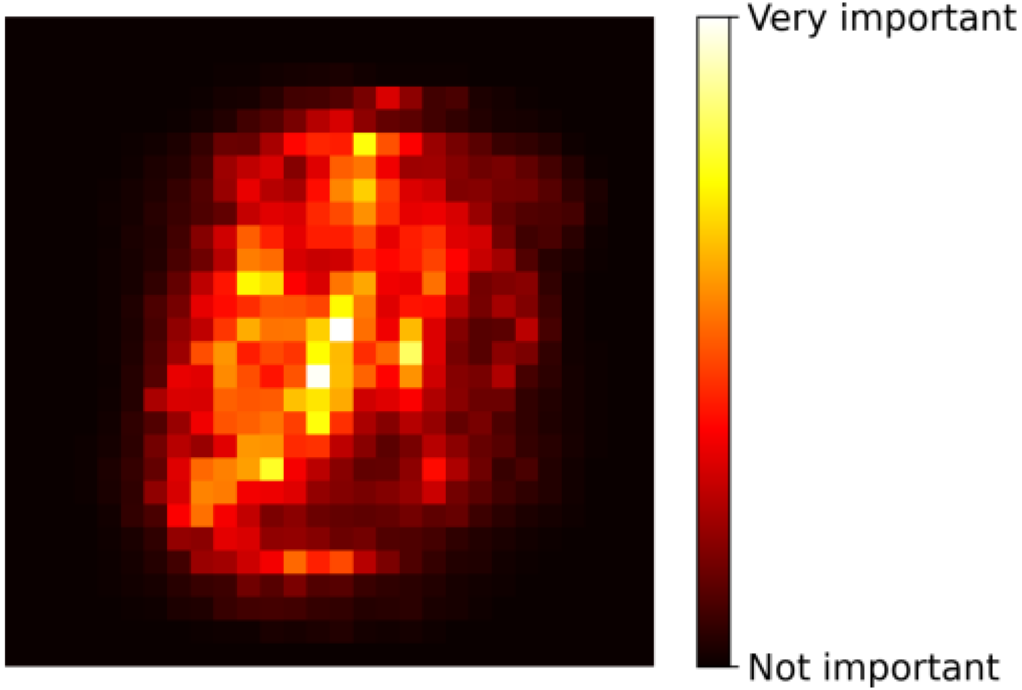


```python
from sklearn.datasets import load_iris

iris = load_iris(as_frame=True)
rnd_clf = RandomForestClassifier(n_estimators=500, random_state=42)
rnd_clf.fit(iris.data, iris.target)
for score, name in zip(rnd_clf.feature_importances_, iris.data.columns):
    print(round(score, 2), name)
```

```
0.11 sepal length (cm)
0.02 sepal width (cm)
0.44 petal length (cm)
0.42 petal width (cm)
```

MNIST Pixel Importance


Very important
Not important