SVM & Ensemble Learning

Zichao Yang

Zhongnan University of Economics & Law

Date: September 23, 2024

Roadmap

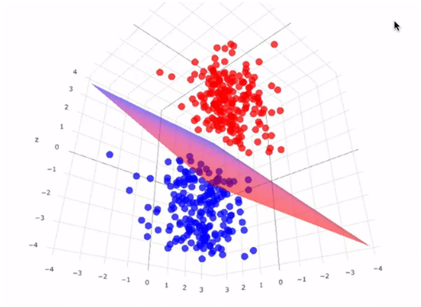
- Support Vector Machines (SVM)
- Decision Trees
- Ensemble Learning and Random Forests

I. Support Vector Machines (SVM)

Support Vector Machine (SVM)

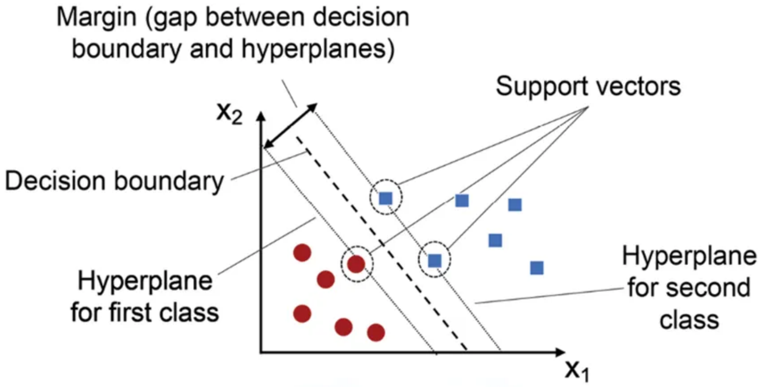
Support Vector Machine (SVM) is capable of performing linear or nonlinear classification and regression. It is particularly well suited for classification of complex small- or medium-size datasets.

SVMs work by finding the optimal hyperplane that separates data points into different classes.



Source: Support Vector Machines (SVM): An Intuitive Explanation

Linear SVM: A 2D Example


Margin (gap between decision boundary and hyperplanes)
Support vectors
x_2
Decision boundary
Hyperplane for first class
Hyperplane for second class
x_1

Source: Support Vector Machines (SVM): An Intuitive Explanation

Support Vector Machine (SVM)

Decision Boundary: a **hyperplane** that separates data points into different classes in a high-dimensional space. In two-dimensional space, a hyperplane is simply a line that separates the data points into two classes. In N-dimensional space, a hyperplane has (N-1)-dimensions.

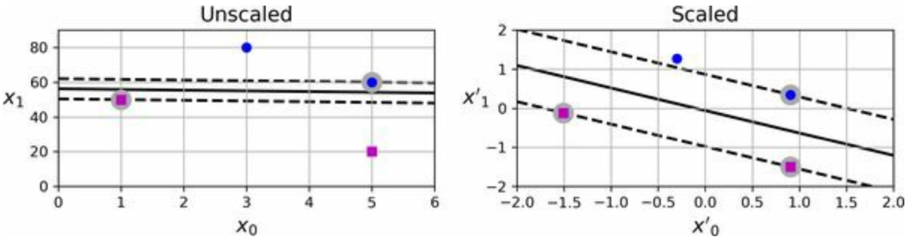
Margin: the distance between the decision boundary (hyperplane) and the closest data points from each class.

Support Vectors: the data points (aka, instances) that lie closest to the decision boundary. The support vectors are used to calculate the margin, which is the distance between the decision boundary and the closest data points from each class.

The goal of SVMs is to maximize this margin while minimizing classification errors. This is called **large margin classification**.

SVM: Feature Scales

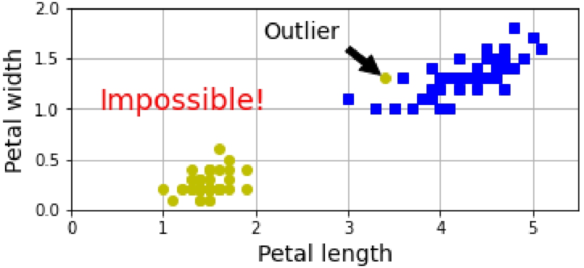
SVMs are sensitive to the feature scales. In the left plot, the vertical scale is much larger than the horizontal scale, so the widest margin is close to horizontal and much smaller than the right one.



Hard Margin Classification

In **hard margin classification**, the goal is to find a hyperplane that completely separates the two classes in the feature space, with no data points allowed in the margin or on the wrong side of the hyperplane.

Hard margin classification assumes that the **data is linearly separable**, and this method is **sensitive to outliers**.



Linear SVM: Under the Hood

The linear SVM classifier computes the decision function: $\boldsymbol{w}^\intercal \boldsymbol{x} + b = w_1 x_1 + \ldots + w_n x_n + b$, and the prediction function is:

$$\hat{y} = \begin{cases} 0 & \text{if} \quad \boldsymbol{w}^\intercal \boldsymbol{x} + b < 0, \\ 1 & \text{if} \quad \boldsymbol{w}^\intercal \boldsymbol{x} + b \geq 0 \end{cases}$$

Now if we want to model the margin, we need to make an assumption, we assume the hyperplane that defines the margin can be written as:

Hyperplane for the first class: $\boldsymbol{w}^\intercal \boldsymbol{x} + b = -1$

Hyperplane for the second class: $\boldsymbol{w}^\intercal \boldsymbol{x} + b = 1$

Q: why do we choose 1 and -1, not some other values like 3, -4 or 0 ?

Linear SVM: Under the Hood

Q: why do we choose 1 and -1, not some other values like 3, -4 or 0 ?

(1) We want the decision boundary to have equal distance from both the classes, that's why we take magnitude equal (1 and -1).

(2) Let's say the equation of our hyperplane as $w_1x_1 + b = 2$, we can always re-scale it to be $w_1'x_1 + b' = 1$. Hence for mathematical convenience, we take it as 1.

(3) If we choose $w_1x_1 + b = 2$ and $w_1x_1 + b = 4$, we can always shift the margin to be $w_1x_1 + b' = -1$ and $w_1x_1 + b' = 1$.

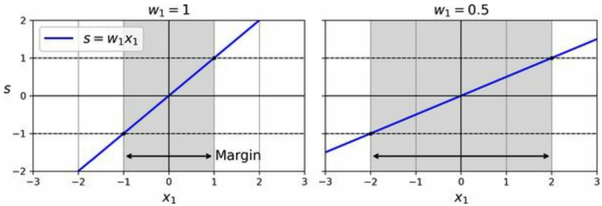
Note: the value of b can only shift the position of the margin, cannot expand/shrink the margin.

Linear SVM: Under the Hood

Now, how do we measure if the prediction function is good enough?

Recall, the goal of SVMs is to maximize the margin while minimizing classification errors. For hard margin classification, there should be no classification error.

The slope of the decision function equals to the norm of the weight vector $||\boldsymbol{w}||$. The plot shows that dividing the slope by 2 will multiply the margin by 2. Hence, we want to keep $||\boldsymbol{w}||$ as small as possible.



Linear SVM: Under the Hood

Furthermore, if we define $t^{(i)} = -1$ fro negative points ($y^{(i)} = 0$) and $t^{(i)} = 1$ for positive points ($y^{(i)} = 1$), then the “no classification error” requirement can be expressed as:

$$\boldsymbol{t}^{(i)}(\boldsymbol{w}^\intercal \boldsymbol{x} + b) \geq 1$$

Hence the objective function for hard margin linear SVM classifier is:

$$\begin{aligned} &\min_{\boldsymbol{w},\boldsymbol{b}} \quad \frac{1}{2}\boldsymbol{w}^\intercal \boldsymbol{w} \\ s.t. \quad &\boldsymbol{t}^{(i)}(\boldsymbol{w}^\intercal \boldsymbol{x} + b) \geq 1 \quad \text{for } i = 1, 2, ..., m \end{aligned}$$

Soft Margin Classification

Soft Margin Classification tries to find a good balance between enlarging the margin and limiting the margin violations.

To model the soft margin objective, we need to introduce a new variable $\zeta^i > 0$ for each point, and it measures how much the point is allowed to violate the margin.

We now have two conflicting goals: (1) make $\frac{1}{2}\boldsymbol{w}^\intercal\boldsymbol{w}$ as small as possible, (2) make $\zeta^i > 0$ as small as possible. So we need to introduce a hyperparameter C to define the trade-off between these two goals:

$$\min_{\boldsymbol{w},\boldsymbol{b}} \quad \frac{1}{2}\boldsymbol{w}^\intercal\boldsymbol{w} + C\sum_{i=1}^{m}\zeta^i$$

$$s.t. \quad \boldsymbol{t}^{(i)}(\boldsymbol{w}^\intercal\boldsymbol{x} + b) \geq 1 - \zeta^i \quad \text{and} \quad \zeta^i \geq 0 \quad \text{for } i = 1, 2, ..., m$$

Larger C: lower tolerance to margin violations. (Try it out in python!)

Linear SVC: different methods

(1) **LinearSVC** class: LinearSVC(C=1)

LinearSVC User Guide

(2) **SVC** class: SVC(kernel = 'linear', C = 1)

SVC User Guide

(3) **SGDClassifier** class: SGDClassifier(loss='hinge', alpha= 0.0001)

SGDClassifier User Guide, Hinge loss function: $\max(0, 1 - t)$.

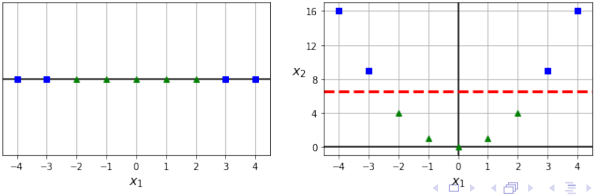
Let's try them out in python!

Nonlinear SVM Classification: Polynomial Terms

In machine learning research, many datasets are not even close to being linearly separable.

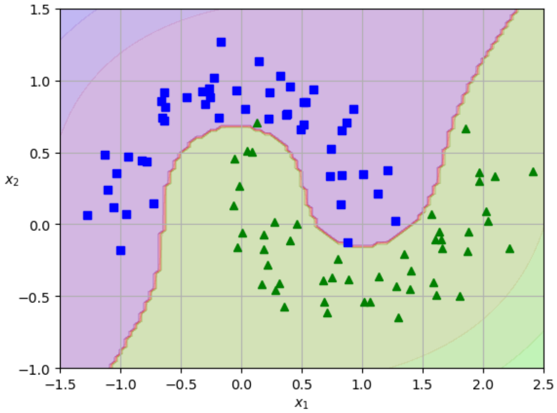
One approach to handling nonlinear datasets is to add more features, such as polynomial features; in some cases this can result in a linearly separable dataset.

For example, the left panel shows a dataset x_1 that is not linearly separable. But if you add a second feature $x_2 = (x_1)^2$, the resulting 2D dataset on the right panel is perfectly linearly separable.



Nonlinear SVM Classification: Polynomial Terms

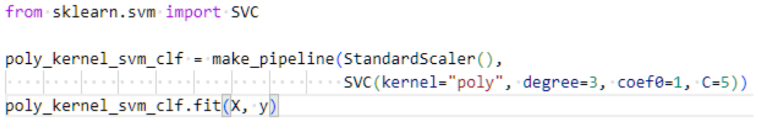
To implement this idea using Scikit-Learn, you can create a pipeline containing a **PolynomialFeatures** transformer, followed by a StandardScaler and a LinearSVC classifier. Let's see an example using the moons dataset.



Nonlinear SVM Classification: Polynomial Kernel

Adding polynomial features is simple to implement, but this method has one drawback: a low polynomial degree cannot deal with very complex datasets, a high polynomial degree creates a huge number of features, making the model too slow.

Fortunately, when using SVMs you can apply a mathematical technique called the **kernel trick**, which makes it possible to get the same result as if you had added many polynomial features, even with a very high degree, without actually having to add them.

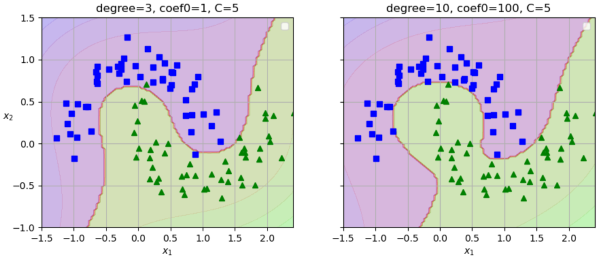


```python
from sklearn.svm import SVC

poly_kernel_svm_clf = make_pipeline(StandardScaler(),
                                    SVC(kernel="poly", degree=3, coef0=1, C=5))
poly_kernel_svm_clf.fit(X, y)
```

Let's try it out in python!

Nonlinear SVM Classification: Polynomial Kernel

The following plots show SVM classifiers using third-degree and 10th-degree polynomial kernels.



If your model is overfitting, reduce the polynomial degree. If it is underfitting, increase the degree.

The hyperparameter coef0 controls how much the model is influenced by high-degree terms versus low-degree terms.

Nonlinear SVM Classification: Gaussian RBF Kernel

Another technique to tackle nonlinear problems is to add features computed using a similarity function, which measures how much each instance resembles a particular landmark. For example:

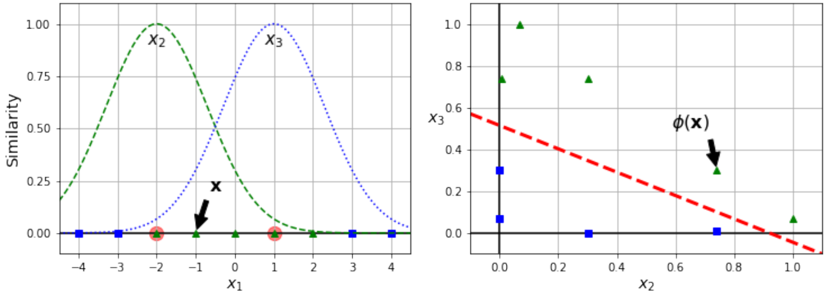
(1) We have a 1D dataset and add two landmarks to it at $x = -2$ and $x = 1$.

(2) We define the similarity function to be the Gaussian RBF with $\gamma = 0.3$. This is a bell-shaped function, whose values varying from 0 (very far away from the landmark) to 1 (at the landmark).

(3) Now we are ready to compute the new features. Let's look at the instance $x = -1$: it is located at a distance of 1 from the first landmark and 2 from the second landmark. Therefore, its new features are $x = \exp(-0.31) \approx 0.74$ and $x = \exp(-0.32) \approx 0.30$.

Nonlinear SVM Classification: Gaussian RBF Kernel

The following plots show how similarity function works.



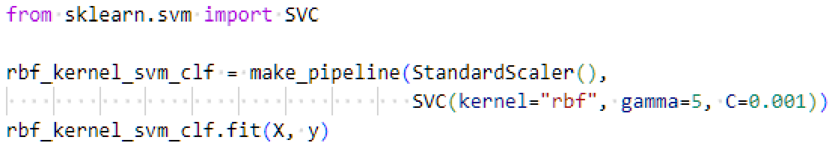
Then, how to select the landmarks?

The simplest approach is to create a landmark at the location of each and every instance in the dataset.

Nonlinear SVM Classification: Gaussian RBF Kernel

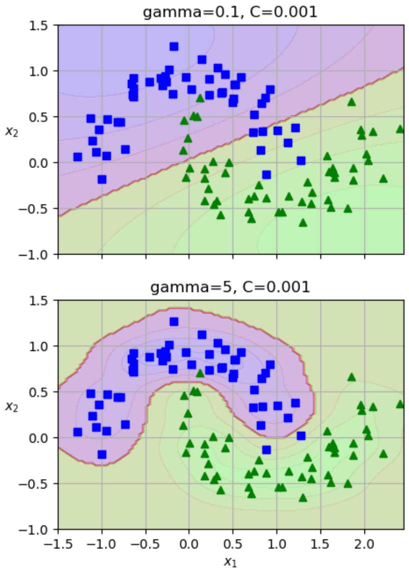
Just like the polynomial features method, it is computationally expensive to compute all the landmarks. The kernel trick in SVM makes it possible to obtain a similar result as if you had added many similarity features, but without actually doing so.

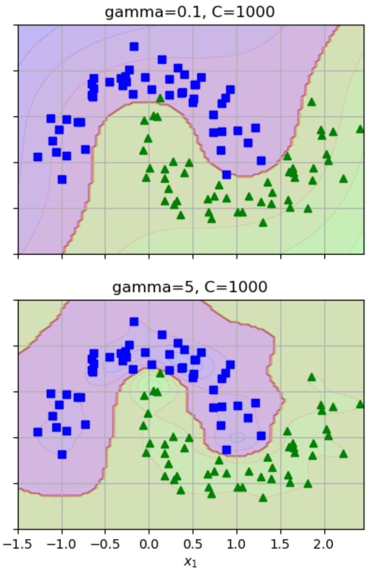
Let's try the SVC class with the Gaussian RBF kernel:



```python
from sklearn.svm import SVC

rbf_kernel_svm_clf = make_pipeline(StandardScaler(),
                                   SVC(kernel="rbf", gamma=5, C=0.001))
rbf_kernel_svm_clf.fit(X, y)
```

gamma: increasing gamma makes the bell-shaped curve narrower. If your model is overfitting, you should reduce it.
C: larger C indicates lower tolerance to margin violations.


gamma=0.1, C=0.001
gamma=0.1, C=1000
gamma=5, C=0.001
gamma=5, C=1000
x_1
x_2
−1.5 −1.0 −0.5 0.0 0.5 1.0 1.5 2.0
1.5 1.0 0.5 0.0 −0.5 −1.0



How to choose the suitable kernel?

(1) You should always try the linear kernel first.
Remember: LinearSVC(.) is faster than SVC(kernel='linear').

(2) If the training set is not too large, try the Gaussian RBF kernel; it works well in most cases.

(3) If you have spare time and computing power, you can try other kernels using cross-validation and grid search.

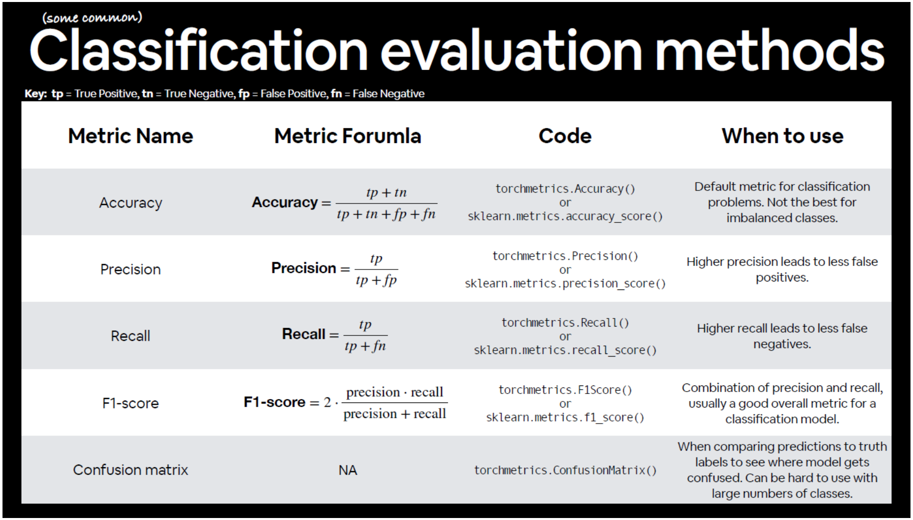
SVM Exercise

Apply what you learned to the Breast cancer wisconsin (diagnostic) dataset.

Hint: your code should begin with

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# load the data
data = load_breast_cancer()
# standardize the data
# train the model
# predict the test data
```

Model Performance Metrics



(some common)

Classification evaluation methods

Key: tp = True Positive, **tn** = True Negative, **fp** = False Positive, **fn** = False Negative

Metric Name	Metric Forumla	Code	When to use
Accuracy	**Accuracy** $= \frac{tp+tn}{tp+tn+fp+fn}$	`torchmetrics.Accuracy()` or `sklearn.metrics.accuracy_score()`	Default metric for classification problems. Not the best for imbalanced classes.
Precision	**Precision** $= \frac{tp}{tp+fp}$	`torchmetrics.Precision()` or `sklearn.metrics.precision_score()`	Higher precision leads to less false positives.
Recall	**Recall** $= \frac{tp}{tp+fn}$	`torchmetrics.Recall()` or `sklearn.metrics.recall_score()`	Higher recall leads to less false negatives.
F1-score	**F1-score** $= 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$	`torchmetrics.F1Score()` or `sklearn.metrics.f1_score()`	Combination of precision and recall, usually a good overall metric for a classification model.
Confusion matrix	NA	`torchmetrics.ConfusionMatrix()`	When comparing predictions to truth labels to see where model gets confused. Can be hard to use with large numbers of classes.

Source: Daniel Bourke - TensorFlow for Deep Learning Bootcamp: Zero to Mastery

SVM Regression

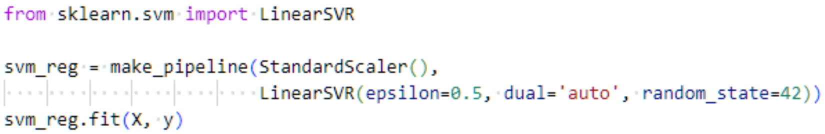
To use SVMs for regression, we need to reverse the objective:

Instead of trying to find the largest possible margin between two classes while limiting margin violations, SVM Regression tries to **fit as many instances as possible into the margin** while limiting margin violations.

The width of the margin is controlled by a hyperparameter, ϵ.

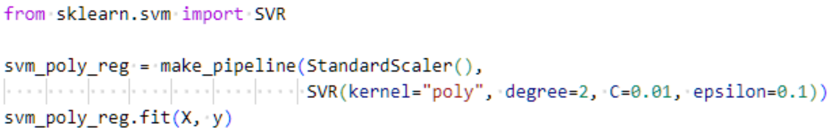
SVM Regression

(1) Linear SVM Regression



```
from sklearn.svm import LinearSVR

svm_reg = make_pipeline(StandardScaler(),
                        LinearSVR(epsilon=0.5, dual='auto', random_state=42))
svm_reg.fit(X, y)
```

(2) Nonlinear SVM Regression



```
from sklearn.svm import SVR

svm_poly_reg = make_pipeline(StandardScaler(),
                             SVR(kernel="poly", degree=2, C=0.01, epsilon=0.1))
svm_poly_reg.fit(X, y)
```

Let’s try them out in python!

II. Decision Trees

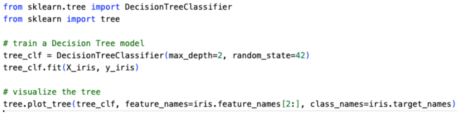
Decision Trees

Like SVMs, Decision Trees are versatile Machine Learning algorithms, and can perform both classification and regression tasks.

More importantly, Decision Trees are also th fundamental components of Random Forests, which is a widely used Machine Learning algorithm.

Training and Visualizing a Decision Tree

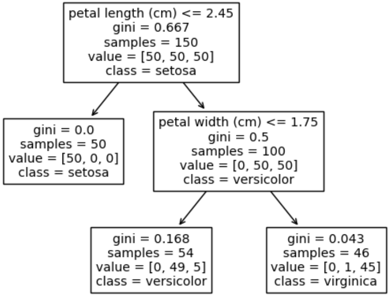
(1) Train a Decision Tree model:



```python
from sklearn.tree import DecisionTreeClassifier
from sklearn import tree

# train a Decision Tree model
tree_clf = DecisionTreeClassifier(max_depth=2, random_state=42)
tree_clf.fit(X_iris, y_iris)

# visualize the tree
tree.plot_tree(tree_clf, feature_names=iris.feature_names[2:], class_names=iris.target_names)
```

max_depth : The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than *min_samples_split* samples.

(2) Decision Tree Visualization:



What about the other values in the visualization graph?

Decision Tree Interpretation

petal length (width) : two conditions to generate a Decision Tree with max_depth=2.

samples: counts how many training instances it applied to.

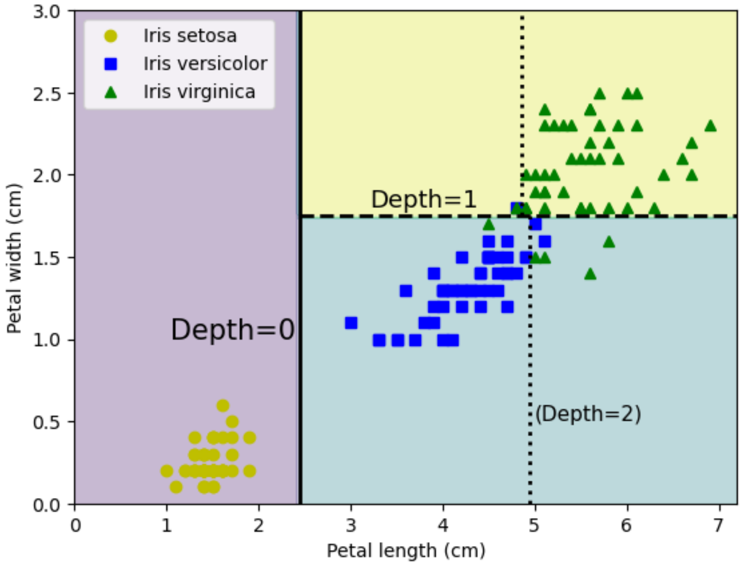
value: tells us how many trianing instances of each class this node applies to. For example, [0, 1, 45] means this node applies to 0 setosa, 1 versicolor, and 45 virginica.

gini (G_i): measures a node's impurity: a node is "pure" (gini=0) if all training instances it applied to belong to the same class.

$$G_i = 1 - \sum_{k=1}^{n} p_{i,k}^2$$

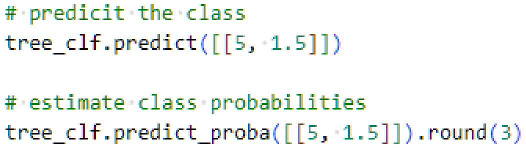
$p_{i,k}$ is the ratio of class k instances among the training instances in the i^{th} node.

Making Prediction



Estimating Class Probabilities

A Decision Tree can estimate the probability that an instance belongs to a particular class k.



```
# predicit the class
tree_clf.predict([[5, 1.5]])

# estimate class probabilities
tree_clf.predict_proba([[5, 1.5]]).round(3)
```

Let's try it out in Python!

The CART Training Algorithm

Scikit-Learn uses the Classification and Regression Tree (CART) algorithm, which produces only binary trees: non-leaf nodes always have two children.

The algorithm works by first splitting the training set into two subsets using a single feature k and a threshold t_k (e.g., "petal length $\leq$ 2.45 cm"). It searches for the pair (k, t_k) that produces the purest subsets, weighted by their size, see the following cost function for CART:

$$J(k, t_k) = \frac{m_{left}}{m} G_{left} + \frac{m_{right}}{m} G_{right}$$

$$\text{where} \begin{cases} G_{left/right} \text{ measures the impurity of the left/right subset,} \\ m_{left/right} \text{ is the number of instances in the left/right subset.} \end{cases}$$

The CART Training Algorithm

Once the CART algorithm has successfully split the training set in two, it splits the subsets using the same logic, then the sub-subsets, and so on, recursively. The CART algorithm is a greedy algorithm and we need a stopping condition.

The CART algorithm stops once it reaches the maximum depth (the max_depth hyperparameter), or if it cannot find a split that will reduce impurity.

A few other hyperparameters control additional stopping conditions: $min\_samples\_split$, $min\_samples\_leaf$, $min\_weight\_fraction\_leaf$, and max_leaf_nodes. You can find more info from the User Guide.

Gini Impurity or Entropy?

When we calculate the CART cost function, the Gini impurity is used by default, but you can also select the entropy impurity.

gini (G_i): measures a node's impurity: a node is "pure" (gini=0) if all training instances it applied to belong to the same class.

entropy (H_i): a node's entropy is 0 if all training instances it applied to belong to the same class.

$$H_i = -\sum_{k=1, p_{i,k} \neq 0}^{n} p_{i,k} \log_2(p_{i,k})$$

$p_{i,k}$ is the ratio of class k instances in the i^{th} node.

Q: Should you use Gini impurity or entropy?
Most of the time it does not matter. You can try both if you have time.

Regularization Hyperparameters

The Regression Tree model is often called a **nonparametric model**, not because it does not have any parameters (it often has a lot) but because the number of parameters is not determined prior to training, so the model structure is free to stick closely to the data.

To avoid overfitting the training data, you need to define the **regularization hyperparameter** to restrict the decision tree's freedom during training.

The common regularization hyperparameter is the max_depth. And you can set up others as we mentioned before: $min\_samples\_split$, $min\_samples\_leaf$, $min\_weight\_fraction\_leaf$, and max_leaf_nodes. (see User Guide)

Let's see an example in Python!

Decision Tree Regression

The regression model looks very similar to the classification model. The main difference is that instead of predicting a class in each node, it predicts a value.

CART cost function for regression:

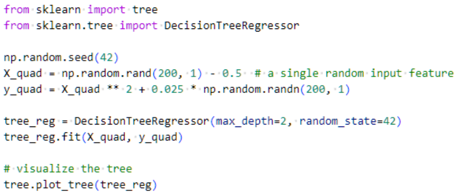
$$J(k, t_k) = \frac{m_{\text{left}}}{m} MSE_{\text{left}} + \frac{m_{\text{right}}}{m} MSE_{\text{right}}$$

$$\text{where} \begin{cases} MSE_{\text{node}} = \sum_{i \in \text{node}} (\hat{y}_{\text{node}} - y^{(i)})^2 \\ \hat{y}_{\text{node}} = \frac{1}{m_{\text{node}}} \sum_{i \in \text{node}} y^{(i)} \end{cases}$$

The CART algorithm works mostly the same way as before, except that instead of trying to split the training set in a way that minimizes impurity, it now tries to split the training set in a way that minimizes the MSE.

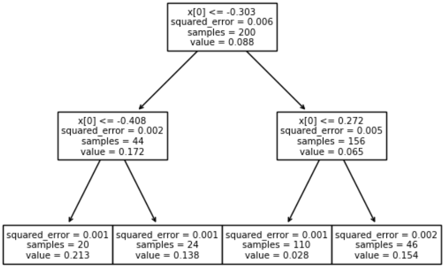
Decision Tree Regression

(1) Decision Tree Regression:



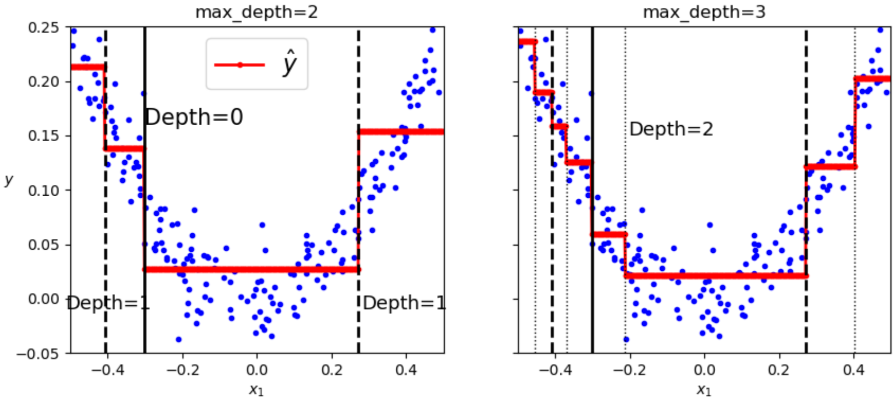
```
from sklearn import tree
from sklearn.tree import DecisionTreeRegressor

np.random.seed(42)
X_quad = np.random.rand(200, 1) - 0.5  # a single random input feature
y_quad = X_quad ** 2 + 0.025 * np.random.randn(200, 1)

tree_reg = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg.fit(X_quad, y_quad)

# visualize the tree
tree.plot_tree(tree_reg)
```

(2) Regression Visualization:



Decision Tree Regression Illustration

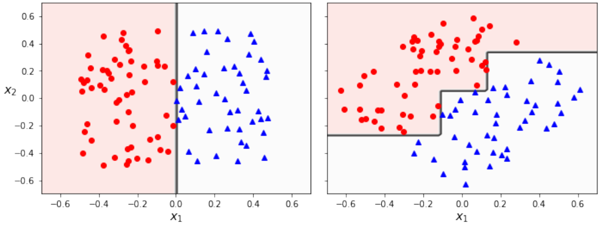
Q: How does the Decision Tree model approaches regression problems?



Drawback: Instability

Decision Tree Model has a few limitations:

(1) decision trees love orthogonal decision boundaries (all splits are perpendicular to an axis), which makes them sensitive to the data's orientation. (see the following picture)



(2) decision trees have quite a high variance: small changes to the hyperparameters or to the data may produce very different models.

III. Ensemble Learning and Random Forests

Ensemble Learning

In machine learning, if you aggregate the predictions of a group of predictors, you will often get better predictions than with the best individual predictor.

A group of predictors is called an **ensemble**; thus, this technique is called **ensemble learning**, and an ensemble learning algorithm is called an **ensemble method**.

For example, you can train a group of decision tree classifiers, each on a different random subset of the training set. You can then 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**.

Voting Classifiers

Suppose you have trained a few classifiers, each one achieving about 80% accuracy. You may have a logistic regression classifier, an SVM classifier, a random forest classifier, a k-nearest neighbors classifier, and perhaps a few more. Can you achieve a even better performance?

A very simple way to create an even better classifier is to aggregate the predictions of each classifier. In what way should we aggregate these predictions?

The first method is to let the ensemble's prediction be the class that gets the most votes. This majority-vote classifier is also called a **hard voting** classifier.

Hard Voting Classifier Illustration

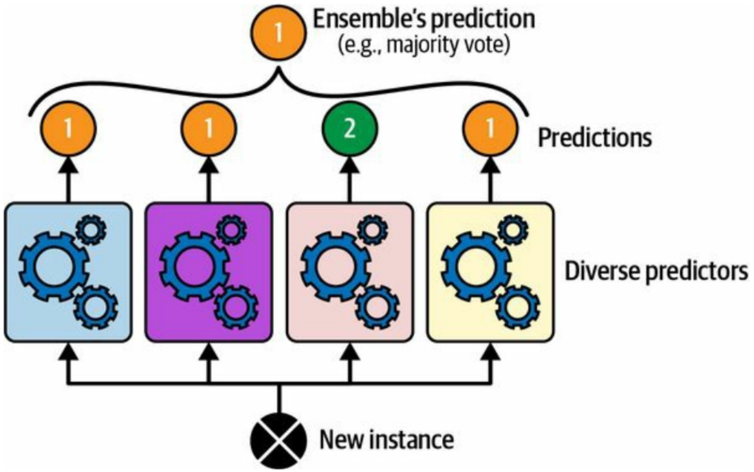


Figure 7-2. Hard voting classifier predictions

Source: Géron, Aurélien. Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow. " O'Reilly Media, Inc.", 2022.

Advantage of Ensemble Learning

Even if each classifier has a mediocre performance (i.e. slightly better than random guessing), the ensemble can still has great performance, provided there are a sufficient number of diverse classifiers.

Let's assume that the probability of one classifier correctly predict the class is p, and can be modeled by a binomial distribution. Given n classifiers, the probability mass function of the binomial distribution is:

$$P(X = k) = \binom{n}{k} \cdot p^k \cdot (1 - p)^{n-k}$$

where X is the number of correctly predicted instances. $\binom{n}{k}$ represents the number of ways to choose k correct predictions out of n classifiers. Now the probability of getting a majority of correct votes is:

$$P(\text{Majority Correct}) = \sum_{k=\frac{n}{2}+1}^{n} P(X = k)$$

Advantage of Ensemble Learning

Suppose you build an ensemble containing 1,000 classifiers that are individually correct only 51% of the time. If you predict the majority voted class, let's calculate in python the accuracy rate of the ensemble.

You can hope for around 73% accuracy!

Ensemble methods work best when the predictors are as independent from one another as possible. One way to get diverse classifiers is to train them using very different algorithms.

Soft Voting

In **soft voting**, each model in the ensemble provides a probability distribution over the classes rather than a hard assignment of a single class label. The final prediction is then made by averaging or taking a weighted average of these probability distributions.

Mathematically, for a classification task, if you have M models and N classes, the soft voting prediction p_{soft} for class i would be:

$$p_{soft,i} = \frac{1}{M} \sum_{j=1}^{M} p_{j,i}$$

where M is the number of classifiers, N is the number of classes, $p_{j,i}$ is the predicted probability by the j^{th} model for class i.

Let's try it out in Python!

Bagging and Pasting

Previously, we have discussed that one way to get a diverse set of predictors (classifiers) is to use very different training algorithms.

Another approach uses the same training algorithm for every predictors but train them on different random subsets of the training set.

Q: How to generate these random subsets?

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**.

Remember: both bagging and pasting allow training instances to be sampled several times across multiple predictors, but only bagging allows training instances to be sampled several times for the same predictor.

Bagging and Pasting

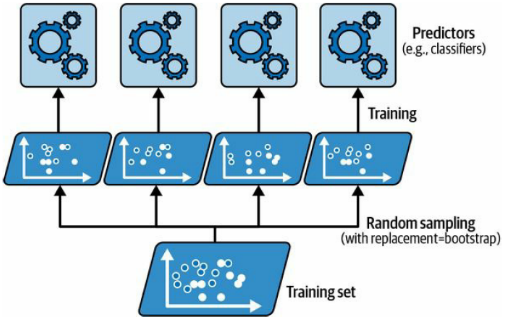


Figure 7-4. Bagging and pasting involve training several predictors on different random samples of the training set

Source: Géron, Aurélien. Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow. " O'Reilly Media, Inc.", 2022.

Bagging and pasting methods can scale very well because predictors can all be trained in parallel, via different CPU cores or servers.

Bagging and Pasting in Scikit-Learn

Suppose we have 500 training instances and we want to train an ensemble of 500 different Decision Tree classifiers. Before trying the code in Python, let's discuss the following two questions:

Q1: can we directly use the whole 500 instances to train the 500 different Decision Tree classifiers?

Q2: what is the difference between Bagging and Pasting methods in this specific example?

Bagging and Pasting in Scikit-Learn

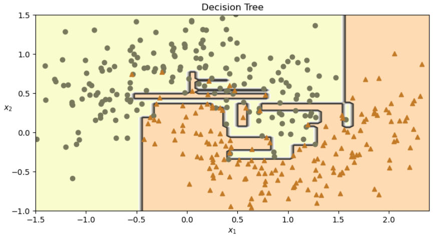
A1: if we directly use the whole 500 instances to train the 500 Decision Tree classifiers, eventually these 500 classifiers will be identical, and we won't be able to detect nuances in different sub-samples. That is why here we randomly sample 100 instances to train each classifier.

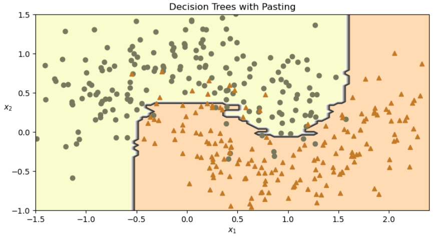
A2: The difference lies in how to randomly sample these 100 instance. If these 100 instances are sampled with replacement (i.e., bootstrapping = True), then the method is called bagging; if not, then it is called pasting.

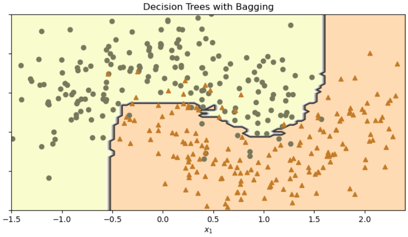
Bagging introduces a bit more diversity in the subsets that each predictor is trained on, so bagging ends up with a slightly higher bias but less variance. Overall, bagging is often preferred over pasting.

Now let's try Bagging and Pasting methods in python!

Bagging and Pasting in Scikit-Learn







Out-of-Bag Evaluation

Suppose we have a training set with m instances, **BaggingClassifier** samples m training instances with replacement. The probability of never being selected is: $(1 - \frac{1}{m})^m$, so:

$$\lim_{m \to \infty} (1 - \frac{1}{m})^m = \frac{1}{e} \approx 37\%$$

The remaining 37% of the training instances that are not sampled are called **out-of-bag (OOB)** instances.
Note: they are not the same 37% for all classifiers.

Hence, in Scikit-Learn, we can set **oob_score=True** when creating a BaggingClassifier to request an automatic OOB evaluation after training. Let's try it out in Python!

Random Patches and Random Subspaces

The **BaggingClassifier** class supports sampling the features as well. Sampling is controlled by two hyperparameters: **max_features** and **bootstrap_features**. They work the same way as max_samples and bootstrap, but for feature sampling instead of instance sampling.

Sampling both training instances and features is called the **random patches** method.

Keeping all training instances (bootstrap = False and max_samples = 1.0) but sampling features (bootstrap_features = True and/or max_features < 1.0) is called the **random subspaces** method.

This technique is particularly useful when you are dealing with high dimensional inputs. Sampling features results in even more predictor diversity, trading a bit more bias for a lower variance.

Random Forests

A **random forest** is an ensemble of decision trees, generally trained via the bagging (bootstrap = True), with max_samples = the size of the training set.

RandomForestClassifier() = BaggingClassifier(DecisionTreeClassifier())

The random forest algorithm introduces extra randomness when growing trees; instead of searching for the very best feature when splitting a node, it searches for the best feature among a random subset of features.

Random Forests

The following two methods are roughly equivalent:

```
# Method 1
from sklearn.ensemble import RandomForestClassifier
rnd_clf = RandomForestClassifier(n_estimators=500,
                                 max_leaf_nodes=16,
                                 n_jobs=-1, random_state=42)

# Method 2
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier
bag_clf = BaggingClassifier(DecisionTreeClassifier(max_features="sqrt",
                                                   max_leaf_nodes=16),
                        n_estimators=500, n_jobs=-1, random_state=42)
```

Extra-Trees

- **Random Forests**:
  - **Feature Selection**: Random Forests build each decision tree by considering a random subset of features at each split.
  - **Splitting Strategy**: It evaluates these features to find the best split based on information gain or Gini impurity.
- **Extra-Trees**:
  - **Feature Selection**: same as Random Forests.
  - **Splitting Strategy**: Extra-Trees choose the split threshold randomly without searching for the optimal values.

This randomness during the tree-building process makes Extra-Trees more computationally efficient compared to Random Forests.

It is hard to tell in advance whether a **RandomForestClassifier** will perform better or worse than an **ExtraTreesClassifier**. Need to try both and compare them using cross-validation. (Try it out in Python!)

Feature Importance

Another great quality of random forests is that they make it easy to measure the relative importance of each feature.

Scikit-Learn measures a feature's importance by looking at how much the tree nodes that use that feature reduce impurity on average, across all trees in the forest.

More precisely, each node's importance is measured by number of training samples that are associated with it. Scikit-Learn then scales the results so that the sum of all importances is equal to 1.

You can access the result using the **feature_importances_** variable. Let's try it out in Python!

Boosting

Boosting (originally called hypothesis boosting) refers to any ensemble method that can combine several weak learners into a strong learner.

The general idea of most boosting methods is to train predictors sequentially, each trying to correct its predecessor.

Here, we introduce two boosting methods: AdaBoost and Gradient Boosting.

AdaBoost

AdaBoost stands for Adaptive Boosting. The basic idea is that the new predictor corrects its predecessor by paying a bit more attention to the training instances that the predecessor underfitted. This results in new predictors focusing more and more on the hard cases.

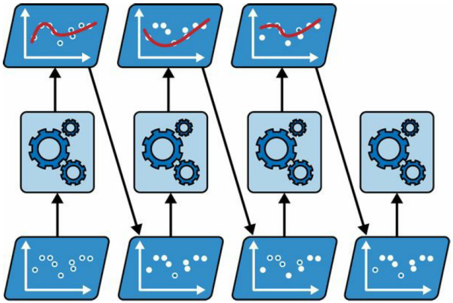


Figure 7-7. AdaBoost sequential training with instance weight updates

Source: Géron, Aurélien. Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow. " O'Reilly Media, Inc.", 2022.

AdaBoost

There is one important drawback to this sequential learning technique: training cannot be parallelized since each predictor can only be trained after the previous predictor has been trained and evaluated. As a result, it does not scale as well as bagging or pasting.

Scikit-Learn uses a multiclass version of AdaBoost called **SAMME** (*Stagewise Additive Modeling using a Multiclass Exponential loss function*). When there are just two classes, SAMME is equivalent to AdaBoost.

If the predictors can estimate class probabilities, Scikit-Learn can use a variant of SAMME called **SAMME.R** (the R stands for “Real”).

Let’s try it out in Python!

Gradient Boosting

Gradient boosting works by sequentially adding predictors to an ensemble, each one correcting its predecessor. This method tries to fit the new predictor to the residual errors made by the previous predictor.

```
# Step 1: fit a DecisionTreeRegressor to the dataset
tree_reg1 = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg1.fit(X, y)
# Step 2: train another decision tree regressor on the residual errors
y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2, random_state=43)
tree_reg2.fit(X, y2)
# Step 3: keep doing it...
y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2, random_state=44)
tree_reg3.fit(X, y3)
# Predict a new instance by adding up the predicitons of all the trees:
X_new = np.array([[-0.4], [0.], [0.5]])
y_pred = sum(tree.predict(X_new) for tree in
                (tree_reg1, tree_reg2, tree_reg3))
```

Gradient Boosting

You can use Scikit-Learn's **GradientBoostingRegressor** class to train GBRT ensembles more easily (there's also a **GradientBoostingClassifier** class for classification).

```
from sklearn.ensemble import GradientBoostingRegressor
gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=3,
learning_rate=1.0, subsample = 1.0, random_state=42)
gbrt.fit(X, y)
```

Let's try it out in Python!

On Hyperparameters in Gradient Boosting

The **learning_rate** hyperparameter scales the contribution of each tree. Instead of fully correcting the error, the predictions of this new tree can be scaled down by the learning rate and then added to the existing model. This incremental update can be represented as:

$$\text{new model} = \text{old model} + \text{learning rate} \times \text{new tree}$$

A low value needs more trees in the ensemble to fit the training set, but the predictions will usually generalize better.

On Hyperparameters in Gradient Boosting

subsample: by default the value is 1. This hyperparameter decides the fraction of samples to be used for fitting a tree. If we pick a value in $(0, 1)$, then this model is called **Stochastic Gradient Boosting**.

n_iter_no_change: the hyperparameter is used for early stopping, which can terminate training when validation score is not improving. By default, it is set to **None** to disable early stopping. If set to a number, it will set aside **validation_fraction** size of the training data as validation and terminate training when validation score is not improving.

Stacking

The **stacking** method is based on a simple idea: instead of using trivial functions (such as hard voting) to aggregate the predictions of all predictors, why not train a model to perform this aggregation?

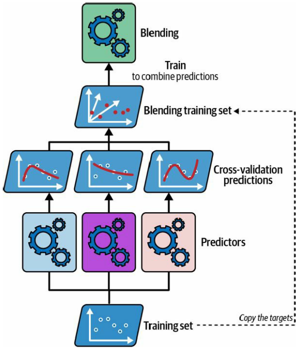


Figure 7-12. Training the blender in a stacking ensemble

Source: Géron, Aurélien. Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow. " O'Reilly Media, Inc.", 2022.

Stacking

Scikit-Learn provides two classes for stacking ensembles: **StackingClassifier** and **StackingRegressor**.

```
from sklearn.ensemble import StackingClassifier
stacking_clf = StackingClassifier(
estimators=[
        ('lr', LogisticRegression(random_state=42)),
        ('rf', RandomForestClassifier(random_state=42)),
        ('svc', SVC(probability=True, random_state=42))
],
final_estimator=RandomForestClassifier(random_state=43),
cv=5 # number of cross-validation folds
)
stacking_clf.fit(X_train, y_train)
```

For each predictor, the stacking classifier will call predict_proba() if available; if not it will fall back to decision_function() or, as a last resort, call predict(). If no final estimator, StackingClassifier will use LogisticRegression and StackingRegressor will use RidgeCV.