Return Prediction Part I

Zichao Yang

Zhongnan University of Economics & Law

Date: April 16, 2025

Roadmap

In this chapter, authors discuss different asset pricing methods.

- Penalized Linear Models
- Dimension Reduction
- Decision Trees
- Neural Networks
- Return Prediction Models For “Alternative” Data

Introduction

A return prediction is, by definition, a measurement of an asset's conditional expected excess return:

$$R_{i,t+1} = E_t[R_{i,t+1}] + \epsilon_{i,t+1}$$

where $E_t[R_{i,t+1}] = E[R_{i,t+1}|\mathcal{I}_t]$, and $\epsilon_{i,t+1}$ collects all he remaining unpredictable variation in returns.

When building statistical models of market data, we need to keep in mind that the data is generated from an extraordinarily complicated process:

A large number of investors who vary widely in their preferences and individual information sets interact and exchange securities to maximize their well-being.

...then how to model $E_t[R_{i,t+1}]$?

Introduction

We make a quantum leap of defining a concrete function to describe its behavior:

$$E_t[R_{i,t+1}] = g^*(z_{i,t})$$

By this notation, the authors represent $E_t[R_{i,t+1}]$ as an immutable but otherwise general function g^* of the P-dimensional predictor variables $z_{i,t}$ available to us.

This function assumes that, once we condition on $z_{i,t}$, the function can fully explains the heterogeneity in expected returns across all assets and over all time, as g^* **depends neither on** i **nor** t.

This framework offers an organizing template for the machine learning literature on return prediction.

Introduction

This framework is different from some standard asset pricing approaches that re-estimate a cross-sectional model each time period, or that independently estimate time series models for each asset (Giglio et al., 2022).

While some economists might view this framework as excessively flexible, others still view it as implausibly restrictive:

- It is difficult to imagine that researchers can condition on the same information set as market participants (Hansen-Richard critique).
- Given the constantly evolving technological and cultural landscape surrounding markets, not to mention the human whims that can influence prices, the concept of a universal $g^*(\cdot)$ seems far-fetched.

In the following sections, we will discuss how to apply ML to discover the $g^*(z_{i,t})$.

Data

Wharton Research Data Services: stock-level panel data.

Global Factor Data: Jensen et al. (2021) construct 153 stock signals, and provide source code and documentation.

Open Source Asset Pricing: Chen and Zimmermann (2021) also post code and data for US stocks.

Experimental Design: Model Selection

Sufficiently large models can fit the training data exactly, hence model selection becomes an essential part of the research process.

As discussed in last lecture, common approaches for model selection are based on information criteria (i.e., AIC, BIC) or cross-validation.

Information criteria aim to choose models that are expected to perform well on new, unseen data by balancing the goodness of fit and model complexity.

Cross-validation has the same goal as AIC and BIC, but approaches the problem in a more data-driven way. It compares models based on their "pseudo"-out-of-sample performance.

Example: Fixed Design

The full sample of T observations is split into three disjoint sub-samples: training sample, validation sample and test sample.

The training sample includes the T_{train} observations from $t = 1, ..., T_{\text{train}}$.

Training sample is used to train candidate models.

The validation sample includes the $T_{\text{validation}}$ observations from $t = T_{\text{train}} + 1, ..., T_{\text{train}} + T_{\text{validation}}$.

The validation sample determines which specific hyperparameter values should be selected.

Example: Fixed Design

Once a model specification is selected its parameters are typically re-estimated using the full $T_{\text{train}} + T_{\text{validation}}$ observations to exploit the full efficiency of the in-sample data.

Finally, the test sample includes the T_{test} observations from $t = T_{\text{train}} + T_{\text{validation}} + 1, ..., T_{\text{train}} + T_{\text{validation}} + T_{\text{test}}$.

Test sample is used for a final evaluation of a method's predictive performance.

Two points are worth highlighting in this simplified design example:

Example: Fixed Design

(1) The final model in this example is estimated once and for all using data through $T_{\text{train}} + T_{\text{validation}}$ **without cross-validation**.

A reason for relying on a fixed sample split would be that the candidate models are very computationally intensive to train. Meanwhile, if the time ordering information is important, it also prevents us from employing cross-validation.

(2) The sample splits in this example respect the **time series ordering** of the data. The motivation for this design is to **avoid inadvertent information leakage** backward in time.

In time series applications, we can even introduce an embargo sample between the training and validation samples so that serial correlation across samples does not bias validation.

Example: Recursive Design

In the fixed design, all points in the test sample are predicted using a model trained on the same dataset. An analyst may wish to use the most **up-to-date sample** to estimate a model and make out-of-sample forecasts. How to achieve that?

To make out-of-sample forecast at t, we use data in $1, ..., t-1$ as the training and validation samples, then the selected model is re-estimated using all data through $t-1$ and an out-of-sample forecast for t is generated.

Move to $t+1$, Training and validation use observations $1, ..., t$, then the selected model produce out-of-sample forecast for $t+1$.

This recursion iterates until a last out-of-sample forecast is generated for observation T.

Example: Recursive Design

This recursive method is also called **Time-ordered Cross-validation**.

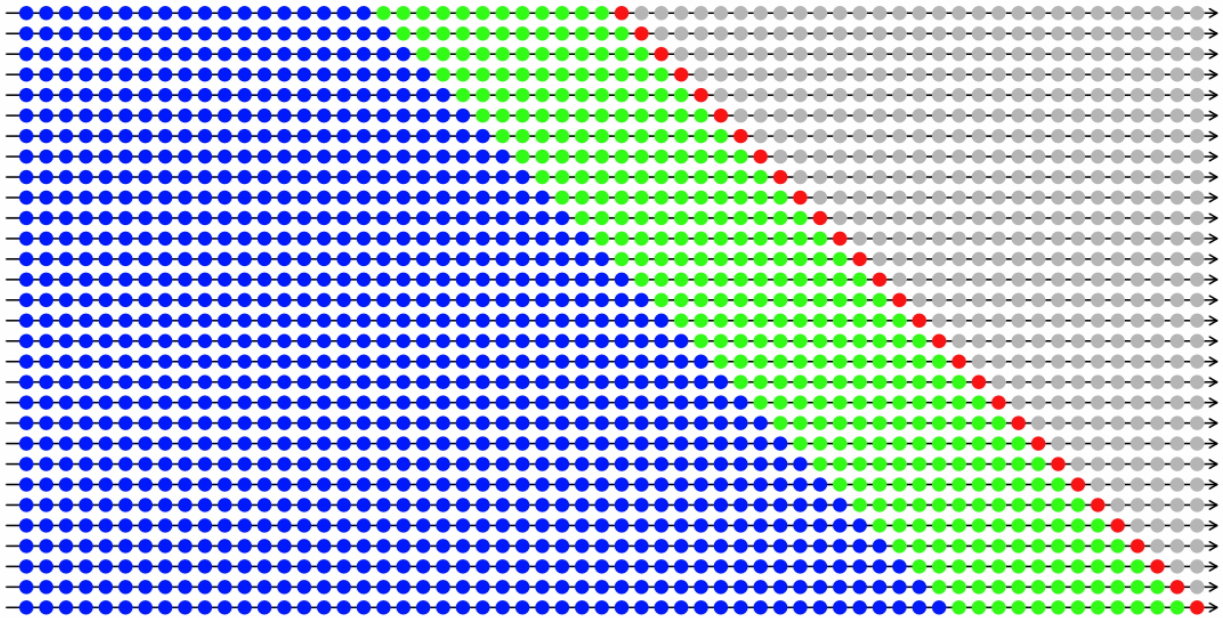


Figure 3.2: Illustration of Recursive Time-ordered Cross-validation

Note: Blue dots represent training observations, green dots represent validation observations, and red dots represent test observations. Each row represents a step in the recursive design. This illustration corresponds to the case of an expanding (rather than rolling) training window.

Example: Recursive Design

Because validation is re-conducted each period, in this recursive design, the selected model can change throughout the recursion.

A common variation on this design is to use **rolling a training window** rather than an **expanding window**.

Rolling window method is beneficial if there is suspicion of structural instability in the data or if there are other modeling or testing benefits to maintaining equal training sample sizes throughout the recursion.

A Benchmark: Simple Linear Models

First, let's take a look at the foundational panel model for stock returns: the simple linear model.

$$R_{i,t+1} = \beta' z_{i,t} + \epsilon_{i,t+1}$$

For example, Fama and Macbeth (1973) regression:

$$R_{it} = \gamma_{0t} + \gamma_{1t}\beta_i + \gamma_{2t}\beta_i^2 + \gamma_{3t}s_i + \eta_{it}$$

where R_{it} is the one-period percentage return on security i from $t-1$ to t. $\beta_i \equiv \frac{cov(R_i,R_m)}{\sigma^2(R_m)}$ is the risk of asset i in the portfolio m, measured relative to the total risk of m, $\sigma^2(R_m)$. s_i captures the risk of security i that is not related to β_i.

A Benchmark: Simple Linear Models

Another two examples: Haugen and Baker (1996) and Lewellen (2015) are precursors to the literature on machine learning for the cross section of returns.

First, they employ a comparatively large number of signals: Haugen and Baker (1996) use roughly 40 continuous variables and sector dummies, and Lewellen (2015) uses 15 continuous variables.

Second, they recursively train the panel linear model above and emphasize the out-of-sample performance of their trained models.

Penalized Linear Models

Later on, the size of a linear model's parameterization quickly ballooned to thousands of parameters and earned the nickname "**factor zoo**" (see Feng et al. 2020).

When the number of predictors P approaches the number of observations NT, the linear model becomes inefficient or even inconsistent. **It begins to overfit noise rather than extracting signal**.

This is particularly troublesome for the problem of return prediction where the **signal-to-noise ratio is notoriously low**.

As we discussed in last lecture, researchers turn to different regularization methods to mitigate overfitting in these complex models.

Penalized Linear Models

Gu et al. (2020) consider a baseline linear specification to predict the panel of stock-month returns.

Their model contains approximately **1,000** predictors that are interactions of roughly 100 stock characteristics and 10 aggregate macro-finance predictors.

They compared the results from a vanilla **OLS model** and from a model that incorporates **elastic net penalization**.

$$\mathcal{L}(\beta;\rho,\lambda)=\sum_{i=1}^{N}\sum_{t=1}^{T}(R_{i,t+1}-\beta' z_{i,t})^2+\lambda(1-\rho)\underbrace{\sum_{j=1}^{P}|\beta_j|}_{Lasso}+\frac{1}{2}\lambda\rho\underbrace{\sum_{j=1}^{P}\beta_j^2}_{Ridge}$$

Penalized Linear Models

Gu et al. (2020) show that OLS cannot achieve a stable fit of a model with so many parameters at once, resulting in disastrous out-of-sample performance. The predictive R^2 is -35% per month. predictive R^2

They further show that, by introducing elastic net penalization. The out-of-sample prediction R^2 becomes positive.

Moral of story: it is not the weakness of the predictive information embodied in the 1,000 predictors, but the **statistical cost** (overfitting and inefficiency) of the heavy parameter burden, that deteriorates the performance of OLS.

Penalized Linear Models

Freyberger et al. (2020) combine penalized regression with a generalized additive model (GAM) to predict the stock-month return.

A **Generalized Additive Model (GAM)** is a statistical model that extends linear models by allowing non-linear functions of the predictor variables while maintaining the additivity of the effects.

In this application, the GAM is defined as:

$$g(z_{i,t}) = \sum_{k=1}^{K} \tilde{\beta}_k' p_k(z_{i,t})$$

where $p_k(z_{i,t})$ is the nonlinear transformation of the P variables in $z_{i,t}$. And the model contains K nonlinear transformations.

Penalized Linear Models

For these K nonlinear transformations, each transformation k has its own $Q \times 1$ vextor of linear regression coefficients $\tilde{\beta}_k$.

Then, Freyberger et al. (2020) apply a penalty function known as group lasso (Huang et al., 2010), which takes the form:

$$\lambda \sum_{j=1}^{Q} \left(\sum_{k=1}^{K} \tilde{\beta}_{k,j}^2 \right)^{1/2}$$

where $\tilde{\beta}_{k,j}^2$ is the coefficient on the k^{th} basis function applied to stock signal j.

Group lasso selects either all K terms associated with a given characteristic j, or none of them.

Penalized Linear Models

Chinco et al. (2019) use Lasso to study return predictability with some unique features:

- high frequency data, forecast one-minute-ahead stock returns using a rolling 30-minute regressions.
- use completely separate models for each stock
- feature set includes three lags of one-minute returns:

$$R_{i,t} = \alpha_i + \beta'_{i,1} R_{t-1} + \beta'_{i,2} R_{t-2} + \beta'_{i,3} R_{t-3} + \epsilon_{i,t}$$

This model setting accommodates cross-stock prediction effects.

The authors find that the dominant predictors vary rather dramatically from period to period.

Dimension Reduction

One drawback of the above penalized linear models is that they may produce **suboptimal forecasts when predictors are highly correlated**.

Imagine a setting in which each predictor is equal to the forecast target plus some i.i.d. noise term. In this situation, instead of keeping some predictors, the sensible forecasting solution is to simply use the average of predictors in a univariate predictive regression.

Here, we will discuss another regularization method: dimension reduction. Two classic techniques are **principal components regression (PCR)** and **partial least squares (PLS)**.

Dimension Reduction: Basic Idea

Suppose there is a generic predictive regression:

$$y_{t+h} = x_t'\theta + \epsilon_{t+h}$$

where y may refer to market returns, x_t is $P \times 1$ vector of predictors, and h is the prediction horizon.

The idea of dimension reduction is to **replace the high-dimensional predictors x_t by a set of low-dimensional "factors"**, f_t, which summarizes useful information in x_t.

$$x_t = \beta f_t + u_t$$

Hence, we can rewrite the original regression model as:

$$y_{t+h} = f_t'\alpha + \tilde{\epsilon}_{t+h}$$

Principal Components Analysis (PCA)

PCA is the common method used to represent a high-dimensional predictors x_t using a set of low-dimensional factors f_t.

These new factors, f_t, are called **principal components**. These principal components are **orthogonal**, and they should capture the **maximum amount of variance** in the data, one after the other.

Principal Components Analysis (PCA)

PCA combines predictors into a few linear combinations, $\hat{f}_t = \Omega_K x_t$, and finds the combination weights Ω_K(aka, PCA).

$$w_j = \arg\max_{w_j} \hat{\text{Var}}(x_t' w_j)$$

$$s.t. \quad w_j' w_j = 1, \quad \hat{\text{Cov}}(x_t' w_j, x_t' w_l) = 0, \quad l = 1, 2, ..., j-1$$

Q: How to interpret this optimization problem?

Principal Components Analysis (PCA)

(1) $x_t'w_j$ can be seen as projecting x_t to w_j. We want these projections capture as much as the variance in original data. The choice of components is not based on the forecasting objective, but aims to best **preserve the covariance structure** among the predictors.

(2) $w_j'w_j = 1$ ensures that w_j represents a principal component direction, and has a unit length.

(3) $\hat{\text{Cov}}(x_t'w_j, x_t'w_l) = 0$ ensures that each new principal component is orthogonal.

How to do PCA in practice?

There are two methods to do PCA in practice: **All at Once Method** and **One by One Method**.

All at Once Method: The standard method via eigen decomposition calculates all principal components by solving all eigenvectors and eigenvalues of the covariance matrix in a single computational step.

Compute the covariance matrix C: $C = \frac{1}{n-1} X'X$ (X is standardized).

Perform eigen decomposition on C: $Cv = \lambda v$, where λ are the eigenvalues and v are the eigenvectors of C.

Sort the eigenvalues and eigenvectors in descending order of λ.

Select the top K principal components based on the sorted eigenvalues.

How to do PCA in practice?

One by One Method: this can be useful in very high-dimensional data. (aka, Power Iteration Method)

- Initialization: begin with an arbitrary non-zero vector b_0 that has unit norm.
- Iteration: Repeatedly update b_k by applying the covariance matrix C and then normalizing: $b_{k+1} = \frac{Cb_k}{\|Cb_k\|}$ Large Eigenvalue Dominance Effect
- Convergence: b_k converges to the eigenvector associated with the largest eigenvalue.
- Estimation of the Largest Eigenvalue: $\lambda = b_k' C b_k$
- Deflation: To find the next principal components, modify the covariance matrix to remove the influence of the components already identified. $C' = C - \lambda b_k b_k'$
- Repeat above steps to find the second largest eigenvector.

Principal Components Regression (PCR)

Principal components regression (PCR) is a two-step procedure:

Step One: conducts PCA, combines predictors into a few linear combinations, $\hat{f}_t = \Omega_K x_t$, and finds the combination weights Ω_K.

Step Two: uses the estimated components $\hat{f}_t$ in a standard linear predictive regression.

Let's try it out in python!

Principal Components Regression (PCR)

Jurado et al. (2015) use a clever application of PCR to estimate macroeconomic risk.

The authors argue that a good conditional variance measure must effectively adjust for conditional means,

$$\text{Var}(y_{t+1}|\mathcal{I}_t) = E[(y_{t+1} - E[y_{t+1}|\mathcal{I}_t])^2|\mathcal{I}_t]$$

If the amount of mean predictability is underestimated, conditional risks will be overestimated. And they show that:

- Episodes of uncertainty are fewer and farther between than previously believed.
- Reveal a tight link between rises in risk and depressed macroeconomic activity.

Principal Components Regression (PCR)

PCR constructs predictors solely based on covariation among the predictors.

The idea is to accommodate as much of the total variation among predictors as possible using a relatively small number of dimensions.

PCR fails to consider the ultimate forecasting objective in how it conducts dimension reduction.

So, let's turn to Partial Least Squares (PLS).

Partial Least Squares (PLS)

Partial least squares (PLS) directly exploits covariation between predictors and the forecast target.

PLS seeks components of X that maximize predictive correlation with the forecast target, thus weight in the j^{th} PLS component are:

$$w_j = \arg\max_{w} \hat{\text{Cov}}(y_{t+h}, x_t' w),$$
$$s.t. \quad w'w = 1, \quad \hat{\text{Cov}}(x_t' w, x_t' w_l) = 0, \quad l = 1, 2, ..., j-1$$

At the core of PLS is a collection of univariate ("partial") models that forecast y_{t+h} one predictor at a time. Then, it constructs a linear combination of predictors weighted by their univariate predictive ability.

Partial Least Squares (PLS)

Kelly and Pruitt (2015) shows that PLS is resilient when the predictor set contains dominant factors that are irrelevant for prediction, which is a situation that limits the effectiveness of PCR.

The authors argue that the PLS estimator learns to bypass these high variance but low predictability components in favor of components with stronger return predictability. (advantage compared to PCR)

Let's try it out in python!

Pervasive Factor Model VS. Weak Factor Model

A **pervasive factor model** assumes that factors influencing the model's outcomes are broad-based and have a significant impact across a wide range of observations or variables.

- **Factors are influential**: Factors can explain a large portion of the variance in the data.
- **High detectability**: Factors are relatively easier to identify through statistical techniques like PCA.

A **weak factor model** deals with factors that have a subtle or minor impact on the model's outcomes.

- **Factors are less influential**: These factors account for a smaller portion of the variance in the data. They might influence only certain sectors, assets, etc.
- **Lower detectability**: weak factors are more challenging to detect and quantify accurately.

Scaled PCA

Giglio et al. (2022) show that the performance of the PCA and PLS predictors hinges on the signal-to-noise ratio, the performance is inconsistent when we have the weak factor problem.

Huang et al. (2021) propose a scaled-PCA procedure, which assigns weights to variables based on their correlations with the prediction target, before applying PCA.

The weighting scheme enhances the signal-to-noise ratio and thus helps factor recovery.

Supervised PCA

Giglio et al. (2022) propose a supervised PCA (SPCA):

- Select a subset of predictors within which at least one factor is strong.
- Extract the first factor from the subset using PCA, projects the target and all the predictors, on the first factor, and constructs residuals.
- Repeat above step with the residuals, extracting factors one by one until the correlations between residuals and the target vanish.

Tree Models

Modern asset pricing models suggests that **interaction effects** are potentially important.

However, lacking a prior assumptions about the relevant interactions, adding multi-way interactions quickly builds up **computational overhead**.

So is there a better way to tackle the problem?

Decision trees and **Random Forest** provide a way to incorporate multi-way predictor interactions at much lower computational cost.

Decision Trees Related Literature

Rossi and Timmermann (2015) use boosted trees to predict the realized covariance between a daily index of aggregate economic activity and the daily return on the market portfolio. They show that these conditional covariance estimates have a significantly positive association with the conditional equity risk premium, and imply investor risk aversion.

Rossi (2018) uses boosted regression trees with macro-finance predictors to directly forecast aggregate stock returns (and volatility) at the monthly frequency.

Random Forest Related Literature

Moritz and Zimmermann (2016) conduct conditional portfolio sorts by estimating regression trees from a large collection of stock characteristics, rather than using pre-defined sorting variables and break points.

Built on the above work, Bryzgalova et al. (2020) use a method they call "AP-trees" to conduct portfolio sorts.

Cong et al. (2022) push the frontier of asset pricing tree methods by introducing the Panel Tree (P-Tree) model.

Tree Models

In the following part, we will discuss in detail two popular tree models:

- Decision Trees
- Random Forest and Ensemble Learning

Decision Trees

Trees partition data observations into groups that share common feature interactions. **The logic** is that, by finding homogenous groups of observations, one can use past data for a given group to forecast the behavior of a new observation that arrives in the group.

The general prediction function associated with a tree of K "leaves" nodes (aka, end nodes) and depth L is:

$$g(z_{i,t};\theta,K,L)=\sum_{k=1}^{K}\theta_k\mathbf{1}_{\{z_{i,t}\in C_k(L)\}}$$

where $C_k(L)$ is one of the K partitions, and it is a function of the depth L. θ_k is the sample average of outcomes within the partition k. $g(.)$ is the predicted value given feature $z_{i,t}$.

Training and Visualizing a Decision Tree

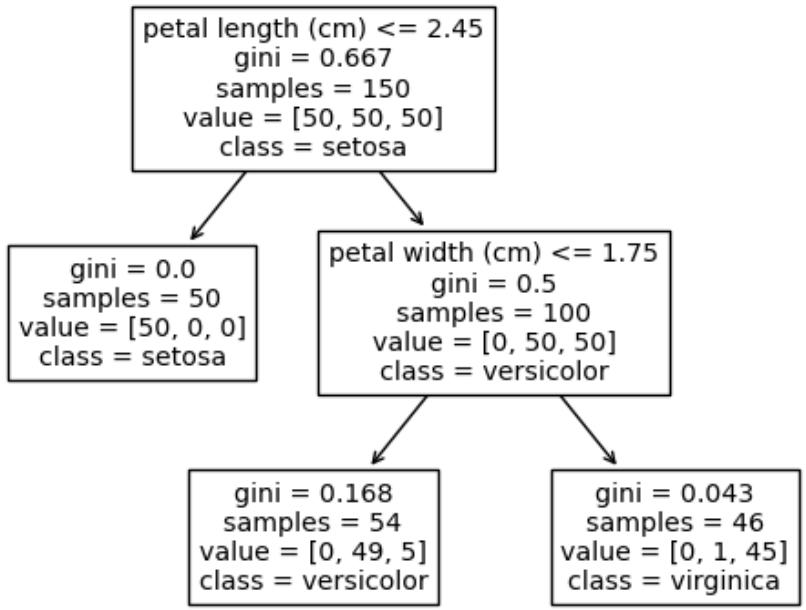
(1) Train a Decision Tree model:

```
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.

User Guide:
(1) Decision Tree Classifier
(2) Decision Tree Regressor

(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.

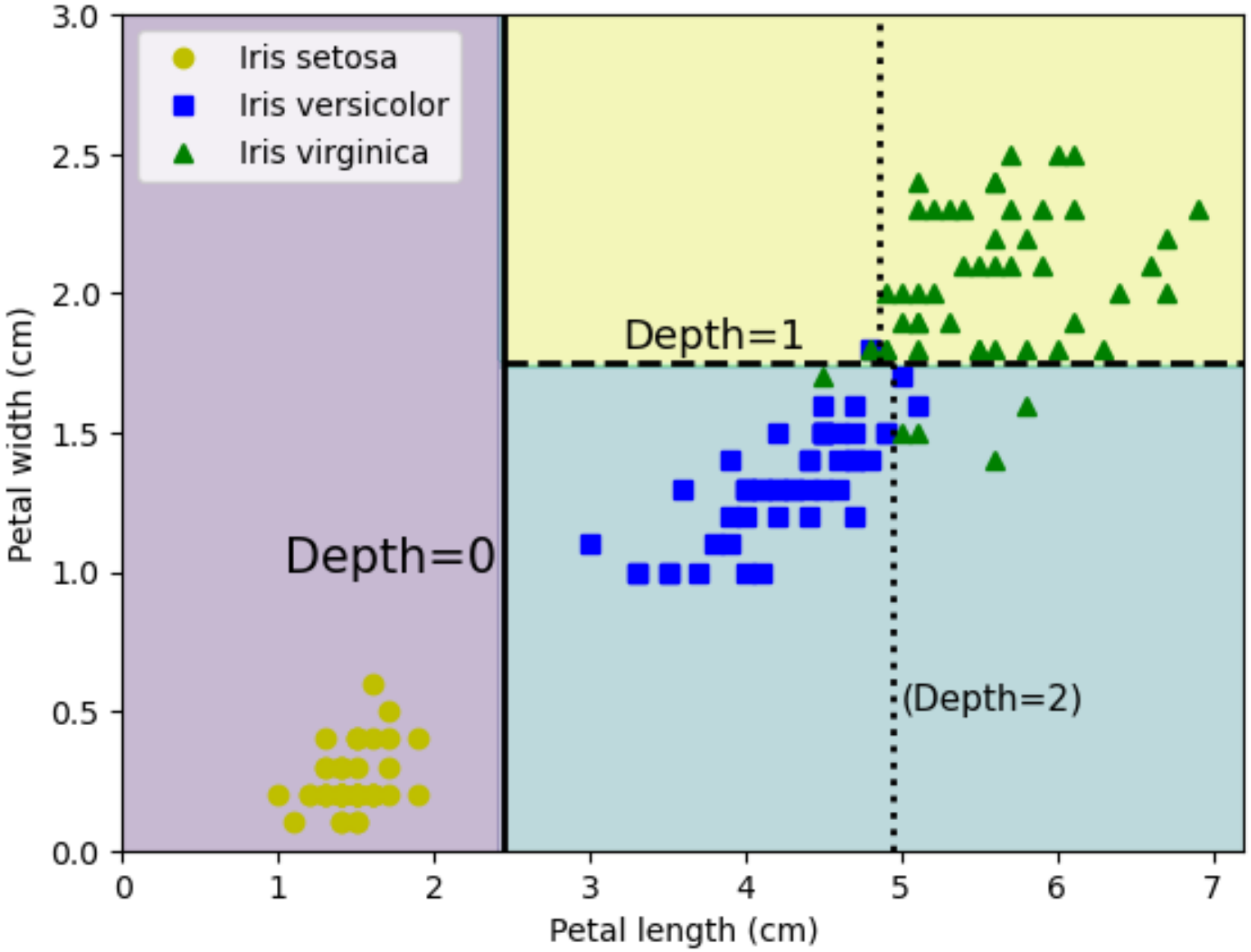
value: tells us how many trianing instances of each class this node applies to. For example, [0, 1, 45] means this node applies to 0 setosa, 1 versicolor, and 45 virginica.

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.

```
# predict 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 ≤ 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:

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

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 Decision 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:

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

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

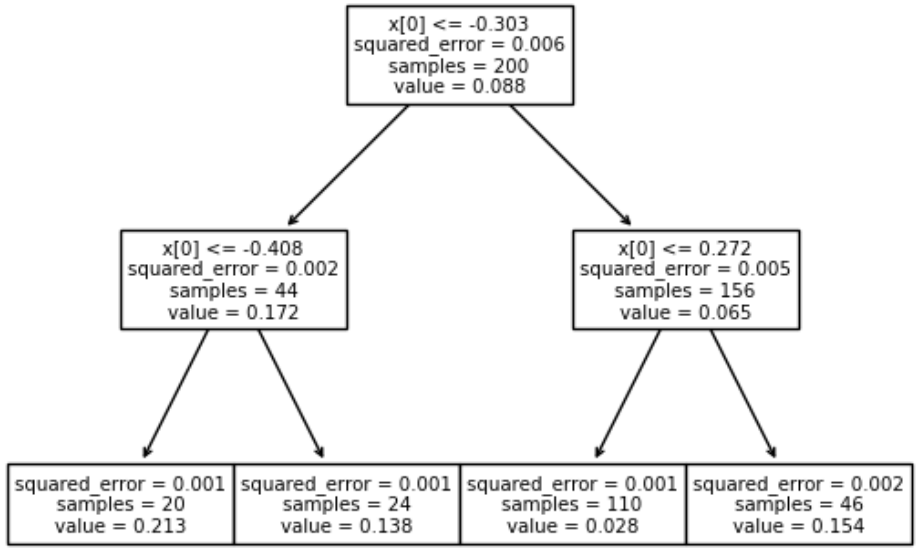
The CART algorithm works mostly the same way as before, except that instead of trying to split the training set in a way that minimizes impurity, it now tries to split the training set in a way that minimizes the MSE.

Decision Tree Regression

(1) Decision Tree Regression:

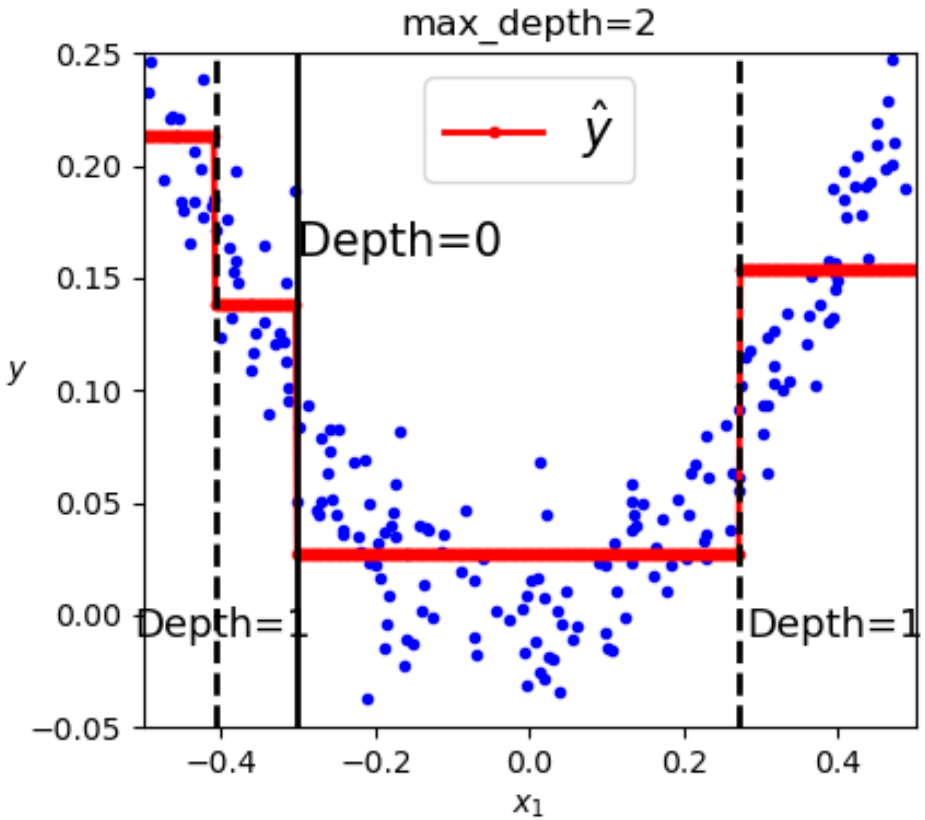
```
from sklearn import tree
from sklearn.tree import DecisionTreeRegressor

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)
```

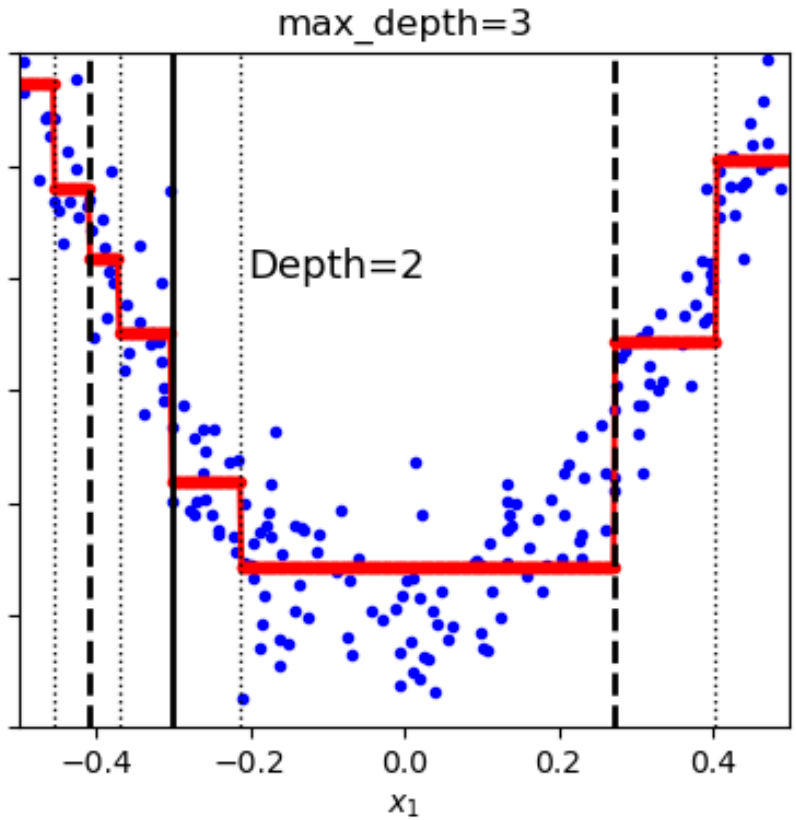
(2) Regression Visualization:



Decision Tree Regression Illustration

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

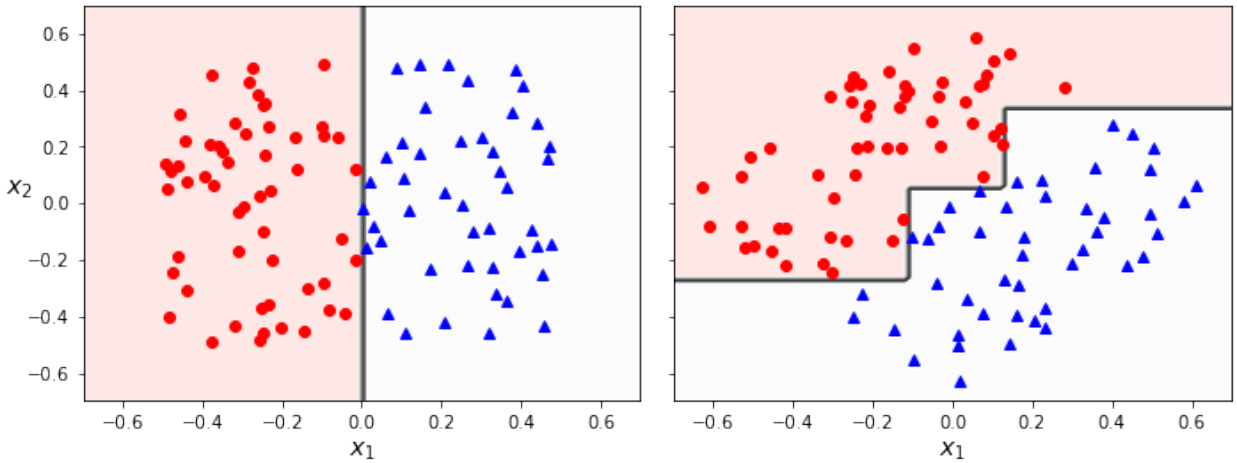




Drawback: Instability

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.

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

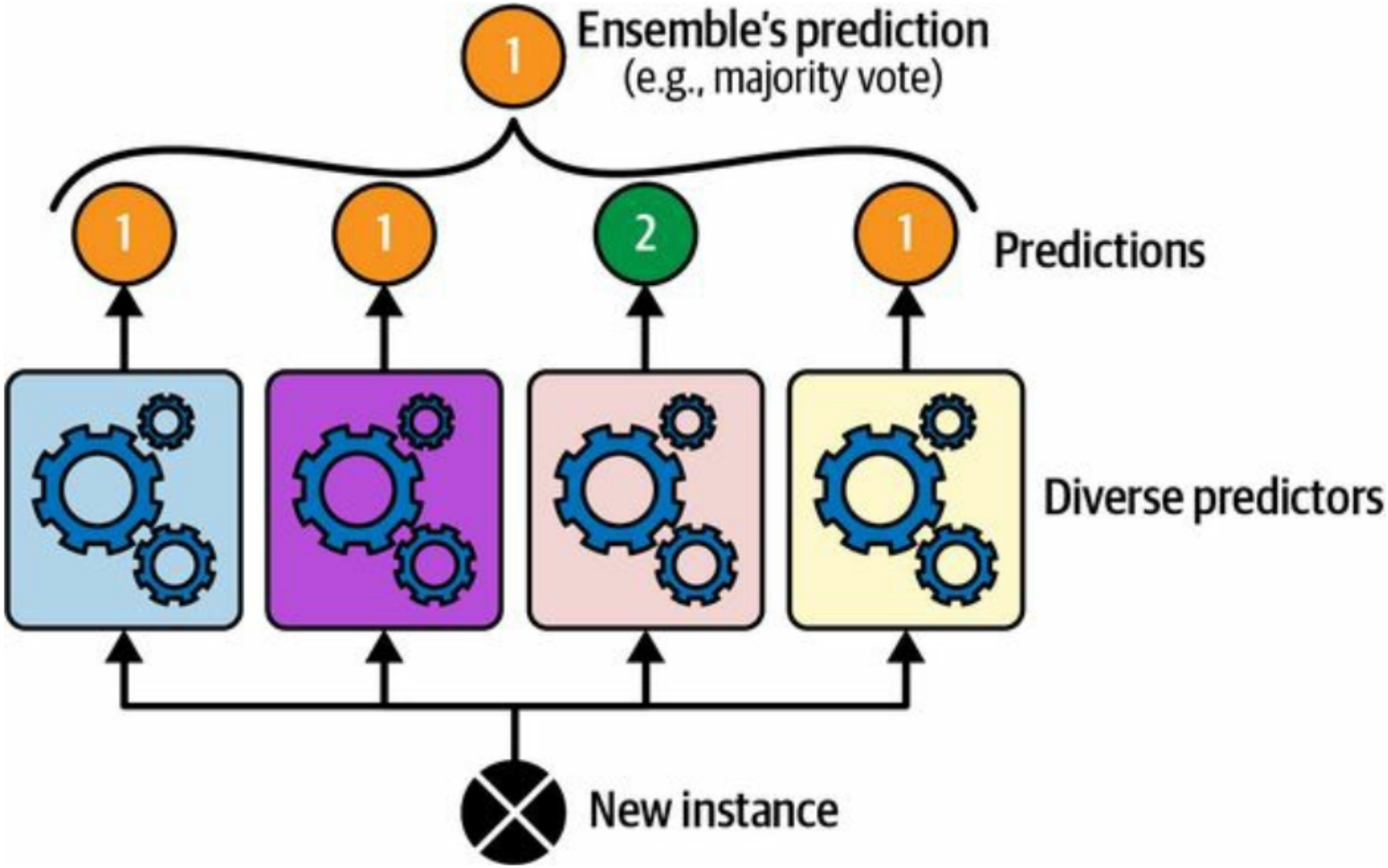


Figure 7-2. Hard voting classifier predictions

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!

Ensemble Learning with Cross Validation

Scikit-learn provides the method **VotingClassifier()** for conducting ensemble learning.

```
from sklearn.ensemble import VotingClassifier
voting_clf = VotingClassifier(
        voting='hard',  # hard voting
        estimators=[
                ('lr', LogisticRegression(random_state=42)),
                ('rf', RandomForestClassifier(random_state=42)),
                ('svc', SVC(random_state=42))
        ]
)
voting_clf.fit(X_train, y_train)
```

VotingClassifier User Guid

Let's try it out in python.

Ensemble Learning with Cross Validation

In ensemble learning, we need to pick the hyperparameters for each model. Obviously, cross validation is an essential part in employing ensemble learning.

How to introduce cross validation into ensemble learning?

```
from sklearn.model_selection import GridSearchCV
GridSearchCV()
```

GridSearchCV User Guid

Can you modify the previous **VotingClassifier()** code, for example, change the method to soft voting, and add KNeighborsClassifier(), GaussianNB() and Cross Validation? 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

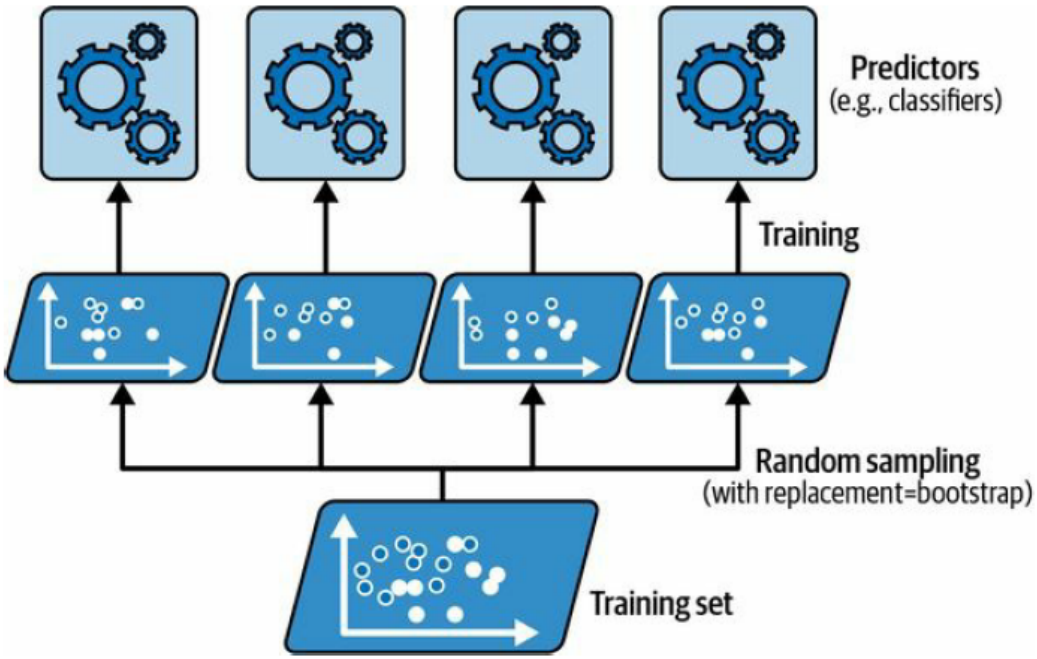


Figure 7-4. Bagging and pasting involve training several predictors on different random samples of the training set

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

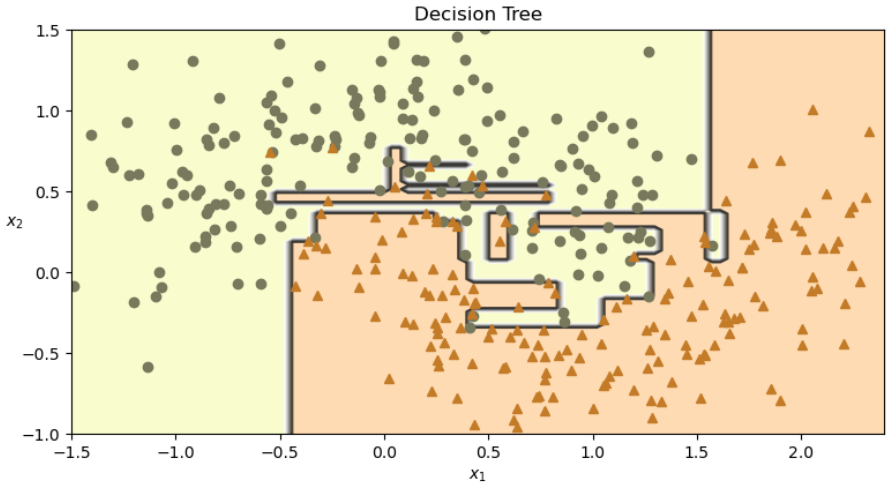
A1: if we directly use the whole 500 instances to train the 500 Decision Tree classifiers, eventually these 500 classifiers will be identical, and we won't be able to detect nuances in different sub-samples. That is why here we randomly sample 100 instances to train each classifier.

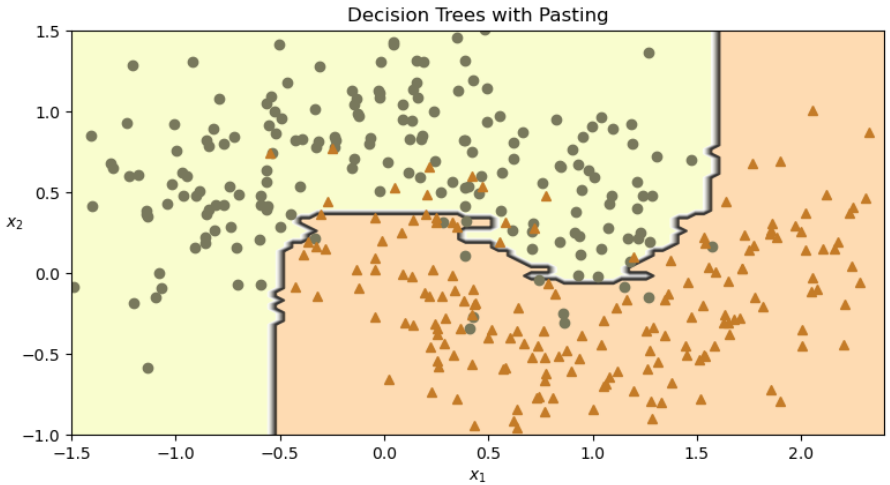
A2: The difference lies in how to randomly sample these 100 instance. If these 100 instances are sampled with replacement (i.e., bootstrapping = True), then the method is called bagging; if not, then it is called pasting.

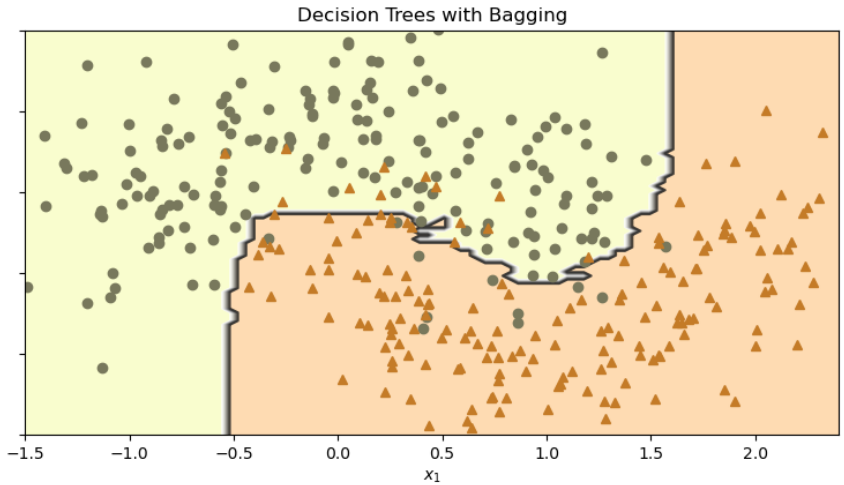
Bagging introduces a bit more diversity in the subsets that each predictor is trained on, so bagging ends up with a slightly higher bias but less variance. Overall, bagging is often preferred over pasting.

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 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 the Gradient Boosting method.

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, subample = 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?

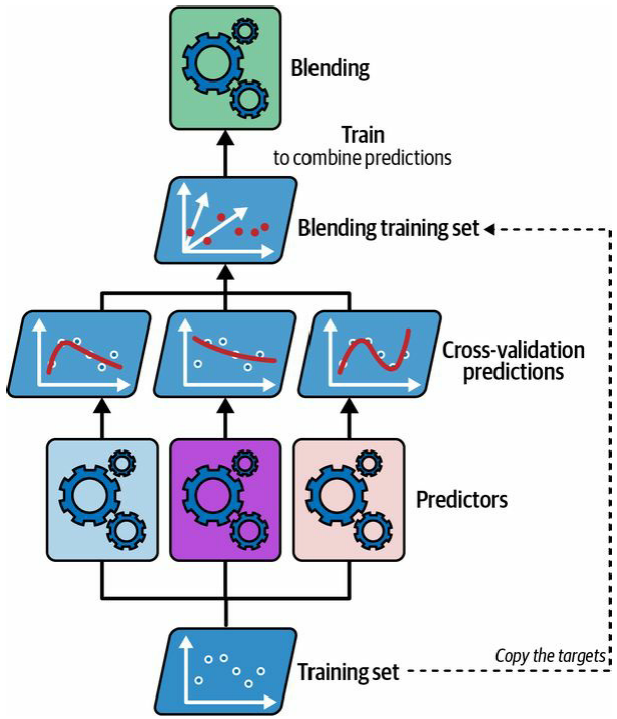


Figure 7-12. Training the blender in a stacking ensemble

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.

Extra: Predictive R^2 vs. Traditional R^2

Traditional R^2 is used to assess how well the regression model explains the variability of the observed data. High R^2 values indicate that a large proportion of the variance in the dependent variable is explained by the model. However, a high R^2 does not necessarily imply that the model has good predictive accuracy on new, unseen data.

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

$SS_{res} = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2$, it measures the variance in the observed data that is not explained by the model.

$SS_{TOT} = \sum_{i=1}^{n}(y_i - \bar{y}_i)^2$, where $\bar{y}$ is the mean of the observed data. It measures the total variance in the observed data.

Extra: Predictive R^2 vs. Traditional R^2

Predictive R^2 focuses on the model's predictive power on new, unseen data.

$$R^2 = 1 - \frac{SS_{test,res}}{SS_{test}}$$

$SS_{test,res} = \frac{1}{n_{test}} \sum_{i=1}^{n_{test}} (y_{test,i} - \hat{y}_{test,i})^2$, is the out-of-sample mean squared prediction error (MSPE).

$SS_{test} = \frac{1}{n_{test}-1} \sum_{i=1}^{n_{test}} (y_{test,i} - \bar{y}_{test,i})^2$, where $\bar{y}_{test}$ is the mean of the actual outcomes in the test set.

Extra: Large Eigenvalue Dominance Effect

When you repeatedly apply a matrix C to an arbitrary vector b_k, and normalize the result at each step, the direction of the resulting vector increasingly aligns with the eigenvector associated with the largest eigenvalue of C.

Intuition behind this effect:

(1) The process exploits the fact that the dominant eigenvalue's influence grows exponentially faster than that of the other eigenvalues.

(2) This alignment with the dominant eigenvector happens because, in essence, each iteration amplifies the component of the initial vector b_k in the direction of the dominant eigenvector more than those in the directions of other eigenvectors.

Extra: Large Eigenvalue Dominance Effect

What is the intuition behind the operation $C \cdot b_k$?

The covariance matrix C summarizes how variables change together and represents the data's structure in terms of variability.

The operation $C \cdot b_k$ transforms b_k according to the variances and covariances encoded in C. It's akin to asking, "If I look at the dataset in the direction defined by b_k, how does the dataset's inherent variability stretch or compress this direction?"

Each iteration of $C \cdot b_k$ followed by normalization amplifies the component of b_k that aligns with the direction of maximum variance in the data. Over iterations, this process ensures that b_k converges to the principal eigenvector of C, which is the direction along which the data varies the most.

References

- Giglio, S., B. Kellly, and D. Xiu. (2022). "Factor Models, Machine Learning, and Asset Pricing". Annual Review of Financial Economics. 14: 1–32.
- Jensen, T. I., B. Kelly, and L. H. Pedersen. (2021). "Is There a Replication Crisis in Finance?" Journal of Finance
- Chen, A. Y. and T. Zimmermann. (2021). "Open Source Cross-Sectional Asset Pricing". Critical Finance Review
- Fama, E. F. and J. D. Macbeth. (1973). "Risk, Return, and Equilibrium: Empirical Tests". Journal of Political Economy. 81(3): 607–636.
- Haugen, R. A. and N. L. Baker. (1996). "Commonality in the determinants of expected stock returns". Journal of Financial Economics. 41(3): 401–439.

References

- Lewellen, J. (2015). "The Cross-section of Expected Stock Returns". Critical Finance Review. 4(1): 1–44.
- Feng, G., Giglio, S., and Xiu, D. (2020). Taming the factor zoo: A test of new factors. The Journal of Finance, 75(3), 1327-1370.
- Gu, S., B. Kelly, and D. Xiu. (2020). "Empirical asset pricing via machine learning". The Review of Financial Studies. 33(5): 2223–2273.
- Freyberger, J., A. Neuhierl, and M. Weber. (2020). "Dissecting characteristics nonparametrically". The Review of Financial Studies. 33(5): 2326–2377.
- Chinco, A., A. D. Clark-Joseph, and M. Ye. (2019). "Sparse Signals in the Cross-Section of Returns". Journal of Finance. 74(1): 449–492.

References

- Kelly, B. and S. Pruitt. (2015). "The three-pass regression filter: A new approach to forecasting using many predictors". Journal of Econometrics. 186(2): 294–316. issn: 0304-4076.
- Jurado, K., S. C. Ludvigson, and S. Ng. (2015). "Measuring uncertainty". The American Economic Review. 105(3): 1177–1216.
- Huang, D., F. Jiang, K. Li, G. Tong, and G. Zhou. (2021). "Scaled PCA: A New Approach to Dimension Reduction". Management Science, forthcoming.
- Giglio, S., D. Xiu, and D. Zhang. (2022). "Prediction when Factors are Weak". Tech. rep. Yale University and University of Chicago.

References

- Rossi, A. G. and A. Timmermann. (2015). "Modeling Covariance Risk in Merton's ICAPM". The Review of Financial Studies. 28(5): 1428–1461.
- Rossi, A. G. (2018). "Predicting stock market returns with machine learning". Georgetown University.
- Moritz, B. and T. Zimmermann. (2016). "Tree-Based Conditional Portfolio Sorts: The Relation Between Past and Future Stock Returns". Tech. rep. Ludwig Maximilian University Munich.
- Bryzgalova, S., M. Pelger, and J. Zhu. (2020). "Forest through the Trees: Building Cross-Sections of Asset Returns". Tech. rep. London School of Business and Stanford University.
- Cong, L. W., G. Feng, J. He, and X. He. (2022). "Asset Pricing with Panel Tree Under Global Split Criteria". Tech. rep. City University of Hong Kong.