Hands on Machine Learning part I

THE FUNDAMENTALS OF MACHINE LEARNING

(Student notes)

Book by: **Aurélien Géron**

Author: Ihona Maria Correa

1. INTRODUCTION

Machine Learning is about making machines get better at some task by learning from data, instead of having to explicitly code rules.

Machine Learning systems can be classified based on several factors including how they learn, the way they learn over time, and how they make predictions. Here's a brief overview:

2. **Based on How They Learn:**

    a. **Supervised Learning:** In this type, the model is trained with labeled data. It learns to predict outputs from input data. Example applications include image recognition and spam filtering.
    b. **Unsupervised Learning:** The model is trained with unlabeled data and learns to identify patterns or structures in the data. Common applications include clustering and dimensionality reduction.
    c. **Semi-supervised Learning:** This involves a combination of a small amount of labeled data and a large amount of unlabeled data. It's useful when acquiring a fully labeled dataset is expensive or time-consuming.
    d. **Reinforcement Learning:** The model learns to make decisions by performing actions in an environment to achieve a goal. It is commonly used in robotics, gaming, and navigation systems.

3. **Based on the Way They Train Over Time:**

    a. **Batch Learning:** The model is trained using all available data at once. This is a one-time training process, and the model doesn’t learn incrementally from new data unless it is retrained.
    b. **Online Learning:** In this approach, the model is trained incrementally by continuously feeding it data samples. This method is useful for systems that receive data as a continuous flow and need to adapt to change rapidly.

4. **Based on the Way They Make Predictions:**

    a. **Instance-based Learning:** The system learns the examples by heart and then uses a similarity measure to make predictions. It makes decisions based on the most similar learned instances.
    b. **Model-based Learning:** The system builds a model from the sample data and then uses that model to make predictions. This involves understanding the underlying patterns or structures in the data

A Machine Learning project looks like this:

- Study the data (EDA)
- Select a Model
- Train it on the train data
- Apply the model to make predictions on new cases (“inference”). hoping the model will generalize well in this new cases.

Main challenges in ML:

- Insufficient quantity of training data
- Non-representative training data
- Poor-quality data (garbage in- grange out)
- Irrelevant features
- Overfitting and underfitting
- Data mismatch

DATA-PREPARATION

5. Data cleaning: Removing or replacing null-values.

   simple imputer: removes the missing value and replaces it

   dropna(), drop(), and fillna()

   Scikit-Learn provides a handy class to take care of missing values: SimpleImputer.
   from sklearn.impute import SimpleImputer
   imputer = SimpleImputer(strategy="median")
   imputer.fit(housing_num) #fit the imputer on the training data

6. Feature scaling: Machine Learning algorithms don’t perform well when numerical attributes have very different scales. Note that scaling the target values is generally not required. There are two common ways to get all attributes to have the same scale: min-max scaling and standardization.

   Min-max scaling (many people call this normalization) is the simplest: values are shifted and rescaled so that they end up ranging from 0 to 1.

Standardization is different: first it subtracts the mean value (so standardized values always have a zero mean), and then it divides by the standard deviation so that the resulting distribution has unit variance. Unlike min-max scaling, standardization does not bound values to a specific range, which may be a problem for some algorithms (e.g., neural networks often expect an input value ranging from 0 to 1). However, standardization is much less affected by outliers. (X-mu/sd)

7. Code non-numeric values (One hot encoding, dummy coding, label encoding)

MODEL EVALUATION

The only way to know how well a model will generalize to new cases is to actually try it out on new cases. You typically split your data into two sets: the training set (80% normally) and the test set (20%). As these names imply, you train your model using the training set, and you test it using the test set. The error rate on new cases is called the generalization error (or out-of sample error), and by evaluating your model on the test set, you get an estimate of this error. This value tells you how well your model will perform in instances it has never seen before.

To do data division with sklearn: function [train_test_split]

from sklearn.model_selection import train_test_split

train_test_split is a random sampling method. This is generally fine if your dataset is large enough (especially relative to the number of attributes), but if it is not, you run the risk of introducing a significant sampling bias (the sampling bias happens when the samples fail to represent the whole population). Stratified sampling in ML is a technique used to create more balanced training and test sets, ensuring that all categories of the data are adequately represented, thus leading to more robust and accurate models. You should not have too many strata (groups), and each stratum should be large enough.

from sklearn.model_selection import StratifiedShuffleSplit

If the training error is low (i.e., your model makes few mistakes on the training set) but the generalization error is high, it means that your model is overfitting the training data.

- Overfitting: Means that the model performs well on the training data, but it does not generalize well on unseen data. Overfitting happens when the model is too complex relative to the amount and noisiness of the training data. Here are possible solutions:

  • Simplify the model by selecting one with fewer parameters (e.g., a linear model rather than a high-degree polynomial model), by reducing the number of attributes in the training data, or by constraining the model.

• Gather more training data.

• Reduce the noise in the training data (e.g., fix data errors and remove outliers).

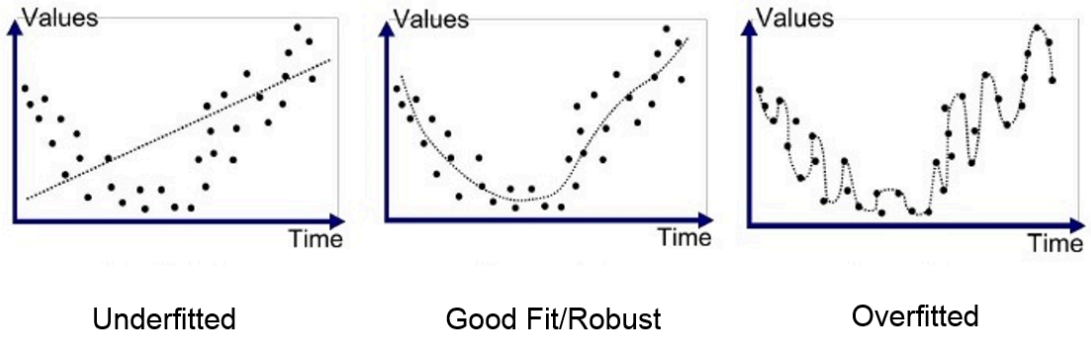
• Regularization: Reduce the degrees of freedom (The amount of regularization to apply during learning can be controlled by a hyperparameter).

- Underfitting: it occurs when your model is too simple to learn the underlying structure of the data. Possible solutions:

  • Select a more powerful and complex model, with more parameters.

  • Feed better features to the learning algorithm (feature engineering).

  • Reduce the constraints on the model (e.g., reduce the regularization hyperparameter).



Evaluating a model is simple enough: just use a test set. But suppose you are hesitating between two types of models : how can you decide between them?

HYPERPARAMETER TUNNING

in Machine Learning (ML) is a crucial process that involves adjusting the parameters of a learning algorithm to optimize its performance. These parameters, known as hyperparameters, are not learned from the data but are set prior to the training process. They control the overall behavior of the learning algorithm and can significantly impact the performance of the model.

For example, if you want to do some regularization to prevent overfitting, you can test different values for the hyperparameter and see which one performs better.

The best way to do it is simply hold out part of the training set to evaluate several candidate models and select the best one.
The new held-out set is called the validation set (or sometimes the development set, or dev set).

1. You train multiple models with various hyperparameters on the reduced training set (i.e., the full training set minus the validation set).
2. Select the model that performs best on the validation set.
3. After this holdout validation process, you train the best model on the full training set (including the validation set), and this gives you the final model.
4. Lastly, you evaluate this final model on the test set to get an estimate of the generalization error.

Grid Search and *Randomized Search* are two popular methods used in machine learning for hyperparameter tuning, which is the process of finding the optimal set of parameters for a learning algorithm.

- **Grid -search:**

Grid Search systematically explores multiple combinations of hyperparameters. It creates a grid of all possible combinations of the hyperparameters you want to test and then evaluates the model performance for each combination. For example, if you have two hyperparameters, each with three possible values, Grid Search will evaluate the model 3 x 3 = 9 times. It's exhaustive and ensures that you don't miss the best combination, but it can be slow and has a high computational cost.

When you have no idea what value a hyperparameter should have, a simple approach is to try out consecutive powers of 10

- **Randomized Search:**

Randomly selects combinations of hyperparameters to test. Instead of trying every possible combination like Grid Search, it samples a specified number of combinations from the parameter space. It's much faster than Grid Search when dealing with a large hyperparameter space. It can also sometimes find a good combination of hyperparameters more quickly and efficiently. However, There's a chance it might miss the optimal combination, especially if the number of iterations is too low.

- Before final evaluation, it is a good practise to do the cross-validation:

Cross-validation is a crucial technique in machine learning used to assess the generalizability of a model to unseen data. It works by dividing the dataset into multiple parts and ensures that the model is trained and validated on different subsets of these parts. The entire dataset is divided into 'k' equal parts or folds. Common choices for 'k' include 5 or 10.

We use cross-validation to get an estimate of a model's generalization performance:

- ➔ *If a model performs well on the training data but generalizes poorly according to the cross-validation metrics, then your model is overfitting. (training data > cross-val)*
- ➔ *If it performs poorly on both, then it is underfitting.*

Your next step is to select a performance measure. A typical performance measure for regression problems is the Root Mean Square Error (**RMSE**). It gives an idea of how much error the system typically makes in its predictions, with a higher weight for large errors. Even though the RMSE is generally the preferred performance measure for regression tasks, in some contexts you may prefer to use another function. For example, suppose that there are many outlier districts. In that case, you may consider using the mean absolute error (**MAE**, also called the average absolute deviation. RMSE is more sensitive to outliers than the MAE. But when outliers are exponentially rare (like in a bell-shaped curve), the RMSE performs very well and is generally preferred.

END-TO-END ML project:

1. Look at the big picture.
2. Get the data.
3. Discover and visualize the data to gain insights.
4. Prepare the data for Machine Learning algorithms.
5. Select a model and train it.
6. Fine-tune your model.
7. Present your solution.
8. Launch, monitor, and maintain your system.

A sequence of data processing components is called a data pipeline. Pipelines are very common in Machine Learning systems, since there is a lot of data to manipulate and many data transformations to apply.

Scikit-Learn's API

- Estimators: Any object that can estimate some parameters based on a dataset is called an estimator (e.g., an imputer is an estimator). The estimation itself is performed by the fit() method.

- Transformers: Some estimators (such as an imputer) can also transform a dataset; these are called transformers. Once again, the API is simple: the transformation is performed by the transform() method with the dataset to transform as a parameter.

  It returns the transformed dataset. All transformers also have a convenience method called *fit_transform()* that is equivalent to calling fit() and then transform().
  You can also have **custom transformers**.

- Predictors: Finally, some estimators, given a dataset, are capable of making predictions.For example, the LinearRegression model is a predictor.

There are many data transformation steps that need to be executed in the right order. Fortunately, Scikit-Learn provides the **Pipeline** class to help with such sequences of transformation.

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

num_pipeline = Pipeline([
        ('imputer', SimpleImputer(strategy="median")),
        ('attribs_adder', CombinedAttributesAdder()),
        ('std_scaler', StandardScaler()),
    ])

housing_num_tr = num_pipeline.fit_transform(housing_num)
```

When you call the pipeline's fit() method, it calls fit_transform() sequentially on all transformers, passing the output of each call as the parameter to the next call until it reaches the final estimator.

If the data keeps evolving, you will need to update your datasets and retrain your model regularly. You should probably automate the whole process as much as possible.

Here are a few things you can automate:

• Collect fresh data regularly and label it (e.g., using human raters).
• Write a script to train the model and fine-tune the hyperparameters automatically. This script could run automatically, for example every day or every week, depending on your needs.
• Write another script that will evaluate both the new model and the previous model on the updated test set, and deploy the model to production if the performance has not decreased (if it did, make sure you investigate why).
• You should also make sure you evaluate the model's input data quality.

2. REGRESSION

A. LINEAR REGRESSION

A linear model makes a prediction by simply computing a weighted sum of the input features, plus a constant called the bias term (also called the intercept term).

- **Purpose:** Used to predict a continuous dependent variable based on one or more independent variables.

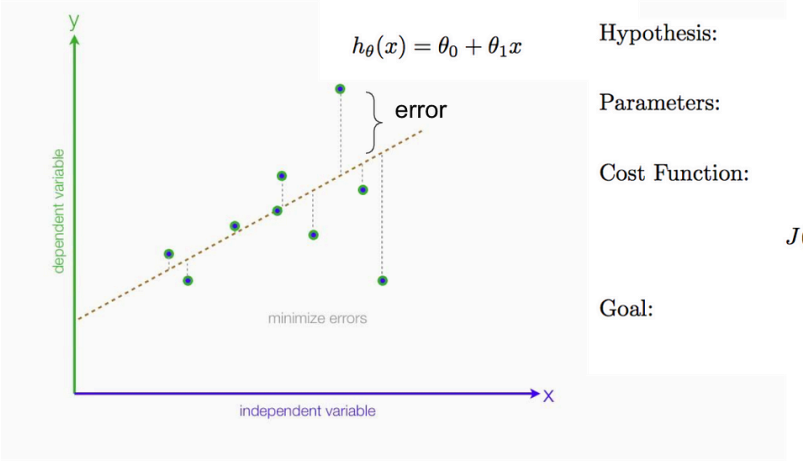
- **Assumption:** Assumes a linear relationship between the input(s) (independent variables) and the output (dependent variable).

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

In this equation:

- $\hat{y}$ is the predicted value.
- n is the number of features.
- x_i is the i^{th} feature value.
- θ_j is the j^{th} model parameter (including the bias term θ_0 and the feature weights $\theta_1, \theta_2, \cdots, \theta_n$).

In summary, linear regression uses a hypothesis function to model the relationship between variables, a cost function to measure the accuracy of this model, and an optimization algorithm like gradient descent to find the best parameters that minimize the cost function. This process allows for the prediction of continuous outcomes based on the values of independent variables.



The most common performance measure of a regression model is the Root Mean Square Error (RMSE). Therefore, to train a Linear Regression model, we need to find the value of θ that minimizes the RMSE. In practice, it is simpler to minimize the mean squared error (**MSE**) than the RMSE, and it leads to the same result (because the value that minimizes a function also minimizes its square root).

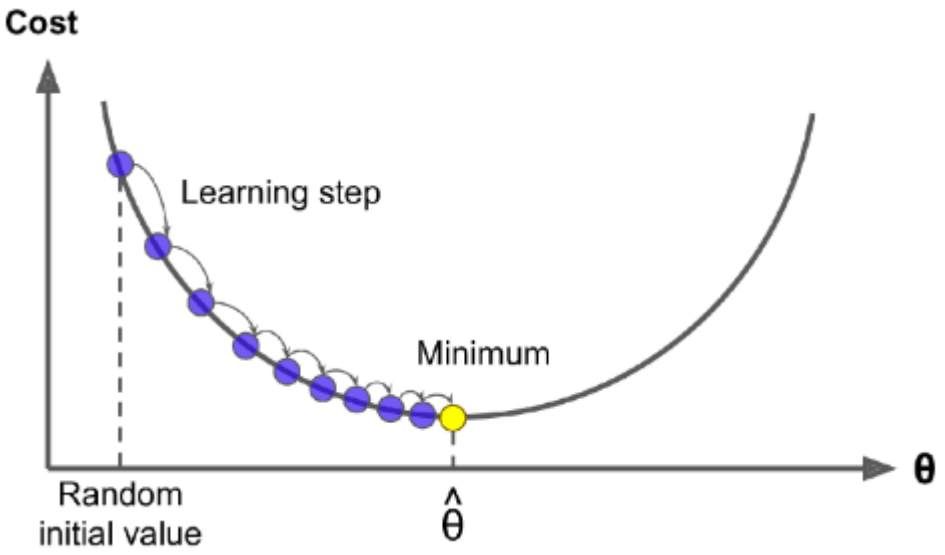
- Optimisation algorithms:

- Normal equation and Singular Value Decomposition (SVD): They are linear regarding the number of instances in the training set, which is good, but they are very slow when the number of features grows large. SVD uses the typical “LinearRegression” from Sckit learn.

Other ways to train a Linear Regression model when there are a large number of features or too many training instances to fit in memory:

- Gradient descent:

You start by filling θ with random values (this is called random initialization). Then you improve it gradually, taking one baby step at a time, each step attempting to decrease the cost function (e.g., the MSE), until the algorithm converges to a minimum. An important parameter in Gradient Descent is the size of the steps, determined by the learning rate hyperparameter. If the learning rate is too small, then the algorithm will have to go through many iterations to converge, which will take a long time. On the other hand, if the learning rate is too high, you might jump across the valley and end up on the other side, possibly even higher up than you were before.



Book

Fortunately, the MSE cost function for a Linear Regression model happens to be a convex function (there are no local minimus).

When using Gradient Descent, you should ensure that all features have a *similar scale* (e.g., using Scikit-Learn's StandardScaler class), or else it will take much longer to converge.

We have different types of gradient descent (Batch GD, stochastic, mini-batch GD…)

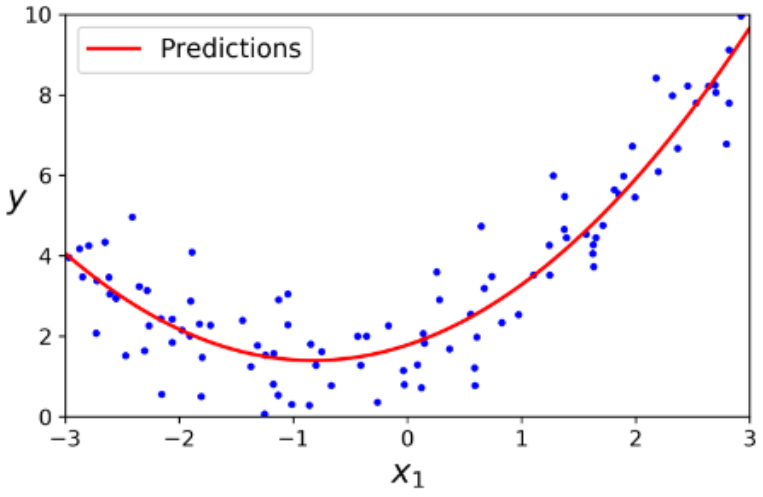
- Stochastic Gradient Descent: When the cost function is very irregular, this can actually help the algorithm jump out of local minima, so Stochastic Gradient Descent has a better chance of finding the global minimum. When using Stochastic Gradient Descent, the training instances must be independent and identically distributed.

```
from sklearn.linear_model import SGDRegressor
sgd_reg = SGDRegressor(max_iter=1000, tol=1e-3, penalty=None, eta0=0.1)
sgd_reg.fit(X, y.ravel())
```

(Final note) Llinear regressors algorithms reviewed until now: There is almost no difference after training: all these algorithms end up with very similar models and make predictions in exactly the same way.

B. POLYNOMIAL REGRESSION

What if your data is more complex than a straight line? Surprisingly, you can use a linear model to fit nonlinear data. A simple way to do this is to add powers of each feature as new features, then train a linear model on this extended set of features. This technique is called Polynomial Regression.



Note that when there are multiple features, Polynomial Regression is capable of finding relationships between features (which is something a plain Linear Regression model cannot do).

The Bias/Variance Trade-off

An important theoretical result of statistics and Machine Learning is the fact that a model's generalization error can be expressed as the sum of three very different errors:

Bias

This part of the generalization error is due to wrong assumptions, such as assuming that the data is linear when it is actually quadratic. A high-bias model is most likely to underfit the training data.

Variance

This part is due to the model's excessive sensitivity to small variations in the training data. A model with many degrees of freedom (such as a high-degree polynomial model) is likely to have high variance and thus overfit the training data.

Irreducible error

This part is due to the noisiness of the data itself. The only way to reduce this part of the error is to clean up the data (e.g., fix the data sources, such as broken sensors, or detect and remove outliers).

Increasing a model's complexity will typically increase its variance and reduce its bias.Conversely, reducing a model's complexity increases its bias and reduces its variance.This is why it is called a trade-off.

C. REGULARIZED LINEAR MODELS

A good way to **reduce overfitting** is to regularize the model (i.e., to constrain it): the fewer degrees of freedom it has, the harder it will be for it to overfit the data. A simple way to regularize a polynomial model is to reduce the number of polynomial degrees.

For a linear model, regularization is typically achieved by constraining the weights of the model.

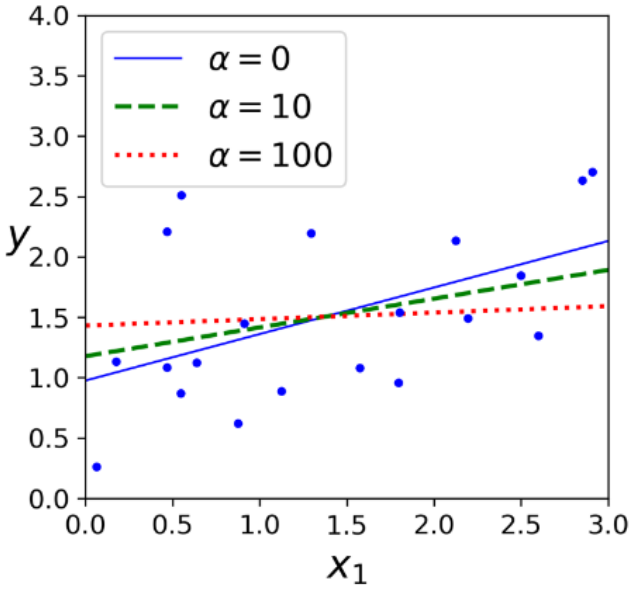
1. Ridge regression

is a regularized version of Linear Regression. Here, a regularization term is added to the cost function. Note that the regularization term should only be added to the cost function during training. Once the model is trained, you want to use the

unregularized performance measure to evaluate the model's performance, like Mean Squared Error (MSE) or R-squared.)

The hyperparameter **α** controls how much you want to regularize the model. If α = 0, then Ridge Regression is just Linear Regression. If α is very large, then all weights end up very close to zero and the result is a flat line going through the data's mean

It is important to scale the data (e.g., using a StandardScaler) before performing Ridge Regression, as it is sensitive to the scale of the input features. This is true of most regularized models !!!



Note how increasing α leads to flatter (i.e., less extreme, more reasonable) predictions, thus reducing the model's variance but increasing its bias.

In polynomial regression we can also use ridge regression.

```
>>> from sklearn.linear_model import Ridge
>>> ridge_reg = Ridge(alpha=1, solver="cholesky")
>>> ridge_reg.fit(X, y)
>>> ridge_reg.predict([[1.5]])
array([[1.55071465]])
```

Ridge Applies a penalty to the sum of the squares of the model coefficients, known as L2 regularization. Useful when you have many features that contribute to the output and when ***multicollinearity*** (high correlation between predictor variables) is present.

2. Lasso regression

Just like Ridge Regression, it adds a regularization term to the cost function, but it uses the ℓ1 norm of the weight vector instead of half the square of the ℓ2 norm (Applies a penalty to the sum of the absolute values of the model coefficients, known as L1 regularization).
Can shrink some coefficients completely to **zero**, thus performing ***feature selection***. This is useful when you suspect that some features are not important or when you want a sparse model.

Use Case: Useful in scenarios where you want to reduce the number of features in your model, or when you have reasons to believe many features are irrelevant or redundant.

3. Elastic net

Elastic Net is a middle ground between Ridge Regression and Lasso Regression.

The regularization term is a simple mix of both Ridge and Lasso's regularization terms, and you can control the **mix ratio r**. When r = 0, Elastic Net is equivalent to Ridge Regression, and when r = 1, it is equivalent to Lasso Regression.

This approach allows for feature selection like Lasso, while still retaining the regularization properties of Ridge. It's especially beneficial when there are ***multiple correlated features***.

➔ Which one to choose?

It is almost always preferable to have at least a little bit of regularization, so generally you should avoid plain Linear Regression. *Ridge* is a good default.

- **Ridge** is a good default choice when the features are expected to have approximately equal importance or when the data is particularly noisy.
- **Lasso** is preferred when you want a simpler, more interpretable model or when you have a high number of irrelevant features.
- **Elastic Net** is a good choice when you have a large number of correlated features, or when you're unsure whether Lasso or Ridge would be more appropriate.

3. CLASSIFICATION

A) Binary classification: binary classifiers distinguish between two classes. Logistic Regression or Support Vector Machine classifiers are strictly binary classifiers.

Examples: Stochastic Gradient Descent (SGD) classifier, using Scikit-Learn's SGDClassifier class. This classifier has the advantage of being capable of handling very large datasets efficiently.

B) Multiclass classifiers: multiclass classifiers (also called multinomial classifiers) can distinguish between more than two classes. SGD classifiers, Random Forest classifiers, and naive Bayes classifiers) are capable of handling multiple classes natively also.

C) Multilabel Classification: Each instance can be assigned multiple labels (categories) simultaneously. This is the case of a face recognition classifier, where it can recognize several people in the same picture.

D) Multioutput classification: It is simply a generalization of multilabel classification where each label can be multiclass (i.e., it can have more than two possible values).

CLASSIFIER PERFORMANCE MEASURES

Measuring accuracy using cross-validation: accuracy is generally NOT the preferred performance measure for classifiers, especially when you are dealing with skewed datasets (i.e., when some classes are much more frequent than others).

Confusion matrix:To compute the confusion matrix, you first need to have a set of predictions so that they can be compared to the actual targets.

Confusion Matrix

	Actually Positive (1)	Actually Negative (0)
Predicted Positive (1)	True Positives (TPs)	False Positives (FPs)
Predicted Negative (0)	False Negatives (FNs)	True Negatives (TNs)

Precision and recall:

		Predicted	
		0	**1**
Actual	**0**	TN	FP
	1	FN	TP

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

```
>>> from sklearn.metrics import precision_score, recall_score
>>> precision_score(y_train_5, y_train_pred) # == 4096 / (4096 + 1522)
0.7290850836596654
>>> recall_score(y_train_5, y_train_pred) # == 4096 / (4096 + 1325)
0.7555801512636044
```

- **Precision** is about being accurate in your positive predictions. High precision means when your model predicts something as positive, it's likely to be truly positive. In simpler terms, it answers the question: "Of all the instances the model labeled as positive, how many were actually positive?"

- **Recall** is about capturing as many actual positives as possible. High recall means your model is good at catching most of the positive cases.Recall answers the question: "Of all the actual positive instances, how many did the model correctly identify?"

There is a **trade-off** among them. Let's exemplify this:

Imagine you have a model to identify spam emails. If you set it to be very cautious (high precision), it will only mark an email as spam if it's very sure, leading to fewer spam emails in your inbox but possibly missing some spam (lower recall).

On the other hand, if you set it to be aggressive in marking spam (high recall), you'll catch almost all spam emails, but you might also incorrectly mark some good emails as spam (lower precision).

This illustrates the trade-off: increasing precision often decreases recall, and increasing recall often decreases precision.

The solution is the **F1 score**: The F1 score is the harmonic mean of precision and recall. It takes both false positives and false negatives into account, providing a single score that balances the two.

$$F1 = 2 \times \frac{(Precision \times Recall)}{(Precision + Recall)}$$

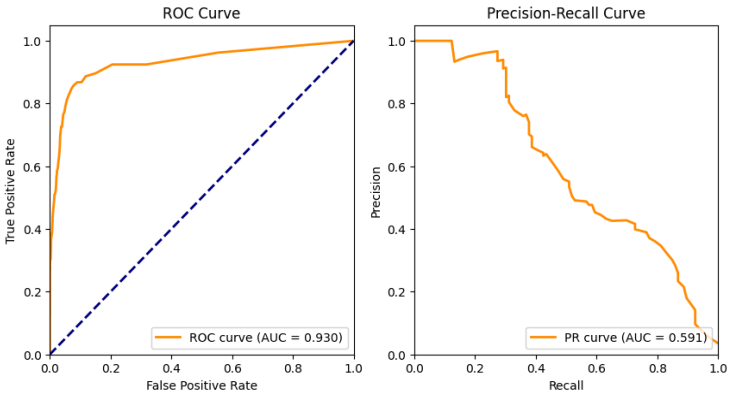
```
>>> from sklearn.metrics import f1_score
>>> f1_score(y_train_5, y_train_pred)
0.7420962043663375
```

The Precision-Recall (PR) curve is a tool used in machine learning to evaluate the performance of a classification model, especially in scenarios where one class is much rarer than the other (imbalanced datasets). It is a graphical representation that illustrates the trade-off between precision and recall for different thresholds. The PR curve plots Precision (y-axis) against Recall (x-axis). Each point on this curve represents a different threshold setting for classifying a positive instance.
The curve shows the trade-off between precision and recall for different threshold values. A high area under the PR curve represents both high recall and high precision, where high precision relates to a low false positive rate, and high recall relates to a low false negative rate.

ROC-courve:The receiver operating characteristic (ROC) curve is another common tool used with binary classifiers.The ROC curve plots the true positive rate (another name for recall, “TPR”) against the false positive rate (FPR).he ROC curve plots sensitivity (recall) versus 1 – specificity.

```
from sklearn.metrics import roc_curve

fpr, tpr, thresholds = roc_curve(y_train_5, y_scores)
```

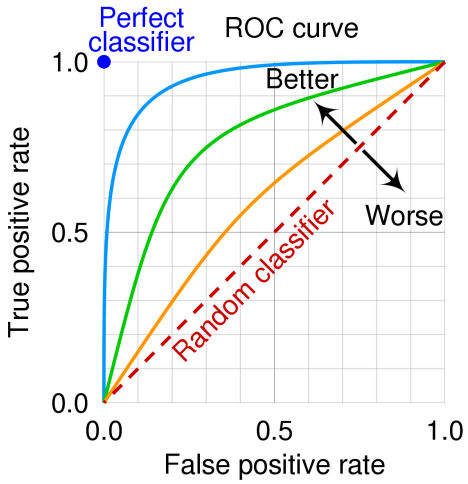


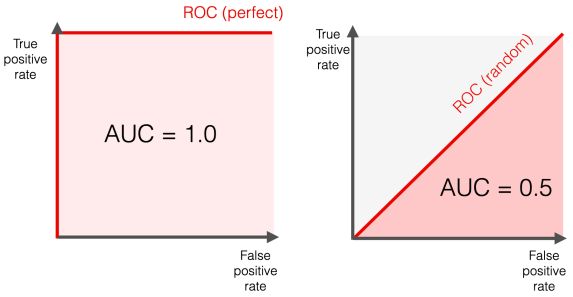
```
def plot_roc_curve(fpr, tpr, label=None):
    plt.plot(fpr, tpr, linewidth=2, label=label)
    plt.plot([0, 1], [0, 1], 'k--') # Dashed diagonal
    [...] # Add axis labels and grid

plot_roc_curve(fpr, tpr)
plt.show()
```

ROC interpretation:

- A curve that lies above the diagonal line (which represents random chance) indicates a good classification model.
- The closer the curve follows the left-hand border and then the top border of the ROC space, the more accurate the model.





One way to compare classifiers is to measure the area under the curve (**AUC**). A perfect classifier will have a ROC AUC equal to 1, whereas a purely random classifier will have a ROC AUC equal to 0.5.

As a rule of thumb, you should prefer the PR curve whenever the positive class is rare or when you care more about the false positives than the false negatives. Otherwise, use the ROC curve.

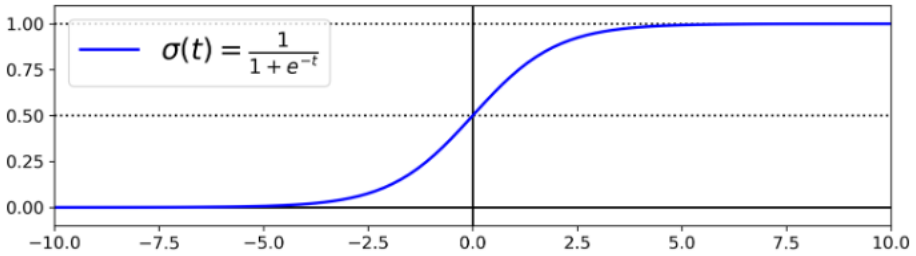
TYPES OF CLASSIFICATION ALGORITHMS

A) LOGISTIC REGRESSION

Traditional Logistic Regression is used for binary classification problems, where there are only two possible outcomes. It models the probability that a given input belongs to a certain class (e.g., the probability of an email being spam). The output is a value between 0 and 1, which can be interpreted as a probability.

So how does Logistic Regression work? Just like a Linear Regression model, a Logistic Regression model computes a weighted sum of the input features (plus a bias term), but instead of outputting the result directly like the Linear Regression model does, it outputs the logistic of this resul.

In this case, we want to cut (“acotar”) the hypothesis function, for example between 0 and 1 , and to achieve that the sigmoid function is used.



As we can see from above, the function returns a continuous value cut between 0 and 1. We then have to decide a threshold to get a discrete value (a aprtir de cuando sera 0 y a aprtir de cuando sera considerado como 1).

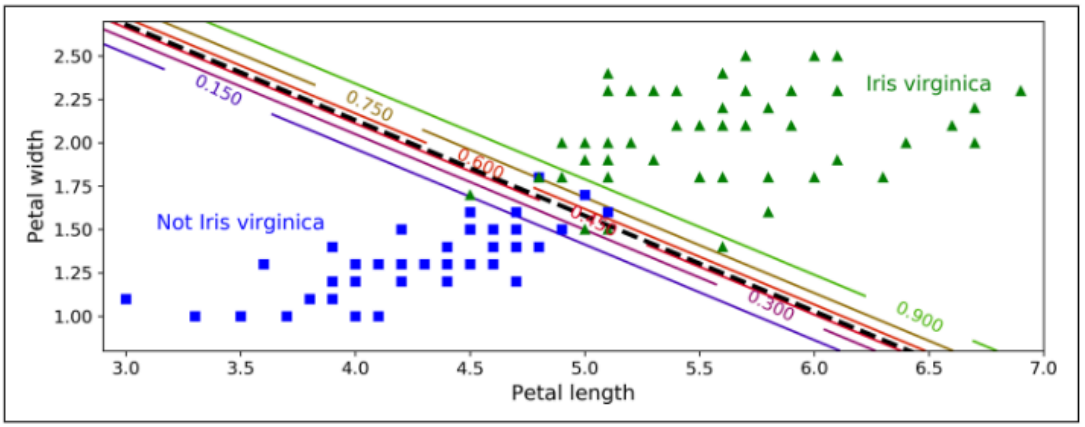


Figure 4-24. Linear decision boundary

Just like the other linear models, Logistic Regression models can be regularized using ℓ1 or ℓ2 penalties. Scikit-Learn actually adds an ℓ2 penalty by default.

In this case, we can’t use the same cost function as in the linear regression because local minimus are generated.

B) SOFTMAX REGRESSION

The Logistic Regression model can be generalized to support multiple classes directly, without having to train and combine multiple binary classifiers . This is called Softmax Regression, or Multinomial Logistic Regression. Instead of yielding a single probability score, it outputs a probability distribution across several classes. This is achieved through the softmax function.

The Softmax Regression classifier predicts only one class at a time (i.e., it is multiclass, not multioutput), so it should be used only with mutually exclusive classes, such as different types of plants. You cannot use it to recognize multiple people in one picture.

The objective is to have a model that estimates a high probability for the target class (and consequently a low probability for the other classes). Minimizing the cost function, called the **cross entropy**, should lead to this objective because it penalizes the model when it estimates a low probability for a target class. Cross entropy is frequently used to measure how well a set of estimated class probabilities matches the target classes.

Cross entropy quantifies the difference between two probability distributions - the true distribution (true labels) and the estimated distribution (model predictions).
A lower cross entropy value indicates a model that makes predictions close to the actual class labels, which is desired. A perfect model would have a cross entropy of 0.
cross-entropy can be used to optimize most classification problems, especially those where models provide probabilistic outputs. cross-entropy might not be ideal in certain situations, such as when dealing with highly imbalanced datasets

Scikit-Learn's LogisticRegression uses one-versus-the-rest by default when you train it on more than two classes, but you can set the multi_class hyperparameter to "multinomial" to switch it to Softmax Regression. You must also specify a solver that supports Softmax Regression, such as the "lbfgs" solver (see Scikit-Learn's documentation for more details). It also applies ℓ2 regularization by default, which you can control using the hyperparameter C.

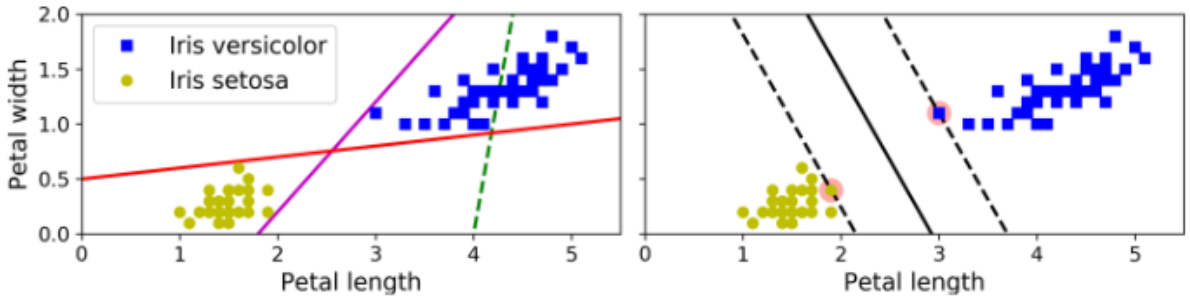
4. SUPPOR VECTOR MACHINES (SVM)

A Support Vector Machine (SVM) is a powerful and versatile Machine Learning model, capable of performing **linear** or **nonlinear** classification, regression, and even outlier detection. It is one of the most popular models in Machine Learning. SVMs are particularly well suited for classification of complex small- or medium-sized datasets.

Unlike Logistic Regression classifiers, SVM classifiers do NOT output probabilities for each class.

4.1. SVM CLASSIFICATION

4.1.1 Linear SVM classification



You can think of an SVM classifier as fitting the widest possible street (represented by the parallel dashed lines) between the classes. This is called *large margin classification.*

SVMs are sensitive to the feature scales, must **scale** before using them!

a) Hard margin classification:

If we strictly impose that all instances must be off the street and on the right side, this is called hard margin classification. There are two main issues with hard margin classification. First, it only works if the data is linearly separable. Second, it is sensitive to outliers.

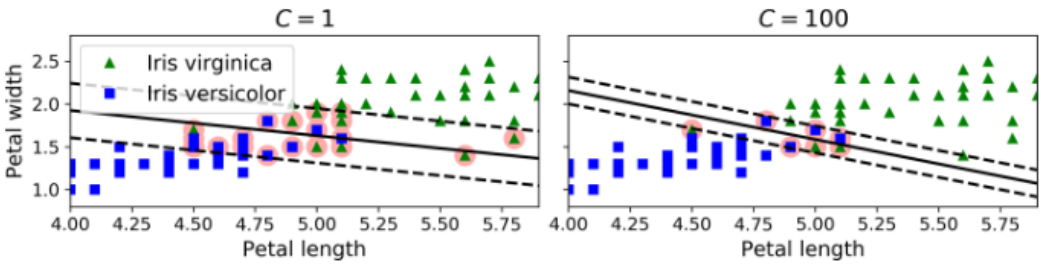
Ideal Scenario: It is used when the data is linearly separable, meaning there is a clear gap between the two classes with no overlaps. The goal is to find a decision boundary (a hyperplane in feature space) that maximally separates the two classes.
The rigid nature of hard margin classification can lead to *overfitting*, especially if the data has outliers or is not perfectly linearly separable.

b) Soft margin classification:

To avoid the limitations of the hard margin classification we can use a more flexible model. It's used when the data is almost linearly separable but with some overlap or noise.
The goal is still to find a decision boundary that separates the classes, but it allows some misclassifications or margin violations for greater generalization to unseen data.

The degree of margin violations is controlled by a hyperparameter (often denoted as 'C'). A higher value of 'C' leads to fewer margin violations (approaching a hard margin), whereas a lower value allows more violations (softer margin). Therefore, If your SVM model is overfitting, you can try regularizing it by reducing C.

Margin violations are bad. It's usually better to have few of them. However, in this case the model on the left has a lot of margin violations but will probably generalize better.




```
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC

iris = datasets.load_iris()
X = iris["data"][:, (2, 3)]  # petal length, petal width
y = (iris["target"] == 2).astype(np.float64)  # Iris virginica

svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("linear_svc", LinearSVC(C=1, loss="hinge")),
    ])

svm_clf.fit(X, y)
```

Instead of using the LinearSVC class, we could use the SVC class with a linear kernel. When creating the SVC model, we would write SVC(kernel="linear", C=1).

4.1.2. Non-linear SVM classification

In many cases, 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).

Polynomial Kernel

Adding polynomial features is simple to implement and can work great with all sorts of Machine Learning algorithms (not just SVMs). That said, at a low polynomial degree, this method cannot deal with very complex datasets, and with a high polynomial degree it creates a huge number of features, making the model too slow.

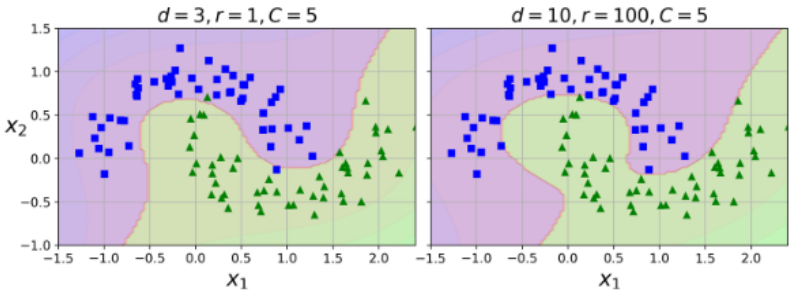
SOLUTION:

When using SVMs you can apply an almost miraculous mathematical technique called the *kernel trick* (explained in a moment). The kernel trick makes it possible to get the same result as if you had added many polynomial features, even with very high-degree polynomials, without actually having to add them. So there is no combinatorial explosion of the number of features because you don't actually add any features. This trick is implemented by the SVC class. Let's test it on the moons dataset.

This code trains an SVM classifier using a third-degree polynomial kernel:

```
from sklearn.svm import SVC
poly_kernel_svm_clf = Pipeline([
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="poly", degree=3, coef0=1, C=5))
    ])
poly_kernel_svm_clf.fit(X, y)
```

Obviously, if your model is overfitting, you might want to reduce the polynomial degree. Conversely, if it is underfitting, you can try increasing it. (you can search for the best hyperparameter d using grid search).



Kernel Function

A kernel function, in the context of machine learning, is a method used to compute the similarity or a dot product of two vectors (data points) in a high-dimensional space without explicitly mapping the points into that space. This concept is known as the "kernel trick." Kernel functions enable operations in a higher-dimensional space without incurring the computational cost of actually working in that space.

4.2. SVM REGRESSION

The SVM algorithm is versatile: not only does it support linear and nonlinear classification, but it also supports linear and nonlinear regression.

SVM Regression tries to fit as many instances as possible on the street while limiting margin violations (i.e., instances off the street). The width of the street is controlled by a hyperparameter, ϵ.

Regression models trained on some random linear data, one with a large margin ($\epsilon = 1.5$) and the other with a small margin ($\epsilon = 0.5$).

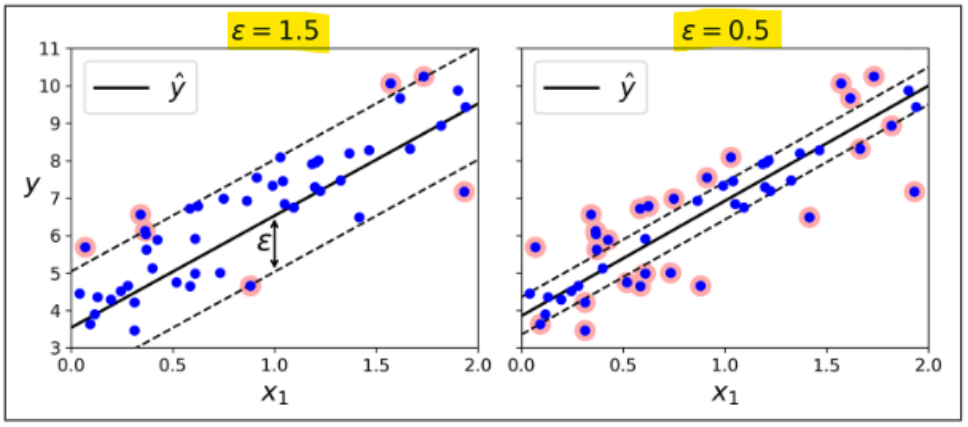


Figure 5-10. SVM Regression

SVM Regression : *from sklearn.svm import* ***LinearSVR***

- To tackle nonlinear regression tasks, you can use a kernelized SVM model.

5. DECISION TREES

- Can perform both classification and regression tasks.
- Decision Trees are also the fundamental components of Random Forests , which are among the most powerful Machine Learning algorithms available today.
- One of the many qualities of Decision Trees is that they require very little data preparation. In fact, they **don't** require feature scaling or centering at all.
- Scikit-Learn uses the CART algorithm, which produces only binary trees.
- A Decision Tree can also estimate the probability that an instance belongs to a particular class k.
- Often produces a solution that's reasonably good but not guaranteed to be optimal.
- Making predictions requires traversing the Decision Tree from the root to a leaf → It can be *computationally expensive*!

- CART ALGORITHM

Scikit-Learn uses the **Classification and Regression Tree (CART)** algorithm to train Decision Trees (also called "growing" trees). The algorithm works by first splitting the training set into two subsets using a single feature k and a threshold tk (e.g., "petal length ≤ 2.45 cm"). How does it choose k and tk? It searches for the pair (k, tk) that produces the purest subsets (weighted by their size).

It stops recursing once it reaches the maximum depth (defined by the *max_depth* hyperparameter), or if it cannot find a split that will reduce impurity. A few other hyperparameters to control additional stopping conditions are:

- max_depth : maximum depth of the Decision Tree. Reducing max_depth will regularize the model and thus reduce the risk of overfitting.
- min_samples_split: the minimum number of samples a node must have before it can be split min_samples_leaf (the minimum number of samples a leaf node must have)
- min_weight_fraction_leaf (same as min_samples_leaf but expressed as a fraction of the total number of weighted instances).
- max_leaf_nodes (the maximum number of leaf nodes).

- max_features(the maximum number of features that are evaluated for splitting at each node).

Increasing min_* hyperparameters or reducing max_* hyperparameters will regularize
the model.
If left unconstrained, the tree structure will adapt itself to the training data, fitting it very closely—indeed, most likely **overfitting**. To avoid overfitting the training data, you need to restrict the Decision Tree's freedom during training (regularization!). The regularization hyperparameters depend on the algorithm used, but generally you can at least restrict the maximum depth of the Decision Tree (max_depth).

- GINI IMPURITY VS ENTROPY

In machine learning, particularly in decision tree algorithms and their ensemble versions like Random Forests, **Gini Impurity** and **Entropy** are two commonly used criteria for determining how to split the data at each node. Both are measures of the disorder or impurity in a set of data.

So, should you use Gini impurity or entropy? The truth is, most of the time it does not make a big difference: they lead to similar trees. Gini impurity is slightly faster to compute, so it is a good default. However, when they differ, Gini impurity tends to isolate the most frequent class in its own branch of the tree, while entropy tends to produce slightly more balanced trees.

Gini Impurity:

- ➢ **Definition:** Gini Impurity measures the frequency at which any element of the dataset will be mislabeled when it is randomly labeled according to the distribution of labels in the dataset.
- ➢ **Interpretation:** A Gini score of 0 denotes that all elements belong to a single class, implying perfect purity. The higher the Gini score, the higher the impurity of the nodes.
- ➢ **Usage:** Gini Impurity is typically faster to compute and is the default in many machine learning algorithms (like in scikit-learn's decision trees). It tends to isolate the most frequent class first and can be slightly biased if the classes are imbalanced.

Entropy:

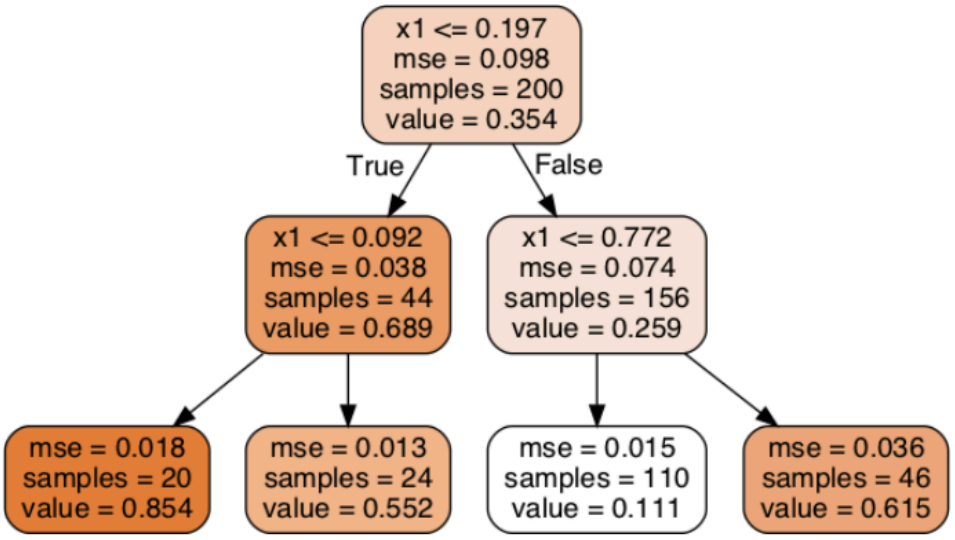
- ➢ **Definition:** Entropy, inspired by the concept from information theory, is a measure of the randomness or unpredictability in the data set.

- ➢ **Interpretation:** An Entropy of 0 indicates complete order (all elements belong to a single class), while higher values indicate more disorder.
- ➢ **Usage:** Entropy is a bit slower to compute due to the log function. It is more sensitive to class imbalance than Gini and can lead to slightly more balanced decision trees.

- REGRESSION

```
from sklearn.tree import DecisionTreeRegressor

tree_reg = DecisionTreeRegressor(max_depth=2)
tree_reg.fit(X, y)
```



LIMITATION: the main issue with Decision Trees is that they are very sensitive to small variations in the training data. Random Forests can limit this instability by averaging predictions over many trees, as we will see in the next chapter.

6. ENSEMBLE LEARNING AND RANDOM FORESTS

If you aggregate the predictions of a group of predictors (such as classifiers or regressors), 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.

As an example of an Ensemble method, you can train a group of Decision Tree classifiers, each on a different random subset of the training set. To make predictions, you obtain the predictions of all the individual trees, then predict the class that gets the most votes (see the last exercise in Chapter 6). Such an ensemble of Decision Trees is called a Random Forest, and despite its simplicity, this is one of the most powerful Machine Learning algorithms available today.

You will often use Ensemble methods near the end of a project, once you have already built a few good predictors, to combine them into an even better predictor.
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. This increases the chance that they will make very different types of errors, improving the ensemble's accuracy.

VOTING CLASSIFIER

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

log_clf = LogisticRegression()
rnd_clf = RandomForestClassifier()
svm_clf = SVC()

voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='hard')
voting_clf.fit(X_train, y_train)
```

Let's look at each classifier's accuracy on the test set:

```
>>> from sklearn.metrics import accuracy_score
>>> for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
...     clf.fit(X_train, y_train)
...     y_pred = clf.predict(X_test)
...     print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
...
LogisticRegression 0.864
RandomForestClassifier 0.896
SVC 0.888
VotingClassifier 0.904
```

There you have it! The voting classifier slightly outperforms all the individual classifiers.

You can choose between voting='hard' or voting='soft' (soft sometimes gets better results).

BAGGING AND PASTING

Another approach is to use the same training algorithm for every predictor and train them on different random subsets of the training set. When sampling is performed **with replacement**, this method is called bagging. When sampling is performed without replacement, it is called pasting.

Predictors can all be trained in parallel, via different CPU cores or even different servers. Similarly, predictions can be made in parallel. This is one of the reasons bagging and pasting are such popular methods: they scale very well.

Scikit-Learn offers a simple API for both bagging and pasting with the `BaggingClassifier` class (or `BaggingRegressor` for regression). The following code trains an ensemble of 500 Decision Tree classifiers:[5] each is trained on 100 training instances randomly sampled from the training set with replacement (this is an example of bagging, but if you want to use pasting instead, just set `bootstrap=False`). The `n_jobs` parameter tells Scikit-Learn the number of CPU cores to use for training and predictions (`-1` tells Scikit-Learn to use all available cores):

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

bag_clf = BaggingClassifier(
    DecisionTreeClassifier(), n_estimators=500,
    max_samples=100, bootstrap=True, n_jobs=-1)
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
```

The `BaggingClassifier` automatically performs soft voting instead of hard voting if the base classifier can estimate class probabilities (i.e., if it has a `predict_proba()` method), which is the case with Decision Tree classifiers.

➔ Overall, **bagging** often results in better models, which explains why it is generally preferred. However, 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.

RANDOM FOREST

Instead of building a BaggingClassifier and passing it a DecisionTreeClassifier, you can instead use the RandomForestClassifier class, which is more convenient and optimized for Decision Trees.
Yet 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).

You can access the result using the feature_importances_ variable.

Random Forests are very handy to get a quick understanding of what features actually matter, in particular if you need to perform feature selection

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 (as discussed earlier). It is possible to make trees even more random by also using random thresholds for each feature rather than
searching for the best possible thresholds

A forest of such extremely random trees is called an Extremely Randomized Trees ensemble. Once again, this technique trades more bias for a lower variance. It also makes Extra-Trees much **faster** to train than regular Random Forests, because finding the best possible threshold for each feature at every node is one of the most time-consuming tasks of growing a tree. You can create an Extra-Trees classifier using Scikit-Learn's ExtraTreesClassifier class.

It is hard to tell in advance whether a RandomForestClassifier will perform better or worse than an ExtraTreesClassifier. Generally, the only way to know is to try both and compare them using cross-validation (tuning the hyperparameters using grid search).

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.

There are many boosting methods available, but by far the most popular are **AdaBoost** (short for Adaptive Boosting. SVMs are generally not good base predictors for AdaBoost; they are slow and tend to be unstable with it.) and **Gradient Boosting**.

a) ADA BOOST

One way for a new predictor to correct its predecessor is to pay 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. This is the technique used by AdaBoost.

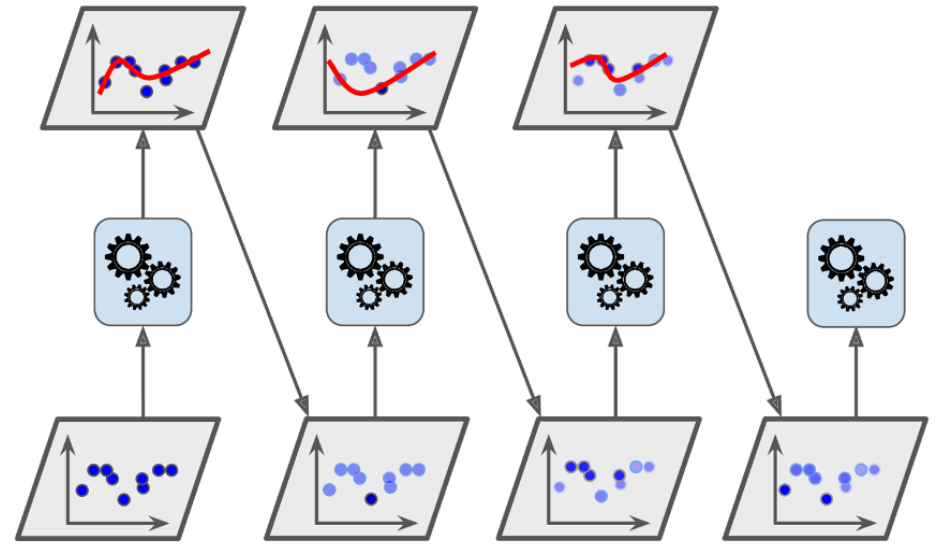


Figure 7-7. AdaBoost sequential training with instance weight updates

Once all predictors are trained, the ensemble makes predictions very much like bagging or pasting, except that predictors have different weights depending on their overall accuracy on the weighted training set.

There is one important drawback to this sequential learning technique: it cannot be parallelized (or only partially), 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.

```
from sklearn.ensemble import AdaBoostClassifier

ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1), n_estimators=200,
    algorithm="SAMME.R", learning_rate=0.5)
ada_clf.fit(X_train, y_train)
```

If your AdaBoost ensemble is overfitting the training set, you can try reducing the number of estimators or more strongly regularizing the base estimator.

b) GRADIENT BOOSTING

Just like AdaBoost, Gradient Boosting works by sequentially adding predictors to an ensemble, each one correcting its predecessor. However, instead of tweaking the instance weights at every iteration like AdaBoost does, this method tries to fit the new predictor to the residual errors made by the previous predictor.

It is worth noting that an optimized implementation of Gradient Boosting is available in the popular Python library XGBoost, which stands for Extreme Gradient Boosting.

7. DIMENSIONALITY REDUCTION

Many Machine Learning problems involve thousands or even millions of features for each training instance. Fortunately, in real-world problems, it is often possible to reduce the number of features considerably, turning an intractable problem into a tractable one.
Reducing dimensionality does cause some information loss (just like compressing an image to JPEG can degrade its quality), so even though it will speed up training, it may make your system perform slightly worse. In some cases, reducing the dimensionality of the training data may filter out some noise and unnecessary details and thus result in higher performance, but in general it won't; it will just speed up training.

Apart from speeding up training, dimensionality reduction is also extremely useful for data visualization.

It is an unsupervised learning task.

Main approaches for dimensionality reduction:

a) PROJECTION

Projection techniques work by projecting high-dimensional data into a lower-dimensional space. The idea is to "flatten" the data onto a subspace that captures the essence of the data. One common assumption is that although the data exists in a high-dimensional space, it might inherently occupy a lower-dimensional subspace.

-Principal Component Analysis (PCA): The most famous projection method, PCA identifies the axis that accounts for the largest amount of variance in the training set and projects the data onto this axis, continuing this process with axes orthogonal to the first to capture as much of the remaining variance as possible.

Before you can project the training set onto a lower-dimensional hyperplane, you first need to choose the right hyperplane. It seems reasonable to select the axis that preserves the maximum amount of variance, as it will most likely lose less information than the other projections. Another way to justify this choice is that it is the axis that minimizes the mean squared distance between the original dataset and its projection onto that axis. This is the rather simple idea behind PCA.

PCA identifies the axis that accounts for the largest amount of variance in the training set. It also finds a second axis, orthogonal to the first one, or third, or fourth, etc, that accounts for the largest amount of remaining variance.
PCA assumes that the dataset is centered around the origin (automatically done with Scikit-Learn, but needs to be centered otherwise).

```
from sklearn.decomposition import PCA

pca = PCA(n_components = 2)
X2D = pca.fit_transform(X)
```

pca.components_.T[:, 0]) *# unit vector that defines the first principal component*

pca.explained_variance_ratio_ *# The ratio indicates the proportion of the dataset's variance that lies along each principal component*

>>> pca.explained_variance_ratio_
array([0.84248607, 0.14631839])

This output tells you that 84.2% of the dataset's variance lies along the first PC, and 14.6% lies along the second PC. This leaves less than 1.2% for the third PC, so it is reasonable to assume that the third PC probably carries little information.

Choosing the right number of dimensions:

Instead of arbitrarily choosing the number of dimensions to reduce down to, it is simpler to choose the number of dimensions that add up to a sufficiently large portion of the variance (e.g., 95%). Unless, of course, you are reducing dimensionality for data visualization—in that case you will want to reduce the dimensionality down to 2 or 3.

The following code performs PCA without reducing dimensionality, then computes the minimum number of dimensions required to preserve 95% of the training set's variance:

```
pca = PCA()
pca.fit(X_train)
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum >= 0.95) + 1
```

Alternatively, you can set n_components to be a float between 0.0 and 1.0, indicating the ratio of variance you wish to preserve:

```
pca = PCA(n_components=0.95)
X_reduced = pca.fit_transform(X_train)
```

Kernel PCA (kPCA): Makes it possible to perform complex nonlinear projections for dimensionality reduction.

The following code uses Scikit-Learn's `KernelPCA` class to perform kPCA with an RBF kernel (see Chapter 5 for more details about the RBF kernel and other kernels):

```
from sklearn.decomposition import KernelPCA

rbf_pca = KernelPCA(n_components = 2, kernel="rbf", gamma=0.04)
X_reduced = rbf_pca.fit_transform(X)
```

As kPCA is an unsupervised learning algorithm, so there is no obvious performance measure to help you select the best kernel and hyperparameter values. You can use grid search to select the kernel and hyperparameters that lead to the best performance on that task.

-Linear Discriminant Analysis (LDA): Used for dimensionality reduction in a supervised context, LDA projects data in a way that maximizes class separability.

b) MAINFOLD LEARNING

Manifold Learning, or Nonlinear Dimensionality Reduction, is based on the idea that the data lies on a manifold within the higher-dimensional space. A manifold is a shape that might be very twisted and curved in high-dimensional space. The goal is to understand the structure of this manifold to use a lower-dimensional space that captures the intrinsic geometry of the data.

Locally Linear Embedding (LLE) is another powerful nonlinear dimensionality reduction (NLDR) technique. It is a Manifold Learning technique that does not rely on projections. LLE works by first measuring how each training instance linearly relates to its closest neighbors (c.n.), and then looking for a low-dimensional representation of the training set where these local relationships are best preserved.

```
from sklearn.manifold import LocallyLinearEmbedding

lle = LocallyLinearEmbedding(n_components=2, n_neighbors=10)
X_reduced = lle.fit_transform(X)
```

8. UNSUPERVISED LEARNING TECHNIQUES

Although most of the applications of Machine Learning today are based on supervised learning, the vast majority of the available data is unlabeled: we have the input features **X**, but we do not have the labels **y**.

Clustering

- Introduction: The goal is to group similar instances together into clusters. Clustering is a great tool for data analysis, customer segmentation, recommender systems, search engines, image segmentation, semi-supervised learning, dimensionality reduction, and more.

Just like in classification, each instance gets assigned to a group. However, unlike classification, clustering is an unsupervised task.
Some algorithms look for instances centered around a particular point, called a centroid.
In this section, we will look at two popular clustering algorithms, K-Means and DBSCAN, and explore some of their applications, such as nonlinear dimensionality reduction, semi-supervised learning, and anomaly detection.

- K-MEANS

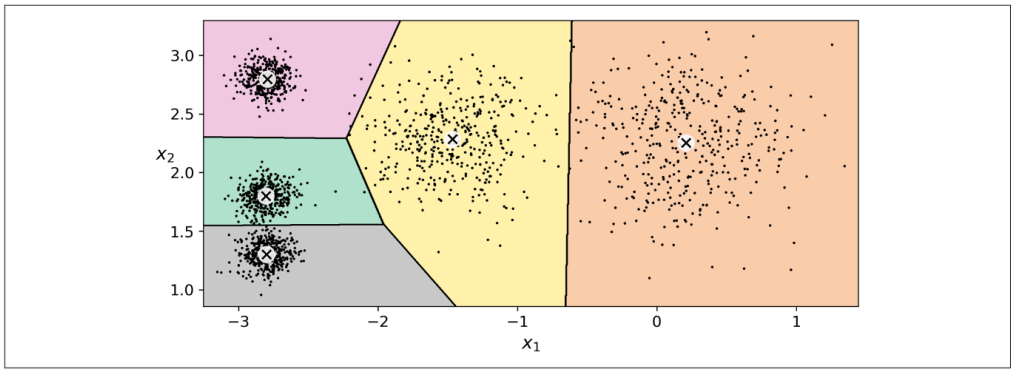
You have to specify the number of clusters k that the algorithm must find.

```
from sklearn.cluster import KMeans
k = 5
kmeans = KMeans(n_clusters=k)
y_pred = kmeans.fit_predict(X)
```

We can also take a look at the five centroids that the algorithm found:

```
>>> kmeans.cluster_centers_
array([[-2.80389616,  1.80117999],
       [ 0.20876306,  2.25551336],
       [-2.79290307,  2.79641063],
       [-1.46679593,  2.28585348],
       [-2.80037642,  1.30082566]])
```

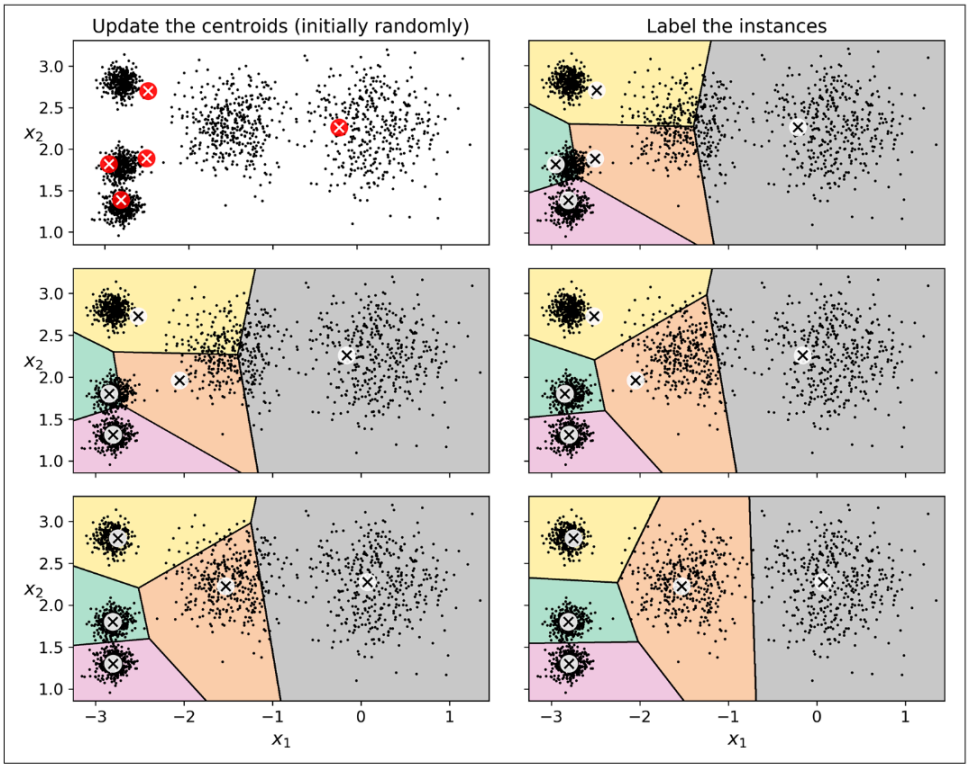
If you plot the cluster's decision boundaries, you get a Voronoi tessellation (see Figure 9-3, where each centroid is represented with an X).



In the image above, the vast majority of the instances were clearly assigned to the appropriate cluster, but a few instances were probably mislabeled (especially near the boundary between the top-left cluster and the central cluster). Indeed, the K-Means algorithm does not behave very well when the blobs have very different diameters because all it cares about when assigning an instance to a cluster is the distance to the centroid.

Instead of assigning each instance to a single cluster, which is called **hard clustering**, it can be useful to give each instance a score per cluster, which is called **soft clustering**.

How does k-means work? Just start by placing the centroids *randomly* (e.g., by picking k instances at random and using their locations as centroids). Then label the instances, update the centroids, label the instances, update the centroids, and so on until the centroids stop moving. The algorithm is guaranteed to converge in a finite number of steps. Although the algorithm is guaranteed to converge, it may not converge to the right solution (i.e., it may converge to a local optimum): whether it does or not depends on the centroid initialization.



K-Means is generally one of the fastest clustering algorithms.

- ☐ It is important to scale the input features before you run K-Means, or the clusters may be very stretched and K-Means will perform poorly. (not all cases does scaling help, be aware of the nature of the data (e.g mall customer segmentation example)

Let's look at some ways to make the algorithm converge at the optimal solution:

a) Centroid initialization methods:

If you don't know approximately where the clusters are, you can run the algorithm multiple times with different random initializations and keep the best solution. The number of random initializations is controlled by the *n_init* hyperparameter: by default, it is equal to 10, which means that the whole algorithm described earlier runs 10 times when you call fit(), and ScikitLearn keeps the best solution. But how exactly does it know which solution is the
best? It uses a performance metric! That metric is called the *model's inertia*, which is the mean squared distance between each instance and its closest centroid. The KMeans class runs the algorithm n_init times and keeps the model with the lowest inertia:

>>> kmeans.inertia_

b) Accelerated K-Means and mini-batch K-Means:

It considerably accelerates the algorithm by avoiding many unnecessary distance calculations. This is in sklearn by default.

Also, another improvement Instead of using the full dataset at each iteration, the algorithm is capable of using mini-batches, moving the centroids just slightly at each iteration. This speeds up the algorithm typically by a factor of three or four and makes it possible to cluster huge datasets that do not fit in memory. Scikit-Learn implements this algorithm in the *MiniBatchKMeans* class:

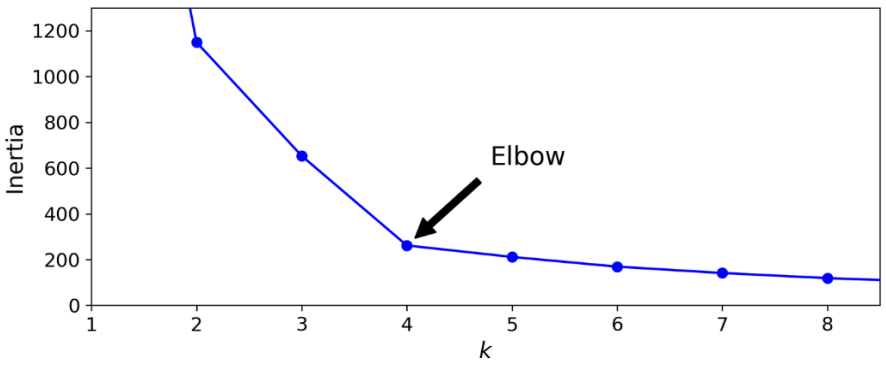
```
from sklearn.cluster import MiniBatchKMeans

minibatch_kmeans = MiniBatchKMeans(n_clusters=5)
minibatch_kmeans.fit(X)
```

Although the Mini-batch K-Means algorithm is much faster than the regular K-Means algorithm, its inertia is generally slightly worse, especially as the number of clusters increases.

c) Finding the optimal number of clusters

You might be thinking that we could just pick the model with the lowest inertia, right? Unfortunately, it is not that simple. The inertia is not a good performance metric when trying to choose **k** (number of clusters) because it keeps getting lower as we increase k. Indeed, the more clusters there are, the closer each instance will be to its closest centroid, and therefore the lower the inertia will be.



A more precise approach (but also more computationally expensive) is to use the silhouette score, which is the mean silhouette coefficient over all the instances.

The silhouette coefficient can vary between –1 and +1. A coefficient close to +1 means that the instance is well inside its own cluster and far from other clusters, while a coefficient close to 0 means that it is close to a cluster boundary, and finally a coefficient close to –1 means that the instance may have been assigned to the wrong cluster.

To compute the silhouette score, you can use Scikit-Learn's `silhouette_score()` function, giving it all the instances in the dataset and the labels they were assigned:

```
>>> from sklearn.metrics import silhouette_score
>>> silhouette_score(X, kmeans.labels_)
0.655517642572828
```

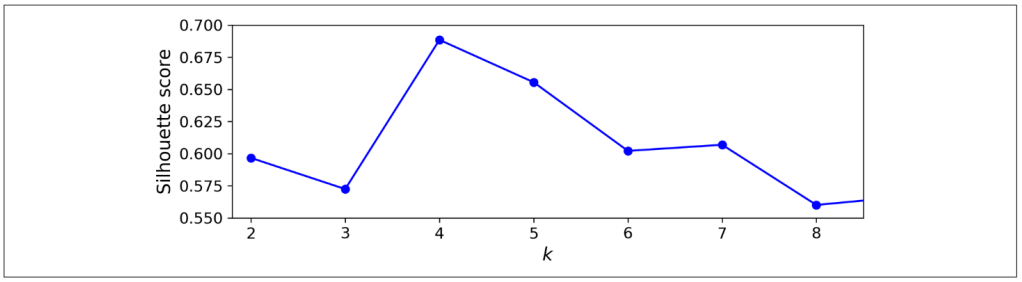


Figure 9-9. Selecting the number of clusters k using the silhouette score

As you can see, this visualization is much richer than the previous one: although it confirms that $k = 4$ is a very good choice, it also underlines the fact that $k = 5$ is quite good as well, and much better than $k = 6$ or 7. This was not visible when comparing inertias.

➔ **k-means limits**: Despite its many merits, most notably being fast and scalable, K-Means is not perfect. As we saw, it is necessary to run the algorithm several times to avoid suboptimal solutions, plus you need to specify the number of clusters, which can be quite a hassle. Moreover, K-Means does not behave very well when the clusters have varying sizes, different densities, or nonspherical shapes.

➔ **K-means applications:**

1.Image segmentation: All pixels that are part of the same object type get assigned to the same segment.

2.Preprocessing: Clustering can be an efficient approach to dimensionality reduction, in particular as a preprocessing step before a supervised learning algorithm.

```
from sklearn.pipeline import Pipeline

pipeline = Pipeline([
    ("kmeans", KMeans(n_clusters=50)),
    ("log_reg", LogisticRegression()),
])
pipeline.fit(X_train, y_train)
```

But we chose the number of clusters k arbitrarily; we can surely do better. Since K-Means is just a preprocessing step in a classification pipeline, finding a good value for k is much simpler than earlier. There's no need to perform silhouette analysis or minimize the inertia; the best value of k is simply the one that results in the best classification performance during cross-validation. We can use GridSearchCV to find the optimal number of clusters:

```
from sklearn.model_selection import GridSearchCV

param_grid = dict(kmeans__n_clusters=range(2, 100))
grid_clf = GridSearchCV(pipeline, param_grid, cv=3, verbose=2)
grid_clf.fit(X_train, y_train)
```

Let's look at the best value for *k* and the performance of the resulting pipeline:

```
>>> grid_clf.best_params_
{'kmeans__n_clusters': 99}
>>> grid_clf.score(X_test, y_test)
0.9822222222222222
```

3. Semi-supervised learning: Here we have plenty of unlabeled instances and very few labeled instances.

- DBSCAN

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is a clustering algorithm that groups together closely packed points and marks points that are in low-density regions as outliers. Here's a simple explanation of how it works:

- For each instance, the algorithm counts how many instances are located within a small distance ε (*epsilon*) from it. This region is called the instance's ε-neighborhood.
- If an instance has at least min_samples instances in its ε-neighborhood (including itself), then it is considered a *core instance*. In other words, core instances are those that are located in dense regions.

- All instances in the neighborhood of a core instance belong to the same cluster.
- This neighborhood may include other core instances; therefore, a long sequence of neighboring core instances forms a single cluster.
- Any instance that is not a core instance and does not have one in its neighborhood is considered an anomaly.

```
from sklearn.cluster import DBSCAN
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=1000, noise=0.05
dbscan = DBSCAN(eps=0.05, min_samples=5)
dbscan.fit(X)
```

The key ideas here are:

- ➢ Points in dense regions (based on the ε-neighborhood and minPts) form clusters.
- ➢ Points in sparse regions are marked as noise or outliers.DBSCAN can find clusters of any shape, unlike algorithms like K-means, which assume clusters are spherical.
- ➢ You don't need to specify the number of clusters in advance.

Basically DBSCAN groups closely packed data points together and marks points in low-density regions as outliers. After running DBSCAN on a dataset:

- **dbscan.components_** gives the positions (features) of the core samples, which are the central points in the dense regions.
- **dbscan.labels_** provides the cluster labels for each point in the dataset, where points considered as 'noise' (outliers) are labeled as -1.
- **dbscan.core_sample_indices_** is an array of indices of core samples, allowing us to correlate core samples with their labels.

```
>>> dbscan.labels_
array([ 0, 2, -1, -1, 1, 0, 0, 0, ..., 3, 2, 3, 3, 4, 2, 6, 3])
```

Notice that some instances have a cluster index equal to –1, which means that they are considered as anomalies by the algorithm.

Somewhat surprisingly, the DBSCAN class does NOT have a predict() method, although it has a fit_predict() method. In other words, it cannot predict which cluster a new instance belongs to. This implementation decision was made because different classification algorithms can be better for different tasks, so the authors decided to let the user choose which one to use. For example:

```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=50)
knn.fit(dbscan.components_, dbscan.labels_[dbscan.core_sample_indices_])
```

By training a KNN classifier on the results of DBSCAN, you create a model that can predict the cluster of new data points based on the clusters identified by DBSCAN. This method is particularly useful in scenarios where you have unlabeled data that you want to cluster and then need a way to quickly assign new data points to these clusters without rerunning the entire clustering algorithm. This would be a semi-supervised learning.

In short, DBSCAN is a very simple yet powerful algorithm capable of identifying any number of clusters of any shape. It is robust to outliers, and it has just two hyperparameters (eps and min_samples). If the density varies significantly across the clusters, however, it can be impossible for it to capture all the clusters properly.

- GAUSSIAN-MIXTURES

A Gaussian mixture model (GMM) is a probabilistic model that assumes that the instances were generated from a mixture of several Gaussian distributions whose parameters are unknown. All the instances generated from a single Gaussian distribution form a cluster that typically looks like an ellipsoid. Each cluster can have a different ellipsoidal shape, size, density, and orientation

```
from sklearn.mixture import GaussianMixture

gm = GaussianMixture(n_components=3, n_init=10)
gm.fit(X)
```

Let's look at the parameters that the algorithm estimated:

```
>>> gm.weights_
array([0.20965228, 0.4000662 , 0.39028152])
>>> gm.means_
array([[ 3.39909717,  1.05933727],
       [-1.40763984,  1.42710194],
       [ 0.05135313,  0.07524095]])
>>> gm.covariances_
array([[[ 1.14807234, -0.03270354],
        [-0.03270354,  0.95496237]],

       [[ 0.63478101,  0.72969804],
        [ 0.72969804,  1.1609872 ]],

       [[ 0.68809572,  0.79608475],
        [ 0.79608475,  1.21234145]]])
```

Soft Clustering

GMMs assign a probability (likelihood) to each data point belonging to each cluster rather than a hard assignment.

This means that each data point is assigned a probability distribution over all clusters, indicating the likelihood of it belonging to each cluster.

Number of clusters selection

With K-Means, you could use the inertia or the silhouette score to select the appropriate number of clusters. But with Gaussian mixtures, it is not possible to use these metrics because they are not reliable when the clusters are not spherical or have different sizes. Instead, you can try to find the model that minimizes a *theoretical information criterion*, such as the Bayesian information criterion (BIC) or the Akaike information criterion (AIC).

Equation 9-1. Bayesian information criterion (BIC) and Akaike information criterion (AIC)

$$BIC = \log(m)p - 2\log\left(\hat{L}\right)$$

$$AIC = 2p - 2\log\left(\hat{L}\right)$$

In these equations:

- m is the number of instances, as always.
- p is the number of parameters learned by the model.
- $\hat{L}$ is the maximized value of the *likelihood function* of the model.

To compute the BIC and AIC, call the `bic()` and `aic()` methods:

```
>>> gm.bic(X)
8189.74345832983
>>> gm.aic(X)
8102.518178214792
```

Both the BIC and the AIC penalize models that have more parameters to learn (e.g., more clusters) and reward models that fit the data well. They often end up selecting the same model. When they differ, the model selected by the BIC tends to be simpler (fewer parameters) than the one selected by the AIC, but tends to not fit the data quite as well (this is especially true for larger datasets).

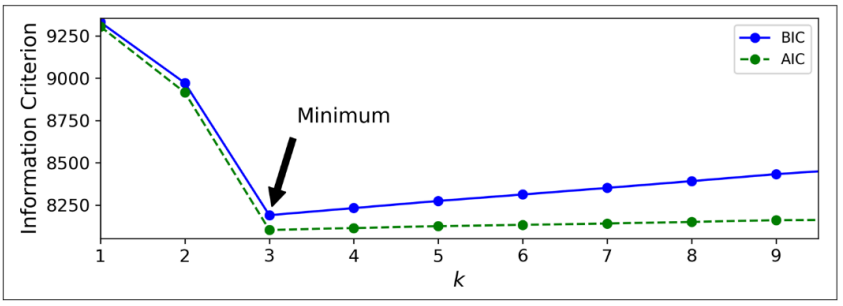


Figure 9-21. AIC and BIC for different numbers of clusters k

Likelihood Function:

- The likelihood function, on the other hand, is a function used in statistical inference.
- It tells you how likely a set of observed data is to occur given certain parameters of a statistical model.
- It's like flipping the script on the PDF (probability density function): instead of telling you the likelihood of getting a certain data point given the parameters, it tells you the likelihood of getting the observed data given certain parameter values.
- It's often used in maximum likelihood estimation, where we try to find the parameter values that maximize the likelihood of observing the data we have.
- The likelihood function helps us determine the best-fitting parameters for our statistical model based on the observed data.

Rather than manually searching for the optimal number of clusters, you can use the BayesianGaussianMixture class, which is capable of giving weights equal (or close) to zero to unnecessary clusters.

```
>>> from sklearn.mixture import BayesianGaussianMixture
>>> bgm = BayesianGaussianMixture(n_components=10, n_init=10)
>>> bgm.fit(X)
>>> np.round(bgm.weights_, 2)
array([0.4 , 0.21, 0.4 , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ])
```

Gaussian mixture models work great on clusters with ellipsoidal shapes, but if you try to fit a dataset with different shapes, you may have bad surprises.