Chapter 3: A Tour of Machine Learning Classifiers Using Scikit-Learn

Created	@August 18, 2025 3:36 PM
Class	ML with Pytorch and Scikit Learn

Best practices:

Use sklearn's test train split to split training and test data

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split()
X, y, test_size=0.3, random_state=1, stratify=y
```

Use StandarScaler to standardize your data:

```
from sklearn. from sklearn.preprocessing import StandardScaler
 sc = StandardScaler()
 sc.fit(X_train)
 X_train_std = sc.transform(X_train)
 X_test_std = sc.transform(X_test)
```

Use in built evaluation metrics like Accuracy score:

```
 from sklearn.metrics import accuracy_score
 >>> print('Accuracy: %.3f' % accuracy_score(y_test, y_pred))
 Accuracy: 0.97
```

1. Modeling class probabilities via logistic regression and Sigmoid function 🗿

- Logistic Regression is a **classification algorithm** (not regression in the usual sense).
- It is used when the output variable (**Y**) is **categorical** (e.g., Yes/No, 0/1, Spam/Not Spam).
- Unlike Linear regression which predicts continuous values e.g. for stocks, in classification we only predict class probabilities

2. The Sigmoid Function (a.k.a Logistic Function)

The **sigmoid function** squashes any real number into a value between **0 and 1**, making it perfect for probabilities.

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

- =wTx+b (linear combination of features)
- Output:
  - If σ(z)\sigma(z)σ(z) ≈ 1 → Class = 1
  - If σ(z)\sigma(z)σ(z) ≈ 0 → Class = 0

📌 The **decision boundary** is usually at 0.50.50.5:

$$\hat{y} = \begin{cases} 1 & \text{if } \sigma(z) \geq 0.5 \\ 0 & \text{if } \sigma(z) < 0.5 \end{cases}$$

$$\hat{y} = \sigma(z) = \frac{1}{1 + e^{-(w^T x + b)}}$$

Comparison with Threshold in Adaline

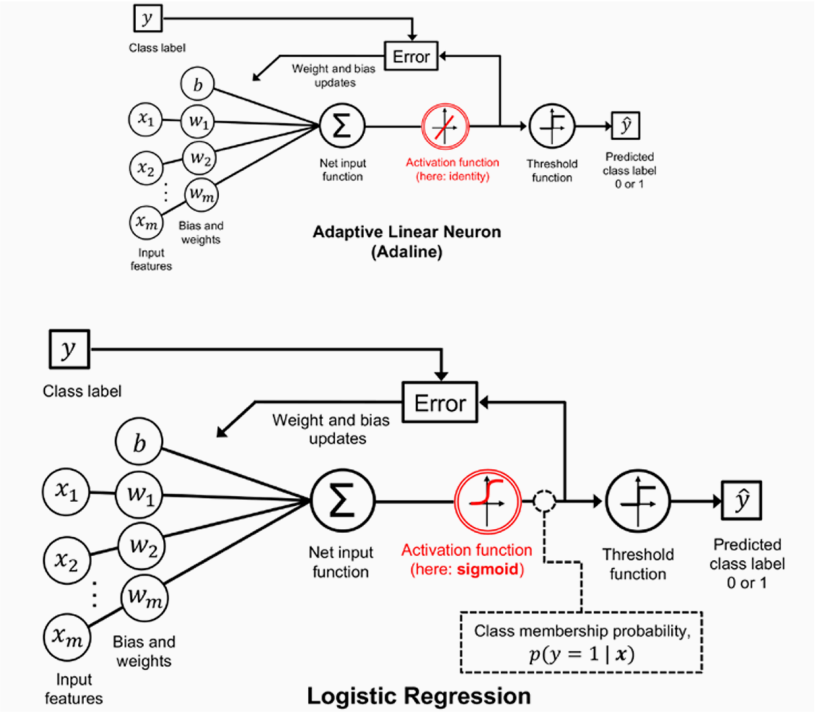

Figure 3.3: Logistic regression compared to Adaline

For our code implementation the code for logistic regression from scratch remains the exact same as SGD except for the **activation function which we will replace from a threshold function like before to our sigmoid function**

CODE IMPLEMENTATION: 🗿

```
def activation(self, z):
  #Sigmoid activation
  return 1./ (np.exp( -np.clip(z, -250, 250))) #-np.clip is used to limit the value of z within -250 to 250 to prevent overfitt
ing error
```

That'll be all for the sigmoid function 🗿

TACKLING OVERFITTING WITH REGULARIZATION

Overfitting occurs when a machine learning model performs well on training data but fails to generalize to unseen test data.

This problem indicates high variance in the model, typically caused by excessive complexity—too many parameters relative to the underlying data structure.

Similarly, our model can also suffer from underfitting (high bias), which means that our model is not complex enough to capture the pattern in the training data well and therefore also suffers from low performance on unseen data (It has skill issues)

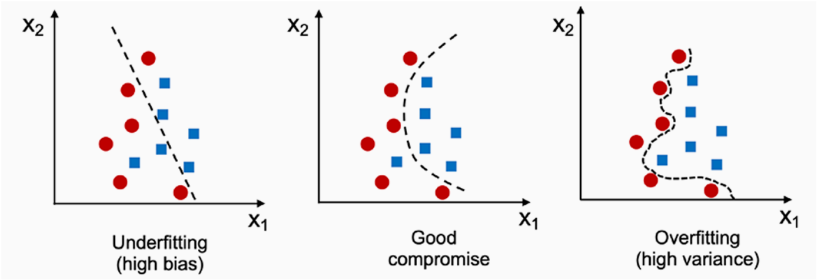

Figure 3.8: Examples of underfitted, well-fitted, and overfitted models

formula:

$$L(\mathbf{w}) = \mathcal{L}(\mathbf{w}) + \lambda \sum_{j=1}^{n} w_j^2$$

- L(w) → the original loss (e.g., MSE, cross-entropy).

- λ\lambdaλ → regularization strength (hyperparameter).
- ∑j=1nwj2\sum_{j=1}^{n} w_j^2∑j=1nwj2 → sum of squared weights (L2 norm squared).

✅ Effect:

- Prevents the model from having **very large weight values**.
- Helps reduce **overfitting** by encouraging simpler models.
- Larger λ → more shrinkage (weights pushed closer to zero).
- Smaller λ→ model behaves closer to unregularized training.

Support Vector Machines (SVMs)

Another powerful and widely used learning algorithm is the support vector machine (SVM), which can be considered an extension of the perceptron. Using the perceptron algorithm, we minimized misclassification errors.

However, in SVMs, our optimization objective is to maximize the margin.
The margin is defined as the distance between the separating hyperplane (decision boundary) and the training examples that are closest to this hyperplane, which are the so-called support vectors.

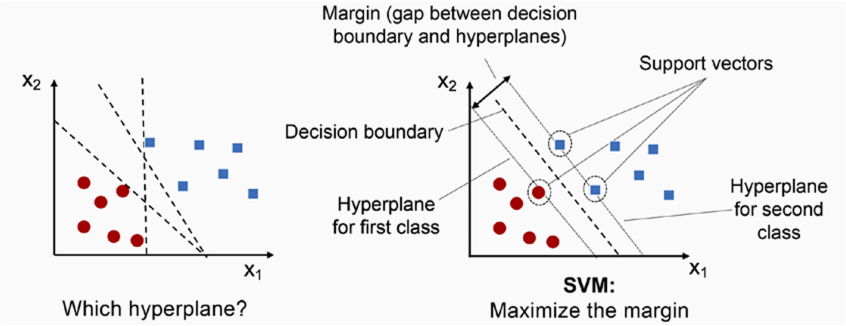

Figure 3.10: SVM maximizes the margin between the decision boundary and training data points

The rationale behind having decision boundaries with large margins is that they tend to have a lower generalization error, whereas models with small margins are more prone to overfitting.

CODE IMPLEMENTATION:

```
from sklearn.svm import SVC
svm = SVC(kernel='linear', C=1.0, random_state=1)
svm.fit(X_train_std, y_train
```

Kernelized SVMs (For Non-Linear Data)

The Problem: Non-linear data

Not all datasets can be separated by a straight line (linear boundary).

Example: XOR problem – no straight line separates the two classes.

Instead of trying to separate data in the **original input space**, map it into a **higher-dimensional feature space (3D MAP)** where the data becomes linearly separable.

Mathematically, we imagine a mapping:

$$\phi : \mathbb{R}^n \to \mathbb{R}^m, \quad m \gg n$$

But computing ϕ this can be expensive. Hence, we use the kernel trick:

Kernel Function

A kernel is a function that computes the **inner product** in feature space **without explicitly mapping the data.**

$$K(\mathbf{x}_i, \mathbf{x}_j) = \langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j) \rangle$$

Meaning we never compute ϕ directly, only the kernels

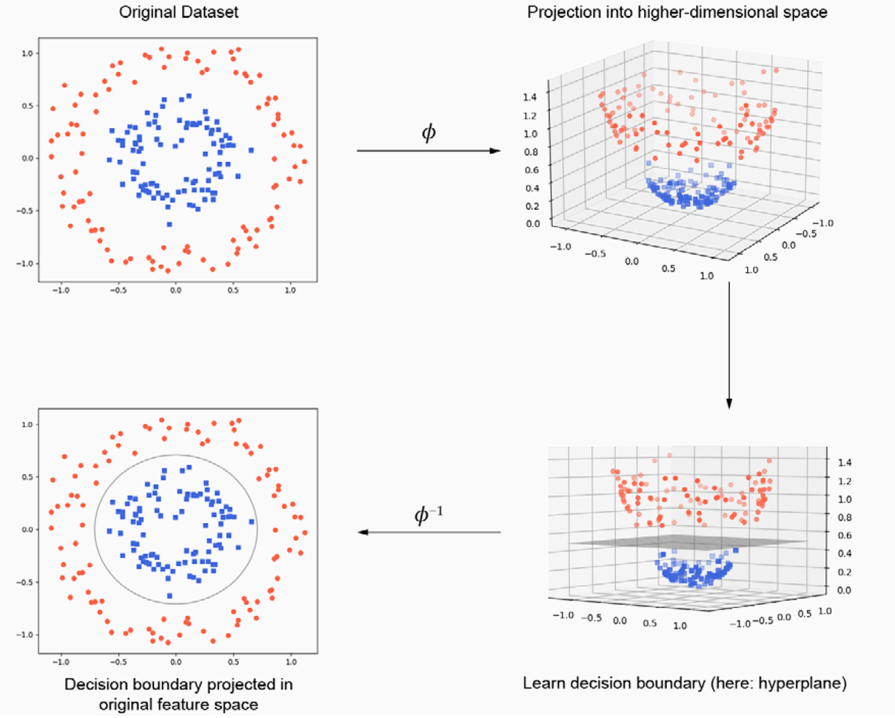

Figure 3.14: The process of classifying nonlinear data using kernel methods

Although we did not go into much detail about how to solve the quadratic programming task to train an SVM, in practice, we just need to replace the dot product:

$$\mathbf{x}^{(i)T}\mathbf{x}^{(j)}$$

with

$$\phi(\mathbf{x}^{(i)})^T\phi(\mathbf{x}^{(j)})$$

To save the expensive step of calculating this dot product between two points explicitly, we define a so-called **kernel function**:

$$K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \phi(\mathbf{x}^{(i)})^T\phi(\mathbf{x}^{(j)})$$

One of the most widely used kernels is the radial basis function (RBF) kernel, which can simply be called the Gaussian kernel

CODE IMPLEMENTATION

```
svm = SVC(kernel='rbf', random_state=1, gamma=0.10, C=10.0)
svm.fit(X_xor, y_xor)
```

DECISION TREES (MY FAV BTW)

Decision Trees, as the name suggests, work by **splitting information into branches based on conditions**, often in the form of **"yes" or "no" decisions**.

- The model learns a series of **if–then rules** (questions) from the training dataset.
- At each node, the tree chooses a feature and a threshold (for numerical features) or category (for categorical features) that best separates the data.
- This process continues until the data is classified into groups (leaves), which represent the final **class labels** or **predicted values**.

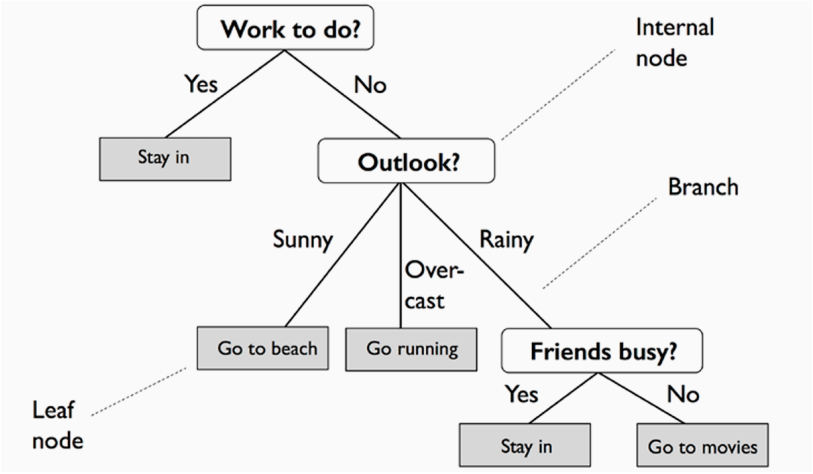

Figure 3.18: An example of a decision tree

To split the nodes at the most informative features, we need to define an objective function to optimize via the tree learning algorithm. Here, our objective function is to maximize the IG at each split, which we define as follows:

$$\mathrm{IG}(D_p, f) \;=\; H(D_p) \;-\; \sum_{j=1}^{k} \frac{N_j}{N_p} H(D_j)$$

where

$$\begin{aligned} D_p &= \text{parent dataset} \\ f &= \text{feature used for splitting} \\ D_j &= \text{subset of } D_p \text{ after splitting by feature } f \\ N_j &= \text{number of samples in subset } D_j \\ N_p &= \text{total number of samples in parent dataset } D_p \\ k &= \text{number of subsets after splitting} \end{aligned}$$

Here, we calculate the ***Information Gain (IG):***

Here, f is the feature used for splitting; Dp and Dj represent the datasets of the parent and jth child node; I is our impurity measure; Np indicates the total number of training examples at the parent node; and Nj is the number of examples in the jth child node.

Information gain is the difference between the parent node's impurity and the sum of the child nodes' impurities. When child nodes have lower impurity values, the information gain increases.

The three impurity measures or splitting criteria that are commonly used in binary decision trees are Gini impurity (IG), entropy (IH), and the classification error (IE).

CODE IMPLEMENTATION

```
from sklearn.tree import DecisionTreeClassifier

tree_model = DecisionTreeClassifier(criterion='gini', x_depth=4,random_state=1)
tree_model.fit(X_train, y_train)
```

ENTROPY FOR NON-EMPTY CLASSES ($p(i|t) \neq 0$):

Entropy measures **impurity or disorder** in a dataset.

- **0 → completely pure** (all samples same class)
- **Higher → more mixed / uncertain**

Formula

$$H(D) = -\sum_{i=1}^{k} p_i \log_2(p_i)$$

Where, *Pi:*

$$p_i = \frac{\text{number of samples in class } i}{\text{total number of samples}}$$

$$\text{IG}(D_p, f) = H(D_p) - \sum_{j=1}^{k} \frac{N_j}{N_p} H(D_j)$$

$$\begin{aligned} D_p &= \text{parent dataset} \\ f &= \text{feature used for splitting} \\ D_j &= \text{subset of } D_p \text{ after splitting by } f \\ N_j &= \text{number of samples in subset } D_j \\ N_p &= \text{total samples in parent dataset } D_p \\ k &= \text{number of subsets after splitting} \end{aligned}$$

Step-by-step to compute entropy

1. Count number of samples per class → Ni_iNi
2. Compute probability pi=Ni/Np_i = N_i / Np=Ni/N
3. Compute log2(pi)\log_2(p_i)log2(pi)
4. Multiply by pi_pi → weight by frequency
5. Sum for all classes and multiply by −11−1 → entropy

$$\textbf{Step-by-Step Intuition for Entropy:}1.\textbf{Meaning of } \log_2(p_i)$$
$$\log_2(p_i) \text{ asks: “2 to what power gives } p_i\text{?”}$$
$$\text{Example: } p_i = 0.25 \implies \log_2(0.25) = -2 \text{ because } 2^{-2} = 0.25$$
$$\text{Note: The log is negative for probabilities less than 1, meaning rare events give more “information.”}2.\textbf{Why mult}$$
$$\text{We weight the “surprise” of each class by its probability:}$$
$$p_i \cdot \log_2(p_i)$$
$$\text{Example: If a class occurs } 80\% \text{ of the time, its contribution is smaller than a } 20\% \text{ class, which is more surprising.}3$$
$$\text{Since } \log_2(p_i) \leq 0, \text{ multiplying by } -1 \text{ makes entropy positive.}\textbf{Entropy formula:}$$
$$H = -\sum_i p_i \log_2(p_i)\textbf{Worked example:}$$
$$\text{Suppose 2 classes, } 80\% \text{ Yes, } 20\% \text{ No:}$$
$$\log_2(0.8) \approx -0.322, \quad \log_2(0.2) \approx -2.322$$
$$0.8 \cdot (-0.322) = -0.258, \quad 0.2 \cdot (-2.322) = -0.464$$
$$\text{Sum: } -0.258 + (-0.464) = -0.722$$
$$\text{Apply negative: } H = -(-0.722) = 0.722 \text{➡ Entropy} = 0.722$$

CODE IMPLEMENTATION

```
import numpy as np

def entropy(p):
  return - p * np.log2(p) - (1-p) * np.log2(1-p)
```

RANDOM FORESTS

- Random Forest is an **ensemble learning method** that combines multiple decision trees to improve predictive accuracy and control overfitting.
- It can be used for both **classification** and **regression** problems.
- Idea: Build many decision trees and combine their predictions (majority voting for classification, average for regression).
- **Classification:** Majority vote from all trees.
- **Regression:** Average prediction from all trees.

CODE IMPLEMENTATION:

```
from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier(n_estimators=25, random_state=42, n_jobs=2)
model.fit(X_train, y_train)
```

K-NEAREST NEIGHBOURS (KNNs)

These will be the last supervised Algorithm in the 3rd chapter

KNN is a typical example of a lazy learner. It is called “lazy” not because of its apparent simplicity, but because it doesn’t learn a discriminative function from the training data but memorizes the training dataset instead

The KNN algorithm itself is fairly straightforward and can be summarized by the following steps:

1. Choose the number of k and a distance metric
2. Find the k-nearest neighbors of the data record that we want to classify
3. Assign the class label by majority vote

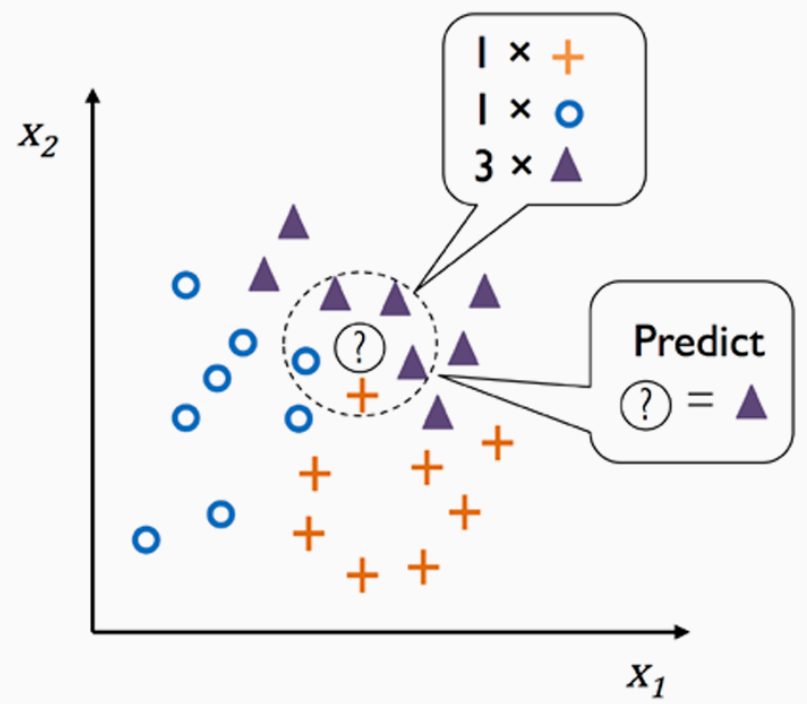

Figure 3.25: How k-nearest neighbors works

Based on the chosen distance metric, the KNN algorithm finds the k examples in the training dataset that are closest (most similar) to the point that we want to classify.

The class label of the data point is then determined by a majority vote among its k nearest neighbors.

CODE IMPLEMENTATION

```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=5, p=2, metric='minkowski')
knn.fit(X_train_std, y_train)
```

Lastly, it is important to mention that KNN is very susceptible to overfitting due to the **curse of dimensionality**

Curse of Dimensionality refers to various problems that arise when working with **high-dimensional data** (data with many features).

As the number of dimensions increases:

- Data becomes **sparse**.
- Distance metrics become **less meaningful**.
- Algorithms (like k-NN, clustering, or even regression) often **perform poorly** unless data size grows exponentially.

THE END!