Machine Learning for Classification

contact: *cursifrancesco@gmail.com*

Introduction

Classification is an approach for **supervised machine learning (ML)** to find the mapping between some input features and the output class of the element they correspond to.

The typical ML classification approaches are:

- Logistic regression;
- Naive Bayes Classifiers;
- K-Nearest neighbour (KNN);
- Support Vector Machine (SVM);
- Decision trees.

Classification Metrics:

Whatever classification algorithm is used, there exist some generally used metrics to estimate the ”goodness” of the predictions:

		Prediction	
		Positive	**Negative**
Ground Truth	**Positive**	True Pos (TP)	False Neg (FN)
	Negative	False Pos (FP)	True Neg (TN)

TABLE I: Confusion Matrix

- **Confusion Matrix:** it’s a matrix reporting the number of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN) for each predicted class.
- **Precision:** measures the percentage of correctly predicted positives and it is defined as $\frac{TP}{TP+FP}$.
- **Recall:** measures the percentage of true positives over the total actual positives $\frac{TP}{TP+FN}$.
- **Accuracy:** measures the percentage of positives over all the predictions $\frac{TP+FN}{TP+FP+FN+FP}$.
- **Specifity:** measures the percentage of the predicted negatives $\frac{TN}{TN+FP}$.
- **F1 Score:** is an overall measure that combines both precision and recall $2\frac{Pres*Recall}{Pres+Recall}$.

Logistic Regression

Logistic regression is generally used for **binary classification**, meaning only two classes are defined (e.g. positive and negative).

Given a dataset of set of input features $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N \in \mathbb{R}^m$, and output features $y_1, y_2, \ldots, y_N$, with each y_i that can have a value of either 0 or 1, the decision boundary to identify to which class an element i belongs to is computed as:

$$\hat{y}_i = \frac{1}{1 + e^{-\boldsymbol{\theta}^T \boldsymbol{x}_i}} \ , \tag{1}$$

where $\boldsymbol{\theta}$ is a vector of learnable parameters. This function is called **sigmoid** function and represents the predicted output probability of the element to belong to a certain class, namely $p(\hat{y}_i = 1)$

In order to optimize for the parameters $\boldsymbol{\theta}$, the **Binary Cross-Entropy Loss** is used:

$$L(\boldsymbol{\theta}) = \frac{1}{N} \sum_{i=1}^{N} y_i \log(\hat{y}_i) + (1 - y_i)(1 - \log(\hat{y}_i)) \ . \tag{2}$$

Once the cost function is minimized, the parameters $\boldsymbol{\theta}$ are obtained and the model trained.

Since a logistic regression model can only predict binary classes, inn case of multi-class classification problems the **one-vs-all** method is used. In this case, different models are implemented considering only binary cases. The final prediction will be the output of the model with highest probability.

Example:

Let's consider the iris dataset, that consists of $N = 150$ datapoints. The input features belong to a 4-dimensional ($m = 4$) space of sepal-length, sepal-width, petal-length, petal-width. The output are instead 3 classes, corresponding to the 3 possible iris types: Iris-setosa, Iris-versicolor, Iris-virginica.

Since ML approaches generally require numerical data and not categorical ones, the classes values need to be converted. Generally, two approaches exist:

- **Label Encoding**: each categorical class is assigned a number $0, 1, \ldots, C - 1$, where C is the number of classes. For instance, for the iris dataset the encoding would be $0, 1, 2$ meaning that Iris-setosa=0, Iris-versicolor=1, Iris-virginica =2. This approach works better if the classes are ordered (e.g. low, medium, high).
- **One-hot Encoding**: in this case, each categorical class is assigned a vector value, whose dimension equals the number of classes and with binary entries (either 0 or 1). For instance Iris-setosa=$[1\ 0\ 0]$, Iris-versicolor=$[0\ 1\ 0]$, Iris-virginica=$[0\ 0\ 1]$.

For the simple binary classification problem, only the data for two classes can be selected, for instance Iris-setosa and Iris-versicolor. Because of the encoding, the output vector will only have values 0 and 1s. The logistic regression model will then output the probability of the prediction to be 1, meaning of class Iris-versicolor.

The code below shows how to implement logistic regression in Python, using scikit-learn:

```
from sklearn.linear_model import LogisticRegression
clf = LogisticRegression(random_state=0).fit(X_train, y_train)
theta = clf.coef_  #learned parameters
theta0 = clf.intercept_ #learned offset (bias)
y_pred = clf.predict(X_train) #output class prediction
#get predicted probabilities for each class
prob_pred = clf.predict_proba(X_train)
```

Logistic regression works well for binary classification problems, but it might be slow in training depending on the data size. However, once the model is learnt, only its parameters are needed for the prediction, therefore it is fast.

Naive Bayes Classifiers

A Naive Bayes classifier is a probabilistic machine learning model that's used for classification task and that relies on Bayes Theorem:

$$p(B|A) = \frac{p(A|B)p(B)}{p(A)} \ . \tag{3}$$

A classification problem, to predict that the output belongs to a class C_k can therefore be formulated as:

$$p(y = C_k|\boldsymbol{x}) = \frac{p(\boldsymbol{x}|y = C_k)p(y = C_k)}{p(\boldsymbol{x})} \ . \tag{4}$$

Naive Byes Classifiers assume that the input features are all independent from each other, meaning that $p(\boldsymbol{x}|y) = p(x_1|y)p(x_2|y)\dots p(x_m|y)$ and $p(\boldsymbol{x}) = p(x_1)p(x_2)\dots p(x_m)$, which results in $p(y|\boldsymbol{x}) \propto p(y = C_k)\prod_{i=1}^{m} p(x_i|y = C_k)$.

Generally, Gaussian distribution can be chosen for describing each probability, which is then defined as:

$$p(z) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(\frac{z-\mu_z}{2\sigma})^2} \ . \tag{5}$$

Naive Bayes Classifiers work well for multiclass problems, yet in this case the predicted output class is obtained as the one with highest probability.

Example:

Let's consider the iris dataset. In this case the features are the 4 parameters sepal-length, sepal-width, petal-length, petal-width, and there are three possible classes.

The probability of each class, being them discrete, can be computed as

$$p(y = 0) = \frac{N_0}{N}, \ p(y = 1) = \frac{N_1}{N}, \ p(y = 2) = \frac{N_2}{N} \ , \tag{6}$$

where N_0, N_a, N_2 are the number of times each class occur, which for the given dataset correspond to $p(y = 0) = 38/150, \ p(y = 1) = 42/150, \ p(y = 2) = 40/150$.

The probability of each feature, instead, is computed from the Gaussian distribution, for each feature independently. Therefore each mean and variance of each feature distribution, for a given class output, is computed resulting in the sets:

$$\begin{aligned}
&(\mu_{1,y=0}, \sigma_{1,y=0}) \rightarrow p(x_1|y = 0); \ (\mu_{1,y=1}, \sigma_{1,y=1}) \rightarrow p(x_1|y = 1); \ (\mu_{1,y=2}, \sigma_{1,y=2}) \rightarrow p(x_1|y = 2) \\
&(\mu_{2,y=0}, \sigma_{2,y=0}) \rightarrow p(x_2|y = 0); \ (\mu_{2,y=1}, \sigma_{2,y=1}) \rightarrow p(x_2|y = 1); \ (\mu_{2,y=2}, \sigma_{2,y=2}) \rightarrow p(x_2|y = 2) \\
&(\mu_{3,y=0}, \sigma_{3,y=0}) \rightarrow p(x_3|y = 0); \ (\mu_{3,y=1}, \sigma_{3,y=1}) \rightarrow p(x_3|y = 1); \ (\mu_{3,y=2}, \sigma_{3,y=2}) \rightarrow p(x_3|y = 2) \\
&(\mu_{4,y=0}, \sigma_{4,y=0}) \rightarrow p(x_4|y = 0); \ (\mu_{4,y=1}, \sigma_{4,y=1}) \rightarrow p(x_4|y = 1); \ (\mu_{4,y=2}, \sigma_{4,y=2}) \rightarrow p(x_4|y = 2)
\end{aligned} \tag{7}$$

It then turns out that:

$$\begin{aligned}
p(\boldsymbol{x}|y = 0) &= p(x_1|y = 0)p(x_2|y = 0)p(x_3|y = 0)p(x_4|y = 0) \\
p(\boldsymbol{x}|y = 1) &= p(x_1|y = 1)p(x_2|y = 1)p(x_3|y = 1)p(x_4|y = 1) \\
p(\boldsymbol{x}|y = 0) &= p(x_1|y = 2)p(x_2|y = 2)p(x_3|y = 2)p(x_4|y = 2)
\end{aligned} \tag{8}$$

from which the predicted distributions can be obtained.

In Python, such procedure is already included in the libraries and a Naive Bayes classifier can be easily performed as:

```python
from sklearn.naive_bayes import GaussianNB
classifier = GaussianNB()
classifier.fit(X_train, y_train)
y_pred  =  classifier.predict(X_test)
```

The advantage of Naive Bayes Classifiers is that they are very easy to implement and do not require any sort of training. Only the means and standard deviations of the distributions need to be retained, making it also computationally efficient and fast in predictions.

The main drawback is in the assumption of independence of the features, which might be very unrealistic.

K- Nearest Neighbour

K-Nearest Neighbour (KNN) is a classification technique that classifies an element according to the values of its k nearest ones in the feature space.

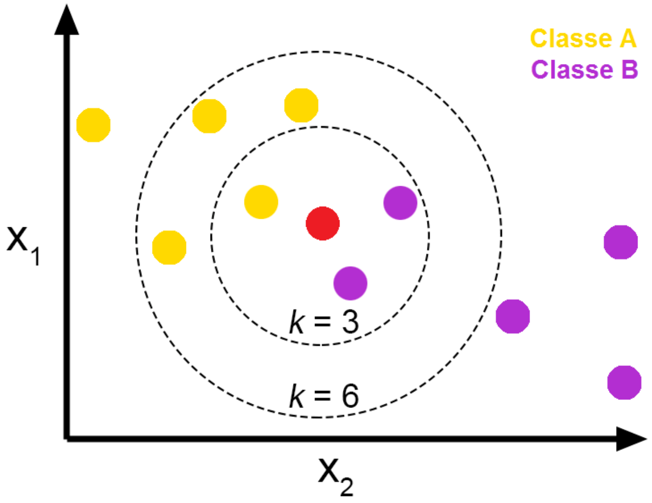

Fig. 1: Example of KNN with different K values.

Consider an input point $\boldsymbol{x}_j$ (the red dot in Figure 1), given a specified value of k for the nearest points. The output class can be obtained by means of $uniform$ voting or $weighted$ voting.

In the $uniform$ voting:

- compute distance of $\boldsymbol{x}_j$ from each closest k point;
- get the number of elements for each class;
- output the class with highest number of elements.

In the $weighted$ voting:

- compute distance of $\boldsymbol{x}_j$ from each closest k point;
- compute the weights as inverse of the distances;
- normalize the weights over the sum of all the weights;
- sum the weights for the points belonging to each class and find the mean one;
- assign the output class to the class with highest weight.

A. *Example:*

Let's consider the iris dataset and we want to classify the input value $\boldsymbol{x}_{test} = [6.7\ 3.0\ 5.0\ 1.7]$ with $k = 5$ nearest neighbours. We use Euclidean distance as metrics meaning that:

$$d(\boldsymbol{x}_j, \boldsymbol{x}_k) = ||\boldsymbol{x}_j - \boldsymbol{x}_k|| = \sqrt{(x_{j,1} - x_{k,1})^2 + (x_{j,2} - x_{k,2})^2 + \ldots + (x_{j,m} - x_{k,m})^2} \tag{9}$$

where, in this case $m = 4$ is the input features' dimension.

Let's suppose that the algorithm identifies the k closest points as those reported in the table, and their corresponding classes. The table also reports the distances for each neighbour to our test point In this case, if the $uniform$ voting is chosen there are 3 closest points belonging to class 1, and only 2 to class 2, so the predicted class will be class 1.

	Coordinates	**Class**	**Distance**	**Inverse Distance**
$\boldsymbol{x}_1$	[6.9 3.1 4.9 1.5]	1	0.31622777	3.1623
$\boldsymbol{x}_2$	[6.7 3.1 4.7 1.5]	1	0.37416574	2.6726
$\boldsymbol{x}_3$	[6.5 3.0 5.2 2.0]	2	0.41231056	2.4254
$\boldsymbol{x}_4$	[6.8 2.8 4.8 1.4]	1	0.42426407	2.3570
$\boldsymbol{x}_5$	[6.3 2.8 5.1 1.5]	2	0.5	2.0000

TABLE II: K nearest neighbours for $\boldsymbol{x}_{test} = [6.7\ 3.0\ 5.0\ 1.7]$

If $weighted$ voting is chosen, then the weights are assigned as inverse distances. For class 1 the weight would be $w_1 = 3.1623 + 2.6726 + 2.3570 = 8.1919$, for class 2 it is $w_2 = 2.4254 + 2.0000 = 4.4254$. The weights can also be normalized, but in any case since $w_1 > w_2$ the predicted class is class 1.

In Python a KNN classifier can be built as:

```python
from sklearn.neighbors import KNeighborsClassifier
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib.colors import ListedColormap

model = KNeighborsClassifier(n_neighbors=10,weights='distance')
model.fit(X_train,y_train)
y_pred = model.predict(X_test)

#get indexes and distances for a test point
test_idx = 17
k_dist, k_idx= model.kneighbors(X_test[test_idx,:].reshape(1,-1), 5, return_distance=True)
y_pred = model.predict(X_test[test_idx,:].reshape(1,-1))
k_dist = k_dist[0] #take only values for single query
k_idx = k_idx[0] #take only values for single query
Points = X_train[k_idx,:]
Classes = y_train[k_idx]
```

It is also possible to draw the Voronoi regions to show the decision boundaries. For simplicity, a 2D feature space is considered:

```python
X_train = X_train[:,0:2] #take only 2 features
h = 0.02  # step size in the mesh

# Create color maps
cmap_light = ListedColormap(["orange", "cyan", "cornflowerblue"])
cmap_bold = ["darkorange", "c", "darkblue"]

#make the plots based on the two voting approahces
for weights in ["uniform", "distance"]:
    # we create an instance of Neighbours Classifier and fit the data.
    clf = KNeighborsClassifier(n_neighbors=10, weights=weights)
    clf.fit(X_train, y_train)

    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    x_min, x_max = X_train[:, 0].min() - 1,  X_train[:, 0].max() + 1
    y_min, y_max =  X_train[:, 1].min() - 1,  X_train[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.figure(figsize=(8, 6))
    plt.contourf(xx, yy, Z, cmap=cmap_light)

    # Plot also the training points
    sns.scatterplot(
        x= X_train[:, 0],
        y= X_train[:, 1],
        # hue=iris.target_names[y],
        palette=cmap_bold,
        alpha=1.0,
        edgecolor="black",
    )
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.title(
        "3-Class classification (k = %i, weights = '%s')" % (10, weights)
    )
    # plt.xlabel(iris.feature_names[0])
    # plt.ylabel(iris.feature_names[1])

plt.show()
```

KNN works well for multiclass problems and it's fast in training (it actually does not require any parameter training). The main drawback is that it needs all the datapoints to be stored, to make many computations to find and sort the minimum distances, resulting in high computational costs.

Support Vector Machine

Support Vector Machine (SVM) is an algorithm that tries to find the optimal hyperplane (in a linear case) to split two classes. It is also called **Maximum Margin Classifier** because it tries to maximize the distance between the datapoints of the two classes, namely the margin.

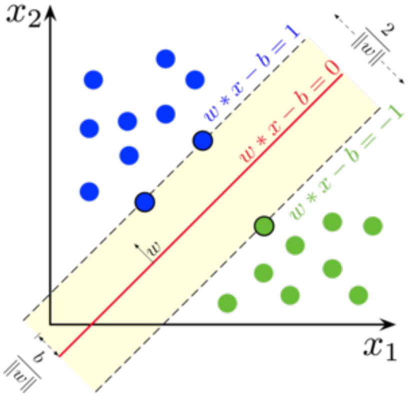

Fig. 2: SVM example

This margin is a kind of safety measure to reduce the possibility of misclassifing an input. The margin is the shaded area in the Figure 2.

The **Support Vectors** are the data points lying at the border of the margin, meaning those closest to the separation line.

Given an input point $\boldsymbol{x}_i$, and two possible classes identified by the values $C_1 = 1$ and $C_2 = -1$, the predicted output will be:

$$\hat{y}_i = \begin{cases} C_1, \text{if } \boldsymbol{\theta}^T \boldsymbol{x_i} \geq 1, \\ C_2, \text{if } \boldsymbol{\theta}^T \boldsymbol{x_i} \leq -1 \end{cases} \tag{10}$$

The support vectors are those points belonging to the lines $\boldsymbol{\theta}^T \boldsymbol{x_i} = 1$ and $\boldsymbol{\theta}^T \boldsymbol{x_i} = -1$. Here we are supposing the vector $\boldsymbol{\theta}$ includes both the values of the weights $\boldsymbol{w}$ for the inclination of the hyperplane and the offset b.

Given a dataset of input values $\boldsymbol{x}_i$ and corresponding output values y_i, with $y_i = 1$ or $y_i = -1$ and $i = 1 \dots N$, the hyperplane is thus identified by means of an optimization problem as:

$$\begin{aligned} &\min_{\boldsymbol{\theta}} ||\boldsymbol{\theta}||^2 \\ &\text{s.t } \boldsymbol{\theta}^T \boldsymbol{x_i} \geq 1 \text{ if } y_i = 1 \\ &\text{s.t } \boldsymbol{\theta}^T \boldsymbol{x_i} \leq -1 \text{ if } y_i = -1 \end{aligned} \tag{11}$$

which can also be reformulated as:

$$\begin{aligned} &\min_{\boldsymbol{\theta}} ||\boldsymbol{\theta}||^2 \\ &\text{s.t } y_i \boldsymbol{\theta}^T \boldsymbol{x_i} \geq 1 \text{ for } i = 1 \dots N \end{aligned} \tag{12}$$

If the data are not linearly separable, then the **Kernel trick** can be used to transofrm the data points into nonlinear variables $\boldsymbol{z}_i = \boldsymbol{f}(\boldsymbol{x}_i)$, where $\boldsymbol{f}(\cdot)$ is any nonlinear function (e.g. Gaussian distribution). In this scenario, a nonlinear decision boundary can be obtained from:

$$\begin{aligned} &\min_{\boldsymbol{\theta}} ||\boldsymbol{\theta}||^2 \\ &\text{s.t } y_i \boldsymbol{\theta}^T \boldsymbol{f}(\boldsymbol{x}_i) \geq 1 \text{ for } i = 1 \dots N \end{aligned} \tag{13}$$

In Python, an SVM classifier can be learnt as:

```python
from sklearn.svm import SVC #SVC = SVM classifier

svclassifier = SVC(kernel='rbf')
svclassifier.fit(X_train,y_train)
y_pred = svclassifier.predict(X_test)
```

Similarly to logistic regression, SVM work well for binary classification. For multiclass problems the **One-vs-all** approach needs to be used. Moreover, once the model is learnt (meaning the parameters $\boldsymbol{\theta}$ for the decision boundary obtained), the prediction is very easy and fast and does not require storing the whole data.

Decision Trees

Decision Trees are a ML approach that classify data according to a sort of $if..else$ logic.

Decision trees consist of nodes and branches. The top nodes are the roots and those with no additional splitting are the leaves.

The approach used for classification is named **Divide and Conquer** since decision trees iteratively separate the datapoints into smaller and smaller partitions.

Different approaches are used to terminate the algorithm and the splitting, for instance by defining a maximum depth or until the elements in each leaf belong o a single class only (or up to a certain percentage).

There exist various metrics to properly split the data and they are all based on maximizing the **Information Gain**. Typical measures are:

- **Classification Error**
- **Entropy**
- **Gini Index**

In Python a decision tree classifier can be obtained as:

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

clf = DecisionTreeClassifier()

# Train Decision Tree Classifer
clf = clf.fit(X_train,y_train)

#Predict the response for test dataset
y_pred = clf.predict(X_test)

#Plot the tree
fig = plt.figure(figsize=(25,20))
_ = tree.plot_tree(clf,
                   feature_names=['sepal-length', 'sepal-width', 'petal-length', 'petal-width'],
                   class_names=["0", "1", "2"],
                   filled=True)
```

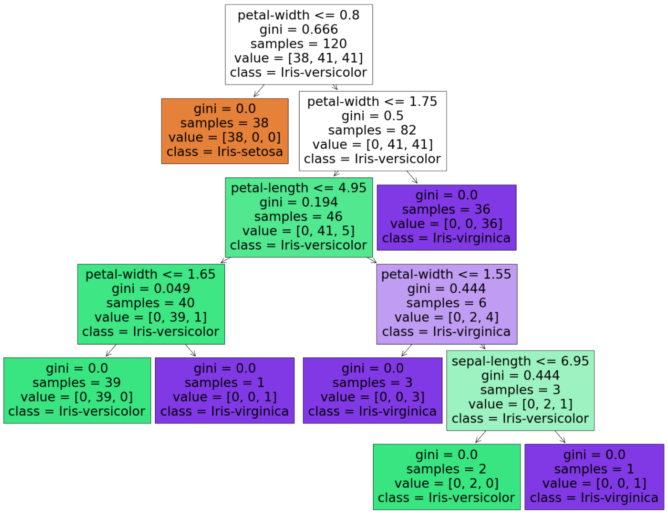

Fig. 3: Example of Decision Tree for the iris dataset

One of the main issues of decision trees is that they tend to overfit the data. One solution is to use **Ensamble methods** like **Bagging**. In Bagging, different decision trees are created and the final predicted class is obtained by voting. For instance, if three trees are created and two of them vote for the output to be of class A, then the predicted output will be class A.

Random Forests are another possible solution. In Bagging, the decision trees are all built from the same features, eventually only from different subsets. In Random forests, instead, each tree is built both on a random subset of dataoints and considering only a random set of input features.