



**Politecnico
di Torino**

Lab 4: Perceptron and SVM

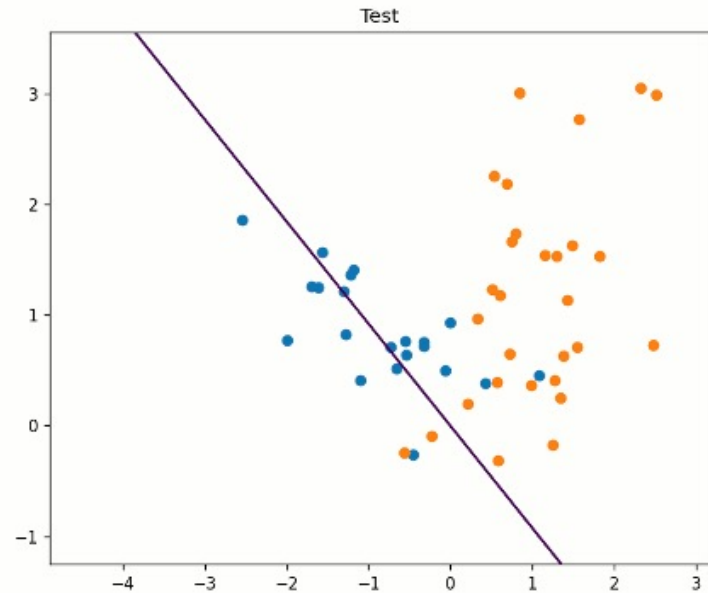
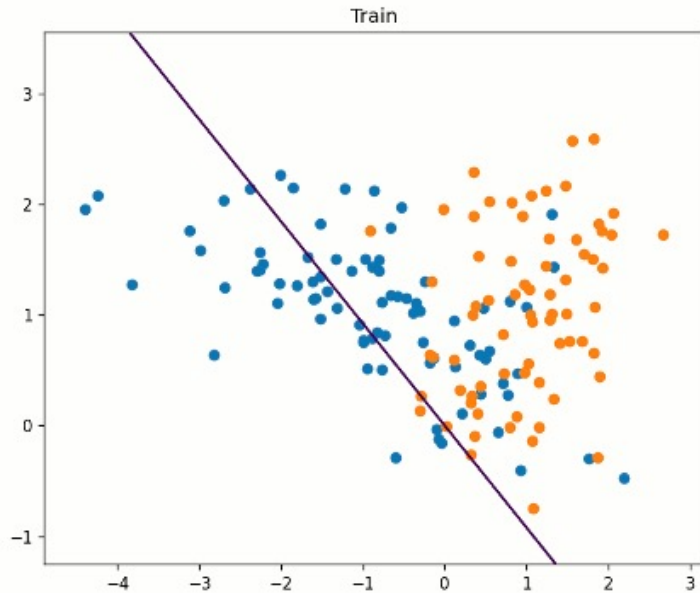
Machine Learning and Deep Learning 2023

Chiara Plizzari, Gabriele Berton

Perceptron

The perceptron is a **binary classifier**

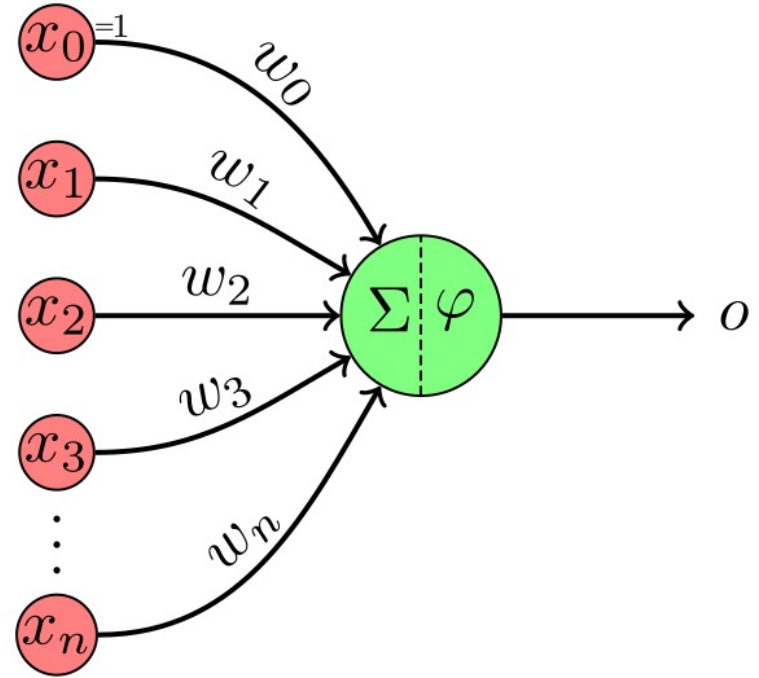
Iteration: 1/100



Perceptron

$$z = \sum_{i=0}^n x_i w_i$$

$$\phi(z) = \begin{cases} -1 & z \leq 0 \\ 1 & z > 0 \end{cases}$$



Training the Perceptron

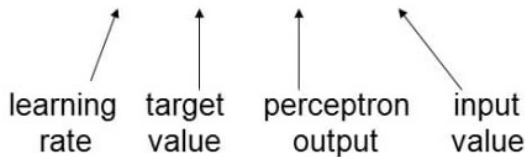
Two hyper-parameters to tune:

- **Learning rate:** correction factor applied to weights when the model wrongly predicts a sample;
- **n_iter:** number of repetitions of the training process.

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = \eta(t - o)x_i$$



learning rate target value perceptron output input value

Let's dive into the code

```
import numpy as np

class Perceptron:
    def __init__(self, learning_rate=0.1):
        self.learning_rate = learning_rate
        self.n_iter = n_iter
        self._b = 0.0
        self._w = None
        self.misclassified_samples = []
```

Let's dive into the code

We use the **predict** function that predicts a binary output (-1 or +1)

```
def f(self, x: np.array) -> float:
    return np.dot(x, self._w) + self._b

def predict(self, x: np.array) -> int:
    return np.where(self.f(x) >= 0.0, 1, -1)
```

Let's dive into the code

The core method which actually trains the model is ***fit***.

It receives two parameters:

- ***x***: an $m \times n$ matrix where m is the number of samples in the training set and n is the number of features.
- ***y***: array of labels associated to i -th sample in x .

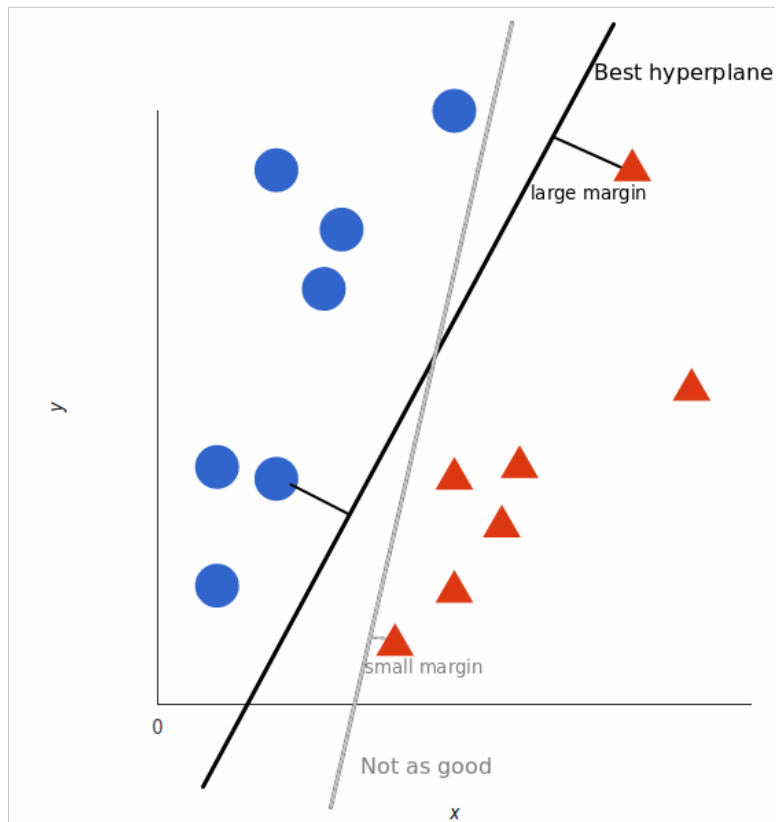
For each sample of each iteration it computes *update* which is the distance between the predicted label and actual label scaled by *learning_rate*. If the prediction is correct *update* is equal to 0. Otherwise the weights array is updated to take into account the error.

Let's dive into the code

```
def fit(self, x: np.array, y: np.array):
    self._b = 0.0
    self._w = np.zeros(x.shape[1])
    self._misclassified_samples = []

    for _ in range(self.n_iter):
        errors = 0 # errors during this iteration
        for xi, yi in zip(x, y):
            update = self.learning_rate * (yi - self.predict(xi))
            self._b += update
            self._w += update * xi
            errors += int(update != 0.0)
        self.misclassified_samples.append(errors)
```

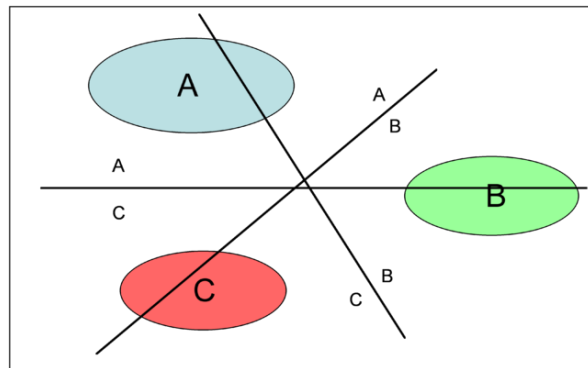
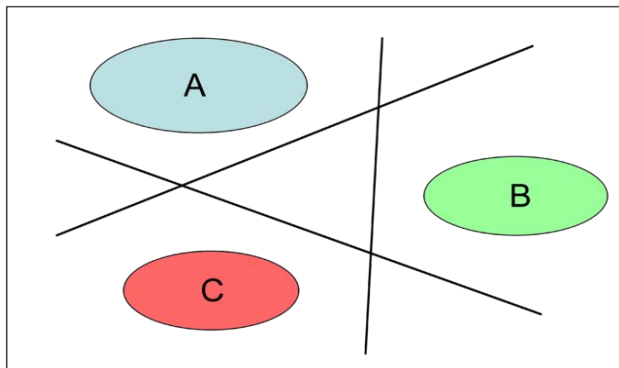

How does classification work in SVM?



How to deal with multiple classes?

The SVM classifier is defined as a binary classifier. How to deal with multiple classes?

- One-vs-all
- One-vs-one



In practice the SVM libraries will handle this for you, just keep in mind this is what is happening behind the scenes

What hyper parameters for SVM?

Linear: C

Polygon: C, degree and coef0

RBF: C and gamma

- other kernels are available

Check the docs:

```
#Import svm model
from sklearn import svm

#Create a svm Classifier
clf = svm.SVC(kernel='linear', C=1E10) # Linear Kernel

#Train the model using the training sets
clf.fit(X_train, y_train)

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

- <http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html>
- <http://scikit-learn.org/stable/modules/svm.html#svm-kernels>

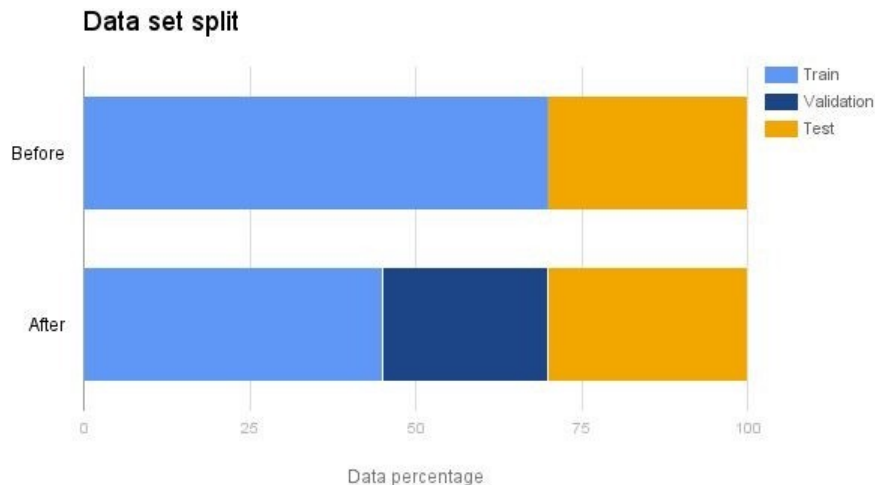
Let's dive into the code

- **Regularization C:** Controls the trade-off between decision boundary and misclassification term. A smaller value of C creates a small-margin hyperplane and a larger value of C creates a larger-margin hyperplane.
- **Gamma:** Low value of gamma considers only nearby points in calculating the separation line, while a value of gamma considers all the data points in the calculation of the separation line.

How to choose the parameters?

You cannot choose the optimal values based on your test set; in real problems you won't have access to it!

The proper way to evaluate a method is to split the data into 3 parts



How to choose the parameters?

You choose some parameters and **train** your model on the **training set**.

You then evaluate your performances on the **validation set**.

Once you find the parameters which work best on the **validation set**, you apply the **same** model on the **test set**. This is the correct estimate of the accuracy you will get on unseen data.

Changing the model based on performance on the test set is cheating!

Example - choosing C

Linear SVM -> we must optimize C

We select an appropriate range

$C = [10^{-3}, 10^{-2}, 10^{-1}, 10^0, 10^1, 10^2, 10^3]$



C value:	10^{-3}	10^{-2}	10^{-1}	10^0	10^1	10^2	10^3
Score on Validation	89%	90%	92%	85%	80%	80%	80%

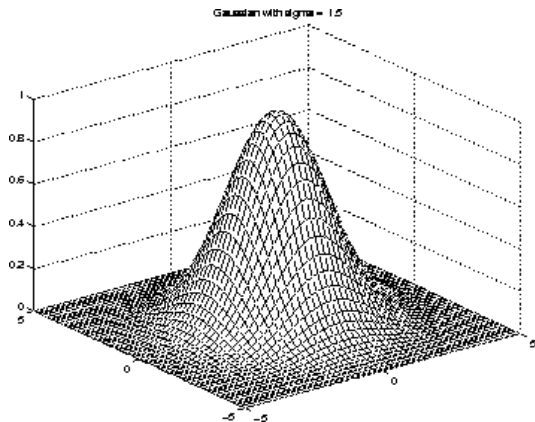
Example - grid search

What if there are two (or more) parameters to optimize?

Grid search: try all possible combinations of these parameters.

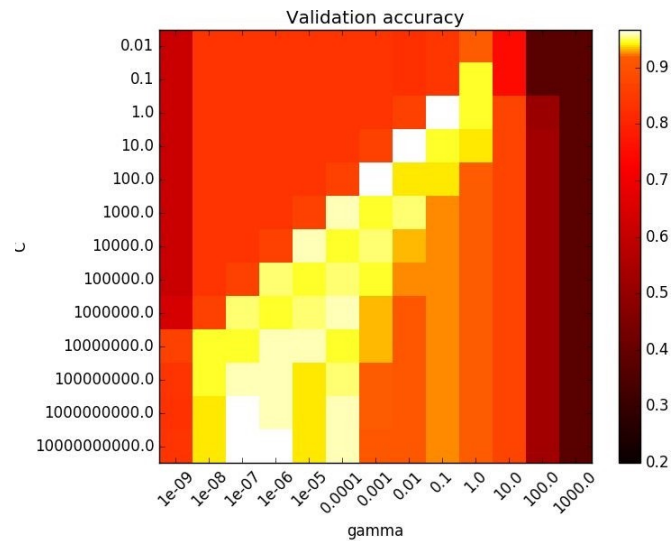
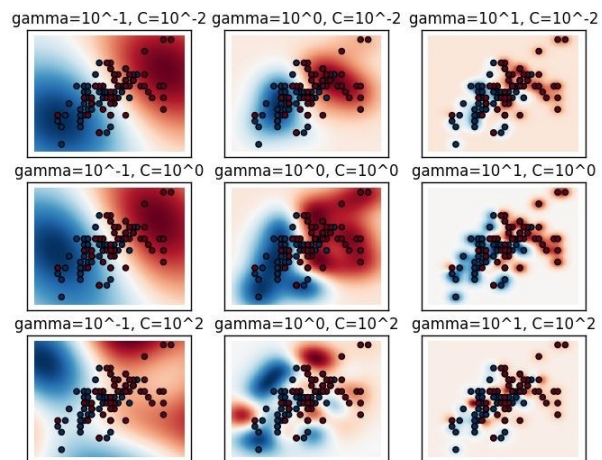
In the **RBF** case we have:

C and **gamma**



Gamma \ C	10^{-2}	10^{-1}	10^0	10^1
10^{-9}	66%	75%	60%	60%
10^{-7}	48%	67%	78%	79%
10^{-5}	80%	82%	90%	85%
10^{-3}	45%	66%	78%	74%

Grid search



Exercise 1

1. Load *Iris* dataset
2. Select the first two features and the first two categories.
3. Randomly split data into train, validation and test sets in proportion 5:2:3
4. For learning rate in [0.01, 0.1, 0.5]:
 - Train a perceptron on the training set;
 - Plot the data and the decision regions (hint: `plt.scatter()`, `plt.contourf()`, `np.meshgrid()`)
 - Evaluate on the validation set.
5. Use the best value on the validation and test on the test set. How do performance vary?

Exercise 2

1. Load *Iris* dataset
2. Simply select the first two dimensions
3. Randomly split data into train, validation and test sets in proportion 5:2:3
4. For C from 10^{-3} to 10^3 : (*multiplying at each step by 10*)
 - a. Train a **linear** SVM on the training set.
 - b. Plot the data and the decision boundaries
 - c. Evaluate the method on the validation set
5. Plot a graph showing how the accuracy on the validation set varies when changing C
6. How do the boundaries change? Why?
7. Use the best value of C and evaluate the model on the **test set**. How well does it go?

Exercise 2

8. Repeat point 4. (train, plot, etc..), but this time use an RBF kernel
9. Evaluate the best C on the **test set**.
10. Are there any differences compared to the linear kernel? How are the boundaries different?
11. Perform a grid search of the best parameters for an RBF kernel: we will now tune both ***gamma*** and C at the same time. Select an appropriate range for both parameters. Train the model and score it on the validation set.
12. Show the table showing how these parameters score on the validation set.
13. Evaluate the best parameters on the test set. Plot the decision boundaries.

Your turn now! Questions?

