Notes on understanding classifiers

[work in progress]

Tomasz Golan

January 29, 2016

Abstract

This note is related to Simple Classifier project (https://github.com/TomaszGolan/simpleClassifiers) which was an exercise to understand two classifiers: k-Nearest Neighbors (kNN) and Support Vector Machine (SVM) . Two classes of 2D points are considered within the project: *separable* (below or above $f(x) = x$)) and *inseparable* (inside or outside circle). The purpose of this note is to explain technical details of both algorithms and demonstrate how they work on simple examples. Please note, I am not an expert in the field and the note is rather how I understand the problem.

1 Introduction

A classifier, in machine learning, is an algorithm for finding a class of studying object, based on a set of objects with known class (training set). Thus, it is a type of supervised learning.

If a classification can only distinguish between two training sets it is called binary classification, otherwise it is multiclass classification. SVM is a binary classifier, so several SVMs combination is required for multiclass classification. KNN is a multiclass classifier. In this note, only binary problems are considered, so combining binary classifiers problem is not discussed.

For any supervised learning the choice of a training set is crucial. They should uniformly cover the whole phase space. The optimal size of the training set is unique for a problem. It depends on the classification method, the complexity of the problem, and on the separability of classes. To determine the optimal size of the training set one can use learning curves, which presents the error of classification as a function of the training set size.

For complex problems, the choice of the proper features to feed a classifier is important and non-trivial. They must be chosen to well define a membership to some class. Unless you are a genius you will probably need to make some tests to determine the right set of properties to learn a classifier. In this note, only simple examples are discussed, so there is no problem with the choice of features.

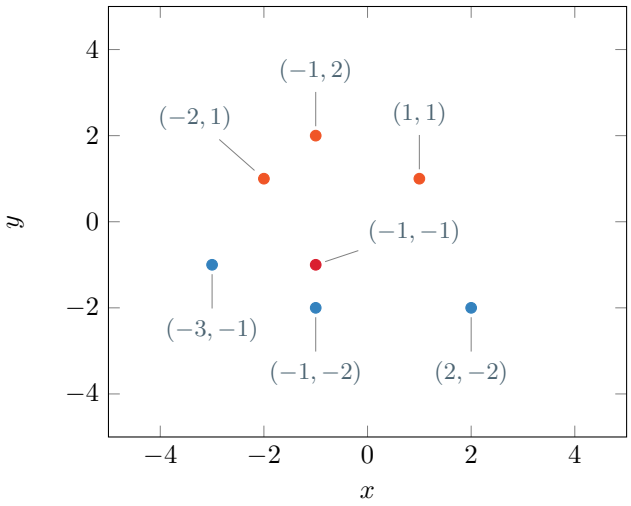


Figure 1: Two classes of points: blue and orange. The red dot represents the point to classify.

class	coordinates	distance^2
blue	$(-3, -1)$	4
blue	$(-1, -2)$	1
blue	$(2, -2)$	10
orange	$(-1, 2)$	9
orange	$(1, 1)$	8
orange	$(-2, 1)$	9

Table 1: The list of distances between the red point and each training example from Fig. 1.

2 k-Nearest Neighbors

k-Nearest Neighbors is one of the simplest machine learning algorithms[1]. Lets assume N training examples in our feature space are given by:

- a feature vector $\vec{x} = (x_1, x_2, ..., x_n)$
- a class membership y

For a given vector $\vec{x}'$ its membership y' must be determined. The idea is to look at k nearest neighbors and let the majority decide - if among the k nearest neighbors the most of training vectors belongs to the class y_j, the given vector also belongs to y_j. The optimal value of k is unique for the problem and must be determine empirically.

Lets consider the example presented on Fig. 1. There are two classes of points: blue and orange. The size of the training set is 6 (3 for each class). The red point is the one to classify (by eye it looks like it should belongs to blue points).

[1] kNN is also used for regression. In this note, only the use for classification is discussed.

class	coordinates	distance^2	weight
blue	$(-3, -1)$	4	0.250
blue	$(-1, -2)$	1	1.000
blue	$(2, -2)$	10	0.100
orange	$(-1, 2)$	9	0.111
orange	$(1, 1)$	8	0.125
orange	$(-2, 1)$	9	0.111

Table 2: The list of distances between the red point and each training example from Fig. 1. The vote weight is defined as 1 / distance^2.

The distances^2 between the red point and each training example are presented in Tab. 1. Euclidean metric was used, however, the choice of a metric is arbitrary and Chebyshev distance, cosine similarity etc. can be used as well. Now, lets take a look how the red point is classified for different k:

blue for $k = 1, 2, 3$

orange for $k = 5$

tie for $k = 4, 6$

This was ultra-simple example to demonstrate the method, but still some conclusions can be drawn:

- for binary classification k should be odd to avoid a tie
- large k does not necessary mean better results (especially when $k \sim$ training sets size)
- one can consider simple expansion with *vote weight* depending on the distance

Lets consider once again the given example, but now each point votes for a class membership with the weight given by 1 / distance^2, as in Tab. 2. Now, for any $k \in [1, 6]$ the red point is classified as blue:

k = 1: 1.000 vs 0.000

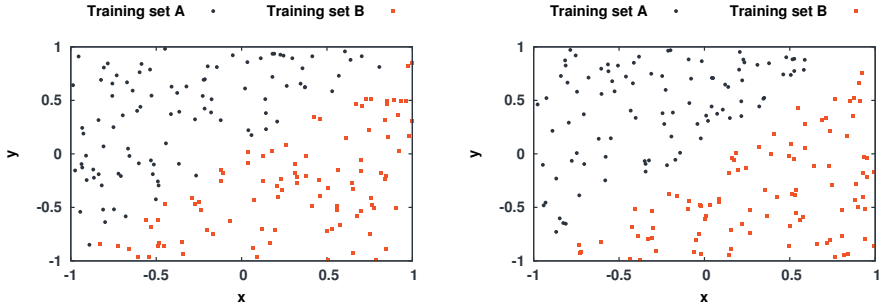
k = 2: 1.250 vs 0.000

k = 3: 1.250 vs 0.125

k = 4: 1.250 vs 0.236

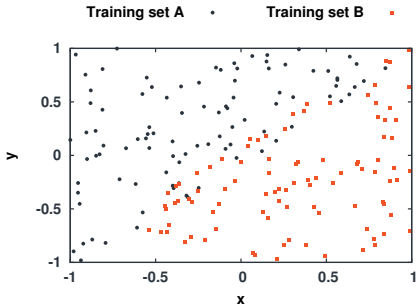
k = 5: 1.250 vs 0.347

k = 6: 1.350 vs 0.347



(a) Default

(b) Distant



(c) Overlapped

Figure 2: Training sets for points separable by $f(x) = x$.

2.1 Separable points

Lets consider something defined mathematically better than *blue* and *orange*, i.e. two classes of points which can be separated linearly, e.g. by the line $f(x) = x$ as presented on Fig. 2 in three scenarios:

- points are exactly separated by $f(x) = x$, Fig. 2(a)
- there is a small gap to make points more separated, Fig. 2(b)
- points overlap a little, Fig. 2(c)

For each scenario the efficiency of kNN as a function of the size of training sets (n) and k is checked in the following way:

1. generate n random points above $f(x) = x$ (training set A)
2. generate n random points below $f(x) = x$ (training set B)
3. generate a random point
4. find k nearest neighbors of this point
5. let them vote:
   - with vote weight = 1 (unweighted)
   - with vote weight = 1/distance (weighted)
6. repeat points 3-5 N times to get good enough statistics ($N = 10^5$ for presented results, but every 100 test points new training sets are generated)
7. calculate the score = no. of correctly guessed points / no. of all test points

The efficiency of kNN as a function of k and n can be found on Fig. 3. Clearly, weighted voting gives better results, especially for $k \sim n$. However, even unweighted kNN still gives score better than 90%. The efficiency is almost the same when extra gap is generated between training sets (Figs. 3(c) and 3(d)), but it becomes much worse if training sets overlap (Figs. 3(e) and 3(f)).

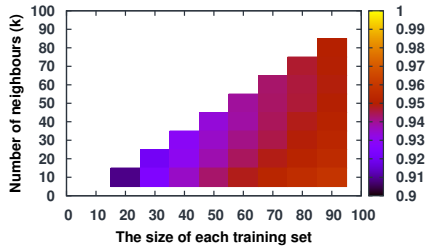
Lets take a look what happens when training sets are not uniformly distributed, as presented on Fig. 4. It still looks (by eye) that training sets are separated by $f(x) = x$, see Fig. 4(a). However, from obvious reasons some points are going to be incorrectly classified, as presented on Fig. 4(b).

2.2 Inseparable points

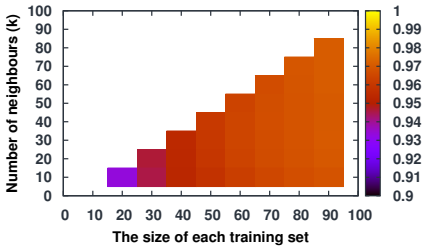
Lets now consider points which are not linearly separable, i.e. points inside and outside a circle, as presented on Fig. 5. Once again, three cases are investigated: default (points exactly separated by a circle, Fig. 5(a)), distant (with extra gap between training sets, Fig. 5(b)), and overlapped (training sets are allowed to overlap a little, Fig. 5(c)).

The efficiency, presented on Fig. 6, is calculated exactly the same way as in Sec. 2.1. Please note the different scale which now starts at 0.4. Clearly,

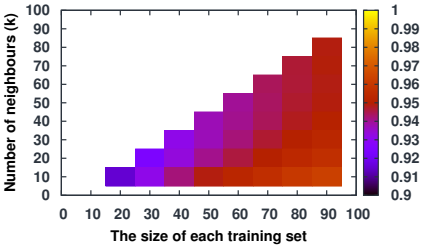
[2] Actually, squares of distances are considered to avoid square roots (it does not affect the result).



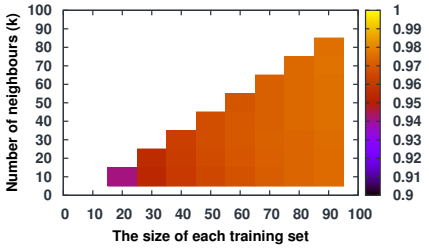
(a) Default, unweighted



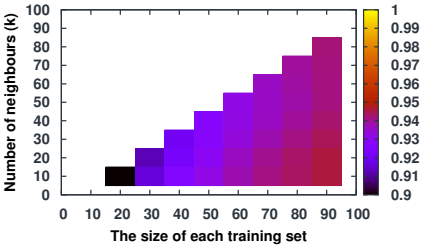
(b) Default, weighted



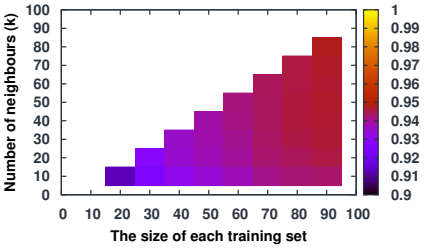
(c) Distant, unweighted



(d) Distant, weighted

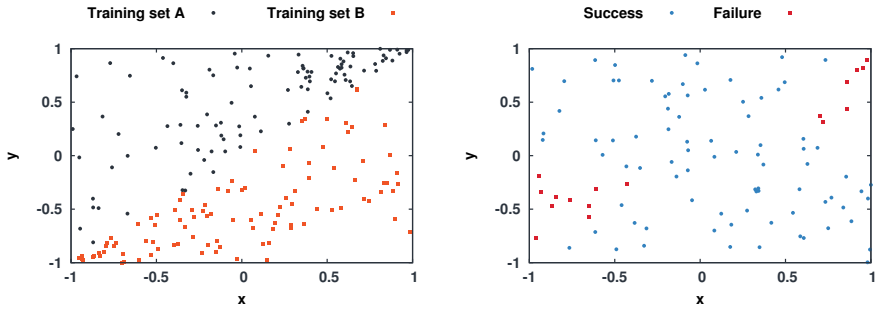


(e) Overlapped, unweighted



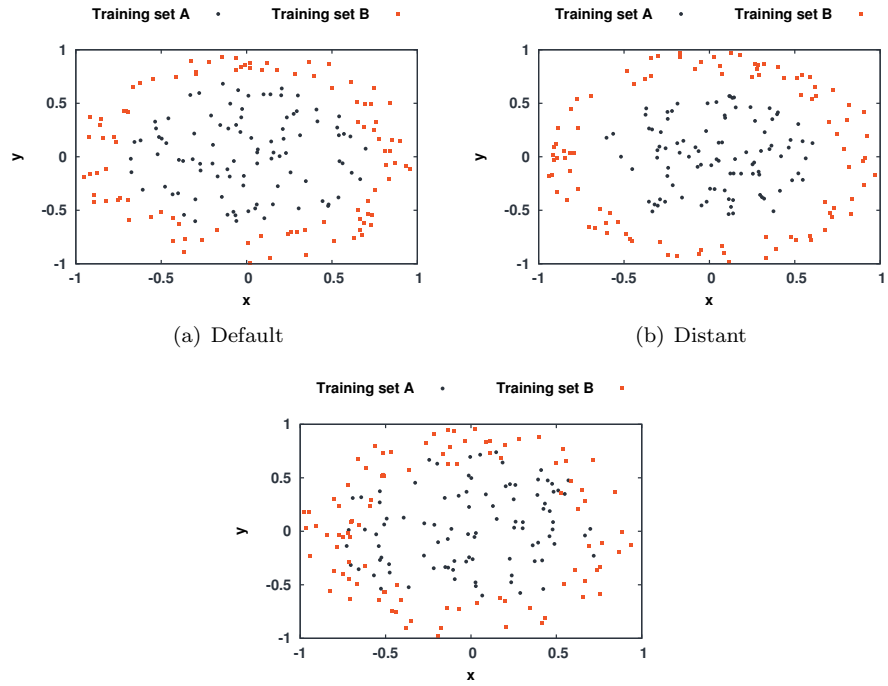
(f) Overlapped, weighted

Figure 3: kNN efficiency for points separable by $f(x) = x$.



(a) Training sets (b) Testing set

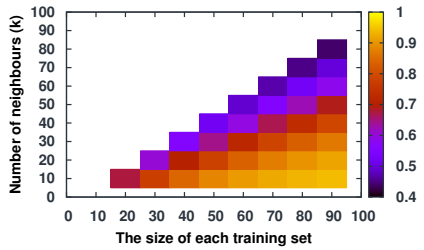
Figure 4: The example of the kNN classification using wrong training sets ($n = 100$, $k = 50$).



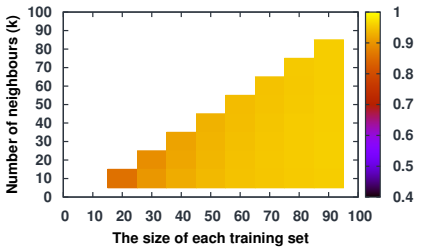
(a) Default (b) Distant

(c) Overlapped

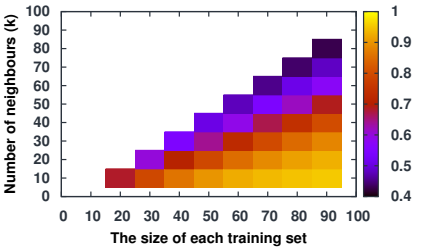
Figure 5: Training sets for inseparable points (inside / outside a circle).



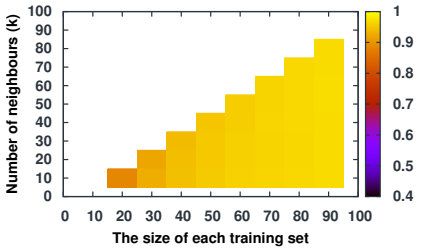
(a) Default, unweighted



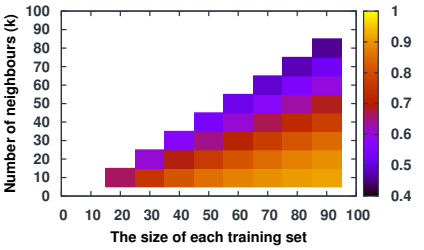
(b) Default, weighted



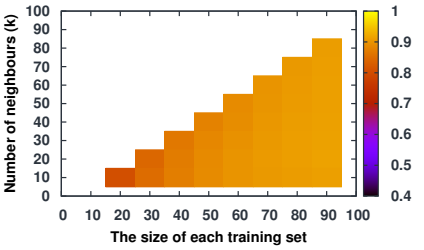
(c) Distant, unweighted



(d) Distant, weighted



(e) Overlapped, unweighted



(f) Overlapped, weighted

Figure 6: kNN efficiency for inseparable points.

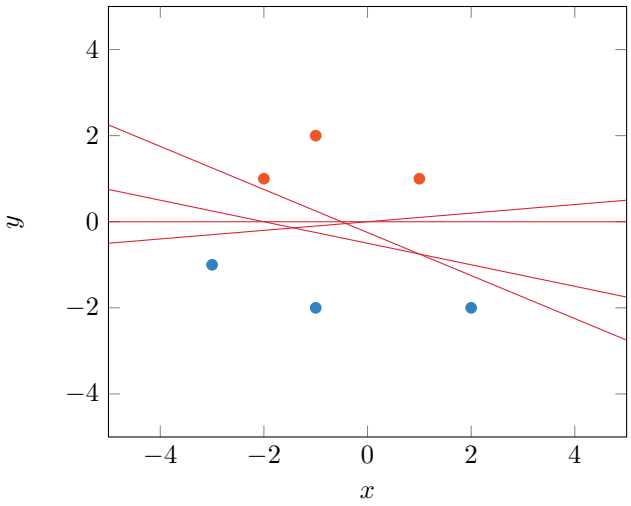


Figure 7: Examples of separation lines for two-dimensional data.

unweighted kNN requires now larger training sets to get an efficiency $\geq 90\%$. It is expected, though. The more complex problem the larger training set is necessary. Weighted kNN still works (surprisingly?) pretty well even for small training sets.

3 Support Vector Machine

Support Vector Machine is one those algorithms which sounds pretty easy until you actually start using it. It is a binary classifier, so it can only work with two classes at a time. It is an extension of perceptron algorithm (sometimes known as perceptron with optimal stability).

The perceptron algorithm looks for a line (or a hyperplane in general) which separates training sets. Thus, it works only for linearly separable classes. The hyperplane is found using online learning, by updating the hyperplane based on incorrectly classified training points. For linearly separable classes it guarantees convergence, but it does not control the final orientation of the hyperplane. As presented on Fig. 7, there is infinitely many hyperplanes (lines) like that.

One can expect that the best separation hyperplane should be possible as far as possible from both testing samples. Intuitively, the "real" boundary separating classes (which is unknown) is located far from the center of training sets. This way, one may expect better classification of unusual testing points, so better generalization of the problem.

SVM guarantees (for linearly separable classes) finding the maximum-margin hyperplane. It can be easily extend for "almost" linearly separable classes, like those presented on Fig. 2(c) (see Sec. **??**). For classes linearly inseparable, *kernel trick* is used to transform feature space to higher dimension, where classes are linearly separable and maximum-margin hyperplane can be found (see Sec. **??**).

The task is "easy" - find the separation hyperplane and classification is straightforward.

3.1 Linear SVM

Lets consider our training sets are given by:

$$\{(\vec{x}_i, y_i)\}_{i=1,\ldots,N} \tag{1}$$

where $\vec{x}_i = (x_{i1}, ..., x_{in}) \in \mathbb{R}^n$ are features vector in n-dimensional space, $y_i \in \{-1, 1\}$ defines membership to given class[3], N is the size of training sets. A hyperplane can be defined by its normal vector ($\vec{\omega} = (\omega_1, ..., \omega_n)$) and an absolute term ($\omega_0$):

$$\omega_0 + \underbrace{\omega_1 \cdot x_1 + ... + \omega_n \cdot x_n}_{\langle \vec{\omega}, \vec{x} \rangle} = 0 \tag{2}$$

The distance (d) between a point $\vec{x}$ and a hyperplane defined by $\vec{\omega}$ and ω_0 is given by the following formula:

$$d\left(\vec{x}, \vec{\omega}, \omega_0\right) = \frac{|\omega_0 + \langle \vec{\omega}, \vec{x} \rangle|}{||\vec{\omega}||} \tag{3}$$

where $||\vec{\omega}|| = \sqrt{\omega_1^2 + ... + \omega_n^2}$ is norm of the vector $\vec{\omega}$. Having that, the margin of separation (τ) for training set from Eq. 1 and a hyperplane as in Eq. 2 is defined as:

$$\tau(\vec{\omega}, \omega_0) = \min_{i=1,\ldots,N} \frac{y_i \cdot (\omega_0 + \langle \vec{\omega}, \vec{x}_i \rangle)}{||\vec{\omega}||} \tag{4}$$

so it is a distance between the hyperplane and the closest point. One should note, that absolute value from Eq. 3 disappears because of smart choice of y values. Points on the "positive" side of the hyperplane have $y = 1$, and those on the "negative" side have $y = -1$, which guarantees $\tau \geq 0$ (assuming all testing points are classified correctly).

SVM is looking for the optimal hyperplane, so the one with the maximum margin of separation. The optimization task to solve is:

$$\text{maximize} \quad \tau(\vec{\omega}, \omega_0) \tag{5a}$$

$$\text{subject to} \quad \forall_i \;\; y_i \cdot (\omega_0 + \langle \vec{\omega}, \vec{x}_i \rangle) \geq \tau(\vec{\omega}, \omega_0) \cdot ||\vec{\omega}|| \tag{5b}$$

Thus, maximize τ and make sure all points are on the right side of the hyperplane (with the distance not smaller than τ). Please note, that there are still infinitively many hyperplanes fulfill these conditions (as multiplying Eq. 2 by a constant does not change the position of the hyperplane). One can use an arbitrary bond to determine the hyperplane unambiguously. It is convenient to use the following bond:

$$\tau(\vec{\omega}, \omega_0) \cdot ||\vec{\omega}|| = 1 \tag{6}$$

[3] $\{-1, 1\}$ was chosen for convenience in further calculations.

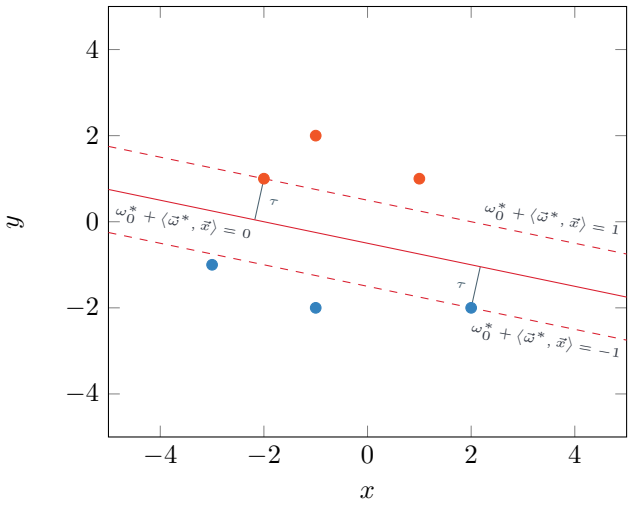


Figure 8: The example of maximum-margin hyperplane separating two training sets. *Note, it is just a demonstrative cartoon, not real calculations.*

With this bond, maximizing τ can be considered as minimizing the length of normal vector $||\vec{\omega}||$, so the optimization conditions from Eq. 5 can be rewritten as:

$$\text{minimize} \quad Q(\vec{\omega}) = \frac{1}{2}||\vec{\omega}||^2 \tag{7a}$$

$$\text{subject to} \quad \forall_i \;\; y_i \cdot (\omega_0 + \langle \vec{\omega}, \vec{x}_i \rangle) \geq 1 \tag{7b}$$

where $\frac{1}{2}$ in Eq. 7a is just for convenience in further calculations. For the same reasons, the square of $||\vec{\omega}||$ is considered. It does not affect the final result as $||\vec{\omega}||$ and $Q(\vec{\omega})$ have the minimum for the same $\vec{\omega}$.

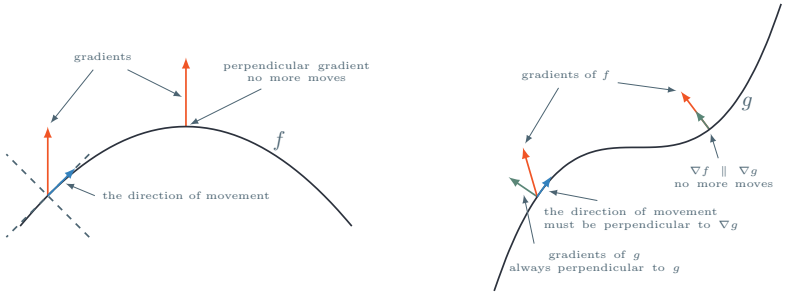
The example of maximum-margin hyperplane separating two training sets is presented on Fig. 8, where ω_0^* and $\vec{\omega}^*$ denote the solution of Eq. 7. Please note, this is just a demonstrative cartoon, not a real solution.

Eq. 7 is a quadratic programming problem. It is a kind of mathematical optimization which is extensively studied. There are many methods to solve this kind of problems. In the context of SVM Lagrange multipliers is commonly used. Sec. 3.2 is an introduction / reminder on Lagrange multipliers. In Sec. **??** Eq. 7 is solved using this method.

3.2 Lagrange multipliers

Lagrange multipliers is the method to optimize (find minimum or maximum) of the function $f(x_1, ..., x_n) : \mathbb{R}^n \to \mathbb{R}$ with given constraint[4] $g(x_1, ..., x_n) = 0$.

[4]In general, the constraint could be given by $g(x_1, ..., x_n) = c$, but it does not hurt to introduce $g'(x_1, ..., x_n) = g(x_1, ..., x_n) - c = 0$.



(a) Looking of the local minimum / maximum of the function f.

(b) Moving along the constraint g.

Figure 9: The illustration of constraint optimization.

The method is based on the fact that the gradient of the function f must be parallel to the gradient of the constraint g:

$$\nabla f(x_1, ..., x_n) = \lambda \nabla g(x_1, ..., x_n) \tag{8}$$

where λ is a Lagrange multiplier. Why Eq. 8 is true? Lets consider intuitive geometric explanation. If you want a strict proof grab a math book!

In general, the gradient ∇f gives the directions to head to minimize / maximize f. If going exactly in the direction of the gradient is impossible, but it is possible to go in the direction of a component of the gradient it still good (just increase / decrease is slower). If the gradient is perpendicular, there is no such component and the local minimum / maximum is found. Fig. 9(a) illustrates this.

If constraint $g = 0$ is given, the movement is restricted to fulfill the condition. In other words, only moves along g are possible. To keep g constant, only moves perpendicular to the gradient of g are allowed (otherwise increase or decrease of g would occur). This is demonstrated on Fig. 9(b). The direction of the movement is determined by ∇g (must be perpendicular) and ∇f. More precisely it is given by a component of ∇f perpendicular to ∇g. If ∇f is parallel to ∇g there is no such component and the local minimum / maximum with constraint is found.

For convenience, lets introduce Lagrangian defined as:

$$L(x_1, ..., x_n) = f(x_1, ..., x_n) - \lambda g(x_1, ..., x_n) \tag{9}$$

so the Eq. 8 can be rewritten as:

$$\nabla L(x_1, ..., x_n) = 0 \tag{10}$$

Lets go back to SVM optimization problem defined by Eq. 7. The number of constraints there is equal to the size of the training set. Fortunately, the generalization of the Lagrangian (Eq. 9) is straightforward:

$$L(x_1, ..., x_n) = f(x_1, ..., x_n) - \sum_i \lambda_i g_i(x_1, ..., x_n) \tag{11}$$

where λ_i is Lagrange multiplier for the constraint g_i. The reasoning stays the same, but the movement of f is now restricted by many constrains:

$$\begin{aligned} \nabla f(x_1, ..., x_n) &= \lambda_1 \nabla g_1(x_1, ..., x_n) \\ \nabla f(x_1, ..., x_n) &= \lambda_2 \nabla g_2(x_1, ..., x_n) \\ &... \\ \nabla f(x_1, ..., x_n) &= \lambda_N \nabla g_N(x_1, ..., x_n) \end{aligned}$$

There is still one piece missing. The optimization problem given by Eq. 7 use inequality constraints. The generalization of Lagrange multipliers method to inequality constrains is given by Karush–Kuhn–Tucker conditions, described in Sec. 3.3.

3.3 Karush–Kuhn–Tucker conditions