# CS261: A Second Course in Algorithms
# Lecture #7: Linear Programming: Introduction and Applications*

Tim Roughgarden†

January 26, 2016

## 1 Preamble

With this lecture we commence the second part of the course, on *linear programming*, with an emphasis on applications on duality theory.[1] We'll spend a fair amount of quality time with linear programs for two reasons.

First, linear programming is very useful algorithmically, both for proving theorems and for solving real-world problems.

> Linear programming is a remarkable sweet spot between power/generality and computational efficiency.

For example, all of the problems studied in previous lectures can be viewed as special cases of linear programming, and there are also zillions of other examples. Despite this generality, linear programs can be solved efficiently, both in theory (meaning in polynomial time) and in practice (with input sizes up into the millions).

Even when a computational problem that you care about does not reduce directly to solving a linear program, linear programming is an extremely helpful subroutine to have in your pocket. For example, in the fourth and last part of the course, we'll design approximation algorithms for $NP$-hard problems that use linear programming in the algorithm and/or analysis. In practice, probably most of the cycles spent on solving linear programs is in service of solving integer programs (which are generally $NP$-hard). State-of-the-art

†Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: tim@cs.stanford.edu.

[1]The term "programming" here is not meant in the same sense as computer programming (linear programming pre-dates modern computers). It's in the same spirit as "television programming," meaning assembling a scheduled of planned activities. (See also "dynamic programming".)

algorithms for the latter problem invoke a linear programming solver over and over again to make consistent progress.

Second, linear programming is conceptually useful —- understanding it, and especially LP duality, gives you the "right way" to think about a host of different problems in a simple and consistent way. For example, the optimality conditions we've studied in past lectures (like the max-flow/min-cut theorem and Hall's theorem) can be viewed as special cases of linear programming duality. LP duality is more or less the ultimate answer to the question "how do we know when we're done?" As such, it's extremely useful for proving that an algorithm is correct (or approximately correct).

We'll talk about both these aspects of linear programming at length.

## 2 How to Think About Linear Programming

### 2.1 Comparison to Systems of Linear Equations

Once upon a time, in some course you may have forgotten, you learned about linear systems of equations. Such a system consists of $m$ linear equations in real-valued variables $x_1, \ldots, x_n$:

$$\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\
&\vdots \\
a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= b_m.
\end{aligned}$$

The $a_{ij}$'s and the $b_i$'s are given; the goal is to check whether or not there are values for the $x_j$'s such that all $m$ constraints are satisfied. You learned at some point that this problem can be solved efficiently, for example by Gaussian elimination. By "solved" we mean that the algorithm returns a feasible solution, or correctly reports that no feasible solution exists.

Here's an issue, though: what about inequalities? For example, recall the maximum flow problem. There are conservation constraints, which are equations and hence OK. But the capacity constraints are fundamentally inequalities. (There is also the constraint that flow values should be nonnegative.) Inequalities are part of the problem description of many other problems that we'd like to solve. The point of linear programming is to solve systems of linear equations *and inequalities*. Moreover, when there are multiple feasible solutions, we would like to compute the "best" one.

### 2.2 Ingredients of a Linear Program

There is a convenient and flexible language for specifying linear programs, and we'll get lots of practice using it during this lecture. Sometimes it's easy to translate a computational problem into this language, sometimes it takes some tricks (we'll see examples of both).

To specify a linear program, you need to declare what's allowed and what you want.

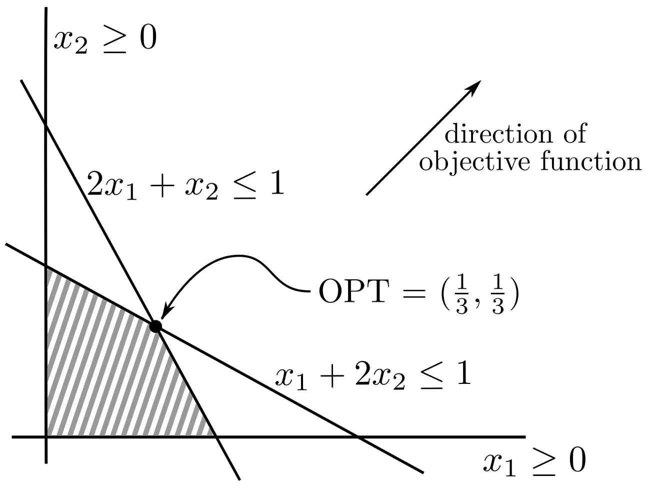
**Ingredients of a Linear Program**

1. *Decision variables* $x_1, \ldots, x_n \in \mathbb{R}$.
2. *Linear constraints*, each of the form
$$\sum_{j=1}^{n} a_j x_j \quad (*) \quad b_i,$$
where (*) could be $\leq$, $\geq$, or $=$.
3. A *linear objective function*, of the form
$$\max \sum_{j=1}^{n} c_j x_j$$
or
$$\min \sum_{j=1}^{n} c_j x_j.$$

Several comments. First, the $a_{ij}$'s, $b_i$'s, and $c_j$'s are *constants*, meaning they are part of the input, numbers hard-wired into the linear program (like 5, -1, 10, etc.). The $x_j$'s are free, and it is the job of a linear programming algorithm to figure out the best values for them. Second, when specifying constraints, there is no need to make use of both "$\leq$" and "$\geq$"inequalities — one can be transformed into the other just by multiplying all the coefficients by -1 (the $a_{ij}$'s and $b_i$'s are allowed to be positive or negative). Similarly, equality constraints are superfluous, in that the constraint that a quantity equals $b_i$ is equivalent to the pair of inequality constraints stating that the quantity is both at least $b_i$ and at most $b_i$. Finally, there is also no difference between the "min" and "max" cases for the objective function — one is easily converted into the other just by multiplying all the $c_j$'s by -1 (the $c_j$'s are allowed to be positive or negative).

So what's not allowed in a linear program? Terms like $x_j^2$, $x_j x_k$, $\log(1 + x_j)$, etc. So whenever a decision variable appears in an expression, it is alone, possibly multiplied by a constant (and then summed with other such terms). While these linearity requirements may seem restrictive, we'll see that many real-world problems can be formulated as or well approximated by linear programs.

## 2.3 A Simple Example



$x_2 \geq 0$
direction of objective function
$2x_1 + x_2 \leq 1$
$\text{OPT} = (\frac{1}{3}, \frac{1}{3})$
$x_1 + 2x_2 \leq 1$
$x_1 \geq 0$

Figure 1: a toy example of linear program.

To make linear programs more concrete and develop your geometric intuition about them, let's look at a toy example. (Many "real" examples of linear programs are coming shortly.) Suppose there are two decision variables $x_1$ and $x_2$ — so we can visualize solutions as points $(x_1, x_2)$ in the plane. See Figure 2.3. Let's consider the (linear) objective function of maximizing the sum of the decision variables:

$$\max x_1 + x_2.$$

We'll look at four (linear) constraints:

$$\begin{array}{rcl} x_1 & \geq & 0 \\ x_2 & \geq & 0 \\ 2x_1 + x_2 & \leq & 1 \\ x_1 + 2x_2 & \leq & 1. \end{array}$$

The first two inequalities restrict feasible solutions to the non-negative quadrant of the plane. The second two inequalities further restrict feasible solutions to lie in the shaded region depicted in Figure 2.3. Geometrically, the objective function asks for the feasible point furthest in the direction of the coefficient vector $(1, 1)$ — the "most northeastern" feasible point. Put differently, the level sets of the objective function are parallel lines running northwest to southeast.[2] Eyeballing the feasible region, this point is $(\frac{1}{3}, \frac{1}{3})$, for an optimal objective function value of $\frac{2}{3}$. This is the "last point of intersection" between a level set of the objective function and the feasible region (as one sweeps from southwest to northeast).

[2]Recall that a *level set* of a function $g$ has the form $\{\mathbf{x} : g(\mathbf{x}) = c\}$, for some constant $c$. That is, all points in a level set have equal objective function value.

## 2.4 Geometric Intuition

While it's always dangerous to extrapolate from two or three dimensions to an arbitrary number, the geometric intuition above remains valid for general linear programs, with an arbitrary number of dimensions (i.e., decision variables) and constraints. Even though we can't draw pictures when there are many dimensions, the relevant algebra carries over without any difficulties. Specifically:

1. A linear constraint in $n$ dimensions corresponds to a halfspace in $\mathbb{R}^n$. Thus a feasible region is an intersection of halfspaces, the higher-dimensional analog of a polygon.[3]

2. The level sets of the objective function are parallel $(n-1)$-dimensional hyperplanes in $\mathbb{R}^n$, each orthogonal to the coefficient vector $\mathbf{c}$ of the objective function.

3. The optimal solution is the feasible point furthest in the direction of $\mathbf{c}$ (for a maximization problem) or $-\mathbf{c}$ (for a minimization problem). Equivalently, it is the last point of intersection (traveling in the direction $\mathbf{c}$ or $-\mathbf{c}$) of a level set of the objective function and the feasible region.

4. When there is a unique optimal solution, it is a vertex (i.e., "corner") of the feasible region.

There are a few edge cases which can occur but are not especially important in CS261.

1. There might be no feasible solutions at all. For example, if we add the constraint $x_1 + x_2 \geq 1$ to our toy example, then there are no longer any feasible solutions. Linear programming algorithms correctly detect when this case occurs.

2. The optimal objective function value is unbounded ($+\infty$ for a maximization problem, $-\infty$ for a minimization problem). Note a necessary but not sufficient condition for this case is that the feasible region is unbounded. For example, if we dropped the constraints $2x_1 + x_2 \leq 1$ and $x_1 + 2x_2 \leq 1$ from our toy example, then it would have unbounded objective function value. Again, linear programming algorithms correctly detect when this case occurs.

3. The optimal solution need not be unique, as a "side" of the feasible region might be parallel to the levels sets of the objective function. Whenever the feasible region is bounded, however, there always exists an optimal solution that is a vertex of the feasible region.[4]

[3] A finite intersection of halfspaces is also called a "polyhedron;" in the common special case where the feasible region is bounded, it is called a "polytope."

[4] There are some annoying edge cases for unbounded feasible regions, for example the linear program $\max(x_1 + x_2)$ subject to $x_1 + x_2 = 1$.

## 3 Some Applications of Linear Programming

Zillions of problems reduce to linear programming. It would take an entire course to cover even just its most famous applications. Some of these applications are conceptually a bit boring but still very important — as early as the 1940s, the military was using linear programming to figure out the most efficient way to ship supplies from factories to where they were needed.[5] Several central problems in computer science reduce to linear programming, and we describe some of these in detail in this section. Throughout, keep in mind that all of these linear programs can be solved efficiently, both in theory and in practice. We'll say more about algorithms for linear programming in a later lecture.

### 3.1 Maximum Flow

If we return to the definition of the maximum flow problem in Lecture #1, we see that it translates quite directly to a linear program.

1. *Decision variables:* what are we try to solve for? A flow, of course, Specifically, the amount $f_e$ of flow on each edge $e$. So our variables are just $\{f_e\}_{e \in E}$.

2. *Constraints:* Recall we have conservation constraints and capacity constraints. We can write the former as
$$\underbrace{\sum_{e \in \delta^-(v)} f_e}_{\text{flow in}} - \underbrace{\sum_{e \in \delta^-(v)} f_e}_{\text{flow out}} = 0$$
for every vertex $v \neq s, t$.[6] We can write the latter as
$$f_e \leq u_e$$
for every edge $e \in E$. Since decision variables of linear programs are by default allowed to take on arbitrary real values (positive or negative), we also need to remember to add nonnegativity constraints:
$$f_e \geq 0$$
for every edge $e \in E$. Observe that every one of these $2m + n - 2$ constraints (where $m = |E|$ and $n = |V|$) is linear — each decision variable $f_e$ only appears by itself (with a coefficient of 1 or -1).

3. *Objective function:* We just copy the same one we used in Lecture #1:
$$\max \sum_{e \in \delta^+(s)} f_e.$$
Note that this is again a linear function.

[5] Note this is well before computer science was field; for example, Stanford's Computer Science Department was founded only in 1965.

[6] Recall that $\delta^-$ and $\delta^+$ denote the edges incoming to and outgoing from $v$, respectively.

## 3.2 Minimum-Cost Flow

In Lecture #6 we introduced the minimum-cost flow problem. Extending specialized algorithms for maximum flow to generalized algorithms takes non-trivial work (see Problem Set #2 for starters). If we're just using linear programming, however, the generalization is immediate.[7] The main change is in the objective function. As defined last lecture, it is simply

$$\min \sum_{e \in E} c_e f_e,$$

where $c_e$ is the cost of edge $e$. Since the $c_e$'s are fixed numbers (i.e., part of the input), this is a linear objective function.

For the version of the minimum-cost flow problem defined last lecture, we should also add the constraint

$$\sum_{e \in \delta^+(s)} f_e = d,$$

where $d$ is the target flow value. (One can also add the analogous constraint for $t$, but this is already implied by the other constraints.)

To further highlight how flexible linear programs can be, suppose we want to impose a lower bound $\ell_e$ (other than 0) on the amount of flow on each edge $e$, in addition to the usual upper bound $u_e$. This is trivial to accommodate in our linear program — just replace "$f_e \geq 0$" by $f_e \geq \ell_e$.[8]

## 3.3 Fitting a Line

We now consider two less obvious applications of linear programming, to basic problems in machine learning. We first consider the problem of fitting a line to data points (i.e., linear regression), perhaps the simplest non-trivial machine learning problem.

Formally, the input consists of $m$ data points $\mathbf{p}^1, \ldots, \mathbf{p}^m \in \mathbb{R}^d$, each with $d$ real-valued "features" (i.e., coordinates).[9] For example, perhaps $d = 3$, and each data point corresponds to a 3rd-grader, listing the household income, number of owned books, and number of years of parental education. Also part of the input is a "label" $\ell_i \in \mathbb{R}$ for each point $\mathbf{p}^i$.[10] For example, $\ell_i$ could be the score earned by the 3rd-grader in question on a standardized test. We reiterate that the $\mathbf{p}^i$'s and $\ell_i$'s are fixed (part of the input), not decision variables.

[7]While linear programming is a reasonable way to solve the maximum flow and minimum-cost flow problems, especially if the goal is to have a "quick and dirty" solution, but the best specialized algorithms for these problems are generally faster.

[8]If you prefer to use flow algorithms, there is a simple reduction from this problem to the special case with $\ell_e = 0$ for all $e \in E$ (do you see it?).

[9]Feel free to take $d = 1$ throughout the rest of the lecture, which is already a practically relevant and computationally interesting case.

[10]This is a canonical "supervised learning" problem, meaning that the algorithm is provided with labeled data.

Informally, the goal is to expresses the $\ell_i$ as well as possible as a linear function of the $\mathbf{p}_i$'s. That is, the goal is to compute a linear function $h : \mathbb{R}^d \to \mathbb{R}$ such that $h(\mathbf{p}^i) \approx \ell_i$ for every data point $i$.

The two most common motivations for computing a "best-fit" linear function are prediction and data analysis. In the first scenario, one uses labeled data to identify a linear function $h$ that, at least for these data points, does a good job of predicting the label $\ell_i$ from the feature values $\mathbf{p}^i$. The hope is that this linear function "generalizes," meaning that it also makes accurate predictions for other data points for which the label is not already known. There is a lot of beautiful and useful theory in statistics and machine learning about when one can and cannot expect a hypothesis to generalize, which you'll learn about if you take courses in those areas. In the second scenario, the goal is to understand the relationship between each feature of the data points and the labels, and also the relationships between the different features. As a simple example, it's clearly interesting to know when one of the $d$ features is much more strongly correlated with the label $\ell^i$ than any of the others.

We now show that computing the best line, for one definition of "best," reduces to linear programming. Recall that every linear function $h : \mathbb{R}^d \to \mathbb{R}$ has the form

$$h(\mathbf{z}) = \sum_{j=1}^{d} a_j z_j + b$$

for some coefficients $a_1, \ldots, a_d$ and intercept $b$. (This is one of several equivalent definitions of a linear function.[11] So it's natural to take $a_1, \ldots, a_d, b$ as our decision variables.

What's our objective function? Clearly if the data points are colinear we want to compute the line that passes through all of them. But this will never happen, so we must compromise between how well we approximate different points.

For a given choice of $a_1, \ldots, a_d, b$, define the error on point $i$ as

$$E_i(\mathbf{a}, b) = \left| \underbrace{\left( \sum_{j-1}^{d} a_j p_j^i - b \right)}_{\text{prediction}} - \underbrace{\ell^i}_{\text{"ground truth"}} \right|. \tag{1}$$

Geometrically, when $d = 1$, we can think of each $(\mathbf{p}^i, \ell^i)$ as a point in the plane and (1) is just the vertical distance between this point and the computed line.

In this lecture, we consider the objective function of minimizing the sum of errors:

$$\min_{\mathbf{a},b} \sum_{i=1}^{m} E_i(\mathbf{a}, b). \tag{2}$$

This is not the most common objective for linear regression; more standard is minimizing the squared error $\sum_{i=1}^{m} E_i^2(\mathbf{a}, b)$. While our motivation for choosing (2) is primarily pedagogical,

[11] Sometimes people use "linear function" to mean the special case where $b = 0$, and "affine function" for the case of arbitrary $b$.

this objective is reasonable and is sometimes used in practice. The advantage over squared error is that it is more robust to outliers. Squaring the error of an outlier makes it a squeakier wheel. That is, a stray point (e.g., a faulty sensor or data entry error) will influence the line chosen under (2) less that it would with the squared error objective (Figure 2).[12]

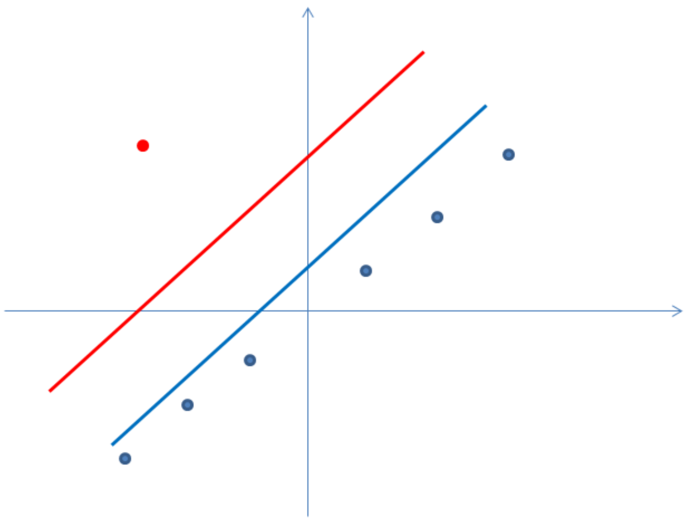


Figure 2: When there exists an outlier (red point), using the objective function defined in (2) causes the best-fit line not to "stray" as far away from the non-outliers (blue line) as when using the squared error objective (red line), because the squared error objective would penalize more greatly when the chosen line is far from the outlier.

Consider the problem of choosing $\mathbf{a}, b$ to minimize (2). (Since the $a_j$'s and $b$ can be anything, there are no constraints.) The problem: *this is not a linear program.* The source of nonlinearity is the absolute value sign $|\cdot|$ in (1). Happily, in this case and many others, absolute values can be made linear with a simple trick.

The trick is to introduce extra variables $e_1, \ldots, e_m$, one per data point. The intent is for $e_i$ to take on the value $E_i(\mathbf{a}, b)$. Motivated by the identify $|x| = \max\{x, -x\}$, we add two constraints for each data point:

$$e_i \geq \left( \sum_{j=1}^{d} a_j p_j^i - b \right) - \ell^i \tag{3}$$

and

$$e_i \geq - \left[ \left( \sum_{j=1}^{d} a_j p_j^i - b \right) - \ell^i \right]. \tag{4}$$

[12]Squared error can be minimized efficiently using an extension of linear programming known as *convex programming.* (For the present "ordinary least squares" version of the problem, it can even be solved analytically, in closed form.) We may discuss convex programming in a future lecture.

We change the objective function to

$$\min \sum_{i=1}^{m} e_i. \tag{5}$$

Note that optimizing (5) subject to all constraints of the form (3) and (4) is a linear program, with decision variables $e_1, \ldots, e_m, a_1, \ldots, a_d, b$.

The key point is: at an optimal solution to this linear program, it must be that $e_i = E_i(\mathbf{a}, b)$ for every data point $i$. Feasibility of the solution already implies that $e_i \geq E_i(\mathbf{a}, b)$ for every $i$. And if $e_i > E_i(\mathbf{a}, b)$ for some $i$, then we can decrease $e_i$ slightly, so that (3) and (4) still hold, to obtain a superior feasible solution. We conclude that an optimal solution to this linear program represents the line minimizing the sum of errors (2).

## 3.4 Computing a Linear Classifier

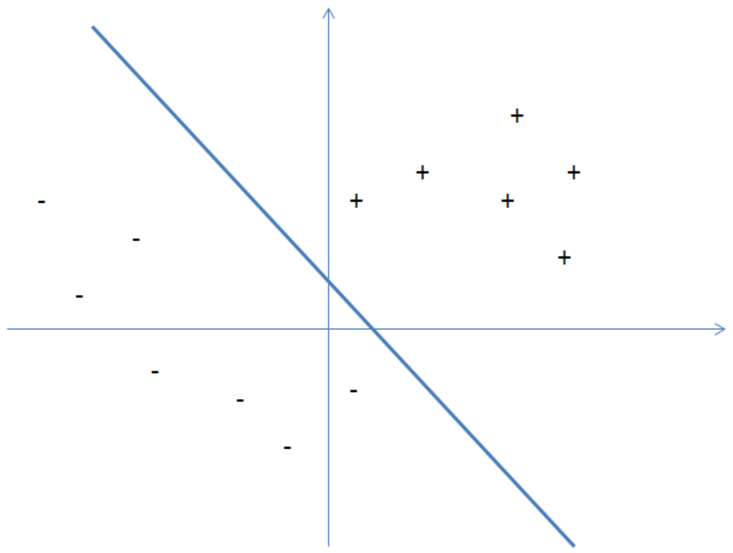


Figure 3: We want to find a linear function that separates the positive points (plus signs) from the negative points (minus signs)

Next we consider a second fundamental problem in machine learning, that of learning a linear classifier.[13] While in Section 3.3 we sought a real-valued function (from $\mathbb{R}^d$ to $\mathbb{R}$), here we're looking for a binary function (from $\mathbb{R}^d$ to $\{0, 1\}$). For example, data points could represent images, and we want to know which ones contain a cat and which ones don't.

Formally, the input consists of $m$ "positive" data points $\mathbf{p}^1, \ldots, \mathbf{p}^m \in \mathbb{R}^d$ and $m'$ "negative" data points $\mathbf{q}^1, \ldots, \mathbf{q}^{m'}$. In the terminology of the previous section, all of the labels

[13]Also called halfspaces, perceptrons, linear threshold functions, etc.

are "1" or "0," and we have partitioned the data accordingly. (So this is again a supervised learning problem.)

The goal is to compute a linear function $h(\mathbf{z}) = \sum_{j=1} a_j z_j + b$ (from $\mathbb{R}^d$ to $\mathbb{R}$) such that

$$h(\mathbf{p}^i) > 0 \tag{6}$$

for all positive points and

$$h(\mathbf{q}^i) < 0 \tag{7}$$

for all negative points. Geometrically, we are looking for a hyperplane in $\mathbb{R}^d$ such all positive points are on one side and all negative points on the other; the coefficients $\mathbf{a}$ specify the normal vector of the hyperplane and the intercept $b$ specifies its shift. See Figure 3. Such a hyperplane can be used for predicting the labels of other, unlabeled points (check which side of the hyperplane it is on and predict that it is positive or negative, accordingly). If there is no such hyperplane, an algorithm should correctly report this fact.

This problem almost looks like a linear program by definition. The only issue is that the constraints (6) and (7) are strict inequalities, which are not allowed in linear programs. Again, the simple trick of adding an extra decision variable solves the problem. The new decision variable $\delta$ represents the "margin" by which the hyperplane satisfies (6) and (7). So we

$$\max \delta$$

subject to

$$\sum_{j=1}^{d} a_j p_j^i + b - \delta \geq 0 \qquad \text{for all positive points } \mathbf{p}^i$$

$$\sum_{j=1}^{d} a_j q_j^i + b + \delta \leq 0 \qquad \text{for all negative points } \mathbf{q}^i,$$

which is a linear program with decision variables $\delta, a_1, \ldots, a_d, b$. If the optimal solution to this linear program has strictly positive objective function value, then the values of the variables $a_1, \ldots, a_d, b$ define the desired separating hyperplane. If not, then there is no such hyperplane. We conclude that computing a linear classifier reduces to linear programming.

## 3.5 Extension: Minimizing Hinge Loss

There is an obvious issue with the problem setup in Section 3.4: what if the data set is not as nice as the picture in Figure 3, and there is no separating hyperplane? This is usually the case in practice, for example if the data is noisy (as it always is). Even if there's no perfect hyperplane, we'd still like to compute something that we can use to predict the labels of unlabeled points.

We outline two ways to extend the linear programming approach in Section 3.4 to handle non-separable data.[14] The first idea is to compute the hyperplane that minimizes some notion

[14]In practice, these two approaches are often combined.

of "classification error." After all, this is what we did in Section 3.3, where we computed the line minimizing the sum of the errors.

Probably the most natural plan would be to compute the hyperplane that puts the fewest number of points on the wrong side of the hyperplane — to minimize the number of inequalities of the form (6) or (7) that are violated. Unfortunately, this is an $NP$-hard problem, and one typically uses notions of error that are more computationally tractable. Here, we'll discuss the widely used notion of *hinge loss*.

Let's say that in a perfect world, we would like a linear function $h$ such that

$$h(\mathbf{p}^i) \geq 1 \tag{8}$$

for all positive points $\mathbf{p}^i$ and

$$h(\mathbf{q}^i) \leq -1 \tag{9}$$

for all negative points $\mathbf{q}^i$; the "1" here is somewhat arbitrary, but we need to pick some constant for the purposes of normalization. The *hinge loss* incurred by a linear function $h$ on a point is just the extent to which the corresponding inequality (8) or (9) fails to hold. For a positive point $\mathbf{p}^i$, this is $\max\{1-h(\mathbf{p}^i),0\}$; for a negative point $\mathbf{q}^i$, this is $\max\{1+h(\mathbf{p}^i),0\}$. Note that taking the maximum with zero ensures that we don't reward a linear function for classifying a point "extra-correctly." Geometrically, when $d=1$, the hinge loss is the vertical distance that a data point would have to travel to be on the correct side of the hyperplane, with a "buffer" of 1 between the point and the hyperplane.

Computing the linear function that minimizes the total hinge loss can be formulated as a linear program. While hinge loss is not linear, it is just the maximum of two linear functions. So by introducing one extra variable and two extra constraints per data point, just like in Section 3.3, we obtain the linear program

$$\min \sum_{i=1}^{m} e_i$$

subject to:

$$e_i \geq 1 - \left(\sum_{j=1}^{d} a_j p^i_j + b\right) \qquad \text{for every positive point } \mathbf{p}^i$$

$$e_i \geq 1 + \left(\sum_{j=1}^{d} a_j q^i_j + b\right) \qquad \text{for every negative point } \mathbf{q}^i$$

$$e_i \geq 0 \qquad \text{for every point}$$

in the decision variables $e_1,\ldots,e_m,a_1,\ldots,a_d,b$.

## 3.6 Extension: Increasing the Dimension

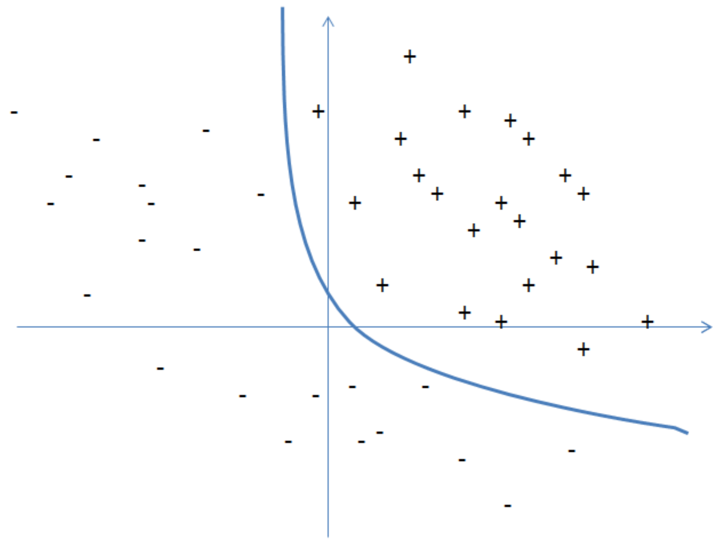


Figure 4: The points are not linearly separable, but they can be separated by a quadratic line.

A second approach to dealing with non-linearly-separable data is to use nonlinear boundaries. E.g., in Figure 4, the positive and negative points cannot be separated perfectly by any line, but they can be separated by a relatively simple boundary (e.g., of a quadratic function). But how we can allow nonlinear boundaries while retaining the computationally tractability of our previous solutions?

The key idea is to generate extra features (i.e., dimensions) for each data point. That is, for some dimension $d' \geq d$ and some function $\varphi : \mathbb{R}^d \to \mathbb{R}^{d'}$, we map each $\mathbf{p}^i$ to $\varphi(\mathbf{p}^i)$ and each $\mathbf{q}_i$ to $\varphi(\mathbf{q}^i)$. We'll then try to separate the images of these points in $d'$-dimensional space using a linear function.[15]

A concrete example of such a function $\varphi$ is the map

$$(z_1, \ldots, z_d) \mapsto (z_1, \ldots, z_d, z_1^2, \ldots, z_d^2, z_1 z_2, z_1 z_3, \ldots, z_{d-1} z_d); \tag{10}$$

that is, each data point is expanded with all of the pairwise products of its features. This map is interesting even when $d = 1$:

$$z \mapsto (z, z^2). \tag{11}$$

Our goal is now to compute a linear function in the expanded space, meaning coefficients

[15]This is the basic idea behind "support vector machines;" see CS229 for much more on the topic.

$a_1, \ldots, a_{d'}$ and an intercept $b$, that separates the positive and negative points:

$$\sum_{i=1}^{d'} a_j \cdot \varphi(\mathbf{p}^i)_j + b > 0 \tag{12}$$

for all positive points and

$$\sum_{i=1}^{d'} a_j \cdot \varphi(\mathbf{q}^i)_j + b < 0 \tag{13}$$

for all negative points. Note that if the new feature set includes all of the original features, as in (10), then every hyperplane in the original $d$-dimensional space remains available in the expanded space (just set $a_{d+1}, a_{d+2}, \ldots, a_{d'} = 0$). But there are also many new options, and hence it is more likely that there is way to perfectly separate the (images under $\varphi$ of the) data points. For example, even with $d = 1$ and the map (11), linear functions in the expanded space have the form $h(z) = a_1 z^2 + a_2 z + b$, which is a quadratic function in the original space.

We can think of the map $\varphi$ as being applied in a preprocessing step. Then, the resulting problem of meeting all the constraints (12) and (13) is exactly the problem that we already solved in Section 3.4. The resulting linear program has decision variables $\delta, a_1, \ldots, a_{d'}, b$ ($d' + 2$ in all, up from $d + 2$ in the original space).[16]

[16]The magic of support vector machines is that, for many maps $\varphi$ including (10) and (11), and for many methods of computing a separating hyperplane, the computation required scales only with the original dimension $d$, even if the expanded dimension $d'$ is radically larger. This is known as the "kernel trick;" see CS229 for more details.