Linear Models

Definition

A #linear-model is a model that assumes the data is #linearly-separable, i.e., the data can be separated either by a line [1] or by a hyperplane.

---

1. in 2D space ↩

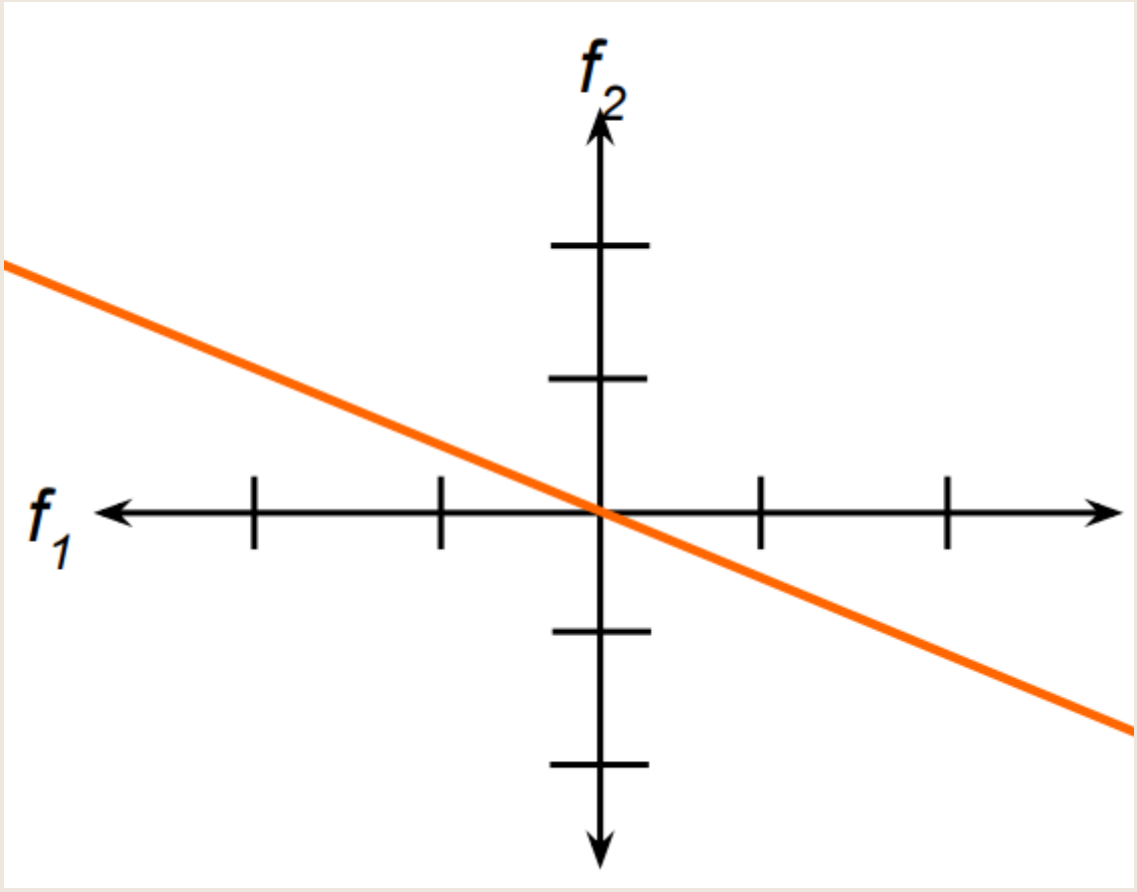
Defining a line

Any pair of values (w_1, w_2) defines a line through the origin.

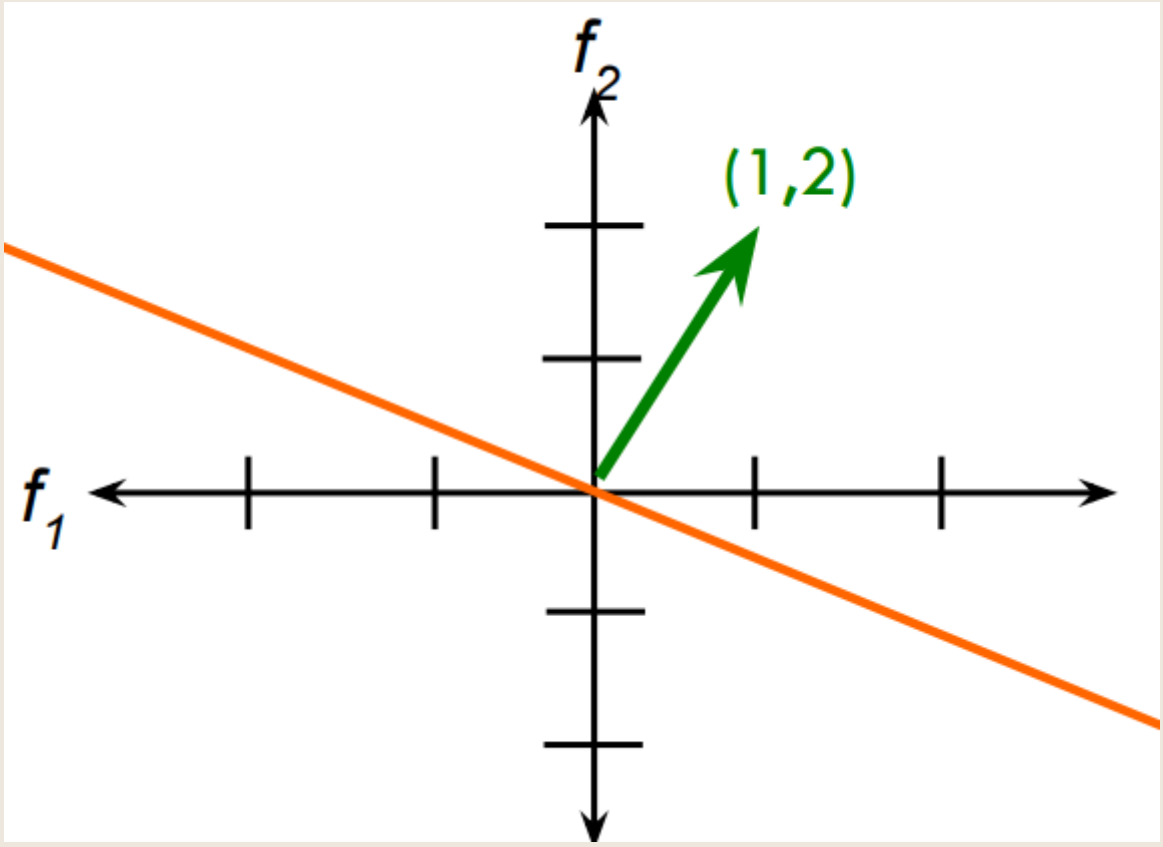
Example

$0 = w_1 f_1 + w_2 f_2$

If we choose $w_1 = 1$ and $w_2 = 2$, we get the points $(-2, 1), (-1, 0.5), (0, 0), (1, -0.5)$ etc lying on the line.



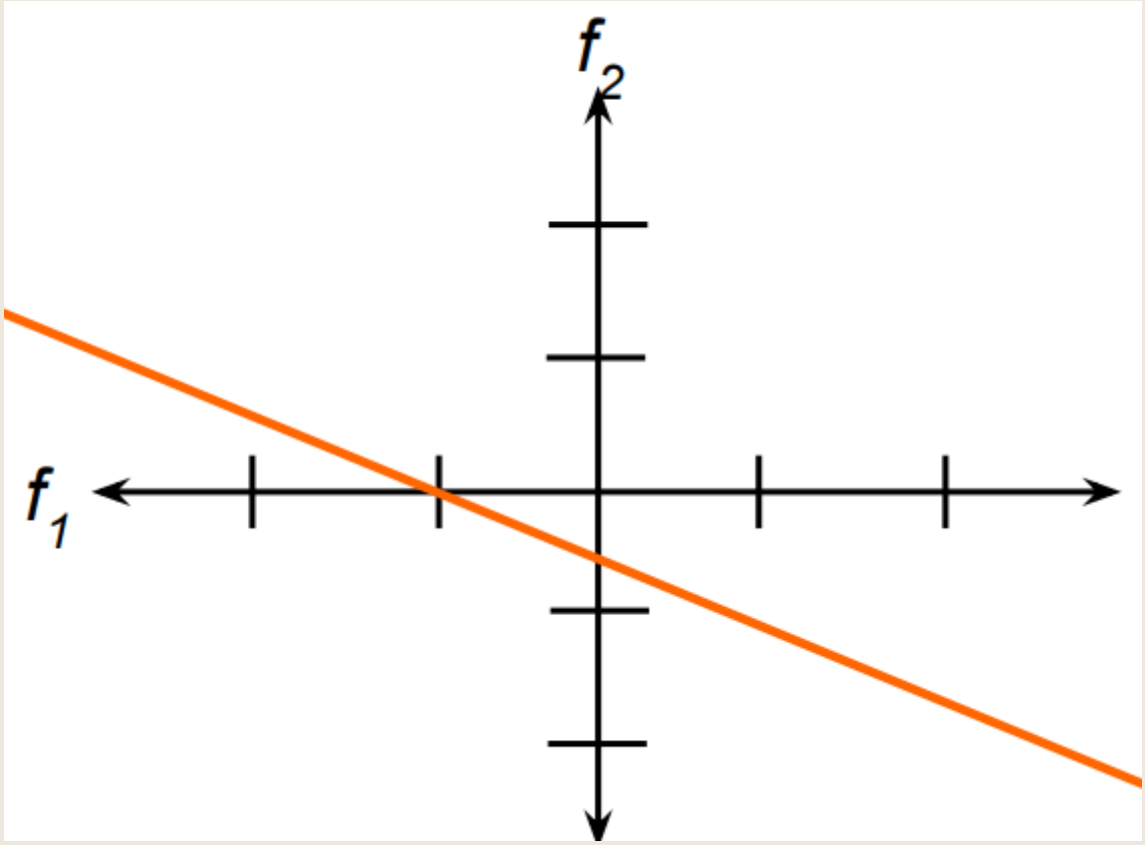
This line can also be viewed as the line perpendicular to the *weight vector* $w = (1, 2)$



We can also intuitively move a line off the origin by using $a = w_1 f_1 + w_2 f_2$ for some a

Example

$a = -1$ and $w = (1, 2) \therefore -1 = 1f_1 + 2f_2$. We get the points $(-2, 0.5), (-1, 0), (0, -0.5), (1, -1)$ etc meaning the line now intersects the f_2 axis at -1:

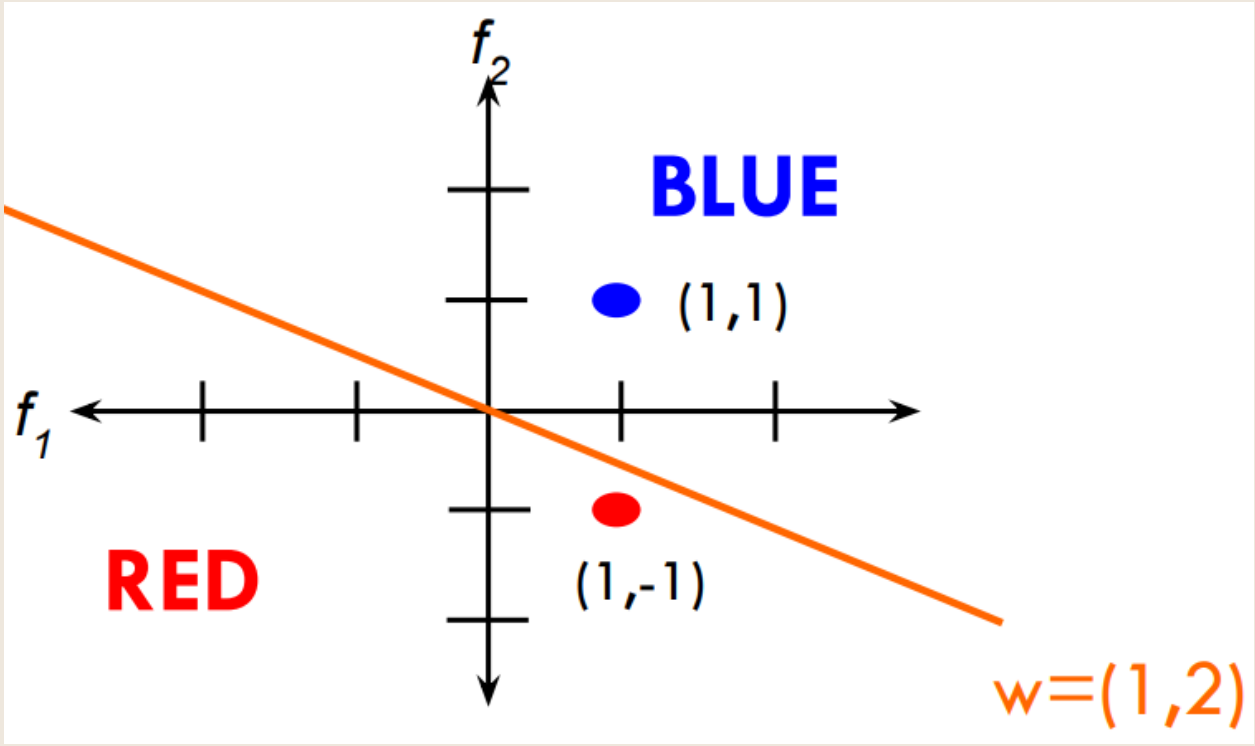


Classifying with a line

Given a point, we can classify it using our weight vector.

Example

For the blue point $(1, 1)$, $1 * 1 + 2 * 1 = 3$.
For the red point $(1, -1)$, $1 * 1 + 2 * -1 = -1$.
We can see that the sign indicates which side of the line the point lies on.



#Dimensionality

A linear model in n-dimensional space[1] is defined by $n + 1$ weights, meaning that in two dimensions we have a line $(0 = w_1 f_1 + w_2 f_2 + b)$[2], in three dimensions we have a plane $(0 = w_1 f_1 + w_2 f_2 + w_3 f_3 + b)$, and in n dimensions, we have a hyperplane $(0 = b + \sum_{i=1}^{n} w_i f_i)$

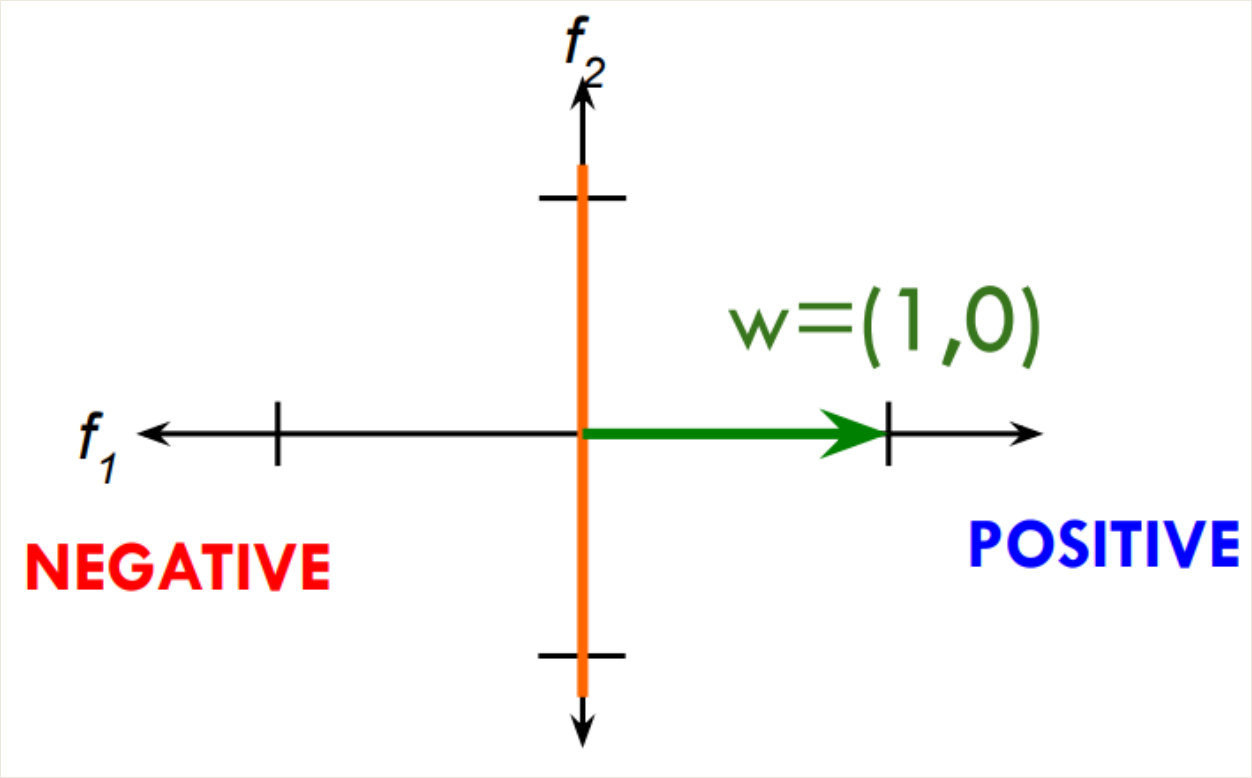
To classify data using this model we simply check the sign of the result.

Learning

To learn a linear model, we use a technique called #online-learning. In #online-learning we only see one example at a time. In comparison with the batch approach, where we're given some training data $\{(x_i, y_i) : 1 \leq i \leq n\}$, typically i.i.d[3], we receive the data points one-by-one. Specifically, the algorithm receives an unlabelled example x_i, predicts a classification of the example, and is then told the correct answer y_i so it can update its model before receiving the next example. This approach is particularly beneficial when we have a data stream or a large-scale dataset, or when we're working on privacy-preserving applications.

Example

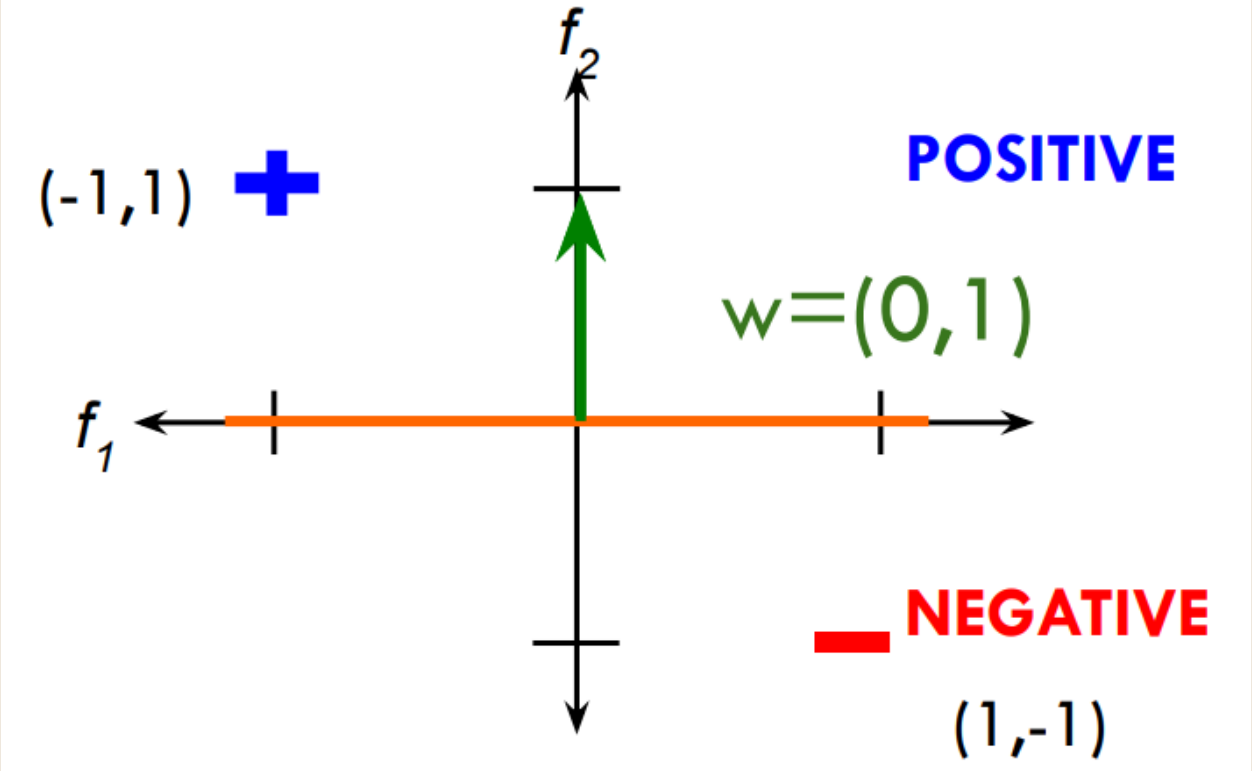
Let's say we have this initial line and weight vector $w = (1, 0)$, meaning we have the equation $0 = 1f_1 + 0f_2$.



We now receive the point $(-1, 1)$, which we classify as being part of the negative class as $1 * -1 + 0 * 1 = -1$. Unfortunately, this point is meant to be part of the positive class, so we have to update the model. Examining the values of the point, we see that f_1 contributed in the wrong direction and that f_2 didn't contribute at all, when it could have. In this case we should decrease w_1 (we'll go from 1 to 0) and increase w_2 (we'll go from 0 to 1) so we now have $w = (0, 1)$.

The first point is now classified correctly, and we receive the next point $(1, -1)$. Because $0 * 1 + 1 * -1 = -1$ we classify it as part of the negative class, and we're told that this is

correct. In this case we do not need to update the model.



#Loss-Functions

When training a model we need to choose a criterion to minimise. In the case of the model $0 = b + \sum_{j=1}^{n} w_j f_j$ we could choose to find the values of w and b for which $\sum_{i=1}^{n} 1[y_i(w \cdot x_i + b) \leq 0]$ is minimal. That is, minimising the $0/1$ loss or number of mistakes. Unfortunately, this particular function is far from continuous and is definitely not differentiable, making it incredibly hard[4] to find the minimum value of it. #Loss-functions are usually "ranked" based on how they score the difference between the actual label y and the predicted label $y^{'}$.

Surrogate #Loss-Functions

In many cases, we would like to minimise the $0/1$ loss. A surrogate #loss-function is a loss function that provides an upper bound on the actual loss function. Generally we want convex surrogate loss functions as they are easier to minimise. Some possibilities are the following:

- $0/1$ loss:
  - $l(y, y^{'}) = 1[yy^{'} \leq 0]$
- Hinge:
  - $l(y, y^{'}) = max(0, 1 - yy^{'})$
- Exponential:
  - $l(y, y^{'}) = exp(-yy^{'})$
- Squared loss:
  - $l(y, y^{'}) = (y - y^{'})^2$

#Gradient-Descent

Using derivatives we can find the slope of the function in our current position. Using this we can then choose which direction to move in order to minimise the function. Mathematically, when starting from a position w, our formula to move will look something like this

$w_j = w_j - \frac{d}{dw_j} loss(w)$[5]

We can add a further parameter to this equation, η[6], which we use as the learning rate:

$w_j = w_j - \eta \frac{d}{dw_j} loss(w)$

Through further complicated mathematical steps we find another possible formula we can use to search for the minimum: $w_j = w_j + \eta \sum_{i=1}^{n} y_i x_{ij} exp(-y_i(w \cdot x_i + b))$

For each example x_i we can take $w_j = w_j + \eta y_i x_{ij} exp(-y_i(w \cdot x_i + b))$ and for simplicity's sake we'll say $c = \eta exp(-y_i(w \cdot x_i + b))$.

When is c large/small?

Reminding ourselves that η is the learning rate, y_i is the label, and $w \cdot x_i + b$ is the prediction, we can see that if the label and the prediction have the same sign then the size of the step we'll take will grow as the prediction grows. On the other hand, if the signs differ, then the size of the step will grow with how different the values are.

Gradient

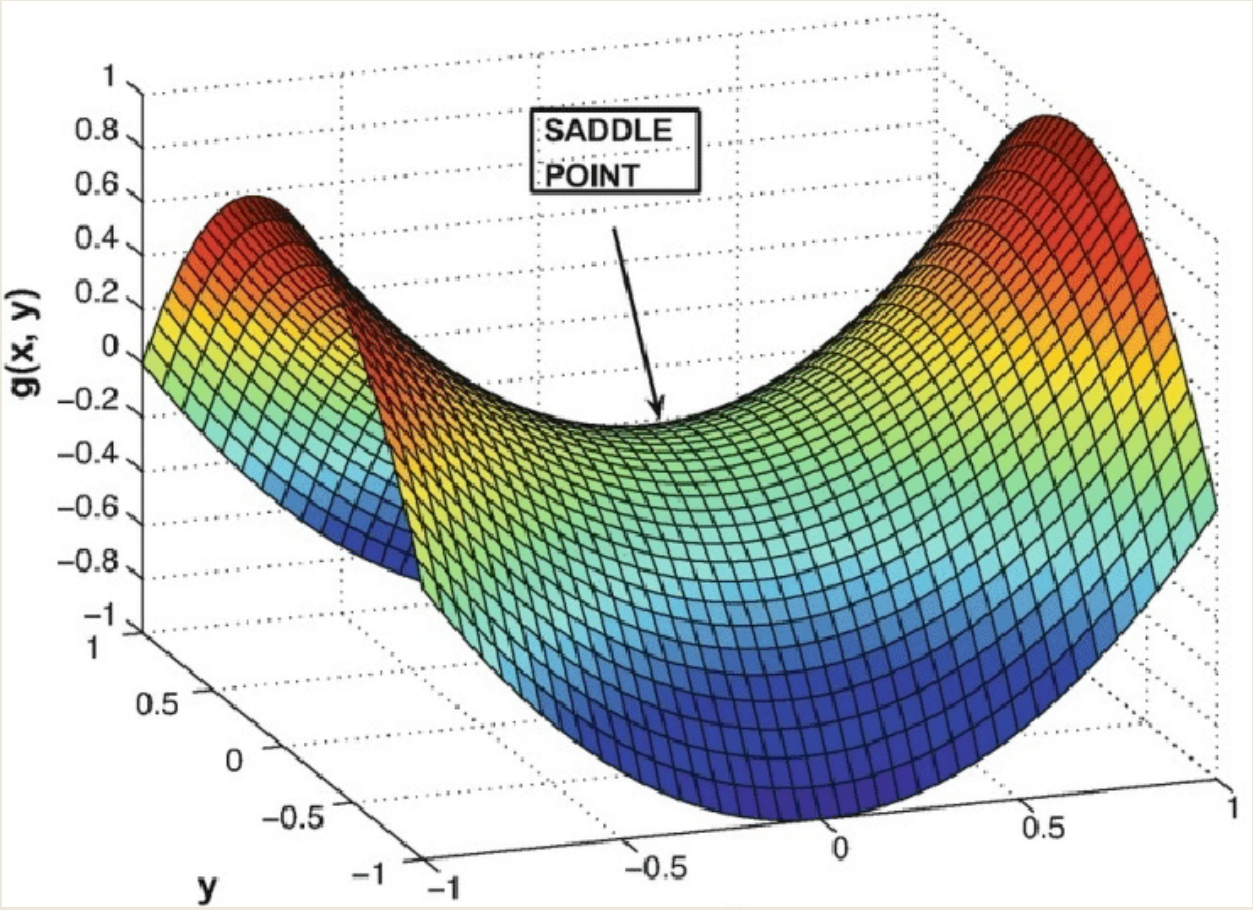
The gradient is the vector of partial derivatives with respect to all the coordinates of the weights: $\Delta_W L = [\frac{\delta L}{\delta w_1} \frac{\delta L}{\delta w_2} \dots \frac{\delta L}{\delta w_N}]$

Note

Not all optimisation problems are convex; they often have local minimums.

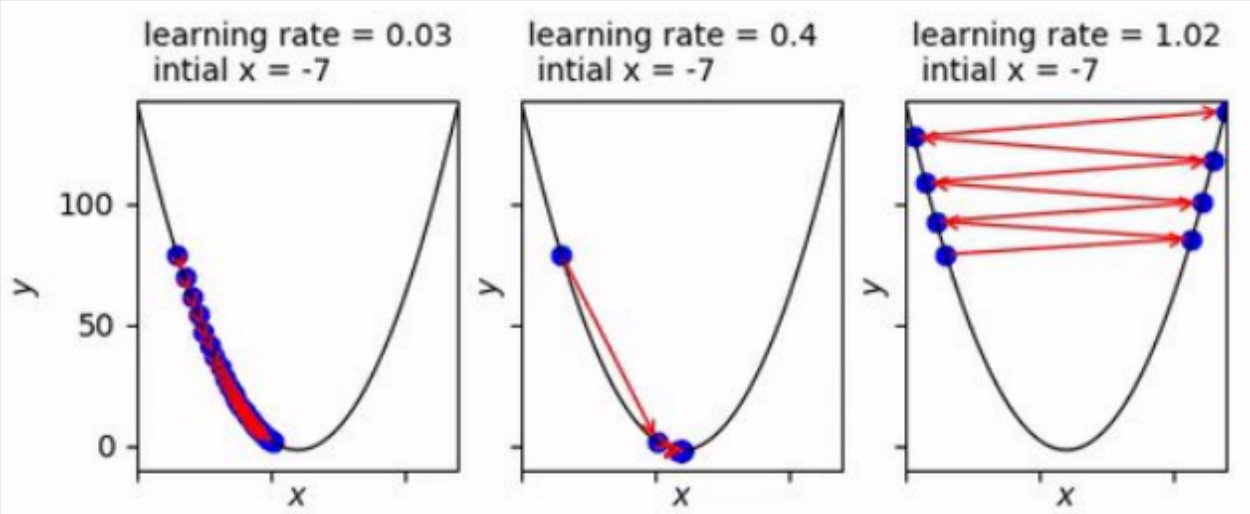
Saddle points

At a saddle point, some directions curve upwards and others curve downwards. Here the gradient is zero even though we aren't at a minimum, meaning we get stuck if we fall exactly on the saddle point. If we can move slightly to one side we can get unstuck. Saddle points are very common in high dimensions.



⚠ **The learning rate η is a very important hyper-parameter**

Choosing an appropriate value for the learning rate is vital to getting getting good results. As evidenced in the picture below, setting it too low will guarantee finding the exact minimum but will also make the whole process take longer than it has to. Setting it too high may make it impossible to apply the algorithm properly in more extreme cases.



#Regularisation

A #regulariser is a criterion we can add to the loss function of a model to make sure we don't overfit. It's a bias on the model that forces the learning to prefer certain types of weights over others $argmin_{w,b} \sum_{i=1}^{n} loss(yy') + \lambda\, regulariser(w,b)$

We generally want to avoid having huge weights, as a relatively small change in a feature could have a drastic impact on the prediction.

#P-Norm

$r(w,b) = \sqrt[p]{\sum |w_j|^p} = ||w||^p$

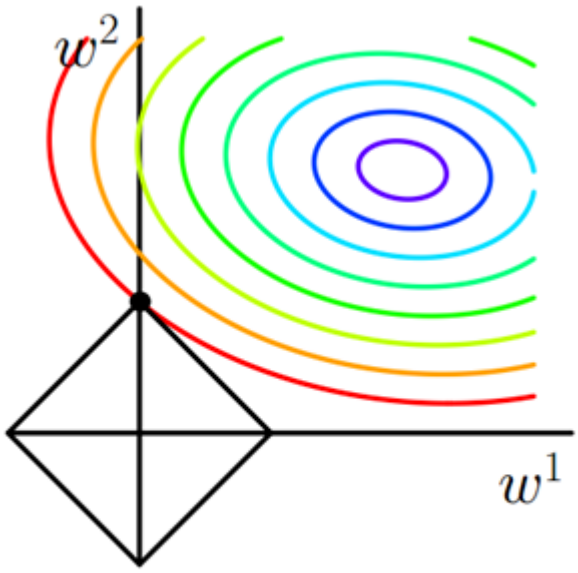
Smaller values of p[7] will encourage sparser vectors, whereas larger values will discourage large weights more.

Sum of weights[8]: $r(w,b) = \sum |w_j|$

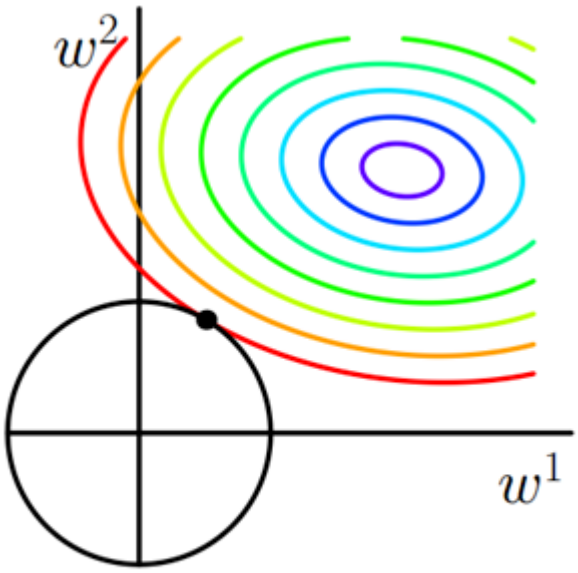
Sum of squared weights[9]: $r(w,b) = \sqrt{\sum |w_j|^2}$

In jargon, 1-norm is referred to as "lasso" and 2-norm is known as "ridge".

L1 and L2 regularised cost functions are relatively easy to graphically represent in two dimensions; in the below diagram the L1 and L2 "balls" are displayed on the origin of the (w_1, w_2) plane.



(a) ℓ_1-ball meets quadratic function. ℓ_1-ball has corners. It's very likely that the meet-point is at one of the corners.



(b) ℓ_2-ball meets quadratic function. ℓ_2-ball has no corner. It is very unlikely that the meet-point is on any of axes."

The optimal solution is where the contour first intersects the norm ball. Note that sparsity occurs in the corners of these balls. Due to this, we can intuitively imply that L2 is a good regularisation of the problem as we are much less likely to encounter scarsity.

TL;DR

L1 will tend to generate a small number of features setting all other features to 0, while L2 will select more features which will all be close to 0.In fact, in the above diagram we can see that w_1 is 0.

#Minimisation

If we can ensure that the sum of the loss function and the regulariser is convex then we can still use gradient descent: $argmin_{w,b} \sum_{i=1}^{n} loss(yy^{'}) + \lambda\ regulariser(w)$

#Convexity

Definition

The line segment between any two points on the function is *above* the function.

Mathematically, f is convex if for all x_1, x_2, $f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2)\ \forall\ 0 < t < 1$ where $f(tx_1 + (1-t)x_2)$ is the value of the function at some point between x_1 and x_2 and $tf(x_1) + (1-t)f(x_2)$ is the value at some point on the line segment between x_1 and x_2.

> **Note**
>
> If both the loss function and the regulariser are convex, then their sum will also be convex.

> **Note**
>
> P-norms are convex for $p \geq 1$

#Gradient-Descent

We have the following optimisation criterion $argmin_{w,b} \sum_{i=1}^{n} exp(-y_i(w \cdot x_i + b)) + \frac{\lambda}{2}||w||^2$ where the loss function [10] penalises examples where the prediction is different to the label and the regulariser[11] penalises large weights.

Regularisation with #P-norms

- L1: $w_j = w_j + \eta(loss\_correction\ - \lambda\ sign(w_j))$
- L2: $w_j = w_j + \eta(loss\_correction\ - \lambda w_j)$
- Lp: $w_j = w_j + \eta(loss\_correction\ - \lambda c w_j^{p-1})$

Summary

- L1 is popular because it tends to result in sparse solutions[12]. However, it is not differentiable, so it only works for gradient descent solvers.
- L2 is also popular because for some loss functions it can be solved in a "single" step[13]
- Lp is less popular since the weights don't tend to be shrunk sufficiently.

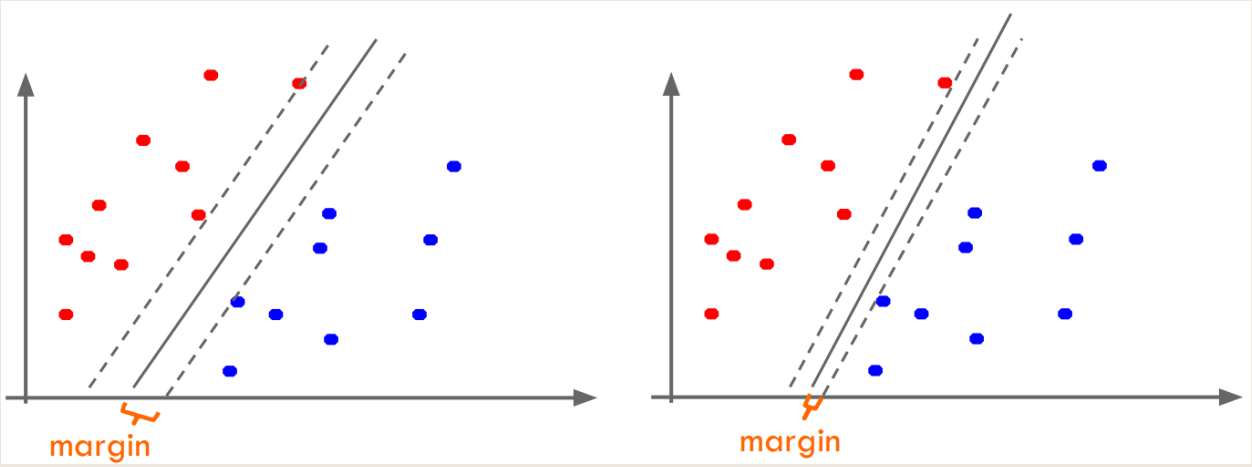
Support Vector Machines

The two main variations in linear classifiers are which line/hyperplane they choose when the data is linearly separable, and how they handle data that is not linearly separable.

> **#Perceptron Example**
>
> For #separable data, the #perceptron will find *some* hyperplane that separates the data, whereas for #non-separable data it will continue adjusting as it iterates through the examples and the final hyperplane will depend on which examples it saw most recently.

Definition

The #margin of a classifier is the distance to the closest points of either class. Large margin classifiers attempt to maximise this.



Support Vectors

Definition

For any separating hyperplane, there exists some set of closest points. These are called the support vectors. For n dimensions, there will be at least $n+1$ support vectors.

Maximising the margin is good, as it implies that only the support vectors matter and the other examples can be ignored.

Measuring the Margin

Given the hyperplane $wx - b = 0$ and margins equal to $wx - b = 1$ and $wx - b = -1$ we can calculate the distance between the two margins to be $\frac{2}{||w||}$, meaning the distance between the hyperplane and each of the margins is $\frac{1}{||w||}$[14].

We want to select the hyperplane with the largest margin where the points are classified correctly and lie outside the margin. Mathematically, $max_{w,b}\ \frac{1}{||w||}$ subject to $y_i(w \cdot x_i + b) \geq 1\ \forall i$. This can be simplified to $min_{w,b}\ ||w||$ subject to $y_i(w \cdot x_i + b) \geq 1\ \forall i$, meaning that maximising the margin is equivalent to minimising the norm of the weights[15].

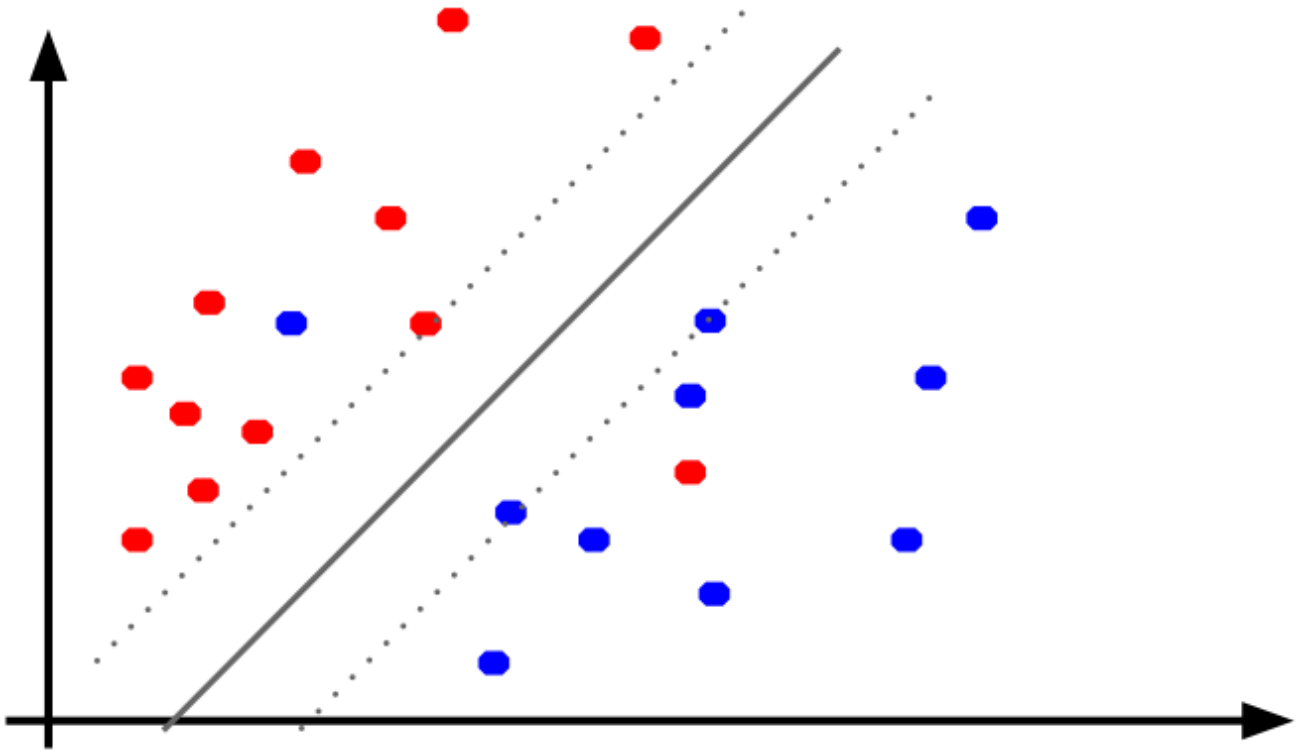
Support Vector Machine Problem

$min_{w,b}\ ||w||^2$ subject to $y_i(w \cdot x_i + b) \geq 1\ \forall i$

> **Note**
>
> This is an example of a quadratic optimisation problem. I.e., maximising/minimising a quadratic function subject to a set of linear constraints.

Soft Margin Classification

> **Problem**
>
> Using what we've seen so far, there are many models we'd like to learn but can't as our constraints are too strict. For example, in the following image we'd like to learn something similar to what's shown but we can't as the constraints we've set so far stop us from having wrongly classified examples[1].
>
> 1. of which there are clearly two in this case ↩



Slack Variables

> **Idea**
>
> To combat this, we can implement what are known as #slack-variables .

$min_{w,b}\ ||w||^2 + C\sum_i \zeta_i$[16] subject to $y_i(w \cdot x_i + b) \geq 1 - \zeta_i,\ \zeta_i \geq 0\ \forall i$

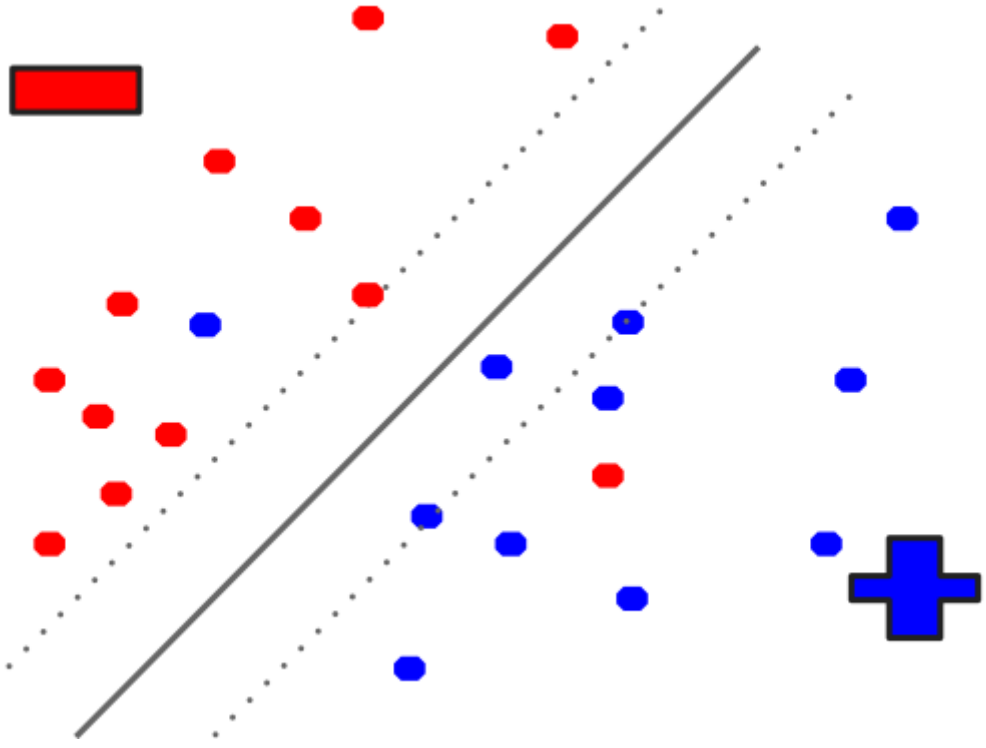
C can be interpreted as a way of controlling overfitting; it's a trade-off between margin maximisation and penalisation of the model. Having a small value of C allows constraints to easily be ignored, leading to a large margin, whereas having a large value of C makes the constraints hard to ignore, therefore leading to a narrow margin.

> **Note**
>
> $C = \infty$ enforces all constraints, leading to a "hard" margin like we had before.

The model as a whole gets penalised based on how far it is from being 100% correct. The "$\geq 1 - \zeta_i$" condition allows the model to make mistakes.

Understanding Soft Margin SVMs

Here is an example of a soft margin support vector machine:



> **What are the slack values for correctly classified points on or outside the margin?** ›

What are the slack values for correctly classified points inside the margin?

What are the slack values for incorrectly classified points?

Effectively, we get that $\zeta_i = 0$ when $y_i(w \cdot x_i + b) \geq 1$[17] and lie outside the margin and $\zeta_i = 0$ otherwise. This can be simplified to $\zeta_i = max(0, 1 - y_i(w \cdot x_i + b)) = max(0, 1 - yy')$.

The Dual Problem

Quadratic optimisation problems are a well-known class of mathematical programming problems for which several non-trivial algorithms exist. One possible solution involves constructing a dual problem where a Lagrange multiplier $\alpha_i$is associated with every inequality constraint in the primal/original problem:
$max_\alpha \sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j x_i^T x_j \; s.t. \;\; \sum_i \alpha_i y_i = 0, \; \alpha_i \geq 0, \; \forall i$

Solution

Given a solution $\alpha_1 \ldots \alpha_n$ to the dual problem, the solution to the primal is

$$w = \sum_i \alpha_i y_i x_i$$

$$b = y_k - \sum_i \alpha_i y_i x_i^T x_k$$

Each non-zero α_i indicates that the corresponding x_i is a support vector. Therefore, the classifying function is

$$f(x) = \sum_i \alpha_i y_i x_i^T x + b$$

Note

Note that we don't need w explicitly.

There are two important observations we can make here, namely:

- The solution relies on an inner product between the test point x and the support vectors x_i.
- Solving the optimisation problem involves computing the inner products between all training points.

Applying a Soft Margin

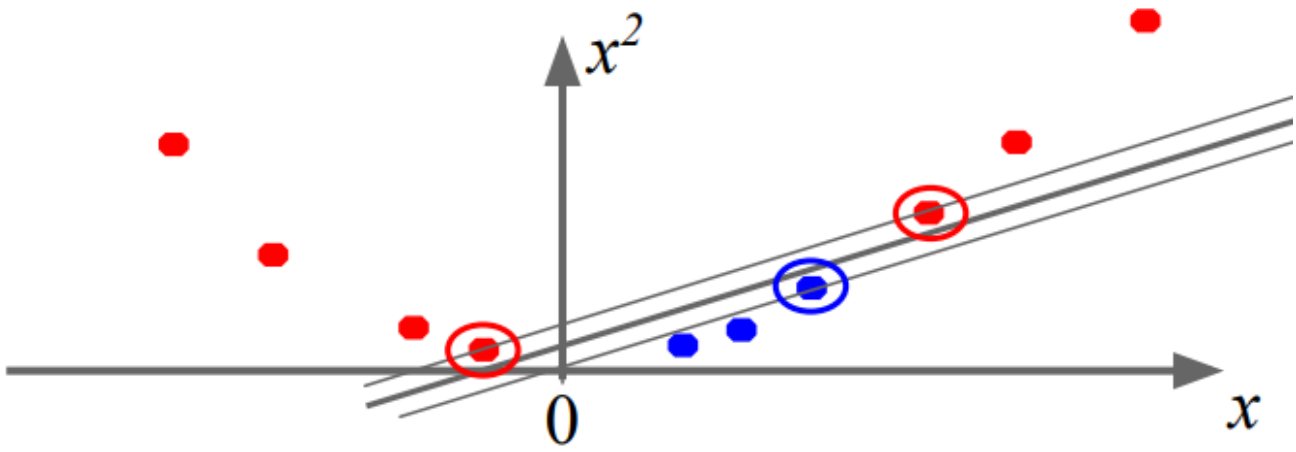
The main difference is in the constraints:
$max_\alpha \sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j x_i^T x_j \; s.t. \;\; \sum_i \alpha_i y_i = 0, \; 0 \leq \alpha_i \leq C, \; \forall i$
Again, x_i with non-zero α_i will be support vectors.

Non-Linear Suport Vector Machines

Datasets that are linearly separable work out great, even with some noise.

> **But what can we do if the dataset isn't linearly separable?**

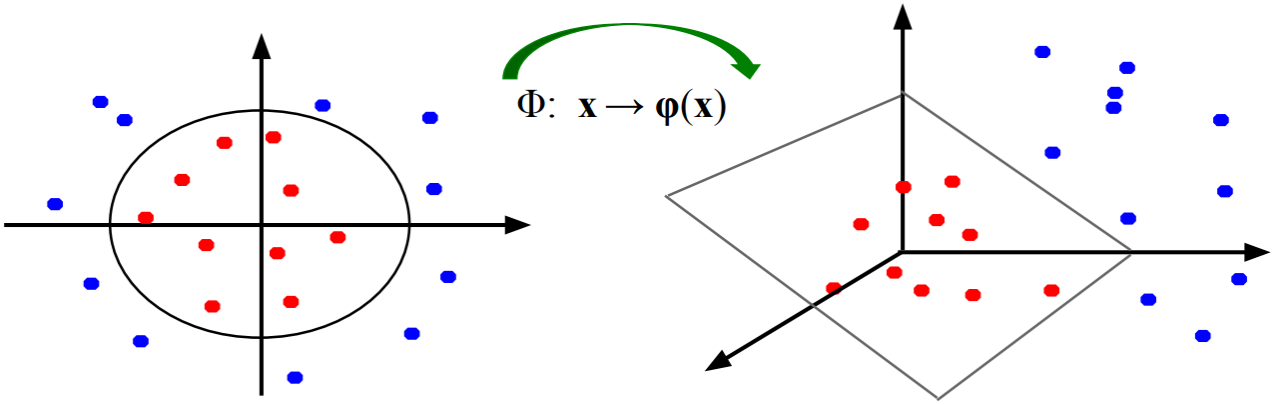
> **We can map the data to a higher-dimensional space**



Here we can see that if we only took into account the x axis then the dataset would not have been linearly separable.

Feature Spaces

In general, the original feature space can be mapped to some higher-dimensional feature space where the training set is separable. For example:



The Kernel Trick

The linear classifier relies on an inner product between vectors $K(x_i, x_j) = x_i^T x_j$. If every datapoint is mapped into high-dimensional space via some transformation $\Phi : x \rightarrow \phi(x)$, the inner product becomes $K(x_i, x_j) = \phi(x_i)^T \phi(x_j)$

> **Note**
>
> A kernel function is a function that is equivalent to an inner product in some feature space.

#Multiclass vs #Multilabel Classification

Multiclass

In #multiclass #classification, each example has *exactly one* label.

Multilabel

In #multilabel #classification, each example has *zero or more* labels.

Applications

Some examples of applications of multilabel classification could be video surveillance[18], scene analysis[19], image annotation, document topics and medical diagnoses.

Ranking Problems

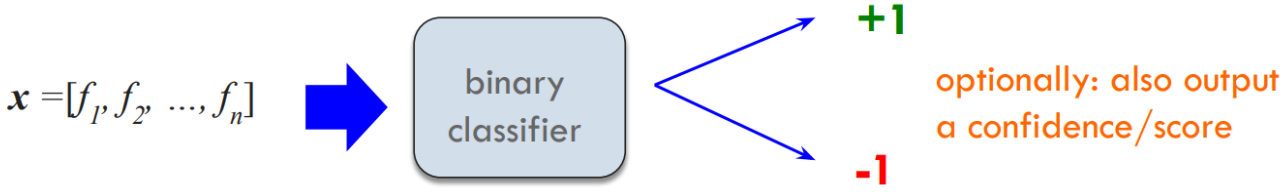
The training data for a ranking problem is a set of rankings where each ranking consists of a set of ranked examples.

Applications

- Searching
- Recommendations
- Image captioning
- Machine translation

A Black Box Approach

Given a generic binary classifier, can we use it to solve the ranking problem?



Problem

How do we train a classifier that predicts preferences?

Problem

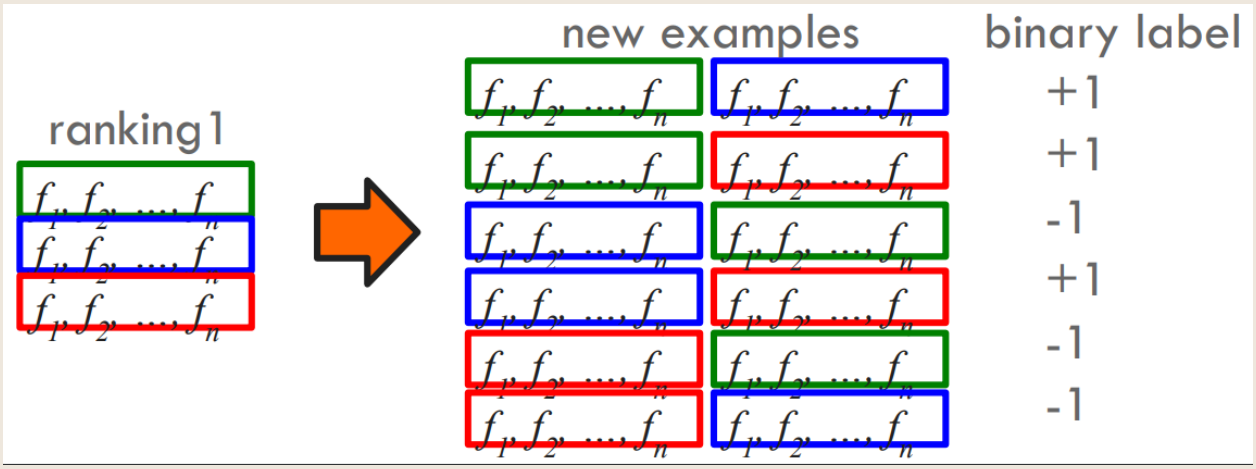
How do we turn the predicted preferences into a ranking?

Predicting 'Better' or 'Worse'

To respond to the first of the above problems, we can train a classifier to decide if the first of a pair of inputs is better or werse than the second.

Example

Consider all the possible pairings of the examples in a ranking. Label the pair as positive if the first example is higher ranked, and negative otherwise.



Problem

Unfortunately, our binary classifier can only take one example as input at a time and we're trying to pass it two.

Solution

#Combined-Feature-Vectors are a great way of comparing two given examples, and there are many possible approaches to constructing them. Two of the most common of these are:

- Difference
  - $f_i' = a_i - b_i$
- Greater than/less than
  - $f_i' = 1 \; if \; a_i > b_i, \; 0 \; otherwise$

The #Preference-Function

Let's assume we have N queries and M samples in our training set. In its simplest form, the goal of ranking is to train a binary classifier to predict a #preference-function. Given a query q and two samples x_i and x_j, the classifier should predict whether x_i should be preferred to x_j with respect to the query q.

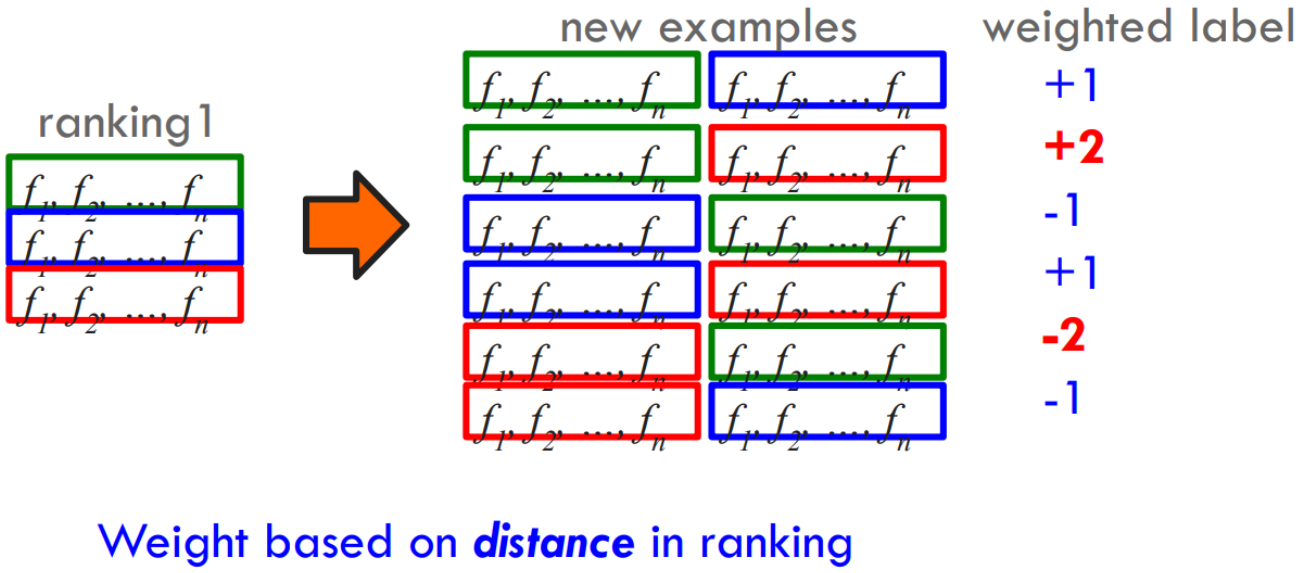
```
def naiveRankTrain(RankingData, BinaryTrain):
        D = []
        for n in range(1, =N):
                for all i, j in range(1, =M) and i ≠ j:
                        if i.preferredTo(j).withQuery(n):
                                D.append(x[n][i][j], 1)
                        else:
                                D.append(x[n][i][j], -1)
        return BinaryTrain(D)
```

```
def naiveRankTest(f, x):
        score = [0] * M
        for i, j in range(1, =M) and i ≠ j:
                y = f(x[i][j])
                score[i] += y
                score[j] -= y
        return argSort(score)
```

#Bipartite-Ranking

The above algorithms are useful for #bipartite-ranking, a problem in which you are only ever trying to predict a binary response, for example "is this document relevant or not?". The only goal is to ensure that all the relevant documents are ahead of all the irrelevant documents; there is no notion that one relevant document is more relevant than another. For non-bipartite ranking problems more sophisticated methods can be used.

This helps us improve on the example in the image in the above section "Predicting 'Better' or 'Worse'" like so



Ranking Improved

If the preferences are more nuanced than "relevant or not" then we can incorporate these preferences at training time; we want to give a higher weight to binary problems that are very different in terms of preference to others.

> **Note**
>
> Rather than producing a list of scores and then calling a sorting algorithm, we can actually use the preference function as the sorting function.

Define a ranking as a function σ that maps the objects[20] we are ranking to the desired position in the list; $1, 2, \ldots, M$. If $\sigma_u < \sigma_v$ then u is preferred to v, i.e., it appears earlier on the ranked document list.

Given data with observed rankings σ, our goal is to learn to predict rankings for new objects, σ^*. We define Σ_M as the set of all ranking functions over M objects.

We want to model the fact that making a mistake on some pairs is worse than making a mistake on others, so we define a cost function ω, where $\omega(i, j)$ is the cost for accidentally putting something in position j when it should have gone in position i. A valid cost function ω must be:

- Symmetric
  - $\omega(i, j) = \omega(j, i)$
- Monotonic
  - If $i < j < k$ or $i > j > k$ then $\omega(i, j) \leq \omega(i, k)$
- Satisfy the triangle inequality
  - $\omega(i, j) + \omega(j, k) \geq \omega(i, k)$

Depending on the problem the cost function may be defined in different ways. If $\omega(i, j) = 1$ whenever $i \neq j$, it simply counts the number of pairwise misordered items.

> **Note**
>
> We can impose only the top K predictions as being correct:
> $\omega(i, j) = 1 \; if \; min(i, j) \leq K \; and \; i \neq j, \; otherwise \; 0$
>
> > **Example**
> >
> > This is particularly useful in web search algorithms, which may only display up to $K = 10$ results to the user at a time.

Error could be calculated like so: $\mathbb{E}_{(x,\sigma)\sim D}[\sum_{u \neq v} 1[\sigma_u < \sigma_v] 1[\hat{\sigma}_v < \hat{\sigma}_u] \omega(\sigma_u, \sigma_v)]$ where $\hat{\sigma} = f(x)$. This means that if the true ranking σ prefers u to v, but the predicted ranking $\hat{\sigma}$ prefers v to u, then you incur a cost of $\omega(\sigma_u, \sigma_v)$.

```
def rankTrain(D_rank, omega, binaryTrain):
        D_bin = []
        for (x, sigma) in D_rank:
                for u ≠ v:
                        y = SIGN(sigma_v - sigma_u)
                        w = omega(sigma_u, sigma_v)
                        D_bin.append(y, w, x_uv)
        return binaryTrain(D_bin)
```

At test time, instead of predicting scores and then sorting the list, we can simply run the quicksort algorithm using the learnt function as a comparison function. In practice at each step a pivot p is chosen, and every object u is compared to p using the learnt function and sorted to the left or right accordingly.

> **Note**
>
> The difference between this algorithm and quicksort is that our comparison function is probabilistic.

---

1. n features ↩
2. where $b = -a$ ↩
3. independent and identically distributed ↩

4. NP-Hard, in fact ↩
5. we subtract because if the derivative is negative it means that to go downhill we should move right, therefore we should increase the input ↩
6. eta ↩
7. $p < 2$ ↩
8. 1-norm/L1 ↩
9. 2-norm/L2 ↩
10. $exp(-y_i(w \cdot x_i + b))$ ↩
11. $||w||^2$ ↩
12. lots of zero-weights ↩
13. no gradient descent required, though solvers are often iterative anyway ↩
14. this is the value we will generally use for the size of a margin ↩
15. as long as the hyperplane satisfies all other constraints ↩
16. zeta ↩
17. the points are correctly classified ↩
18. reidentification/tracking people between CCTV "scenes" ↩
19. visual localisation ↩
20. documents ↩