IAML - Week 5

Antonio León Villares

October 2021

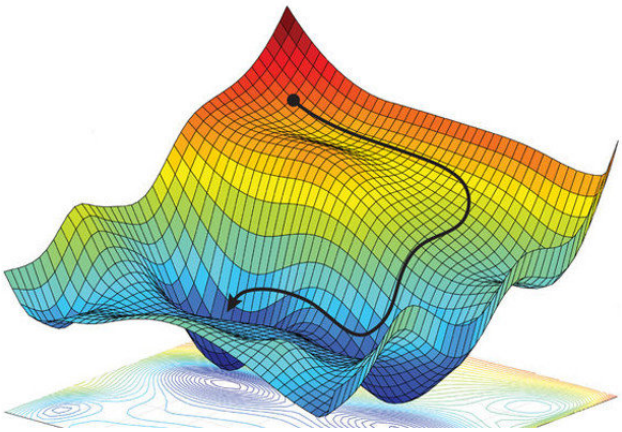
Contents

1 Optimisation

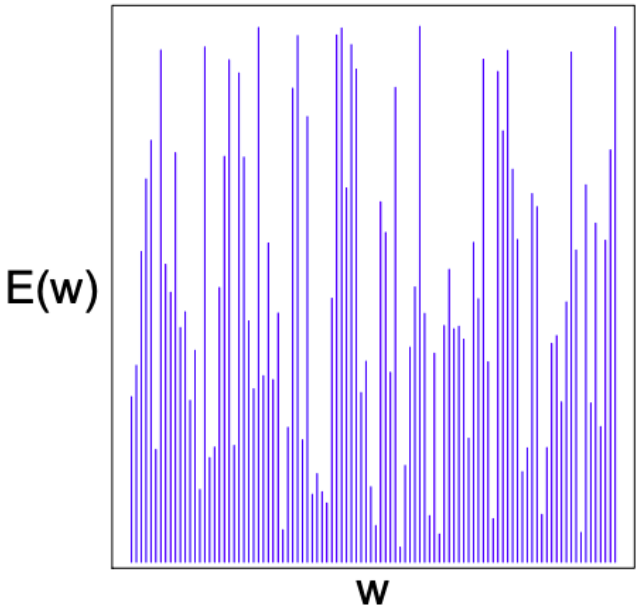
- Sometimes error expression can't be optimised analytically, so numerical methods are used
- Gradient descent uses the partial derivative of the error function to change the weights and reduce error
- Gradient descent can be done in batch, online or mini-batch
- Gradient descent can have problems if the error surface has a shallow minimum, many local minima or if the surface has curvature

1.1 The Importance of Optimisation in Machine Learning

- **How is optimisation related to solving a machine learning problem?**
  - learning problems are often described in terms of optimising a certain quantity
  - for example, in linear regression we want to minimise the least squares error, or max likelihood in Naive Bayes
- **Are minimisation and maximisiation different optimisation problems?**
  - any minimisation problem can be transformed into a corresponding maximisation problem, and viceversa
  - typically we transform optimisation problems into minimisation problems
- **What do we typically optimise in machine learning?**
  - we aim to optimise an error function $E(\underline{w})$, where $\underline{w}$ represent the weights of our model, and $E(\underline{w})$ is compute by considering the error between our model and the training data
- **How can we visualise the process of optimisation?**
  - we can "see" $E(\underline{w})$ as a surface in some dimension
  - optimisation corresponds to moving down the surface, until we reach a minimum of the surface



- **Is every problem in machine learning optimisable?**
  - problems which are optimisable in an intelligent way typically require that $E(\underline{w})$ is smooth
  - if the error function is not smooth, optimisation can only done via brute force (i.e try all possibilities for $\underline{w}$)



1.2 Numerical Optimisation

- **Is it always possible to obtain an analytic solution to an optimisation problem?**
  - more often than not, expression for $E(\underline{w})$ can only be optimised via numerical methods (i.e logistic regression, deep learning)
- **How do derivatives represent the optimisation problem?**
  - if we consider:
    $$\frac{\partial E}{\partial w_i}$$
    this is telling us how E changes locally, given some small change in a component w_i of $\underline{w}$
  - overall, if we consider the **gradient vector**:
    $$\nabla E(\underline{w}) = \frac{\partial E}{\partial \underline{w}}$$
    this is telling us the direction of greatest change of E given changes in $\underline{w}$
  - thus, if we follow the direction of $\nabla E(\underline{w})$ we will be going in the direction of least error (locally at least)
- **How does numerical optimisation minimise the error?**
  - a numerical optimisation problem seeks to solve:
    $$\min_{\underline{w}} E(\underline{w})$$
  - in general, they need to compute $E(\underline{w})$ and follow a systematic procedure to decrease E, either by:
    - using derivatives
    - using higher order derivatives
    - some other method
  - the general algorithm is:

```
initialize w
while E(w) is unacceptably high
        calculate g = ∇E
        Compute direction d from w, E(w), g
                (can use previous gradients as well...)
        w ← w − η d
end while
return w
```

1.3 Gradient Descent

1.3.1 Defining Gradient Descent

- **What is the gradient descent algorithm?**
  - **Gradient Descent** is a numerical optimisation algorithm in which:

$$\underline{d} = \nabla E$$

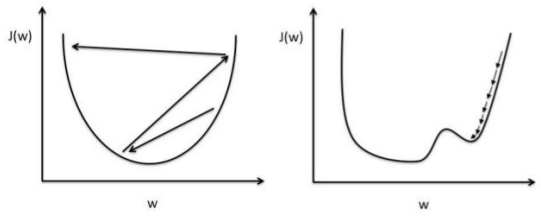
  - in other words, we change the weights by using the gradient vector of the error at a given iteration:

initialize **w**
while $E(\mathbf{w})$ is unacceptably high
    calculate $\mathbf{g} \leftarrow \frac{\partial E}{\partial \mathbf{w}}$
    $\mathbf{w} \leftarrow \mathbf{w} - \eta\,\mathbf{g}$
end while
return **w**

1.3.2 The Learning Rate

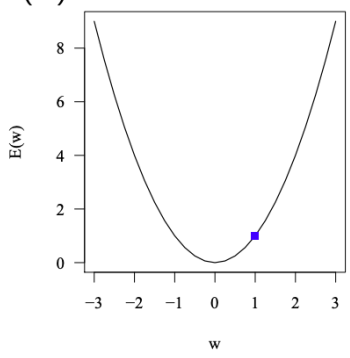
- **What is the learning rate?**
  - the **learning rate** is the positive constant η
- **How does learning rate influence the rate of convergence to a solution?**
  - as its name indicates, the value of η determines the influence of the gradient on the new value of the weights
  - a small η means that learning will be slow, as at each iteration we only take small steps
  - a large η means that learning can be unstable, as it is possible to completely miss out on the minimum, since we will be constantly overshooting

Learning rate



Overshooting Learn too slow

$E(w) = w^2$



- Take $\eta = 0.1$. Works well.

$$w_0 = 1.0$$
$$w_1 = \mathbf{w}_0 - 0.1 \cdot 2w_0 = 0.8$$
$$w_2 = \mathbf{w}_1 - 0.1 \cdot 2w_1 = 0.64$$
$$w_3 = \mathbf{w}_2 - 0.1 \cdot 2w_2 = 0.512$$
$$\dots$$
$$w_{25} = 0.0047$$

- **How does "Bold Driver" Gradient Descent improve on standard gradient descent?**
  - Bold Drive Gradient Descent decreases the chance of picking an η which is too small or too large, by modulating its value at each iteration depending on the effect of η on E:

```
initialize w, η
initialize e ← E(w); g ← ∇E(w) while η > 0
        w_1 ← w − ηg
        e_1 = E(w_1); g_1 = ∇E
        if e_1 ≥ e
            η = η/2
        else
            η = 1.01η; w ← w_1; g ← g_1; e = e_1
end while
return w
```

Figure 1: At each step, computes the error given a new set of (temporary) weights. If the error increases from the previous iteration, we halve η, and repeat. Once we observe that the error decreases, we set the new weights, and increase η. This means that we use small η only when absolutely necessary, and cautiosuly increase it to speed up training.

- the pitfall of BDGD is the fact that it is likely to get stuck in local minima

1.3.3 Batch vs Online Learning

- **What are the differences between batch and online learning?**
  - **batch** learning is as we have been doing thus far: compute the error function using our weights and **all** of the training data:

$$E(\underline{w}, (\underline{x}_1, y_1), \ldots, (\underline{x}_n, y_n)) = \sum_{i=1}^{n} E(\underline{w}, (\underline{x}_i, y_i))$$

  - in the case of gradient descent, it means that we change change the weights only after computing the error using all training instances:

$$\frac{\partial E}{\partial \underline{w}} = \sum_{i=1}^{n} \frac{\partial E_i}{\partial \underline{w}}$$

    where E_i denotes the error from training instance i. This is the same algorithm as presented above.
  - in **online** learning, we change the weights after computing the error of a single instance:

$$\frac{\partial E_i}{\partial \underline{w}}$$

  - an example of **online** learning is **Stochastic Gradient Descent**:
- **When should one use batch or online learning?**

- batch learning is good as:
  - it is very powerful for optimisation
  - easier to analyse

  It should be used when the training data is not expected to be updated, or when the amount of training instances is not too large, as training can be quite expensive
- online learning is good as:
  - it can be used with growing datasets
  - has the possibility of jumping over local minima

- **What is mini-batch learning?**
  - a hybrid between batch and online learning
  - we can split the training data into mini-batches. Then, for each mini-batch we use batch learning, and obtain a set of weights, which are given as starting weights for the next mini-batch, which is then trained, etc ...

1.3.4 Issues with Gradient Descent

- **What are the main issues associated with Gradient Descent?**
  - **setting η**
  - **shallow valleys**
    - if the surface gets shallow close to the minima, learning will be slowed down
    - can use **momentum** to reduce effect of shallowness:

$$\underline{d}_t = \beta \underline{d}_{t-1} + (1-\beta)\eta \nabla E(\underline{w}_t)$$

    - β controls whether we want to give preference to the actual gradient, or the previous direction of movement
    - this comes at the cost on having to decide on values for η and β
  - **curved error surface**
    - since derivatives are local, they might not point towards the local optimum
    - this is caused by **curvature**

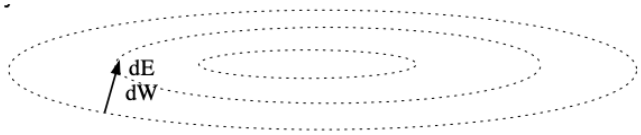


Figure 2: The steepest direction is not pointing in the direction of the optimum (more to the right)

* some functions like parabolas don't have this problem

– **local minima**

* once the algorithm hits a local minimum it will stop
* this is an inherent problem which can only be ameliorated, for instance by repeatedly applying the algorithm
* some functions (squared error, likelihood error) are **convex**, so they only have a **global minimum**

1.3.5 Further Topics (Not Covered)

- Some of these issues (shallow valley, curved error surfaces) can be fixed
  - Some of these are *second-order* methods like Newton's method that use the second derivatives
  - Also there are fancy first-order methods like quasi-Newton methods (e.g., *limited memory BFGS*) and conjugate gradient
  - They are the state of the art methods for logistic regression (as long as there are not too many data points)
  - We will not discuss these methods in the course.
- Other issues (like local minima) cannot be easily fixed

- Sometimes the optimization problem has *constraints*
  - Example: Observe the points $\{0.5, 1.0\}$ from a Gaussian with known mean $\mu = 0.8$ and unknown standard deviation σ. Want to estimate σ by maximum likelihood.
  - Constraint: σ must be positive.
  - In this case to find the maximum likelihood solution, the optimization problem is

$$\max_{\mu,\sigma} \sum_{i=1}^{2} \frac{1}{2\sigma^2}(x_i - \mu)^2$$
$$\text{subject to } \sigma > 0$$

  - There are ways to solve this (in this case: can be done analytically). We will not discuss them in this course.

2 Regularisation

- Towards Data Science: Regularisation in Machine Learning
- On why large weights lead to overfitting

- Regularisation is the introduction of terms in error computation to avoid overfitting
- Ridge Regression uses regularisation to prevent the model weights from getting too large

2.1 Defining Regularisation

- **What is the aim of regularisation, and how does it enforce it during learning?**
  - **regularisation** aims at creating models which are less prone to overfit data
  - it does this by penalising large weights in a model, making the model less flexible
  - larger weights are indicative of overfitting, since they overestimate the effect of a given parameter; they can be made arbitrarily large to perfectly fit the training data
- **On which models can regularisation be applied?**
  - regularisation requires continuous models, such as linear regression
  - it won't work on discrete models, such as decision trees

2.2 Ridge Regression

- **What is ridge regression?**
  - our standard linear regression, but using regularisation to avoid overfitting
- **How is regularisation implemented within the error function for polynomial regression?**
  - since we want to penalise big weights, we can alter error:

$$E(\underline{w}) = |\underline{y} - \mathbf{\Phi}\underline{w}|^2 + \lambda|\underline{w}|^2$$

– this is all derived by using **Lagrangian Multipliers**: we constrain our original error function $E(\underline{w}) = |\underline{y} - \mathbf{\Phi}\underline{w}|^2$ by requiring that $|\underline{w}|^2 < C^2$. Then we know from Lagrangian Multipliers that:

$$\mathcal{L}(\underline{w}) = |\underline{y} - \mathbf{\Phi}\underline{w}|^2 + \lambda(|\underline{w}|^2 - C^2)$$

We reach the formula above by ignoring C as it just a constant which shouldn't affect the error

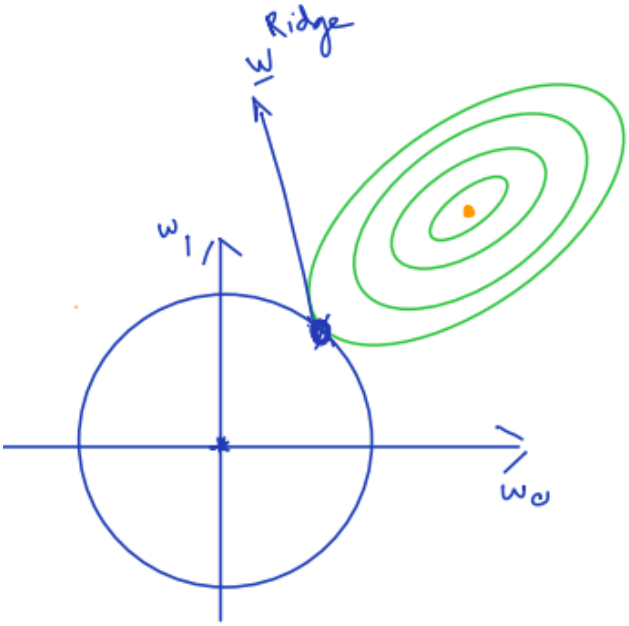


Figure 3: The green concentric ellipses represent our error surface, whilst the circle represents our constraint. The minimisation requires us to find the set of weights at the intersection of the 2 curves.

- **How are weights computed in Ridge Regression?**
  - they are very similar to standard linear regression weights:

$$\underline{w} = (\mathbf{\Phi}^T\mathbf{\Phi} + \lambda\mathbb{I})^{-1}\mathbf{\Phi}^T y$$

- **How does the regularisation parameter affect the error?**
  - if $\lambda \to 0$, we can see that we will be getting the same weights as with our standard linear regression
  - if $\lambda \to \infty$, the only relevant term in $(\mathbf{\Phi}^T\mathbf{\Phi} + \lambda\mathbb{I})^{-1}$ becomes the term with λ, so:

$$\underline{w} \to \frac{1}{\lambda}\mathbf{\Phi}^T y \to 0$$

so our model weights just go to 0

– thus, we can see that, the larger the λ, the more we are penalising a "complex" model, with large weights

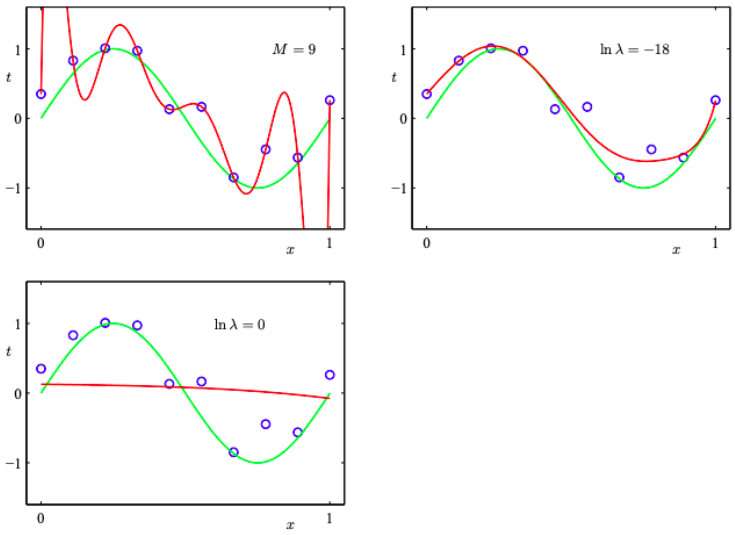


Figure 4: We try to fit the data with a degree 9 polynomial. If we have no regularisation term, we are overfitting. By introducing just a small λ, we already get a much better model. Getting λ too large produces an overly rigid model, which will look nearly linear.

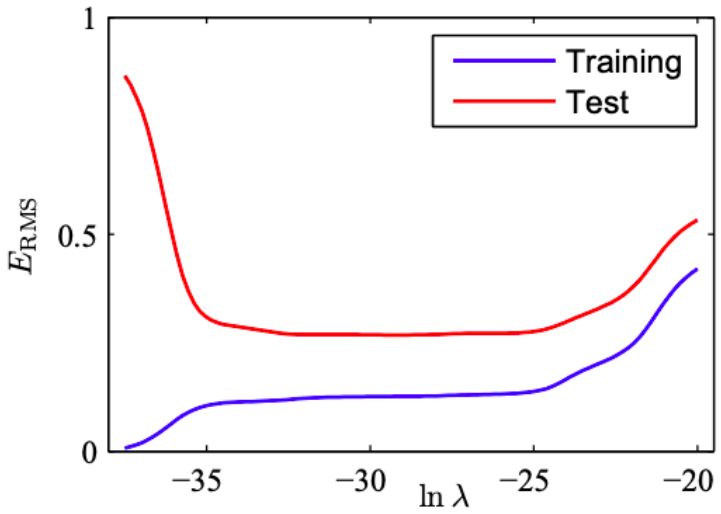
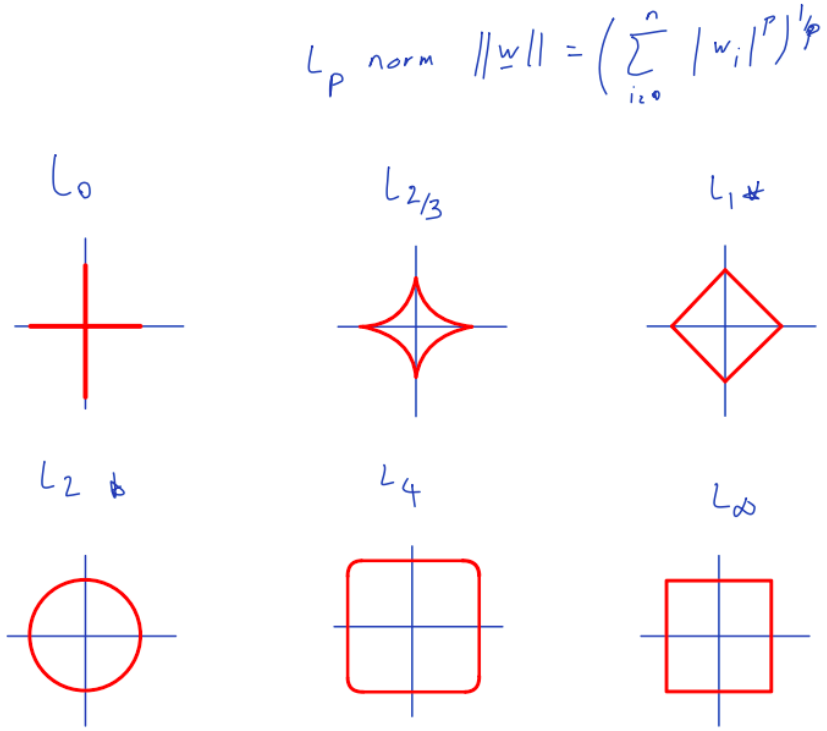
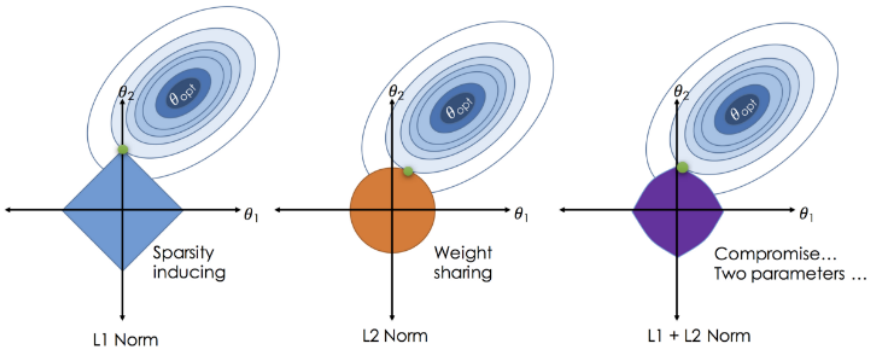


Figure 5: This is a graph giving the training and testing error for the data above, approximated by a degree 9 polynomial. We can see that by slightly increasing λ, we are slightly increasing the training error, but getting much better testing performance.

- **What is L_p regularisation?**
  - using $|\underline{w}|^2 < C^2$ is just one possible type of constraint
  - L_p regularisation refers to creating constraints involving $\underline{w}$ and the $p-$norm. Using different norms means that instead of circles, the constraint might define squares, or other curves, thus changing the weights in different ways



– for example, using $p = 1$, our constraint becomes a square, which leads to **sparse** weights (i.e the vecot rof weights has many 0). This is called **LASSO Regularisation**:



- **How does the ridge regression task differ from standard regression?**
  - we could describe the standard linear regression task via:

    - **Task**: regression
    - **Model Structure**: linear regression model
    - **Score Function**: squared error
    - **Optimisation/Search Method**: analytic solution/calculus
  - ridge regression is quite similar
    - **Task**: regression
    - **Model Structure**: linear regression model
    - **Score Function**: squared error **with quadratic regularisation**
    - **Optimisation/Search Method**: analytic solution/calculus
  - we can see that we can indeed train the same model structure, but use different scoring function

- **How can we use a validation set to evaluate a standard regression model?**
  - we can split our data into training, validation and testing sets
  - if we didn't have a regularisation parameter, then we:
    1. train M models on training data, using polynomials of orders $1, 2, \ldots, M$
    2. measure error on validation set for each of the M models
    3. select the model which performs the best on validation, and use it for testing

- **How can we estimate the regularisation parameter λ?**
  - since λ is continuous, we can't search through all values, as with polynomials of discrete order
  - we typically select a set of geometrically distributed values (for example, $\lambda \in \{0.01, 0.1, 0.5, 1, 10\}$). Then we can train the model, and evalaute with the validation set, tweaking the value if necessary.

- **Why don't we set λ based on the training set?**
  - if we did this, then the model would tend to using $\lambda = 0$, since this leads to the lowest training error (see graph above)

3 Support Vector Machines

- General, thorough explanations on SVM by UoT
- Decent overview of SVM, gives good intuition without getting too mathematical
- A derivation for SVM optimisation

- Slack variables + nice diagrams to illustrate
- Everything SVM, including an intuitive idea of what the SVM decision boundary illustrates + nice diagrams

- SVMs are linear classifiers, which construct a hyperplane, such that it maximises the margin
- We can use slack variables to account for missclassifications in SVM models with non-separable data
- We use the kernel trick to efficiently use SVMs with non-linear data

3.1 Introducing SVM

- **What is the Support Vector Machine algorithm?**
  - an algorithm used to build a **linear** classifier
- **How does SVM decide on the decision boundary?**
  - intuitively, it uses **support vectors** to build the decision boundary
  - these **support vectors** correspond to the class instances that are the most ambiguous

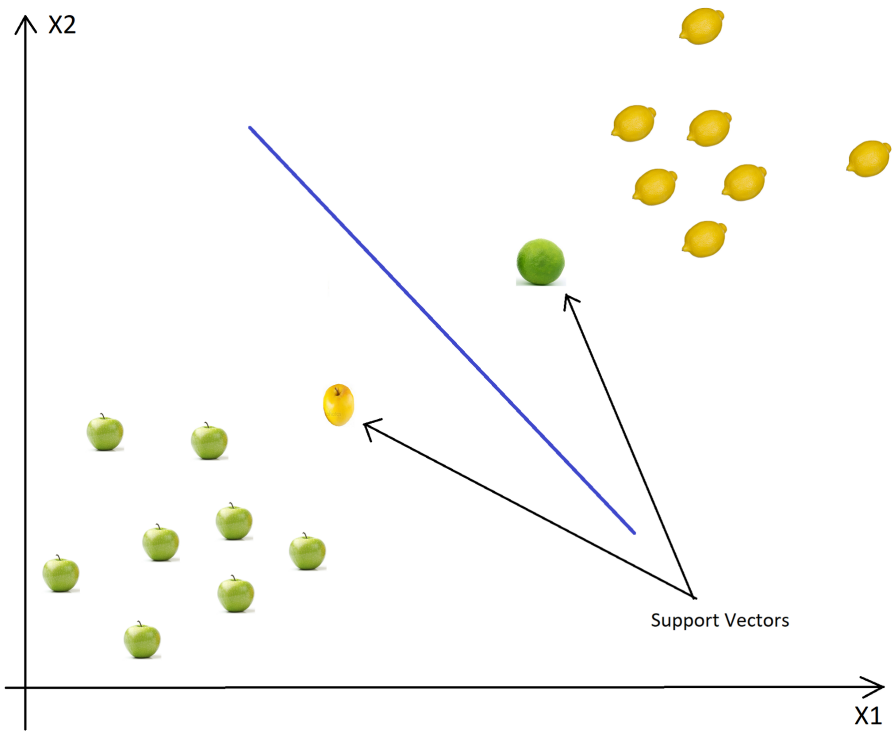


Figure 6: The support vectors are given by an apple that looks like a lemon, and a lemon which looks like an apple. SVM will build a hyperplane with as much distance from the support vectors as possible.

3.2 The Max Margin Optimisation Problem

3.2.1 Setting Up Optimisation for SVM

- **What is the decision boundary of a linear decision classifier?**
  - for a general linear classifier, we construct a hyperplane define by:

$$\underline{w} \cdot \underline{x} + w_0 = 0$$

    where $\underline{w}$ is a vector perpendicular to the hyperplane (this is just the normal equation for a hyperplane from linear algebra)

- **What makes a good decision boundary?**
  - there are infinitely many decision boundaries which can perfectly split 2 separable classes

– finding the best such decision boundary means finding a hyperplane which maximises its distance from any instance of both classes

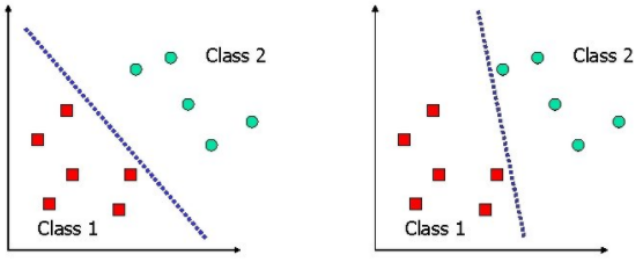


Figure 7: 2 bad decision boundaries

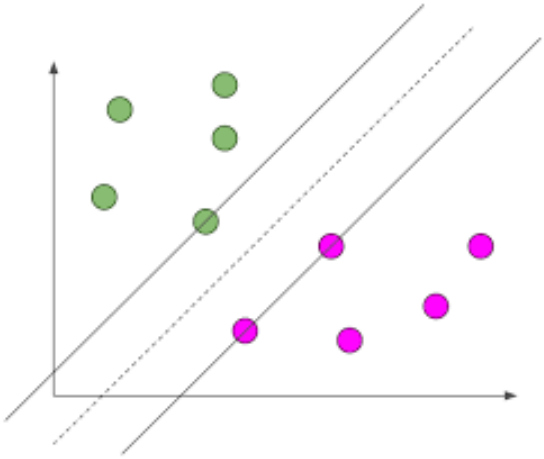


Figure 8: A good decision boundary

- **What is the margin in SVM?**
  - the **margin** is the distance between any given **hyperplane**, and the closest training instance
  - from the discussion above, it follows that **maximising the margin** is the optimisation task for SVM

3.2.2 Deriving a Formula for the Margin

To derive the formula, we consider the following steps:

1. Compute the position vector of the point in the hyperplane closest to the origin (this must be the point which is perpendicular to the hyperplane)

2. Compute the distance between any training instance and the hyperplane, by using the previous result

3. Define constraints which the margin must satisfy

4. Derive a formula for the margin using the constraints

1) Compute the position vector of the point in the hyperplane closest to the origin

The shortest distance between a line (hyperplane) and a point is the perpendicular distance. Let $\underline{z}$ be the position vector of the point in the hyperplane closest to the origin.

By construction of the hyperplane $\underline{w} \cdot \underline{x} + w_0 = 0$, we know that $\underline{w}$ is a vector perpendicular to the hyperplane, and so, $\underline{z}$ must be in the direction of $\underline{w}$. In particular, we have:

$$\underline{z} = \|\underline{z}\| \frac{\underline{w}}{\|\underline{w}\|}$$

Finally, since $\underline{z}$ is on the hyperplane, it must satisfy its equation, so:

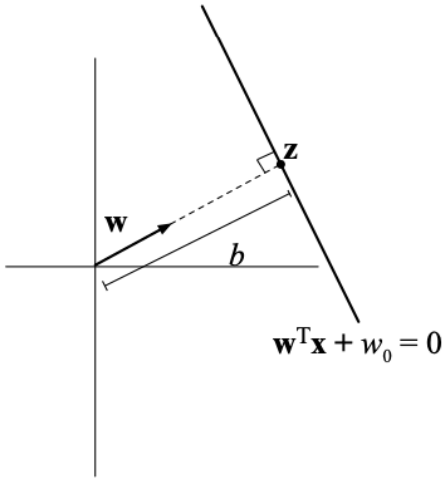
$$\underline{w} \cdot \underline{z} + w_0 = 0$$

If we plug in the expression for $\underline{z}$ above:

$$\begin{aligned}
& \underline{w} \cdot \underline{z} + w_0 = 0 \\
\implies \quad & \underline{w} \cdot \left(\|\underline{z}\| \frac{\underline{w}}{\|\underline{w}\|} \right) + w_0 = 0 \\
\implies \quad & \frac{\|\underline{z}\|}{\|\underline{w}\|} \underline{w} \cdot \underline{w} + w_0 = 0 \\
\implies \quad & \frac{\|\underline{z}\| \|\underline{w}\|^2}{\|\underline{w}\|} + w_0 = 0 \\
\implies \quad & \|\underline{z}\| = -\frac{w_0}{\|\underline{w}\|}
\end{aligned}$$

Thus, the position vector of the vector on the hyperplane closest to the origin is:

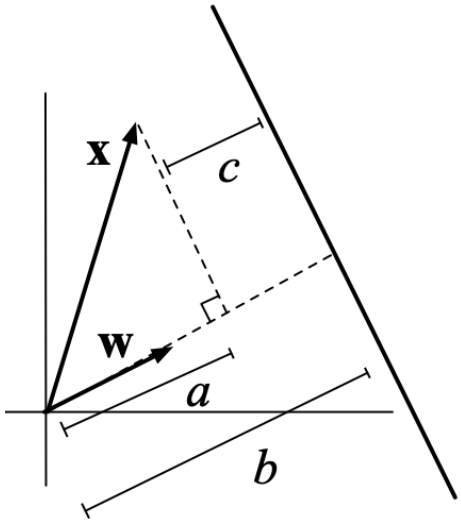
$$\underline{z} = -\frac{w_0}{\|\underline{w}\|^2} \underline{w}$$



2) Compute the distance between any training instance and the hyperplane

Let $\underline{x}$ be some training instance. It is easy to see that its (shortest) distance from $\underline{x}$ to the hyperplane is:

$$|\|\underline{z}\| - \|proj_{\underline{w}}\underline{x}\|\|$$



We know that:

$$proj_{\underline{w}}\underline{x} = \frac{\underline{w} \cdot \underline{x}}{\|\underline{w}\|^2}\underline{w}$$

So:

$$|\|\underline{z}\| - \|proj_{\underline{w}}\underline{x}\|| = \left| \frac{-w_0}{\|\underline{w}\|} - \frac{\underline{w} \cdot \underline{x}}{\|\underline{w}\|} \right| = \frac{1}{\|\underline{w}\|} |\underline{w} \cdot \underline{x} + w_0|$$

3) Define constraints which the margin must satisfy

We know that the margin must then satisfy:

$$\max_{\underline{w}} \frac{1}{\|\underline{w}\|} |\underline{w} \cdot \underline{x} + w_0|$$

The first constraint we enforce is that, if $\underline{x}$ is a training instance:

$$\min_{\underline{w}} |\underline{w} \cdot \underline{x} + w_0| = 1$$

This is always enforcible, since $\underline{w}$ and w_0 depend on the training instances, so we can always define them to satisfy this. This is a useful constraint for 2 reasons:

- both $\underline{w} \cdot \underline{x} + w_0 = 0$ and $c\underline{w} \cdot \underline{x} + cw_0 = 0$ define the exact same hyperplane. By creating the constraint we remove scalability, and ensure we can select a single hyperplane.
- if we have $\min_i |\underline{w} \cdot \underline{x} + w_0| = 1$, then we can ensure that the margin can be simply described by:

$$\frac{1}{\|\underline{w}\|}$$

But then this constraint leads to another set of constraints:

$$y_i = 1 \implies \underline{w} \cdot \underline{x} + w_0 \geq 1$$

$$y_i = -1 \implies \underline{w} \cdot \underline{x} + w_0 \leq -1$$

Which can be simplified into a single constraint:

$$y_i(\underline{w} \cdot \underline{x} + w_0) \geq 1$$

Finally, notice that:

$$\max_{\underline{w}} \frac{1}{\|\underline{w}\|}$$

is equivalent to:

$$\min_{\underline{w}} \|\underline{w}\|$$

which is equivalent to:

$$\min_{\underline{w}} \|\underline{w}\|^2$$

Thus, our optimisation problem is:

$$\min_{\underline{w}} \|\underline{w}\|^2$$

Subject to the constraint:

$$y_i(\underline{w} \cdot \underline{x} + w_0) \geq 1$$

3.2.3 Solving the Optimisation Problem

- **What are the optimal parameters for SVM?**
  - since we are optimising with respect to a constraint, we can once again use Lagrange Multipliers:

    $$\mathcal{L}(\underline{w}) = \|\underline{w}\|^2 - \sum_i \alpha_i \left(y_i(\underline{w} \cdot \underline{x}_i + w_0) - 1\right)$$

    where we have rewritten our constraint as:

    $$y_i(\underline{w} \cdot \underline{x}_i + w_0) - 1 \geq 0$$

    Each α_i corresponds to one Lagrange Multiplier for each data point $\underline{x}_i$
  - applying the method:

    $$\frac{\partial \mathcal{L}}{\partial \underline{w}} = 2\underline{w} - \sum_i \alpha_i y_i \underline{x}_i$$

    where:
    - $\frac{\partial}{\partial \underline{w}}(\|\underline{w}\|^2)$ is just $\nabla(w_1^2 + w_2^2 + \ldots + w_n^2) = 2\underline{w}$
    - $\alpha_i \left(y_i(\underline{w} \cdot \underline{x}_i + w_0) - 1\right) = \alpha_i y_i(\underline{w} \cdot \underline{x}_i + w_0) - \alpha_1 = \alpha_i y_i \underline{w} \cdot \underline{x}_i + \alpha_i y_i w_0 - \alpha_1$ so the partial derivative with respect to $\underline{w}$ will be $\alpha_i y_i \underline{x}_i$
  - so it is easy to see that our weights will be given by $\frac{\partial \mathcal{L}}{\partial \underline{w}} = 0$ so:

    $$\underline{w} = \sum_i \alpha_i y_i \underline{x}_i$$

    where we have removed the 2, since each α_i are constants
  - to get w_0, we do the same thing, but consider $\frac{\partial \mathcal{L}}{\partial w_0} = 0$, which results in:

    $$\frac{\partial \mathcal{L}}{\partial w_0} = \sum_i \alpha_i y_i = 0$$

- **How can we optimise for α_i?**
  - if we plug back in our expression for $\underline{w}$ into $\mathcal{L}$:

$$L = \frac{1}{2}\|\bar{w}\|^2 - \sum_i \alpha_i \left[y_i (\bar{w} \cdot \bar{x}_i + b) - 1 \right]$$

$$L = \frac{1}{2}\left(\sum_i \alpha_i y_i \bar{x}_i\right) \cdot \left(\sum_j \alpha_j y_j \bar{x}_j\right) - \left(\sum_i \alpha_i y_i \bar{x}_i\right) \cdot \left(\sum_j \alpha_j y_j \bar{x}_j\right) - \underbrace{\sum_i \alpha_i y_i b}_{0} + \sum_i \alpha_i$$

$$L = \sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \bar{x}_i \cdot \bar{x}_j \quad (6)$$

$$\text{st} \quad \sum \alpha_i y_i = 0 \quad \& \quad \alpha_i \geq 0$$

  - this expression is convex, so it has no local minima

- **What are support vectors?**
  - it turns out that, in the above formulation, if $\underline{x}_i$ is within the margin (so $\underline{w} \cdot \underline{x}_i + w_0 = 1$, then α_i will be non-zero. Otherwise $\alpha_i = 0$
  - in other words, $\underline{w}$ is dependent only on those vectors with $\alpha_i \neq 0$
  - these vectors are what are known as **support vectors**
  - intuitively, it makes sense that $\underline{w}$ should only depend on these support vectors, as any other training instance further away shouldn't affect the placement of the hyperplane
- **How does SVM classify a new data instance?**
  - since $\underline{w} = \sum_i \alpha_i y_i \underline{x}_i$, we classify a new instance $\underline{u}$ as class 1 if:

$$\left(\sum_i \alpha_i y_i \underline{x}_i\right) \cdot \underline{u} + w_0 \geq 0$$

  - moreover, classfication is solely dependent on support vectors

3.3 Handling Non-Separable Data With SVM: The Slack Term

- **Why can't the implementation above handle non-separable data?**
  - if the data is no separable, then our optimisation problem

$$\min_{\underline{w}} \|\underline{w}\|^2$$

    subject to the constraint:

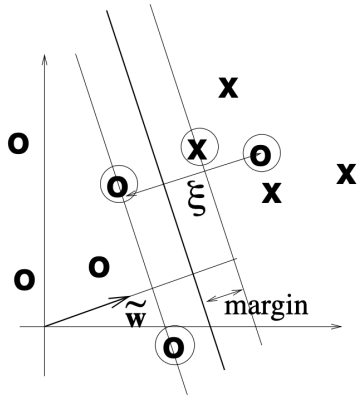
$$y_i(\underline{w} \cdot \underline{x} + w_0) \geq 1$$

    won't have a solution

- for example, if we have a positive training instance which is below the hyperplane, so $\underline{w} \cdot \underline{x} + w_0 \leq -1$, but $y_i = 1$, so it won't satisfy the constraint

- **Why can't we ignore outliers when training a SVM model?**
  - if we did this, then we'd be free of ignoring all the data points which don't get correctly classified

- **What is a slack variable?**
  - if we have non-separable data, we can make SVM "ignore" points, as to create a hyperplane which correctly classifies **most** points
  - we then use **slack variables** to quantify the effect of ignoring one training instance when building the hyperplane
  - slack variables are given by the distance ξ from the outlier to the margin that it should have



- **How do we account for slack variables in our optimisation problem?**
  - beyond making the margin as large as possible, we'd want the slack variables to be as small as possible, so we want to minimise:

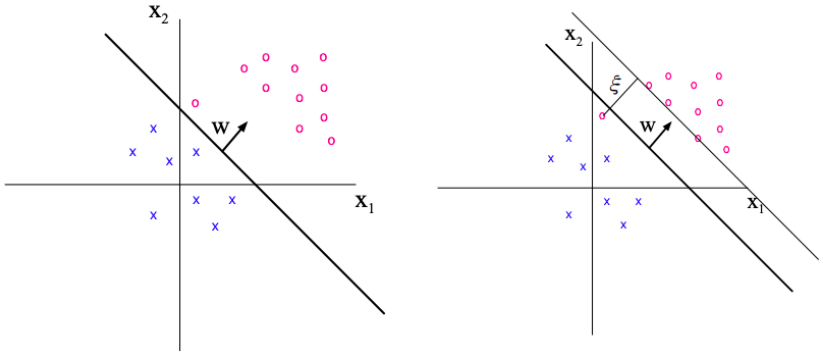
$$\|\underline{w}\|^2 + C \sum_i \xi_i^k$$

subject to some constraints, which can be naturally refactored to incorporate the slack variables:

$$y_i = 1 \implies \underline{w} \cdot \underline{x} + w_0 \geq 1 - \xi_i$$

$$y_i = -1 \implies \underline{w} \cdot \underline{x} + w_0 \leq -1 + \xi_i$$

- **How do we set k and C with slack variables?**
  - we typically use $k = 1$
  - C modulates how lenient we want to be with misclassifications: as C gets larger, we penalise misclassifications much more
- **What is the resulting SVM model, after accounting for slack variables?**
  - overall, we will end up with the same model for $\underline{w}$, but this time, the support vectors will further include all those vectors for which $\xi_i > 0$ (that is, missclassified instances)
  - intuitively, this is because by including ξ_i, we are "moving" the missclasified instance onto its margin, so it acts as if it were part of the instances which define said margin
- **How is using slack variables equivalent to using regularisation?**
  - from the optimisation formula above, we can see it is very similar to ridge regression:
    - $C \sum_i \xi_i^k$ evaluates how well we have fit the data (equivalent to the standard error/likelihood)
    - $\|\underline{w}\|^2$ penalises weight vectors with large norm
  - thus, we can view C as a **regularisation parameter** λ
- **Should we use slack variables even in separable data?**
  - it certainly doesn't hurt, and in some cases it can even help
  - for example, even if data is separable, there might be a class instance which is very separated from all other members, so it could be thought as an "outlier"; we'd get a better hyperplane if we just ignored it



3.4 Comparing SVM and Logistic Regression

- Minimize (over $\mathbf{w}$ and $\xi_i \in \mathbb{R}^+ \; \forall i$)

$$||\mathbf{w}||^2 + C \sum_i \xi_i$$

subject to

$$y_i f(\mathbf{x}_i) \geq 1 - \xi_i, \; i = 1, \ldots, n,$$

where $\xi_i \geq 0$ for all all i, and $f(\mathbf{x}_i) = \mathbf{w}^T \mathbf{x}_i + w_0$

- This optimization problem can be re-written as

$$\min_{\mathbf{w} \in \mathbb{R}^d} \; ||\mathbf{w}||^2 + C \sum_i \max(0, 1 - y_i f(\mathbf{x}_i))$$

(To see this check the three conditions where $y_i f(\mathbf{x}_i) > 1$, $y_i f(\mathbf{x}_i) = 1$ and $y_i f(\mathbf{x}_i) < 1$.)

Figure 9: We can define hinge loss as $g_h(z) = max(0, 1 - z)$

Let y labels take on values 1 or -1.

$$\begin{aligned} p(y = +1|\mathbf{x}) &= \frac{1}{1 + e^{-f(\mathbf{x})}} \\ p(y = -1|\mathbf{x}) &= 1 - \frac{1}{1 + e^{-f(\mathbf{x})}} = \frac{1}{1 + e^{f(\mathbf{x})}} \end{aligned}$$

Combining these, we have

$$p(y_i|\mathbf{x}_i) = \frac{1}{1 + e^{-y_i f(\mathbf{x}_i)}}$$

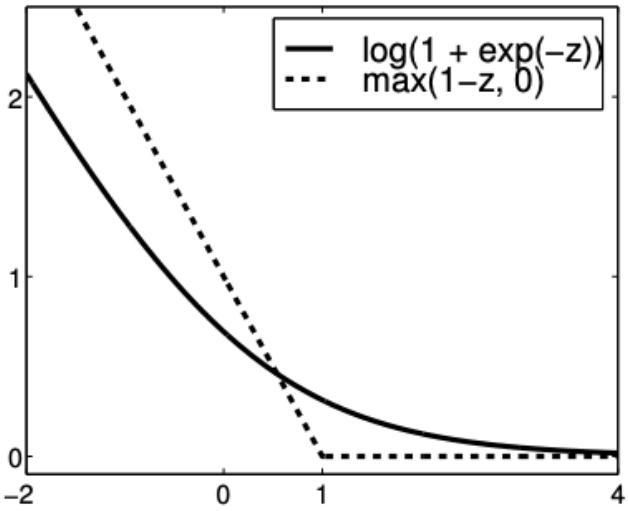
Hence the log loss is given by

$$-\log p(y_i|\mathbf{x}_i) = \log(1 + e^{-y_i f(\mathbf{x}_i)})$$

- Incorportating a ridge regression penalty, optimization problem is

$$\min_{\mathbf{w} \in \mathbb{R}^d} ||\mathbf{w}||^2 + C \sum_i \log(1 + e^{-y_i f(\mathbf{x}_i)})$$

  where $C = 1/\lambda$
- Define $g_\sigma(z) = \log(1 + e^{-z})$
- SVM classifier has a similar optimization problem, but with $g_{hinge}(z)$



3.5 Non-Linear SVMs: Using the Kernel Trick

- A very thorough, incredible,fantastic, great, lovely explanation regarding the kernel trick
- Kernel Trick explained live - easy to understand for recap
- Another one on the kernel trick
- Kernel Trick by Towards Data Science
- **What is a kernel?**
  - recall, for logistic or linear regression, we made use of basis functions to transform the non-linear data in our input space into another space, which was separable linearly
  - for SVMs we use something very similar: a **kernel**
- **What is the kernel trick?**
  - if our original input space uses vectors $\underline{x}$, we might want another space which can be transformed to via $\phi(\underline{x})$
  - now, to classify an instance $\underline{u}$, we consider the sign of:

$$\left(\sum_i \alpha_i y_i (\underline{x}_i \cdot \underline{u})\right) + w_0$$

  - so if we do the transformation to the new space, we must consider the sign of:

$$\left(\sum_i \alpha_i y_i (\phi(\underline{x}_i) \cdot \phi(\underline{u}))\right) + w_0$$

– the issue is that $\phi : \mathbb{R}^d \to \mathbb{R}^D$, so if D is too large (or even infinite), having to transform every training instance to the new coordinates, $\phi(\underline{x}_i)$, and then taking the dot product of this with $\phi(\underline{u})$ can be expensive

– the ideas is to use the **kernel trick**: instead of having to convert everything to the new coordinates, we define a convenient function $K(\underline{x}_i, \underline{x}_j)$ which can immediately give us the result of taking the dot product in $\mathbb{R}^D$ space. In other words:

$$K(\underline{x}_i, \underline{x}_j) = \phi(\underline{x}_i) \cdot \phi(\underline{x}_j)$$

but without needing to explicitly compute $\phi(\underline{x}_i)$ **or** $\phi(\underline{x}_j)$

– by using this, we are reducing the amount of computation that we have to do, and we can even handle infinite dimensional transformations (for example, the radial basis kernel is:

$$K(\underline{x}_i, \underline{x}_j) = \exp\left(-\frac{\|\underline{x}_i - \underline{x}_j\|^2}{2\sigma^2}\right)$$

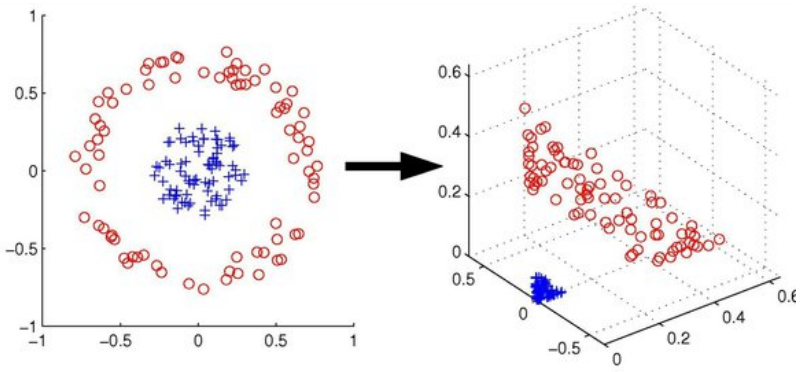
which is infinite, as it is computed by using its Taylor Expansion)

- Example 1: for 2-d input space

$$\phi(\mathbf{x}_i) = \begin{pmatrix} x_{i,1}^2 \\ \sqrt{2}x_{i,1}x_{i,2} \\ x_{i,2}^2 \end{pmatrix}$$

then

$$k(\mathbf{x}_i, \mathbf{x}_j) = (\mathbf{x}_i^\top \mathbf{x}_j)^2$$



F gure 1: Example of a labeled data nseparable n 2-D mens on s separable n 3-D mens on. *Source: [4]*

In the above example, the or g nal data s n 2-d mens on. Suppose we denote t as, $\mathbf{x} = \{x_1, x_2\}$. We can see n F g. 1 (left) that $\mathbf{x}$ s nseparable n ts space. But they are separable n th s transformed space (see F g. 1, r ght),

$$\Phi(\mathbf{x}) \rightarrow x_1^2, x_2^2, \sqrt{2}x_1x_2$$

where, Φ s a transform funct on from 2-D to 3-D appl ed on $\mathbf{x}$. These po nts can be separated w th $\Phi(\mathbf{x}) \rightarrow x_1^2, x_2^2$ transformat on as well, but the one above w ll help expla n the use of a h gher d mens on space. The $\sqrt{2}$ s not necessary but w ll make our further explanat ons *mathemat cally conven ent*.

We can now have a dec s on boundary n th s 3-D space of Φ that w ll look l ke,

$$\beta_0 + \beta_1 x_1^2 + \beta_2 x_2^2 + \beta_3 \sqrt{2} x_1 x_2 = 0 \quad (2)$$

If we were do ng log st c regress on, our model would look l ke th s Eq. 2. In SVM, a s m lar dec s on boundary (a class f er) can be found us ng the Kernel Tr ck n ts ob ect ve funct on. In short, for that we need to f nd the dot products of $\langle \Phi(\mathbf{x}_i), \Phi(\mathbf{x}_j) \rangle$ (see Eq. 13 n [6]). (Note that the dot product s also a s m lar ty measure.) Let's do that. I'll do t l ke th s,

My way:

$$\langle \Phi(\mathbf{x}_i), \Phi(\mathbf{x}_j) \rangle$$

$$= \langle \{x_{i1}^2, x_{i2}^2, \sqrt{2}x_{i1}x_{i2}\}, \{x_{j1}^2, x_{j2}^2, \sqrt{2}x_{j1}x_{j2}\} \rangle \quad (3.1)$$

$$= x_{i1}^2 x_{j1}^2 + x_{i2}^2 x_{j2}^2 + 2x_{i1}x_{i2}x_{j1}x_{j2} \ (3.2)$$

Instead, my fr end Sam, who s smarter, d d the follow ng,

Sam's way:

$$\langle \mathbf{x}_i, \mathbf{x}_j \rangle^2$$

$$= \langle \{x_{i1}, x_{i2}\}, \{x_{j1}, x_{j2}\} \rangle^2$$

$$= (x_{i1}x_{j1} + x_{i2}x_{j2})^2 \quad (4.1)$$

$$= x_{i1}^2 x_{j1}^2 + x_{i2}^2 x_{j2}^2 + 2x_{i1}x_{i2}x_{j1}x_{j2} \ (4.2)$$

Figure 10: If we transformed to the new space and then computed the dot product we would need to: convert the 2 variables into the new space (3 elements), and then compute the dot product for 3 terms. I other words, 6 + 3 = 9 operations. By using the kernel trick, we just need to do a dot product of two elements in the original coordinates, and just square the result. Overall, 3 operations.

- **What does the kernel represent?**
  - we can interpret $K(\underline{x}_i, \underline{x}_j)$ as giving a measure of the relationship of $\underline{x}_i$ and $\underline{x}_j$ in the new feature space. In particular, it is a measure of "distance" in the feature space.
- **How can we use kernels when optimising max margins?**
  - as shown above, since the kernel represents the dot product in the new feature space, we get that:

    $$\underline{w} = \sum_i \alpha_i y_i \phi(\underline{x}_i)$$

    and our prediction is given by the sign of:

    $$\left(\sum_i \alpha_i y_i (\phi(\underline{x}_i) \cdot \phi(\underline{u})) \right) + w_0$$

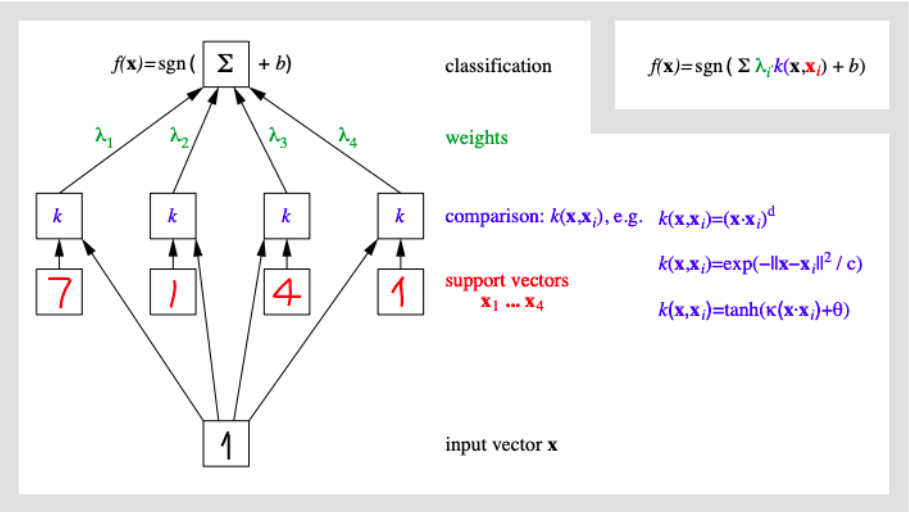


Figure 11: Example of using kernels and support vectors to classify numbers.