Lecture 3: Linear classification, generalisation and regularisation

EMAT31530/Jan 2018/Raul Santos-Rodriguez

Have a look at ...

... Russell and Norvig (Ch. 18.2 18.4 18.6 18.7 18.8 18.9)

... Hastie, Tibshirani, Friedman. The elements of statistical learning, (Ch. 4.5 and 7)

... Python: `https://keras.io/`

... Python: `http://scikit-learn.org/`

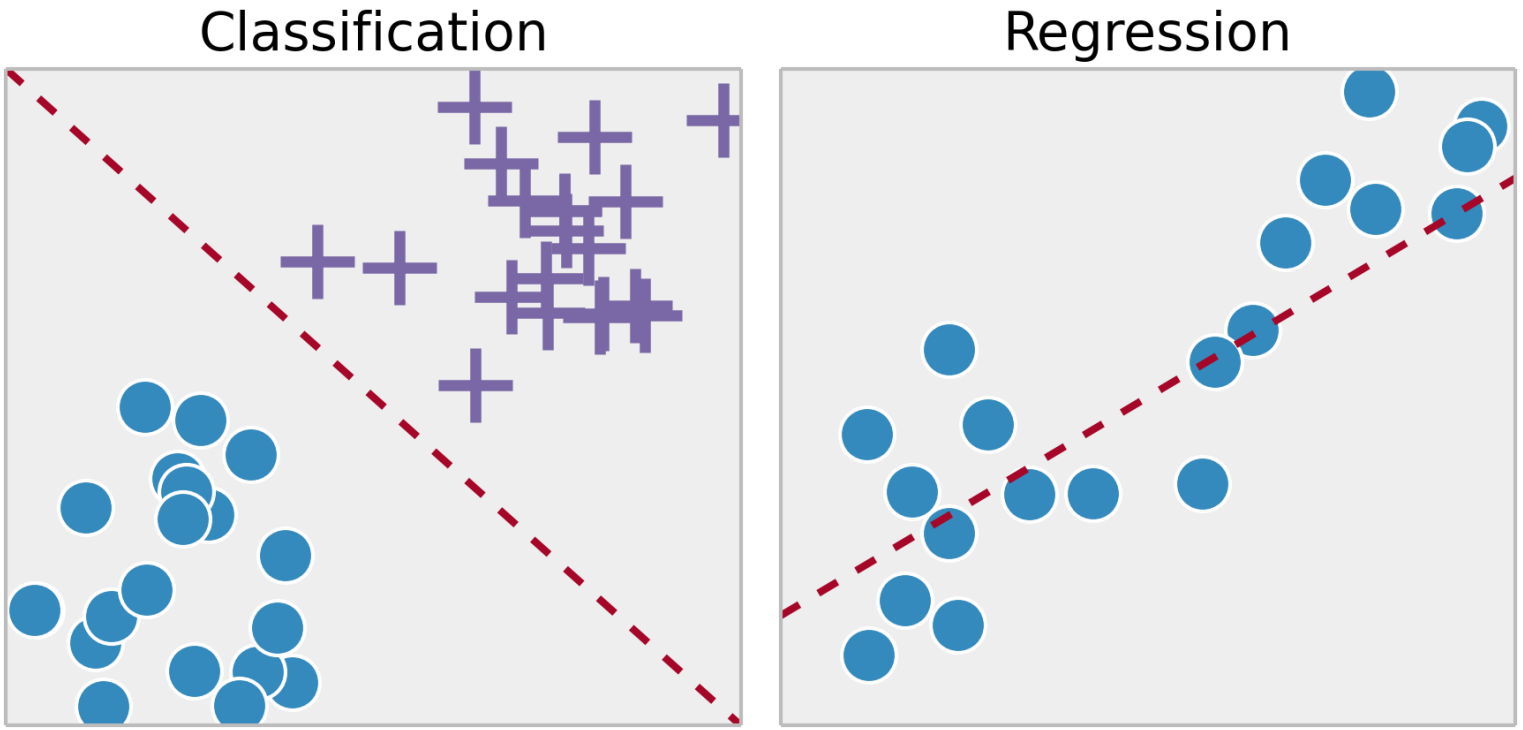
... Java: `http://www.cs.waikato.ac.nz/ml/weka/`

Outline

This lecture introduces the topics of linear classification, generalisation and regularisation. The goal of the lecture is for you to

- Understand the linear classification setting
- Understand the trade-off between bias and variance
- Understand how to make the most of your data

Classification vs Regression



Classification vs Regression

We are given a training dataset D_{train} of n instances of input-output pairs $\{\mathbf{x}_{1:n}, y_{1:n}\}$. Each input $\mathbf{x}_i \in \mathbb{R}^{d \times 1}$ is a vector with d attributes. The output (target) is y_i. Depending on the nature of y_i, there are different types of prediction tasks:

- Regression: y is a real number

$$y = \mathbf{x}^T \boldsymbol{\theta}$$

- Classification: y is discrete; yes/no (binary), one of K labels (multiclass), subset of K labels (multilabel)

$$y = sign(\mathbf{x}^T \boldsymbol{\theta})$$

Binary classification: $y \in \{+1, -1\}$

Score

Measures how confident we are in our prediction: $score = f_\theta(\mathbf{x}) = \mathbf{x}^T\theta$

Margin

Measures how correct we are: $margin = y(\mathbf{x}^T\theta)$

Loss function

Loss function $L(y, f_\theta(\mathbf{x}))$ measures how happy we are with the prediction

Linear classification

Binary classification: $y \in \{+1, -1\}$

Score

Measures how confident we are in our prediction: $score = f_\theta(\mathbf{x}) = \mathbf{x}^T\theta$

Margin

Measures how correct we are: $margin = y(\mathbf{x}^T\theta)$

Loss function

Loss function $L(y, f_\theta(\mathbf{x}))$ measures how happy we are with the prediction

- We define the prediction **score** to be the inner product of the input and the weights. The score intuitively represents the degree to which the classification is positive or negative. Geometric intuition: a classifier $f_{\boldsymbol{\theta}}$ defines a hyperplane with normal vector $\boldsymbol{\theta}$. Points with $f_{\boldsymbol{\theta}} = 0$ define the **decision boundary**.
- Suppose the correct label is $y \in \{+1, -1\}$. The **margin** of an input $\mathbf{x}$ is $(\mathbf{x}^T\boldsymbol{\theta})y$, which measures how correct the prediction that $\boldsymbol{\theta}$ makes is. The larger the margin, the better, and non-positive margins correspond to classification errors. Note that if we look at the actual prediction $f_{\boldsymbol{\theta}}(\mathbf{x})$, we can only ascertain whether the prediction was right or not. By looking at the score and the margin, we can get a more nuanced view onto the behaviour of the classifier. Geometrically, if $||\boldsymbol{\theta}|| = 1$, then the margin of an input $\mathbf{x}$ is exactly the distance from $\mathbf{x}$ to the decision boundary.
- A **loss function** $L(y, f_{\boldsymbol{\theta}}(\mathbf{x}))$ measures how unhappy you would be if you used $\boldsymbol{\theta}$ to make a prediction on $\mathbf{x}$ when the correct output is y.

Question

When does a binary classifier err on an example?

- margin less than 0
- margin greater than 0
- score less than 0
- score greater than 0

First approach: Perceptron

The Perceptron Algorithm

1. Initialise t (number of iterations) to 1 and start with the all-zeroes weight vector $\boldsymbol{\theta}^1 = [0 \ldots 0]^T$. Also let's automatically scale all examples $\mathbf{x}$ to have (Euclidean) length 1, since this doesn't affect which side of the hyperplane they are on.

2. Given example $\mathbf{x}$, predict positive iff $\mathbf{x}^T\boldsymbol{\theta} > 0$.

3. On a mistake, update as follows:

   - Mistake on positive: $\boldsymbol{\theta}^{t+1} \leftarrow \boldsymbol{\theta}^t + \mathbf{x}$.
   - Mistake on negative: $\boldsymbol{\theta}^{t+1} \leftarrow \boldsymbol{\theta}^t - \mathbf{x}$.

   $t \leftarrow t + 1$.

`http://intelligence.org/files/AIPosNegFactor.pdf`, Sec. 7.2

[Rosenblatt, 1958]

First approach: Perceptron

If we make a mistake on a positive $\mathbf{x}$ we get

$$\mathbf{x}^T\boldsymbol{\theta}^{t+1} = \mathbf{x}^T(\boldsymbol{\theta}^t + \mathbf{x}) = \mathbf{x}^T\boldsymbol{\theta}^t + 1$$

Similarly if we make a mistake on a negative $\mathbf{x}$ we have

$$\mathbf{x}^T\boldsymbol{\theta}^{t+1} = \mathbf{x}^T(\boldsymbol{\theta}^t - \mathbf{x}) = \mathbf{x}^T\boldsymbol{\theta}^t - 1$$

In both cases we move closer (by 1) to the value we wanted.

Several problems:

- Convergence: what if the data is not linearly separable?
- Which linear hyperplane is best?

First approach: Perceptron

The Perceptron Algorithm

1. Initialise t (number of iterations) to 1 and start with the all-zeroes weight vector $\boldsymbol{\theta}^1 = [0 \dots 0]^T$. Also let's automatically scale all examples $\mathbf{x}$ to have (Euclidean) length 1, since this doesn't affect which side of the hyperplane they are on.
2. Given example $\mathbf{x}$, predict positive iff $\mathbf{x}^T\boldsymbol{\theta} > 0$.
3. On a mistake, update as follows:
   - Mistake on positive: $\boldsymbol{\theta}^{t+1} \leftarrow \boldsymbol{\theta}^t + \mathbf{x}$.
   - Mistake on negative: $\boldsymbol{\theta}^{t+1} \leftarrow \boldsymbol{\theta}^t - \mathbf{x}$.

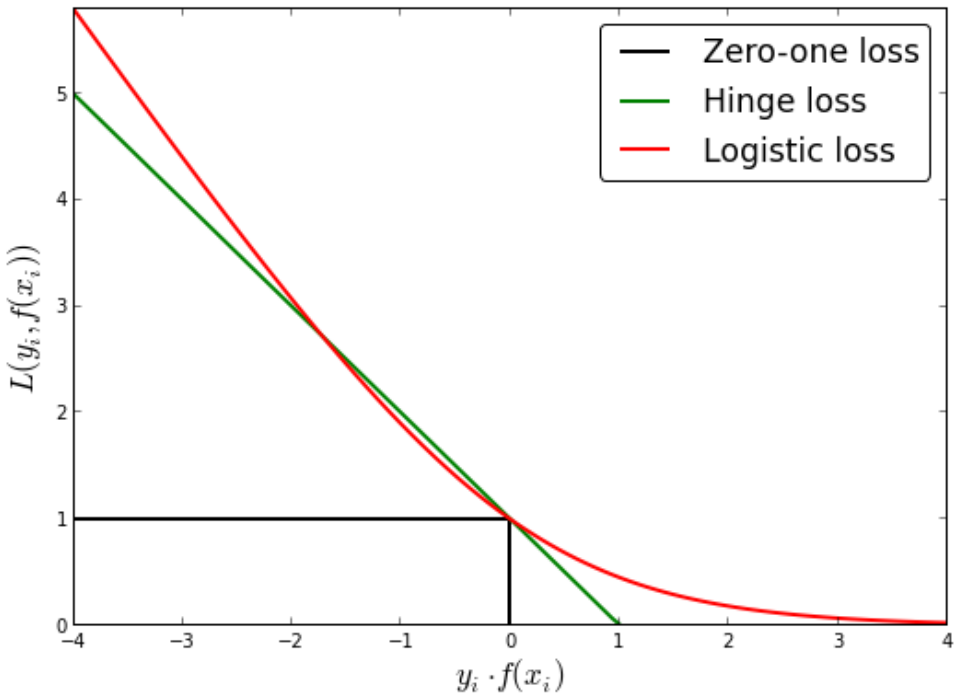
   $t \leftarrow t + 1$.

http://intelligence.org/files/AIPosNegFactor.pdf, Sec. 7.2

[Rosenblatt, 1958]

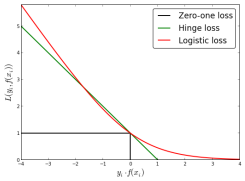
A general framework: Loss minimisation

Zero-one loss: $L_{0-1}(y, f_\theta(\mathbf{x})) = \mathbf{1}[(\mathbf{x}^T\theta)y \leq 0]$



A general framework: Loss minimisation

Zero-one loss: $L_{0-1}(y, f_{\theta}(\mathbf{x})) = \mathbf{1}[(\mathbf{x}^T\boldsymbol{\theta})y \leq 0]$



Zero-one loss. This corresponds exactly to our familiar notion of whether our predictor made a mistake or not. We can also write the loss in terms of the margin. We can plot the loss as a function of the margin. From the graph, it is clear that the loss is 1 when the margin is negative and 0 when it is positive. But the main problem with zero-one loss is that it is hard to optimise (in fact, it is probably NP hard in the worst case). And in particular, we cannot apply gradient-based optimisation to it, because the gradient is zero (almost) everywhere. Solution: **surrogate losses**. Examples:

Hinge loss

$$L_{hinge}(y, f_{\boldsymbol{\theta}}(\mathbf{x})) = \max(0, 1 - y f_{\boldsymbol{\theta}}(\mathbf{x}))$$

Logistic loss

$$L_{logistic}(y, f_{\boldsymbol{\theta}}(\mathbf{x})) = \log(1 + \exp(-y f_{\boldsymbol{\theta}}(\mathbf{x})))$$

Loss minimisation

$$\arg\min_{\boldsymbol{\theta}} \mathit{TrainLoss}(\boldsymbol{\theta})$$

where

$$\mathit{TrainLoss}(\boldsymbol{\theta}) = \sum_{(\mathbf{x},y)\in D_{train}} L(y, f_{\boldsymbol{\theta}}(\mathbf{x}))$$

Gradient Descent

Gradient

The gradient $\nabla \mathit{TrainLoss}(\boldsymbol{\theta})$ is the direction that increases the loss the most.

Gradient Descent (GD)

$\boldsymbol{\theta} \leftarrow$ any point in the parameter space, e.g., $\boldsymbol{\theta} = [0 \ldots 0]^T$

do

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla \mathit{TrainLoss}(\boldsymbol{\theta})$$

until convergence **or** $t \leq \mathit{max\_iterations}$

Problem: each iteration requires going over all training examples (expensive when have lots of data!)

Gradient Descent

Gradient

The gradient $\nabla TrainLoss(\boldsymbol{\theta})$ is the direction that increases the loss the most.

Gradient Descent (GD)

$\boldsymbol{\theta} \leftarrow$ any point in the parameter space, e.g., $\boldsymbol{\theta} = [0 \dots 0]^T$

do

$\quad \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla TrainLoss(\boldsymbol{\theta})$

until convergence **or** $t \leq max\_iterations$

Problem: each iteration requires going over all training examples (expensive when have lots of data!)

A general approach is to use iterative optimisation, which essentially starts at some starting point $\boldsymbol{\theta}$ (say, all zeros), and tries to tweak $\boldsymbol{\theta}$ so that the objective function value decreases.

To do this, we will rely on the gradient of the function, which tells us which direction to move in to decrease the objective the most. The gradient is a valuable piece of information, especially since we will often be optimising in high dimensions (d on the order of thousands). This iterative optimisation procedure is called **gradient descent**. Gradient descent has two hyperparameters, the step size α (which specifies how aggressively we want to pursue a direction) and the number of iterations max_it. We can now apply gradient descent on any of our objective functions that we defined before and have a working algorithm. But it is not necessarily the best algorithm. One problem (but not the only problem) with gradient descent is that it is slow. Recall that the training loss is a sum over the training data. If we have one million training examples (which is, by today's standards, only a modest number), then each gradient computation requires going through those one million examples, and this must happen before we can make any progress. Can we make progress before seeing all the data?

Stochastic gradient descent

Gradient Descent (GD)

$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla \mathit{TrainLoss}(\boldsymbol{\theta})$

Stochastic Gradient Descent (SGD)

For each $(\mathbf{x}, y) \in D_{train}$

$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla L(\boldsymbol{\theta}, \mathbf{x}, y)$

Quantity, not quality!

Stochastic gradient descent

Gradient Descent (GD)

$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla \mathit{TrainLoss}(\boldsymbol{\theta})$

Stochastic Gradient Descent (SGD)

For each $(\mathbf{x}, y) \in D_{train}$

$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla L(\boldsymbol{\theta}, \mathbf{x}, y)$

Quantity, not quality!

Rather than looping through all the training examples to compute a single gradient and making one step, SGD loops through the examples $(\mathbf{x}, y)$ and updates the weights $\boldsymbol{\theta}$ based on each example. Each update is not as good because we are only looking at one example rather than all the examples, but we can make many more updates this way. In practice, we often find that just perform one pass over the training examples with SGD, touching each example once, often performs comparably to taking ten passes over the data with GD.

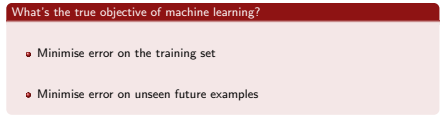
The advantages of Stochastic Gradient Descent are:

- Efficiency.
- Ease of implementation (lots of opportunities for code tuning).

Question

What's the true objective of machine learning?

- Minimise error on the training set
- Minimise error on unseen future examples



So far in this class, we have tried to cast everything as a well-defined optimisation problem. We have even written down an objective function, which is the sum loss (error) on the training data. But it turns out that that is not really the true goal. That is only what we tell our optimisation algorithms so that there is something concrete and actionable. The true goal is to minimise error on unseen future examples; in other words, we need to **generalise**. As we will see, this is perhaps the most important aspect of machine learning and statistics.

Training error

Loss minimisation

$$\arg\min_{\boldsymbol{\theta}} \mathit{TrainLoss}(\boldsymbol{\theta})$$

where

$$\mathit{TrainLoss}(\boldsymbol{\theta}) = \sum_{(\mathbf{x},y)\in D_{train}} L(y, f_{\boldsymbol{\theta}}(\mathbf{x}))$$

Is this a good objective?

Training error

Loss minimisation

$$\arg\min_{\theta} TrainLoss(\theta)$$

where

$$TrainLoss(\theta) = \sum_{(\mathbf{x},y)\in D_{train}} L(y, f_\theta(\mathbf{x}))$$

Is this a good objective?

Now let's be a little more critical about what we have set out to optimise. So far, we have declared that we want to minimise the training loss.

Example

A classifier that minimises the training error

1. **During training**: store all pairs in D_{train}
2. **During prediction**: predict the output for x_{new}

   if x_{new} in D_{train} **then return** its corresponding y

   else return error!

Minimises the objective perfectly (zero), but ...

A classifier that minimises the training error

- **During training**: store all pairs in D_{train}
- **During prediction**: predict the output for x_{new}
  if x_{new} in D_{train} **then return** its corresponding y
  else return error!

Minimises the objective perfectly (zero), but ...

It overfits to the training data and doesn't generalise to unseen examples.

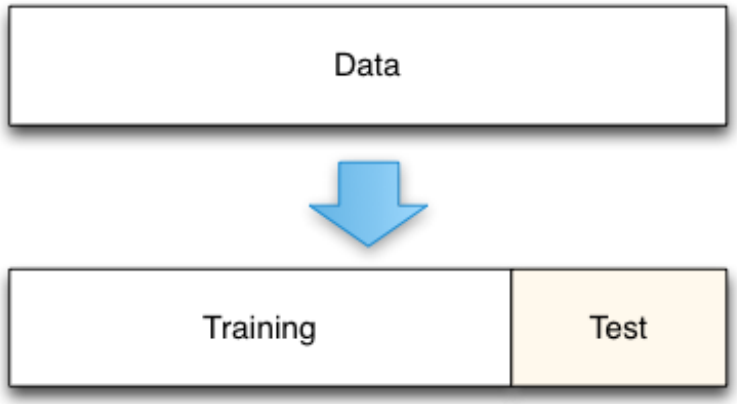
Test set

Key idea: our goal is to minimise error on unseen future examples..

... but we dont have unseen examples. The next best thing:

Test set

Test set D_{test} contains examples not used for training.



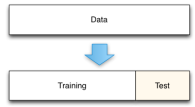
Test set

Key idea: our goal is to minimise error on unseen future examples..

... but we dont have unseen examples. The next best thing:

Test set

Test set D_{test} contains examples not used for training.

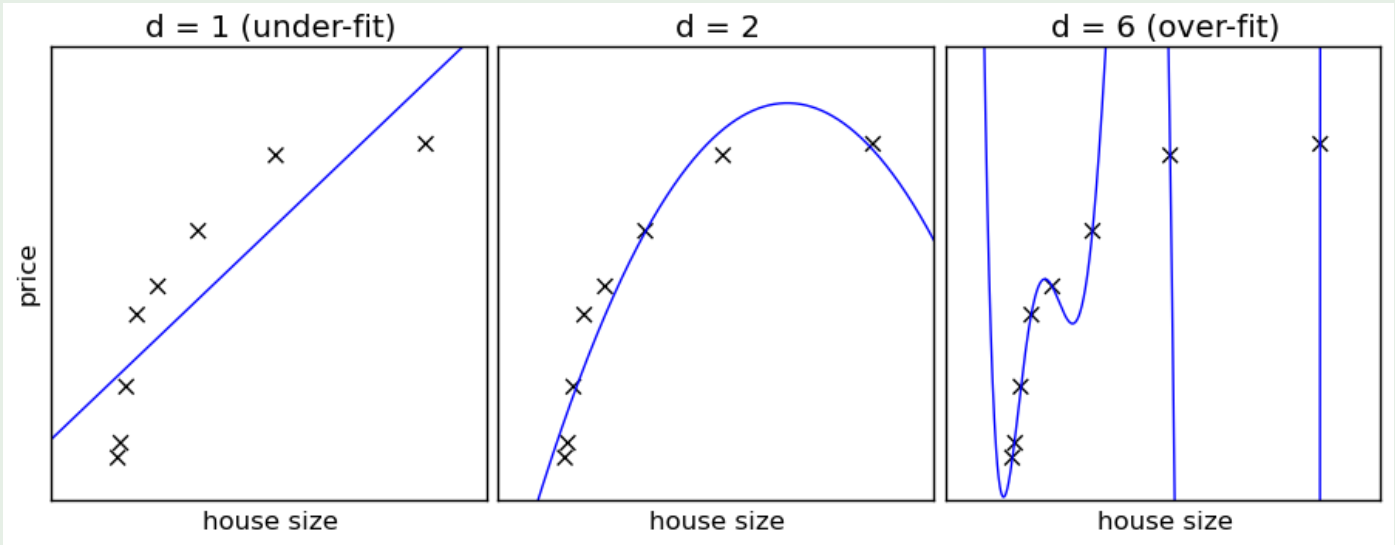


What we really care about is how accurate that system is on those unseen future inputs. Of course, we can't access unseen future examples, so the next best thing is to create a **test set**. As much as possible, we should treat the test set as a pristine thing that is unseen and from the future. We definitely should not tune our predictor based on the test error, because we wouldn't be able to do that on future examples. Of course at some point we have to run our algorithm on the test set, but just be aware that each time this is done, the test set becomes less good of an indicator of how well you are actually doing.

Overfitting example: regression

House prices

Imagine that you would like to build an algorithm which will predict the price of a house given its size. Naively, we'd expect that the cost of a house grows as the size increases, but there are many other factors which can contribute. Imagine we approach this problem with polynomial regression. We can tune the degree d to try to get the best fit.

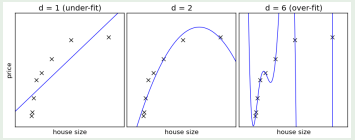


[sklearn tutorial]

Overfitting example: regression

House prices

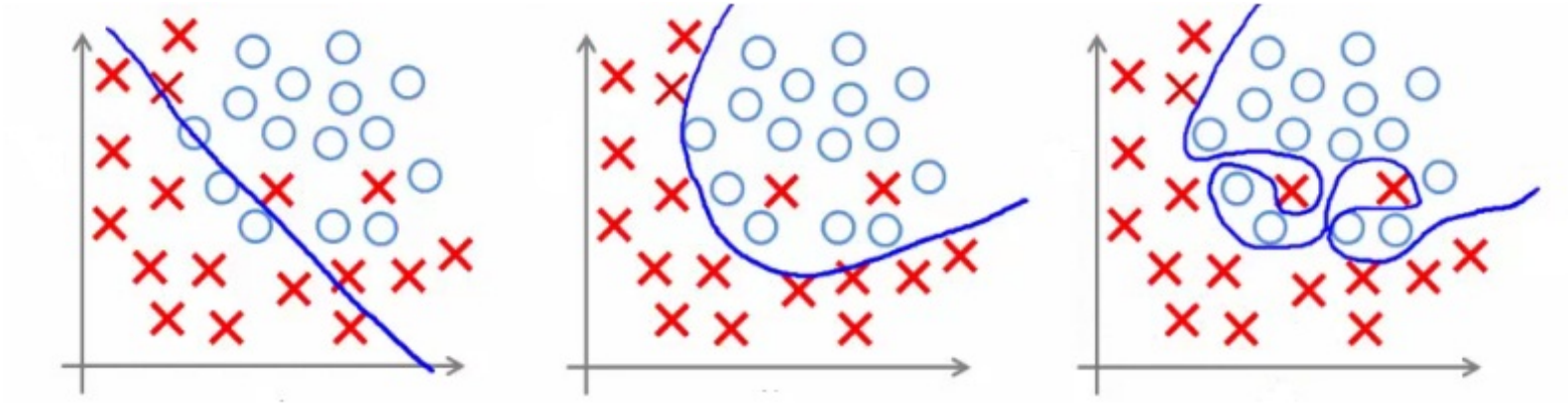
Imagine that you would like to build an algorithm which will predict the price of a house given its size. Naively, we'd expect that the cost of a house grows as the size increases, but there are many other factors which can contribute. Imagine we approach this problem with polynomial regression. We can tune the degree d to try to get the best fit.



[sklearn tutorial]

- For $d = 1$, the data is **underfit**. This means that the model is too simplistic: no straight line will ever be a good fit to this data. In this case, we say that the model suffers from high bias. The model itself is biased, and this will be reflected in the fact that the data is poorly fit.
- At the other extreme, for $d = 6$ the data is **overfit**. This means that the model has too many free parameters (6 in this case) which can be adjusted to perfectly fit the training data. If we add a new point to this plot, though, chances are it will be very far from the curve representing the degree-6 fit. In this case, we say that the model suffers from high variance. The reason for this label is that if any of the input points are varied slightly, it could result in an extremely different model.
- In the middle, for $d = 2$, we have found a good mid-point. It fits the data fairly well, and does not suffer from the bias and variance problems seen in the figures on either side.

Overfitting example: classification



Under-fitting

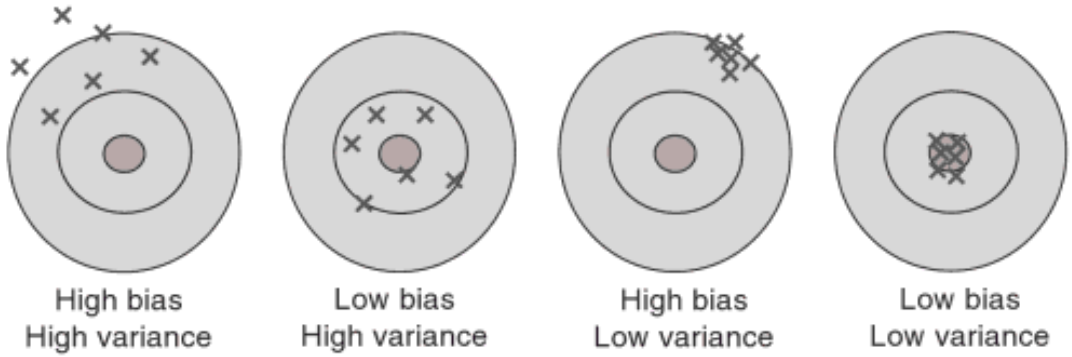
(too simple to explain the variance)

Appropriate-fitting

Over-fitting

(forcefitting -- too good to be true)

Bias and Variance: dartboard analogy



Bias Variance Decomposition. Figure 1. The bias-variance decomposition is like trying to hit the bullseye on a dartboard. Each dart is thrown after training our "dart-throwing" model in a slightly different manner. If the darts vary wildly, the learner is *high variance*. If they are far from the bullseye, the learner is *high bias*. The ideal is clearly to have both low bias and low variance; however this is often difficult, giving an alternative terminology as the bias-variance "dilemma" (*Dartboard analogy*, Moore & McCabe (2002))

Bias: how good is the hypothesis class?

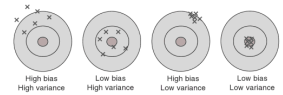
Variance: how good is the learned predictor relative to the hypothesis class?

Test error is bias + variance

[Encyclopedia of Machine Learning]

Bias and Variance: dartboard analogy

Bias and Variance: dartboard analogy



Bias Variance Decomposition. Figure 1. The bias-variance decomposition is like trying to hit the bullseye on a dartboard. Each dart is thrown after training our "dart-throwing" model in a slightly different manner. If the darts vary wildly, the learner is *high variance*. If they are far from the bullseye, the learner is *high bias*. The ideal is clearly to have both low bias and low variance; however this is often difficult, giving an alternative terminology as the bias-variance "dilemma" (*Dartboard analogy*, Moore & McCabe (2002))

Bias: how good is the hypothesis class?

Variance: how good is the learned predictor relative to the hypothesis class?

Test error is bias + variance

[Encyclopedia of Machine Learning]

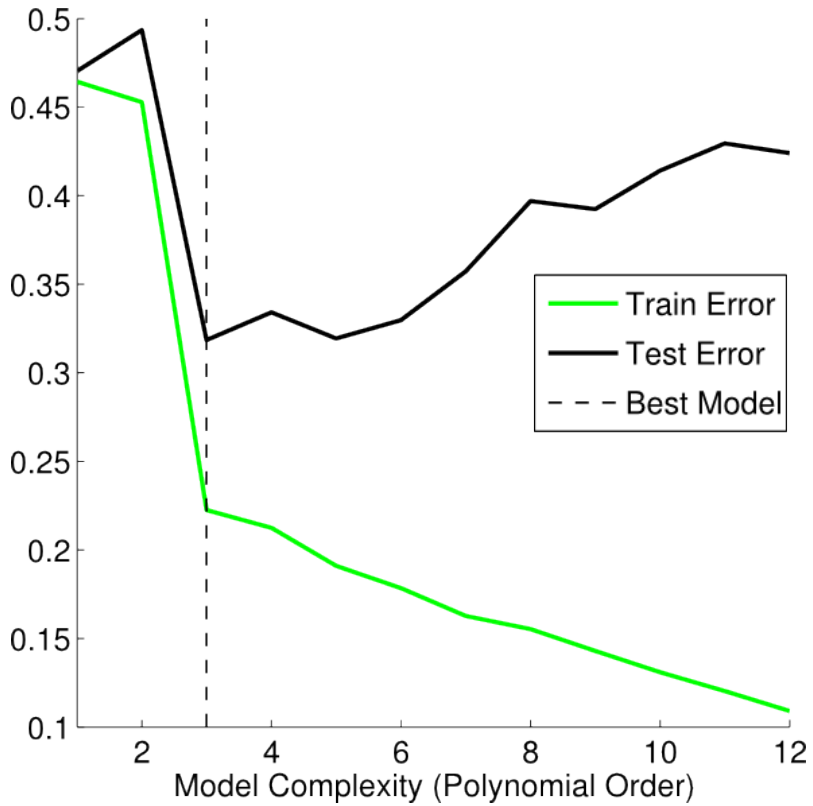
When will a learning algorithm generalise well? In particular, when does a learning algorithm generalise from the training set to the test set?
Beyond intrinsic uncertainty/noise in the data, any learning algorithm has error that comes from two sources:

- **Bias** is the algorithm's tendency to consistently learn the wrong thing by not taking into account all the information in the data (underfitting).
- **Variance** is the algorithm's tendency to learn random things irrespective of the real signal by fitting highly flexible models that follow the error/noise in the data too closely (overfitting).

$$error = bias + variance$$

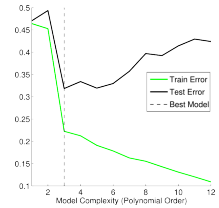
Bias is also used to denote by how much the average accuracy of the algorithm changes as input/training data changes. Similarly, variance is used to denote how sensitive the algorithm is to the chosen input data. In other words, the bias of a model quantifies how precise a model is across training sets. The variance quantifies how sensitive the model is to small changes in the training set. A robust model is not overly sensitive to small changes. The dilemma involves minimising both bias and variance; we want a precise and robust model. Simpler models tend to be less accurate but more robust. Complex models tend to be more accurate but less robust.

Training error and test error



Underfitting: bias too high
Overfitting: variance too high
Fitting: balancing bias and variance

Training error and test error



Underfitting: bias too high
Overfitting: variance too high
Fitting: balancing bias and variance

Another way to visualise generalisation is by looking at how the various errors vary as a function of the size of the hypothesis class. When the hypothesis class is too small (small polynomial degree from the housing prices example), then the bias is large, but the variance is small. If the hypothesis class is too large (large polynomial degree), then the bias is very small, but the variance is large. The ideal hypothesis class is one where we balance bias and variance, and we get a nice fit.

Controlling the size of the hypothesis class

Linear predictors are specified by parameter vector $\boldsymbol{\theta} \in \mathbb{R}^d$.

Two alternatives:

1. **Keep the dimensionality d small**
2. **Keep the norm $||\theta||$ (length of θ) small: Regularisation**

Controlling the size of the hypothesis class

Linear predictors are specified by parameter vector $\boldsymbol{\theta} \in \mathbb{R}^d$.

Two alternatives:

1. **Keep the dimensionality d small**
2. **Keep the norm $||\boldsymbol{\theta}||$ (length of $\boldsymbol{\theta}$) small: Regularisation**

So the hypothesis class $\mathcal{F} = \{f_{\boldsymbol{\theta}}\}$ is all the predictors as $\boldsymbol{\theta}$ ranges. By controlling the number of possible values of $\boldsymbol{\theta}$ that the learning algorithm is allowed to choose from, we control the size of the hypothesis class and thus guard against overfitting.

Controlling the size of the hypothesis class: Dimensionality

Manual feature selection:

- Add features if they reduce test error
- Remove features if they don't decrease test error

Automatic feature selection (beyond the scope of this class):

- Forward selection: Maximum Relevance Minimum Redundancy (MRMR)
- L1 regularisation: Lasso

Controlling the size of the hypothesis class: Dimensionality

Manual feature selection:

- Add features if they reduce test error
- Remove features if they don't decrease test error

Automatic feature selection (beyond the scope of this class):

- Forward selection: Maximum Relevance Minimum Redundancy (MRMR)
- L1 regularisation: Lasso

The most intuitive way to reduce overfitting is to reduce the number of features (or feature templates). Mathematically, you can think about removing a feature as simply only allowing its corresponding weight to be zero.

Controlling the size of the norm: Regularisation

Regularised objective

$$\min_{\boldsymbol{\theta}} \mathit{TrainLoss}(\boldsymbol{\theta}) + \frac{\lambda}{2}||\boldsymbol{\theta}||^2$$

Idea: shrink the weights towards zero by λ

Controlling the size of the norm: Regularisation

Controlling the size of the norm: Regularisation

Regularised objective

$$\min_{\theta} \mathit{TrainLoss}(\theta) + \frac{\lambda}{2}||\theta||^2$$

Idea: shrink the weights towards zero by λ

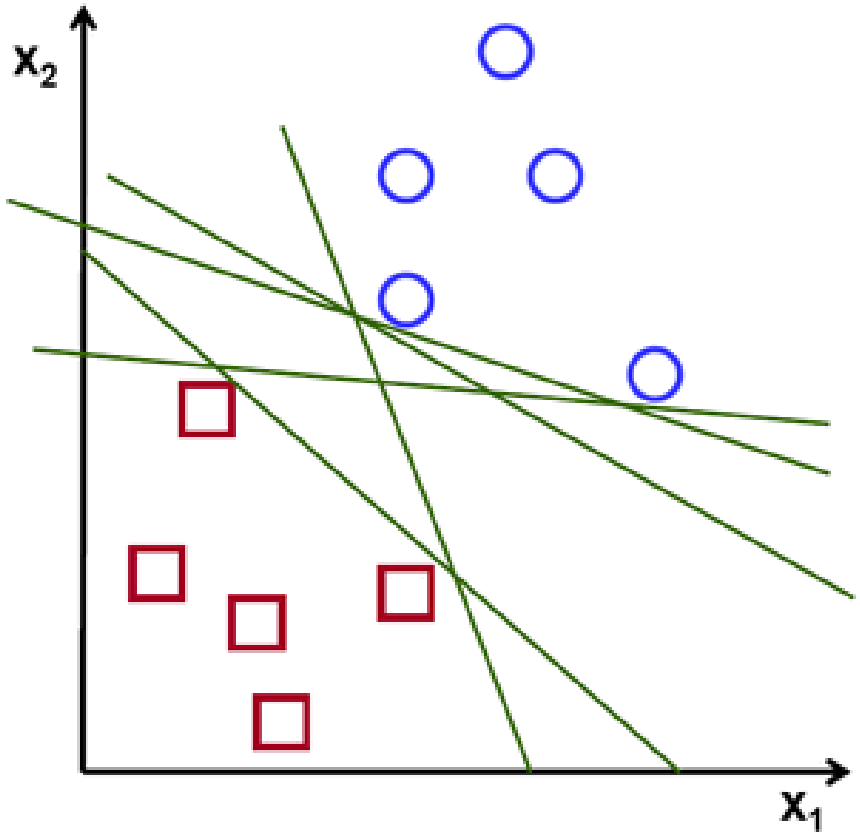
A related way to keep the weights small is called **regularisation**, which involves adding an additional term to the objective function which penalises the norm (length) of $\boldsymbol{\theta}$. This is probably the most common way to control the norm.

Controlling the size of the norm: Support Vector Machines

Support Vector Machines

Idea: Large margin

Objective: *HingeLoss* + *regularisation*

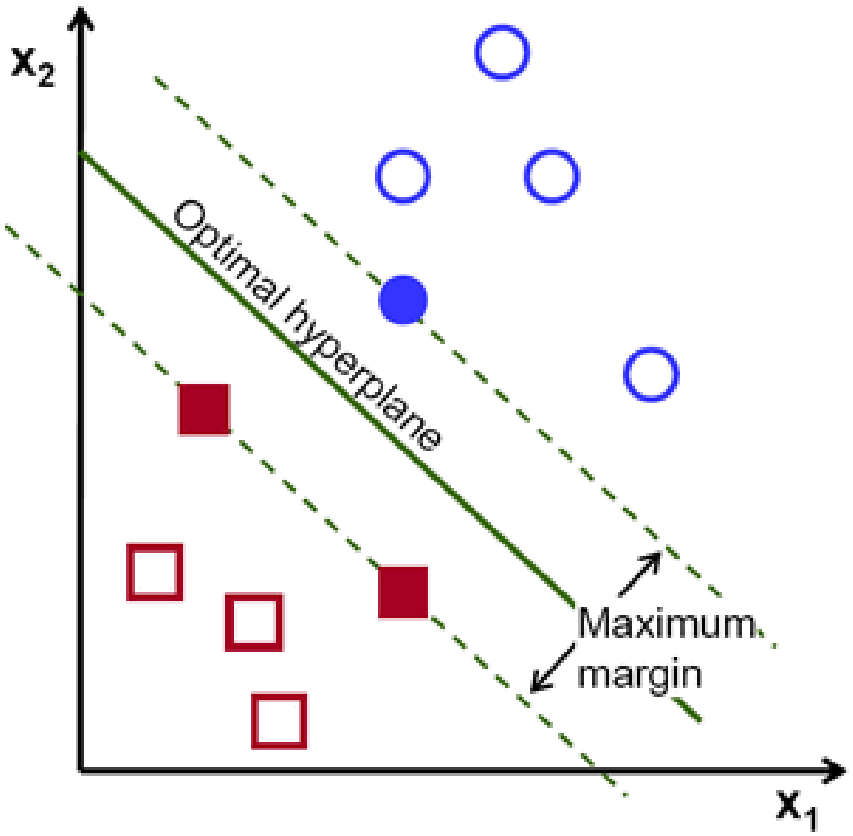


Controlling the size of the norm: Support Vector Machines

Support Vector Machines

Idea: Large margin

Objective: *HingeLoss* + *regularisation*



Controlling the size of the norm: Ridge regression

In regression, all the answers so far are of the form

$$\hat{\boldsymbol{\theta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

They require the inversion of $\mathbf{X}^T\mathbf{X}$. This can lead to problems if the system of equations is poorly conditioned. A solution is to add a small element to the diagonal:

$$\hat{\boldsymbol{\theta}} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I}_d)^{-1}\mathbf{X}^T\mathbf{y}$$

This is the ridge regression estimate. It is the solution to the following regularised quadratic cost function

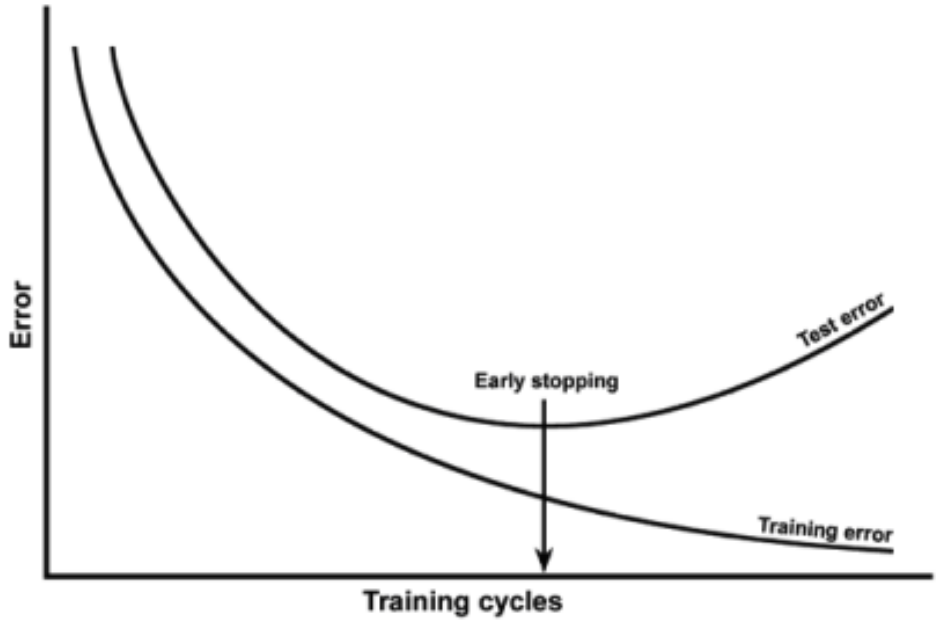
$$J(\boldsymbol{\theta}) = (\mathbf{y} - \mathbf{X}\boldsymbol{\theta})^T(\mathbf{y} - \mathbf{X}\boldsymbol{\theta}) + \lambda\boldsymbol{\theta}^T\boldsymbol{\theta}$$

Controlling the size of the norm: Early stopping

Idea simply make *max_it* smaller.

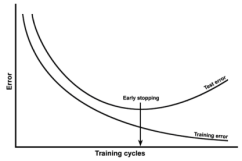
Intuition fewer updates, then $||\boldsymbol{\theta}||$ can't get too big.

Lesson try to minimise the training error, but don't try too hard.



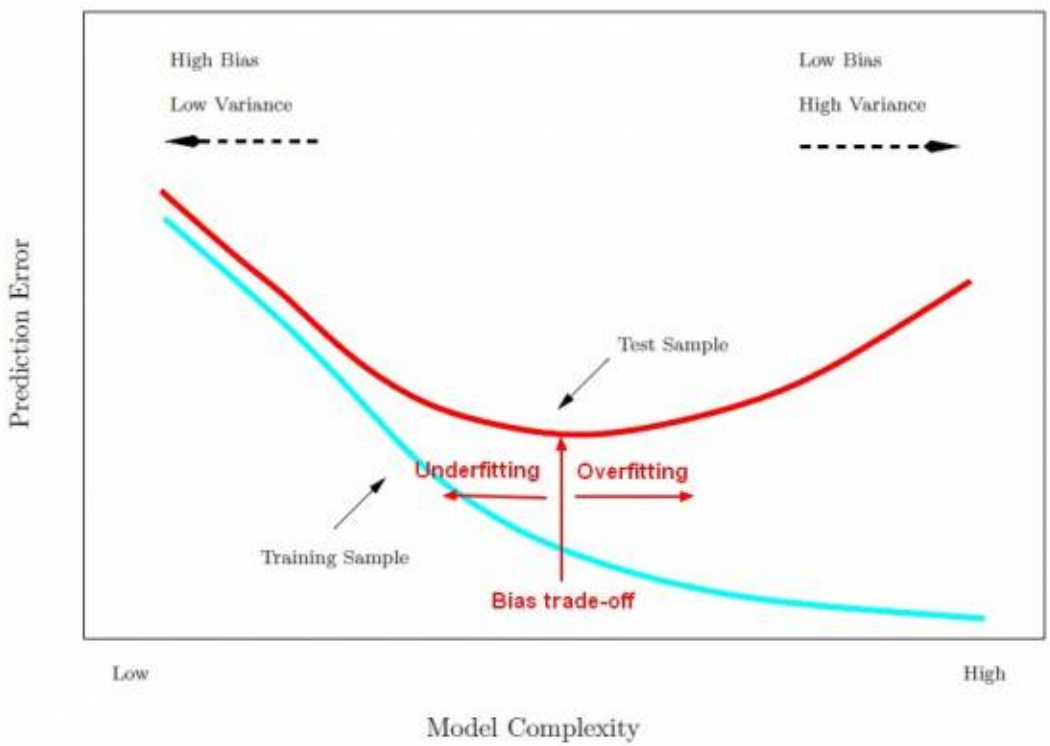
Controlling the size of the norm: Early stopping

Idea simply make *max_it* smaller.
Intuition fewer updates, then $||\theta||$ can't get too big.
Lesson try to minimise the training error, but don't try too hard.



A really cheap way to keep the weights small is to do **early stopping**. As we run more iterations of gradient descent, the objective function improves. If we cared about the objective function, this would always be a good thing. However, our true objective is not the training loss.

Summary so far



Simple solutions that fit the data well

How do we choose the hyperparameters

Hyperparameters

Properties of the learning algorithm (features, regularisation parameter λ, number of iterations *max_it*, step size α, etc.).

Choose hyperparameters to minimise D_{train} error? No - solution would be to include all features, set $\lambda = 0$, $max\_it \to \infty$.

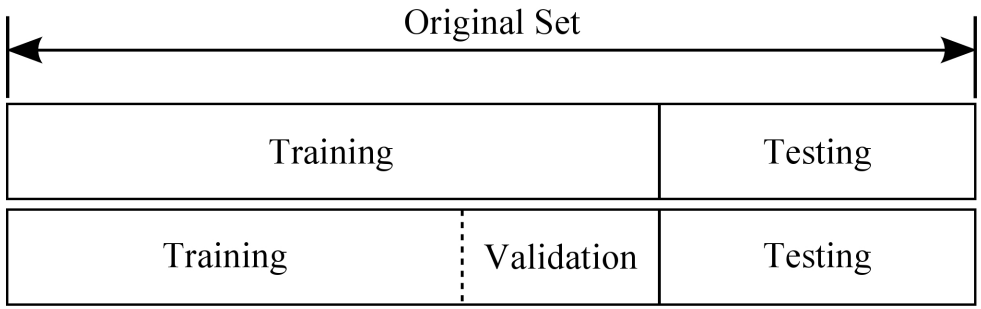
Choose hyperparameters to minimise D_{test} error? No - choosing based on D_{test} makes it an unreliable estimate of error!

Validation

Solution: randomly take out 10-50% of training and use it instead of the test set to estimate test error.

Validation set

A validation (development) set is taken out of the training data which acts as a surrogate for the test set.

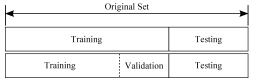


Validation

Solution: randomly take out 10-50% of training and use it instead of the test set to estimate test error.

Validation set

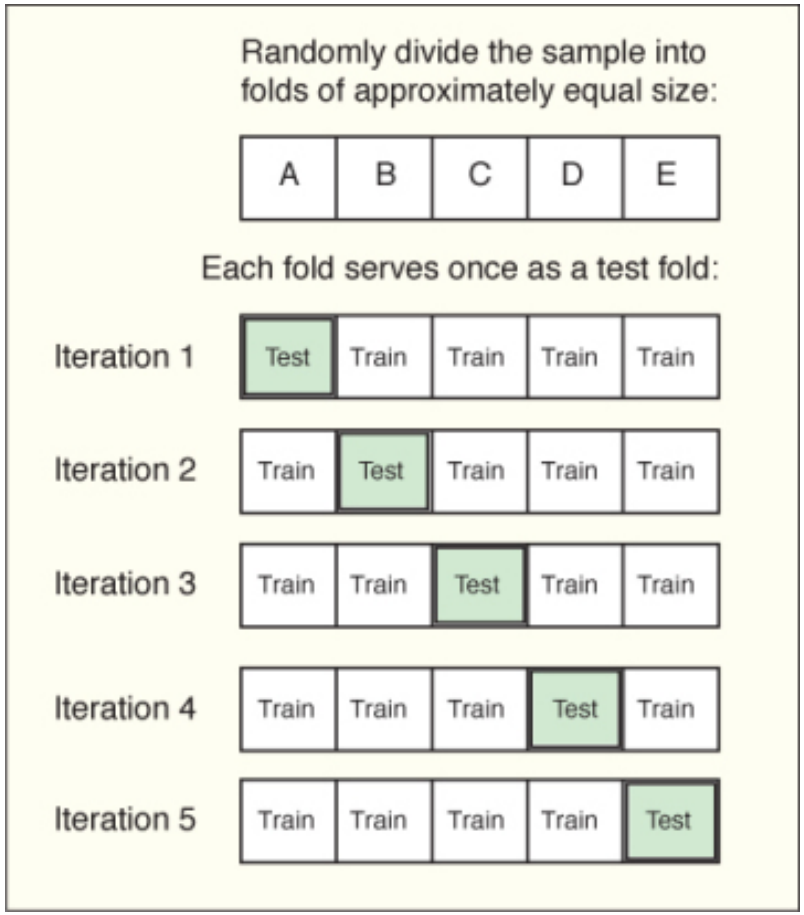
A validation (development) set is taken out of the training data which acts as a surrogate for the test set.



Generally, our learning algorithm has multiple hyperparameters to set. These hyperparameters cannot be set by the learning algorithm on the training data because we would just choose a degenerate solution and overfit. On the other hand, we can't use the test set either because then we would spoil the test set. The solution is to invent something that looks like a test set. There is no other data lying around, so we will have to steal it from the training set. The resulting set is called the **validation set**. To sum up:

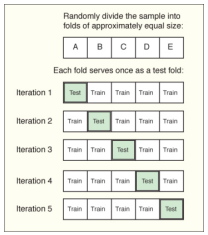
- Training set is used to learn the models.
- Validation set is used to estimate prediction error for model selection.
- Test set is used fro assessment of the generalisation error of the chosen model.

Cross-validation



Thomas W. Miller, 2013

Cross-validation



Thomas W. Miller, 2013

A random splitting of a sample into training and test sets could be fortuitous, especially when working with small data sets, so we sometimes conduct statistical experiments by executing a number of random splits and averaging performance indices from the resulting test sets. One variation on the training-test split is multifold **cross-validation**. We partition the sample data into M folds of approximately equal size and conduct a series of tests. For the five-fold cross-validation shown in the figure, we would first train on sets B through E and test on set A. Then we would train on sets A and C through E, and test on B. We continue until each of the five folds has been utilised as a test set. We assess performance by averaging across the test sets. In **leave-one-out** cross-validation, the logical extreme of multifold cross-validation, there are as many test sets as there are observations in the sample.

Machine learning in practice (I)

If our algorithm shows high bias, try

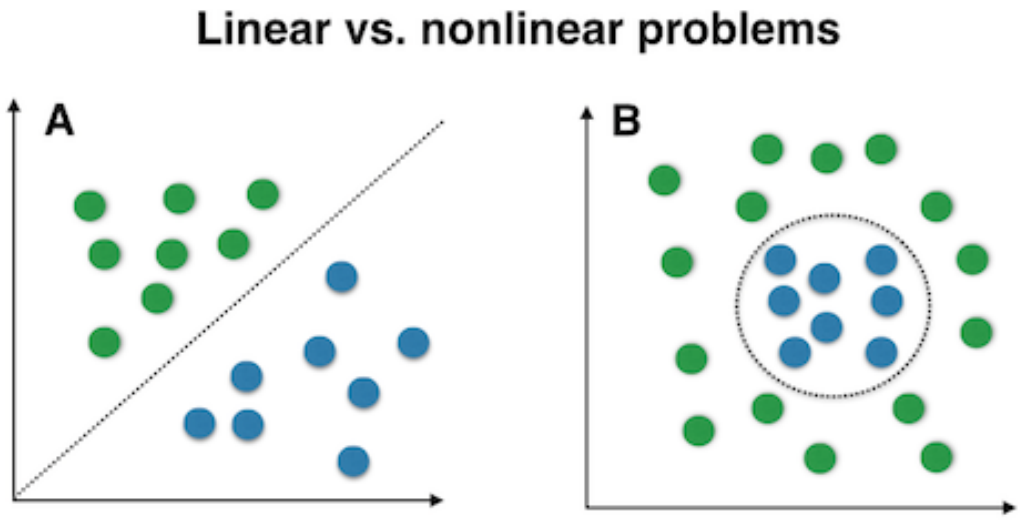
- **Adding more features**. In the example of predicting home prices, it may be helpful to make use of information such as the neighbourhood the house is in, the year the house was built, the size of the lot, etc. Adding these features to the training and test sets can improve a high-bias estimator.
- **Using a more sophisticated model**. Adding complexity to the model can help improve on bias. For a polynomial fit, this can be accomplished by increasing the degree. Each learning technique has its own methods of adding complexity.
- **Using fewer samples**. Though this will not improve the classification, a high-bias algorithm can attain nearly the same error with a smaller training sample. For algorithms which are computationally expensive, reducing the training sample size can lead to very large improvements in speed.
- **Decreasing regularisation**. Regularisation is a technique used to impose simplicity in some machine learning models, by adding a penalty term that depends on the characteristics of the parameters. If a model has high bias, decreasing the effect of regularisation can lead to better results.

Machine learning in practice (II)

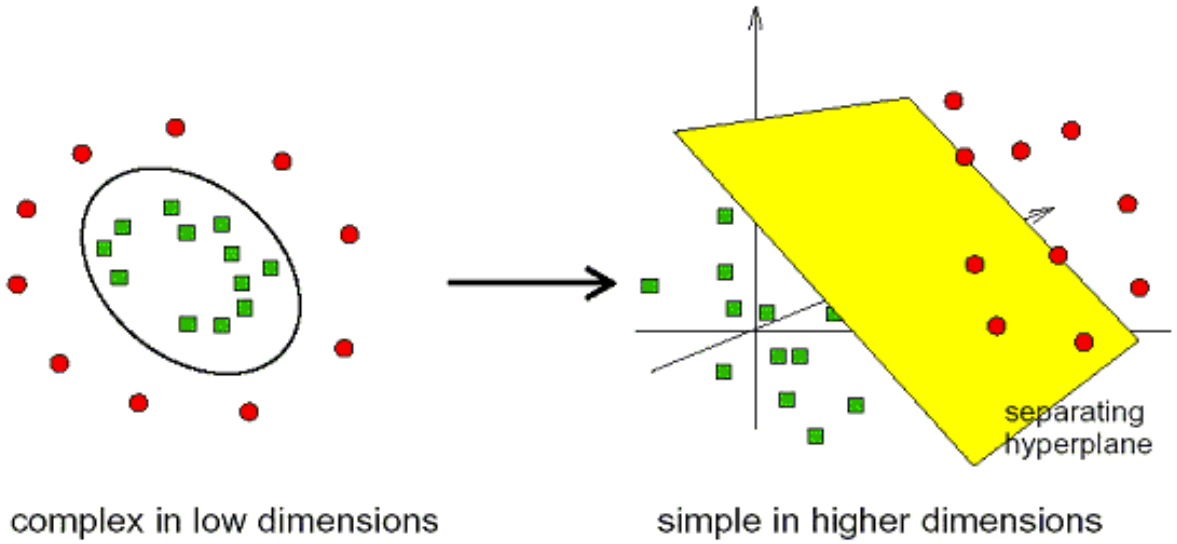
If our algorithm shows high variance, try

- **Using fewer features**. Using a feature selection technique may be useful, and decrease the overfitting of the estimator.
- **Using more training samples**. Adding training samples can reduce the effect of over-fitting, and lead to improvements in a high variance estimator.
- **Increasing regularisation**. Regularisation is designed to prevent over-fitting. In a high-variance model, increasing regularisation can lead to better results.

Linear vs Non-linear

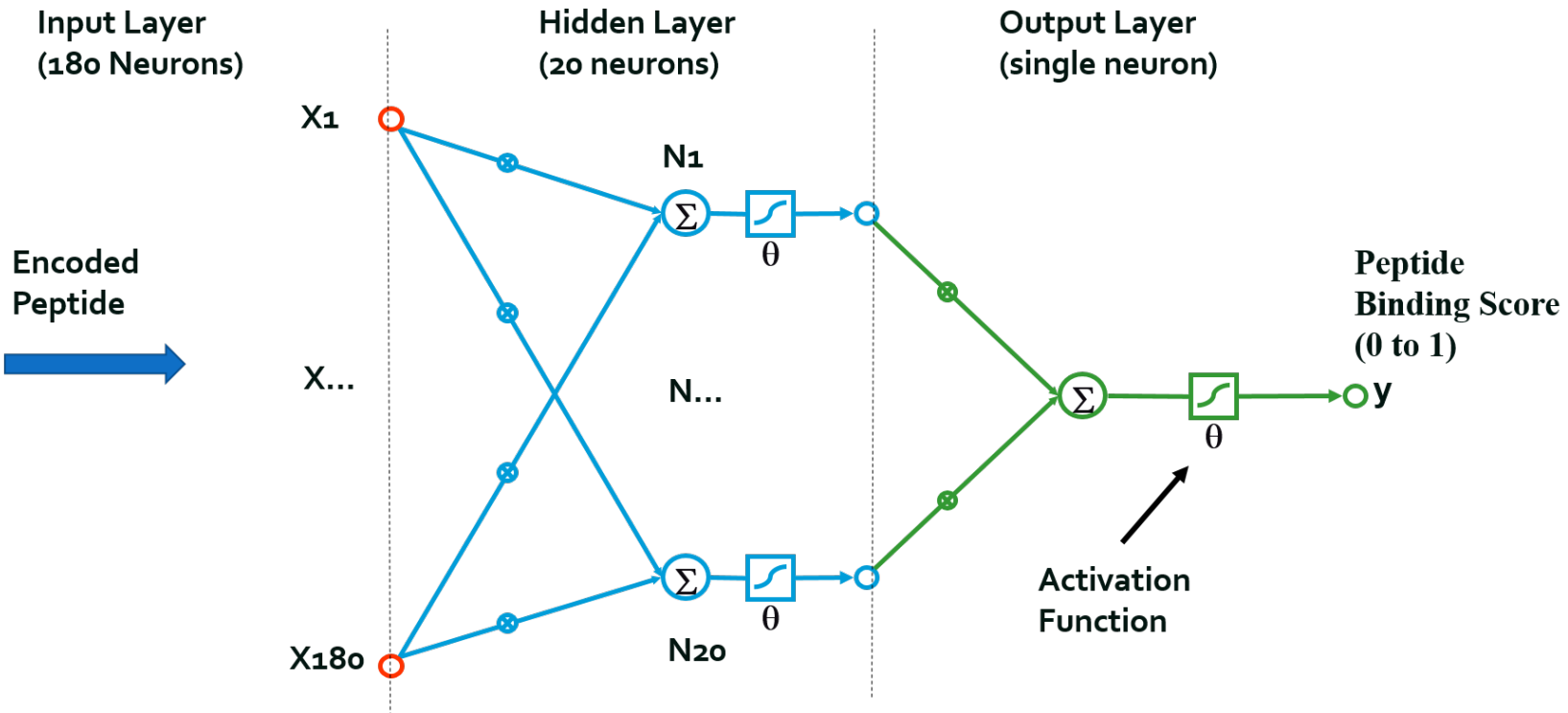


Non-linear prediction



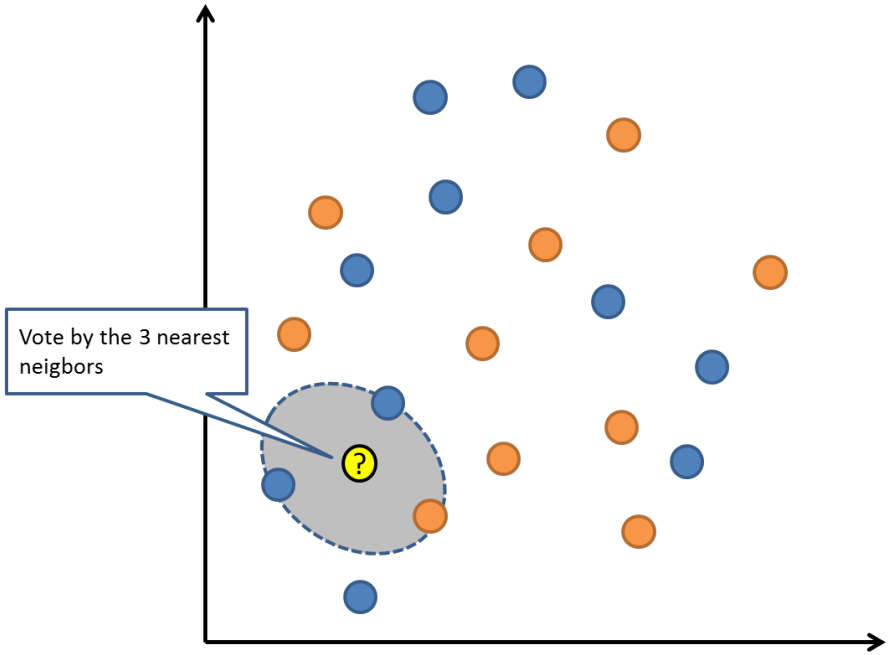
E.g., Kernel methods (Suppor Vector Machines)

Non-linear prediction



E.g., Neural Networks (Multi-layer Perceptron)

Non-linear prediction



E.g., Non-parametric methods (K-Nearest Neighbours)

Summary

We've looked at:

- Supervised learning: perceptron
- Loss minisation
- Generalisation: bias and variance
- Regularisation

A few things to have a look at:

- Proof that, if the data is separable, the perceptron algorithm converges!
- `https://techcrunch.com/2017/01/30/is-a-master-algorithm-the-solution-to-our-machine-learning-problem`

Next lecture

We will introduce recommender systems. We will also start discussing the concepts of unsupervised learning:

- clustering

- dimensionality reduction (PCA)