Introduction to Basic Supervised and Unsupervised Learning II

IRS ML Section*

Hwa Chong Institution

1 Supervised and Unsupervised Learning

Machine learning tasks can generally be classified into two categories: supervised learning and unsupervised learning. In this section, we shall distinguish between these two types of tasks and provide some examples of each, many of which will be covered later in the curriculum. You are strongly encouraged to do your own reading on whichever tasks you find interesting - the internet is your best friend.

1.1 Supervised Learning

Supervised learning is the task of learning a function $F : X \to Y$ based on a set of input-output pairs $S = \{(x_i, y_i)\}_{i=1}^{n}$ where $x_i \in X$ and $y_i \in Y$. S is usually known as the *training data set*. Each x_i is typically a vector[1] of *features*, and the y_i are called *labels*. One may think of each y_i as the "correct answer" to its corresponding x_i.

One often aims to construct some F that is as "correct" as possible for pairs in S, i.e. the deviation between $F(x_i)$ and y_i is minimized over all i. However, another important aim is *generalization* - constructing an F that also gives maximally correct answers for input-output pairs **not** in S.

Here are some examples of supervised learning tasks:

Linear regression. As a diligent, dedicated member of IRS ML, you would have completed this task in the previous lesson. In this case, $X = \mathbb{R}^k$ and $Y = \mathbb{R}$.

Image classification. Here, X is the set of all H by W images with C color channels, namely $(\mathbb{R}^+ \cup \{0\})^{H \times W \times C}$. The set of labels Y can be represented as $\{0, 1, ..., N\}$, where N is the number of classes that an image can be categorized into.

Handwriting recognition. X is the set of all images of handwritten text, while Y is the set of all possible text sequences.

* Written by Wing Yip and Chenghao

[1] In this context, a vector may be regarded as a list of numbers. You will learn about vectors in more detail later in the curriculum.

Spam detection. X is the set of all possible text sequences, while $Y = \{0, 1\}$. Each label corresponds to the designation of some text sequence as "spam" or "not spam".

1.2 Unsupervised Learning

Unsupervised learning is the learning of patterns within data that has no labels. It often involves the grouping of data points into clusters, or the generation of new examples that aim to mimic the training data.

Applications of unsupervised learning include:

Anomaly detection. Machine learning is commonly used to identify anomalies in time series data. Clustering techniques in unsupervised learning excel at this, as abnormal data tends to lie far away from clusters of normal data. This allows the data to be differentiated purely based on distances between data points without the need for labels.

Image generation. In recent years, unsupervised image generation has become an increasingly popular topic in machine learning literature; in particular, generative adversarial networks (GANs) have received extensive coverage. In unsupervised image generation, one usually aims to generate images similar to those in the training set, without reference to specific "correct" images as labels.

2 Underfitting and Overfitting

Generally, machine learning aims to accurately approximate the relationship governing a set of data points. In doing so, the algorithm used in approximation (the *model*) that one selects must be complex enough to approximate the relationship at hand; if not, one risks that the model is unable to accurately predict the labels of the training data, an issue known as **underfitting**. However, an overly complex model is also undesirable, as such a model may interpret mere noise in the data as part of the actual relationship. This issue is known as **overfitting**. Since the presence of labels is often required in order to quantitatively evaluate a model's accuracy, underfitting and overfitting are typically discussed within the context of supervised learning.

In this section, we will gain a visual intuition of underfitting and overfitting through a simple example, and learn some methods to avoid both problems.

2.1 Example

Consider a set of points whose y-values are given by some quartic function of the x-values, plus some random noise. (Fig. 1)

One may attempt to fit these data points using a quadratic polynomial (Fig. 2).

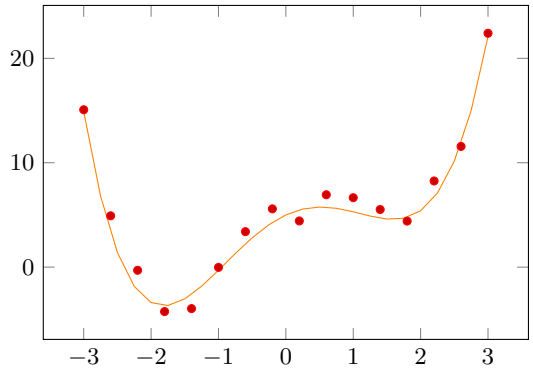

Fig. 1. Data points governed by a quartic relationship

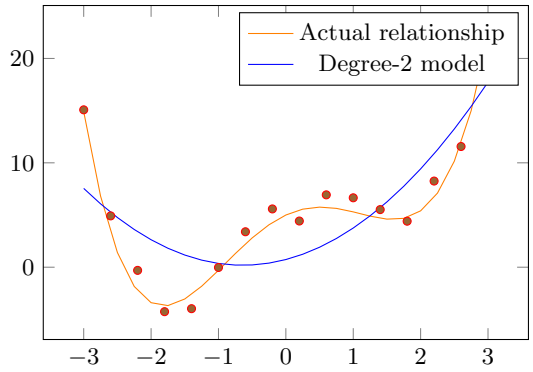

Fig. 2. A quadratic approximation of the relationship

It is apparent that the actual y-values deviate significantly from those predicted by the quadratic polynomial, meaning that the quadratic approximation is poor.

One notices that the dataset displays at least three "wiggles", or turning points - i.e. where the sign of the derivative of the polynomial governing the data appears to change. However, the quadratic polynomial can have at most one turning point, causing it to have insufficient "wiggle room" to approximate the quartic relationship. This is a situation of **underfitting**, where our model is not complex enough to model (or "fit") the relationship in the data.

Increasing the degree of our polynomial gives us a more complex model. As it turns out, a quartic polynomial can approximate the quartic relationship rather well (Fig. 3). This is because it is just complex enough to fit the data.

One might now be tempted to increase the degree of the polynomial even further - after all, shouldn't a more complex model be able to fit the data even better? At

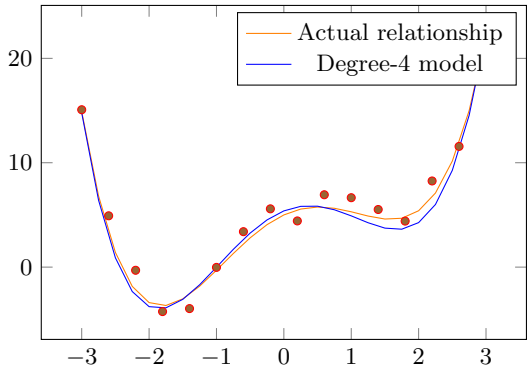

Fig. 3. A quartic approximation of the relationship

first glance, this seems to be true of the degree-11 approximation below, which passes through every single data point: (Fig. 4)

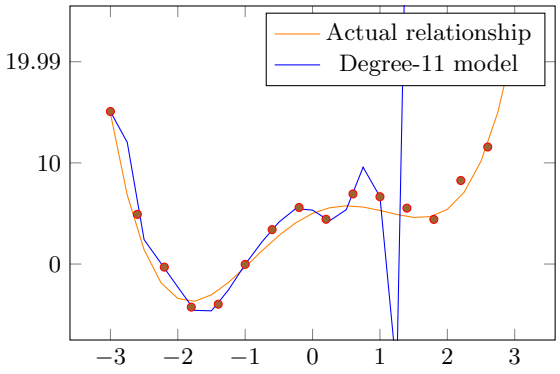

Fig. 4. A degree-11 approximation of the relationship

However, using an overly complex model such as this one is also undesirable. The noise in the training data is represented by the random deviation of the data points from the orange line, and is not representative of the trend governing the data. Hence, an accurate model should not take this noise into account. However, the degree-11 approximation interprets the noise as part of the trend, as shown by how it passes through all points even though they are perturbed by noise. This phenomenon is known as **overfitting**.

The main detriment of overfitting is that it hurts prediction accuracy on points outside the training set; in other words, overfitting increases *generalization error*. An intuitive explanation is that the overly complex model "memorizes" the noise within the training data, and thus models an inaccurate relationship that does

not generalize to data it has not seen. This is demonstrated by how, between the data points, the degree-11 polynomial deviates appreciably from the actual relationship, and begins to take very extreme values for $x > 1$.

In summary, underfitting is usually caused by a model that is too simple, while overfitting is generally caused by one that is too complex. Both phenomena are undesirable.

2.2 Avoiding underfitting

In most cases, underfitting can be resolved by increasing the complexity of one's model. In the context of deep learning, this typically means increasing the number of layers in one's model, or increasing the number of units in one or more hidden layers.

When using models that rely on iterative training (usually repeated looping over a set of training examples), another reason for underfitting could be that the model has not run for enough iterations to reach an accurate solution. In this case, one can simply train the model for more iterations. It is often useful to plot a graph of the error or loss of one's model as it trains, and only terminate training when the loss appears to converge to an asymptote. However, do note that deep learning models sometimes display multiple apparent convergences before loss again decreases sharply (e.g. Fig. 5), so when in doubt, continue training.

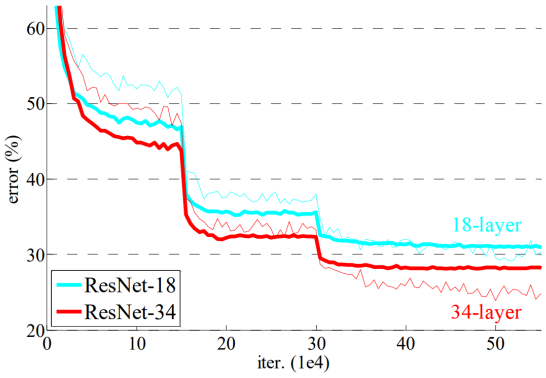

Fig. 5. Instance of the "multiple convergence" phenomenon during training of the well-known ResNet architecture. One observes two seemingly asymptotic convergences followed by sharp drops in loss.

2.3 Avoiding overfitting

Overfitting is a more persistent issue, especially in deep learning. As such, a number of techniques have been developed to tackle it, some of which are listed below.

Regularization. In machine learning, one often seeks to minimize a loss function, which is an example of an *optimization problem*. In general, solutions to optimization problems may not be unique - for instance, there are an infinite number of polynomials that pass through all the data points in the previous example. Regularization serves to alter the learning algorithm such that it prefers **less complex** solutions that are less prone to overfitting.

One common regularization technique is to ensure that the weights or coefficients in one's model remain small. Intuitively, one expects that the output of a satisfactory model ought not to change significantly if its input is slightly perturbed, i.e. $F(x+\epsilon) \approx F(x)$ for small ϵ. This helps the model to generalize to previously unseen points close to the training data distribution, as such points would likely have similar y-values to those of the training data itself. However, the value of a polynomial with large coefficients would likely change more for a given perturbation in its input. This hints that models with large coefficients may not generalize well to nearby points even if they predict the training data perfectly, which is exactly the definition of overfitting.

A typical way to prevent weights from becoming too large is to add a *regularization term* to the loss function that penalizes weights of large magnitude. Consider the mean squared error loss function:

$$J(x, y; \theta) = \frac{1}{2N} \sum_{i=1}^{N} (y_i - F(x_i))^2$$

where the model parameters are collectively represented by the matrix or vector θ. To penalize large weights (that is, make the loss higher when weights are large), we could add the magnitude, or norm, of the weights:

$$J(x, y; \theta) = \frac{1}{2N} \sum_{i=1}^{N} (y_i - F(x_i))^2 + \|\theta\|$$

If we want to control the extent to which the magnitude of the weights influences our loss function, we could also add a parameter $\lambda > 0$ that scales the regularization term, as below:

$$J(x, y; \theta) = \frac{1}{2N} \sum_{i=1}^{N} (y_i - F(x_i))^2 + \lambda\|\theta\| \tag{1}$$

The regularization method in (1) is known as $\mathbf{L^1}$ **regularization**, as the L^1 norm $\|\theta\|$ of the weights is penalized.[2] Another common regularization technique is L^2 regularization, which instead penalizes the (squared) L^2 norm by using the

[2] See https://en.wikipedia.org/wiki/Lp_space#Definition

regularization term $\lambda\|\theta\|^2$. By causing the magnitude of the weights to contribute to the loss, we encourage the gradient descent algorithm to reduce the magnitude of the weights. This makes overfitting less likely, as previously explained.

Thinking question: L^1 regularization shrinks the weights for unimportant features to zero (thus performing *feature selection*), while L^2 regularization does not exhibit this behaviour. By considering $\partial L/\partial\theta$, can you explain why this is so?

Validation and test data. In ensuring that one's model does not overfit, it makes sense to have a way to measure the generalization ability of the model. This can be done by calculating the accuracy of the model's predictions on data it has not previously seen, but for which labels are known.

When training a model iteratively, labelled data is commonly split into **training**, **validation** and **test** sets. After every iteration of training on the training set, the model is made to predict on the validation set, which only contains examples that the model has not encountered during training. The prediction accuracy on the validation set, known as the *validation accuracy*, is thus a measure of generalization performance. Since it is computed after every training iteration, one can stop training if validation accuracy begins to fall, which is an indicator of overfitting due to overtraining.[3] After training is completed, the model then makes a prediction on the test set as a gauge of its final accuracy.

Importantly, for validation and test accuracies to be valid measures of generalization performance, the training, validation and test sets should all be distinct from each other. If points from the training set are included in the validation or test set, the generalization ability of the model would be assessed based on data it has already seen, which makes no sense. This issue is known as data leakage, and can lead to overly optimistic estimates of model performance. This sometimes results in the misguided creation of extremely inaccurate or completely unusable models.

3 Classical ML Algorithms

In this section, we shall discuss some classical algorithms for machine learning - that is, non-deep learning ones. While deep learning has seen immense popularity in recent years due to its wide applicability and often high accuracy, one of its major shortcomings is the huge amount of computational power required to train and use deep learning models. In contrast, classical ML algorithms can achieve comparable accuracy on several tasks with a fraction of the computational cost; hence, learning about them remains worthwhile.

[3] Intuitively, overtraining occurs when an excessively complex (or *overparametrized*) model - typically a neural network - begins to fit the noise in the data after seeing it too many times.

3.1 Supervised Learning

K-Nearest Neighbours
The k-nearest neighbours algorithm is a supervised learning algorithm that can be used for either classification or regression. The prediction of the algorithm on a previously unseen point p in data space is determined by the k nearest training examples to p.

In classification, p is assigned to the class that is most common among its k nearest neighbours.

In regression, the predicted y-value is given by the mean of the y-values of the k nearest neighbours. The contribution of each neighbour to the predicted y-value can be weighted by $1/d$, where d is the distance from the neighbour to p.

Naive Bayes Classifier The naive Bayes classifier classifies a data point $\mathbf{x} = (x_1, x_2, ..., x_n)$ into one of several categories. It uses Bayes' theorem to calculate the probability that the point belongs to each class, and selects the class with the highest probability. The classifier is described as "naive" because it naively assumes that, given that the point belongs to a certain class, all features x_k are independent of each other.[4]

Decision Trees & Random Forest
Decision Trees (DTs) are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning **simple decision rules** inferred from the data features. A tree can be seen as a piecewise constant approximation.

Random forest is just a combination of multiple decision trees.

3.2 Unsupervised Learning

K-Means Clustering
Clustering is the process of dividing the entire data into groups (also known as clusters) based on the patterns in the data.

- Choose the number of clusters k
- Select k random points from the data as centroids
- Assign all the points to the closest cluster centroid
- Recompute the centroids of newly formed clusters
- Repeat steps 3 and 4 until centroids of newly formed clusters do not change

Applications Recommendation Engines, Customer Segmentation

[4] A more detailed explanation can be found in section 2 of https://cs229.stanford.edu/notes2020spring/cs229-notes2.pdf.

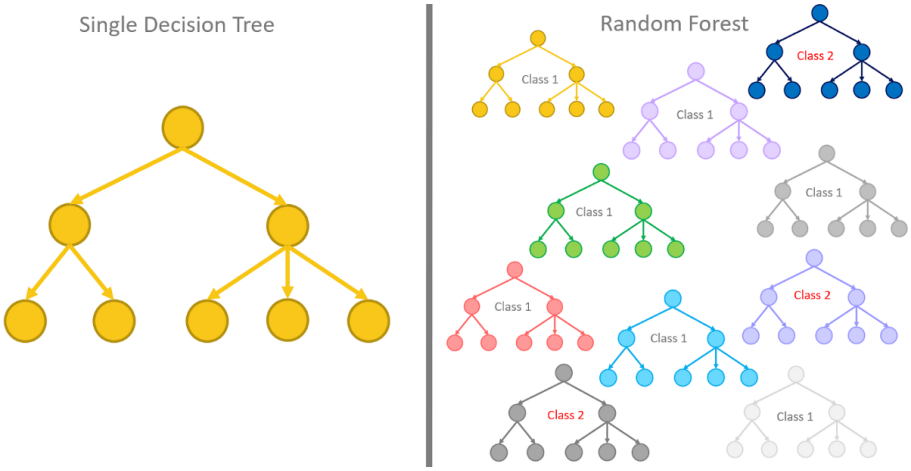

Fig. 6. A schematic diagram for a decision tree

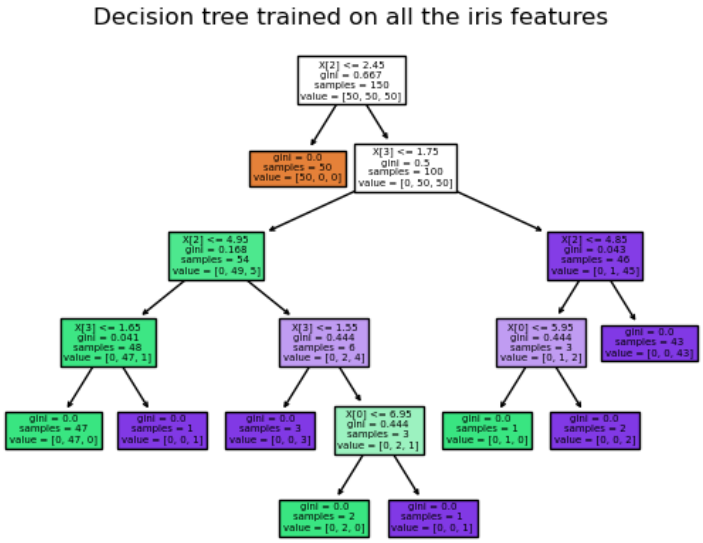

Fig. 7. Real-life example of a decision tree

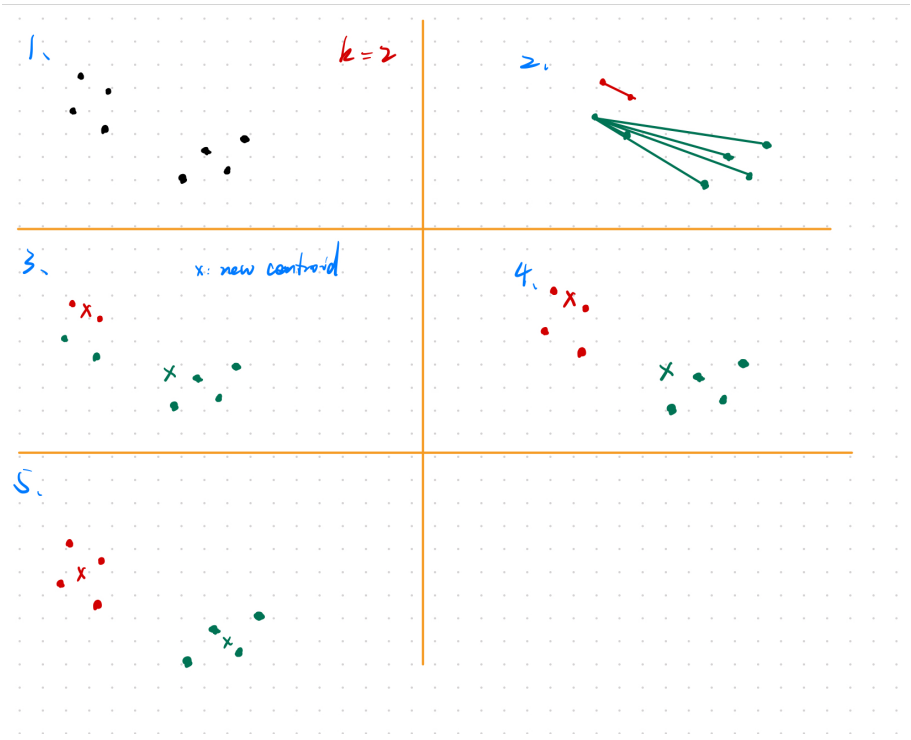

Fig. 8. K-means Clustering