

2020-10-23

## Foundations of Machine Learning: Week 2: Classification

Foundations of Machine Learning:  
Week 2: Classification

Professor Christopher Yau  
christopher.yau@manchester.ac.uk  
October 23, 2020

# Foundations of Machine Learning: Week 2: Classification

Professor Christopher Yau  
christopher.yau@manchester.ac.uk

October 23, 2020

## Machine Learning for Classification

In this lecture, we will examine the foundational framework for training supervised machine learning classification models:

- ▶ Classification algorithms represented as parameterised mathematical functions.
- ▶ Training data.
- ▶ Loss function.
- ▶ Gradient-based optimisation.

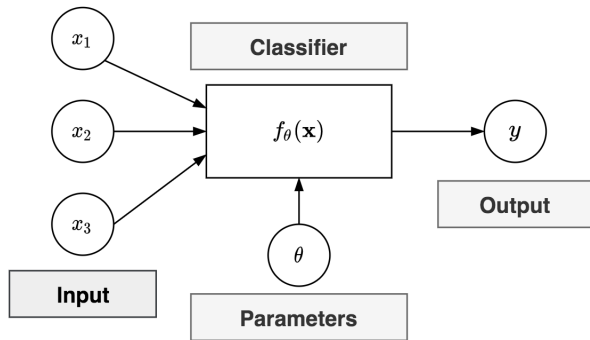
In this lecture, we will examine the foundational framework for training supervised machine learning classification models:

We will consider classification algorithms as parameterised mathematical functions.

How we use training data to learn the parameters of these functions.

What a loss function is in the context of learning and how gradient-based optimisation is used to optimise loss functions.

A schematic of a classification system:

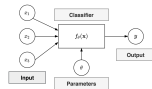


The **classifier** is used to transform the inputs  $\mathbf{x}$  into an output  $y$ . The **parameters**  $\theta$  control the behaviour of the classifier.

2020-10-23

### Overview

A schematic of a classification system:



The **classifier** is used to transform the inputs  $\mathbf{x}$  into an output  $y$ . The **parameters**  $\theta$  control the behaviour of the classifier.

We will be particularly in the scenario in which we wish to design a classifier represented by a mathematical function  $f_{\theta}(\mathbf{x})$  defined by parameters  $\theta$  with potentially multidimensional inputs or predictors  $\mathbf{x}$  and a single output  $y$ . Our study is of the classic building blocks that underpin modern classification methods in machine learning but still remain relevant and important even though they maybe *under the hood*.

# Binary classification

A *binary classifier* takes in an **input** (or predictor)  $\mathbf{x}$  and outputs a classification  $y \in \{-1, 1\}$  (or  $y \in \{0, 1\}$ ).

Classification function  $f_\theta$ , parameters  $\theta$ :

$$z = f_\theta(\mathbf{x}),$$
$$y = \begin{cases} -1, & z < 0, \\ +1, & z > 0. \end{cases}$$

The intermediate variable  $z$  is called a *latent variable*.

## Binary classification

$$z = f_\theta(\mathbf{x}),$$
$$y = \begin{cases} -1, & z < 0, \\ +1, & z > 0. \end{cases}$$

1. We will begin with a binary classifier which takes in an input  $\mathbf{x}$  and outputs a classification  $y$  where we often enumerate the two choices of  $y$  by either -1 and 1 or 0 and 1. These are entirely equivalent but make sure when deriving the classifier you are consistent with your use of output labelling.
2. A classifier can be described as a mathematical function which we denote using  $f$  and it comes with it some parameters  $\theta$  which we can use to *train* the performance of the  $f$  so that it works on our particular problem.
3. One set up is as follows: we feed an input  $\mathbf{x}$  into the classifier which applies the function  $f$  with the parameters  $\theta$  to the input. It outputs a variable  $z$  - known as a latent variable. Then, depending on the sign of  $z$ , the final classification  $y$  is given as -1 when  $z$  is less than 1, or  $y$  equals 1 when  $z$  is greater than 0.

# Training a classifier

The process of *training* a classifier refers to learning the classification function  $f_\theta$ .

Often, we specify the mathematical form of  $f_\theta$  and the task is to estimate the best fit parameters  $\theta$  from training data (**parametric modelling**):

## Example 1

$$f_\theta(\mathbf{x}) = a \sin(x_1) + b \cos(x_2)$$

where  $\theta = \{a, b\}$ .

## Example 2

$$f_\theta(\mathbf{x}) = \text{NeuralNetwork}_\theta(\mathbf{x})$$

where  $\theta$  corresponds to the neural network *weights*.

## Training a classifier

The process of training a classifier refers to learning the classification function. In this case, we normally prespecify the form of the function  $f$  so we only need to learn the parameters  $\theta$ . This is known as parametric modelling. For example, this is a function involving sines and cosines applied to two inputs  $x_1$  and  $x_2$ . We need to learn the parameters  $a$  and  $b$ . Alternatively, if we are using something more sophisticated like a neural network, the parameters might be the collection of neural network weights. But the structure of the neural network is fixed and doesn't need to be inferred.

The process of training a classifier refers to learning the classification function  $f_\theta$ .

Often, we specify the mathematical form of  $f_\theta$  and the task is to estimate the best fit parameters  $\theta$  from training data (**parametric modelling**):

### Example 1

$$f_\theta(\mathbf{x}) = a \sin(x_1) + b \cos(x_2)$$

where  $\theta = \{a, b\}$ .

### Example 2

$$f_\theta(\mathbf{x}) = \text{NeuralNetwork}_\theta(\mathbf{x})$$

where  $\theta$  corresponds to the neural network weights.

# Training data

Given  $n$  pairs of observations:

$$\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_n, y_n)\}$$

A perfectly trained classifier reports the correct output for every input, i.e.

$$f_{\theta}(\mathbf{x}_1) \rightarrow \hat{y}_1, \hat{y}_1 = y_1,$$

$$f_{\theta}(\mathbf{x}_2) \rightarrow \hat{y}_2, \hat{y}_2 = y_2,$$

$$\vdots$$

$$f_{\theta}(\mathbf{x}_n) \rightarrow \hat{y}_n, \hat{y}_n = y_n,$$

The *classification* made by the classifier is often denoted by a hat  $\hat{y}$  to differentiate it from the true classification  $y$ .

## Training data

Given  $n$  pairs of observations:

$$\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_n, y_n)\}$$

A perfectly trained classifier reports the correct output for every input, i.e.

$$f_{\theta}(\mathbf{x}_1) \rightarrow \hat{y}_1, \hat{y}_1 = y_1,$$

$$f_{\theta}(\mathbf{x}_2) \rightarrow \hat{y}_2, \hat{y}_2 = y_2,$$

$$\vdots$$

$$f_{\theta}(\mathbf{x}_n) \rightarrow \hat{y}_n, \hat{y}_n = y_n,$$

The classification made by the classifier is often denoted by a hat  $\hat{y}$  to differentiate it from the true classification  $y$ .

In order to train our algorithms and learn these parameters we need data and, for classification, data comes in pairs of matched inputs and outputs and we need a collection of them.

In theory, a perfectly trained classifier predicts the correct output for every input and we normally train a classifier to achieve this goal.

Note this means that the final trained properties of a classifier might be biased by the properties of the training data.

We often use a hat to denote the predicted value of a variable to differentiate it from the true value.

# Loss function

A *loss function*  $l$  measures the discrepancy between a set of predictions and the actual values:

$$L_{\theta}(y, \hat{y}) = \sum_{i=1}^n l_{\theta}(y_i, \hat{y}_i)$$

where a suitable loss function for this problem is given by

$$l_{\theta}(y_i, \hat{y}_i) = \begin{cases} 0, & y_i = \hat{y}_i, \\ 1, & y_i \neq \hat{y}_i. \end{cases}$$

We incur a cost of 1 if we make a misclassification but 0 otherwise.

## Loss function

A loss function  $l$  measures the discrepancy between a set of predictions and the actual values:

$$L_{\theta}(y, \hat{y}) = \sum_{i=1}^n l_{\theta}(y_i, \hat{y}_i)$$

where a suitable loss function for this problem is given by

$$l_{\theta}(y_i, \hat{y}_i) = \begin{cases} 0, & y_i = \hat{y}_i, \\ 1, & y_i \neq \hat{y}_i. \end{cases}$$

We incur a cost of 1 if we make a misclassification but 0 otherwise.

But how do we measure training performance?

Well, for this, we use something known as a loss function. This measures the discrepancy between a set of predictions and their actual values.

Often because each training sample is independent of the others. We can breakdown the loss function into a sum about individual errors.

One simple loss function is known as the 0-1 loss and this says I pay a penalty of one if I misclassify a training sample but I incur no penalty if my prediction is correct.

There are many types of loss function and you can have different choices of loss function for the same classification model.

1. A loss function.

# Loss minimisation

The loss function varies with  $\theta$  because our classifications change as we alter the classification function  $f_\theta$ .

Our learning objective is to find parameters  $\theta$  that minimise the loss function:

$$\hat{\theta} \leftarrow \arg \min_{\theta} L_{\theta}(y, \hat{y})$$

We cannot do this exhaustively in most instances as there are usually an infinite many  $\theta$  values to choose from.

There is normally no closed-form mathematical expression for the  $\hat{\theta}$  that minimises the loss function either.

## Loss minimisation

The loss function is called a function because the loss varies with the model parameters. As I change the parameters, the performance of the classifiers changes, it makes different predictions, and therefore the loss changes. Our training or learning objectives is usually to find a set of parameters which gives me the minimum loss over all possibilities for the parameter. In general, we cannot do this exhaustively as there maybe an infinite number of possible parameters and no closed-form mathematical expression for it either.

The loss function varies with  $\theta$  because our classifications change as we alter the classification function  $f_\theta$ .  
Our learning objective is to find parameters  $\theta$  that minimise the loss function:

$$\hat{\theta} \leftarrow \arg \min_{\theta} L_{\theta}(y, \hat{y})$$

We cannot do this exhaustively in most instances as there are usually an infinite many  $\theta$  values to choose from.  
There is normally no closed-form mathematical expression for the  $\hat{\theta}$  that minimises the loss function either.



**Gradient-based** methods are a general class of numerical methods to find the optimal value of any arbitrary loss function which is differentiable.

At the minimum of the loss function, the gradient with respect to  $\theta$  is zero:

$$g_{\theta}(y, \hat{y}) = \left. \frac{\partial L_{\theta}(y, \hat{y})}{\partial \theta} \right|_{\theta=\hat{\theta}} = 0$$

A simple approach for finding  $\hat{\theta}$  is therefore to change  $\theta$  in the direction of decreasing loss until we can decrease the loss no further.

$$\theta^{(t+1)} = \theta^{(t)} - \lambda g_{\theta^{(t)}}(y, \hat{y})$$

where  $\lambda$  is learning rate (or "step size").

## Gradient-based minimisation

Consequently, we must use numerical methods to explore the parameter space to try and find optimal parameters.

One such class of methods are known as gradient-based methods. These apply to any loss function which is differentiable and we can compute derivatives with respect to its parameters.

The idea is that given a current value of a parameter, I want to change that parameter in a direction (given by the gradient of the loss function) that causes the loss to decrease (or increase) if I want to find the minimum (or maximum) value.

At the optimal value, I won't be able to move anymore because the gradient will be zero and there will no direction in which I can change the parameters to improve the loss.

In practice, we only take small steps at a time because the gradient changes as we change parameters, and we use something known as a learning rate or step size to control how far we move.

# Gradient-based minimisation

**Gradient-based** methods are a general class of numerical methods to find the optimal value of any arbitrary loss function which is differentiable.

At the minimum of the loss function, the gradient with respect to  $\theta$  is zero:

$$g_{\theta}(y, \hat{y}) = \left. \frac{\partial L_{\theta}(y, \hat{y})}{\partial \theta} \right|_{\theta=\hat{\theta}} = 0$$

A simple approach for finding  $\hat{\theta}$  is therefore to change  $\theta$  in the direction of decreasing loss until we can decrease the loss no further.

$$\theta^{(t+1)} = \theta^{(t)} - \lambda g_{\theta^{(t)}}(y, \hat{y})$$

where  $\lambda$  is *learning rate* (or "step size").

## Gradient-based minimisation

Iteration number  $t = 0$  and a starting point,  $\theta^{(0)}$ :

1. **Test for convergence.** If the conditions for convergence are satisfied, then we can stop and  $\theta^{(k)}$  is the solution.
2. **Compute a search direction.** Compute the vector  $g_{\theta^{(k)}}$  that defines the search direction.
3. **Compute the step length.** Find a positive scalar value,  $\lambda^{(k)}$  such that  $f(\theta^{(k)} + \lambda^{(k)} g_{\theta^{(k)}}) < f(\theta^{(k)})$ .
4. **Update the variables.** Set:  $\theta^{(k+1)} = \theta^{(k)} + \lambda^{(k)} g_{\theta^{(k)}}$ ,  $k = k + 1$  and go back to 1.

Iteration number  $t = 0$  and a starting point,  $\theta^{(0)}$ :

1. **Test for convergence.** If the conditions for convergence are satisfied, then we can stop and  $\theta^{(k)}$  is the solution.
2. **Compute a search direction.** Compute the vector  $g_{\theta^{(k)}}$  that defines the search direction.
3. **Compute the step length.** Find a positive scalar value,  $\lambda^{(k)}$  such that  $f(\theta^{(k)} + \lambda^{(k)} g_{\theta^{(k)}}) < f(\theta^{(k)})$ .
4. **Update the variables.** Set:  $\theta^{(k+1)} = \theta^{(k)} + \lambda^{(k)} g_{\theta^{(k)}}$ ,  $k = k + 1$  and go back to 1.

So an algorithm for gradient descent might look like this:

We take a starting point  $\theta^{(0)}$  for our parameter and we test for convergence. This is checking whether we should stop the algorithm and we might terminate because the gradient is so small and we are essentially close enough to the optimal already.

If not, we compute a search direction by computing the gradient of the loss function with respect to each of its parameters.

We then compute the step size if we are using an adaptive algorithm to change the step size as we go. Adaptive algorithms are useful because we may wish to take large steps initially when we are far away from the optimal and then smaller steps as we get closer to the minimum.

We then update the variables and repeat the whole process.

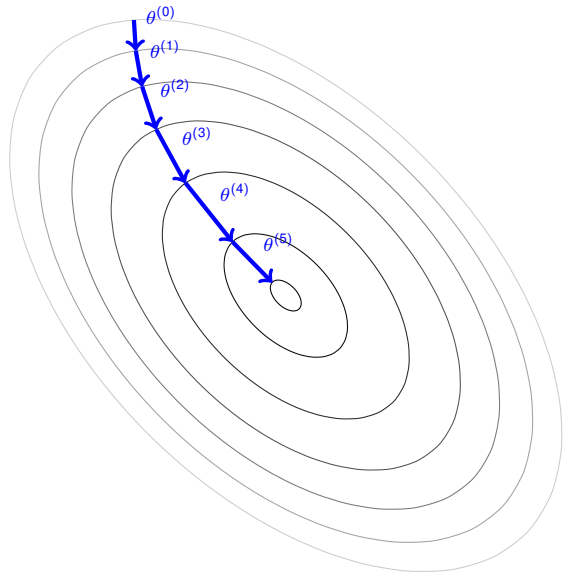
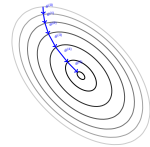
## Foundations of Machine Learning: Week 2: Classification

## Variations of gradient optimisers

There are many variants of gradient-based optimisation algorithms and these essentially differ based on exactly how they compute the search direction and the way they compute the step size for adaptive algorithms.

- There are many variants of gradient-based optimisation algorithms and these essentially differ based on exactly how they compute the search direction and the way they compute the step size for adaptive algorithms.

There are many variants of gradient-based optimisation algorithms and these essentially differ based on exactly how they compute the search direction and the way they compute the step size for adaptive algorithms.



This graphic just illustrates gradient-based optimisation.

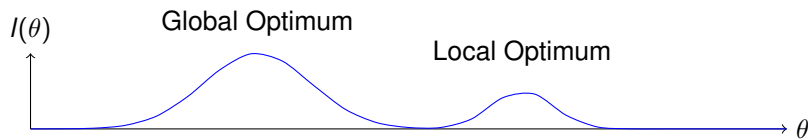
The contours corresponds to different levels of the loss function with the same loss value. You can think of it like a geographical contour map so each grey line designates points with the same height.

We want to reach the top of this hill and so we are going to climb up the hill by following the steepest route to get to the top as quick as possible.

# Limitations of Gradient Descent

The main limitation of gradient-based methods is multimodal loss functions.

If the loss function has two or more local minima (maxima) then the search algorithm will converge on one of these but not necessarily the globally optimum one.

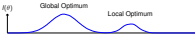


A common practical trick is to run the optimisation algorithm multiple times from different initial points and to pick the run that leads to the more optimal loss but this does not guarantee finding the global optimal and increases run time.

## Limitations of Gradient Descent

The main limitation of gradient-based methods is multimodal loss functions. If the loss function has two or more local minima (maxima) then the search algorithm will converge on one of these but not necessarily the globally optimum one. So if I start searching from the right in this cartoon, I will converge to the local optimum. But, if I had started on the left, I would have converged to the global one. A common practical trick is to run the optimisation algorithm multiple times from different initial points and to pick the run that leads to the more optimal loss but this does not guarantee finding the global optimal and increases run time.

The main limitation of gradient-based methods is multimodal loss functions. If the loss function has two or more local minima (maxima) then the search algorithm will converge on one of these but not necessarily the globally optimum one.



A common practical trick is to run the optimisation algorithm multiple times from different initial points and to pick the run that leads to the more optimal loss but this does not guarantee finding the global optimal and increases run time.

# Limitations of Gradient Descent

When computing the gradient of the loss function,  $g_{\theta}(y, \hat{y})$ , we use all the training data. This is known as **batch learning**.

If the training data is large, batch learning maybe inefficient or unfeasible (e.g. due to memory constraints).

We could attempt to *estimate* the gradient using a subset of training data.

These are known as **stochastic gradient descent** (SGD) algorithms.

## Limitations of Gradient Descent

Another limitation of gradient-based methods occurs with large data sets. Batch learning refers to when we do learning with all the data at our disposal. However, if the training data is large, it may not be feasible to do this due to memory constraints, processing times, etc. Further, if we have lots of data, is it really necessary to use all of it? If we need to compute the gradient, would that value change if we had used 1M data points versus 100,000? Often there are diminishing returns and it might be entirely possible to get good gradient estimates using subsets of data. This is the motivation behind stochastic gradient descent algorithms.

# Stochastic Gradient Descent

Recall the loss function is given by a sum over  $n$  training samples:

$$L_{\theta}(y, \hat{y}) = \sum_{i=1}^n l_{\theta}(y_i, \hat{y}_i)$$

so the loss gradient wrt to  $\theta$  is:

$$g_{\theta}(y, \hat{y}) = \sum_{i=1}^n \frac{\partial l_{\theta}(y_i, \hat{y}_i)}{\partial \theta}$$

which too involves a sum over  $n$  terms.

If  $n$  is very large (say  $n = 10^8$ ), this may involve a significant amount of computation.

## Stochastic Gradient Descent

Recall the loss function is given by a sum over  $n$  training samples:

$$L_{\theta}(y, \hat{y}) = \sum_{i=1}^n l_{\theta}(y_i, \hat{y}_i)$$

so the loss gradient wrt to  $\theta$  is:

$$g_{\theta}(y, \hat{y}) = \sum_{i=1}^n \frac{\partial l_{\theta}(y_i, \hat{y}_i)}{\partial \theta}$$

which too involves a sum over  $n$  terms.

If  $n$  is very large (say  $n = 10^8$ ), this may involve a significant amount of computation.

Mathematically, recall the loss function is given by the sum over  $n$  training samples.

So the gradient is also a sum of  $n$  terms as well after we differentiate with respect to the parameters.

If the data is large, say  $n = 10^8$ , this would involve a substantial amount of computation.

# Mini-batch SGD learning

Use a subset (mini-batch) of the training samples at each iteration to *estimate* the loss gradient  $g_\theta$ :

$$g_\theta(y, \hat{y}) \approx \hat{g}_\theta(y, \hat{y}) = \frac{n}{|\mathcal{S}|} \sum_{i \in \mathcal{S}} \frac{\partial l_\theta(y_i, \hat{y}_i)}{\partial \theta}$$

where  $\mathcal{S}$  denotes a small subset of the  $n$  data points.

The idea is that the mini-batch gives a *noisy* estimate of the true gradient but because the subset is much smaller than  $n$  it is quick to compute.

Without proof, mini-batch SGD algorithms have provable convergence properties.

Empirically, they sometimes work better than batch gradient descent as the noisy gradient estimates helps to overcome local minima.

## Mini-batch SGD learning

Use a subset (mini-batch) of the training samples at each iteration to estimate the loss gradient  $g_\theta$ :

$$g_\theta(y, \hat{y}) \approx \hat{g}_\theta(y, \hat{y}) = \frac{n}{|\mathcal{S}|} \sum_{i \in \mathcal{S}} \frac{\partial l_\theta(y_i, \hat{y}_i)}{\partial \theta}$$

where  $\mathcal{S}$  denotes a small subset of the  $n$  data points.

The idea is that the mini-batch gives a noisy estimate of the true gradient but because the subset is much smaller than  $n$  it is quick to compute.

Without proof, mini-batch SGD algorithms have provable convergence properties.

Empirically, they sometimes work better than batch gradient descent as the noisy gradient estimates helps to overcome local minima.

So the idea in stochastic gradient descent is to use a mini-batch or random subset of the data and to estimate the gradient from that subset.

So we might pick 5 samples compute the gradient from that and then scale by  $n/5$  to give an estimate of the gradient with all  $n$  samples.

We won't go into the mathematical proofs in this course but it can be shown that this procedure works and you can optimise effectively like this.

In fact, sometimes SGD works better than normal batch learning because the stochasticity introduced by the random subsampling can help us “jump” over small local optima in our loss function.

This is because each step it takes is not necessarily guaranteed to decrease the loss because the gradient is only estimated.



# Online learning

In the extreme case we can use only a *single* training sample at a time.

This is known as **online learning** because we can process training data one-at-a-time either:

1. The data is actually arriving one at a time,
2. The data is too large or the available computer memory insufficient to hold all the data at once for (mini-)batch processing.

## Foundations of Machine Learning: Week 2: Classification

2020-10-23

### Online learning

We can take this to the extreme and use only one data point at a time!  
This is known as online learning and is very useful in applications where data truly does arrive one at a time, such as in real-time applications, or if the data is too large or sensitive to store.  
Theoretically, this SGD algorithm will still work although it maybe very unstable.

[Online learning](#)

In the extreme case we can use only a single training sample at a time.

This is known as **online learning** because we can process training data one-at-a-time either:

1. The data is actually arriving one at a time,
2. The data is too large or the available computer memory insufficient to hold all the data at once for (mini-)batch processing.

# Why this matters? SGD - The State of Art

**Deep Learning** uses **Neural Networks** which can contain anywhere from 10s to billions of parameters.

We can use neural networks as a model for our classification function  $f$ .

The **ADAM** algorithm<sup>1</sup> frequently used for fitting modern deep neural networks is an example of a SGD algorithm.

<sup>1</sup>URL: <https://arxiv.org/pdf/1412.6980.pdf>

2020-10-23

## Foundations of Machine Learning: Week 2: Classification

### Why this matters? SGD - The State of Art

Why does all this matter?

The key thing is that variations of stochastic gradient descent are key tools in modern machine learning.

Deep neural networks contains 10s to billions of parameters and so SGD algorithms are the only feasible way of fitting these models.

One particular variant, the ADAM algorithm is extensively used, and is a SGD algorithm.

Deep Learning uses **Neural Networks** which can contain anywhere from 10s to billions of parameters.  
We can use neural networks as a model for our classification function  $f$ .  
The **ADAM** algorithm<sup>1</sup> frequently used for fitting modern deep neural networks is an example of a SGD algorithm.

<sup>1</sup>URL: <https://arxiv.org/pdf/1412.6980.pdf>

# Differential Programming

Modern, mainstream machine learning methodology (including deep learning) has coalesced around a broader and more general concept of **differential programming**.

If the performance of a computer program can be represented as a loss function  $L_\theta$  which is *differentiable* with respect to its parameters  $\theta$ .

⇒ The program can be optimised using gradient-based optimisation.

Differential programming encompasses a broad range of computational techniques and language implementations that make the design of programs that are efficiently differentiable using **autodifferentiation** (or **autodiff**).

## └ Differential Programming

Furthermore, modern machine learning has given rise to the concept of differentiable programming.

If the performance of a computer program can be represented by a loss function, we could seek to optimise that program via its parameters using a gradient based approach.

A large body of tools have been developed, such as autodifferentiation or autodiff, to help modern machine learning developers more rapidly develop learning algorithms to optimise different types of computer programs.

All of these are essentially based on the ideas I have discussed today.

Modern, mainstream machine learning methodology (including deep learning) has coalesced around a broader and more general concept of differential programming.

If the performance of a computer program can be represented as a loss function  $L_{\theta}$  which is differentiable with respect to its parameters  $\theta$ ,

Differential programming encompasses a broad range of computational techniques and language implementations that make the design of programs that are efficiently differentiable using autodifferentiation (or autodiff).

# Summary

In this lecture, we examined the foundational framework for training supervised machine learning classification models:

- ▶ Classification algorithms represented as parameterised mathematical functions.
- ▶ Training data.
- ▶ Loss function.
- ▶ Gradient-based optimisation.

## Foundations of Machine Learning: Week 2: Classification

2020-10-23

### Summary

In this lecture, I have describe the framework for training machine learning classification models. We have discussed how classifiers can be described using parametric functions with parameters to be learnt or optimised. We use training data containing input-output exemplars to estimate those parameters via a loss function which describes discrepancies and how our algorithm is performing. And I discussed how gradient-based optimisation provides a general technique for learning models.

#### Summary

In this lecture, we examined the foundational framework for training supervised machine learning classification models:

- ▶ Classification algorithms represented as parameterised mathematical functions.
- ▶ Training data.
- ▶ Loss function.
- ▶ Gradient-based optimisation.

