upGrad

Table of Contents

Logistic Regression

The Problem of Classification

Based on your experience with the field of machine learning so far, you might be able to think of a few use cases where it is useful to deploy machines to learn from our data and perform calculations for us. One such case is generating numerical values, as you learnt in linear regression. We can use several other machine learning techniques to get a numerical output based on a given set of input "features." For example, predicting the price of a house, based on various input variables such as number of rooms, area etc. Here, the target is continuous and can take infinitely many values.

But other than that, you may have seen examples where we see machine learning being used to assign a name or label to data. For example, consider the AI image search engines, which allow you to click an image, and within seconds, tell you what kind of object is present in your image. This is an example of classification – image classification to be precise – which lets us give a label to a data point based on whatever features it contains. Another example is identifying whether a patient is healthy or not.

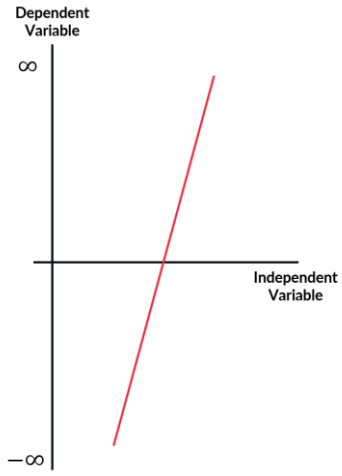
This assignment of labels to data is called **classification**. More formally, classification is the task of predicting a categorical outcome based on input features. In the simplest case, this outcome is binary. In other cases, it may involve multiple possible classes.

Examples of binary classification are determining if a student passes or fails an exam, identifying a transaction as fraudulent or safe and determining whether a text was written by human or LLM. Whereas multiclass classification includes cases where we have more than two classes – like identifying a flower or tagging the category of a news article.

The core idea of classification is to split the data into exclusive, discriminable categories and to make a single prediction.

Translating Numbers to Classes

In the case of linear regression, we have a linear model which relates a target to predictors. This gives a straight line (or hyperplane) equation which is unbounded and can go from $-\infty$ to ∞. For continuous sets of data, such as marks scored, this makes sense.



However, this does not fit well with the idea of dividing data points into groups or categories. A good approach must convey an objective boundary of classification, one which can be generalised over all data points of the sample, and new ones.

One such method would be to predict the probability of a data point belonging to a given class. We cannot do this with linear regression. For example, trying to predict the probability of passing using the hours of study as a predictor will not be possible. We can try to predict the score, but not the probability of passing.

Probabilities

So, how do we predict the probability then? If you think about it, directly calculating the theoretical probability would not be feasible – or even possible in some cases – real world distributions are unknown and too complex to model. We won't be able to consider the effects of all the factors that come into play.

And to calculate the experimental probability would require a humongous amount of data, which would again, not even ensure consistent performance on new, unseen data points.

We need a way that "translates" the insights from the linear model to probabilistic values – based on the features which we have knowledge of.

This is what logistic regression achieves. It combines the structure of a linear model and transforms into values that are interpretable as probabilities.

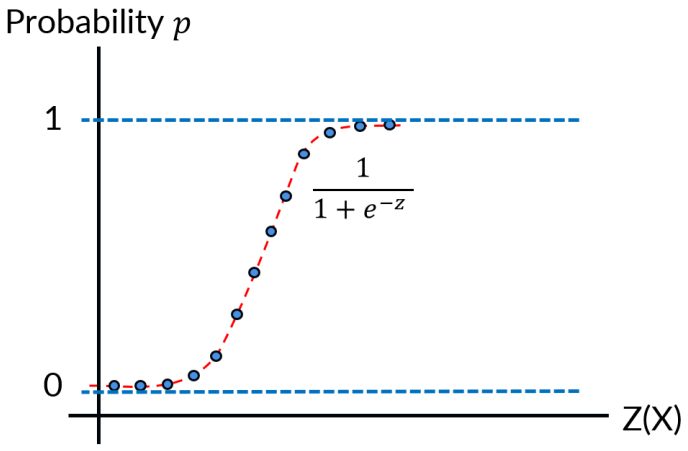
Now, logistic regression itself is not something different. It can be thought of as a generalisation of a linear model. More commonly, logistic regression is widely used for classification tasks, though it outputs probabilities from a regression-like model.

Regression just defines the relationship between a target and its features to output a numerical value. Logistic regression does the same, with a classification layer on top.

In logistic regression, we pass a linear model z to a sigmoid or logistic function,

$$p = \sigma(z) = \frac{1}{1 + e^{-z}}$$

If we plot this sigmoid curve, we will see this ranges from 0 to 1. Consider the plot showing the predicted values of probabilities for the values of z.

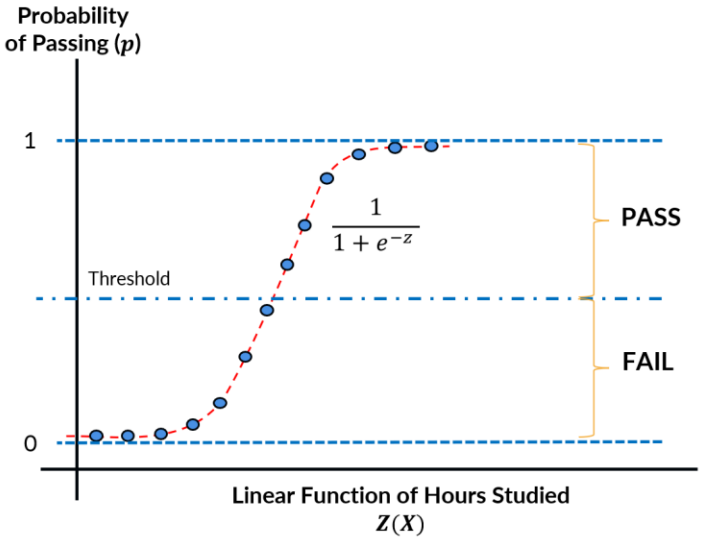


We can say that a point on the on the straight line is sent through the logistic function and acts as a probability value $\in [0, 1]$. Consider z as a linear function of study hours x. When the value of z increases, the probability output from the sigmoid function goes near 1. This makes sense, as increasing the number of study hours should translate to a higher probability of passing.

Probabilities and Classes

Now that we have established a function to output probabilities, the obvious next question is how to achieve classification labels from these probabilities. We use a **classification threshold** or a **cutoff value** of the probability.

If the probability of point x belonging to class *A* is more than the chosen cutoff value, we label the point to class *A*.



Let's take an example threshold of 0.5 in the above example. If the probability of a student passing the exam after studying for x hours is more than 0.5, we say that the student will probably pass.

In general, if the predicted probability

- $\hat{p} >$ threshold, we classify the point as 1 (*Does the point belong to class A: True*)
- $\hat{p} <$ threshold, we classify the point as 0 (*Does the point belong to class A: False*)

Note that the classifications 0 and 1 are different from probability [0, 1].

Here, these assignment of labels are only based on the predicted probabilities, whether these match with the **true labels** is another story.

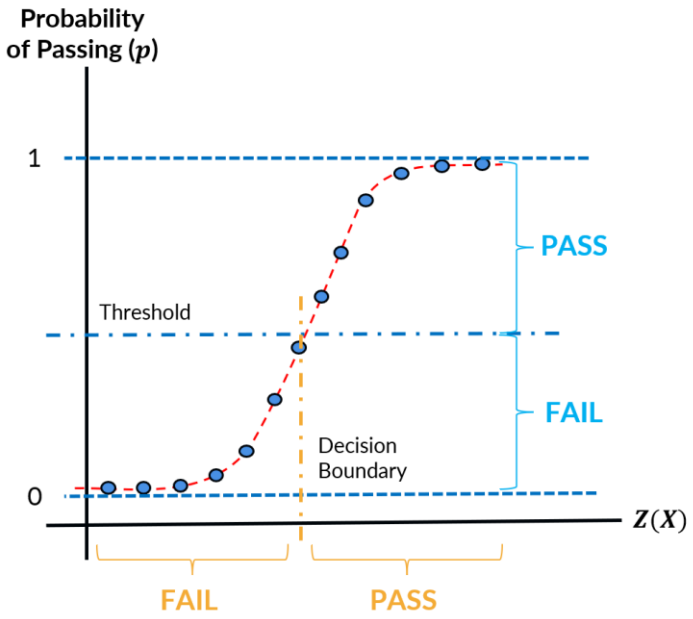
Now, this threshold **need not** mandatorily be 0.5, this is just a standard intuitive value that is used as a classification boundary. We can change the threshold based on use-case and the requirements of the domain.

This threshold in the y range converts to a **decision boundary** in the domain of x

$$\text{Decision boundary: } \hat{p} = 0.5 \Leftrightarrow z = 0$$

This is a boundary in the feature space where $z = 0$. In case of only one predictor, the decision boundary is a single point value, in case of two predictors, it is a line, and a hyperplane further ahead.

Note that the decision boundary corresponds to the **set of predictor values** that satisfy this condition of $z = 0$.



So, if we increase the threshold value to be greater than 0.5, the value of decision boundary will also increase, as an increase in the value of p means a higher corresponding value of z. With threshold = 0.7, the boundary is at $z \approx 0.85$.

If we lower the threshold, the decision boundary will also be lowered and will be at a negative value of z.

This means, that by increasing the cutoff probability to pass, we are asking for stronger evidence (a higher score or study hours) for classifying a student as "likely to pass".

The Logit

With some calculations, we can say, for a chosen value of p, the equivalent value of z is

$$z = \log\left(\frac{p}{1-p}\right) = \mathbf{logit}(\boldsymbol{p})$$

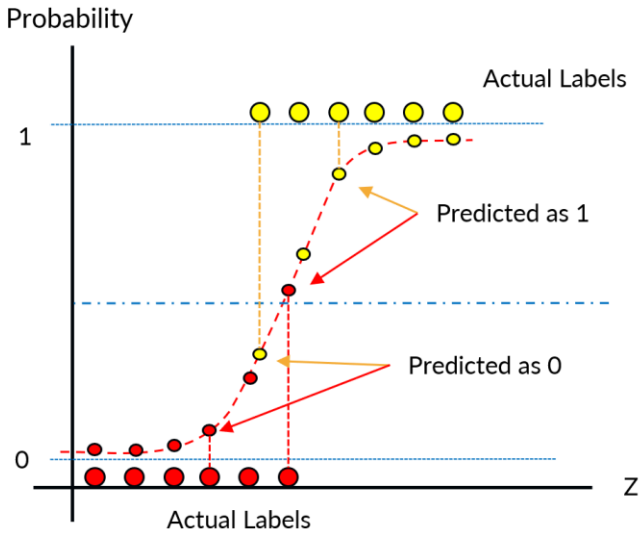
This is called the **logit function**. Logit function is essentially the inverse of sigmoid.

The chosen value of threshold passed to the logit function gives us the value of z for the decision boundary.

Here, we see an interesting expression come up, a familiar one if you are acquainted with introductory statistics, $\frac{p}{(1-p)}$ – also known as **odds.** We will discuss more on this later.

Classifications and Misclassifications

Till now, we have only seen how we assign labels based on predicted probabilities. But how do we determine if these assigned labels are correct or not?



In this image, we see the threshold is labelling the points as 1 or 0.

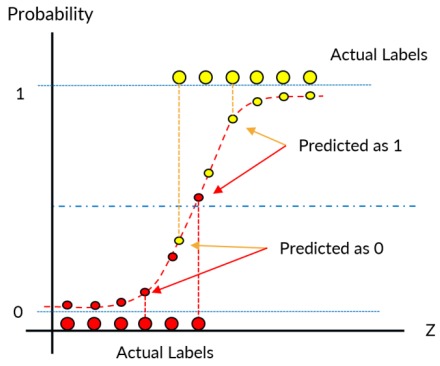
The true label of the yellow points is 1, and for red points, the true label is 0.

- If we classify the yellow points as 1, the classification is correct. If we classify them as 0, we are misclassifying.
- Similarly, if we classify red points as 1, we are correct; and if we classify red points as 0, we are misclassifying.

We can derive a measure to capture the performance of a logistic model in terms of correct and wrong classifications.

Classification Accuracy

Simply put, the classification accuracy is the percentage or fraction of the correct classifications out of all classifications by the model.



So, in the above example,

- The number of total classifications are twelve
- The number of correct classifications (both 0 and 1) are ten
    - Five correct classifications for red (0) and five correct classifications for yellow (1)
- Similarly, the wrong classifications are two (one of each)

So, the accuracy here becomes,

$$\text{Accuracy} = \frac{\text{correct classifications}}{\text{total classifications}} = \frac{10}{12} = 0.833 = 83.3\%$$

The Class of Interest

This concept of 0 and 1 labels brings us to the **class of interest**. How do we decide which class to give the label of 1?

The positive class or the class of interest is the category we are mainly interested in working with. The other class(es) become negative class(es). We choose the positive class based on the use case.

If we take "PASS" as the class of interest, we are interested in correctly predicting the passing factors/students. In case we want to understand the failing factors better, we can take "FAIL" as the class of interest.

The class of interest is an important aspect while evaluating the model performance.

The importance of this is better understood with an example of higher impact. Consider a case of diagnosing diseased patients and identifying healthy ones. Suppose our model is able to classify the patients as follows:

		Predictions	
		Terminal Disease	Healthy
Truths	Terminal Disease	5	3
	Healthy	1	11

We see that the number of patients with true label = "Terminal Disease" is 8, out of which, 5 are classified as diseased, and 3 are misclassified as healthy.

There are 12 patients with true label = "Healthy", out of which, 1 is misclassified as diseased and 11 are classified as healthy. Here,

$$\text{Accuracy} = \frac{5 + 11}{5 + 3 + 1 + 11} = 0.8$$

An accuracy of 80% is good, and we can be satisfied with the model; until we take a deeper look at the classification counts.

You will see that out of the 8 diseased patients, we only classified 5 correctly. This means we diagnosed only 62% of the diseased patients. We told around 40% of the diseased patients that they are healthy. This can be disastrous if you are a hospital. So here, we can take the class "Terminal Disease" as the class of interest.

Consider another case, where only 2% of the patients in the sample are diseased. Even if the model predicts all the patients as healthy, the accuracy will be 98%. This will cause all the diseased patients to be left untreated.

So, instead of just focussing on accuracy, you also need to keep a watch on how much of the positive class are you correctly able to classify. There are several other problems you need to solve for.

For example, out of the patients who are classified as healthy, how many are actually healthy; and out of those who are actually healthy, how many have we classified as healthy.

Here are some more examples and what we need to consider:

- Case – Loan defaulters / Safe Customers
    - We want to identify customers with even a slight chance of loan defaults
- Case – Positive sentiment / Negative sentiment
    - We want to identify negative sentiments better

Confusion Matrix

A confusion matrix tells us the counts of correctly and wrongly classified data points, for both positive and negative classes.

It displays the number of true and predicted values for both cases.

Consider the following image which shows a confusion matrix for binary classification.

		Predictions: Terminal Disease	Predictions: Healthy
Truths	Terminal Disease	5 **[TP]**	3 **[FN]**
	Healthy	1 **[FP]**	11 **[TN]**

Positive class → "Terminal Disease"

Here,

- **True Positive (TP):** Positive label has been correctly predicted as positive
- **True Negative (TN):** Negative label has been correctly predicted as negative
- **False Positive (FP):** Negative label has been incorrectly predicted as positive (Type I error)
- **False Negative (FN):** Positive label has been incorrectly predicted as negative (Type II error)

So, a confusion matrix gives a complete picture of classification – the classification and misclassification count for all cases – which tells us how the model is actually performing for all classes, and not just how accurate it is.

These numbers from the confusion matrix allow us to calculate additional performance metrics to evaluate the model's classification.

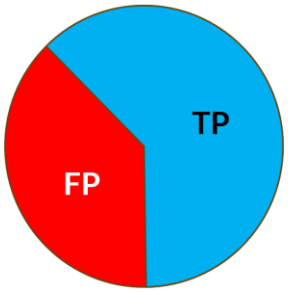
Precision

Precision measures the quality of positive predictions: *Of all the points we predicted as positive, how many were actually positive?*

For example, out of all the patients who have been classified as having the terminal disease, how many actually have the disease?

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

Precision is measured as the percentage of true positives in all the predicted positives.



Predicted Positives

Recall

We measure recall in two ways – for measuring how well the model captures the true labels.

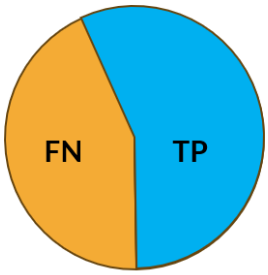
Positive Recall or Sensitivity

Positive recall, also known as recall or sensitivity, measures how well we capture all actual positives.

We measure recall as the percentage of true positives (correctly classified) out of all positives, or the true positive rate.

$$\text{Sensitivity} = \text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

It answers: *"Of all the actual positive cases, how many did we correctly identify?"* A high recall means we miss fewer positives.



Actual Positives

For example, in disease diagnosis, recall shows how many of the sick patients we successfully detected. Missing a sick patient (false negative) can be more dangerous than a false positive.

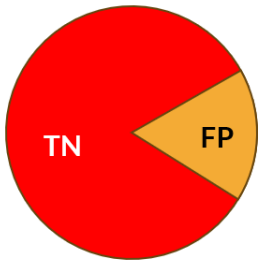
Negative Recall or Specificity

While sensitivity focusses on positive class, specificity measures how well we capture all actual negatives: *Of all the actual negative cases, how many did we correctly identify as negative?*

High specificity means fewer healthy patients are wrongly flagged as diseased.

We measure specificity as the percentage of true negatives (correctly classified) out of all negatives, or the true negative rate.

$$\text{Specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$



Actual Negatives

F1 Score

Often, precision and recall move in opposite directions, as recall increases (we catch more positives), precision decreases (more false alarms).

Sometimes, we want a **balance** between precision and recall. In such cases, we use the **F1-Score**. F1 score is the harmonic mean of precision and recall:

$$\text{F1} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

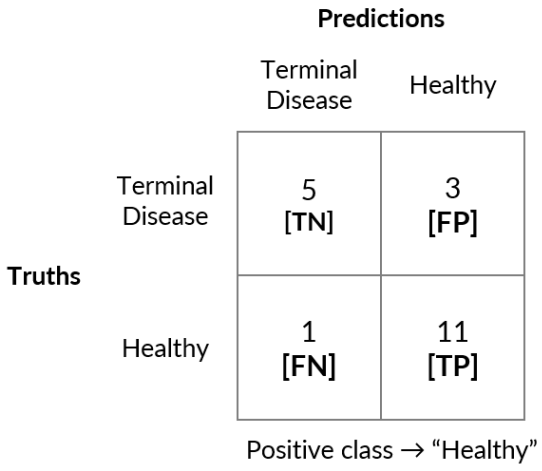
F1 is high only if **both precision and recall are reasonably high**. It punishes extreme imbalance (e.g., very high precision but very low recall).

For example, in fraud detection, both precision (to not flag innocent customers) and recall (catch fraud) are important, so F1 is a good metric.

Choice of Positive Class

The chosen positive class also affects which metrics we want to consider. If we switch the positive class, the confusion matrix flips: true positive is swapped with true negative and false positive is swapped with false negative.

In the image below, we change the positive class to “Healthy”.



So the choice of positive class depends on the particular use-case we are solving for.

- **Medical Diagnosis:** Identify as many diseased patients as possible
    - Positive class: Terminal Disease
    - Recall is more important than precision → Missing true patients is costlier than flagging false ones → Lower the threshold

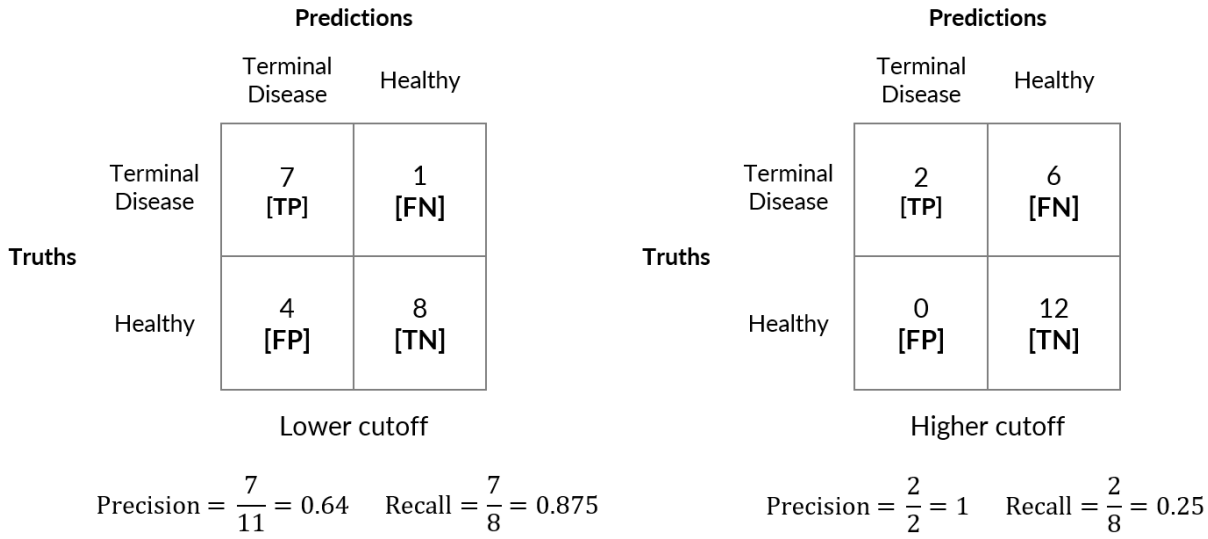
- **Insurance**: Identify as many healthy people as possible
    - Positive class: Healthy
    - Both recall and precision are important → Avoid false classification of diseased people as healthy → Threshold can be raised to balance precision and recall

Impact of Chosen Threshold on Metrics

The threshold value also affects the metrics vastly. If we lower the threshold, more points on the sigmoid will be classified to class label 1, whereas if we increase it, more points will be classified as 0.

While we have seen different metrics for different use-case, we also need to consider that the cutoff also needs to be optimised as per the business objective.

The following image shows how precision and recall change for different values of the cutoff.



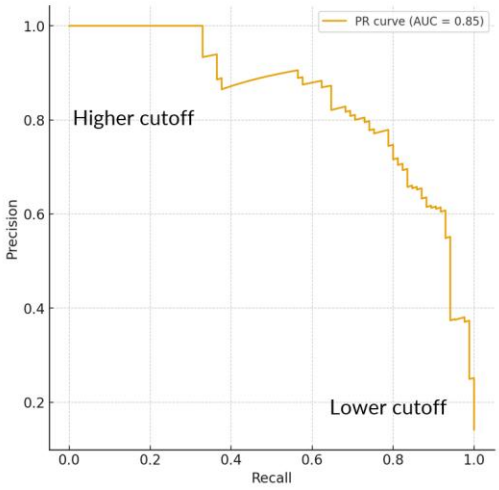
$\text{Precision} = \frac{7}{11} = 0.64 \quad \text{Recall} = \frac{7}{8} = 0.875$

$\text{Precision} = \frac{2}{2} = 1 \quad \text{Recall} = \frac{2}{8} = 0.25$

For reference, the original values of precision and recall for the confusion matrix were 0.83 and 0.625 respectively.

We see that **precision reduces** and **recall increases** when **threshold is lowered**. On the other hand, **precision increases** and **recall decreases** as **we increase the cutoff**.

If we plot the precision and recall for different cutoff values, we will get a plot like this:



Ideally, we would want both the precision and recall reaching 1, to give a maximum area under the curve of 1.

Another common way to study the effect of thresholds is by focusing on recall (TPR) and its complement, the **false positive rate (FPR)**.

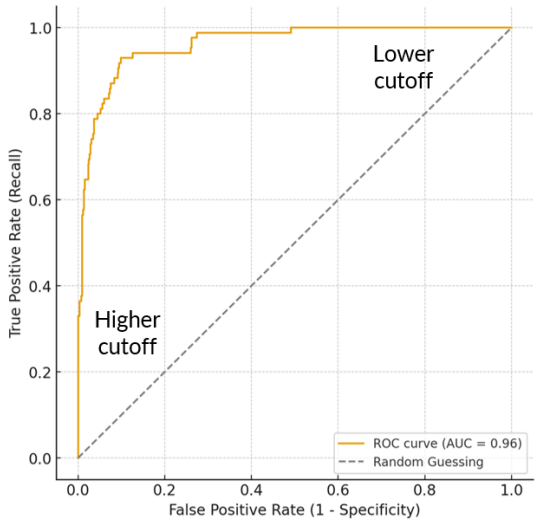
$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}} = 1 - \text{Specificity}$$

TPR tells us how many positives we caught, and FPR tells us how many negatives we mistakenly classified as positives.

By plotting these for different thresholds, we get Receiver Operating Characteristic (ROC) curve.

At very low threshold: almost everything is classified positive; TPR ≈ 1, but FPR is also high. At very high threshold: almost nothing is classified positive; TPR ≈ 0, FPR ≈ 0.

A ROC curve looks like this:



The area under the ROC curve (AUC) is a single-number summary of model performance. A ROC-AUC of 1 signals a perfect classifier, while ROC-AUC = 0.5 is a random guess (diagonal line).

When to use these plots:

- Precision-Recall (PR) Curve is especially useful for imbalanced datasets, since it directly shows performance on the positive class
- ROC Curve is more general and balances the positives and negatives

Multiclass Classification

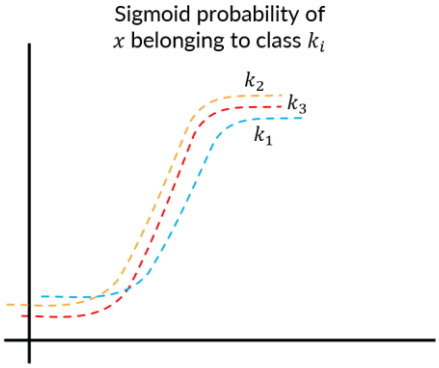
So far, we have seen how probabilities and classifications work for two classes. We predict the probability of a point belonging to a positive class using a sigmoid function.

For multiclass classification, the idea is similar. We have two major strategies for applying logistic regression on multiple classes.

One-vs-Rest

In this method, we train a separate binary classifier for each class. Each classifier predicts the probability of "class k" vs. "not class k." We then use sigmoid individually for them.

While classifying, we assign the class label of the class with the highest probability score.



But we want to express it as a combined probability: the overall probability of x belonging to class k. Moreover, the probabilities across classes may not sum up to 1.

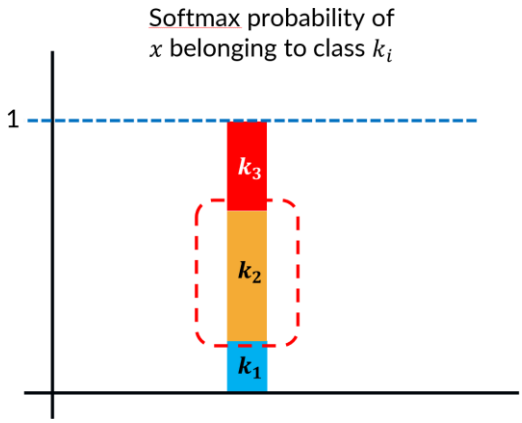
Softmax Function

The softmax function generalises the sigmoid to multiple classes. We take each class as a linear model (z_k with its own coefficients.)

Then the softmax function gives the probability of x belonging to class k,

$$p(y = k \mid x) = \frac{e^{z_k}}{\sum_{j=1}^{3} e^{z_j}} = \frac{e^{z_k}}{e^{z_1} + e^{z_2} + e^{z_3}}$$

This gives normalised probabilities. From the normalised probabilities, we label the data point to the class with the highest probability.



Metrics for Multiclass Problems

Evaluation follows the same principles as binary classification. For a given class of interest, treat it as "positive" and all others as "negative." We can then compute precision, recall, specificity, F1 just as before.

Because the focus is on the positive class, by default we report the metrics for it. Consider the following confusion matrix.

		Predictions		
		Work Email	Personal Email	Spam Email
Truths	Work Email	41	9	10
	Personal Email	4	**5**	1
	Spam Email	8	7	15

Positive class → "Spam Email"

Accuracy is independent of the choice of positive class, and remains the same for all cases.

$$\text{Accuracy} = \frac{41 + 5 + 15}{100} = 0.61$$

We can calculate the other evaluation metrics separately for each class.

For example, if we want to calculate the precision and recall for the class "Spam email," we will consider the 15 correctly classified spam emails as TP. The emails incorrectly identified as spam (10 + 1) will be FP, and the spam emails incorrectly classified as work or personal mails (8 + 7) will become the FN.

For example, the precision would be

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} = \frac{15}{15+1+10} = 0.57$$

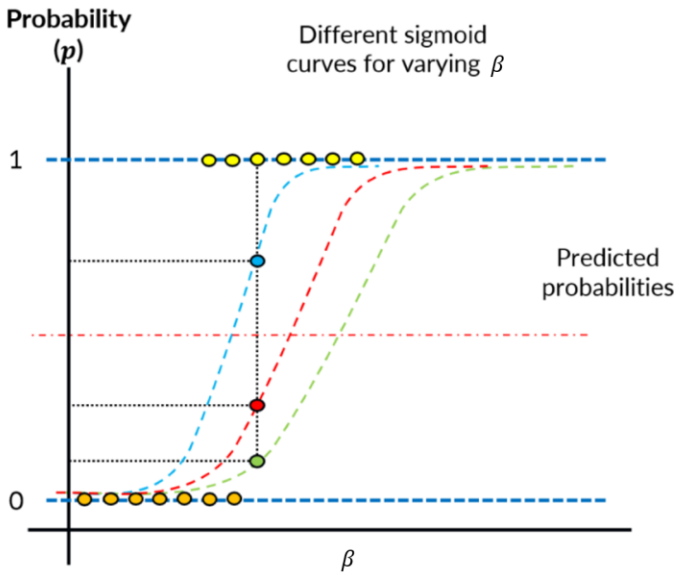
However, these same metrics can be calculated for other classes by considering them as positive and the rest as negative.

To combine results, we aggregate the metrics for all classes:

- **Macro-average**:
  - The unweighted average, mean of a metric across classes
  - Treats all classes as same regardless of sample size
- **Micro-average**:
  - Aggregate the TP, FP and FN counts across all classes before calculating metrics
  - Better when class sizes are imbalanced
- **Weighted Average**
  - Compute the metric for each class and take the weighted average
  - Each class contributes proportional to its size

Loss in Logistic Regression

We know how probability values are predicted, how the class labels are assigned, and how the model's performance for classification is evaluated. But how do we reach the optimised model? That is, how do we find the best coefficients $\beta_0, \beta_1, \ldots, \beta_p$ that make the model reliable?



In linear regression, we used **mean squared error (MSE)** as the cost function. This works because $\hat{y}_i$ are real-valued predictions. But in logistic regression the outputs are **probabilities** in $[0,1]$. The targets y_i are categorical (0 or 1).

We need a different measure, that naturally fits probabilistic predictions.

Likelihood

In terms of confidence in correctness of classification, we can say that a model is good if it assigns high probability to the true labels and bad if it assigns low probability to the true labels.

For this, we need a measure that improves when the classification is correct (predicted class label matches the true one) and confident (higher probability of belonging to the predicted label.) Similarly, the measure should become worse when the classification is confidently wrong.

We are trying to reach the coefficients that give us the sigmoid curve that classifies most accurately and confidently. To measure this, we measure the likelihood. Likelihood acts as a measure of 'closeness' between predictions and true values.

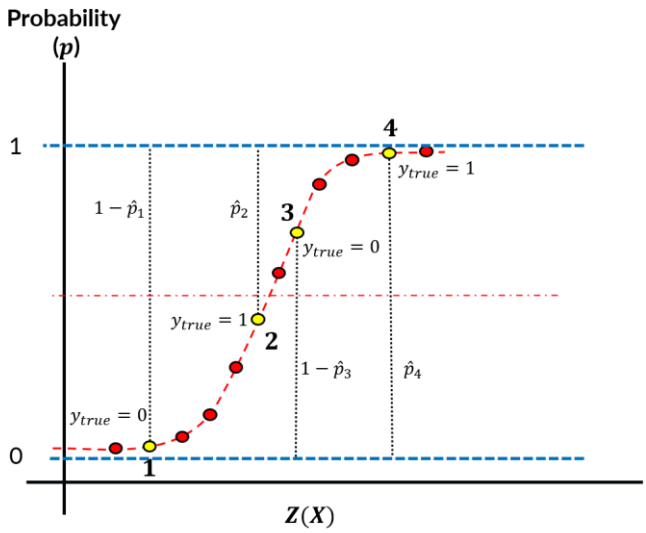
- If the true label is $y = 1$, and the model predicts $\hat{p} = 0.9$, the model is very likely correct
- If the true label is $y = 1$, and the model predicts $\hat{p} = 0.1$, the model is almost certainly wrong

Formally, for a single data point:

$$\text{Likelihood} = \begin{cases} \hat{p}_i, & \text{if } y_i = 1 \\ 1 - \hat{p}_i, & \text{if } y_i = 1 \end{cases}$$

This can be compactly written as:

$$L_i = \hat{p}_i^{y_i}(1 - \hat{p}_i)^{(1-y_i)}$$



Since the data points are assumed independent, the joint likelihood is the product over all n points:

$$L(\beta) = \prod_{i=1}^{n} \hat{p}_i^{y_i} (1 - \hat{p}_i)^{(1-y_i)}$$

Here, $L(\beta)$ depends on the coefficients β through the predicted probabilities $\hat{p}_i$.

The training objective is to maximise likelihood by optimising for the coefficients.

Log Loss or Binary Cross Entropy Loss

For easy calculation using probabilities, we take logs:

$$\ell(\beta) = \log\left(L(\beta)\right) = \sum_{i=1}^{n} [y_i \log \hat{p}_i + (1 - y_i) \log(1 - \hat{p}_i)]$$

This is the **log-likelihood**. Maximising it is equivalent to maximising the original likelihood.

For easy optimisation, we define the **loss function** as the **negative average log-likelihood**:

$$\text{Log Loss} = -\frac{1}{n} \sum_{i=1}^{n} [y_i \log \hat{p}_i + (1 - y_i) \log(1 - \hat{p}_i)]$$

This function appropriately factors in the confidence and correctness of predictions.

- Heavy penalty for **confident** (extreme $\hat{p}$) but **wrong** predictions
  - $y = 1, \hat{p} = 0.01 \rightarrow$ huge loss for the data point
- Small penalty for **confident** and **correct** predictions
  - $y = 1, \hat{p} = 0.99 \rightarrow$ tiny loss for the data point

This entire approach is known as **maximum likelihood estimation (MLE).** The optimisation is performed using solvers such as gradient descent or other advanced methods like higher order matrix calculations.

Coefficients and Odds

Odds

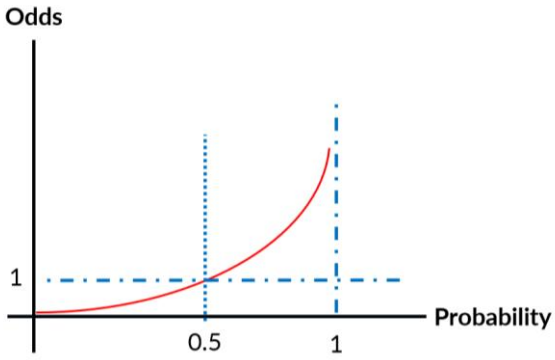
Recall the logit function,

$$z = \log\left(\frac{p}{1-p}\right) = \mathbf{logit}(\boldsymbol{p})$$

Here, we mentioned a term – **odds**.

In logistic regression, we work with probabilities. But it is also convenient to use odds. The odds of an event E occurring are defined as the ratio of the probability of it occurring to the probability of it not occurring,

$$\text{odds}(E) = \frac{p(E)}{1-p(E)}$$



This comes out to be hyperbolic, ranging from $[0,\ 1]$ and $[1, \infty]$ for probabilities $\in [0,\ 1]$.

Log-Odds

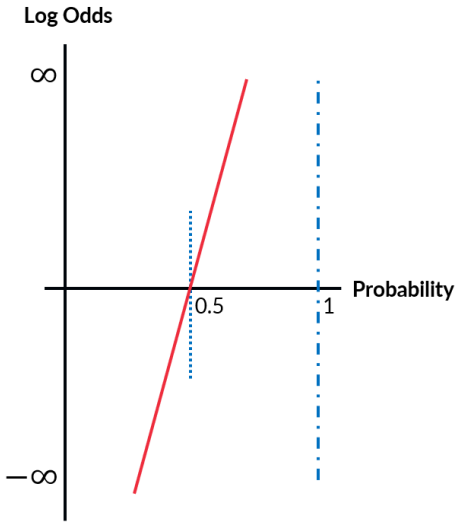
By transposing the sigmoid function, we can also say,

$$e^z = \frac{p}{1-p} = \text{odds}$$

We can represent odds as linear by taking the log of the odds:

$$z = \log(\text{odds}) = \log\left(\frac{p}{1-p}\right) = \text{logit}(p)$$

In other words, the linear model that we pass to the sigmoid function is the log of these odds.



Log-odds transform these into a two-sided linear model.

These log odds are also called the logit, and the logit function transforms the non-linear relationship between *p* and *z* (and because of the linear relationship, **β** as well) into a linear relationship. The logistic function transforms this linear predictor into a probability *p*.

Coefficient Interpretation

A coefficient β_j represents the change in log-odds for a one-unit change in x_j, holding all other predictors constant

If we take the difference in log-odds of x and $x + 1$ and exponentiate it, we get the odds-ratio,

$$\text{Odds Ratio} = \theta = \frac{\text{odds}(\text{x}_\text{j} + 1)}{\text{odds}(\text{x}_\text{j})} = \text{e}^{\beta_\text{j}}$$

This odds ratio indicates the percentage change in odds for each predictor for a unit increase in the predictor.

θ > 1 represents higher odds; θ < 1 represents lower odds

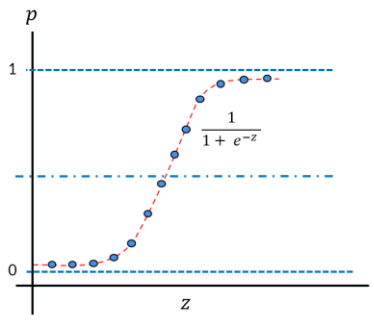
- $\theta_j = 2$: Doubled odds per unit increase in x_j
- $\theta_j = 0.5$: Odds are halved per unit increase in x_j
- $\theta_j = 1$: No change

Because the probability comes from sigmoid, the same change in log-odds is not linear and has different effects on probability at different places. The derivative of sigmoid (slope) is:

$$\frac{\text{d}p}{\text{d}z} = p(1 - p)$$

- Near $p = 0.5$: even a small Δz causes large Δp ($\rightarrow 0.25$)
- Near $p = 0$ or 1: the same Δz causes tiny Δp ($\rightarrow 0$)

So, we have a steep middle and straighter extremes on the sigmoid.



Usually, odd ratios are reported instead of just raw coefficients (due to predictors being on different scales.)

Impact of Coefficients on the Logistic Model

Now, consider a linear model with one feature.

$$p = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1)}}$$

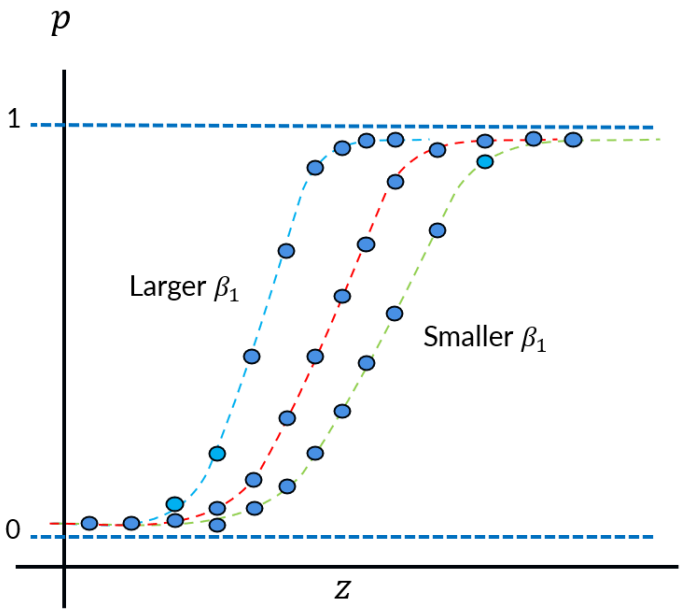
Taking derivative w.r.t. x_1

$$\frac{dp}{dx} = p(1-p) \cdot \beta_1$$

Essentially, this is just the slope of the sigmoid function, scaled by the coefficient of x_1:

- Larger β_1 → Steeper (narrow transition)
- Smaller β_1 → Flatter (gradual transition)

So, the predicted probabilities will also change:



Assumptions of Logistic Regression

Like any statistical model, logistic regression is built on certain assumptions. Understanding these helps us know when the model is appropriate and when its results may be misleading.

1. Binary or Categorical Dependent Variable

- Logistic regression assumes that the target variable is categorical (often binary: 0 or 1)
- For multiple classes, we use approaches like one-vs-rest or softmax

2. Independence of Observations

- Each observation is assumed to be independent of the others
- For example, in time series or clustered data, outcomes are correlated, and violations occur. In such cases, specialised models are more appropriate.

3. Linearity of the Logit

- Logistic regression assumes a linear relationship between predictors and the log-odds of the outcome:

$$log(odds) = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p$$

- This does not mean the predictors must be linearly related to the probability p itself
    - The linearity is with respect to the logit function

4. No Multicollinearity

- Predictors should not be highly correlated with each other.
- Severe multicollinearity makes it difficult to isolate the individual effect of predictors and inflates standard errors
- Variance Inflation Factor (VIF) is often used to check this

5. Appropriate Sample Size

- Logistic regression uses maximum likelihood estimation, which performs best with moderately large to large sample sizes, because estimates stabilise with more data
- Small datasets can lead to unstable or biased coefficient estimates

6. Correct Model Specification

- The model should include all relevant predictors, and the relationship should be correctly specified
- Omitted variables or incorrect functional forms can bias results

Cross-Validation and Regularisation

Generalisation and Complexity

Up to this point, we have studied linear regression and logistic regression in detail. These models allow us to predict numerical outcomes (linear regression) or categorical outcomes (logistic regression).

Both models rely on certain assumptions for their validity, such as linearity, independence of observations, and limited multicollinearity. However, even when assumptions hold, some challenges remain.

We need to check for two aspects of the model:

1. **Generalisation** – A model may learn the patterns in training set too well and overfit. How to ensure the model performance on unseen data?
2. **Complexity** – As models become more flexible (for example, by including many predictors or polynomial transformations), they risk becoming unstable. Small changes in the data may lead to large changes in predictions. How do we prevent the model from overfitting to noise in the training data?

These two concerns are addressed by:

- **Cross-Validation**, a method to reliably estimate performance on new data
- **Regularisation**, a set of techniques to control model complexity and improve generalisation

Importantly, these are **not limited to regression**. They apply broadly to regression models and more advanced machine learning algorithms as well.

Cross-Validation

We have mainly used the dataset by splitting it into two sets: training and testing. Sometimes, we also use a subset of the training dataset, known as validation set, to keep a track of performance during training. While this is a logical and straightforward procedure, it has some limitations, especially when working with smaller datasets.

Moreover, how do we ensure that the performance will not change if we take different samples of the population?

Suppose our dataset is small in size. If we randomly select 80% of the data for training and 20% for testing, we might accidentally end up with an unbalanced split. For instance, the training set could miss important examples or have a different distribution from the test set.

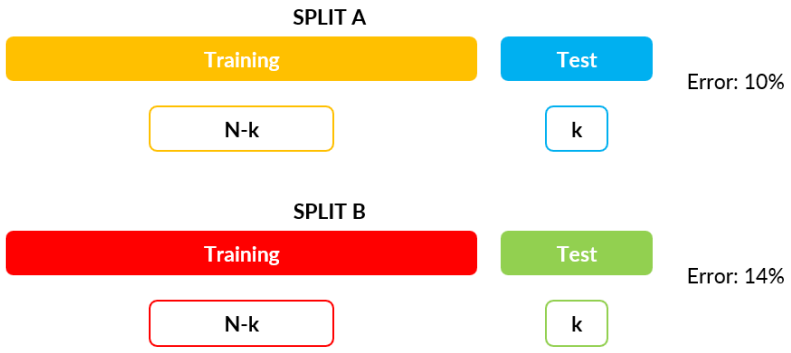
This could lead to two problems:

1. **Unstable performance estimates.**

   Different random splits may give very different accuracy or error values. A model might do well on one split and poor on another, leaving us uncertain about its true performance.

2. **Inefficient use of data.**

   Data points that are kept aside for testing cannot be used in training. With small datasets, this waste of information can be costly, reducing the quality of the trained model.



To address these problems, we need a method that uses all the data effectively while still giving us a reliable estimate of how the model will perform on unseen data. This is where cross-validation comes in.

Cross-Validation - Idea

The key principle behind cross-validation is simple: instead of making one arbitrary train-test split, we rotate the role of data points. Every data point will get a chance to be part of the training set and also a chance to be part of the test set.

This is done by **partitioning the dataset into multiple subsets or folds**, and then performing repeated rounds of training and testing. After these rounds, the performance metrics are averaged, giving a more stable and reliable estimate.

Formally, suppose we have a dataset $D = (x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ consisting of n observations. Cross-validation divides this dataset into k subsets or folds.

The model is trained and evaluated k times. In each iteration, one fold acts as the test set, while the remaining $(k-1)$ folds are combined to form the training set. By the end, each data point has been used for both training and testing exactly once.

This process provides:

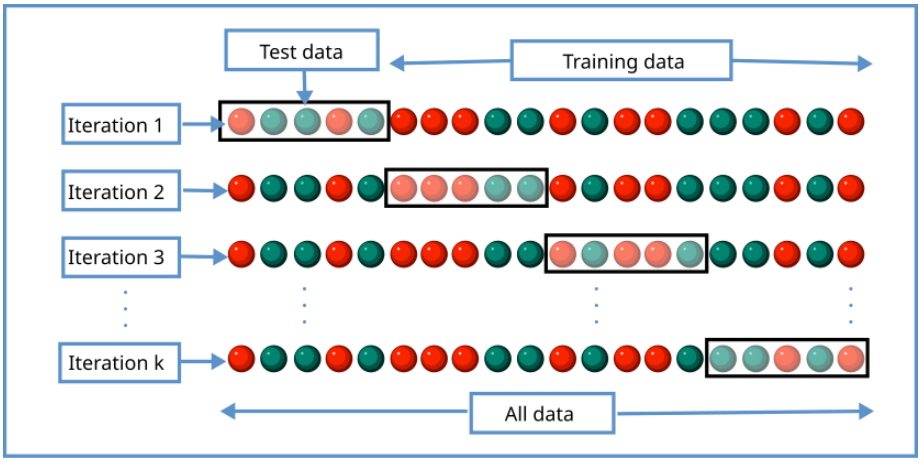
- Reliable performance estimation, as the final metric is an average over all folds
- Better use of data, since every data point contributes to training in most of the iterations

k-Fold Cross-Validation

The most common form of cross-validation is **k-fold cross-validation**. In this method, the dataset is divided into k equal (nearly equal) parts, or folds.

Let us consider an example with $k = 5$. The process would proceed as follows:

1. In the first round, folds 2, 3, 4, and 5 are combined to form the training set, while fold 1 is used as the test set.
2. In the second round, folds 1, 3, 4, and 5 are used for training, and fold 2 is used for testing.
3. This continues until each fold has been used once as the test set.



At the end of this process, we calculate the average of the performance metrics across all five rounds. This gives us the **cross-validated performance estimate** of the model.

The choice of k is important:

- Smaller values of k (like $k = 5$) use less computational power but can lead to higher bias (erroneous)
- Larger values of k (like $k = 10$) provide a more accurate estimates but require more computation
- When $k = n$, meaning each fold has exactly one data point, we reach a special case called **Leave-One-Out Cross-Validation (LOOCV)**

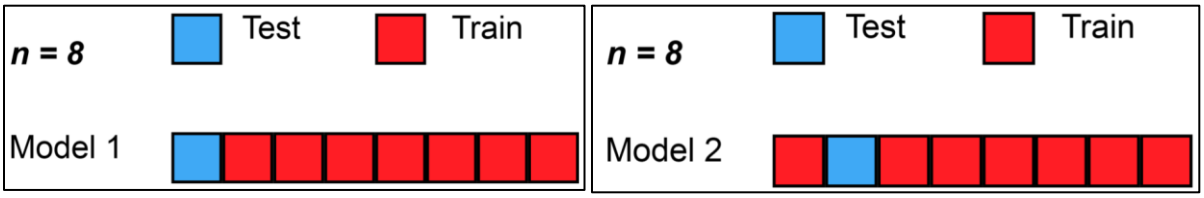
Commonly, we use $k = 5$ or $k = 10$ as a balance between efficiency and reliability.

Leave-One-Out Cross-Validation (LOOCV)

Leave-One-Out Cross-Validation represents an extreme form of cross-validation. Here, each iteration leaves exactly one observation out as the test set, while all other observations are used for training.

If the dataset has n points, then the model is trained n times:

- In round 1, the first point is left out for testing, and the rest are used for training
- In round 2, the second point is left out, and so on, until every point has been used once as the test set



This method has the advantage of using nearly all the data for training in each iteration, which can be beneficial when data is scarce.

However, it is very computationally expensive, especially for large datasets. Moreover, the estimates can have high variance, as each test set contains only one point.

Due to these reasons, LOOCV is mainly of theoretical interest or used in situations where every data point is valuable, such as in medical studies with very small sample sizes.

Other Variations of Cross-Validation

1. **Stratified Cross-Validation**

   When dealing with classification problems with imbalanced classes, it is important to preserve the class proportions within each fold. Stratified cross-validation ensures that each fold has roughly the same distribution of classes as the full dataset.

2. **Repeated k-Fold Cross-Validation**

   The standard k-fold CV process can be repeated multiple times with different random splits of the data into folds. This reduces variability further and produces a more stable performance estimate.

3. **Time-Series Cross-Validation**

   In time-series problems, the temporal order of data must be respected. Hence, data from the future should never be used to predict the past. Special rolling-window or expanding-window cross-validation techniques are used in these cases.

Parameters and Hyperparameters

Before we learn regularisation and tuning, it is important to understand about **parameters** and **hyperparameters**. Although the two words sound similar, they refer to very different concepts.

Parameters

Parameters are the values that the model learns directly from the data during training:

- Linear regression learns regression coefficients ($\beta_0, \beta_1, \ldots, \beta_p$) by minimising the sum of squared errors
- Logistic regression learns coefficients that best separate the classes by maximising the likelihood of the observed outcomes

These parameter values (also known as weights) change whenever the dataset changes. In short, **parameters define the model**, and the learning process estimates their best values.

Example:
If our linear regression model for predicting house prices is

$$\hat{y} = \beta_0 + \beta_1 \cdot \text{size} + \beta_2 \cdot \text{location}$$

the numbers we compute for $\beta_0, \beta_1, \beta_2$ are **parameters**.

Hyperparameters

Hyperparameters, in contrast, are **settings we choose before training begins**. They do not come from the data but instead guide the learning process itself or control the structure of the model.

Up until now, we have not encountered any significant hyperparameters. An example can be found in the learning rate of optimisation algorithms. The learning rate controls the step size. It can be considered a hyperparameter as it is set before training begins and guides the parameter update process but is not learned directly from the data.

If we increase the learning rate, the algorithm may overshoot the optimal coefficients and fail to converge. If we set a very low learning rate, the algorithm will take many iterations to converge and reach the optimal values.

However, as we introduce regularisation, we will see an important hyperparameter: the **regularisation strength**, often denoted by α (or sometimes C in logistic regression).

This value controls how strongly we penalise large coefficients.

- A small penalty allows the model to fit flexibly, possibly leading to overfitting
- A large penalty enforces simplicity, potentially underfitting the data

Choosing a good value for a hyperparameter like α is crucial because it shapes the final model. This requires experimentation and tuning.

Example:

In Ridge regression, we must set the value of α. If we pick $\alpha = 0$, the model behaves like ordinary linear regression. If we pick a very large α, the coefficients are heavily shrunk towards zero.

Parameters are **determined by the model automatically** based on the training data. Hyperparameters are **chosen by us**, and different choices can lead to very different outcomes.

This is where cross-validation is important:

- For each candidate value of a hyperparameter, we can run cross-validation to measure how well the model performs
- The hyperparameter that yields the best average performance is selected

Thus, cross-validation and hyperparameter tuning go together.

Regularisation

We know how to build linear and logistic regression models to fit data and make predictions. These models work by learning parameters that minimise some loss function, such as the sum of squared errors in linear regression or the log-likelihood in logistic regression.

When applied carefully, these models are powerful tools. However, in real-world situations, especially when the dataset is complex or has many predictors, we may encounter overfitting.

Imagine we are predicting house prices using factors like area, number of bedrooms, distance to the city centre, and several neighbourhood-related variables.

- A simple model with only two predictors might miss some important relationships: this is underfitting.
- A very complex model with dozens of predictors, polynomial terms, and interactions might fit the training data perfectly, even capturing unusual quirks specific to that dataset. This is overfitting.

When overfitting occurs, the model's coefficients can grow very large or fluctuate wildly. This makes the model unstable: even small changes in the data can lead to drastically different predictions.

The Need for Regularisation

To address overfitting, we need a way to control the complexity of the model.

Regularisation adds a penalty term to the usual loss function, discouraging the model from fitting the data too closely by penalising large coefficient values.

The updated loss function becomes:

$$\text{Regularised Loss} = \underbrace{\text{Loss}}_{\text{Fit to data}} + \underbrace{\alpha \,.\, \text{Penalty}}_{\text{Complexity control}}$$

The original loss measures how well the model fits the data while the penalty term acts like a brake, preventing coefficients from growing too large. The factor α (alpha) controls the strength of the penalty.

Think of regularisation as giving the model a gentle nudge toward simplicity.

- With no penalty ($\alpha = 0$), the model behaves like ordinary regression and may overfit
- With a small penalty, the model remains flexible but avoids extreme coefficient values
- With a very large penalty, the model becomes overly simple, potentially underfitting

Regularisation for Linear Regression

Linear regression fits a line or plane that minimises the MSE between the predicted and actual values. The loss function for ordinary least squares (OLS) regression is:

$$\text{Loss} = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

where $\hat{y}_i$ is the prediction for observation i.

While OLS works well under ideal conditions, it may overfit when the model becomes too complex or when there are many predictors:

In linear regression, the most basic cause of overfitting is many noisy or irrelevant features. Overfitting expresses itself in linear regression in the form of very large coefficient values. The model stretches the coefficients to extreme magnitudes to perfectly match the training points, making it unstable.

The regularisation penalty discourages the model from assigning very large values to its coefficients. We fit the model to the data as usual, but at the same time penalise large coefficients.

The new loss function becomes:

$$\text{Loss} = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 + \alpha \cdot \sum |\beta|^l$$

α determines how much importance is given to the penalty term relative to the usual OLS loss. l defines the type of regularisation.

- When $\alpha = 0$, the model behaves like ordinary linear regression with no penalty
- As α increases, the penalty grows stronger, shrinking the coefficients and simplifying the model

There are two widely used regularisation techniques for linear regression:

1. **Ridge Regression (L2 penalty)**
2. **Lasso Regression (L1 penalty)**

Both methods add a penalty to the coefficients but in slightly different ways, leading to different effects.

Ridge Regression (L2 Regularisation)

In Ridge regression, the penalty term is the sum of the squares of the coefficients. Or, we take the value of l in the regularised loss as 2.

$$\text{Loss}_{ridge} = \sum(y - \hat{y})^2 + \alpha \sum |\beta|^2$$

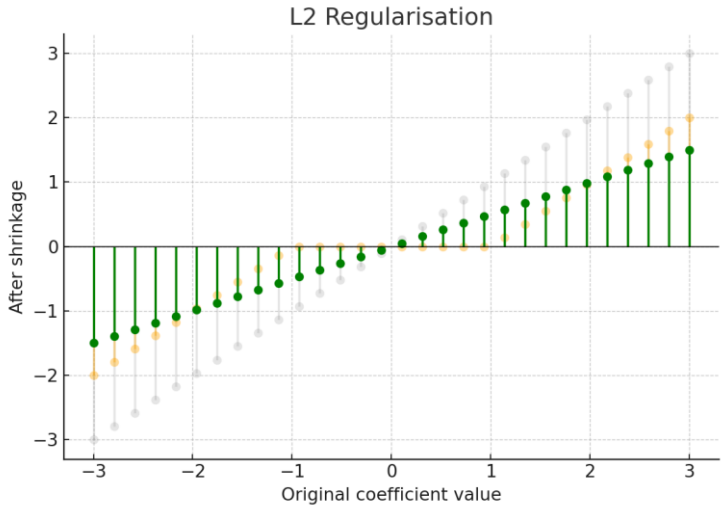
Here:

- β_j represents the coefficient for predictor j
- The summation $\sum \beta_j^2$ is called the L2 norm

The squared penalty term discourages coefficients from becoming too large. It **shrinks them smoothly towards zero** but **rarely exactly to zero**. It penalises larger coefficients more because of the squaring.

Ridge regression is effective when there is multicollinearity, and we want to keep all predictors in the model while controlling their influence.

With a larger α, the coefficients are pulled closer to zero. With a smaller α, the shrinking effect on coefficients is lighter.



Lasso Regression (L1 Regularisation)

In Lasso regression, the penalty term is the sum of the absolute values of the coefficients. Here, we take the value of l in the regularised loss as 1.

$$\text{Loss}_{lasso} = \sum (y - \hat{y})^2 + \alpha \sum |\beta|^1$$

Where:

- β_j represents the coefficient for predictor j
- The summation $\sum \beta_j^1$ is called the L1 norm

The absolute value penalty behaves differently from the squared penalty used in Ridge regression. It shrinks some coefficients smoothly, similar to Ridge, but it can also **force certain coefficients to become exactly zero** when α is large enough.

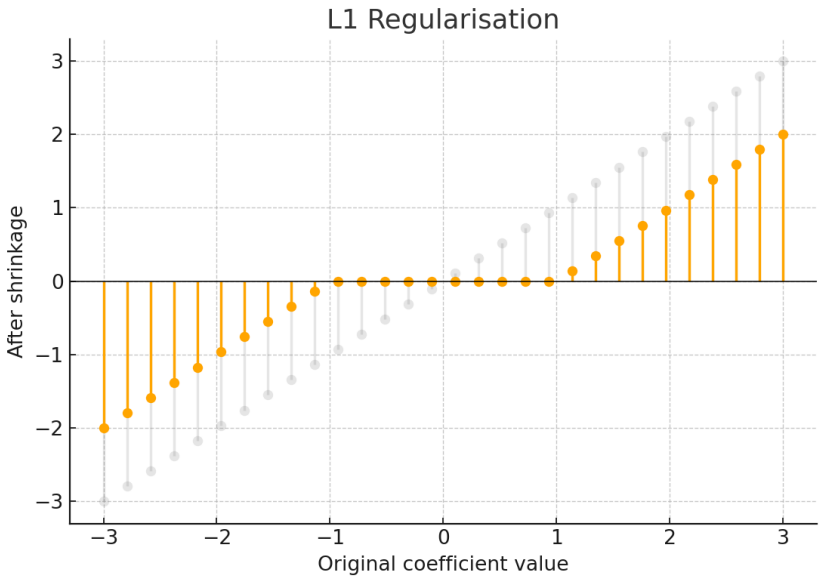
This means Lasso not only reduces the magnitude of coefficients but can remove predictors entirely from the model. As a result, Lasso performs **automatic feature selection**, creating simpler and more interpretable models.

Ridge keeps all predictors but with reduced influence, while Lasso can eliminate unimportant predictors by setting their coefficients to zero.

Lasso regression is particularly useful when only a subset of predictors is important, we want a more interpretable model with fewer variables, or there is a need for automatic feature selection, especially in high-dimensional datasets.

A small α adds a mild penalty, most coefficients stay non-zero. A moderate α shrinks some coefficients exactly to zero and reduces others, simplifying the model.

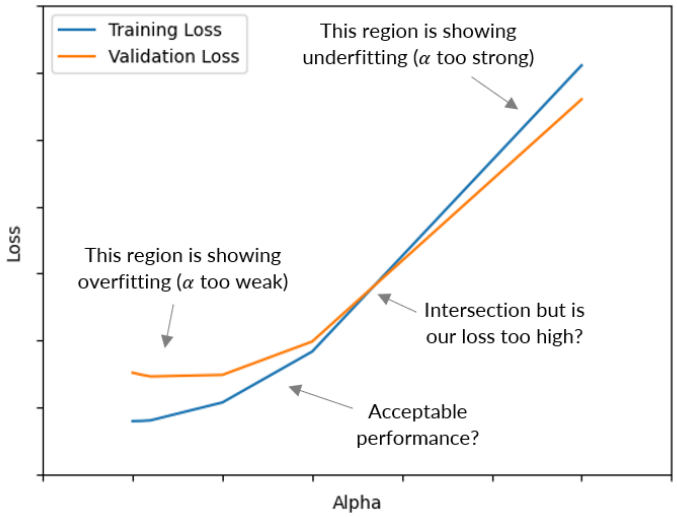
A very high penalty forces most or all coefficients to zero, leading to underfitting.



The choice of α is therefore crucial, as it determines how aggressively the model performs feature selection.

For finding the optimal value in both ridge and lasso, we use hyperparameter tuning. While tuning the α, we train the model multiple times for different values of the hyperparameter, ultimately choosing the value which gives the best result.

Tuning can be done in various ways, for example, using basic loops for different values for alpha, or by using *GridSearchCV*, a common cross-validation functionality provided by the *sklearn* library.



Regularisation for Logistic Regression

Logistic regression, like linear regression, can overfit when there are many predictors or when the model is too complex. Regularisation helps by adding a penalty term to the **negative log-likelihood (log-loss)**, discouraging very large coefficient values and improving generalisation. The regularised loss is:

$$\text{Loss} = -\sum y \log \hat{y} + (1-y)\log(1-\hat{y}) + \alpha \sum |\beta|^l$$

Here, $\hat{p}_i$ is the predicted probability for observation i.

The parameter α controls the strength of regularisation, and l defines the type of regularisation: $l = 2$ for ridge and $l = 1$ for lasso.

When $\alpha = 0$, the model behaves like ordinary logistic regression. As α increases, the penalty grows stronger, shrinking coefficients and simplifying the model.

In *scikit-learn*, the regularisation strength is specified using **C**, which is the inverse of α.

- A **large C** corresponds to a small penalty (weak regularisation)
- A **small C** applies a strong penalty (strong regularisation)

The value of α or C is a hyperparameter and must be tuned using cross-validation to find the balance between underfitting and overfitting.

Ridge Regularisation (L2)

Ridge regression uses the sum of the squares of the coefficients as the penalty term. It is equivalent to setting $l = 2$in the loss function:

$$\text{Loss}_{ridge} = -\sum y \log \hat{y} + (1-y)\log(1-\hat{y}) + \alpha \sum |\beta|^2$$

The L2 penalty **smoothly shrinks coefficients towards zero** but does not make them exactly zero.

Ridge is useful when there are many correlated predictors or when we want to keep all features in the model while controlling their influence.

As α increases (or C decreases), the coefficients are shrunk more strongly, reducing variance but increasing bias. With very small α(or large C), the model behaves like ordinary logistic regression.

Lasso Regularisation (L1)

Lasso regression uses the absolute values of the coefficients as the penalty term. It is equivalent to setting $l = 1$ in the loss function:

$$\text{Loss}_{lasso} = -\sum y \log \hat{y} + (1-y)\log(1-\hat{y}) + \alpha \sum |\beta|^1$$

The L1 penalty can **force some coefficients exactly to zero**, effectively removing certain features from the model.

This makes Lasso useful when we suspect that only a few predictors are truly important, or when we want a simpler, more interpretable model through automatic feature selection.

As α increases (or C decreases), more coefficients are pushed to zero, simplifying the model further.