Introduction to Machine Learning

Dr Niamh Mimnagh

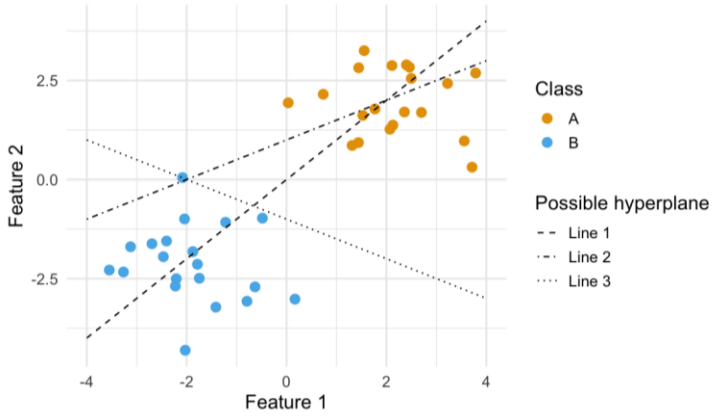
niamh@prstats.org

https://github.com/niamhmimnagh/IMLR04-Introduction-to-Machine-Learning

PR STATS

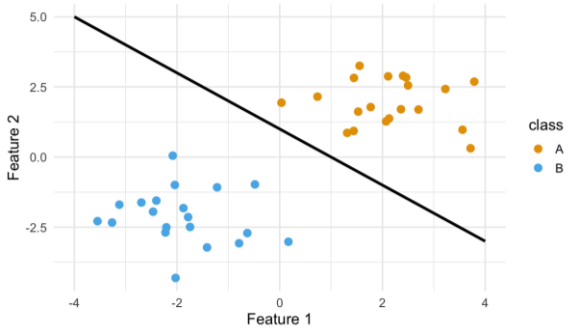
Support Vector Machines

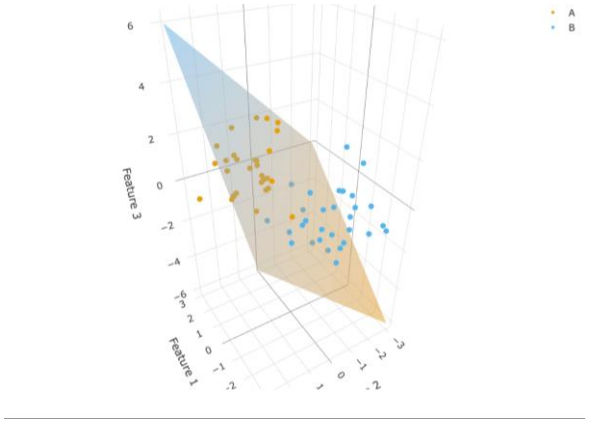
- Support Vector Machines are supervised learning models that can be used for both classification and regression problems.
- The main idea of an SVM is to find the optimal hyperplane that separates the data into different classes.
- Imagine you have two classes that can be separated by a line. There are many possible separating lines. The goal is to find the optimum line for separating the classes.



Support Vector Machines

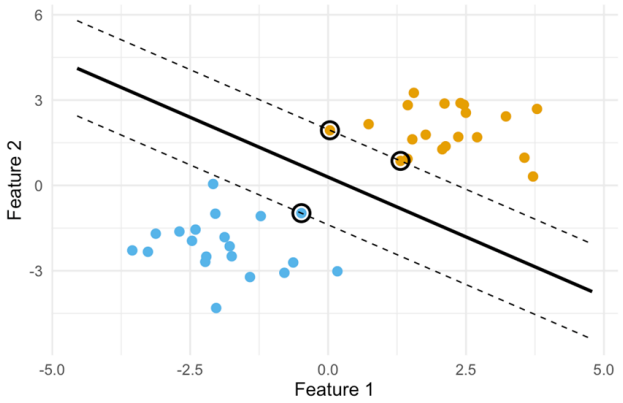
- A hyperplane is the decision boundary that divides the data, which is a line in two dimensions, a plane in three dimensions, and a hyperplane in higher dimensions.
- Equation of a hyperplane: $wx + b = 0$
- w = weights (direction), b = bias (offset)
- SVMs choose the hyperplane that maximises the margin (distance between the hyperplane and the nearest data points from each class).
- The data points that lie closest to the boundary are called support vectors, and they are the most important points for defining the classifier.





Support Vectors

- Support vectors are the training data points that lie closest to the decision boundary (hyperplane).
- These points are the most important because they define the margin and determine the position of the optimal hyperplane.
- An SVM seeks to maximise the margin between classes, as a larger margin usually leads to better generalisation to unseen data.
- By maximising the margin, the model becomes less sensitive to shifts or noise in the training data.
- Mathematically, the optimisation problem can be expressed as minimising the norm of the weight vector ‖w‖ while ensuring that all training points are correctly classified.



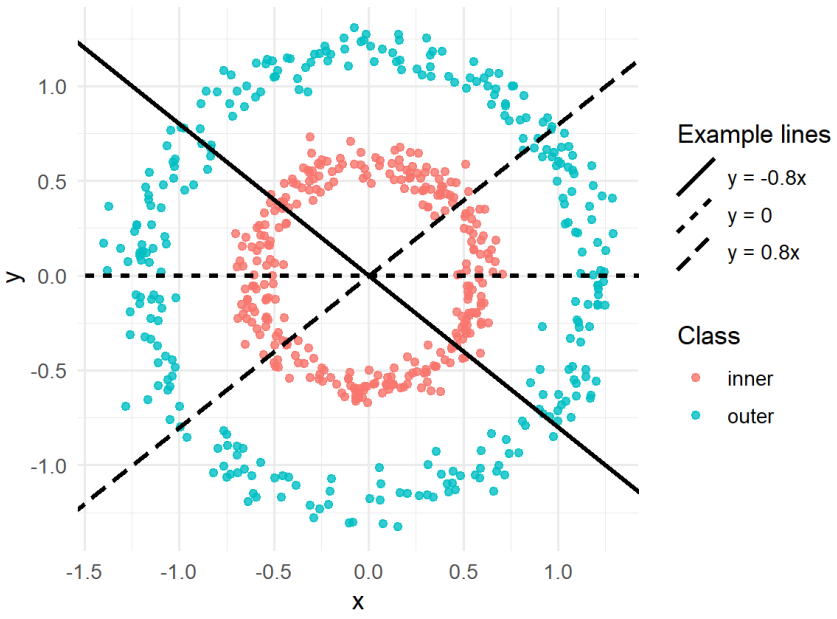
Linear SVMs in R

- A linear SVM fits a straight hyperplane and is a strong baseline when the classes are approximately linearly separable.
- The key hyperparameter is cost, which controls the trade-off between a wider margin and misclassification; larger values enforce fewer violations at the risk of overfitting.
- It is good practice to standardise numeric predictors before fitting a linear SVM, because the margin depends on feature scales.

```
rec  <- recipe(Species ~ ., train) %>%
step_zv(all_predictors()) %>%
step_normalize(all_numeric_predictors())
spec <- svm_linear(mode = "classification", cost = tune()) %>%
set_engine("LiblineaR")
```

Limitations of Linear SVMs

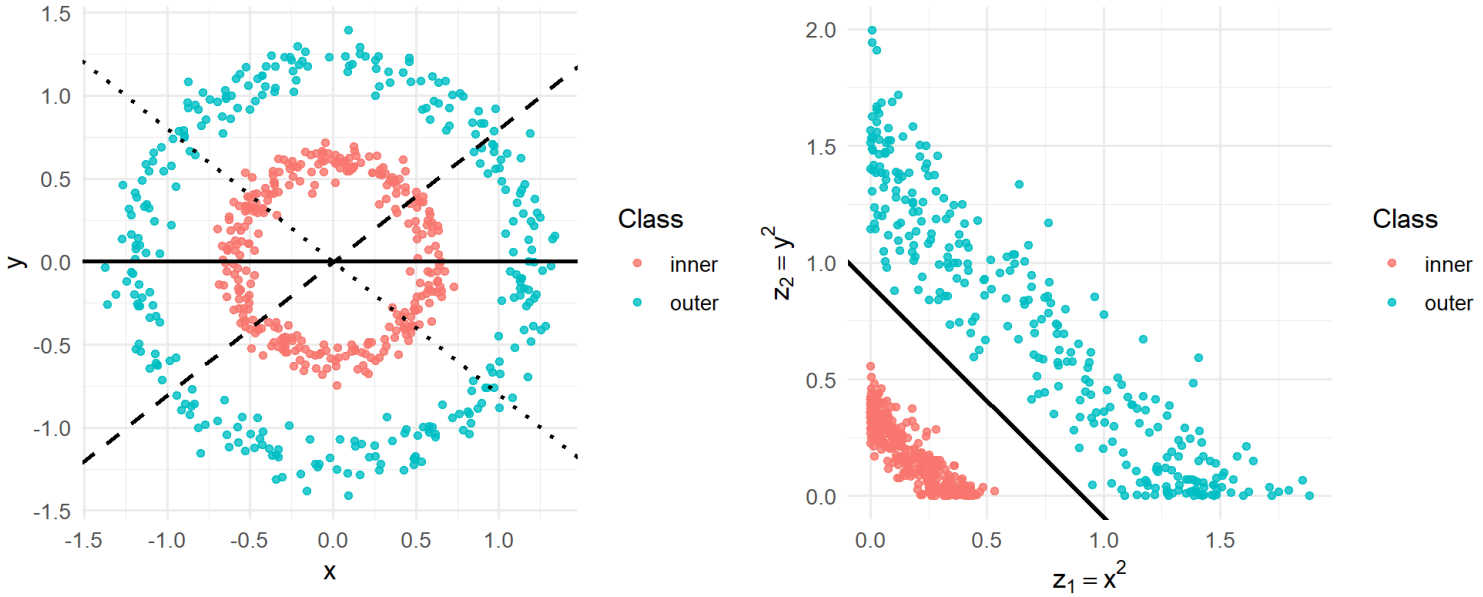
- Linear SVMs assume that the data can be separated by a straight hyperplane, which is not always the case in real-world problems.
- When the classes are not linearly separable, a linear decision boundary will lead to poor classification performance.
- In such situations, the data can be transformed into a higher-dimensional space where a linear separation becomes possible.



Kernels in SVMs

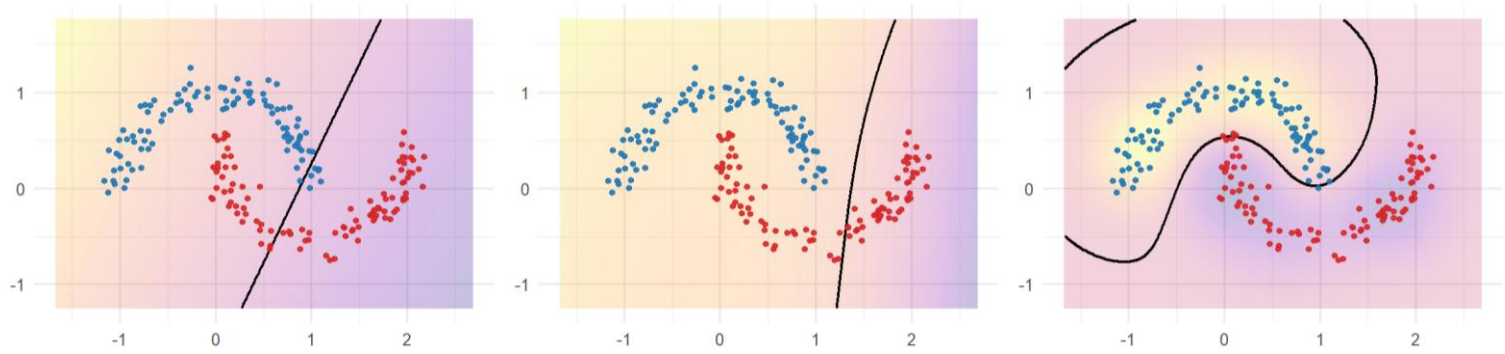
- A kernel is a mathematical function that maps the original data into a higher-dimensional space so linear separation may become possible.
- Kernels allow Support Vector Machines to capture non-linear relationships between variables without explicitly computing the transformation into higher dimensions.
  - The linear kernel is simply the standard dot product of the input vectors and is used when the data are already linearly separable.
  - The polynomial kernel adds polynomial terms of the input features, which enables the model to capture more complex curved boundaries.
  - The Radial Basis Function (RBF) kernel maps points based on their distance from each other, creating flexible and highly non-linear decision boundaries.

Kernels in SVMs



Kernels

- The linear kernel assumes classes can be separated by a straight hyperplane in the original feature space. It is simple, efficient, and often a good first choice.
- The polynomial kernel introduces curved decision boundaries by including polynomial combinations of the features. It is useful when interactions between variables are important, but it can become computationally expensive and sensitive to the choice of polynomial degree.
- The RBF kernel measures similarity based on distance, producing flexible non-linear boundaries. It is widely used because it can adapt to complex shapes, but it requires tuning of its parameters to avoid overfitting.

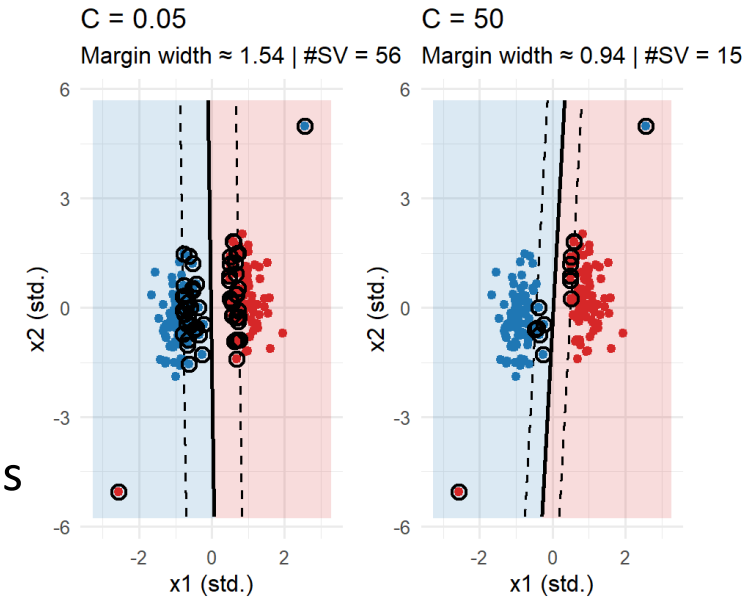


When to Use Which Kernel

- Begin with a linear kernel as a baseline because it is fast, has only cost to tune, and often performs competitively when features are high-dimensional or sparse.
- If the linear model underfits and you see systematic errors, move to a non-linear kernel and compare performance with cross-validation.
- Use the RBF kernel as the default non-linear choice because it is flexible, makes few assumptions about boundary shape, and only requires tuning cost and rbf_sigma.
- Prefer the polynomial kernel when you expect interactions of a specific order or want explicit control over boundary smoothness, and keep the degree low (typically 2–3) to limit overfitting.

Soft Margins

- Real datasets often have overlap, outliers, or noisy labels, so a perfect separating line would overfit.
- A soft-margin SVM allows a number of mistakes on the training data (via “slack” or wiggle room) so the boundary can be more stable.
- The cost parameter controls the trade-off: a large cost tries hard to classify every training point correctly, which makes the margin narrow and can overfit to noise. A small cost is more tolerant of training errors, which keeps the margin wider and often improves generalisation, but it can underfit if set too low.



Hyperparameters

	Hyperparameters	Description & Effect
Linear SVM	cost	Controls penalty for margin violations. Larger values reduce misclassifications but risk overfitting.
Polynomial SVM	cost	
	degree	Order of the polynomial. Higher degrees create more flexible, curved boundaries but may overfit.
	scale_factor (coef0)	Shifts the polynomial and interacts with degree, changing the shape of the boundary.
RBF SVM	cost	
	rbf_sigma	Kernel width: small values can give wiggly, overfit boundaries; large values can give smooth, underfit boundaries.

Nonlinear Boundary with RBF Kernel

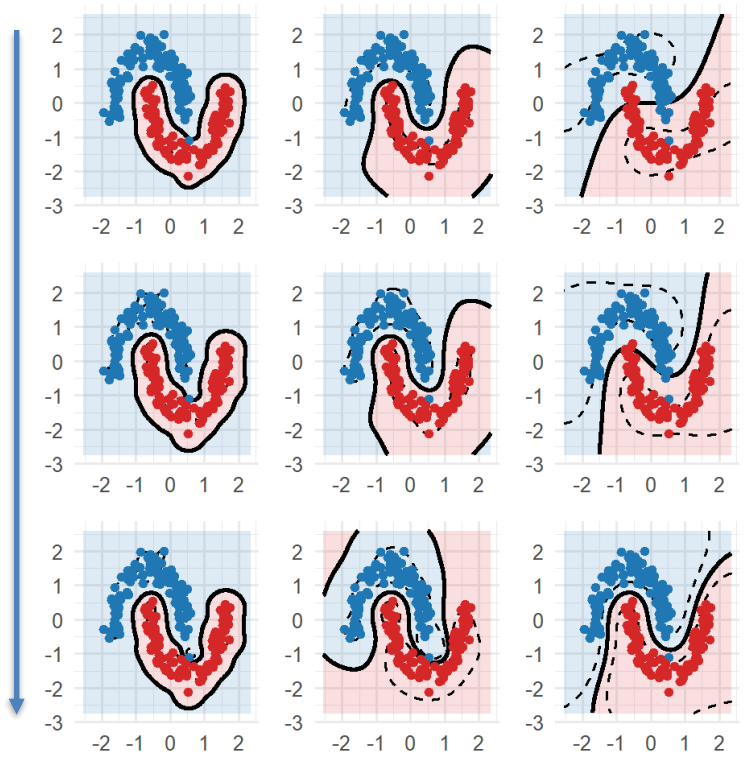
- An SVM with an RBF kernel maps points by their pairwise distances, which allows the model to learn highly flexible, non-linear decision boundaries.
- The two key hyperparameters are cost, which controls the penalty for margin violations, and rbf_sigma, which controls the kernel width; both should be tuned because they jointly determine model smoothness vs. overfitting.
- It is good practice to remove zero-variance predictors and standardise numeric features before fitting, since SVMs are sensitive to feature scales.

```
spec <- svm_rbf(mode = "classification",
                cost = tune(), rbf_sigma = tune()) %>%
  set_engine("kernlab")
```

Nonlinear Boundary with RBF Kernel

Kernel width σ: small → large
local, wiggly → smooth, global

Cost:
small
↓
large



wider, tolerant margins
↓
Strict, narrow margins

Example: Spam

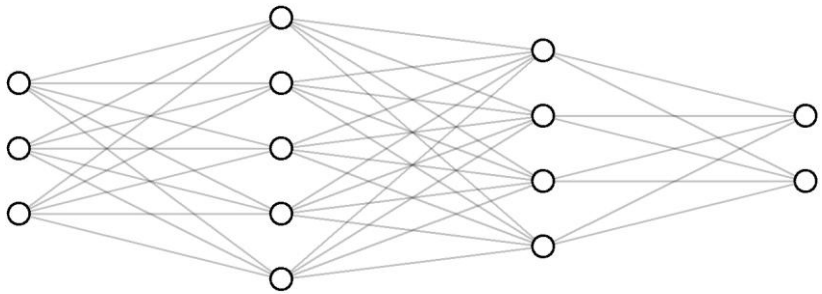
- In this example, we will use the DAAG::spam7 dataset to predict whether an email is spam or not spam.
- The predictors are word and character statistics that often relate to the spam label in non-linear and interacting ways, so a purely linear boundary is unlikely to be sufficient.
- We will fit a Support Vector Machine with a Radial Basis Function (RBF) kernel in tidymodels, tuning cost and rbf_sigma with stratified k-fold cross-validation on the training set and evaluating performance on a held-out test set.
- Our preprocessing will remove zero-variance predictors and standardise numeric features so that distances and margins are comparable across variables.

Advantages and Limitations

- SVMs are effective in high-dimensional spaces because they focus on the support vectors and the margin rather than modelling the full distribution of features.
- SVM models are relatively memory-efficient at prediction time, because decisions depend primarily on the support vectors rather than all training examples.
- SVMs require careful tuning of the parameters and sub-optimal settings can lead to underfitting or overfitting.
- SVMs are sensitive to feature scaling, so numeric predictors must be standardised; otherwise, distances and margins are not comparable across features.

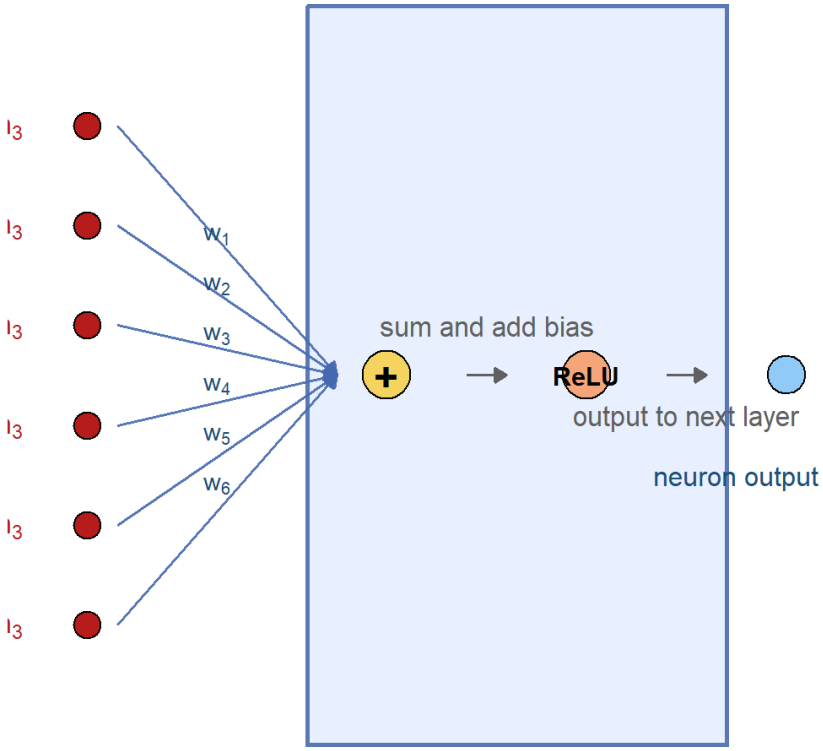
Neural Networks

- A neural network is a machine learning model loosely inspired by biological neurons.
- It is built from many simple neurons arranged in layers that transform inputs step by step.
- Neural networks are useful because they can model complex patterns in images, text, audio, and tabular data.



The Basic Neuron

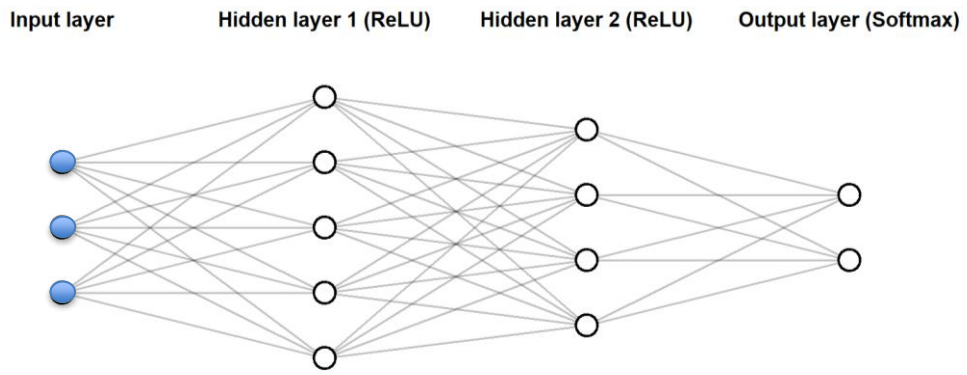
- Each neuron receives a set of numbers from the previous layer, one number per incoming connection.
- A neuron computes a weighted sum $z = w^{T}x + b$ of its inputs and then applies a non-linear activation a = σ(z).
- The weights represent how important each input is, and the bias shifts the overall activation.
- The neuron passes this score through an activation function and outputs this.
- The neuron's output becomes an input to the next layer or the final prediction.



Common Activation Functions

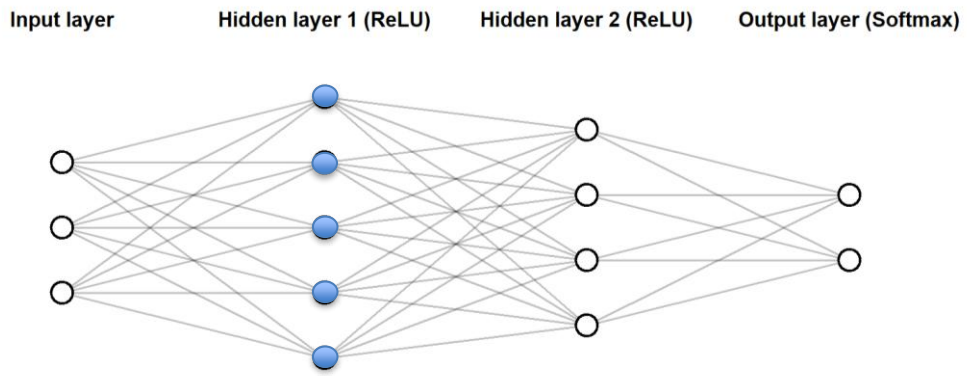
- Without activations, stacking layers collapses to just one linear transformation, which cannot learn curved relationships.
- ReLU (Rectified Linear Unit): returns 0 for negative inputs and passes positive inputs through unchanged. It is the default choice for hidden layers because it trains fast.
- Tanh: maps values to [−1, 1] and is useful when zero-centering helps optimisation.
- Sigmoid: maps values to [0, 1] and is interpreted as a probability. Use it in the final layer for binary classification.
- Softmax: converts a vector of scores into a probability distribution across classes that sums to one. Use it in the final layer for multiclass classification.
- Linear (identity): It leaves the value unchanged. Use it in the final layer for regression when you want real-valued outputs without a bound.

Neural Networks Structure



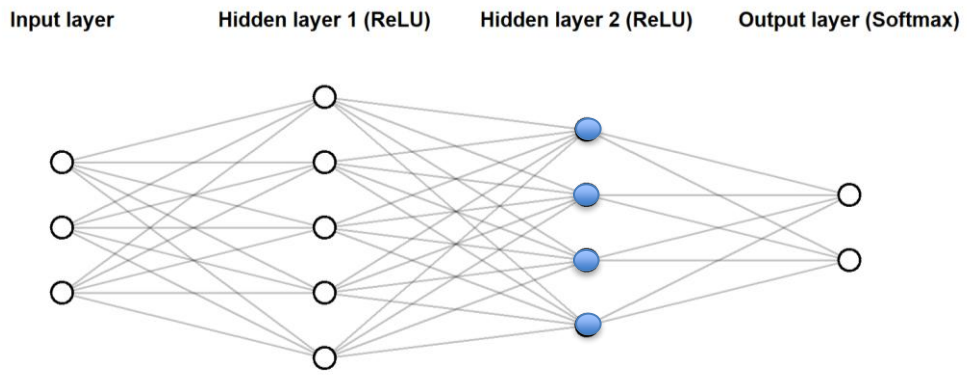
- The input layer receives the feature vector for each example, typically with one node per input feature after preprocessing.
- The input layer does not transform the data; it simply passes the raw feature values forward to the first hidden layer.
- In a standard fully connected network, every input node connects to every neuron in the first hidden layer.

Neural Networks Structure



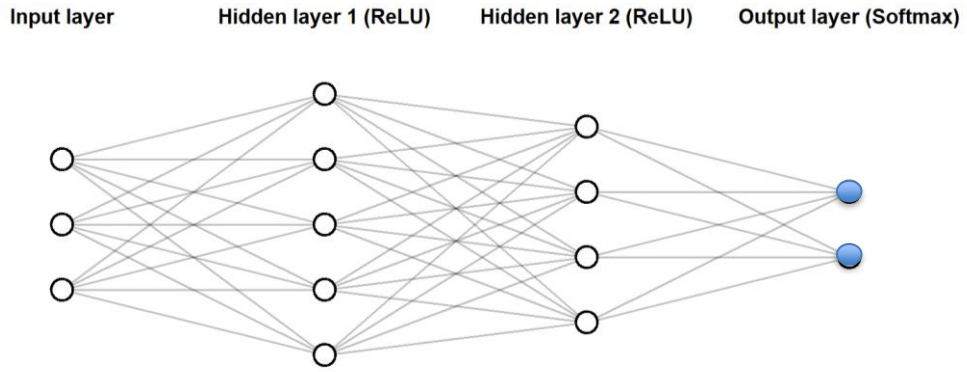
- A hidden layer computes a weighted sum of the incoming values, adds the bias, and applies a non-linear activation to produce its output.
- Hidden layers act as feature builders, transforming the inputs into new, more useful representations that make the final prediction easier to separate.

Neural Networks Structure



- As we move left to right through the network (the forward pass), each layer takes the previous layer's activations and composes them into higher-level patterns.
- The depth (number of layers) and width (units per layer) control the model's capacity.
- Deeper or wider networks can represent more complex functions, but they also require stronger regularisation and more data to avoid overfitting.

Neural Networks Structure



- The output depends on the task:
  - for binary classification, it is usually one neuron with a sigmoid that returns P(positive);
  - for multiclass, it is K neurons with a softmax that return class probabilities summing to 1;
  - for regression, it is one (or more) neurons with a linear activation that output continuous values.
- During training, the network updates weights and biases via backpropagation and gradient descent to minimise a loss function that quantifies how far predictions are from the targets.

Why Use Neural Networks?

- Neural networks are highly flexible function approximators that can learn complex input-output mappings given sufficient data and appropriate regularisation.
- They naturally capture non-linear relationships and interactions between features by stacking layers with non-linear activations.
- They learn hierarchical representations, so early layers detect simple patterns while deeper layers compose them into higher-level features, reducing the need for manual feature engineering.
- They perform strongly across many domains-including images, text, audio, and time series-because architectures like CNNs and Transformers exploit domain structure effectively.

When Not to Use Neural Networks

- When the dataset is small or shallow (e.g., only a few hundred labelled rows), neural nets tend to overfit and underperform regularised linear models, SVMs, or tree-based methods.
- When interpretability is critical (e.g., policy, clinical, or auditor-facing use), neural nets are hard to explain; prefer models with transparent parameters or human-readable structure (trees/GBMs).
- When simpler models perform just as well, you should keep the simpler model; always benchmark baselines first (regularised logistic regression, random forests, gradient boosting), especially on tabular data.
- When you have limited compute or strict latency constraints, neural nets can be heavier to train and serve; lighter models are easier to deploy and maintain.

Neural Networks with nnet

- The nnet engine fits a single-hidden-layer multilayer perceptron, making it a fast, simple baseline for tabular data.
- It works well on small to medium datasets. It is not intended for deep learning (images, long text, many layers).
- Hyperparameters:
  - Hidden units – sets the width of the hidden layer – larger values add capacity but can overfit
  - Penalty – is used to reduce overfitting. Stronger penalty shrinks weights toward zero
  - Epochs – controls how long to train. More epochs can fit better but risk overfitting.

```
mlp_spec <- mlp(
 mode = "classification",
 hidden_units = tune(),
 penalty = tune()) %>%set_engine("nnet")
```

Multilayer Neural Networks

- brulee is an engine that trains fully connected multilayer perceptrons.
- For tabular data, 1–3 hidden layers are usually effective; go deeper only if you have ample data and regularisation.
- Use brulee when you want multiple hidden layers, dropout, finer control over training. Keep nnet for quick, single-layer baselines on small datasets.

```
# -- model spec multilayer MLP (two hidden layers) --
mlp_spec <- mlp(mode          = "classification",
  hidden_units = c(128, 64),  # two hidden layers
activation   = "relu") %>%
  set_engine("brulee", batch_size = 64, verbose = TRUE)
```

Limitations of nnet and brulee

- nnet
  - Only supports one hidden layer, so it cannot represent deeper architectures when the problem truly needs them.
  - Offers few modern training features (no dropout, no learning-rate schedule, limited monitoring), so regularisation is mostly weight decay and early stopping is crude.
  - For many tabular problems, a well-tuned tree ensemble (e.g., GBM) will match or outperform nnet with less effort.
- brulee
  - Focuses on fully connected MLPs for tabular data; it does not build CNNs/RNNs/Transformers, so images, long text, and sequences are better handled with dedicated models.
  - Can overfit small datasets without strong regularisation.
  - Some configurations are restricted (e.g., you should not specify both weight decay and dropout together), and advanced callbacks/early-stopping controls are more limited than in raw torch/keras.

Neural Networks for Regression

- In regression, the network predicts a continuous value, so the final layer has one output unit and uses a linear activation at the end.
- The network is trained with regression losses such as MSE or MAE, and it is evaluated with metrics like RMSE, MAE, and R^2, choosing the metric that best reflects error costs.
- Hidden layers work the same as in classification: they apply weighted sums + biases followed by non-linear activations (typically ReLU) to capture non-linear relationships.
- Preprocessing matters: standardise numeric predictors, one-hot encode categorical variables, and consider transforming skewed targets (e.g., log); remember to invert any target transform for reporting.

Example: Ames Housing Data

- We will use the Ames Housing dataset, which contains around 2,900 real home sales from Ames, Iowa.
- Each row represents a single house sale, and the value we want to predict is the sale price.
- The dataset includes a wide mix of home characteristics such as lot size, living area, number of rooms and bathrooms, year built and remodelled, and features like fireplaces, porches, and heating type. These covariates capture the key drivers of price.
- Our plan is to train a simple neural network to learn the relationship between these characteristics and price, using part of the data to teach the model and the rest to check how well it predicts unseen homes.

Model Interpretability

- Model interpretability means being able to explain how and why a model produced a prediction, at both the overall model level and for individual cases.
- Clear explanations build trust and accountability, which is essential when models affect people and resources.
- Interpretability is critical in high-stakes domains such as medicine, finance, and public policy, where regulations and ethics require understandable decisions.
- Explanations help debug models by revealing data leakage, spurious correlations, and biased signals, which leads to safer, more reliable systems.

Types of Interpretability

- Global interpretability explains how the model behaves on average across the dataset, describing which features matter most and how changing a feature tends to move predictions.
- Local interpretability explains why the model made a specific prediction for one case by attributing contributions to each feature.
- Intrinsic interpretability refers to models that are understandable by design-such as regression with well-prepared features, decision trees, or rule lists where parameters/rules map directly to behaviour.
- Post-hoc interpretability explains complex “black-box” models (e.g., ensembles, SVMs, neural nets) after training using tools like permutation importance, SHAP/LIME, recognising that these explanations are approximations.

Models and Their Interpretability

- Linear regression is highly interpretable because each coefficient shows the expected change in the outcome for a one-unit change in a feature, and signs and magnitudes are easy to explain.
- Decision trees are visual and intuitive as they split on rules that can be followed from the root to a leaf, making predictions traceable step by step.
- Random forests are less transparent and require importance tools such as permutation importance, or SHAP values to summarise which features matter and how they influence predictions.
- Neural networks are black-box models that need external explainers to provide both global insights and case-level attributions.

Variable Importance

- Variable importance measures how much each feature contributes to predictive performance by checking how performance drops when that feature is perturbed.
- You can compute your own, model-agnostic importance for any tidymodels model using permutation importance with the vip package.
- Choose a task-appropriate metric (e.g., RMSE/MAE for regression; ROC AUC/PR AUC or accuracy for classification) and evaluate on a hold-out set or resamples to avoid leakage.
- Repeat the permutation several times to stabilise the estimates.
- This approach puts different models on a common importance scale, so you can compare which features matter across models.

SHAP Values

- SHAP (SHapley Additive exPlanations) assigns each feature a contribution to a single prediction. It is based on Shapley values from cooperative game theory, treating features as “players” and dividing the model’s prediction relative to a baseline.
- A positive SHAP value means the feature pushed the prediction up, while a negative value means it pushed the prediction down.
- In regression, SHAP is in the same units as the target: if the baseline price is $200k and a house is predicted at $260k, the feature attributions might sum to +$60k (e.g., +$40k for living area, +$30k for neighbourhood, −$10k for age).
- In classification, SHAP is typically shown on the probability or log-odds scale: if the baseline spam probability is 0.20 and this email is 0.70, features with positive SHAP increased the probability (e.g., many links), and those with negative SHAP decreased it (e.g., trusted sender).

SHAP Values

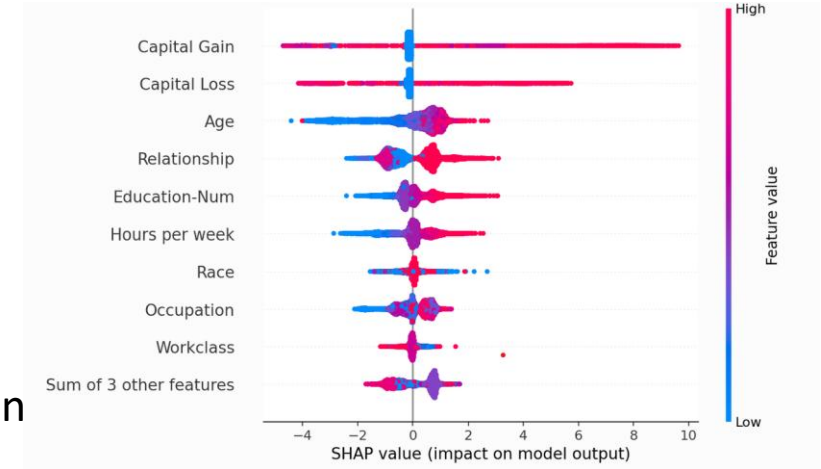
- We can fit a random forest on the iris dataset to predict Sepal Length from the other numeric features, then used SHAP to explain each flower's prediction.
- From the random forest we can obtain the baseline (the model's average prediction for Sepal Length) and the prediction for a certain flower.
- SHAP can then use these to attribute how features move the prediction away from the baseline.
- Baseline: 5.843 cm
- Prediction: 4.666 cm
  - Petal.Length: −0.638 → short petals push the prediction down the most
  - Petal.Width: −0.486 → narrow petals push it down further
  - Sepal.Width: −0.052 → small decrease
- Relative to an average iris, this flower's short petal length is the main reason the model predicts a shorter sepal length.

Advantages of SHAP

- SHAP provides consistent explanations because feature attributions always sum exactly to the prediction relative to a baseline, so the parts add up to the whole.
- It supports local interpretability by explaining an individual prediction with per-feature contributions, which is valuable for case reviews, audits, and decision justifications.
- It also enables global interpretability by aggregating local attributions across many cases, revealing which features matter overall and how their values tend to push predictions up or down.
- Because SHAP uses a common attribution scale, it facilitates model comparison and debugging, making it easier to detect spurious signals, leakage, or drift across different models and time periods.

Beeswarm Plots

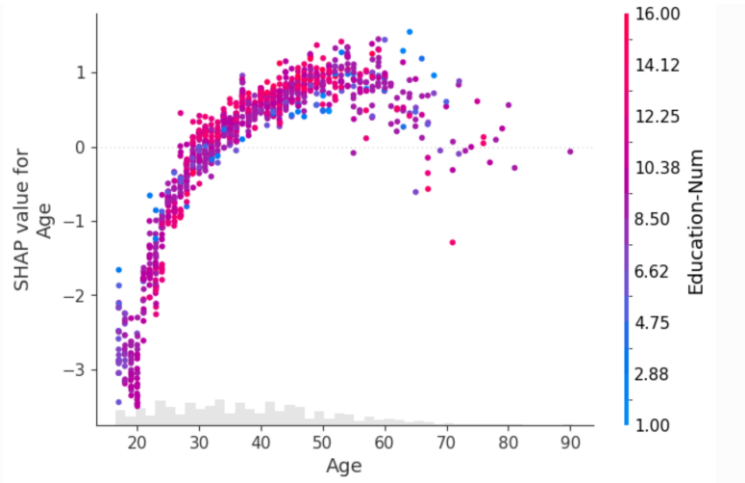
- Each dot represents one observation's contribution to the prediction, measured as a SHAP value relative to the model's baseline.
- Colours encode the feature value for that observation (e.g., blue = low, red = high), so you can see whether large or small values drive the prediction.
- If red dots sit on the right, higher feature values increase the prediction, whereas if red dots sit on the left, higher values decrease the prediction.



- The horizontal spread shows the strength and variability of the feature's influence across cases, with a wider spread indicating a larger impact on the model output.
- Mixed colours indicate that the feature's effect depends on other features (interactions) or is non-monotonic.

Dependence Plots

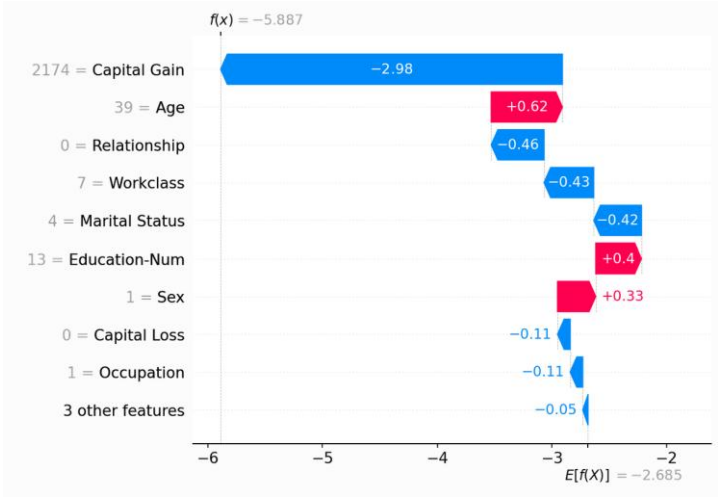
- Each dot represents one observation, with the x-axis showing that observation's value of the chosen feature and the y-axis showing that feature's SHAP value
- A rising trend indicates that larger feature values tend to increase the prediction, while a falling trend indicates that larger values tend to decrease the prediction; flat regions suggest little effect or saturation.



- The colour of the dots encodes a feature value (often an automatically selected potential interacting feature, or sometimes the same feature).
- The vertical spread of dots at a given x-value shows heterogeneity in effect across observations, which typically reflects interactions or local non-linearities learned by the model.

Waterfall Plots

- These explain one prediction by starting at the baseline (the model's average output on a reference set) and adding each feature's SHAP contribution until it reaches the final prediction for that case.
- Bars pointing right are positive contributions that increase the prediction, and bars pointing left are negative contributions that decrease it; the bar length shows the size of the effect on the model's output scale.



- Features are shown in order of impact so you can see the top drivers pushing the prediction up and the top drivers pulling it down.
- The plot often includes an "other features" bar that aggregates the many small remaining effects so the big drivers are easy to see.

Time Series Data

- A time series is data recorded at regular or irregular intervals sequentially over time.
- Observations are not independent: nearby points often influence each other (autocorrelation), so methods must account for temporal dependence.
- Time series typically exhibit structure such as long-term trend, repeating seasonality (daily/weekly/annual), and shorter-lived cycles or shocks.
- The sampling frequency (e.g., hourly, daily, monthly) and any gaps or irregular spacing are part of the data definition and affect how we model and evaluate it.
- Common examples include temperature readings, stock prices, energy demand, website traffic, sensor streams, and sales over time, where forecasting future values is the primary goal.

Challenges with Time Series Data

- Temporal dependence means adjacent observations are correlated, so random shuffles or IID assumptions break; we must respect ordering for features, resampling, and error estimation.
- Seasonality and long-term trends add structured patterns that can dominate signals, so models need explicit seasonal terms, differencing, or decomposition to avoid biased forecasts.
- Non-stationarity is common as data-generating processes drift over time, which changes levels and relationships and requires rolling refits, time-varying parameters, or drift monitoring.

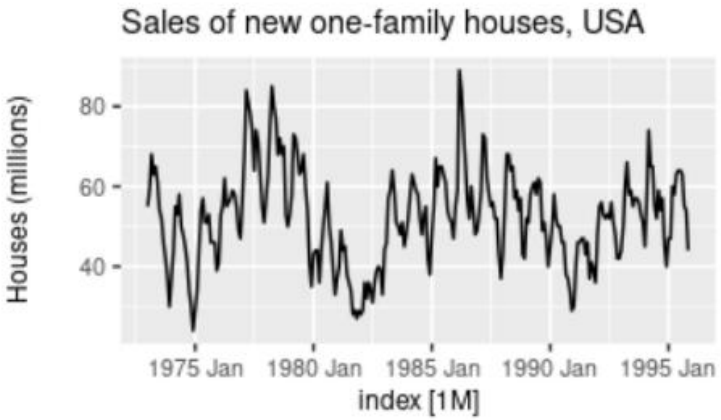
Time Series Patterns

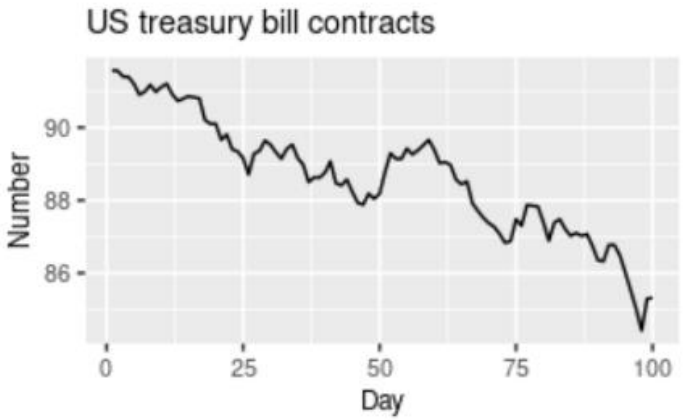
- A trend exists when there is a long-term increase or decrease in the data. It does not have to be linear. Sometimes we refer to a trend as 'changing direction' when it might go from an increasing trend to a decreasing trend.

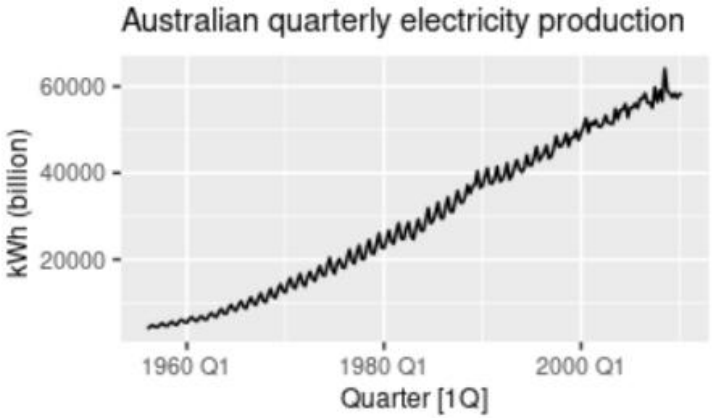
- A seasonal pattern occurs when a time series is affected by seasonal factors such as the time of the year or the day of the week. Seasonality is always of a fixed and known period.

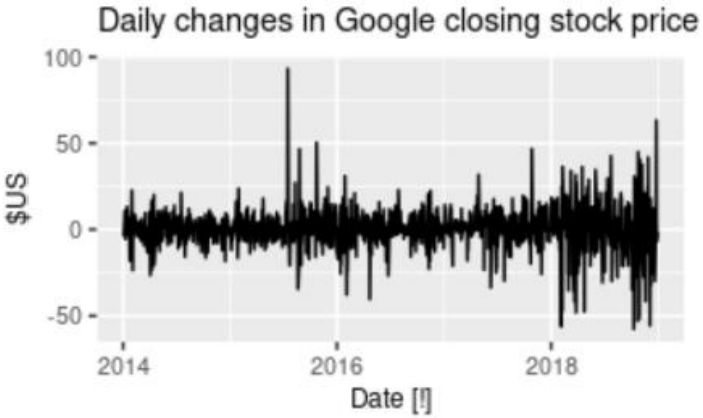
- A cycle occurs when the data exhibits rises and falls that are not of a fixed frequency. These fluctuations are usually due to economic conditions. The duration of these fluctuations is usually at least two years.

Time Series Patterns









Time Series Patterns

- The monthly housing sales show strong seasonality within each year as well as some strong cyclic behaviour with a period of about 6-10 years. There is no apparent trend over this period.
- The US treasury bill contracts show results from the Chicago market for 100 trading days in 1981. Here there is no seasonality, but an obvious downward trend. Possibly if we had a longer series we would see this trend is actually part of a long cycle.
- The Australian quarterly electricity production shows a strong increasing trend, with strong seasonality but no evidence of cyclic behaviour.
- The daily change in Google stock price has no trend, seasonality or cyclic behaviour. There are random fluctuations which don't appear to be very predictable, and no strong patterns that would help with modelling.

Converting Data to tsibble

- A tsibble is a tibble with time semantics, where you declare a time index (e.g., date, month) and an optional key (the series identifier), so each key–time pair is unique and ordered.
- tsibble adds temporal integrity that ordinary tibbles lack, by enforcing order, detecting duplicates, revealing gaps, and recording the interval (regular/irregular) of the series.
- tsibble provides extra structure and safety that helps prevent look-ahead leakage (when features or splits accidentally use future information).

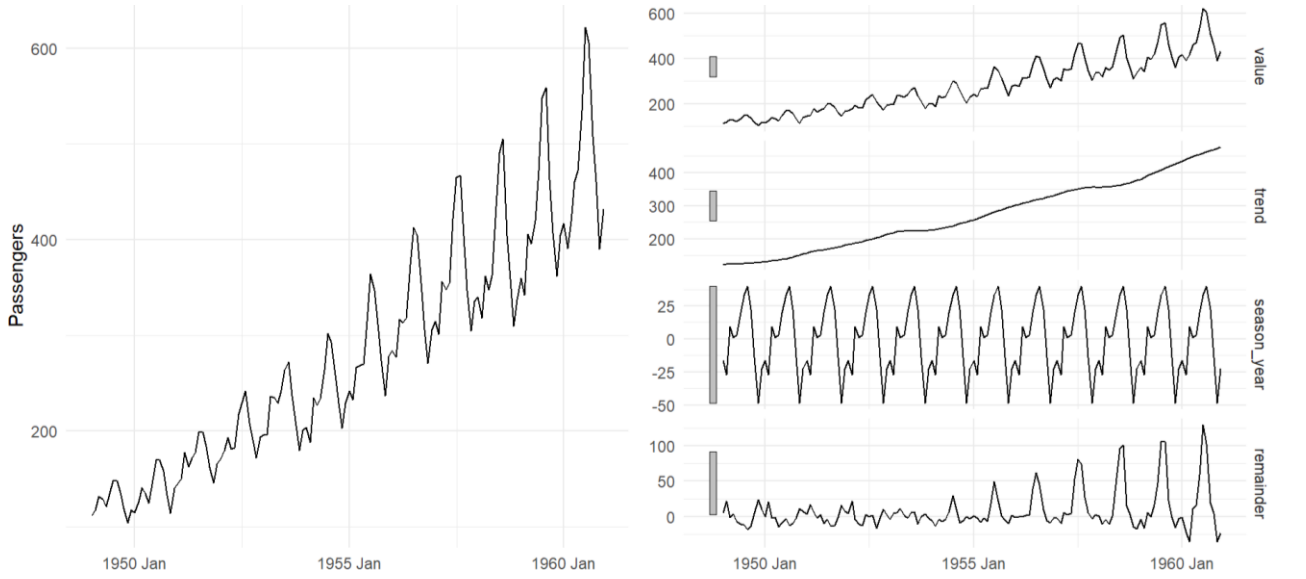
Decomposing Time Series

- Decomposition splits a series into interpretable pieces-a long-term trend, a repeating seasonal pattern, and a leftover remainder—so we can see what drives the data at different time scales.
- STL (Seasonal-Trend decomposition using Loess) lets the seasonality change slowly over time and works for any period (e.g., 7, 12, 24).
- Decomposition helps diagnose non-stationarity, deseasonalise before modelling, and flag anomalies when the remainder is unusually large.

```
library(feasts)
data_ts %>% model(STL(value)) %>% components() %>% autoplot()
```

Decomposing Time Series

- The series is modelled as Data = Trend + Seasonal + Remainder
- The trend component shows gradual up- or down-movement, the seasonal component shows the within-cycle pattern (e.g., within a year or week), and the remainder captures irregular noise, outliers, and events.



Feature Engineering for Time Series

- Feature engineering creates past-aware predictors (e.g., lags, rolling stats, seasonality, differences) so models can learn autocorrelation, trend, and seasonal effects that raw timestamps do not expose.
- These features tell us how strong seasonality is, how quickly the series reverts or persists, and how volatile or stable recent behaviour has been.
- They also let us inject domain signals via calendar features (month, day-of-week, holidays) and exogenous drivers (weather, price, promotions) that influence the response.

Lagged Values

- A lag is the same variable shifted back in time, so a lag-1 feature uses yesterday's value to help predict today.
- We use lags to let the model learn autocorrelation and short-term memory.
- Typical examples use $y_{t-1}, y_{t-7,} y_{t-12}$ for daily/weekly/annual seasonality, and we can lag exogenous predictors as well.
- As predictors, large positive SHAP/coefficients for lags indicate that recent high values push predictions up.
- Lags must be computed with past-only data to avoid leakage.

```
recipe(y ~ time, data = ts_train) |>
step_lag(y, lag = c(1, 7, 12)) |>
step_naomit(all_predictors())
```

Rolling Windows

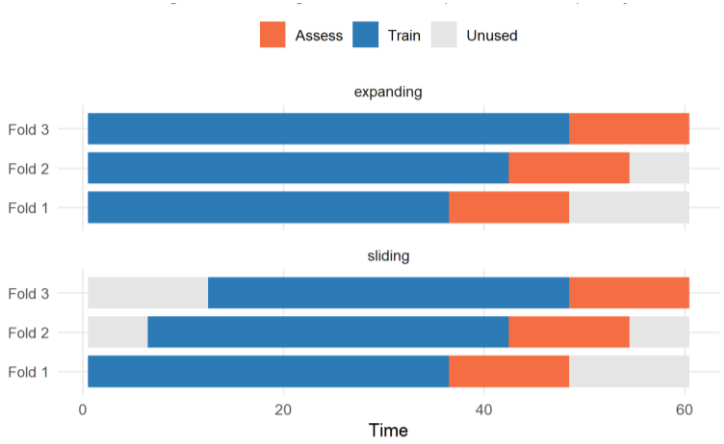
- A rolling feature applies a function over a trailing window (e.g., last 7 observations), producing a numeric summary of recent history.
- We use rolling means/medians to capture the current level, rolling SD for volatility and rolling min/max for recent extremes.
- As predictors, rolling means act like a local baseline, while rolling SD tells the model about recent instability that may widen forecast errors.
- For example, if the response y is retail daily sales, then a 7-day moving average tells us the current baseline demand after smoothing promotions/weekends, a 30-day rolling SD tells us about demand instability (promotion spikes, stock-outs, unusual events) and a longer rolling sum would tell us about seasonal build-up of demand (e.g., pre-holiday total sales)

Seasonal Indicators

- Seasonal indicators turn the time index into categorical labels such as month, quarter, or hour so models can learn recurring patterns.
- We use them when behaviour repeats on the calendar (weekly traffic peaks, December sales spikes) and wand a different baseline by season.
- Examples include month = January, …, December, DOW = Monday, … Sunday.
- As predictors, positive values for a season mean that this period tends to lift the series while negative ones mean it depresses the series.

Time Series Cross Validation

- Time-series cross-validation evaluates a model by forecasting from the past, splitting the series into chronological train/test windows that move forward through time
- In rolling-origin resampling, we fit on data up to a cutoff and assess on the next steps; we then slide the origin forward and repeat with either an expanding or sliding training window.
  - Expanding window: Best when older data are still informative (stable process, strong long-run seasonality).
  - Sliding window: Best when the process drifts or older patterns become irrelevant



Resampling with rsample

- rolling_origin() creates forward-chaining folds: train on a past window and evaluate on the next assess points, then move the origin forward and repeat.
  - initial: sets the training window length for the first fold.
  - assess: sets the forecast horizon evaluated each fold.
  - cumulative: controls window type: TRUE = expanding training window; FALSE = sliding fixed-width window.
  - skip: advances the origin by a fixed step size between folds to reduce overlap or match your forecast cadence.
- All feature engineering happens inside each fold: compute lags, rolling stats, differences, and scaling on the analysis (train) portion only, then apply to the assessment portion.

Example: Predicting Airline Passengers

- AirPassengers is a monthly airline passenger series (1949–1960) with a clear upward trend and strong yearly seasonality. We will predict future passenger counts.
- The target is next month's passengers.
- We create lagged values (e.g., lag 1, 12), rolling means/SDs, seasonal indicators (month as factor).
- We cannot let future information leak (e.g., using full-series scaling or rolling stats computed past the split). Refit preprocessing and the model only on training data at each step.

Best Practices

- Always set aside a test set
- Use cross-validation wisely
- Make your pipeline reproducible and modular
- Focus on both accuracy and interpretability

- Real-World considerations:
  - Fairness, bias, and ethics
  - Model deployment and updating
  - Stakeholder communication
  - Model monitoring in production

Common Mistakes to Avoid

- Data leakage during preprocessing.
  - Leakage happens when information from validation/test data influences training (e.g., scaling/imputation fit on the full dataset, using future rows in time series, target-encoded features computed on all data). Prevent it by splitting first, then fit all preprocessing inside your CV folds/workflow.
- Overfitting due to lack of validation.
  - Models can memorise quirks of the training set and look great in-sample but fail on new data. Prevent it by using cross-validation, keep a final untouched test set, add regularisation, and prefer the simplest model that performs competitively.
- Ignoring class imbalance.
  - With rare positives, “accuracy” can be misleading and the model may neglect the minority class. Prevent it by using stratified splits; evaluate with PR-AUC, F1, recall, balanced accuracy; tune the decision threshold; try class weights or re-sampling (SMOTE/under-sample) within CV.

Final Thoughts

- A unified pipeline boosts reproducibility.
  - Keep one end-to-end workflow. Version your data and code, set seeds, fix random states, log hyperparameters, and save fitted objects so results can be rebuilt exactly.
- Model selection and interpretability go hand-in-hand.
  - Compare candidates on the same folds and metrics, then favour the simplest model that performs competitively. Explain the winner with permutation importance/SHAP, and sanity-check stories on a hold-out set to avoid “pretty but wrong” explanations.
- Focus on generalisation, not just accuracy.
  - Use proper resampling (k-fold, repeated CV, or rolling origin for time series).
- Data first.
  - Diagnose leakage, missingness, and data drift early. Document assumptions and transformations, and address fairness/ethics by checking performance across groups and reporting uncertainty.

Thank You

- Thanks for being part of this machine learning course.

- Keep learning: revisit the R scripts and implement the exercises yourself.
- Keep building: apply a model to a dataset from your work or studies.
- Keep asking: feel free to email me with any questions you might have in the future, related to the course or to applying these methods to your own data.