Class Imbalance – Model Selection – The Full Picture

Machine Learning I

Hugo Franco, Ph.D.

Universidad de La Sabana

M.Sc. program in Applied Analytics

October 4, 2025

Agenda

Core Concepts for Today's Session

- Synthetic Data Generation, Search Algorithms, Gradient Descent, Backpropagation, Model Generalization.

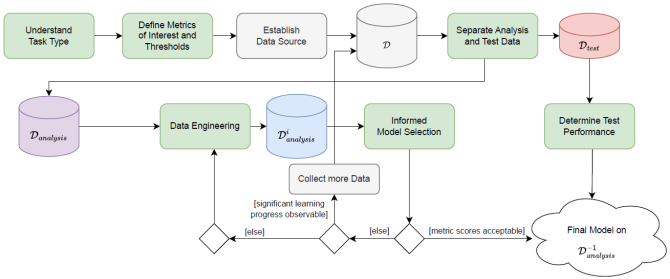
1. Handling Class Imbalance with SMOTE
2. Systematic Model Selection: Hyperparameter Tuning
3. Introduction to Artificial Neural Networks (ANNs)
4. Integrated Workshop: Building Complete ML Pipelines with ANNs

Recap: The Complete Supervised Learning Workflow

The Data Science-Oriented Supervised ML Pipeline

The Golden Rules

1. **Split Early, Split Once:** Separate your test set at the very beginning and do not touch it until the final evaluation.
2. **Prevent Data Leakage:** All preprocessing steps (scaling, imputation, encoding, SMOTE) must be learned *only* on the training data.
3. **Validate Rigorously:** Use cross-validation on the training set to make reliable decisions about models and hyperparameters.



The full workflow separates analysis data ($\mathcal{D}_{\text{train}} + \mathcal{D}_{\text{validation}}$) from the final test data ($\mathcal{D}_{\text{test}}$).

Handling Class Imbalance

The Problem with Imbalanced Data

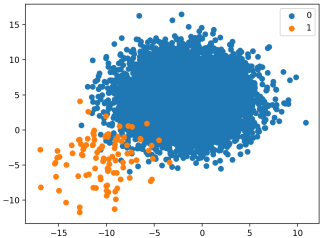
Definition: Occurs when one class (the majority class) vastly outnumbers another (the minority class).

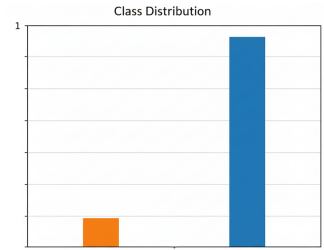
Real-world examples:

- Fraud detection (99.9% non-fraud vs. 0.1% fraud)
- Medical diagnosis of rare diseases
- Manufacturing defect detection

The Accuracy Paradox

A naive model that *always* predicts "Not Fraud" would have 99.9% accuracy but would be completely useless. Standard algorithms become biased towards the majority class.





Why Imbalance is Detrimental?

- **Biased Learning:** Models quickly learn that predicting the majority class minimizes the loss function (like cross-entropy), so they ignore the minority class.
- **Poor Generalization for Minority Class:** The model fails to learn the decision boundary for the minority class, leading to high False Negatives.
- **Misleading Evaluation Metrics:** Accuracy becomes a poor indicator of performance. We must rely on other metrics:
  - **Precision & Recall:** How reliable are positive predictions and how many actual positives did we find?
  - **F1-Score:** The harmonic mean of Precision and Recall.
  - **AUC-ROC:** Evaluates performance across all classification thresholds.

Strategies to Mitigate Class Imbalance

Two primary approaches exist:

1. Algorithmic Approaches

- **Adjust Class Weights:** Penalize the model more for misclassifying the minority class. (e.g., `class_weight='balanced'` in Scikit-learn)
- **Use Different Algorithms:** Tree-based models like Random Forest and XGBoost often handle imbalance better than linear models.

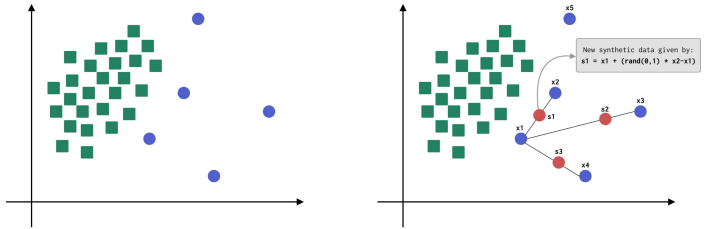
2. Resampling Techniques (Data-Level)

- **Undersampling:** Remove instances from the majority class. [citestart]Risk: Information loss.
- **Oversampling:** Add more copies of the minority class. [citestart]Risk: Overfitting.
- **Synthetic Data Generation:** Create new, artificial minority class instances.

Synthetic Minority Over-sampling TEchnique: SMOTE

Core Idea: Instead of duplicating existing data, SMOTE creates new, synthetic minority class samples that are plausible and add new information.

1. Pick a random minority class sample, x_1.
2. Find its k nearest minority class neighbors (e.g., 3).
3. Randomly choose one of those neighbors, x_2.
4. The new synthetic sample s_1 is created at a random point along the line segment connecting x_1 and x_2.
5. Repeat until a target class balance is reached



a) Class Imbalance

b) SMOTE

SMOTE and Data Leakage

CRITICAL WORKFLOW RULE

SMOTE (and any other oversampling technique) must **only** be applied to the **training data**. It should be a step *inside* your cross-validation loop.
Why?

- If you apply SMOTE to the entire dataset *before* splitting, you will have synthetic data in your validation and test sets.
- The model will be evaluated on data that is artificially similar to the training data.
- This is a form of **data leakage** and will give you a wildly optimistic and invalid performance estimate.

Correct Implementation:

Use a Scikit-learn `Pipeline` with an imblearn sampler. This ensures SMOTE is only fit on the training portion of each cross-validation fold.

SMOTE: Pros and Cons

Advantages

- Mitigates overfitting compared to simple random oversampling.
- No information loss from the majority class.
- Broadens the decision region for the minority class, improving generalization.

Disadvantages

- Can create noise by generating samples in overlapping regions between classes.
- Not effective for very high-dimensional data.
- Variants like Borderline-SMOTE or ADASYN exist to address these issues.

Systematic Model Selection

Parameters vs. Hyperparameters

Parameters

- These are values **learned by the model from the data** during the training process.
- You do not set them manually.
- **Examples:** The weights (w) in a linear regression, the support vectors in an SVM.

Hyperparameters

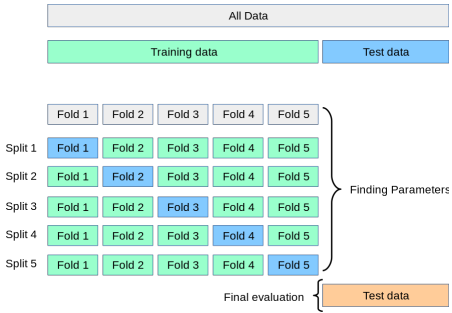
- These are settings you **configure before training** to control the learning process.
- They are not learned from the data. They define the model's architecture or the algorithm's behavior.
- **Examples:** The number of neighbors ('k') in k-NN, the regularization strength ('C') in an SVM, the learning rate in a neural network.

The goal of hyperparameter tuning is to find the set of hyperparameter values that results in the model with the best generalization performance.

Recap: Cross-Validation

A single train/validation split can be sensitive to how the data is divided. **k-Fold Cross-Validation** is a more robust technique:

1. Split the training data into 'k' folds (e.g., k=5).
2. In each iteration, use one fold as the validation set and the remaining k-1 folds for training.
3. Average the performance across all k folds to get a more stable estimate of model performance.



k-Fold CV uses data more efficiently for evaluation.

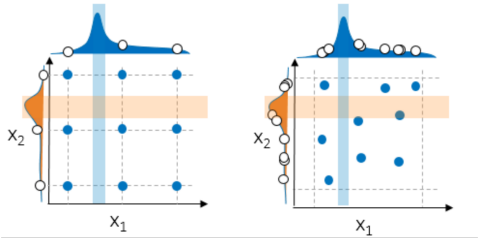
Recap: Parameter Exploration

- *Classification or regression models* can have multiple parameters whose values determine the model's ability to provide a satisfactory solution to the machine learning problem.
- *Model parameters* whose values do not change during the training process are known as "**hyperparameters**."
- They can be related to
  - The *model architecture* (network, decision tree, random forest)
  - The *learning process control parameters* (learning rates, rate of change coefficients).
- *Brute-force parameter exploration* methods (such as Grid Search, left) or *random search* (right) are often used to find the combination of hyperparameters that offers the best performance.

Hyperparameter Tuning Strategies

How do we search for the best combination of hyperparameters?

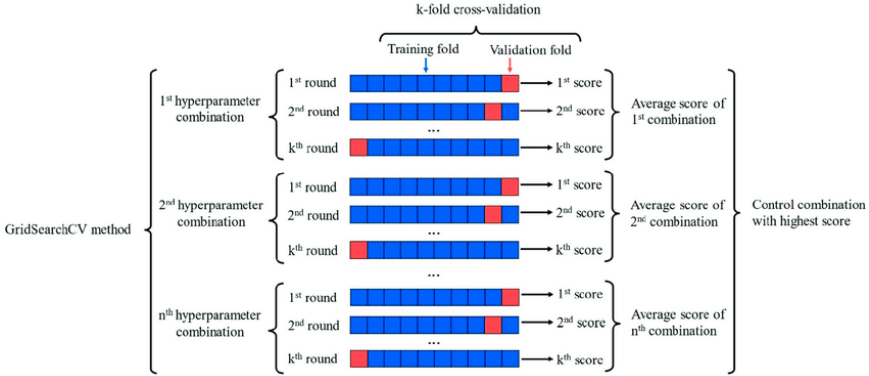
- **Manual Search:** Using intuition and trial-and-error. Inefficient and not reproducible.
- **Grid Search:** Exhaustively search a predefined grid of hyperparameter values.
- **Random Search:** Randomly sample a fixed number of combinations from the hyperparameter space. Often more efficient than Grid Search.
- **Bayesian Optimization:** A more advanced method that uses results from previous trials to inform the next set of hyperparameters to test.



Recap: Hyperparameter tuning

A nested approach, with an inner loop traversing the hyperparameter space and an outer loop optimizing the model performance for each built hyperparameter combination, allows the exhaustive selection of the best model

- An extensive set of potential model configurations (not affected by training) are evaluated



k-Fold CV uses data more efficiently for evaluation.

Grid Search with Cross-Validation

Universidad de La Sabana

The most common and robust approach for systematic tuning.

Basic approach:

1. **Define a Grid:** For each hyperparameter, specify a list of values to try.
   - Example: For SVM, `C: [0.1, 1, 10]`, `gamma: ['scale', 'auto']`
2. **Iterate:** The algorithm will train and evaluate a model for *every single combination* of the specified values. (3 C values $\times$ 2 gamma values $=$ 6 models).
3. **Use Cross-Validation:** For each combination, performance is measured using k-fold cross-validation on the training set. The average validation score is recorded.
4. **Select the Best:** The hyperparameter combination with the highest average cross-validation score is chosen as the best.

Grid Search: Pros and Cons

Advantages

- **Exhaustive:** Guaranteed to find the best combination within the specified grid.
- **Parallelizable:** Each combination can be tested independently, making it suitable for multi-core processing.
- **Systematic and Reproducible.**

Disadvantages

- **Computationally Expensive:** Suffers from the "curse of dimensionality." The number of models to train grows exponentially with the number of hyperparameters and values.
- **May Miss Optimal Points:** The best value might lie between two points in your grid.

Alternative: Random Search

An often more efficient alternative to Grid Search.

- Instead of a grid of discrete values, you define a *distribution* for each hyperparameter (e.g., a uniform distribution between 0.1 and 100 for 'C').
- You specify the number of iterations (`n_iter`).
- The algorithm randomly samples `n_iter` combinations from these distributions.
- As with Grid Search, each combination is evaluated using cross-validation.

Key Insight

Often, only a few hyperparameters are truly important. Random search is more likely to find a good value for the important ones because it doesn't waste time on many combinations of unimportant ones.

The Final Step After Tuning

The Workflow

1. Perform Grid Search (or Random Search) with cross-validation on your **training data** to find the best hyperparameters.
2. Take the best hyperparameter combination identified.
3. Retrain a **new model** using these best hyperparameters on the **entire training set**.
4. Finally, evaluate this single, final model on the **held-out test set** to get an unbiased estimate of its generalization performance.

Introduction to Neural Networks

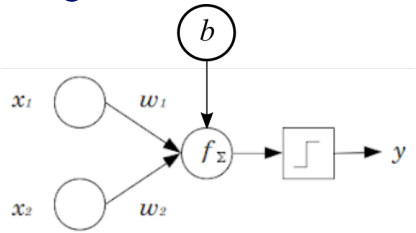
Example: Perceptron - Classification and Regression

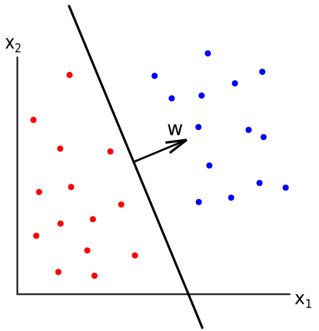
- Similar to other classification methods (such as the support vector machine), the perceptron can be interpreted as a linear model that allows for classification and regression.
- In the case of two features (example)

$$y = f\left(x_1 w_1 + x_2 w_2 + b\right) = f(\mathbf{x}^T \cdot \mathbf{w} + b)$$

- If $f : \mathbb{R}^m \to \mathbb{R}$ is a linear function (particularly the identity), we have a **regression problem.**
- If f is defined to take values around two levels, we have a **binary classification problem** (non-activated unit vs. activated unit). E.g.

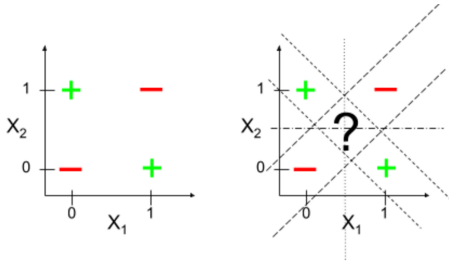
$$f = \begin{cases} 0 & \text{if } \left(\mathbf{x}^T \cdot \mathbf{w} + b\right) \leq 0 \\ 1 & \text{if } \left(\mathbf{x}^T \cdot \mathbf{w} + b\right) > 0 \end{cases}$$



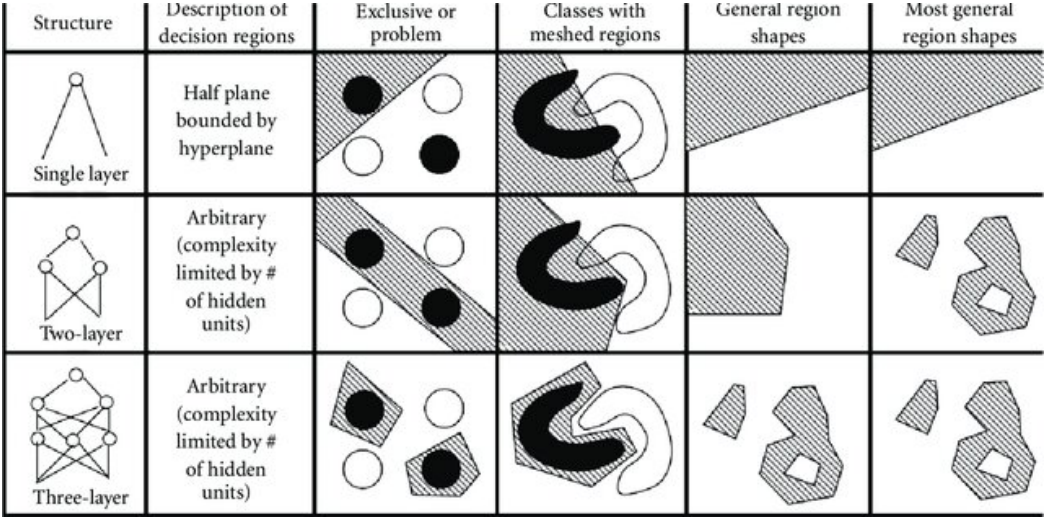


Multilayer Networks Motivation: Classification of complex regions

- If the classes in the dataset are not linearly separable (there is no single hyperplane that completely separates the regions), a perceptron is not sufficient to address the classification problem.
- Example (image on the right) if the data follows an XOR function for two features. (x_1 y x_2), it is not possible to find a single straight line that separates the classes (+) y (-).

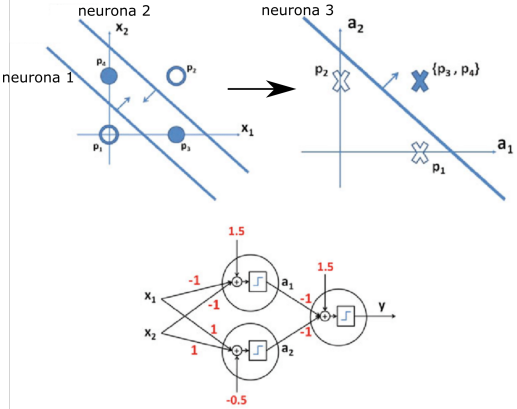


Network architecture complexity vs. regional complexity



Solution: Multilayer Neural Networks

- If more processing units (neurons) are added to the representation, there will be more overlapping hyperplanes that will allow for the delimitation of more complex regions associated with class distribution.
- This, in turn, implies adding more "layers" to the model architecture.
- The solution will then be obtained as the result of successive classification processes in spaces transformed by each unit in each layer, depending on its input.



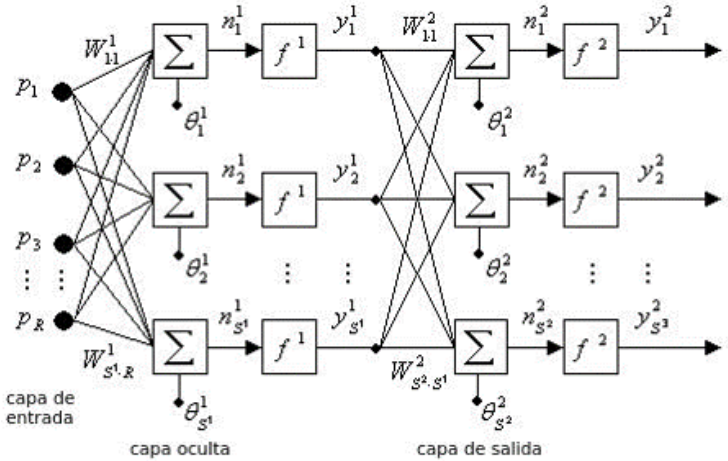
Multilayer perceptron and feedforward networks

- An architecture that groups multiple units (perceptrons or "neurons") into layers is known as a "multilayer perceptron."
  - Each neuron provides a partition of the space determined by its inputs and an activation value that can be fed as input to subsequent units.
  - Output units can have specific activation functions depending on the application; they usually encode a response.
  - Input units simply take each feature vector (in training or prediction) and feed its values to the first hidden layer.

A particularly useful and common type in the literature is the feedforward network, in which neurons in one layer only connect to neurons in the next layer.

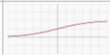
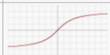
- When every neuron in an input or hidden layer is connected to all the neurons in the layer immediately below it, and only to them, it is known as a "fully connected" network.
- Training (learning) is usually implemented through generalizations of perceptron learning.

Architecture of a feedforward network



Activation functions

Activation functions determine the behavior of the output of the hidden units

Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1-f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU)		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU)		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$

Back-propagation algorithm

How Neural Networks Learn: Gradient Descent

The goal is to find the weights (w) and biases (b) that minimize the cost function (J). **Gradient Descent** is an iterative optimization algorithm that does this:

1. Start with random initial weights.
2. Calculate the gradient (the direction of steepest ascent) of the cost function with respect to each weight.
3. Update each weight by taking a small step in the *opposite* direction of the gradient. The size of this step is controlled by the **learning rate**, α.

$$w_{new} = w_{old} - \alpha \frac{\partial J}{\partial w_{old}}$$

4. Repeat until the cost function converges to a minimum.

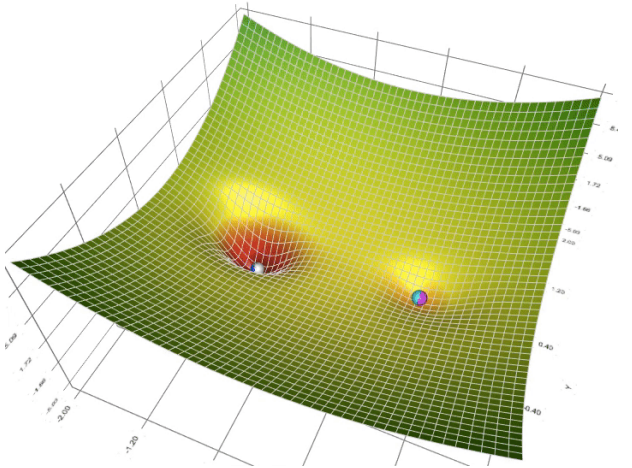
The process of calculating these gradients efficiently is called **Backpropagation**.

Back-propagation algorithm

- In perceptron learning (only one unit), the prediction error (target value minus the actual value obtained) allowed the weights connecting the inputs to the unit to be adjusted directly.
- However, the error associated with the neurons belonging to the hidden layers cannot be calculated directly, as a "correct" output is not known.
  - This is because the training dataset (known input patterns with corresponding target output) only relates the model's inputs (characteristics of each record) with the expected output for each input.

To overcome this limitation, "gradient descent" methods are used, which aim to minimize a loss function ("loss function" generally an estimated measure of error) using iterative algorithms.

Error optimization



- Representation of the Cost function optimization ($J(\theta)$, where θ is the set of the *learnable* model parameters)

Delta rule

- In the case of a multilayer network, the prediction error at each step of the training process for output neurons can only directly affect the neurons in the previous layer.
- The error estimate can be generalized to all hidden and output neurons in the network using the "delta rule":

Given the activation function of a neuron $f(\mathbf{x}^T \cdot \mathbf{w} + b)$, updating the connection weight between neurons i and j (being x_i the output of the i-th neuron, input of the j-th neuron) can be performed by:

$$\Delta w_{ij} = \alpha(y_i - \hat{y}_i) f'(\mathbf{x}^T \cdot \mathbf{w} + b) x_i$$

where the coefficient α corresponds to the learning rate (it regulates the speed of approach to the optimal value) and

$$\frac{\partial f(\mathbf{x}^T \cdot \mathbf{w} + b)}{\partial w_{ij}}$$

is the partial derivative respect to w_{ij} , the weight of the connection between units i and j.

Algorithm's general formulation

- The Back-propagation algorithm uses a more robust formulation based on the error function (E) and its derivative as a function of the change in weights. Suppose The total network error can be written as:

$$E = \frac{1}{n}\sum_{i=1}^{n} \|\mathbf{y}_i - \hat{\mathbf{y}}_i\|^2$$

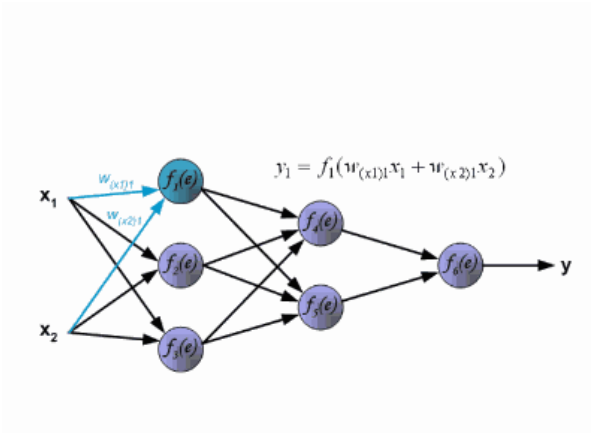
- That is, the rate of change in the error in the output of a neuron, which is sought to be decreased by variations in the weights. The changes in error due to changes in each weight of the network can be expressed as

$$\nabla E = \left\{ \frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \cdots, \frac{\partial E}{\partial w_m} \right\}$$

where m is the number of connections in the network. Thus, the learning rule is obtained

$$\Delta w_i = -\alpha \frac{\partial E}{\partial w_i} = \alpha(y_i - \hat{y}_i) f'(\mathbf{x}^T \cdot \mathbf{w} + b) x_i$$

- It is iterated until all errors associated with each weight fall below a tolerance threshold, which is equivalent to obtaining an acceptable value for the total error of the network.



* in the animation, the learning rate is notated as η instead of α, while δ denotes $(y_i - \hat{y}_i)$

Loss/Cost Functions

The **loss function** measures the error for a *single* training example. The **cost function** is the average loss over the entire training set. Training a network means minimizing this cost function.

Regression Loss Functions

- **Mean Squared Error (MSE):**

$$J = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

The standard choice for regression. Penalizes large errors heavily.

Classification Loss Functions

- **Binary Cross-Entropy:** For binary classification. Measures the difference between two probability distributions (the true labels and the predictions).
- **Categorical Cross-Entropy:** The generalization for multi-class classification.

Cost and Loss functions: Categorical Cross-Entropy

The Categorical Cross-Entropy Loss for **a single example** is:

$$L(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

Where:

- C is the total number of classes.
- $\mathbf{y}$ is the one-hot encoded true label vector (e.g., $[0, 1, 0]$ for class 2 out of 3).
- $\hat{\mathbf{y}}$ is the predicted probability vector (e.g., $[0.1, 0.8, 0.1]$).
- The sum $\sum_{i=1}^{C}$ adds the contribution of each class.

The Categorical Cross-Entropy Cost Function, calculated over the complete dataset after each batch, is:

$$J(\mathbf{w}, b) = -\frac{1}{N} \sum_{j=1}^{N} \sum_{i=1}^{C} y_i^{(j)} \log(\hat{y}_i^{(j)})$$

Where:

- N is the number of training examples.
- C is the number of classes.
- $y_i^{(j)}$ is the i-th element of the one-hot encoded label of the j-th example.
- $\hat{y}_i^{(j)}$ is the predicted probability for class i on the j-th example.
- The parameters $\mathbf{w}$ and b are implicitly inside the prediction $\hat{y}$.

Model Selection in Neural Networks

Hyperparameter: Optimizers

Optimizers are algorithms that adapt the learning rate and weight updates during training to make gradient descent faster and more stable.

- **SGD (Stochastic Gradient Descent):** The classic, updates weights based on one sample or a small batch at a time. Can be slow and noisy.
- **Adam (Adaptive Moment Estimation):** The most common and recommended default optimizer. It adapts the learning rate for each weight individually, combining the advantages of other optimizers like RMSprop and Momentum.
- **RMSprop:** Also maintains a per-parameter learning rate. Good for recurrent neural networks.

Key Hyperparameters of Optimizers

The **learning rate** is the most important hyperparameter to tune. Adam's default learning rate (e.g., 0.001) is often a good starting point.

Other Key NN Hyperparameters

Beyond loss and optimizers, tuning an MLP involves designing its architecture.

- **Loss-cost functions:** Could enhance the stability and convergence of the learning algorithm
- **Number of Hidden Layers:** Deeper networks (more layers) can learn more complex functions, but are harder to train and more prone to overfitting. Start with 1-2 hidden layers.
- **Number of Neurons per Layer:** Wider layers (more neurons) can learn more features at each level. Common practice is to have layers with a decreasing number of neurons (e.g., 64 -> 32).
- **Batch Size:** The number of training samples used in one iteration to update the weights. Common values are 32, 64, 128.
- **Epochs:** The number of times the entire training dataset is passed through the network. Too few epochs lead to underfitting; too many lead to overfitting.

Preventing Overfitting in Neural Networks

Neural networks are very flexible and can easily overfit. Key techniques include:

- **Regularization (L1/L2):** Adds a penalty to the loss function based on the magnitude of the weights, encouraging the model to learn smaller, simpler weights.
- **Dropout:** During training, randomly "drops out" (sets to zero) a fraction of neurons in a layer at each update. This forces the network to learn redundant representations and prevents it from relying too heavily on any single neuron.
- **Early Stopping:** Monitor the validation loss during training. If the validation loss stops decreasing (or starts increasing) for a certain number of epochs ("patience"), stop the training process to prevent overfitting. This is a highly effective and common practice.

Like SVMs and kNN, Neural Networks are sensitive to feature scaling. Always scale your data!

Universidad de La Sabana

Putting It All Together

Case Study 1: Credit Card Fraud Detection (Classification)

The Problem

Build a model to detect fraudulent credit card transactions. This is a classic example of a highly imbalanced classification problem.

The Dataset

- Features: Anonymized transaction features (V1, V2, ... V28), Time, Amount.
- Target: 'Class' (0 for Normal, 1 for Fraud).
- Imbalance: The 'Fraud' class is a very small minority.

Workflow for Fraud Detection

1. **Problem Definition:** Binary classification. Metric of interest is **Recall** (we want to catch as many frauds as possible) and **AUC-ROC**.
2. **Data Prep & Splitting:**
   - Scale the 'Amount' and 'Time' features using `StandardScaler`.
   - Split data into train (80%) and test (20%) sets.
3. **Modeling Pipeline:**
   - Create a pipeline using `imblearn.pipeline`.
   - **Step 1:** SMOTE to handle class imbalance.
   - **Step 2:** An MLP Classifier (`MLPClassifier`).
4. **Hyperparameter Tuning:**
   - Use `GridSearchCV` to tune MLP hyperparameters like `hidden_layer_sizes`, `activation`, and `alpha` (for regularization).
5. **Final Evaluation:** Train the best model on the full training set and evaluate on the held-out test set using a confusion matrix and classification report.

Case Study 1: Discussion Questions

- Why is accuracy a terrible metric for this problem?
- What is the business cost of a False Negative (missing a fraudulent transaction) versus a False Positive (flagging a normal transaction)?
- How does the confusion matrix change before and after applying SMOTE?
- How would you explain the final model's performance to a non-technical stakeholder?

Case Study 2: Predicting House Prices (Regression)

Universidad de La Sabana

The Problem

Build a model to predict the sale price of a house based on its features. This is a classic regression problem.

The Dataset (e.g., Boston Housing or California Housing)

- Features: Mix of numerical and categorical features (e.g., number of rooms, crime rate, proximity to ocean).
- Target: 'MedianHouseValue' (a continuous variable).

Workflow for House Price Prediction

1. **Problem Definition:** Regression. Metric of interest is **Mean Squared Error (MSE)** or **R-squared (R^2)**.
2. **Data Prep & Splitting:**
   - Split data into train (80%) and test (20%) sets.
   - Create separate preprocessing pipelines for numerical and categorical features using `ColumnTransformer`.
   - Numerical: Impute missing values (e.g., with median) then apply `StandardScaler`.
   - Categorical: Impute missing values then apply `OneHotEncoder`.
3. **Modeling Pipeline:** Combine the preprocessor and an `MLPRegressor`.
4. **Hyperparameter Tuning:** Use `RandomizedSearchCV` (often faster for larger search spaces) to tune hyperparameters like `hidden_layer_sizes`, `learning_rate_init`, and `solver`.
5. **Final Evaluation:** Train the best model pipeline on the full training data and evaluate its MSE and R-squared on the test set.

Case Study 2: Discussion Questions

- Why is it crucial to use a `ColumnTransformer` here? What would happen if you scaled the one-hot encoded features?
- How would an Early Stopping callback improve your tuning process for this problem?
- Your final model has an R-squared of 0.85. What does this mean in simple terms?
- If your model performs poorly, what would be your next steps? (e.g., more feature engineering, trying a different model like XGBoost).

Universidad de La Sabana

What We've Learned and Where to Go Next

Course Summary

Key Takeaways

- **Workflow is the core:** A robust, disciplined workflow with proper validation is more important than any single algorithm.
- **Data is the Foundation:** Preprocessing, cleaning, and feature engineering are where most of the work (and value) lies.
- **There is no "Silver Bullet":** Different models have different strengths. The best model depends on the data and the problem.
- **Evaluation Matters:** Choose the right metrics for your problem. Accuracy can be deceiving, especially with imbalanced data.
- **Tuning is Essential:** Default model parameters are rarely optimal. Systematic hyperparameter tuning is crucial for performance.

Final Goal:

Be able to build, tune, and critically evaluate end-to-end machine learning solutions for complex problems.

Universidad de La Sabana

Questions?

Thank you!