PHASE 1
Core Machine Learning

Classical models, evaluation, feature engineering, and tuning — implemented, not just imported

Phase goal

Understand classical ML deeply enough to implement the key algorithms from scratch and to build a leakage-free, well-evaluated pipeline — not just to call `.fit()`.

Estimated time: 6–8 weeks | **Track:** Comprehensive ML & Deep Learning Curriculum

A self-paced, depth-first path from mathematical foundations to production deep learning.

Contents

1 Orientation

Classical ("tabular") machine learning is still where most production value is created outside of the LLM frontier: churn prediction, fraud, pricing, demand forecasting, ranking. It is also the best place to build the disciplines — evaluation, validation, leakage avoidance — that separate practitioners from people who overfit Kaggle leaderboards. Deep learning inherits all of these disciplines; learning them on simple models first is far cheaper.

Key idea

The model is rarely the hard part. **Framing the problem, building trustworthy evaluation, engineering features, and avoiding leakage** are what determine success. A logistic regression evaluated correctly beats a deep net evaluated incorrectly every single time.

2 ML fundamentals

2.1 The learning setups

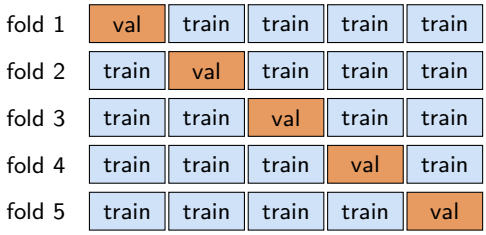
- **Supervised:** learn a mapping $f : \mathbf{x} \mapsto y$ from labeled pairs. Regression ($y \in \mathbb{R}$) or classification (y categorical).
- **Unsupervised:** find structure without labels — clustering, dimensionality reduction, density estimation.
- **Semi-supervised:** a little labeled data plus a lot of unlabeled data (common and economically important).
- **Reinforcement:** an agent learns from reward by interacting with an environment (Phase 6).

2.2 Train / validation / test, and cross-validation

You split data to estimate *generalization* — performance on data the model has never seen. The cardinal rule: the **test set is touched exactly once**, at the very end.

Definition

k-fold cross-validation partitions the training data into k folds, trains on $k-1$ and validates on the held-out fold, rotating k times, then averages. It uses data efficiently and gives a variance estimate of your metric. **Stratified** k-fold preserves class proportions in each fold — essential for imbalanced classification.



Common pitfall

For **time-series** data, random k-fold leaks the future into the past. Use forward-chaining (expanding-window) splits where validation always lies after training in time. For **grouped** data (multiple rows per customer), use `GroupKFold` so the same group never appears in both train and validation — otherwise you measure memorization, not generalization.

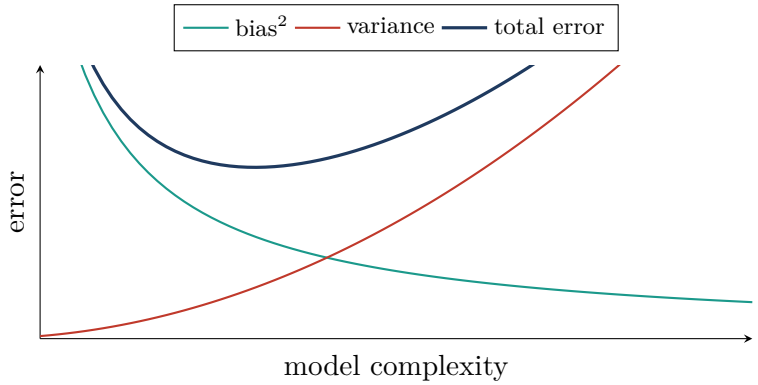
2.3 Bias–variance tradeoff

Expected test error decomposes (for squared loss) into three parts:

$$\mathbb{E}\big[(y-\hat{f}(\mathbf{x}))^2\big] = \underbrace{\big(\text{Bias}[\hat{f}(\mathbf{x})]\big)^2}_{\text{too simple}} + \underbrace{\text{Var}[\hat{f}(\mathbf{x})]}_{\text{too sensitive}} + \underbrace{\sigma^2}_{\text{irreducible}} .$$

Intuition

High bias = underfitting: the model is too simple to capture the signal (a line through curved data). **High variance = overfitting:** the model chases noise and changes wildly with the training sample. Regularization, more data, and simpler models reduce variance; richer models and better features reduce bias. The art is balancing them — and the validation curve (later) is how you *see* the balance.



2.4 Evaluation metrics — choose before you model

2.4.1 Regression

MSE $= \frac{1}{n}\sum(y_i-\hat{y}_i)^2$ (penalizes large errors quadratically); RMSE $= \sqrt{\text{MSE}}$ (same units as y); MAE $= \frac{1}{n}\sum|y_i-\hat{y}_i|$ (robust to outliers); $R^2 = 1 - \frac{\sum(y_i-\hat{y}_i)^2}{\sum(y_i-\bar{y})^2}$ (fraction of variance explained).

2.4.2 Classification

From the confusion matrix (TP, FP, TN, FN):

$$\text{precision} = \frac{TP}{TP+FP}, \quad \text{recall} = \frac{TP}{TP+FN}, \quad F_1 = \frac{2\cdot\text{prec}\cdot\text{rec}}{\text{prec}+\text{rec}}.$$

ROC-AUC measures ranking quality across all thresholds (probability a random positive outranks a random negative). **PR-AUC** is more informative under heavy class imbalance.

Common pitfall

Accuracy is a trap on imbalanced data. If 99% of transactions are legitimate, "predict legit always" scores 99% accuracy and catches zero fraud. Pick the metric from the *business cost*: recall when misses are expensive (disease, fraud), precision when false alarms are expensive (spam filters), F_1/PR-AUC for balance, and calibrated probabilities when you need to threshold by cost.

2.4.3 Ranking

For search/recommendation, **NDCG** (normalized discounted cumulative gain) rewards putting relevant items high; **MRR** (mean reciprocal rank) focuses on the position of the first relevant result. These matter the moment your API powers a search or recommendation feature.

3 Regression & classification

3.1 Linear regression — two solvers

The model is $\hat{y} = \mathbf{w}^\top \mathbf{x} + b$. With the bias folded into $\mathbf{w}$ by appending a 1 to $\mathbf{x}$:

- **Closed form (normal equation):** $\mathbf{w} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$ — exact, but $O(d^3)$ in the number of features and unstable if $\mathbf{X}^\top \mathbf{X}$ is ill-conditioned.
- **Gradient descent:** scales to huge n and d, and generalizes to models with no closed form. (You implemented this in Phase 0.)

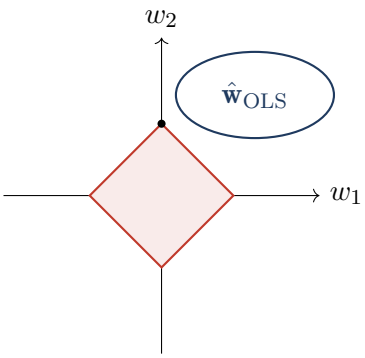
3.2 Regularization: Ridge, Lasso, ElasticNet

Add a penalty on weight magnitude to combat overfitting and multicollinearity:

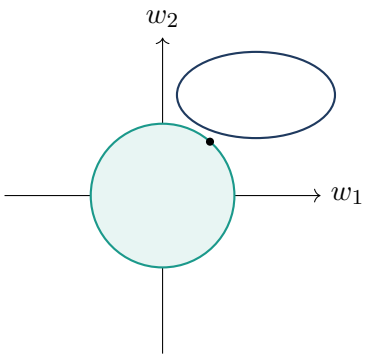
$$\text{Ridge: } \min_{\mathbf{w}} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 + \lambda \|\mathbf{w}\|_2^2, \qquad \text{Lasso: } \min_{\mathbf{w}} \|\mathbf{X}\mathbf{w} - \mathbf{y}\|_2^2 + \lambda \|\mathbf{w}\|_1 .$$

Key idea

Ridge (L2) shrinks all weights smoothly toward zero (and fixes the singular $\mathbf{X}^\top \mathbf{X}$ by adding $\lambda \mathbf{I}$). **Lasso (L1)** drives some weights *exactly* to zero, performing automatic feature selection — its diamond-shaped constraint region has corners on the axes. **ElasticNet** blends both. Recall from Phase 0: Ridge = Gaussian prior on weights (MAP), Lasso = Laplace prior.



Lasso (L1): corner ⇒ sparsity



Ridge (L2): smooth shrinkage

3.3 Logistic and softmax regression

For binary classification, logistic regression models $\mathbb{P}(y = 1 \mid \mathbf{x}) = \sigma(\mathbf{w}^\top \mathbf{x})$ with the sigmoid $\sigma(z) = 1/(1 + e^{-z})$. It is trained by minimizing the **cross-entropy** (negative log-likelihood of the Bernoulli):

$$L(\mathbf{w}) = -\sum_i \left[y_i \log \hat{p}_i + (1 - y_i) \log(1 - \hat{p}_i) \right], \quad \hat{p}_i = \sigma(\mathbf{w}^\top \mathbf{x}_i).$$

For K classes, **softmax regression** outputs $\hat{p}_k = \frac{e^{z_k}}{\sum_j e^{z_j}}$ and minimizes categorical cross-entropy. This exact pairing (softmax + cross-entropy) is the output layer of nearly every classifier you will build in Phases 2–4.

Intuition

"Logistic regression" is a misnomer — it is a *classifier*. It is a single-layer neural network with a sigmoid activation. Master its loss and gradient now and the deep-learning version in Phase 2 is just the same idea stacked.

3.4 Naive Bayes

Apply Bayes' theorem with the ("naive") assumption that features are conditionally independent given the class: $\mathbb{P}(y \mid \mathbf{x}) \propto \mathbb{P}(y) \prod_j \mathbb{P}(x_j \mid y)$. Despite the unrealistic assumption it is fast, needs little data, and is a strong baseline for text classification (bag-of-words).

3.5 K-Nearest Neighbors

KNN predicts using the k closest training points (majority vote or average). It has *no training phase* (lazy learning) but expensive inference, and degrades in high dimensions (the curse of dimensionality). It is a useful conceptual anchor: it makes the role of distance metrics and feature scaling vivid.

4 Tree-based methods

4.1 Decision trees

A tree recursively splits the feature space to make regions as "pure" as possible. Split quality for classification uses **Gini impurity** $G = 1 - \sum_k p_k^2$ or **entropy** $H = -\sum_k p_k \log p_k$; the chosen split maximizes **information gain** (impurity reduction). Trees are interpretable and need no feature scaling, but a single deep tree overfits badly (high variance).

Gini of a split

A node has 40 positives and 10 negatives: $G = 1 - (0.8^2 + 0.2^2) = 0.32$. A candidate split yields children $\{30^+, 2^-\}$ and $\{10^+, 8^-\}$ with Gini $1 - (\frac{30}{32})^2 - (\frac{2}{32})^2 = 0.117$ and $1 - (\frac{10}{18})^2 - (\frac{8}{18})^2 = 0.494$. Weighted child impurity $= \frac{32}{50}(0.117) + \frac{18}{50}(0.494) = 0.253 < 0.32$, so the split *reduces* impurity and is worth making.

4.2 Random forests (bagging)

Bagging (bootstrap aggregating) trains many trees on bootstrap resamples and averages them, slashing variance. Random forests add a twist: at each split only a random subset of features is considered, decorrelating the trees so averaging helps more. Robust, low-tuning, excellent default for tabular data.

4.3 Gradient boosting

Boosting builds trees *sequentially*, each new tree fitting the *residual errors* of the ensemble so far — gradient descent in function space.

Key idea

Gradient-boosted trees (XGBoost, LightGBM, CatBoost) are the reigning champions of tabular ML and win the majority of Kaggle tabular competitions. Defaults to know: tune learning rate × number of trees together, use early stopping on a validation set, and watch `max_depth` / `num_leaves` to control overfitting. LightGBM is

fastest on large data; CatBoost handles categoricals natively.

	Random Forest	Gradient Boosting
Trees built	in parallel, independently	sequentially, on residuals
Reduces mainly	variance	bias (then variance)
Tuning sensitivity	low	higher (LR, depth, #trees)
Risk	rarely overfits	overfits if over-trained

4.4 Interpretability: feature importance & SHAP

Built-in importances (split gain) are quick but biased toward high-cardinality features. **SHAP** values, grounded in cooperative game theory, fairly attribute each prediction to its features and are consistent and locally accurate. SHAP summary and dependence plots are the modern standard for explaining tabular models to stakeholders — and for catching leakage (a single feature with implausibly dominant SHAP is a red flag).

5 Support Vector Machines

An SVM finds the hyperplane that **maximizes the margin** — the distance to the nearest points (support vectors). The soft-margin formulation allows some violations, controlled by C (large C = fewer violations, higher variance). The **kernel trick** replaces dot products $\mathbf{x}_i^\top \mathbf{x}_j$ with a kernel $k(\mathbf{x}_i, \mathbf{x}_j)$, implicitly mapping to a high-dimensional space without computing it — enabling nonlinear boundaries with linear, polynomial, or RBF kernels.

Intuition

SVMs were dominant before deep learning and remain strong on small/medium datasets with clear margins. The RBF kernel "places a bump" around each support vector. The big-picture lesson — maximize margin for robustness — echoes throughout ML (it is the intuition behind large-margin and contrastive losses later).

6 Unsupervised learning

6.1 Clustering

K-Means alternates assigning points to the nearest centroid and recomputing centroids; it minimizes within-cluster variance but assumes spherical, equal-size clusters and needs k chosen (elbow/silhouette methods). **Hierarchical** clustering builds a dendrogram (no preset k). **DBSCAN** finds density-connected clusters of arbitrary shape and labels outliers, without specifying k — but is sensitive to its density parameters.

6.2 Dimensionality reduction

PCA projects onto the top eigenvectors of the covariance matrix (the SVD of centered data) — a *linear* method that maximizes retained variance, great for compression and de-noising. **t-SNE** and **UMAP** are *nonlinear* methods for *visualization* of clusters in 2D/3D.

Common pitfall

t-SNE/UMAP are for *visualization*, not as preprocessing for a downstream model. Distances and cluster sizes in a t-SNE plot are not faithful, and results depend heavily on perplexity/neighbors. Never read "these clusters are far apart so they are very different" off a t-SNE plot. Use PCA when you need a stable, invertible, distance-preserving reduction.

6.3 Anomaly detection

Isolation Forest isolates outliers with random splits (anomalies need fewer splits); **One-Class SVM** learns a boundary around the normal data. Both are unsupervised and directly relevant to fraud/defect/shipment-anomaly use cases.

7 Feature engineering & data prep

In tabular ML this is where most of the winning happens.

- **Missing data:** understand *why* it is missing; impute (mean/median/model-based) and optionally add a "was-missing" indicator.
- **Outliers:** detect (IQR, z-score), then cap/winsorize or model robustly — do not blindly delete.
- **Categorical encoding:** one-hot for low cardinality; **target/mean encoding** for high cardinality (with cross-fold smoothing to avoid leakage); ordinal only when order is real.
- **Scaling:** standardize (z-score) or normalize ($[0,1]$). Required for KNN, SVM, PCA, and neural nets; irrelevant for trees.
- **Class imbalance:** class weighting, undersampling the majority, or **SMOTE** (synthetic minority oversampling). Apply resampling *inside* cross-validation only.

Data leakage — the #1 cause of "too good to be true" results

Leakage is any information from the test set (or the future) sneaking into training. Classic forms: scaling/imputing/encoding on the *full* dataset before splitting; target-encoding without cross-fold isolation; including a feature that is a proxy for the label (e.g. "account closed date" when predicting churn). **Fix:** fit every transform on the training fold only, and wrap the entire flow in a `Pipeline` so cross-validation re-fits transforms per fold.

```python
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.ensemble import HistGradientBoostingClassifier
from sklearn.model_selection import cross_val_score

num = Pipeline([("imp", SimpleImputer(strategy="median")),
                ("sc", StandardScaler())])
cat = Pipeline([("imp", SimpleImputer(strategy="most_frequent")),
                ("oh", OneHotEncoder(handle_unknown="ignore"))])
pre = ColumnTransformer([("num", num, num_cols), ("cat", cat,
    cat_cols)])

model = Pipeline([("pre", pre), ("clf",
    HistGradientBoostingClassifier())])
# Transforms are re-fit on each training fold -> no leakage.
scores = cross_val_score(model, X, y, cv=5, scoring="roc_auc")
```

```
print(scores.mean(), scores.std())
```

8 Model selection & tuning

- **Grid search** exhaustively tries combinations (expensive, fine for few hyperparameters).
- **Random search** samples combinations — usually more efficient than grid for the same budget.
- **Bayesian optimization (Optuna)** models the objective and proposes promising points; best for expensive models and larger search spaces. Use pruning to kill bad trials early.

8.1 Learning vs. validation curves

Learning curves plot train/validation score vs. *training-set size*: a large persistent gap signals high variance (get more data / regularize); both curves low and converged signals high bias (use a richer model / better features). **Validation curves** plot score vs. a single *hyperparameter* to find its sweet spot.

8.2 A/B testing for production models

Offline metrics do not always predict online impact. The gold standard is a randomized A/B test: split traffic, run the new model against the incumbent, and compare a pre-registered business metric with proper statistics (mind the multiple-comparisons and peeking problems from Phase 0). This bridges directly to Phase 7.

Phase 1

- **Hands-On Machine Learning** (Géron), Part I — the single best practical companion to this phase.
- Andrew Ng — *Machine Learning Specialization* (Coursera).
- **StatQuest** — trees, random forests, boosting, SVM, PCA series.
- **scikit-learn** user guide — exemplary documentation; read the "common pitfalls" page.
- **Kaggle** — read winning solution write-ups; they teach feature engineering better than any course.

9 Checkpoint project

End-to-end tabular case study

Take a real tabular dataset (e.g. Kaggle house prices or a customer-churn set) and produce a polished mini case study.

1. **EDA:** distributions, missingness, correlations, target relationship. Write down hypotheses.
2. **Feature engineering:** imputation, encoding, scaling, a few derived features — all inside a `Pipeline`.
3. **Modeling:** compare ≥4 models (linear/logistic + regularization, random forest, gradient boosting, and one more) via stratified cross-validation on a metric you justify.
4. **Tuning:** optimize the best model with Optuna; show the validation curve for at least one hyperparameter.
5. **Interpretation:** SHAP summary + dependence plots; explain the top drivers and sanity-check for leakage.
6. **Write-up:** a README framing the problem, decisions, results (with error bars), and limitations.

Definition of done: a reproducible repo, a single command to train, a held-out test score reported *once*, and a paragraph on what you would deploy and monitor (foreshadowing Phase 7).

10 Phase 1 self-check

- Choose and justify an evaluation metric for an imbalanced business problem.
- Explain bias–variance using a learning curve you drew.
- State when Ridge vs. Lasso vs. ElasticNet is appropriate and why.
- Explain why gradient boosting usually beats a single tree and a random forest on tabular data.
- Build a cross-validated, leakage-free pipeline and explain every transform's placement.
- Read a SHAP summary plot and identify a suspicious (leaky) feature.