# 50 Machine Learning Algorithms — Cheatsheet

> **Blueprints of Intelligence — 50 ML Algorithms decoded.**

## 💡 How to Use?

Each pattern includes:

- **When to use** (practical scenario)
- **Core idea** (intuition)
- **Complexity** (time/space, rough)
- **Python hint** (minimal example)
- **Pitfalls / Variations**

## A. 🎯 Supervised Learning (1–20)

### 1. Linear Regression

🎯 **When**: Predict continuous numeric values from features.

💡 **Idea**: Fit line/plane by minimizing mean squared error.

⏱️ **Complexity**: O(n·d²) closed form; O(n·d·it) gradient descent.

🐍 **Python**:

```
from sklearn.linear_model import LinearRegression
model = LinearRegression().fit(X, y)
```

⚠️ **Pitfalls**: Sensitive to multicollinearity & outliers.

🔀 **Variations**: Weighted regression, polynomial regression.

## 2. Logistic Regression

🎯 **When**: Binary classification (extendable to multi-class).

💡 **Idea**: Apply sigmoid to linear score; optimize log-likelihood.

⏱️ **Complexity**: O(n·d·it).

🐍 **Python**:

```
from sklearn.linear_model import LogisticRegression
clf = LogisticRegression().fit(X, y)
```

⚠️ **Pitfalls**: Assumes linear decision boundary.

🔀 **Variations**: Softmax regression for multi-class.

---

## 3. Ridge Regression (L2)

🎯 **When**: Regression with multicollinearity; need regularization.

💡 **Idea**: Penalize large coefficients with L2 norm.

🐍 **Python**:

```
from sklearn.linear_model import Ridge
model = Ridge(alpha=1.0).fit(X, y)
```

⚠️ **Pitfalls**: Doesn't perform feature selection.

🔀 **Variations**: Logistic ridge regression.

---

## 4. Lasso Regression (L1)

🎯 **When**: Regression with feature selection.

💡 **Idea**: L1 penalty shrinks some coefficients to zero.

🐍 **Python**:

```
from sklearn.linear_model import Lasso
model = Lasso(alpha=0.1).fit(X, y)
```

⚠️ **Pitfalls**: Unstable when features are correlated.

🔀 **Variations**: Sparse models in high dimensions.

---

## 5. Elastic Net

🎯 **When**: Need both shrinkage and feature selection.

💡 **Idea**: Weighted L1 + L2 penalties.

🐍 **Python**:

```
from sklearn.linear_model import ElasticNet
model = ElasticNet(l1_ratio=0.5).fit(X, y)
```

⚠️ **Pitfalls**: Extra hyperparameter (l1_ratio).

🔀 **Variations**: LARS solver for speed.

---

## 6. K-Nearest Neighbors (KNN)

🎯 **When**: Simple classification/regression baseline.

💡 **Idea**: Predict by majority (classification) or mean (regression) of k neighbors.

⏱️ **Complexity**: O(n·d) per query; O(n·d) memory.

🐍 **Python**:

```
from sklearn.neighbors import KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=5).fit(X, y)
```

⚠️ **Pitfalls**: Sensitive to irrelevant features & scaling.

🔀 **Variations**: Weighted KNN, KD-trees for speed.

---

## 7. Support Vector Machine (SVM)

🎯 **When**: Classification with clear margins, works well in high dimensions.

💡 **Idea**: Maximize margin; kernel trick allows nonlinear decision boundaries.

⏱️ **Complexity**: O($n^2$–$n^3$) worst-case; faster with linear solvers.

🐍 **Python**:

```
from sklearn.svm import SVC
clf = SVC(kernel="rbf").fit(X, y)
```

⚠️ **Pitfalls**: Expensive for very large datasets.

🔀 **Variations**: Linear SVM, v-SVM, One-class SVM.

---

## 8. Decision Tree

🎯 **When**: Interpretable models with non-linear interactions.

💡 **Idea**: Recursive partitioning of data by maximizing information gain.

🐍 **Python**:

```
from sklearn.tree import DecisionTreeClassifier
tree = DecisionTreeClassifier(max_depth=5).fit(X, y)
```

⚠️ **Pitfalls**: High variance, prone to overfitting.

🔀 **Variations**: CART, ID3, C4.5, regression trees.

---

## 9. Random Forest

🎯 **When**: Strong baseline for tabular data.

💡 **Idea**: Ensemble of bootstrapped trees with feature bagging.

🐍 **Python**:

```
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier(n_estimators=100).fit(X, y)
```

⚠️ **Pitfalls**: Large memory footprint.

🔀 **Variations**: Extremely Randomized Trees (ExtraTrees).

---

## 10. Gradient Boosting (GBM)

🎯 **When**: State-of-the-art for tabular predictive tasks.

💡 **Idea**: Sequentially add trees to correct previous residuals.

🐍 **Python**:

```
from sklearn.ensemble import GradientBoostingClassifier
gb = GradientBoostingClassifier().fit(X, y)
```

⚠️ **Pitfalls**: Sensitive to hyperparameters.

🔀 **Variations**: XGBoost, LightGBM, CatBoost.

## 11. Extra Trees (Extremely Randomized Trees)

🎯 **When**: Faster and more randomized alternative to RF.

💡 **Idea**: Use random split thresholds.

🐍 **Python**:

```
from sklearn.ensemble import ExtraTreesClassifier
et = ExtraTreesClassifier().fit(X, y)
```

⚠️ **Pitfalls**: Can increase bias.

🔀 **Variations**: Combine with bagging.

---

## 12. Naive Bayes

🎯 **When**: Text classification, spam filtering, NLP.

💡 **Idea**: Apply Bayes theorem assuming feature independence.

🐍 **Python**:

```
from sklearn.naive_bayes import MultinomialNB
nb = MultinomialNB().fit(X, y)
```

⚠️ **Pitfalls**: Independence assumption rarely true.

🔀 **Variations**: GaussianNB, BernoulliNB.

---

## 13. Quadratic Discriminant Analysis (QDA)

🎯 **When**: Classes have different covariances.

💡 **Idea**: Estimate quadratic decision boundary.

🐍 **Python**:

```
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
qda = QuadraticDiscriminantAnalysis().fit(X, y)
```

⚠️ **Pitfalls**: Needs many samples.

🔀 **Variations**: Shrinkage QDA.

---

## 14. Linear Discriminant Analysis (LDA)

🎯 **When**: Classes separable linearly with shared covariance.

💡 **Idea**: Project features to maximize class separation.

🐍 **Python**:

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
lda = LinearDiscriminantAnalysis().fit(X, y)
```

⚠️ **Pitfalls**: Assumes Gaussian distribution.

🔀 **Variations**: Regularized LDA.

---

## 15. Perceptron

🎯 **When**: Online linear classification baseline.

💡 **Idea**: Adjust weights on mistakes.

🐍 **Python**:

```
from sklearn.linear_model import Perceptron
perc = Perceptron().fit(X, y)
```

⚠️ **Pitfalls**: Only works if data is linearly separable.

🔀 **Variations**: Multi-layer perceptron (MLP).

---

## 16. Passive Aggressive Classifier

🎯 **When**: Large-scale online classification.

💡 **Idea**: Passive when correct, aggressive update when wrong.

🐍 **Python**:

```
from sklearn.linear_model import PassiveAggressiveClassifier
pa = PassiveAggressiveClassifier().fit(X, y)
```

⚠️ **Pitfalls**: Requires careful regularization.

---

## 17. Huber Regressor

🎯 **When**: Regression robust to outliers.

💡 **Idea**: Huber loss behaves like MSE near 0, MAE for outliers.

🐍 **Python**:

```
from sklearn.linear_model import HuberRegressor
huber = HuberRegressor().fit(X, y)
```

⚠️ **Pitfalls**: Requires tuning epsilon.

---

## 18. SGD Classifier

🎯 **When**: Large datasets with linear models.

💡 **Idea**: Train using stochastic gradient descent.

🐍 **Python**:

```
from sklearn.linear_model import SGDClassifier
sgd = SGDClassifier(loss="hinge").fit(X, y)
```

⚠️ **Pitfalls**: Sensitive to learning rate schedule.

---

## 19. Multi-layer Perceptron (MLP)

🎯 **When**: Small neural network for nonlinear classification.

💡 **Idea**: Fully connected layers + backpropagation.

🐍 **Python**:

```
from sklearn.neural_network import MLPClassifier
mlp = MLPClassifier(hidden_layer_sizes=(100,)).fit(X, y)
```

⚠️ **Pitfalls**: Tends to overfit small datasets.

🔀 **Variations**: Deep feed-forward nets in PyTorch/Keras.

---

## 20. Logistic Regression (One-vs-Rest)

🎯 **When**: Multi-class classification with binary base learners.

💡 **Idea**: Train 1 classifier per class vs others.

🐍 **Python**:

```
LogisticRegression(multi_class="ovr").fit(X, y)
```

⚠️ **Pitfalls**: May struggle with many classes.

🔀 **Variations**: One-vs-One classification.

---

# B. 🧩 Unsupervised Learning (21–30)

## 21. K-Means Clustering

🎯 **When**: Partition unlabeled data into *K* clusters (e.g., customer segmentation).

💡 **Idea**: Iteratively assign points to nearest centroid → update centroids until stable.

⏱️ **Complexity**: O(n·k·d·it).

🐍 **Python**:

```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=3, random_state=42).fit(X)
```

⚠️ **Pitfalls**: Sensitive to initialization and outliers.

🔀 **Variations**: K-Means++, Mini-Batch KMeans.

---

## 22. Hierarchical Agglomerative Clustering

🎯 **When**: When hierarchy or dendrogram visualization is useful.

💡 **Idea**: Start with each point as its own cluster, merge iteratively.

⏱️ **Complexity**: O($n^2$ log n).

🐍 **Python**:

```
from sklearn.cluster import AgglomerativeClustering
agg = AgglomerativeClustering(n_clusters=3, linkage="ward").fit(X)
```

⚠️ **Pitfalls**: Expensive for very large datasets.

🔀 **Variations**: Single-link, complete-link, average-link.

---

## 23. DBSCAN (Density-Based Spatial Clustering)

🎯 **When**: Arbitrary-shaped clusters + noise detection.

💡 **Idea**: Core points expand clusters; noise points left unassigned.

⏱️ **Complexity**: O(n log n) with KD-tree; worst $O(n^2)$.

🐍 **Python**:

```
from sklearn.cluster import DBSCAN
db = DBSCAN(eps=0.5, min_samples=5).fit(X)
```

⚠️ **Pitfalls**: Choosing eps and min_samples is tricky.

🔀 **Variations**: HDBSCAN for hierarchical density-based clustering.

---

## 24. Mean-Shift Clustering

🎯 **When**: Don't know number of clusters beforehand.

💡 **Idea**: Slide window toward high-density region (modes).

⏱️ **Complexity**: $O(n^2)$.

🐍 **Python**:

```
from sklearn.cluster import MeanShift
ms = MeanShift().fit(X)
```

⚠️ **Pitfalls**: Slow for large datasets.

🔀 **Variations**: Bandwidth selection critical.

---

## 25. Gaussian Mixture Models (GMMs)

🎯 **When**: Probabilistic soft clustering (e.g., speaker ID).

💡 **Idea**: Model as weighted sum of Gaussian distributions; EM for estimation.

⏱️ **Complexity**: O(n·k·d·it).

🐍 **Python**:

```
from sklearn.mixture import GaussianMixture
gmm = GaussianMixture(n_components=3).fit(X)
```

⚠️ **Pitfalls**: Sensitive to initialization, assumes Gaussian shapes.

🔀 **Variations**: Bayesian Gaussian Mixtures (auto choose K).

---

## 26. PCA (Principal Component Analysis)

🎯 **When**: Dimensionality reduction, visualization, noise filtering.

💡 **Idea**: Project data to orthogonal axes of max variance.

⏱️ **Complexity**: O(min(n·d², d³)).

🐍 **Python**:

```
from sklearn.decomposition import PCA
pca = PCA(n_components=2).fit_transform(X)
```

⚠️ **Pitfalls**: Linear only, loses interpretability.

🔀 **Variations**: Sparse PCA, Incremental PCA.

---

## 27. Kernel PCA

🎯 **When**: Nonlinear dimensionality reduction.

💡 **Idea**: Use kernel trick to project into high-dimensional space before PCA.

⏱️ **Complexity**: O(n³) for kernel matrix.

🐍 **Python**:

```
from sklearn.decomposition import KernelPCA
kpca = KernelPCA(kernel="rbf", gamma=0.1).fit_transform(X)
```

⚠️ **Pitfalls**: Memory-heavy for large n.

🔀 **Variations**: Polynomial kernel, sigmoid kernel.

---

## 28. Independent Component Analysis (ICA)

🎯 **When**: Blind source separation (e.g., separating audio signals).

💡 **Idea**: Find statistically independent components.

⏱️ **Complexity**: O(n·d²·it).

🐍 **Python**:

```
from sklearn.decomposition import FastICA
ica = FastICA(n_components=2).fit_transform(X)
```

⚠️ **Pitfalls**: Requires whitening; sensitive to noise.

🔀 **Variations**: Robust ICA.

---

### 29. t-SNE (t-distributed Stochastic Neighbor Embedding)

🎯 **When**: Visualize high-dimensional data in 2D/3D.

💡 **Idea**: Preserve local neighborhood similarities by minimizing KL divergence.

⏱️ **Complexity**: $O(n^2)$.

🐍 **Python**:

```
from sklearn.manifold import TSNE
tsne = TSNE(n_components=2, perplexity=30).fit_transform(X)
```

⚠️ **Pitfalls**: Only for visualization; not scalable beyond ~10k points.

🔀 **Variations**: Barnes-Hut t-SNE, Multicore t-SNE.

---

### 30. UMAP (Uniform Manifold Approximation & Projection)

🎯 **When**: Faster scalable alternative to t-SNE.

💡 **Idea**: Graph-based manifold learning with local/global preservation.

⏱️ **Complexity**: O(n log n).

🐍 **Python**:

```
import umap
embedding = umap.UMAP(n_neighbors=15, min_dist=0.1).fit_transform(X)
```

⚠️ **Pitfalls**: Parameters (n_neighbors, min_dist) strongly affect result.

🔀 **Variations**: Supervised UMAP.

---

## C. 🤝 Ensemble & Advanced (31–40)

---

### 31. AdaBoost (Adaptive Boosting)

🎯 **When**: Need a strong classifier from many weak learners.

💡 **Idea**: Iteratively reweight misclassified samples → train weak learners (usually stumps) → combine weighted votes.

⏱️ **Complexity**: O(n·d·T), T = #learners.

🐍 **Python**:

```
from sklearn.ensemble import AdaBoostClassifier
ada = AdaBoostClassifier(n_estimators=50).fit(X, y)
```

⚠️ **Pitfalls**: Sensitive to noisy data and outliers.

🔀 **Variations**: AdaBoostRegressor, SAMME (multi-class).

---

## 32. Gradient Boosting (GBM)

🎯 **When**: Stronger than AdaBoost; handles regression and classification.

💡 **Idea**: Sequentially fit models to residuals with gradient descent approach.

⏱️ **Complexity**: O(n·d·T).

🐍 **Python**:

```
from sklearn.ensemble import GradientBoostingClassifier
gb = GradientBoostingClassifier(n_estimators=100, learning_rate=0.1).fit(X, y)
```

⚠️ **Pitfalls**: Can overfit if learning rate too high.

🔀 **Variations**: Regularized GBM, stochastic GBM.

---

## 33. XGBoost (Extreme Gradient Boosting)

🎯 **When**: Large-scale datasets; Kaggle competitions.

💡 **Idea**: Optimized GBM with regularization, shrinkage, sparsity-aware learning.

⏱️ **Complexity**: O(n·d·T) with optimizations.

🐍 **Python**:

```
import xgboost as xgb
```

```
model = xgb.XGBClassifier(n_estimators=200, max_depth=5).fit(X, y)
```

⚠️ **Pitfalls**: Many hyperparameters → requires tuning.

🔀 **Variations**: GPU XGBoost, DART (dropout boosting).

---

## 34. LightGBM

🎯 **When**: Large datasets, high-dimensional features, categorical handling.

💡 **Idea**: Histogram-based leaf-wise growth (faster than depth-wise).

⏱️ **Complexity**: O(n·log n).

🐍 **Python**:

```
import lightgbm as lgb
lgbm = lgb.LGBMClassifier(num_leaves=31).fit(X, y)
```

⚠️ **Pitfalls**: Leaf-wise growth may overfit.

🔀 **Variations**: LightGBM GPU mode, categorical split support.

---

## 35. CatBoost

🎯 **When**: Many categorical features with minimal preprocessing.

💡 **Idea**: Ordered boosting to reduce overfitting; built-in categorical encoding.

⏱️ **Complexity**: O(n·d·T).

🐍 **Python**:

```
from catboost import CatBoostClassifier
cat = CatBoostClassifier(iterations=200, depth=6, verbose=0).fit(X, y)
```

⚠️ **Pitfalls**: Training slower than LightGBM.

🔀 **Variations**: CatBoostRegressor.

---

## 36. Bagging (Bootstrap Aggregating)

🎯 **When**: Reduce variance of high-variance learners (e.g., decision trees).

💡 **Idea**: Train models on bootstrap samples → average/vote predictions.

⏱️ **Complexity**: O(n·d·T).

🐍 **Python**:

```
from sklearn.ensemble import BaggingClassifier
bag = BaggingClassifier(n_estimators=50).fit(X, y)
```

⚠️ **Pitfalls**: Doesn't reduce bias.

🔀 **Variations**: Bagging with SVMs, regressors.

---

## 37. Stacking (Stacked Generalization)

🎯 **When**: Combine diverse models into a meta-model.

💡 **Idea**: Base models predict → meta-learner combines predictions.

⏱️ **Complexity**: O(T·n·d) + meta-model training.

🐍 **Python**:

```
from sklearn.ensemble import StackingClassifier
from sklearn.linear_model import LogisticRegression
estimators = [("rf", rf), ("gb", gb)]
stack = StackingClassifier(estimators=estimators, final_estimator=LogisticRegression())
stack.fit(X, y)
```

⚠️ **Pitfalls**: Risk of leakage if not using CV for meta-model.

🔀 **Variations**: Stacking regressors, deep stacking.

---

## 38. Voting Classifier

🎯 **When**: Need a simple ensemble of different models.

💡 **Idea**: Combine predictions via majority vote (hard) or average probs (soft).

⏱️ **Complexity**: Low, just aggregates predictions.

🐍 **Python**:

```
from sklearn.ensemble import VotingClassifier
vote = VotingClassifier(estimators=[("rf", rf), ("svc", clf)], voting="soft").fit(X, y)
```

⚠️ **Pitfalls**: Models must be reasonably calibrated.

🔀 **Variations**: Hard vs soft voting.

---

## 39. Bayesian Optimization Models

🎯 **When**: Hyperparameter tuning for expensive models.

💡 **Idea**: Use surrogate (Gaussian Process, TPE) to guide search of hyperparameters.

⏱️ **Complexity**: Depends on surrogate model.

🐍 **Python**:

```
import optuna
def objective(trial):
    lr = trial.suggest_float("lr", 1e-4, 1e-1, log=True)
    n = trial.suggest_int("n_estimators", 50, 300)
    model = XGBClassifier(learning_rate=lr, n_estimators=n)
    return cross_val_score(model, X, y).mean()
study = optuna.create_study(direction="maximize")
study.optimize(objective, n_trials=50)
```

⚠️ **Pitfalls**: Slower than grid/random for cheap models.

🔀 **Variations**: Tree Parzen Estimator (TPE), Hyperband.

---

## 40. Blending

🎯 **When**: Quick ensemble with a validation holdout.

💡 **Idea**: Train models → combine predictions via weighted average.

⏱️ **Complexity**: Similar to stacking but simpler.

🐍 **Python**:

```
preds = 0.7*rf.predict_proba(X_val) + 0.3*gb.predict_proba(X_val)
```

⚠️ **Pitfalls**: Needs careful holdout selection.

🔀 **Variations**: Linear blending, nonlinear blending.

---

# D. 🧠 Neural Networks & Deep Learning (41–45)

---

## 41. Convolutional Neural Networks (CNNs)

🎯 **When**: Image classification, object detection, vision tasks.

💡 **Idea**: Convolution filters extract spatial features; pooling reduces dimensions.

⏱️ **Complexity**: $O(n \cdot k \cdot f^2)$ where f = filter size.

🐍 **Python**:

```
from tensorflow.keras import Sequential, layers
model = Sequential([
    layers.Conv2D(32, (3,3), activation="relu", input_shape=(28,28,1)),
    layers.MaxPooling2D(2,2),
    layers.Flatten(),
    layers.Dense(10, activation="softmax")
])
```

⚠️ **Pitfalls**: Requires large data; prone to overfitting.

🔀 **Variations**: ResNet, VGG, EfficientNet, Inception.

---

## 42. Recurrent Neural Networks (RNNs)

🎯 **When**: Sequence data (text, time series, speech).

💡 **Idea**: Hidden state carries info through timesteps.

⏱️ **Complexity**: $O(n \cdot d \cdot h)$ per sequence.

🐍 **Python**:

```
from tensorflow.keras import Sequential, layers
model = Sequential([layers.SimpleRNN(64), layers.Dense(1)])
```

⚠️ **Pitfalls**: Vanishing/exploding gradients.

🔀 **Variations**: BiRNNs, stacked RNNs.

---

## 43. Long Short-Term Memory (LSTM)

🎯 **When**: Sequences with long-term dependencies.

💡 **Idea**: Gated cells (input, forget, output) regulate memory.

⏱️ **Complexity**: $O(n \cdot d \cdot h^2)$.

🐍 **Python**:

```
from tensorflow.keras import Sequential, layers
model = Sequential([layers.LSTM(128), layers.Dense(1)])
```

⚠️ **Pitfalls**: Computationally heavy.

🔀 **Variations**: Peephole LSTMs, bidirectional LSTMs.

---

### 44. Gated Recurrent Units (GRU)

🎯 **When**: Faster, lighter alternative to LSTM.

💡 **Idea**: Combines input & forget gate into update gate.

⏱️ **Complexity**: ~30% fewer parameters than LSTM.

🐍 **Python**:

```
from tensorflow.keras import Sequential, layers
model = Sequential([layers.GRU(128), layers.Dense(1)])
```

⚠️ **Pitfalls**: Sometimes less expressive than LSTM.

🔀 **Variations**: Bidirectional GRUs.

---

### 45. Transformers

🎯 **When**: NLP, audio, large-scale sequence modeling.

💡 **Idea**: Self-attention replaces recurrence → parallelizable.

⏱️ **Complexity**: $O(n^2 \cdot d)$ for attention.

🐍 **Python**:

```
from transformers import AutoModel, AutoTokenizer
tok = AutoTokenizer.from_pretrained("bert-base-uncased")
model = AutoModel.from_pretrained("bert-base-uncased")
```

⚠️ **Pitfalls**: High compute/memory cost.

🔀 **Variations**: BERT, GPT, T5, Vision Transformers.

---

## E. 🛠️ Other Useful Algorithms (46–50)

## 46. K-Medoids

🎯 **When**: Robust clustering (less sensitive to outliers).

💡 **Idea**: Like K-Means but cluster centers = actual data points (medoids).

⏱️ **Complexity**: O($n^2 \cdot k \cdot it$).

🐍 **Python**:

```
from sklearn_extra.cluster import KMedoids
kmed = KMedoids(n_clusters=3).fit(X)
```

⚠️ **Pitfalls**: Slower than K-Means.

🔀 **Variations**: Partitioning Around Medoids (PAM).

## 47. Self-Organizing Maps (SOMs)

🎯 **When**: Dimensionality reduction, visualization, topology-preserving.

💡 **Idea**: Neural-grid learns to represent data structure.

⏱️ **Complexity**: O(n·d·epochs).

🐍 **Python**: (3rd-party libs e.g. MiniSom)

```
from minisom import MiniSom
som = MiniSom(7, 7, X.shape[1])
som.train_random(X, 100)
```

⚠️ **Pitfalls**: Outdated vs modern DR techniques.

🔀 **Variations**: Growing SOM.

## 48. Hidden Markov Models (HMMs)

🎯 **When**: Sequential data with latent states (speech, POS tagging).

💡 **Idea**: Transition probabilities between hidden states + emission probabilities.

⏱️ **Complexity**: O($n \cdot s^2$), s = states.

🐍 **Python**:

```
from hmmlearn import hmm
model = hmm.GaussianHMM(n_components=3).fit(X)
```

⚠️ **Pitfalls**: Assumes Markov property; limited flexibility.

🔀 **Variations**: Continuous HMM, factorial HMM.

---

## 49. Q-Learning (Reinforcement Learning)

🎯 **When**: Learn decision-making policies via rewards.

💡 **Idea**: Update Q-values iteratively with Bellman equation.

⏱️ **Complexity**: O(states·actions).

🐍 **Python**:

```
import numpy as np
Q = np.zeros((n_states, n_actions))
# update rule
Q[s,a] = Q[s,a] + alpha*(r + gamma*np.max(Q[s_next]) - Q[s,a])
```

⚠️ **Pitfalls**: Doesn't scale to large state spaces.

🔀 **Variations**: Deep Q-Networks (DQN).

---

## 50. Association Rule Learning (Apriori, Eclat)

🎯 **When**: Market basket analysis, discover frequent itemsets.

💡 **Idea**: Mine frequent patterns, derive rules with support/confidence.

⏱️ **Complexity**: Exponential worst-case, optimized with pruning.

🐍 **Python**:

```
from mlxtend.frequent_patterns import apriori, association_rules
freq = apriori(df, min_support=0.1, use_colnames=True)
rules = association_rules(freq, metric="lift", min_threshold=1.2)
```

⚠️ **Pitfalls**: Generates too many rules; need thresholds.

🔀 **Variations**: FP-Growth algorithm.

---

# ⚡ Cheatsheet Extras

### Model Selection Guide

- **Tabular data** → Random Forest, LightGBM, CatBoost
- **Text** → TF-IDF + Logistic / Transformers
- **Images** → CNN / pretrained backbones
- **Time series** → Feature engineering + LSTM/Transformers
- **Small data** → Regularized linear, Naive Bayes
- **Large-scale** → XGBoost/LightGBM, SGD

### Best Practices

- Scale for distance-based models (KNN, SVM).
- Handle imbalance with class weights / sampling.
- Use cross-validation (stratified/group).
- Automate pipelines (sklearn.Pipeline).
- Monitor drift & feature importance in production.

---