IML Summary/Cheat Sheet

Unfortunately, we're only allowed to bring a hand-written cheatsheet to the exam. Therefore,
THIS CANNOT BE USED AT THE EXAM, YOU MUST HANDWRITE YOUR OWN CHEATSHEET.

However, this may serve as a reference/example of what to include.

1 Mathematical Foundations

1.1 Parametrized Hyperplane

Standard Form: In $\mathbb{R}^N$, a hyperplane is defined as:

$$\vec{w} \cdot \vec{x} = 0, \quad \text{with } \|\vec{w}\| = 1$$

where $\vec{w} = [w_1, \ldots, w_N]$ and $\vec{x} = [x_1, \ldots, x_N]$.
Extended Form: Including bias term:

$$\vec{w} \cdot \vec{x} + b = 0$$

Alternatively, using extended vectors $\tilde{\vec{w}} = [w_0, w_1, \ldots, w_N]$ and $\tilde{\vec{x}} = [1, x_1, \ldots, x_N]$:

$$\tilde{\vec{w}} \cdot \tilde{\vec{x}} = 0, \quad \text{with } \tilde{\vec{x}} = [1|\vec{x}]$$

Key Insight: When using normalized augmented vectors, the first component is always $x_0 = 1$, which corresponds to the bias term.

1.3 Logistic Regression

Overview: Parametric model for binary classification that predicts probabilities using the sigmoid function.
Model: Prediction using sigmoid:

$$y_n = \sigma(\tilde{\vec{w}} \cdot \tilde{\vec{x}}_n)$$

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

Cross-Entropy Loss Function:

$$E(\tilde{\vec{w}}) = -\sum_n \{t_n \ln y_n + (1 - t_n) \ln(1 - y_n)\}$$

where $t_n \in \{0, 1\}$ are target labels and y_n are predicted probabilities.

1.2 Distance Formulations

Signed Distance (Standard): Distance from point $\vec{x}_0$ to hyperplane:

$$d_{\text{signed}} = \frac{\vec{w} \cdot \vec{x}_0 + b}{\|\vec{w}\|}$$

Interpretation:
- $h = 0$: Point on decision boundary
- $h > 0$: Point on one side
- $h < 0$: Point on the other side

Reformulated Signed Distance: Using normalized extended vectors:

$$\tilde{\vec{w}}' = \frac{\tilde{\vec{w}}}{\|\vec{w}\|} = \left[\frac{w_0}{\|\vec{w}\|}, \frac{\vec{w}}{\|\vec{w}\|}\right]$$

Key Result:

$$d_{\text{signed}} = \frac{\tilde{\vec{w}} \cdot \tilde{\vec{x}}}{\|\vec{w}\|}, \quad \forall \tilde{\vec{w}} \in \mathbb{R}^{N+1}$$

Gradient: For gradient descent:

$$\nabla E(\tilde{\vec{w}}) = \sum_n (y_n - t_n) \tilde{\vec{x}}_n$$

Weight Update:

$$\tilde{\vec{w}} \leftarrow \tilde{\vec{w}} - \alpha \nabla E(\tilde{\vec{w}})$$

where α is the learning rate.
Decision Rule: Classify based on threshold:

$$\hat{y} = \begin{cases} 1 & \text{if } y_n \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

2 Support Vector Machines

2.1 Hyperplane Fundamentals

General Hyperplane Equation: In d-dimensional space:

$$w_1x_1 + w_2x_2 + \cdots + w_dx_d + b = 0$$

or $\vec{w}^T\vec{x} + b = 0$ where $\vec{w} \in \mathbb{R}^d$ is the normal vector and $b \in \mathbb{R}$ is the bias term.

Signed Distance: Distance from point $\vec{x}_0$ to hyperplane $\vec{w}^T\vec{x} + b = 0$:

$$d_{\text{signed}}(\vec{x}_0) = \frac{\vec{w}^T\vec{x}_0 + b}{\|\vec{w}\|}$$

- Positive: Point on same side as normal vector $\vec{w}$
- Negative: Point on opposite side of normal vector $\vec{w}$
- Zero: Point lies on the hyperplane

Unsigned Distance: Absolute distance from point to hyperplane:

$$d_{\text{unsigned}}(\vec{x}_0) = \frac{|\vec{w}^T\vec{x}_0 + b|}{\|\vec{w}\|}$$

Geometric Interpretation:

- Decision boundary: $\vec{w}^T\vec{x} + b = 0$
- Positive margin boundary: $\vec{w}^T\vec{x} + b = +1$
- Negative margin boundary: $\vec{w}^T\vec{x} + b = -1$
- Margin width: $\frac{2}{\|\vec{w}\|}$ (distance between parallel hyperplanes)

Regularization Parameter C - Detailed Analysis:

- **Large C (e.g., $C \gg 1$):**
  - Hard margin behavior, heavily penalizes misclassification
  - Low bias, high variance (prone to overfitting)
  - More support vectors, complex decision boundary
- **Small C (e.g., $C \ll 1$):**
  - Soft margin behavior, tolerates misclassification
  - High bias, low variance (may underfit)
  - Fewer support vectors, simpler decision boundary
- **Selection:** Use cross-validation with grid search over $C \in \{10^{-3}, 10^{-2}, \ldots, 10^3\}$

2.2 Linear SVM

Hard-Margin Objective: For linearly separable data, find hyperplane with maximum margin:

$$\min \frac{1}{2}\|\vec{w}\|^2 \quad \text{subject to} \quad y_i(\vec{w}^T\vec{x}_i + b) \geq 1 \quad \forall i$$

Soft-Margin Objective: For non-separable data, allow some misclassification with slack variables:

$$\min \frac{1}{2}\|\vec{w}\|^2 + C\sum_{i=1}^{n}\xi_i$$

subject to $y_i(\vec{w}^T\vec{x}_i + b) \geq 1 - \xi_i, \quad \xi_i \geq 0$ where C is the regularization parameter controlling the trade-off between margin maximization and classification errors.

Margin (γ): The margin is the distance between the hyperplane and the nearest data point of any class. For a normalized weight vector, it is defined as:

$$\gamma = \frac{1}{\|\vec{w}\|}$$

The goal is to maximize this margin, which is equivalent to minimizing $\|\vec{w}\|^2$.

Support Vectors: Points that lie on the margin boundaries:

- Hard margin: Points where $y_i(\vec{w}^T\vec{x}_i + b) = 1$
- Soft margin: Points where $y_i(\vec{w}^T\vec{x}_i + b) = 1 - \xi_i$ or $\xi_i > 0$

2.3 SVM Dual Formulation

Why Dual Form?:

- Easier to solve constrained optimization
- Enables kernel trick (dot products only)
- Reveals which points are support vectors
- Handles non-separable data naturally

From Primal to Dual: Using Lagrange multipliers, the primal problem:

$$\min \frac{1}{2}\|\vec{w}\|^2 + C\sum_{i=1}^{n}\xi_i$$

becomes the dual problem:

Dual Optimization Problem:

$$\max \sum_{i=1}^{n}\alpha_i - \frac{1}{2}\sum_{i,j=1}^{n}\alpha_i\alpha_j y_i y_j \vec{x}_i^T\vec{x}_j$$

subject to $0 \leq \alpha_i \leq C$ and $\sum_{i=1}^{n}\alpha_i y_i = 0$.

Dual Variables Interpretation:

- $\alpha_i = 0$: Point not a support vector
- $0 < \alpha_i < C$: Point on margin boundary
- $\alpha_i = C$: Point violates margin (misclassified or inside margin)

Recovery of Primal Solution:

$$\vec{w} = \sum_{i=1}^{n}\alpha_i y_i \vec{x}_i \quad \text{(only support vectors contribute)}$$

2.4 Kernelized SVM

Kernel Trick: Replace dot products $\vec{x}_i^T \vec{x}_j$ with $K(\vec{x}_i, \vec{x}_j)$:

$$K(\vec{x}, \vec{x}') = \phi(\vec{x})^T \phi(\vec{x}')$$

Kernelized Dual Problem:

$$\max \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} \alpha_i \alpha_j y_i y_j K(\vec{x}_i, \vec{x}_j)$$

Decision Function:

$$f(\vec{x}) = \text{sign}\left(\sum_{i \in SV} \alpha_i y_i K(\vec{x}_i, \vec{x}) + b \right)$$

where SV = support vector indices.

Common Kernels:

- **Linear:** $K(\vec{x}, \vec{x}') = \vec{x}^T \vec{x}'$
- **Polynomial:** $K(\vec{x}, \vec{x}') = (\vec{x}^T \vec{x}' + c)^d$
- **RBF:** $K(\vec{x}, \vec{x}') = \exp(-\gamma \|\vec{x} - \vec{x}'\|^2)$

Polynomial Feature Mapping: For degree $d = 2$ in 2D:

$$\phi(\vec{x}) = [1, x_1, x_2, x_1^2, \sqrt{2} x_1 x_2, x_2^2]$$

Cover's Theorem: *"Complex patterns are more likely to be linearly separable in high-dimensional space."*

Polynomial Kernel Benefits:

- **Efficiency:** $O(n)$ vs $O(n^d)$ computation
- **Memory:** No explicit feature storage
- **Implicit mapping:** Works in feature space automatically

Dimension Growth: From $\mathbb{R}^n \to \mathbb{R}^M$ where:

$$M = \binom{n+d}{d} = \frac{(n+d)!}{n! d!}$$

Example: Circle to Line: Quadratic kernel transforms:

$$\vec{x} = [x_1, x_2] \mapsto \phi(\vec{x}) = [x_1^2, \sqrt{2} x_1 x_2, x_2^2]$$

Result: Circular boundaries become linear in feature space.

Kernel Selection Guidelines:

- **Linear:** High-dimensional data, text classification
- **Polynomial:** Image processing, specific polynomial patterns
- **RBF:** General purpose, smooth boundaries

3 AdaBoost (Adaptive Boosting)

3.1 Overview

Concept: Ensemble method that combines multiple weak learners (typically decision stumps) to create a strong classifier by iteratively focusing on misclassified examples.

Key Idea:

- Train weak learners sequentially
- Increase weights of misclassified examples
- Weight final votes by classifier performance
- Final prediction via weighted majority vote

Weak Learner: A classifier that performs slightly better than random guessing (error < 0.5). Common choice: decision stumps (single-split decision trees).

Algorithm: For training set $\chi = \{\vec{x}_n, t_n\}$ where $t_n \in \{-1, 1\}$ for $1 \leq n \leq N$:

1. Initialize data weights: $\forall n, w_n^1 = \frac{1}{N}$
2. For $t = [1, \ldots, T]$:
   (a) Find classifier $y_t : \chi \to \{-1, 1\}$ that minimizes weighted error $\sum_{t_n \neq y_t(\vec{x}_n)} w_n^t$
   (b) Evaluate:
   $$\epsilon_t = \frac{\sum_{t_n \neq y_t(\vec{x}_n)} w_n^t}{\sum_{n=1}^{N} w_n^t}$$
   $$\alpha_t = \log\left(\frac{1 - \epsilon_t}{\epsilon_t}\right)$$
   (c) Update weights: $w_n^{t+1} = w_n^t \exp(\alpha_t I(t_n \neq y_t(\vec{x}_n)))$

3.2 Mathematical Details

Final Classifier: Weighted combination of weak learners:

$$Y(\vec{x}) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t y_t(\vec{x}) \right)$$

Classifier Weight Interpretation:

- $\epsilon_t < 0.5 \Rightarrow \alpha_t > 0$ (good classifier gets positive weight)
- $\epsilon_t = 0.5 \Rightarrow \alpha_t = 0$ (random classifier gets zero weight)
- $\epsilon_t \to 0 \Rightarrow \alpha_t \to \infty$ (perfect classifier gets high weight)

Outlier Sensitivity Problem: Why AdaBoost Overfits to Outliers:

Weight Explosion Mechanism:

- Outliers are consistently misclassified by weak learners
- Weight update: $w_i^{(t+1)} = w_i^{(t)} \exp(\alpha_t)$ for misclassified samples
- Since $\alpha_t > 0$, outlier weights grow exponentially
- After T iterations: $w_{\text{outlier}} \propto \exp(\sum_{t=1}^{T} \alpha_t)$

Overfitting Consequences: Outliers dominate training, causing weak learners to focus on noise rather than patterns, creating complex boundaries that memorize outliers instead of generalizing.

4 Decision Trees & Ensembles

4.1 Decision Trees (Classification)

Core Concept: Tree-based model that recursively splits data based on feature values to create decision boundaries parallel to axes.

Tree Construction Algorithm:

1. Start with all training data at root
2. For each feature and threshold, compute split quality
3. Choose best split that maximizes information gain
4. Recursively split child nodes
5. Stop when stopping criterion met

Splitting Criteria:

- **Gini Impurity:** $G = 1 - \sum_{i=1}^{C} p_i^2$
- **Entropy:** $H = -\sum_{i=1}^{C} p_i \log_2(p_i)$
- **Information Gain:** $IG = H_{parent} - \sum_{child} \frac{|child|}{|parent|} H_{child}$

where p_i = proportion of class i, C = number of classes.

Best Split Selection: For feature f and threshold t:

$$\text{Split}(f,t) : x_f \leq t \to \text{left}, \quad x_f > t \to \text{right}$$

Stopping Criteria:

- Maximum depth reached
- Minimum samples per leaf/split
- No improvement in purity
- Maximum number of leaves

Prediction: For new sample, traverse tree from root to leaf:

$$\hat{y} = \text{majority class in leaf node}$$

Advantages:

- Interpretable (white-box model)
- Handles mixed data types
- No assumptions about data distribution
- Built-in feature selection
- Handles missing values naturally

Disadvantages:

- High variance (overfitting)
- Biased toward features with more levels
- Axis-parallel splits only
- Instability (small data changes $\to$ different tree)

Pruning Techniques:

- **Pre-pruning:** Stop early based on criteria
- **Post-pruning:** Build full tree, then remove branches
- **Cost complexity:** Minimize $Error + \alpha \times |leaves|$

Complexity:

- **Training:** $O(n \log n \times m)$ where n = samples, m = features
- **Prediction:** $O(\log n)$ average, $O(n)$ worst case

4.2 Decision Tree Regression

Regression Splitting Criterion:

- **MSE:** $MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \bar{y})^2$
- **MAE:** $MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \bar{y}|$

where $\bar{y}$ = mean target value in node.

Split Quality: Choose split that minimizes weighted average:

$$\text{Quality} = \frac{|left|}{|total|} \times MSE_{left} + \frac{|right|}{|total|} \times MSE_{right}$$

Prediction: Leaf node prediction:

$$\hat{y} = \frac{1}{|leaf|}\sum_{i \in leaf} y_i \quad \text{(mean of leaf samples)}$$

Key Differences from Classification:

- Continuous target values
- Use MSE/MAE instead of Gini/Entropy
- Predict mean instead of majority class
- Same tree structure and algorithms

Regularization:

- **min_samples_split:** Minimum samples to split
- **min_samples_leaf:** Minimum samples per leaf
- **max_depth:** Maximum tree depth
- **max_leaf_nodes:** Limit total leaves

4.3 Random Forests

Ensemble Method: Combines multiple decision trees trained on different subsets of data and features.

Algorithm:

1. For $b = 1, 2, \ldots, B$ trees:
2. Sample n training examples with replacement (bootstrap)
3. At each split, randomly select $m < M$ features
4. Build tree using only selected features
5. Combine predictions: vote (classification) or average (regression)

Key Parameters:

- **n_estimators:** Number of trees B
- **max_features:** Features per split m
- Typical choice: $m = \sqrt{M}$ (classification), $m = M/3$ (regression)
- **bootstrap:** Sample with replacement

Final Prediction:

- **Classification:** $\hat{y} = \text{mode}(\{h_1(\vec{x}), \ldots, h_B(\vec{x})\})$
- **Regression:** $\hat{y} = \frac{1}{B}\sum_{b=1}^{B} h_b(\vec{x})$

Why Random Forests Work:

- **Bagging:** Reduces variance through averaging
- **Feature randomness:** Decorrelates trees
- **Bootstrap sampling:** Each tree sees different data
- **Wisdom of crowds:** Ensemble ¿ individual trees

Advantages:

- Lower overfitting than single trees
- Handles large datasets efficiently
- Provides feature importance
- Works well out-of-the-box
- Parallel training possible

Feature Importance: Based on impurity decrease:

$$Importance_j = \sum_{t \in splits\_on\_j} \frac{|samples_t|}{|total|} \times \Delta_{impurity}$$

Out-of-Bag (OOB) Error:

- Each tree trained on 63% of data
- Remaining 37% used for validation
- OOB error estimates generalization without separate validation set

4.4 Ensemble Cascades & Boosting Trees

Gradient Boosting Trees: Sequential ensemble where each tree corrects previous errors.

Algorithm:

1. Initialize: $F_0(\vec{x}) = \arg\min_\gamma \sum_{i=1}^{n} L(y_i, \gamma)$
2. For $m = 1, 2, \ldots, M$:
3. Compute residuals: $r_{im} = -\frac{\partial L(y_i, F_{m-1}(\vec{x}_i))}{\partial F_{m-1}(\vec{x}_i)}$
4. Train tree h_m on $\{(\vec{x}_i, r_{im})\}$
5. Update: $F_m(\vec{x}) = F_{m-1}(\vec{x}) + \eta \cdot h_m(\vec{x})$

Learning Rate: η controls step size:

- Small η: More trees needed, better generalization
- Large η: Faster training, risk of overfitting
- Typical range: $\eta \in [0.01, 0.3]$

Cascade Architecture:

- **Early exit:** Easy samples classified by shallow trees
- **Progressive depth:** Later stages handle harder cases
- **Computational efficiency:** Avoid deep processing for simple inputs

XGBoost Enhancements:

- **Regularization:** $\Omega(f) = \gamma T + \frac{1}{2}\lambda\|\vec{w}\|^2$
- **Second-order approximation:** Uses both gradient and Hessian
- **Column sampling:** Random feature selection per tree
- **Parallel processing:** Efficient implementation

Stacking Cascades:

- **Level 0:** Base learners (trees, SVM, etc.)
- **Level 1:** Meta-learner combines base predictions
- **Cross-validation:** Prevent overfitting in meta-learner

Model Selection Guidelines:

- **Single Tree:** Interpretability needed
- **Random Forest:** Balanced performance, robustness
- **Gradient Boosting:** Maximum accuracy, careful tuning
- **Cascade:** Computational efficiency important

Hyperparameter Tuning:

- Trees: max_depth, min_samples_split
- RF: n_estimators, max_features
- Boosting: learning_rate, n_estimators, regularization

5 Data Preprocessing

5.1 Normalization & Standardization

- **Z-score Standardization:** $x' = \frac{x-\mu}{\sigma}$ — Mean 0, std 1
- **Unit Vector Scaling:** $x' = \frac{x}{\|x\|}$ — Scales to unit norm

6 Cross-Validation

6.1 Validation Techniques

- **K-Fold CV:** Split data into k folds, train on $k-1$, validate on 1, repeat k times

7 Bias-Variance Tradeoff

Error Decomposition:

$$\text{Expected Error} = \text{Bias}^2 + \text{Variance} + \text{Irreducible Error}$$

What is Bias in ML?: Bias measures how far off the average prediction is from the true value across different training sets.
Intuition: Bias captures the model's ability to learn the underlying pattern. High bias means the model makes systematic errors.
Mathematical: Bias $= E[\hat{f}(\vec{x})] - f(\vec{x})$ where $\hat{f}$ is our learned model and f is the true function.

High Bias (Underfitting):
- **Cause:** Model too simple to capture true relationship
- **Example:** Linear model for nonlinear data
- **Symptoms:** Poor performance on both training and test sets
- **Solutions:** Increase model complexity, add features, reduce regularization

Examples of High Bias Models:
- Linear regression for polynomial relationships
- Shallow neural networks for complex patterns
- Decision trees with very few splits
- Over-regularized models (large λ in Ridge/Lasso)

What is Variance in ML?: Variance measures how much the predictions change when trained on different datasets.
Intuition: Variance captures the model's sensitivity to training data. High variance means the model is inconsistent across different training sets.
Mathematical: Variance $= E[(\hat{f}(\vec{x}) - E[\hat{f}(\vec{x})])^2]$

High Variance (Overfitting):
- **Cause:** Model too complex, memorizes training noise
- **Example:** High-degree polynomial, deep neural network
- **Symptoms:** Great training performance, poor test performance
- **Solutions:** Reduce complexity, add regularization, more data

Examples of High Variance Models:
- High-degree polynomial regression
- Deep neural networks without regularization
- Decision trees with no pruning
- KNN with very small k values
- Under-regularized models (small λ)

The Tradeoff:
- **Simple models:** High bias, low variance
- **Complex models:** Low bias, high variance
- **Goal:** Find optimal complexity that minimizes total error
- **Sweet spot:** Balance both via cross-validation

8 Evaluation Metrics

8.1 Confusion Matrix

	Predicted Positive	**Predicted Negative**
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

8.2 Classification Metrics

Primary Metrics:

- **Accuracy** $= \frac{TP+TN}{TP+TN+FP+FN}$
  Interpretation: Overall correctness across all classes. *When to use:* Use when classes are balanced and misclassification costs are equal for all classes.
- **Precision** $= \frac{TP}{TP+FP}$
  Interpretation: Of all positive predictions, how many were correct? Critical when false positives are costly. *When to use:* Use when the cost of a false positive is high (e.g., spam detection, recommending a product that a user won't like, medical diagnosis where a false positive leads to unnecessary treatment).
- **Recall (Sensitivity)** $= \frac{TP}{TP+FN}$
  Interpretation: Of all actual positives, how many were correctly identified? Critical when false negatives are costly. *When to use:* Use when the cost of a false negative is high (e.g., disease detection where a missed case is critical, fraud detection where missing a fraudulent transaction is bad).
- **Specificity** $= \frac{TN}{TN+FP}$
  Interpretation: Of all actual negatives, how many were correctly identified? Measures ability to avoid false alarms. *When to use:* Use when it's important to correctly identify negative instances (e.g., confirming a patient is healthy, airport security screening where correctly identifying non-threatening items is crucial).

Composite Metrics:

- **F1 Score** $= \frac{2\cdot\text{Precision}\cdot\text{Recall}}{\text{Precision}+\text{Recall}} = \frac{2TP}{2TP+FP+FN}$
  Interpretation: Harmonic mean of precision and recall. Balances both metrics equally. *When to use:* Use when you need a balance between Precision and Recall, especially when dealing with imbalanced classes where accuracy can be misleading.
- **F$_\beta$ Score** $= (1+\beta^2)\cdot\frac{\text{Precision}\cdot\text{Recall}}{\beta^2\cdot\text{Precision}+\text{Recall}}$
  Interpretation: Weighted harmonic mean. $\beta > 1$ emphasizes recall, $\beta < 1$ emphasizes precision. *When to use:* Use when one metric (Precision or Recall) is more important than the other. For example, if recall is twice as important as precision, set $\beta = 2$. If precision is twice as important, set $\beta = 0.5$.

9 NumPy Essentials

9.1 Broadcasting

Overview: Mechanism that allows NumPy to perform arithmetic operations on arrays of different shapes without explicit replication of data.

Rules:

- Arrays with fewer dimensions are padded with ones on the left.
- Arrays with size 1 along a dimension are stretched to match the other array's size.
- If dimensions are incompatible, broadcasting fails.

Example:

- Array (3, 1) + Scalar: Scalar is broadcast to (3, 1).
- Array (3, 2) + Array (1, 2): Second array is broadcast to (3, 2).
- Manual broadcasting with `np.newaxis`: Reshape to add dimension, e.g., `arr[:, np.newaxis]`.

9.2 Common Functions

- **np.sort(arr)**: Returns a sorted copy of the array.
- **np.argsort(arr)**: Returns indices that would sort the array.
- **np.max(arr)**: Returns the maximum value in the array.
- **np.argmax(arr)**: Returns the index of the maximum value.
- **np.sqrt(arr)**: Computes the square root of each element.
- **np.exp(arr)**: Computes the exponential of each element.
- **np.square(arr)**: Squares each element.
- **np.sum(arr, axis=...)**: Sums elements along the specified axis. Without axis, sums all elements.
- **np.any(arr)**: Returns True if any element is True (useful for boolean arrays).
- **np.array[..][boolean]**: Boolean indexing to filter array based on condition.
- **np.array([... for ... in ...])**: Array creation via list comprehension, e.g., `np.array([i**2 for i in range(5)])`.