Module III
Performance Metrics

Module III: Performance Metrics 6 L

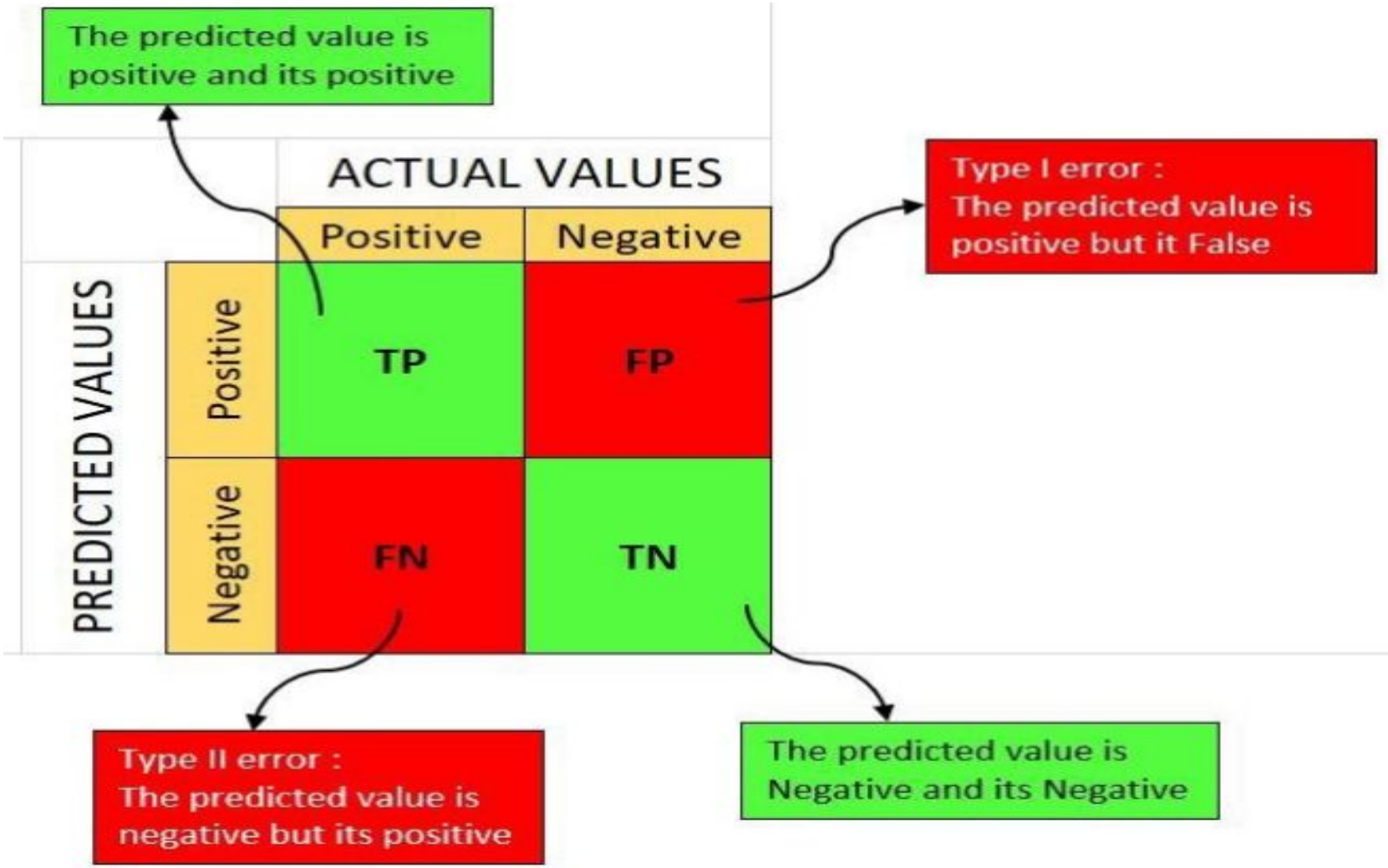
Model Evaluation and Selection: Performance metrics: Confusion matrix, Accuracy, Precision, Recall, F1 Score, Sensitivity, Specificity, Cross-validation and hyperparameter tuning.

Confusion Matrix

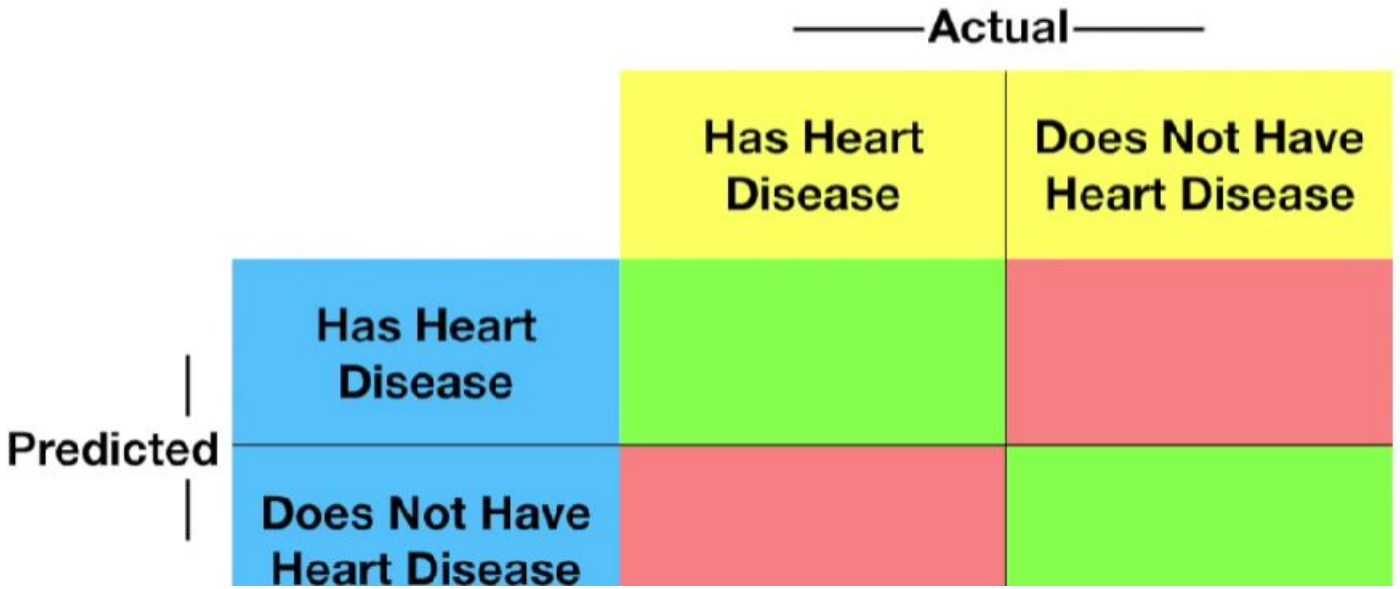
- **Confusion matrix** is a simple table used to measure how well a classification model is performing.
- It compares the predictions made by the model with the actual results and shows where the model was right or wrong.
- This helps you understand where the model is making mistakes so you can improve it.
- It breaks down the predictions into four categories:

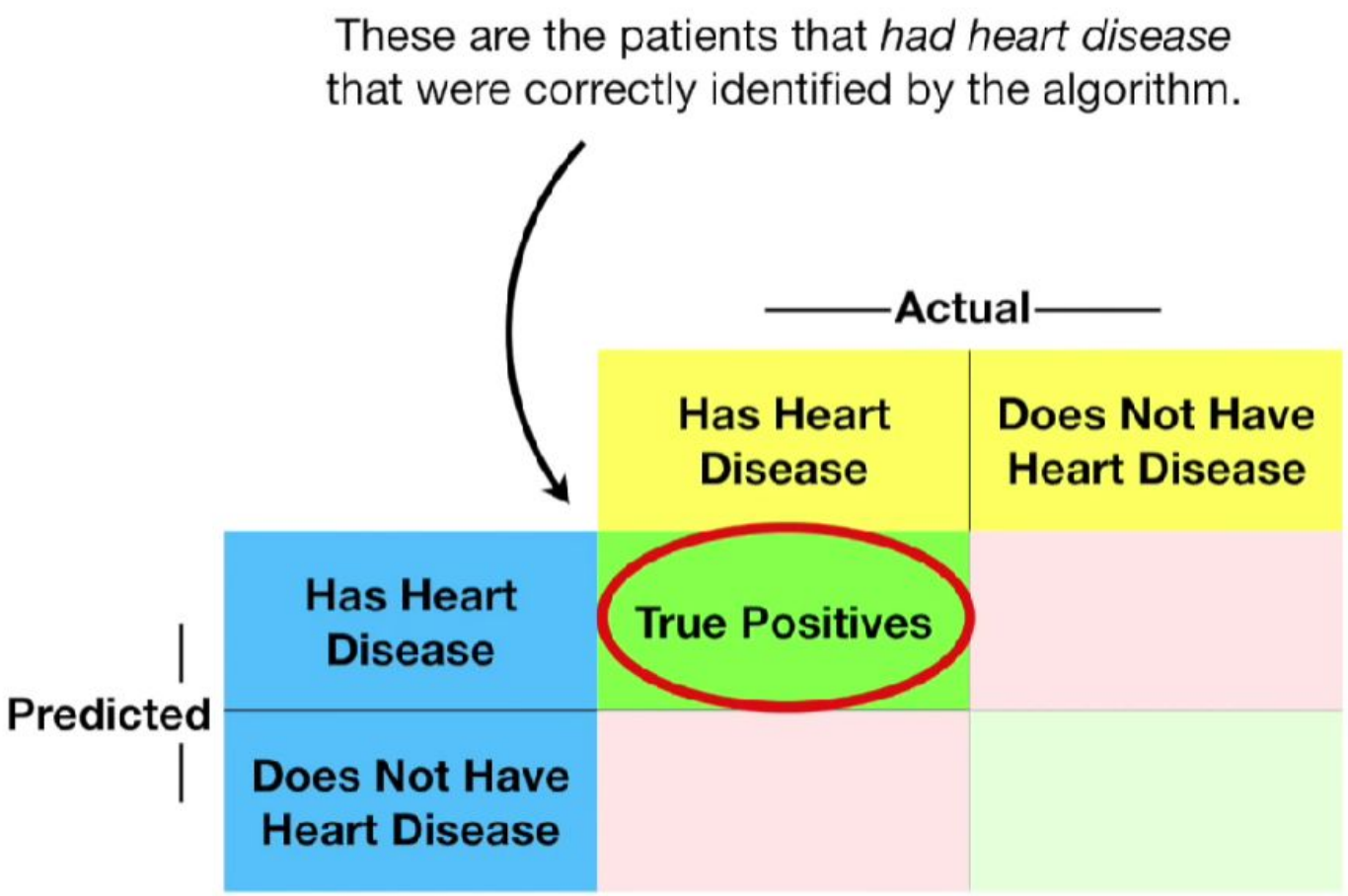
- **True Positive (TP): The model correctly predicted a positive outcome** i.e the actual outcome was positive.
- **True Negative (TN): The model correctly predicted a negative outcome** i.e the actual outcome was negative.
- **False Positive (FP): The model incorrectly predicted a positive outcome** i.e the actual outcome was negative. It is also known as a Type I error.
- **False Negative (FN): The model incorrectly predicted a negative outcome** i.e the actual outcome was positive. It is also known as a Type II error.

Confusion Matrix

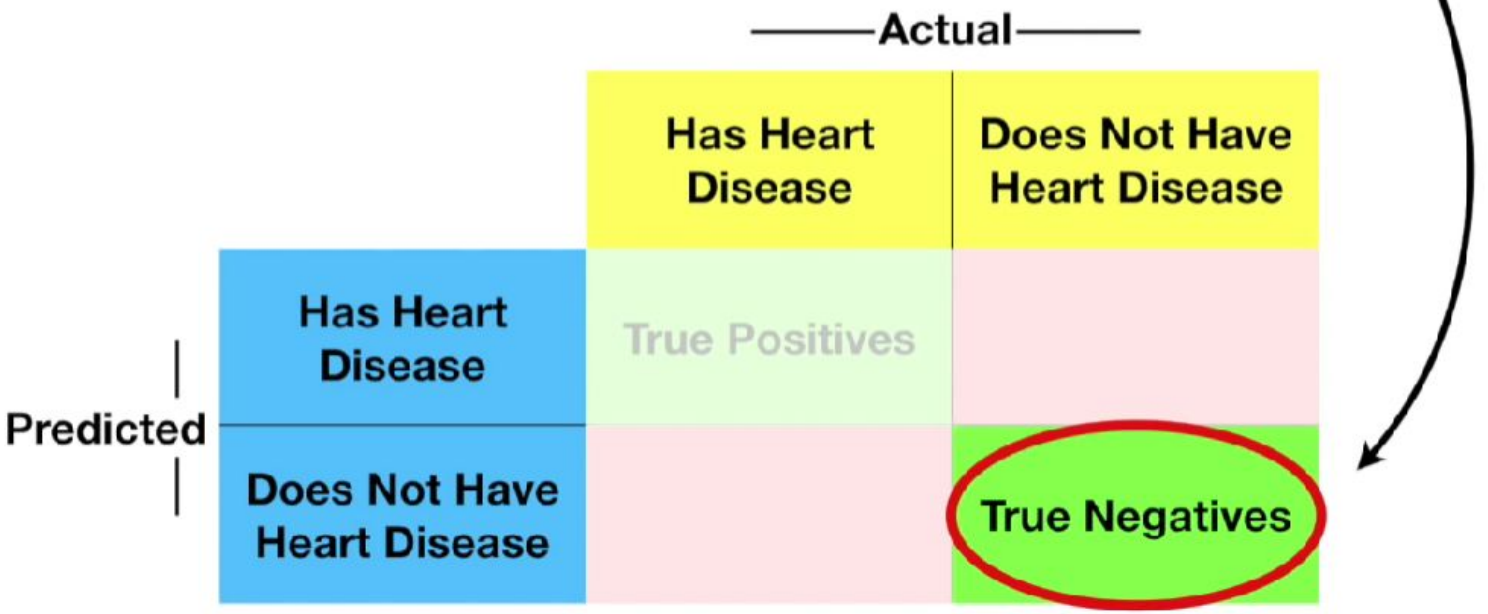


Example

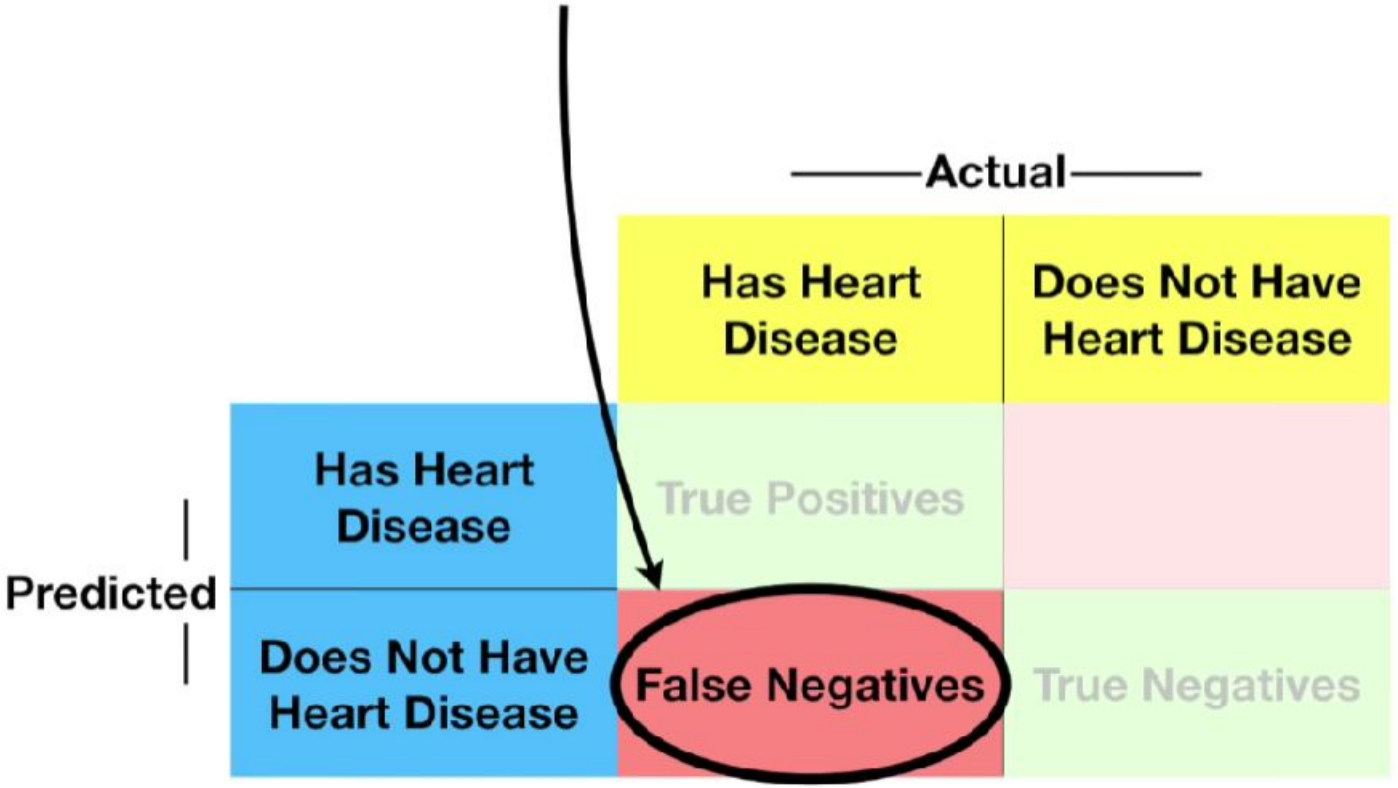

Actual
Has Heart Disease
Does Not Have Heart Disease
Predicted
Has Heart Disease
Does Not Have Heart Disease


These are the patients that *had heart disease* that were correctly identified by the algorithm.
Actual
Has Heart Disease
Does Not Have Heart Disease
Predicted
Has Heart Disease
True Positives
Does Not Have Heart Disease

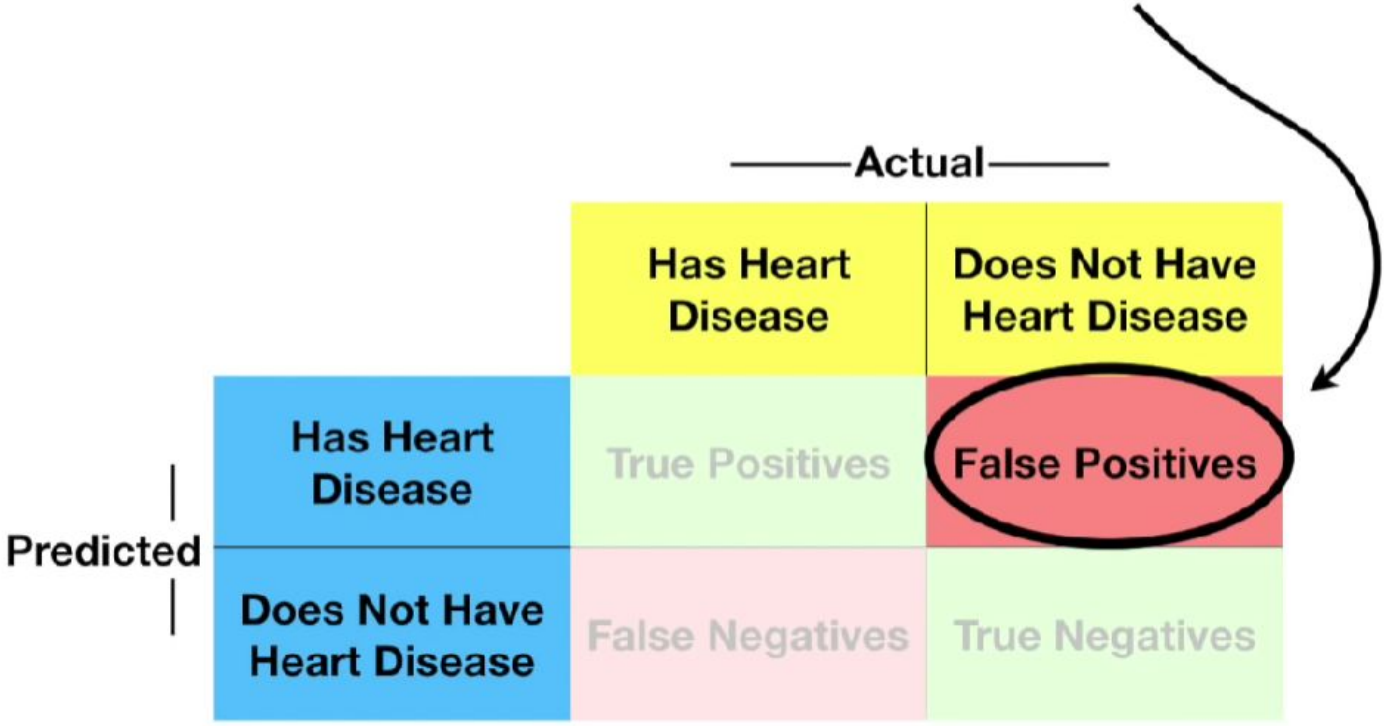
These are the patients that *did not have heart disease* that were correctly identified by the algorithm.



False Negatives are when a patient has heart disease, but the algorithm said they didn't.



False Positives are patients that do not have heart disease, but the algorithm says they do.

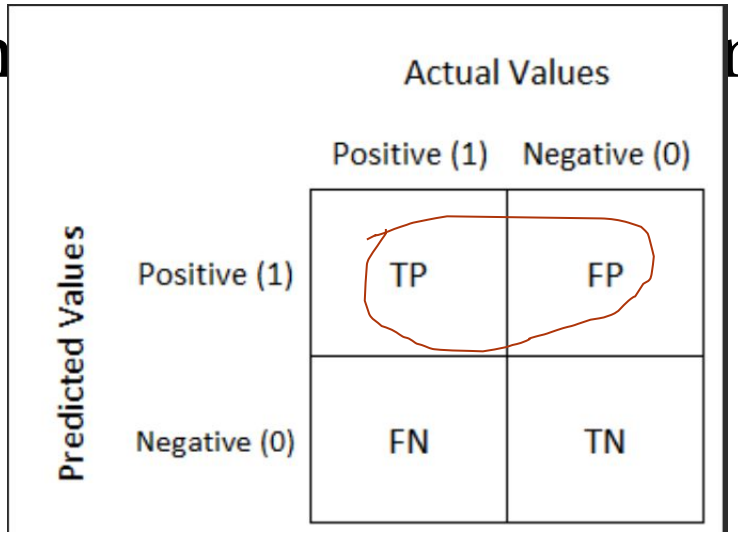


Precision and Recall

- Two metrics used for measuring the performance of a retrieval system are precision and recall.
- Precision measures the number of correct instances retrieved divided by all retrieved instances.
- Recall measures the number of correct instances retrieved divided by all correct instances.

PRECISION

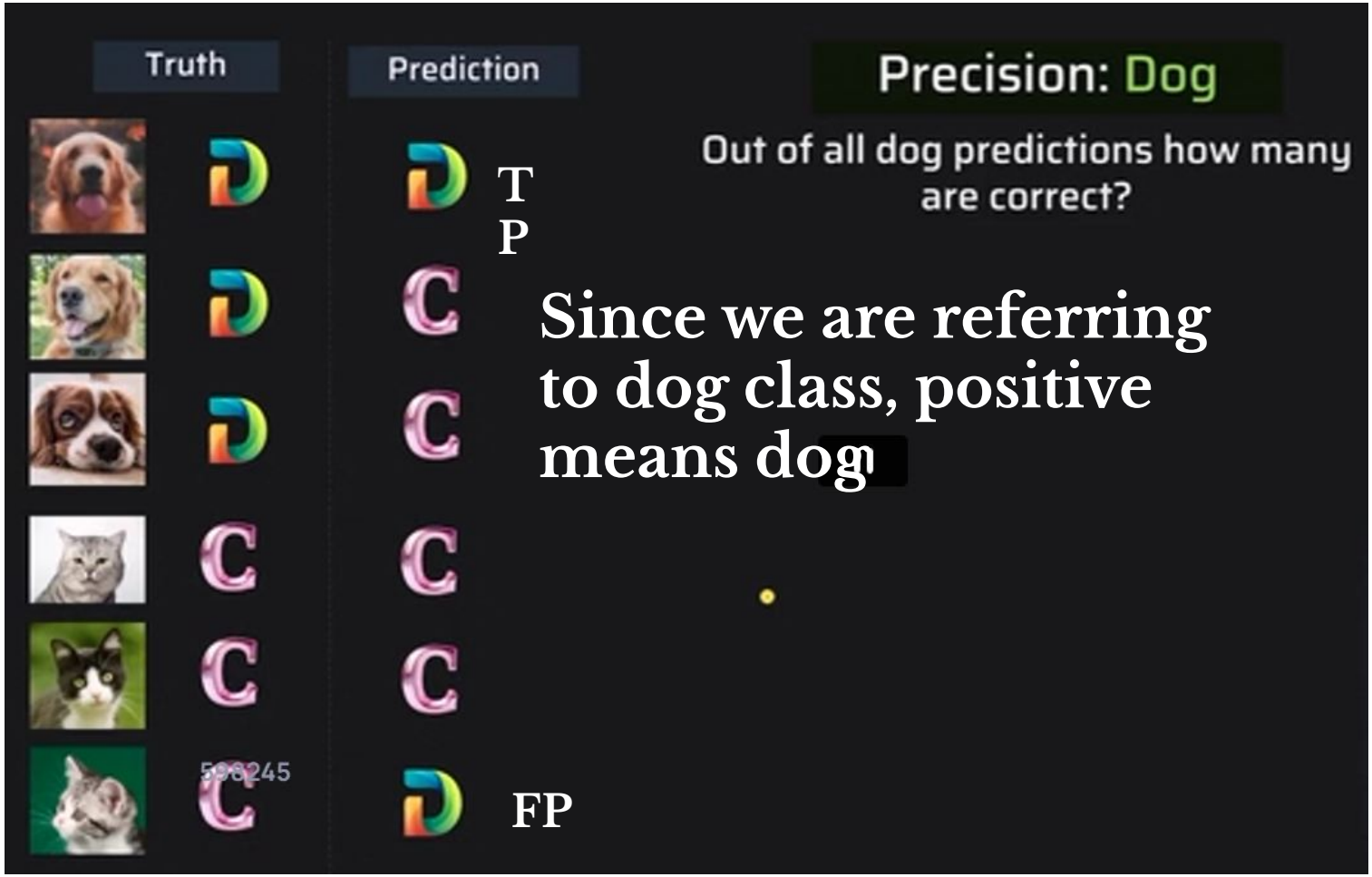
- Precision is the fraction of relevant instances amon nces.



$$\text{Precision} = \frac{True\ Positive}{True\ Positive + False\ Positive}$$

$$= \frac{True\ Positive}{Total\ Predicted\ Positive}$$

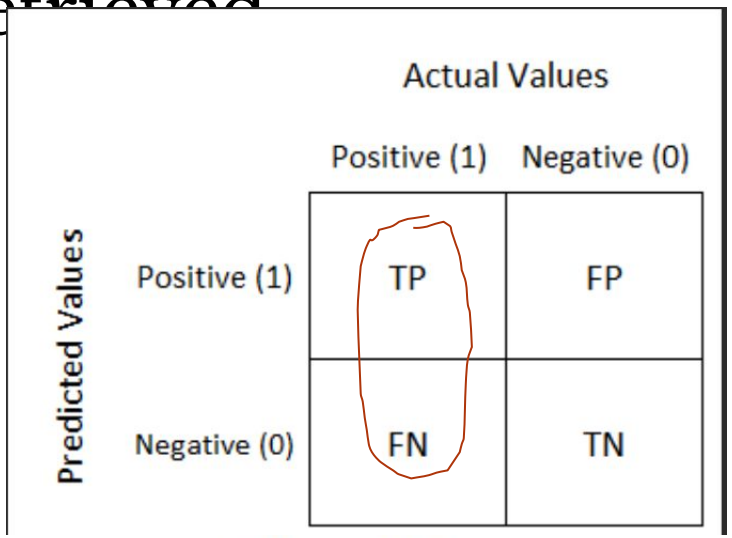
Keep prediction as base



Out of 2 dog prediction, 1 is correct so Precision=1/2=50%

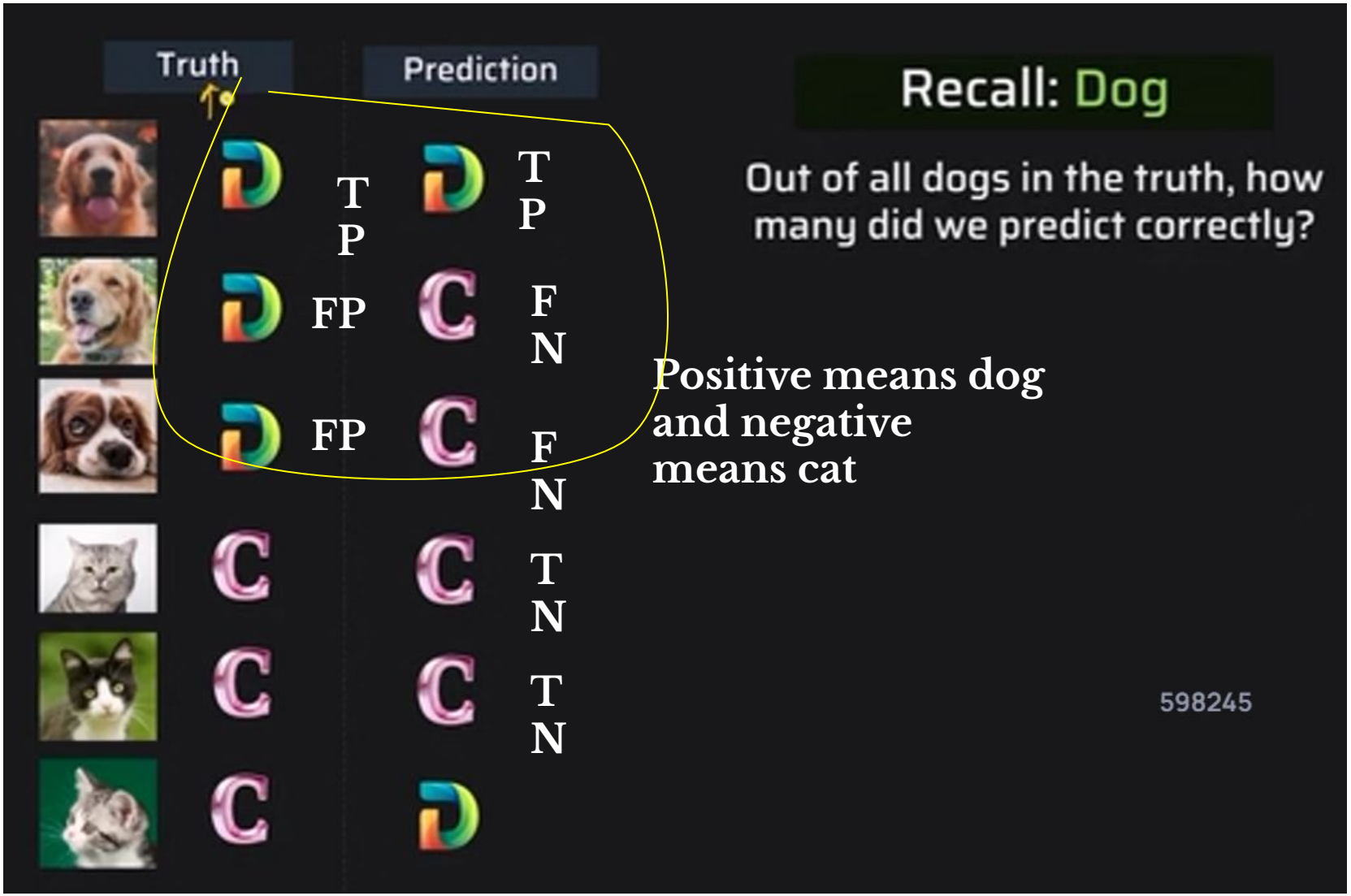
RECALL

- Recall is the fraction of relevant instances that were retrieved

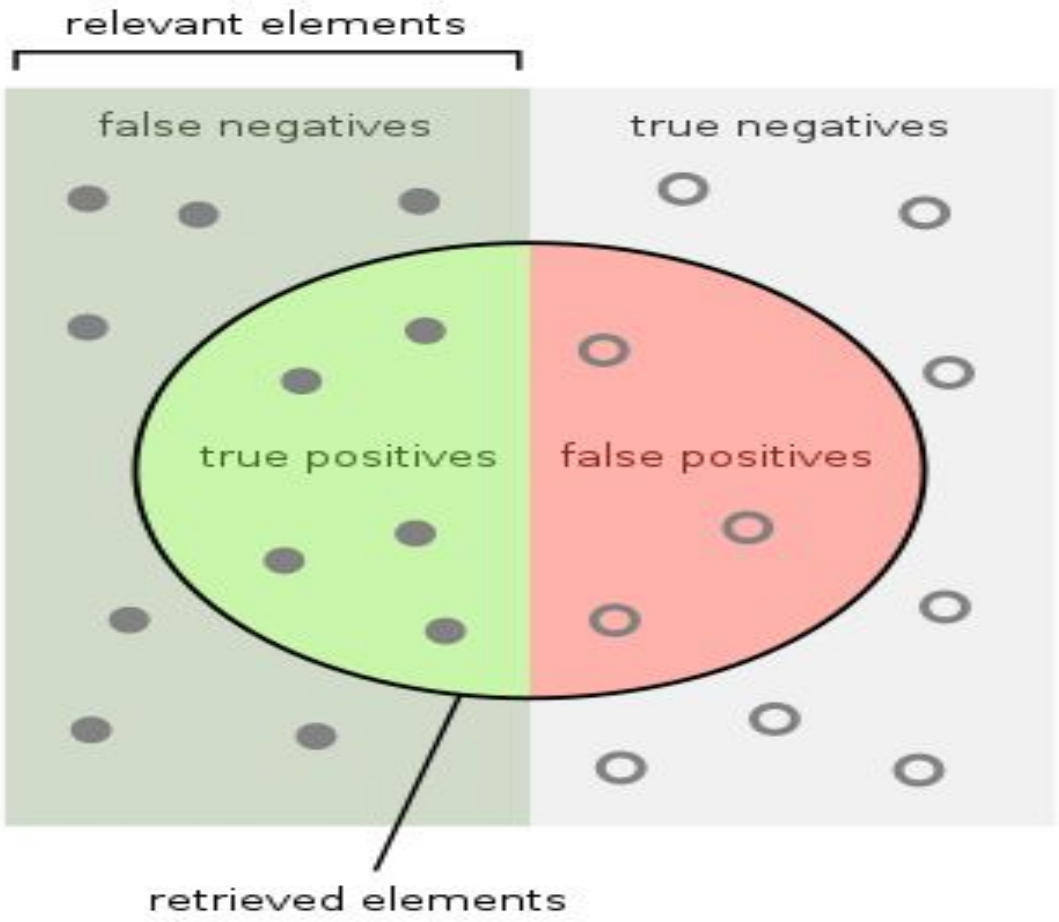


$$\text{Recall} = \frac{True\ Positive}{True\ Positive + False\ Negative}$$

$$= \frac{True\ Positive}{Total\ Actual\ Positive}$$



- Keep truth as a base
- 3 dogs in truth, out of 3 dogs 1 right so 1/3



How many retrieved items are relevant?

$$\text{Precision} = \frac{\text{green}}{\text{green + red}}$$

How many relevant items are retrieved?

$$\text{Recall} = \frac{\text{green}}{\text{green + dark green}}$$

Find!!!

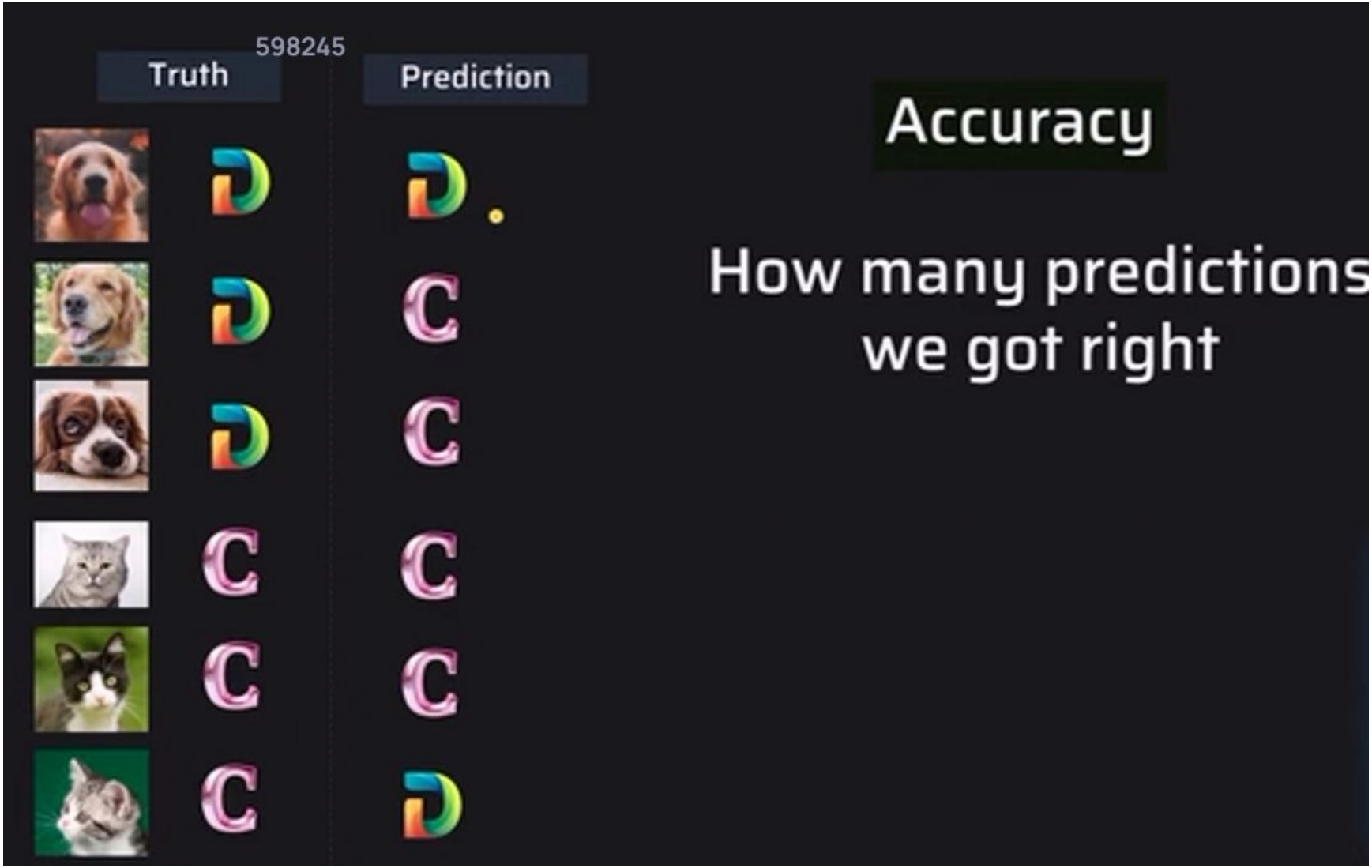
- Precison(cat)
- Recall(cat)

Accuracy

- Accuracy represents the number of correctly classified data instances over the total number of data instances.

$$Accuracy = \frac{TN + TP}{TN + FP + TP + FN}$$

- Accuracy may not be a good measure if the dataset is not balanced (both negative and positive classes have different number of data instances).



- Accuracy=3/6=50%

F1 score

- The F-score is defined as the weighted average of both precision and recall.(harmonic mean of precision and recall)
- Seeks recision and Recall. $F1 = 2 \times \frac{Precision * Recall}{Precision + Recall}$
- F1 Score might be a better measure to use if we need to seek a balance between Precision and Recall AND there is an uneven class distribution (large number of

Sensitivity: True Positive Rate

The true positive rate, also called *sensitivity*, is the fraction of correct predictions with respect to all points in the positive class, that is, it is simply the recall for the positive class

$$TPR = recall_P = \frac{TP}{TP + FN}$$

Specificity: True Negative Rate

The true negative rate, also called *specificity*, is simply the recall for the negative class:

$$TNR = specificity = recall_N = \frac{TN}{FP + TN}$$

False Negative Rate

The false negative rate is defined as

$$FNR = \frac{FN}{TP + FN} = 1 - sensitivity$$

False Positive Rate

The false positive rate is defined as

$$FPR = \frac{FP}{FP + TN} = 1 - specificity$$

AREA UNDER THE ROC CURVE (AUC)

Receiver Operating Characteristic curve

- The **ROC curve** is a graphical representation used to evaluate the performance of a **binary classification model**.
- For example, logistic regression, neural networks, and decision trees (and ensemble models based on decision trees) can be assessed using ROC curves.
- ROC curve commonly use the con $\mathrm{TPR} \stackrel{\text{def}}{=} \frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FN}}$ and $\mathrm{FPR} \stackrel{\text{def}}{=} \frac{\mathrm{FP}}{\mathrm{FP}+\mathrm{TN}}$. te(TPR) and fals is given as:

Axes:

- **X-axis → False Positive Rate (FPR)**

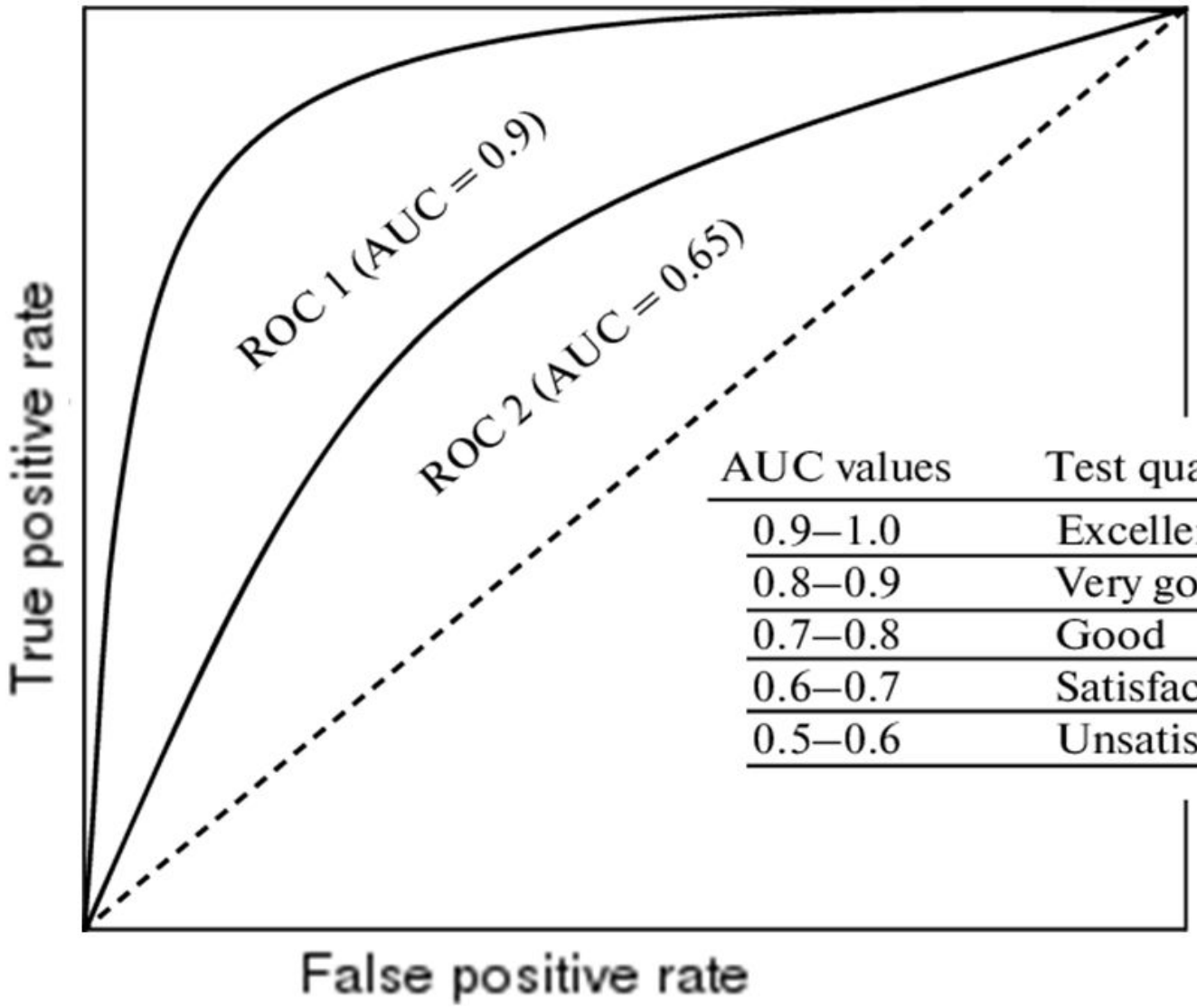
$$FPR = \frac{FP}{FP + TN}$$

(How many negatives were incorrectly classified as positive)

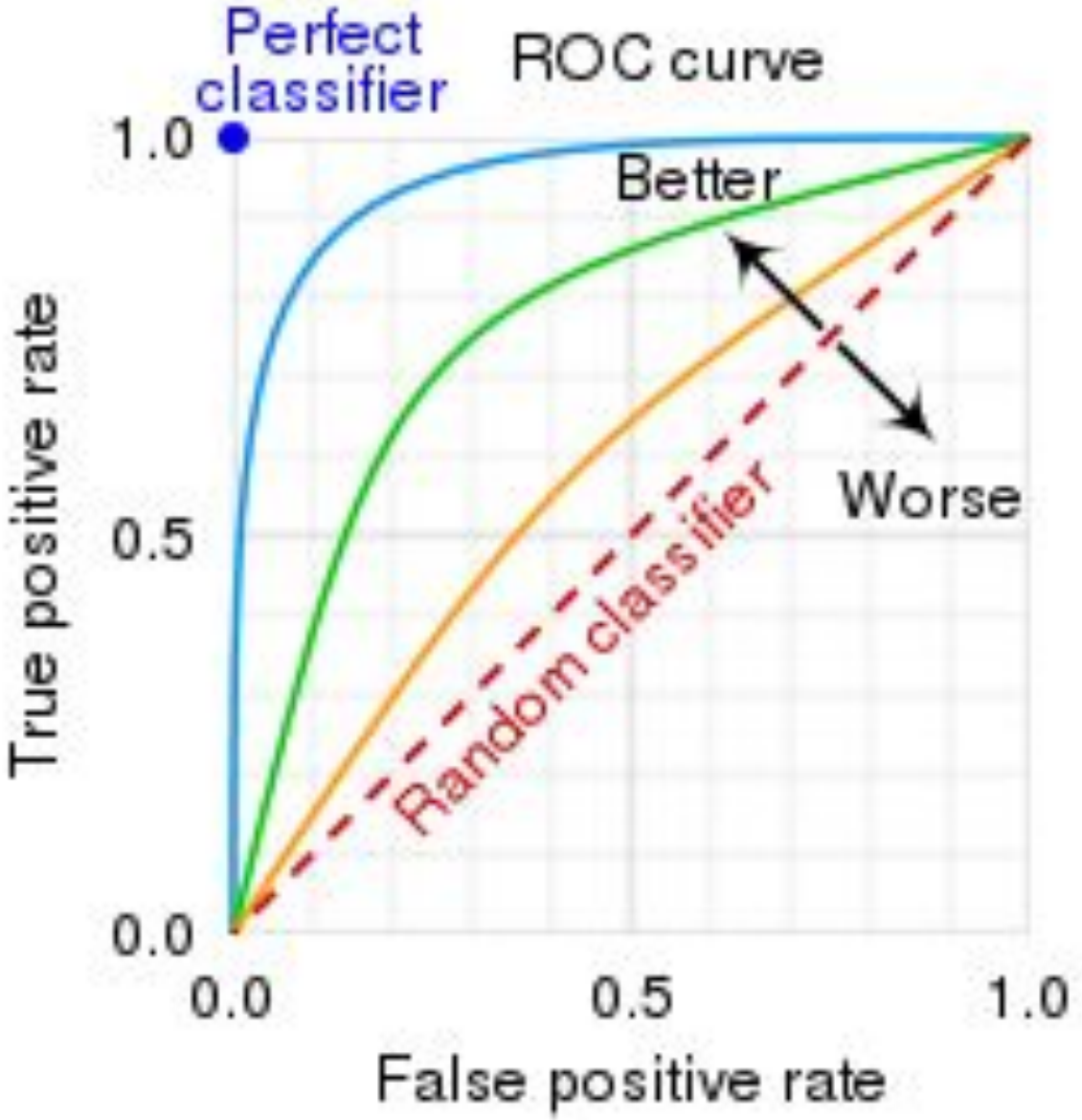
- **Y-axis → True Positive Rate (TPR) / Sensitivity / Recall**

$$TPR = \frac{TP}{TP + FN}$$

(How many actual positives were correctly classified)



AUC values	Test quality
0.9–1.0	Excellent
0.8–0.9	Very good
0.7–0.8	Good
0.6–0.7	Satisfactory
0.5–0.6	Unsatisfactory


Perfect classifier
ROC curve
1.0
Better
0.5
Worse
Random classifier
True positive rate
0.0
0.0
0.5
1.0
False positive rate

Explanation

- **Red Diagonal Line ("Random Classifier")** This represents a model that makes predictions with no learned pattern—just random guessing. Its AUC is 0.5, meaning it performs no better than chance.
- **Orange Curve ("Worse")** This model performs poorly. It may misclassify more often than it gets right. Its ROC curve lies close to or even below the diagonal, indicating low predictive power.
- **Green Curve ("Better")** This model shows good discrimination between classes. Its curve bows toward the top-left corner, meaning it achieves a high true positive rate while keeping false positives low.
- **Blue Point ("Perfect Classifier")** This is the ideal scenario: 100% true positives and 0% false positives. In reality, this is rare, but it serves as

- The **closer your curve is to the top-left**, the better your model is at distinguishing between classes.
- The **Area Under the Curve (AUC)** quantifies this:
- **AUC = 1.0** → perfect
- **AUC > 0.9** → excellent
- **AUC ≈ 0.5** → random

- ROC curves are insensitive to class skew.
- ROC curves focus on **true positive rate (TPR)** and **false positive rate (FPR)**, which are both **normalized** by the actual number of positives and negatives in the dataset:
- TPR = TP / (TP + FN)
- FPR = FP / (FP + TN)
- Because these rates are **proportional**, the ROC curve doesn't change shape even if the dataset has a large imbalance between classes (e.g., 95% negatives and 5% positives). That's why ROC curves are considered **robust to class imbalance**.

Bootstrapping and Cross-validation

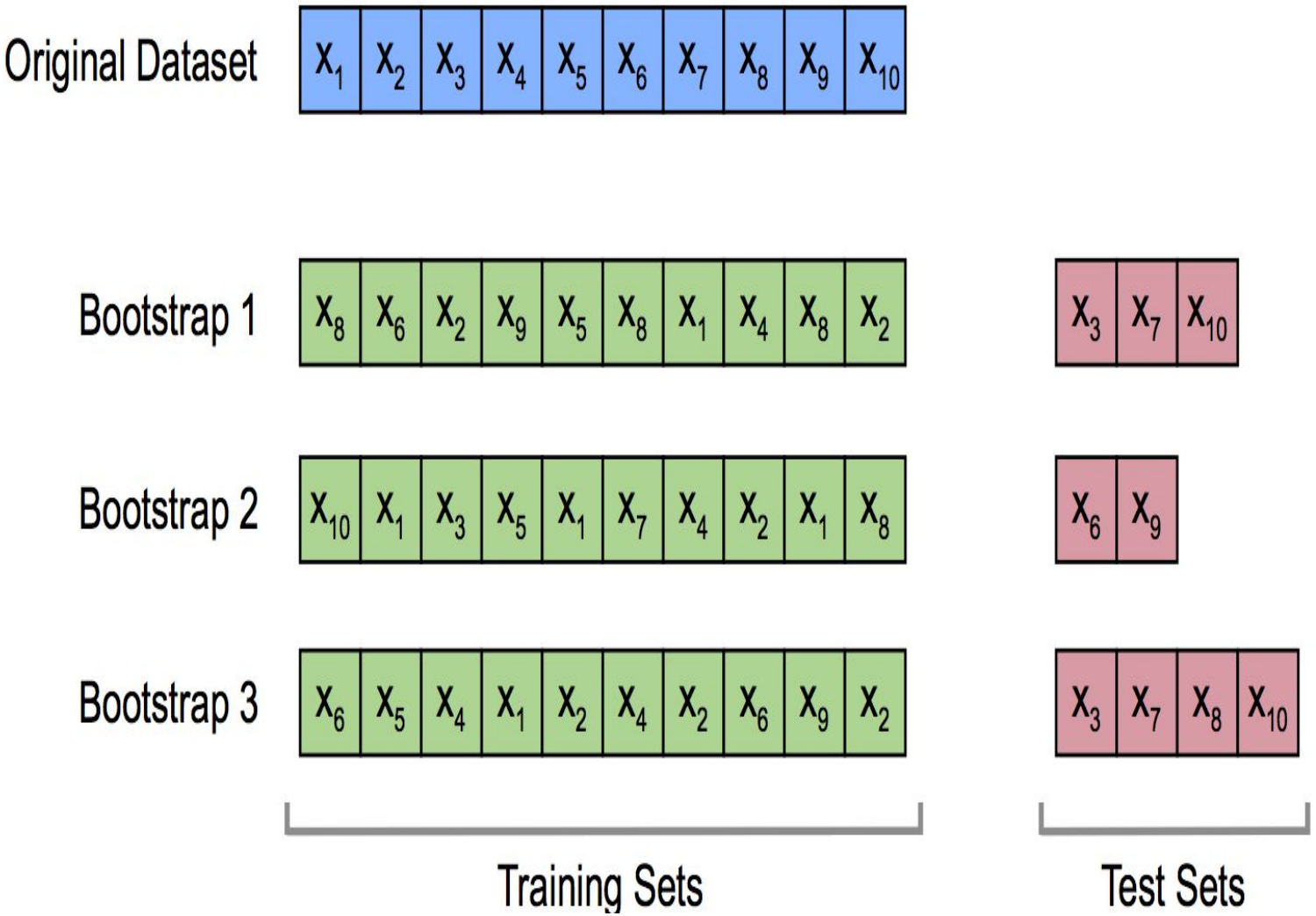
- **Bootstrapping** and **Cross-Validation** are both powerful techniques used to estimate how well a machine learning model will perform on unseen data—but they approach the problem in very different ways.

Bootstrapping

- **Bootstrapping**
- Bootstrapping is a **resampling method** that creates multiple datasets by **sampling with replacement** from the original data.

Key Features:

- Each bootstrap sample is the same size as the original dataset.
- Some data points may appear multiple times; others may be left out.
- The model is trained on the bootstrap sample and tested on the **out-of-bag** data (the points not selected).
- Useful for **small datasets** and estimating


Original Dataset
X1 X2 X3 X4 X5 X6 X7 X8 X9 X10
Bootstrap 1
X8 X6 X2 X9 X5 X8 X1 X4 X8 X2
X3 X7 X10
Bootstrap 2
X10 X1 X3 X5 X1 X7 X4 X2 X1 X8
X6 X9
Bootstrap 3
X6 X5 X4 X1 X2 X4 X2 X6 X9 X2
X3 X7 X8 X10
Training Sets
Test Sets

Pros:

- Flexible and easy to implement.
- Doesn't assume any specific data distribution.
- Good for estimating uncertainty in model predictions.

Cons:

- Can lead to **overfitting** if not used carefully.
- Less effective for very large datasets.

1. Input: Original dataset D with n samples, number of bootstrap iterations B
2. For i = 1 to B:

   a. Create a bootstrap sample Di by randomly sampling n points from D with replacement

   b. Train the model on Di

   c. Evaluate the model on:

   - Di (for training performance)

   - Out-of-bag samples (data not included in Di) for test performance

   d. Store the evaluation metric (e.g., accuracy, RMSE)

3. After B iterations:

   a. Aggregate the stored metrics

   b. Compute statistics like mean, standard deviation, confidence intervals

Cross-Validation

- Cross-validation splits the dataset into **K folds** and evaluates the model by training on **K–1 folds** and testing on the remaining fold. This is repeated **K times**, and the results are averaged.

Key Features:

- Sampling is **without replacement**.
- Each data point is used for both training and testing exactly once.
- Common variants: **K-Fold**, **Stratified K-Fold**, **Leave-One-Out (LOOCV)**.

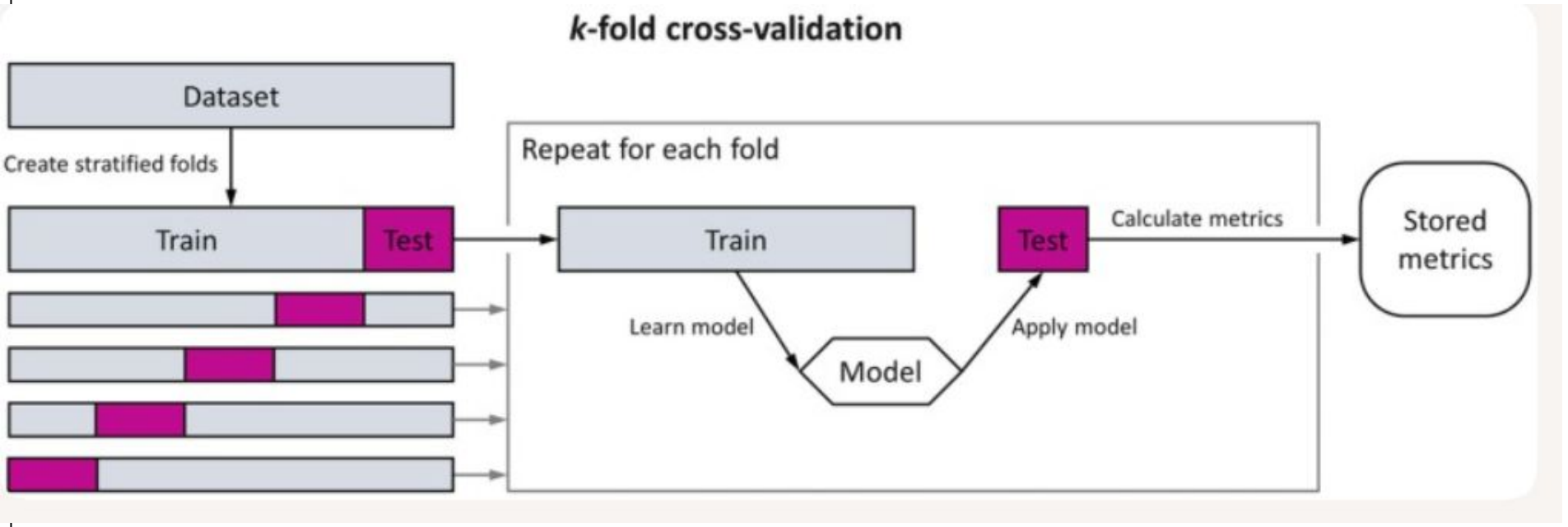
Pros:

- Provides a **robust estimate** of model performance.
- Helps prevent overfitting.
- Ideal for **model selection** and **hyperparameter tuning**.

Cons:

- Computationally expensive for large datasets.

K-fold Cross Validation



What Is K-Fold Cross Validation?

- It's a **resampling technique** used to evaluate the performance of a model by splitting the dataset into **K equal parts (folds)**. The model is trained on **K–1 folds** and tested on the remaining one. This process is repeated **K times**, with each fold used exactly once as the test set.

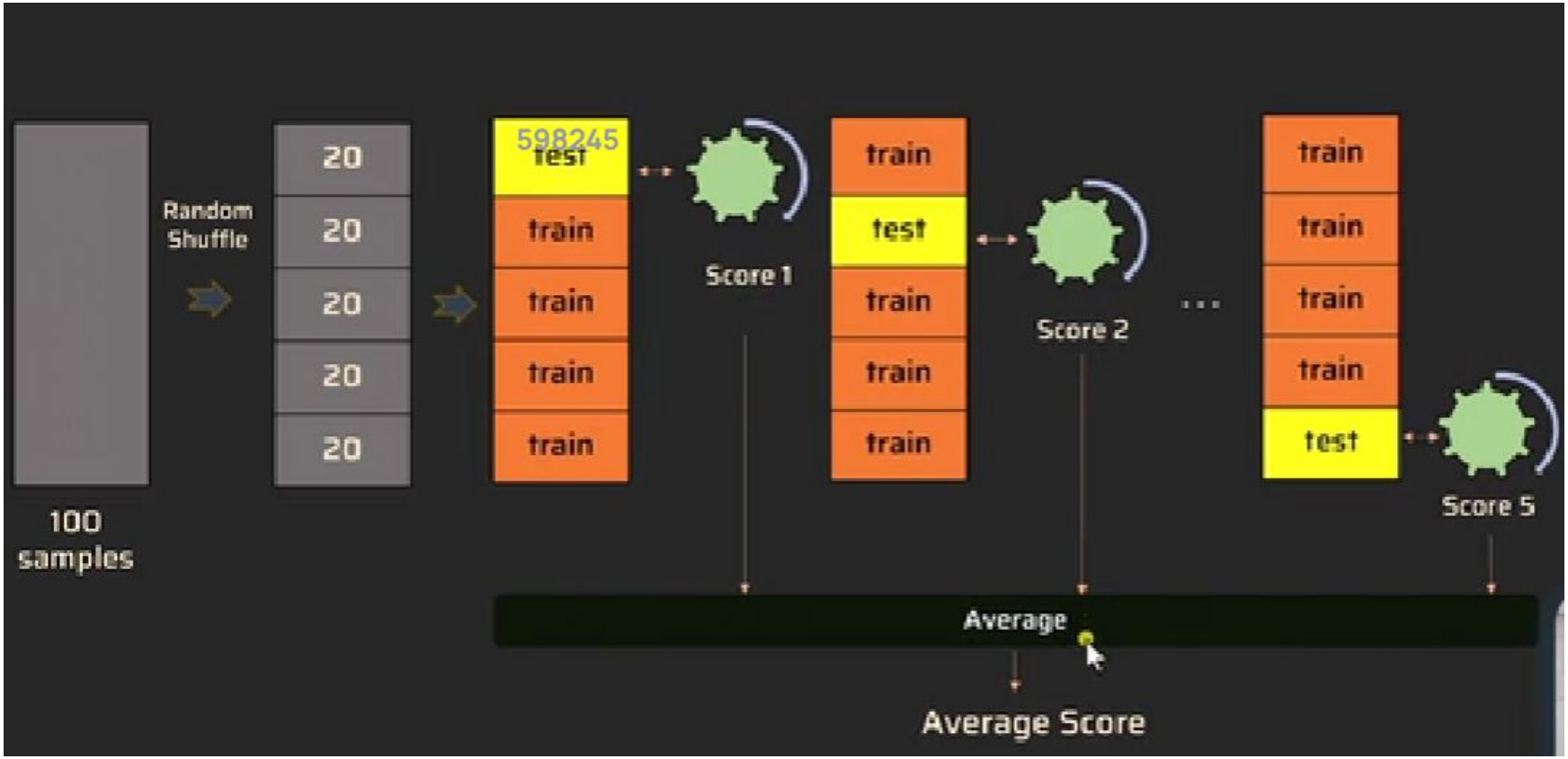
Working

- **Split the data** into K folds.
- **Train the model** on K–1 folds.
- **Test the model** on the remaining fold.
- **Repeat** steps 2–3 K times, each time with a different test fold.
- **Average the results** to get a robust estimate of model performance.

Advantages

- **Reduces overfitting** by testing the model on multiple data splits.
- **Maximizes data usage**, especially helpful when you have limited data.
- **Provides a more reliable performance estimate** than a single train-test split.

5 fold cross validation



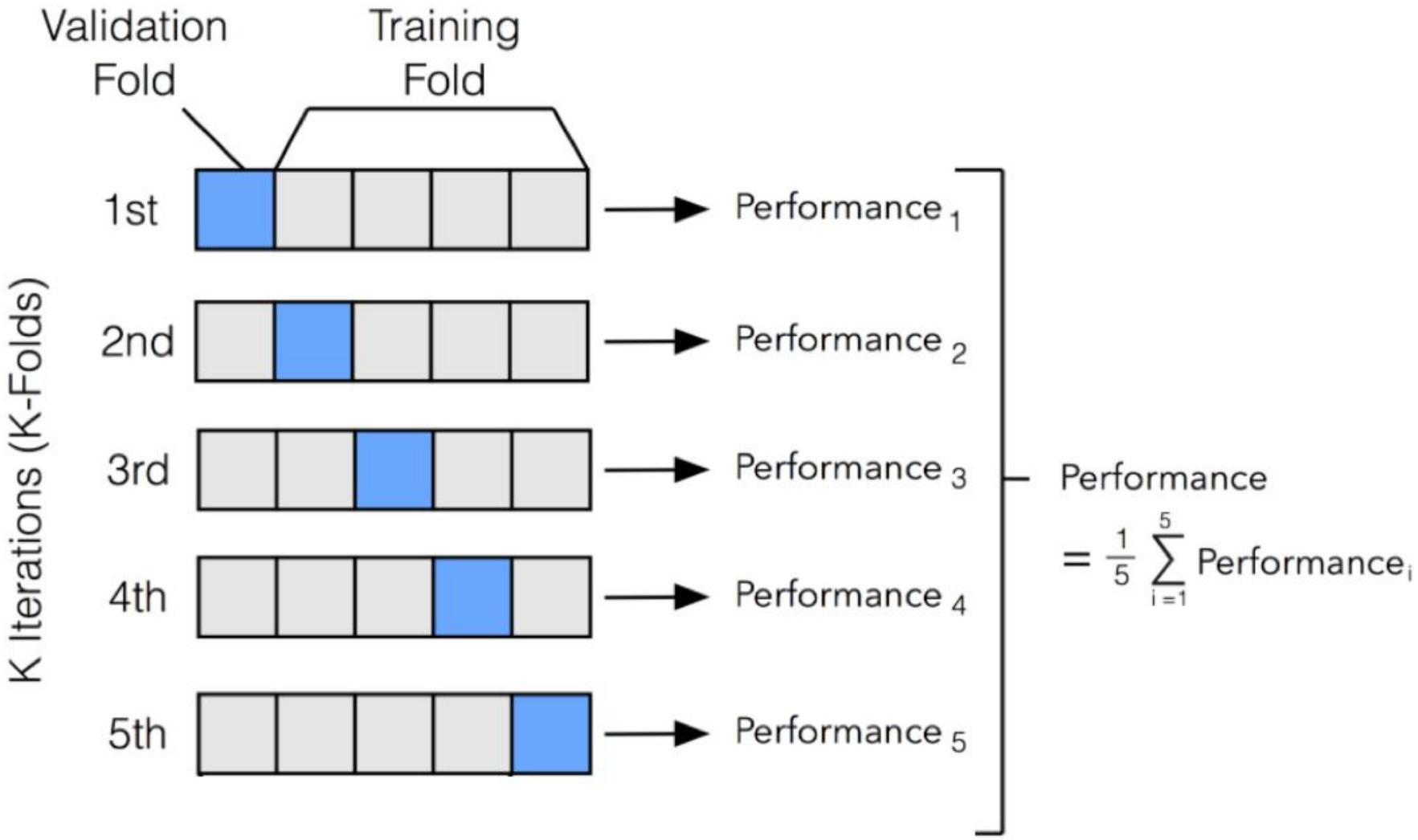
Algorithm

1. Input: Dataset D, number of folds K
2. Split D into K equal-sized folds: F1, F2, ..., FK

3. For each fold i from 1 to K:
    a. Use all folds except Fi to train the model
    b. Use Fi as the test set
    c. Train the model on training folds
    d. Evaluate the model on test fold Fi
    e. Store the evaluation score

4. Compute the average of the K evaluation scores
5. Output: Average score as the model's performance

K-fold cross validation

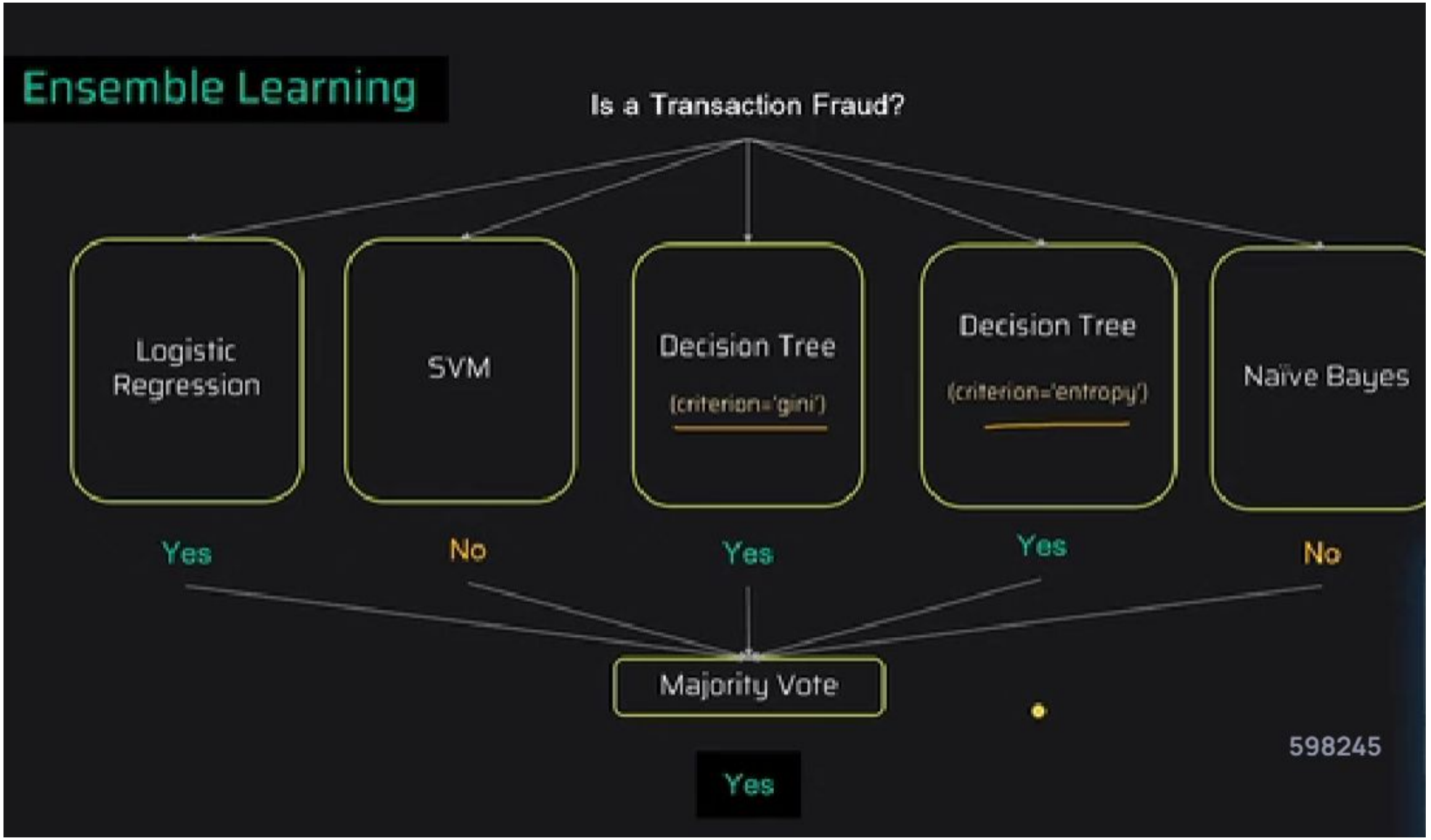


Validation Type	Description
Holdout Validation	Split data into training and testing sets (e.g., 70/30 or 80/20).
K-Fold Cross-Validation	Divide data into *k* subsets (folds); train on *k-1*, test on the remaining.
Stratified K-Fold	Like K-Fold, but preserves class distribution across folds.
Leave-One-Out (LOOCV)	Train on all data except one point; repeat for each point.
Leave-P-Out	Leave *p* samples out for testing; train on the rest.

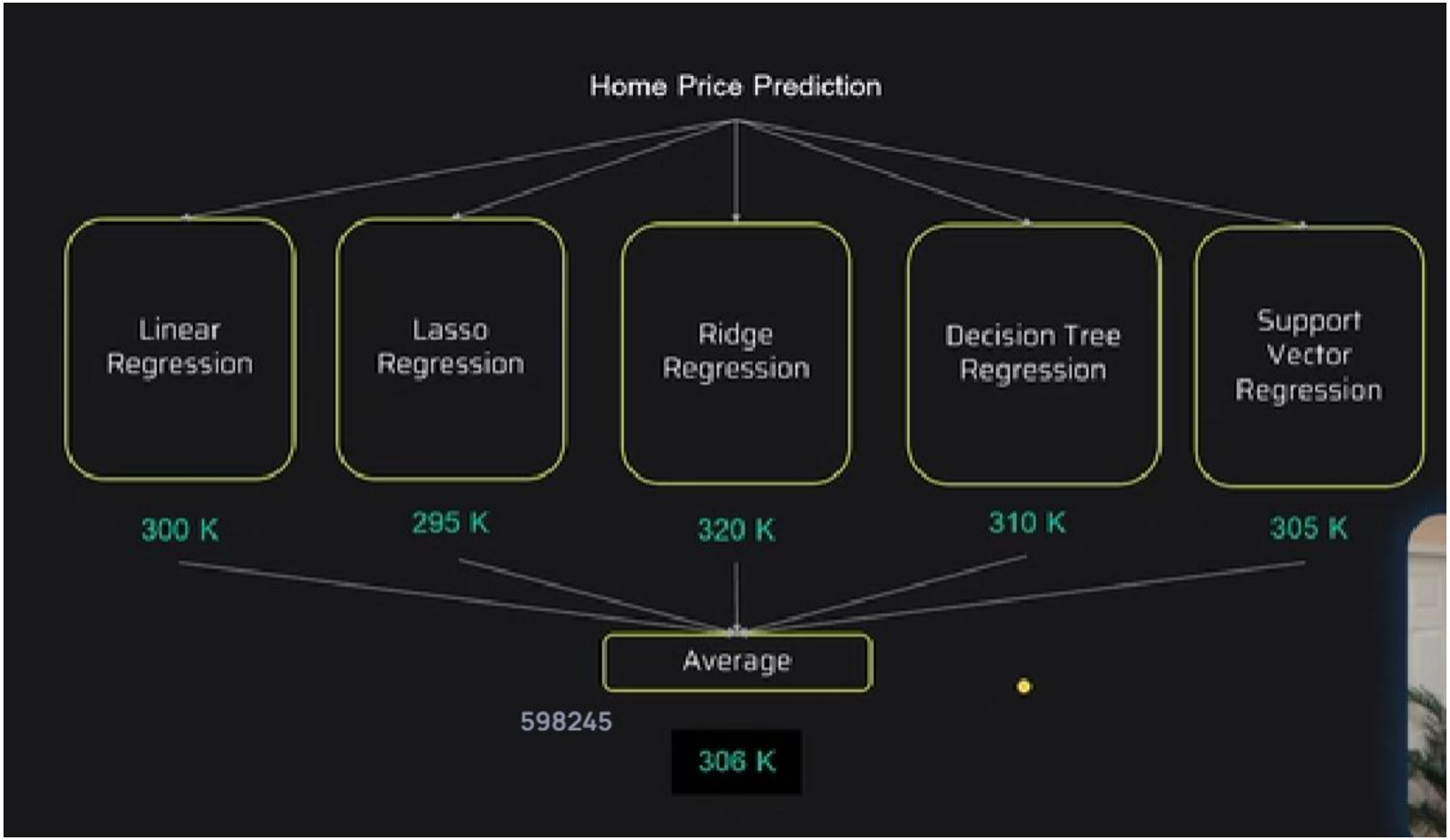
Ensemble Methods

- Ensemble methods combine multiple machine learning models (called base learners) to produce a model with better performance than any single model.
- High variance classifiers are inherently unstable, since they tend to overfit the data.
- On the other hand, high bias methods typically underfit the data, and usually have low variance.
- Ensemble methods create a *combined classifier using the output of multiple base classifiers, which are trained on different data subsets.*
- *Depending on how the training* sets are selected, and on the stability of the base classifiers,

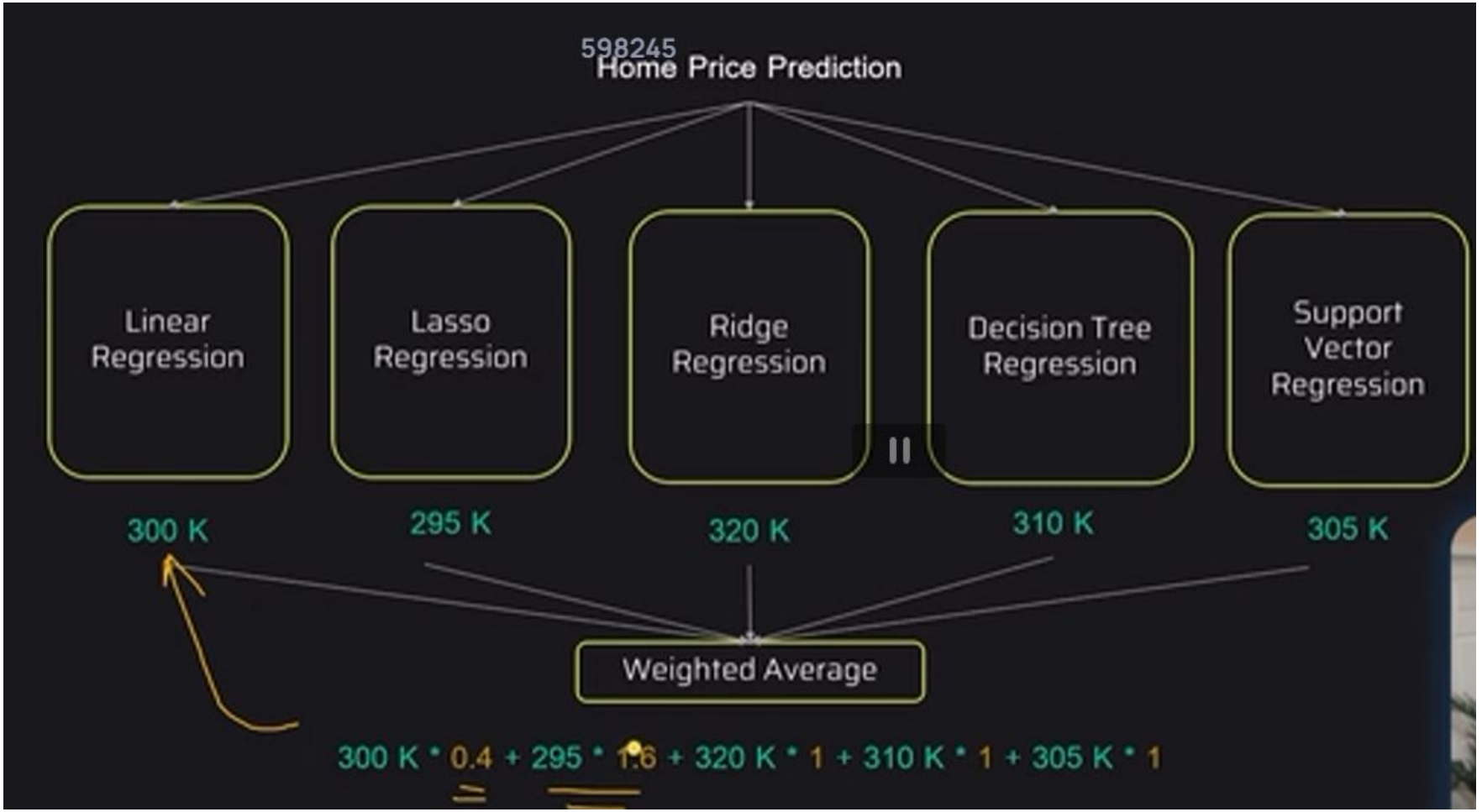
Example

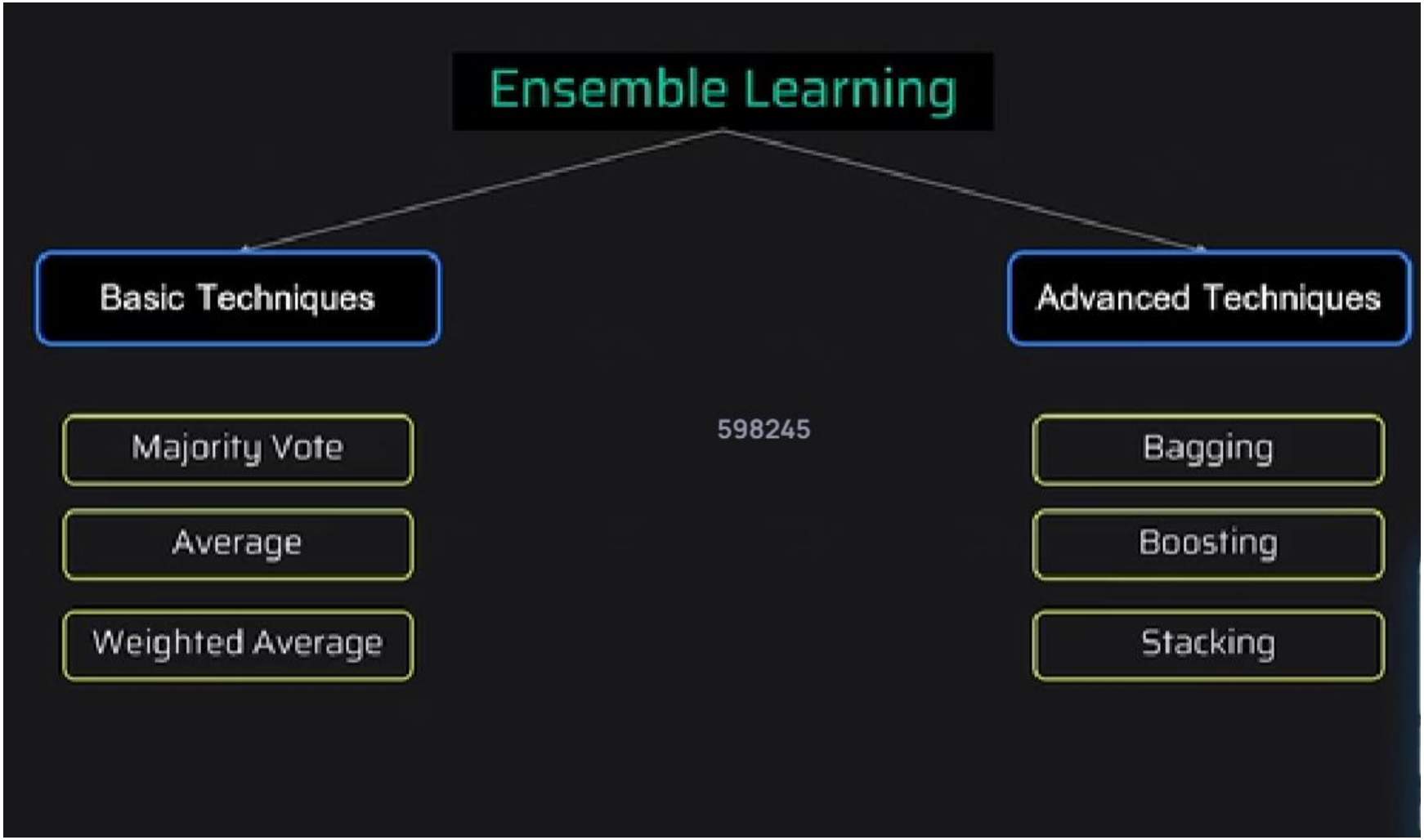

Ensemble Learning
Is a Transaction Fraud?
Logistic Regression
SVM
Decision Tree
(criterion='gini')
Decision Tree
(criterion='entropy')
Naïve Bayes
Yes
No
Yes
Yes
No
Majority Vote
Yes
598245

Example

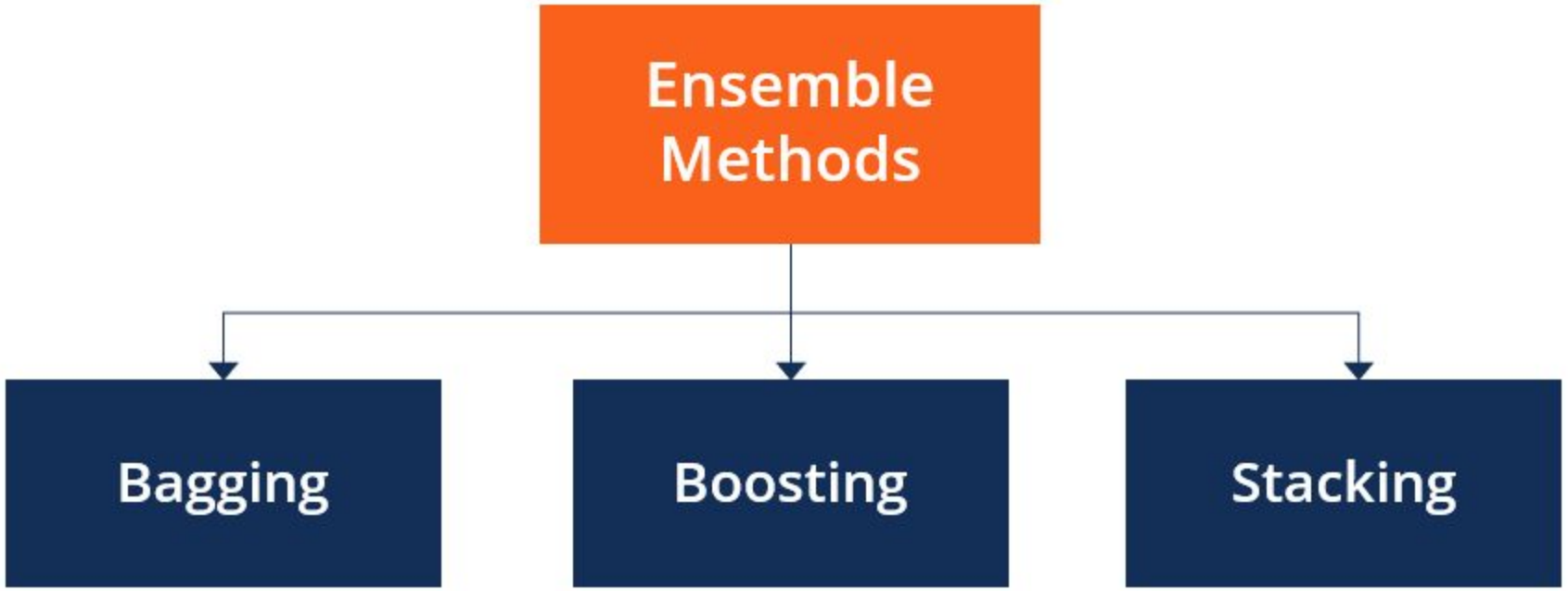

Home Price Prediction
Linear Regression
Lasso Regression
Ridge Regression
Decision Tree Regression
Support Vector Regression
300 K
295 K
320 K
310 K
305 K
Average
598245
306 K

Example


598245
Home Price Prediction
Linear Regression
Lasso Regression
Ridge Regression
Decision Tree Regression
Support Vector Regression
300 K
295 K
320 K
310 K
305 K
Weighted Average
300 K * 0.4 + 295 * 1.6 + 320 K * 1 + 310 K * 1 + 305 K * 1


Ensemble Learning
Basic Techniques
Advanced Techniques
Majority Vote
Average
Weighted Average
598245
Bagging
Boosting
Stacking

Ensemble methods



Bagging

- **Bagging** stands for **Bootstrap Aggregating**. It is an ensemble learning technique that improves the stability and accuracy of machine learning algorithms by combining multiple models.

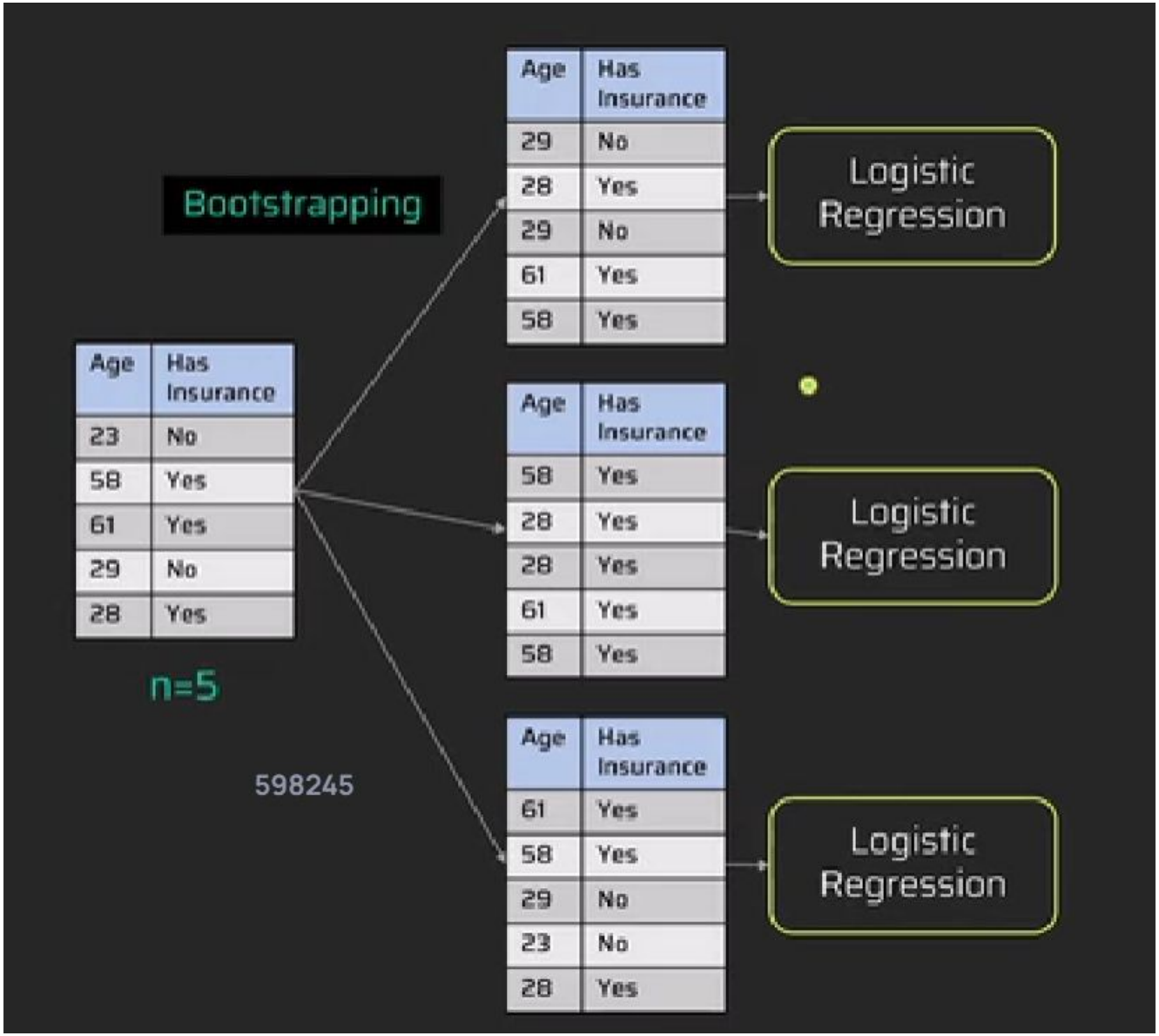
Main Idea

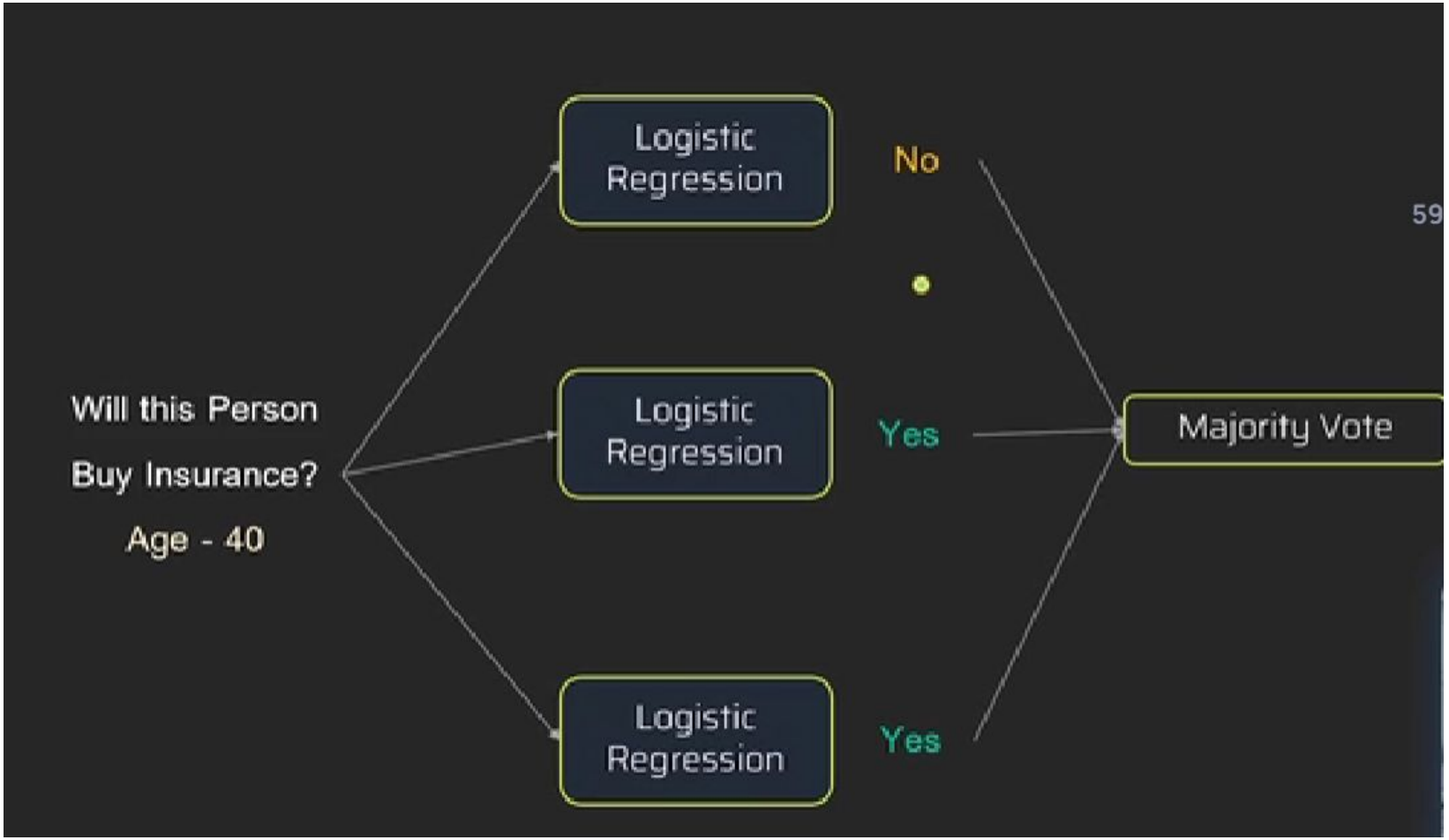
- Instead of using one model, we train multiple models on different subsets of the data.
- Each subset is created using bootstrapping (sampling with replacement).
- The results of these models are then combined (aggregated) to make the final prediction.

Steps in Bagging

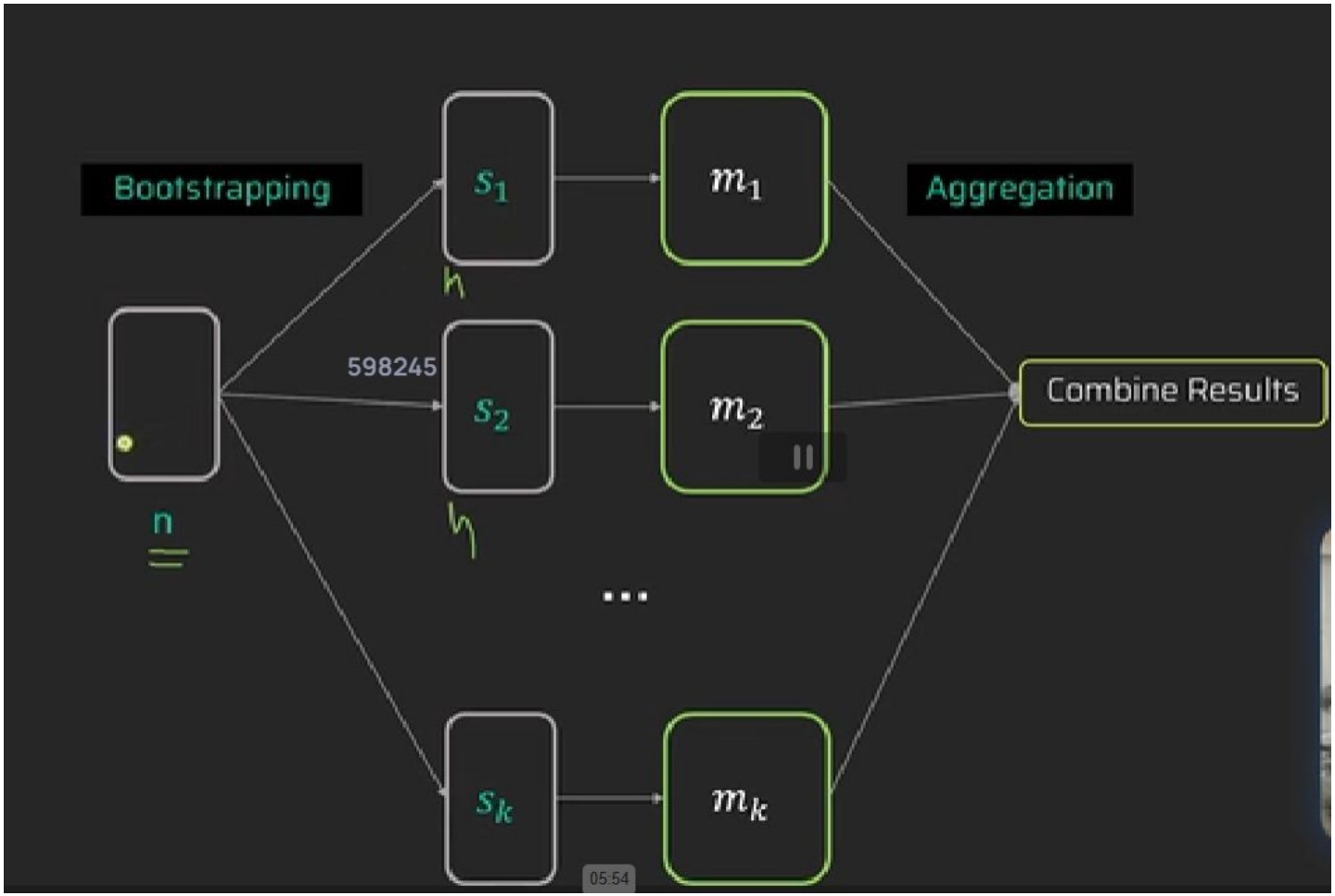
- **Bootstrapping (Data Sampling):**
  - From the original training dataset (size = N), create multiple new datasets (also size = N) by sampling with replacement.
  - Some samples may appear multiple times, some may not appear at all.
- Train Multiple Models:
  - Train a base learner on each bootstrapped dataset.
- Aggregate Results:
  - For classification → use majority voting among models.
  - For regression → use average of predictions.

Random sampling with replacement




Will this Person
Buy Insurance?
Age - 40
Logistic Regression
Logistic Regression
Logistic Regression
No
Yes
Yes
Majority Vote
59

Bagging-Bootstrap aggregation



Bagging in ensemble learning is a technique where multiple models are trained on random subsets of the training data and their predictions are aggregated to improve accuracy and reduce variance.

Each model in ensemble learning is called a **Base learner**.

Bagging is also known as **bootstrap aggregating**.

Benefit of bagging:

- Robust against Outliers
- Reduction in Variance
- A good way to handle high dimensionality
- Improved Accuracy

Boosting

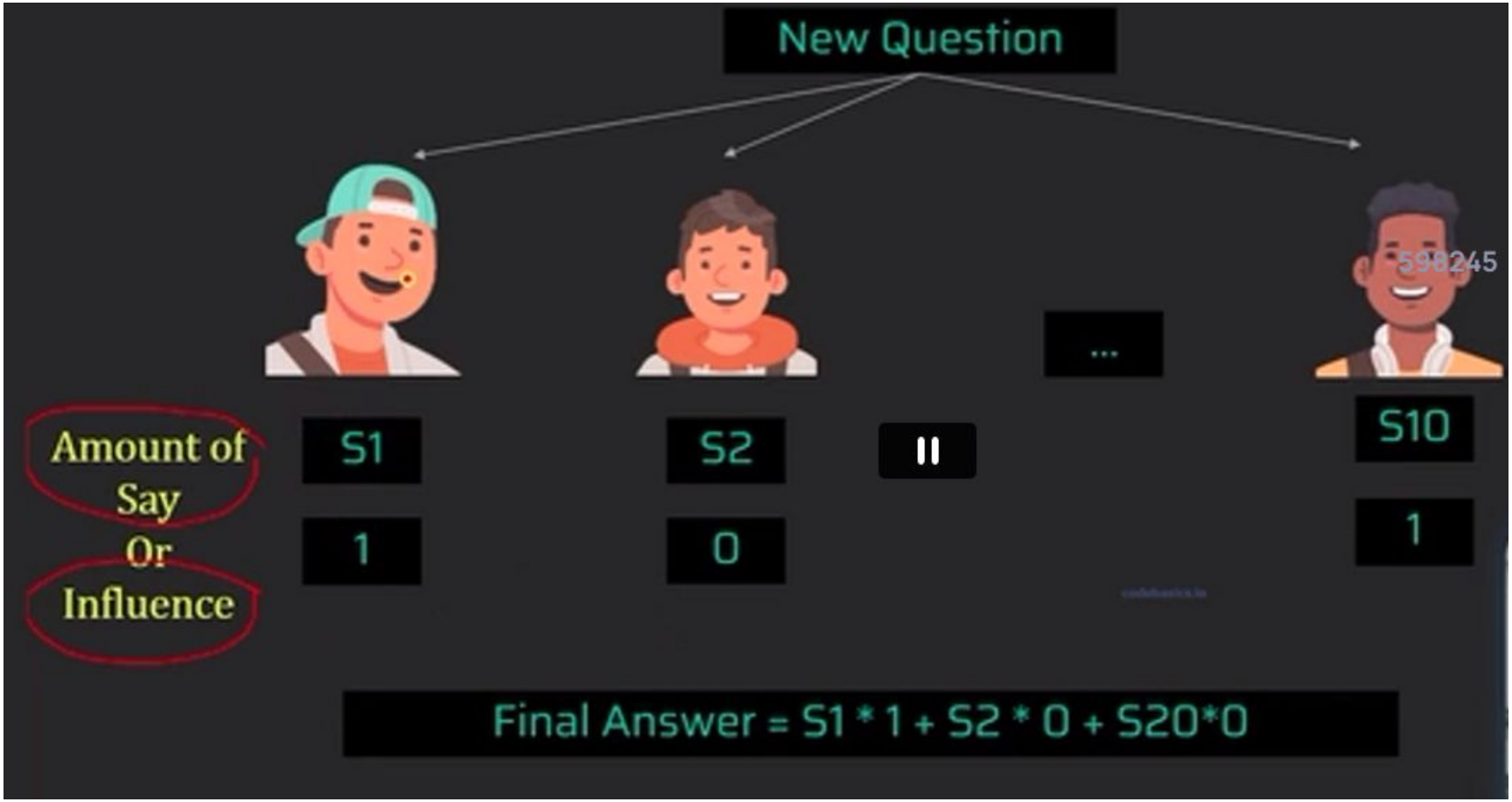
- *Bagging uses a parallel approach; models are trained in parallel and predictions are obtained in parallel.*
- *Boosting uses sequential approach, where the entire dataset is given to the first model, it will do predictions. We will check which all predictions are correct , which are wrong and based on that you will give that dataset with weight to the second model.*
- *In this, the records which generate wrong predictions for model 1 have higher weights. Model 2 will try to focus more on that. Because if model 2 answers incorrectly for those records for which model 1 predict incorrectly that leads to higher*

- Boosting is an **ensemble learning method** that combines multiple weak learners (models that perform just slightly better than random guessing) into a strong learner that makes highly accurate predictions2.
- **How It Works**
- **Sequential Training**: Models are trained one after another. Each new model focuses on the mistakes made by the previous ones.
- **Weight Adjustment**: Misclassified data points get more weight, so the next model pays extra attention to them.
- **Final Ensemble**: All models are combined—usually through weighted voting or averaging—to produce a final, strong prediction

Example: 20 students form a group

- They are preparing for a competition. 100 questions will be given to first students. The teacher compares his answers with actual answers.
- Student2 is also given same set of 100 questions and an information on the performance of student 1.
- Student 2 will put more focus on those questions wrongly answered by student 1. He will try to pay attention to these questions so that he doesn't get wrong answer.
- If student answers question1 right , he will get 1 mark and -1 for wrong answer.
- If student 2 answers question wrongly answered by student 1, he will get +2 else -2.

- Whenever weight is higher, if you answered correctly more points else less marks.
- Eventually all students get some scores.
- The person who scored more, have a more of **amount of say/influence**


New Question
...
S1
S2
II
S10
Amount of Say Or Influence
1
0
1
Final Answer = S1 * 1 + S2 * 0 + S20*0

Boosting in machine learning is a sequential process where multiple models, typically weak learners, are trained one after the other, with each model building upon the errors of its predecessor to improve overall accuracy and form a strong predictive model.

Different Boosting Techniques:

- AdaBoost
- Gradient Boost
- XGBoost (eXtreme Gradient Boosting)
- LightGBM
- CatBoost

Stacking

- Instead of relying on just one model, stacking combines several different models—each with its own strengths—and lets a final model (called a **meta-model**) decide how to best use their predictions.

Working

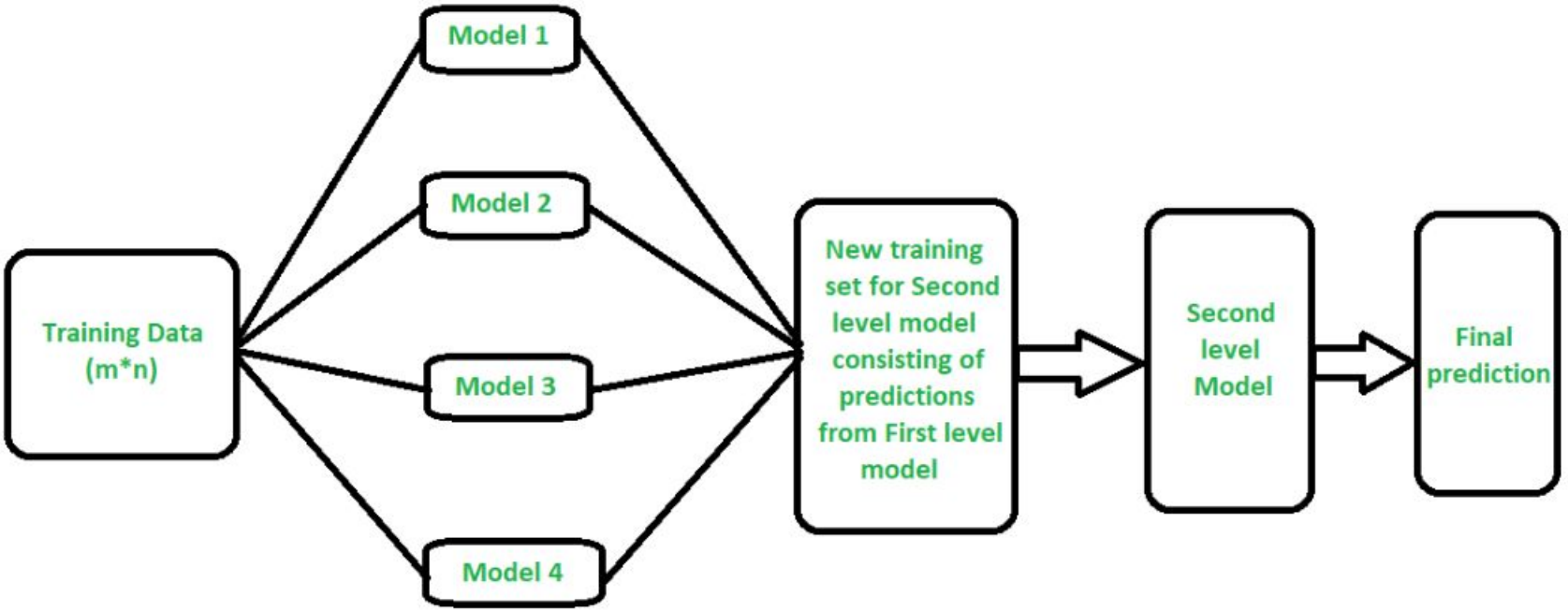
- Stacking is a **two-layer ensemble method**:
- **Level-0 (Base Models)**: These are your initial models—like decision trees, SVMs, or neural networks—that learn directly from the training data.
- **Level-1 (Meta-Model)**: This model doesn't see the raw data. Instead, it learns from the predictions made by the base models and figures out how to combine them for the best result.

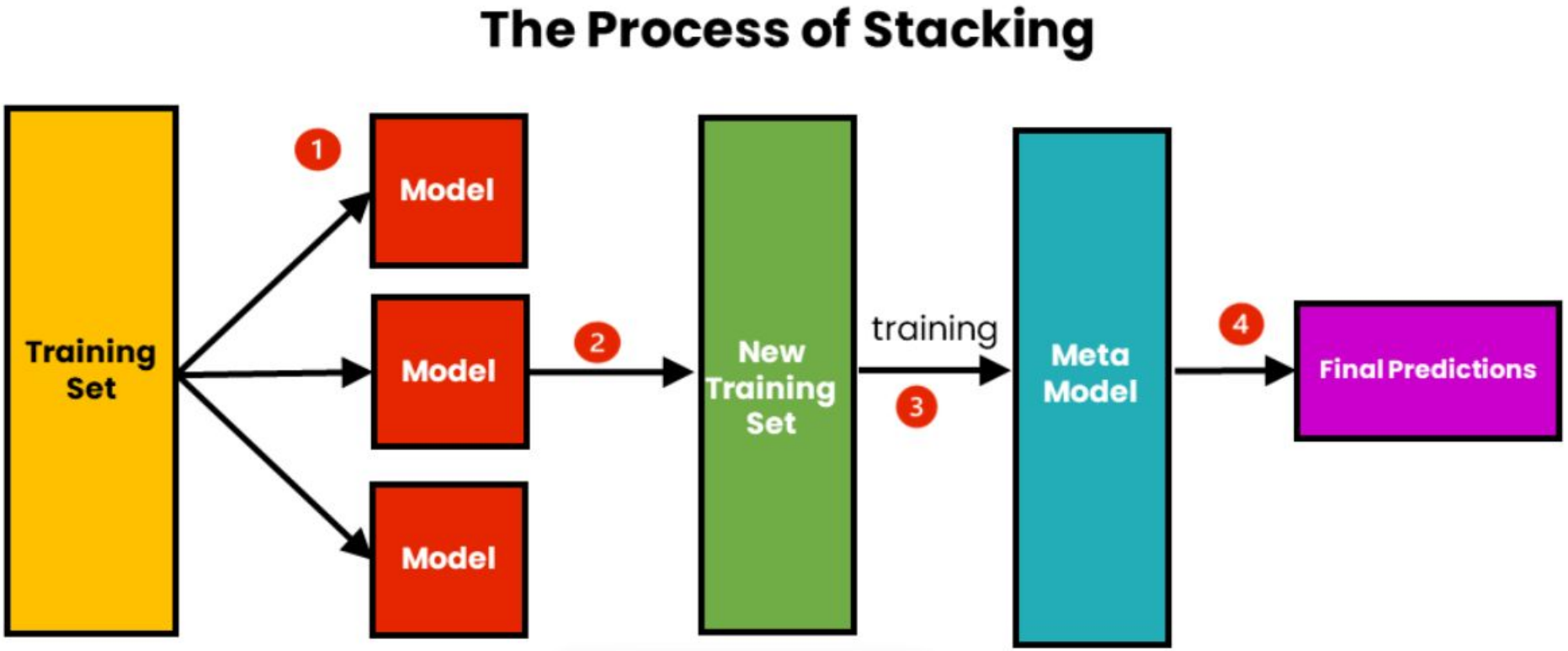
Workflow

- Train multiple base models on the same dataset.
- Collect their predictions (usually on a validation set).
- Use these predictions as input features for the meta-model.
- The meta-model learns how to best combine the base models' outputs to make final predictions.

Advantages

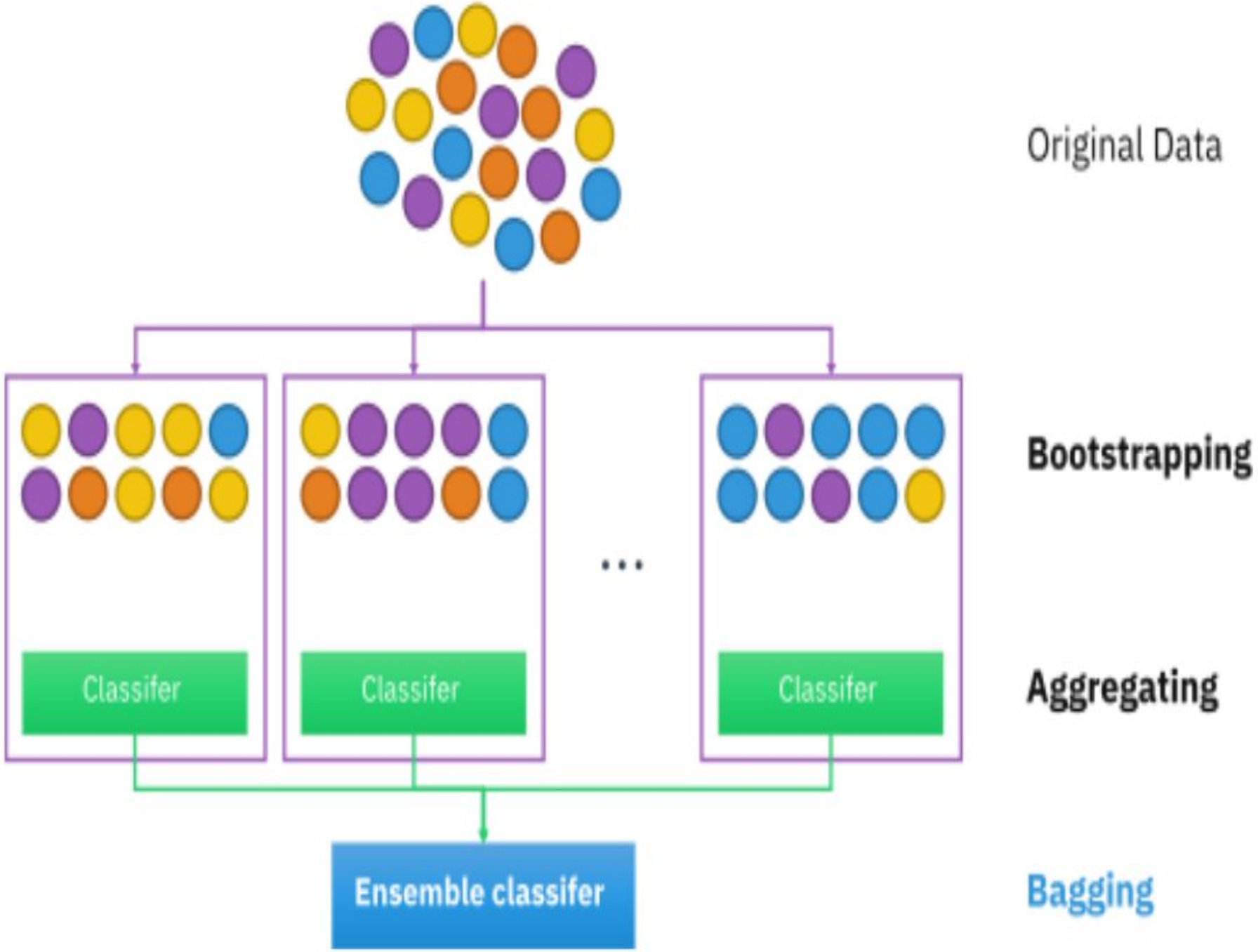
- **Improved Accuracy**: It often outperforms individual models by leveraging their complementary strengths.
- **Model Diversity**: You can mix different types of models—like a tree-based model with a linear model or a neural net.
- **Error Correction**: The meta-model learns


Training Data
(m*n)
Model 1
Model 2
Model 3
Model 4
New training set for Second level model consisting of predictions from First level model
Second level Model
Final prediction

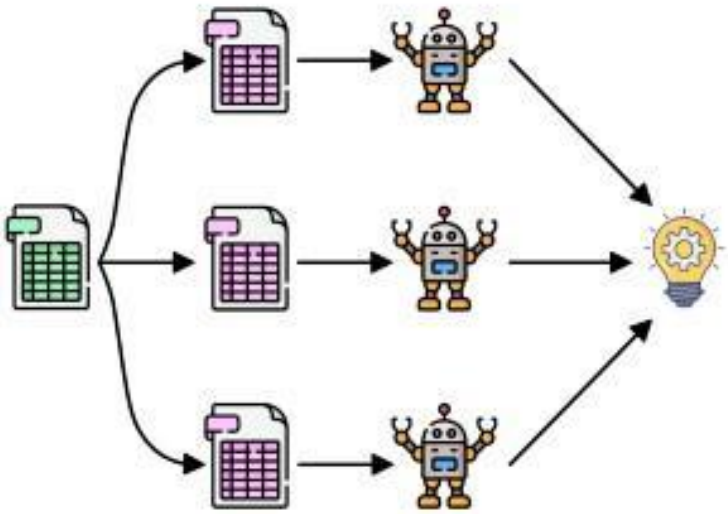

The Process of Stacking
Training Set
1
Model
Model
Model
2
New Training Set
training
3
Meta Model
4
Final Predictions

Ensemble Methods - Summary

- **Ensemble learning** is a general meta approach to machine learning that seeks better predictive performance by combining the predictions from multiple models.
- Bagging(the short form for bootstrap aggregating) involves fitting many decision trees (models) on different samples of the same dataset and averaging the predictions.
- Stacking involves fitting many different models types on the same data and using another model to learn how to best combine the predictions.
- Boosting involves adding ensemble members sequentially that correct the predictions made

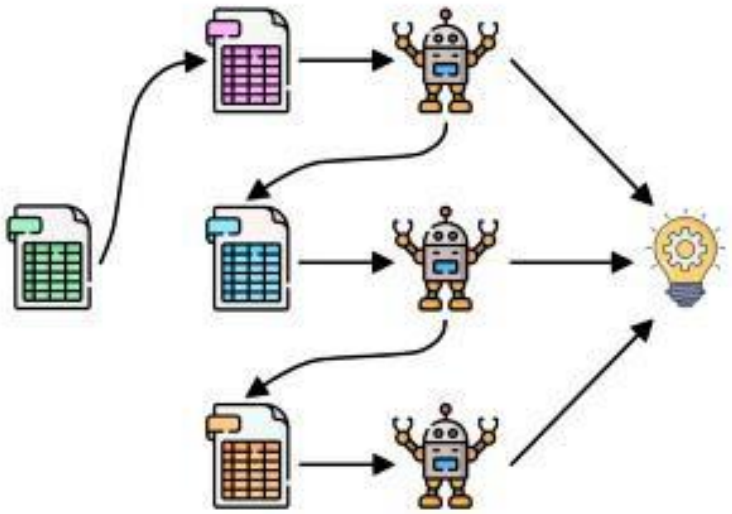

Original Data
Bootstrapping
Classifer
Classifer
Classifer
Aggregating
Ensemble classifer
Bagging

Bagging



Parallel

Boosting



Sequential

Technique	Key Idea	Model Training Style
Bagging	Reduces variance via averaging	Parallel
Boosting	Reduces bias via sequential learning	Sequential
Stacking	Combines diverse models via meta-model	Layered (two-stage)

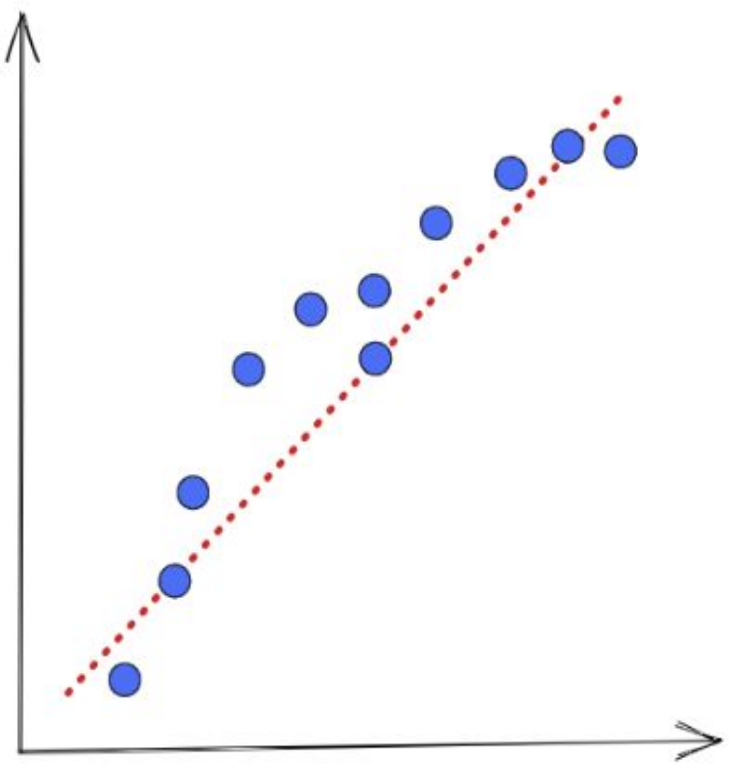
Bias Variance Trade-off / Bias Vaariance Decomposition

- **What is bias?**
- Bias is the difference between the average prediction of our model and the correct value which we are trying to predict.
- Model with high bias pays very little attention to the training data and oversimplifies the model.
- It always leads to high error on training and test data.

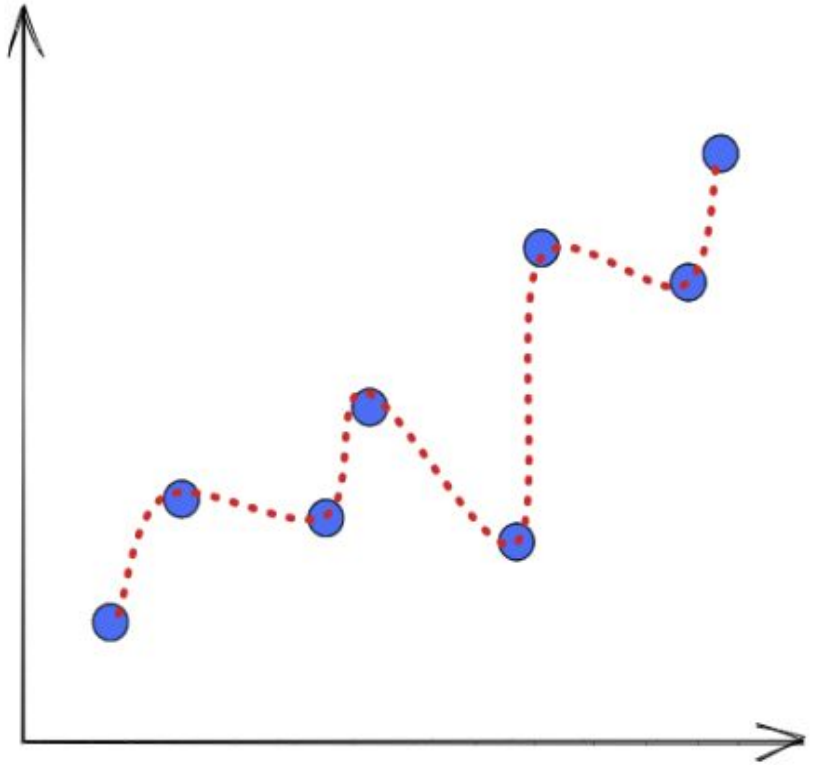
- **What is variance?**
- Variance is the variability of model prediction for a given data point or a value which tells us spread of our data.

- Model with high variance pays a lot of attention to training data and does not generalize on the data which it hasn't seen before.

- Such models perform very well on training data but has high error rates on test data.



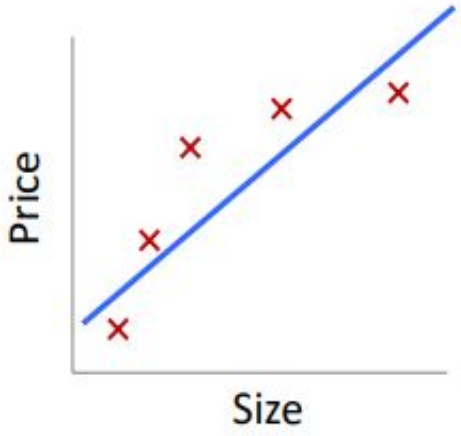
High bias



High variance

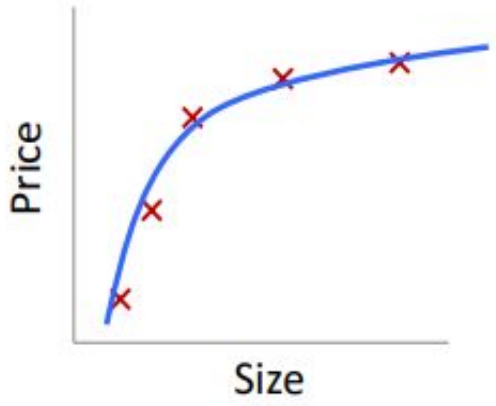
- **Low Bias:** The average prediction is very close to the target value
- **High Bias:** The predictions differ too much from the actual value
- **Low Variance:** The data points are compact and do not vary much from their mean value
- **High Variance:** Scattered data points with huge variations from the mean value and other data points

Quality of Fit



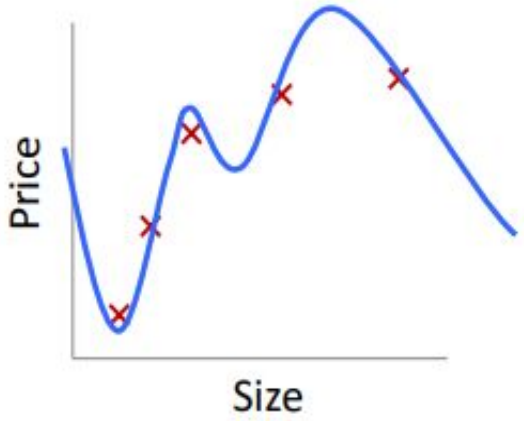
$\theta_0 + \theta_1 x$

Underfitting
(high bias)



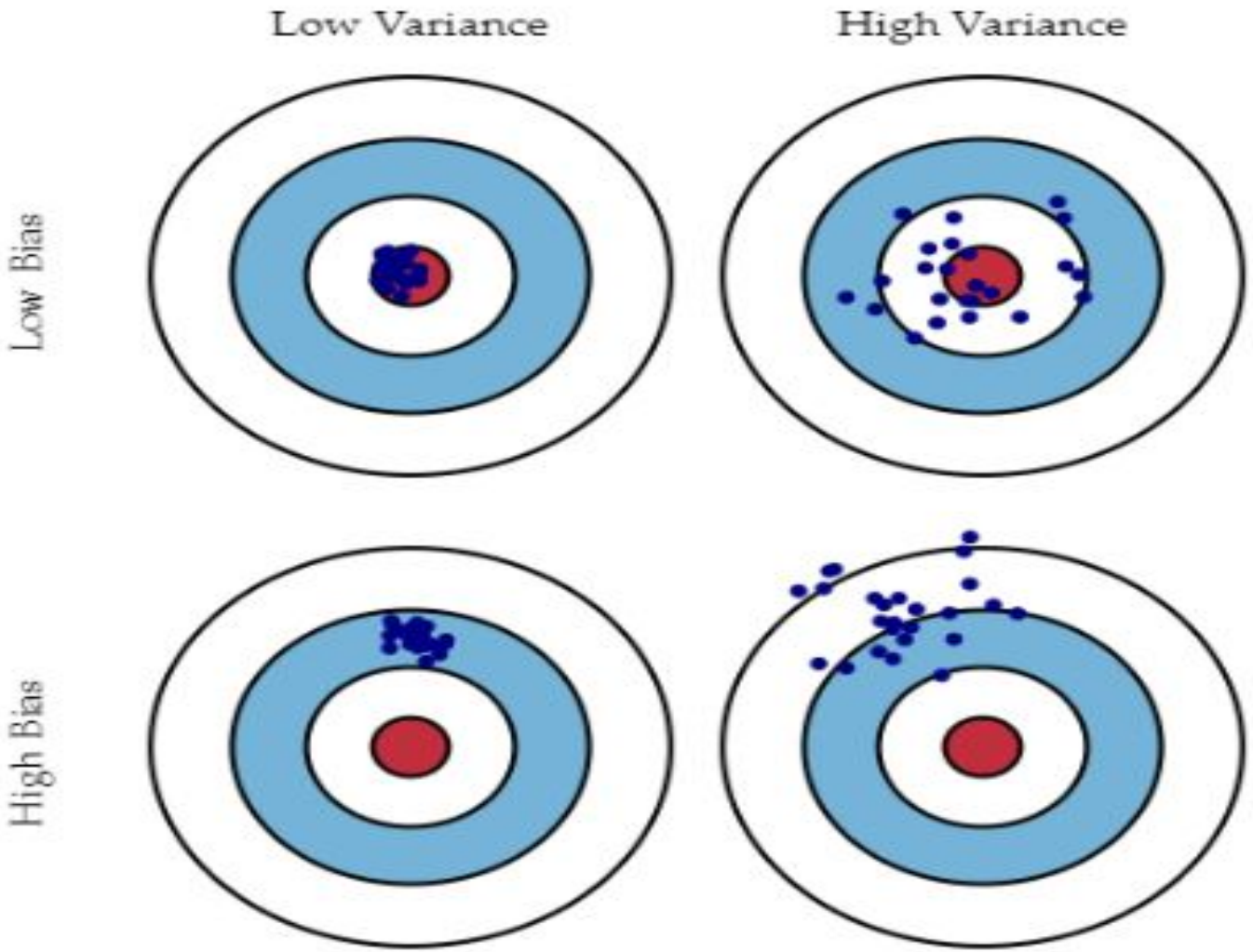
$\theta_0 + \theta_1 x + \theta_2 x^2$

Correct fit

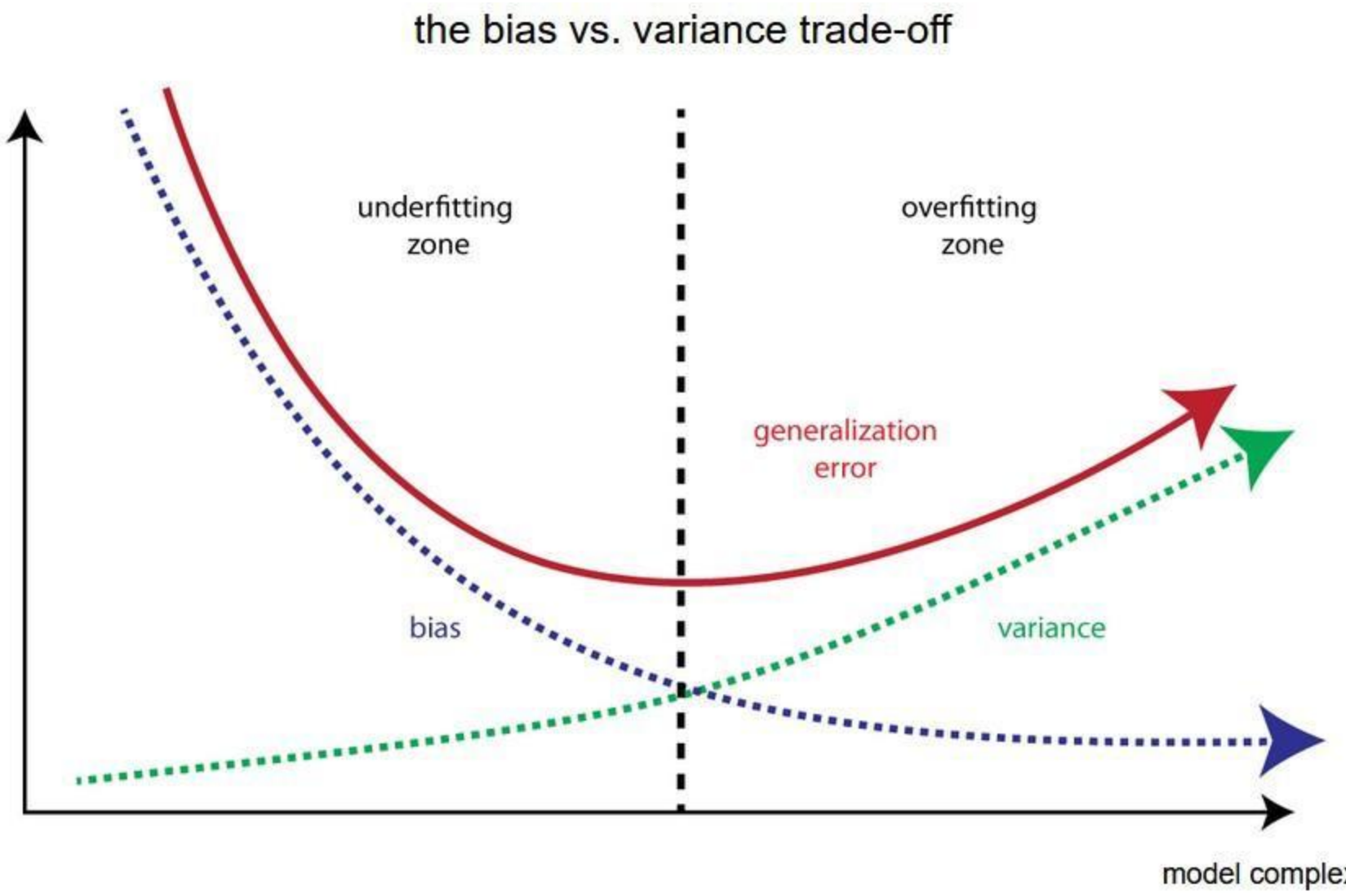


$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$

Overfitting
(high variance)


Low Variance
High Variance
Low Bias
High Bias

Bias – Variance Trade-off



Bias and Variance

- The *bias* error is an error from erroneous assumptions in the learning algorithm. High bias can cause an algorithm to miss the relevant relations between features and target outputs (underfitting).
- The *variance* is an error from sensitivity to small fluctuations in the training set. High variance may result from an algorithm modeling the random noise in the training data (overfitting).

Bias-Variance Decomposition

- The **bias–variance decomposition** is a way of analyzing a learning algorithm's expected generalization error with respect to a particular problem as a sum of three terms, the bias, variance, and a quantity called the *irreducible error*, resulting from noise in the problem itself.

- The total error of a machine learning algorithm has three components: bias, variance and noise.

$$R(h) = \mathrm{E}[L(h(z), \gamma)]$$

In the above function,"R(h)" is the cost function of the algorithm , also known as the risk function. When the risk function is loss it is the squared error.

The expectation function consolidates the losses of all potential weight values.

$$\begin{aligned}
\mathrm{E}[L(h(z), \gamma)] &= \mathrm{E}[(h(z) - \gamma)^2] \\
&= \mathrm{E}[h(z)^2] + \mathrm{E}[\gamma^2] - 2\,\mathrm{E}[\gamma\, h(z)] \\
&= \mathrm{Var}(h(z)) + \mathrm{E}[h(z)]^2 + \mathrm{Var}(\gamma) + \mathrm{E}[\gamma]^2 - 2\mathrm{E}[\gamma]\,\mathrm{E}[h(z)] \\
&= (\mathrm{E}[h(z)] - \mathrm{E}[\gamma])^2 + \mathrm{Var}(h(z)) + \mathrm{Var}(\gamma) \\
&= \underbrace{(\mathrm{E}[h(z)] - f(z))^2}_{\text{bias}^2 \text{ of method}} + \underbrace{\mathrm{Var}(h(z))}_{\text{variance of method}} + \underbrace{\mathrm{Var}(\epsilon)}_{\text{irreducible error}}
\end{aligned}$$

Hyperparameter Tuning

- To find the **optimal combination** of hyperparameters that leads to the **best model performance**—usually measured by accuracy, F1 score, RMSE, etc., depending on the task.
- Hyperparameters are the settings you define **before** training a machine learning model—they shape how the model learns, rather than what it learns
- Hyperparameters are **external configurations** that control:
- The **structure** of the model (e.g., number of layers in a neural network)
- The **training process** (e.g., learning rate, batch size, number of epochs)
- They're not learned from the data—they're set manually or through tuning.

🧪 Examples of Hyperparameters

Model Type	Hyperparameters Examples
Neural Networks	Learning rate, number of layers, batch size
Decision Trees	Max depth, min samples split
k-Nearest Neighbors	Number of neighbors (k), distance metric

1. Grid Search

- Grid Search is one of the most widely used techniques for hyperparameter tuning in machine learning.
- It's a **brute-force method** that systematically tests every possible combination of hyperparameters from a predefined set to find the best-performing configuration.

Grid Search works by:

- **Defining a grid** of hyperparameter values.
- **Training the model** on each combination.
- **Evaluating** performance using cross-validation

Pros	Cons
Simple to understand	Computationally expensive
Exhaustive search	Doesn’t scale well with large grids
Works well with small spaces	May overfit if not cross-validated

2. Random Search

- Instead of exhaustively trying every combination like Grid Search, it **randomly samples** combinations from the hyperparameter space.
- This randomness often leads to faster discovery of good configurations—especially when only a few hyperparameters significantly impact performance

Random Search works by:

- Defining a **range or distribution** for each hyperparameter.
- Randomly selecting combinations from these ranges.
- Evaluating each combination using cross-validation.

Pros	Cons
Faster than Grid Search	May miss optimal combinations
Good for large search spaces	Results can vary between runs
Easy to parallelize	Requires careful distribution setup

3. Bayesian Optimization

- It **learns from past results** to intelligently choose the next best set of hyperparameters to try.
- It's especially useful when model training is expensive or time-consuming.
- Bayesian Optimization treats hyperparameter tuning as a **black-box optimization problem**. You don't need to know the internal workings of the model—just how well it performs for a given set of hyperparameters.
- It builds a **probabilistic model** of the objective function (e.g., accuracy or loss) and uses it to decide where to search next.

Workflow

- 1. Initialize with a few random hyperparameter evaluations.
- 2. Fit surrogate model to these results.
- 3. Use acquisition function to select next hyperparameters.
- 4. Evaluate model performance.
- 5. Update surrogate model.
- 6. Repeat steps 3–5 until stopping criteria are met.

Explanation

Surrogate Model

- A statistical model (often a **Gaussian Process**) that approximates the true objective function.

Acquisition Function

- Guides the search for the next hyperparameter set.

Evaluation and Update

- The selected hyperparameters are evaluated using the actual model.
- The surrogate model is updated with this new result.

Pros	Cons
Efficient for expensive models	More complex to implement
Learns from previous evaluations	Slower for very large datasets
Balances exploration and exploitation	Requires careful choice of surrogate model

4. **Genetic Algorithms**

- GAs simulate the process of biological evolution to find optimal solutions.
- Each potential solution (a set of hyperparameters) is treated as an **individual** in a population.
- These individuals evolve over generations using operations like **selection**, **crossover**, and **mutation**.

Procedure

1. Encoding Hyperparameters

- Each hyperparameter set is encoded as a **chromosome**—a vector where each **gene** represents a specific hyperparameter (e.g., learning rate, number of layers).

2. Initialization

- Start with a **random population** of hyperparameter sets.

3. Fitness Evaluation

- Train a model using each set and evaluate its performance (e.g., accuracy, F1 score). This score is the **fitness** of the individual.

4. Selection

- Choose the best-performing individuals to be **parents** for the next generation. Techniques include:
- **Roulette wheel selection**
- **Tournament selection**

5. Crossover (Recombination)

- Combine genes from two parents to produce **offspring**. This mimics reproduction and allows good traits to propagate.

6. Mutation

- Introduce random changes to some genes to maintain **diversity** and avoid local optima.

7. Replacement

- Select individuals for the next generation—either the best ones or a mix of old and new.

8. Repeat

- Continue for a fixed number of generations or until

Pros	Cons
Global search capability	Computationally intensive
Works with any model (model-agnostic)	Requires careful parameter encoding
No gradient needed	May need tuning of GA parameters
Parallelizable	Slower than Bayesian methods in some cases

5. Gradient-Based Optimization

- Gradient-Based Optimization for hyperparameter tuning is a more advanced and mathematically elegant approach that leverages **gradient information** to guide the search for optimal hyperparameters.
- Unlike brute-force methods like Grid or Random Search, this technique uses calculus to **follow the slope of the loss function** and adjust hyperparameters in the direction that improves model performance

- it's about computing the **gradient of the loss function with respect to hyperparameters**, and then updating those hyperparameters using a rule like gradient descent:

$$\lambda \leftarrow \lambda - \eta \cdot \nabla_\lambda L(\theta, \lambda)$$

Where:

- λ = hyperparameters
- θ = model parameters
- $L(\theta, \lambda)$ = loss function
- η = learning rate
- $\nabla_\lambda L$ = gradient of loss with respect to hyperparameters [1]

Working

- **Initialize hyperparameters** (e.g., learning rate, regularization strength).
- **Train the model** and compute the loss.
- **Use automatic differentiation** (e.g., via TensorFlow or PyTorch) to calculate gradients of the loss with respect to hyperparameters.
- **Update hyperparameters** using gradient descent.
- **Repeat** until convergence or performance plateaus.

Pros	Cons
Efficient for differentiable models	Requires differentiable hyperparameters
Can converge quickly	Complex to implement
Works well with deep learning	Not suitable for discrete hyperparameters

Thank You