Generalization error, also known as out-of-sample error, is a measure of how well a machine learning model performs on unseen data. In other words, it measures the ability of the model to generalize beyond the training data and make accurate predictions on new, previously unseen data.

Generalization error occurs when a model is trained too well on the training data, leading it to be overfit and not able to generalize to new data. This can happen if the model is too complex or if there is not enough diverse training data. On the other hand, if a model is underfit, it may not perform well even on the training data, let alone on unseen data.

The goal of machine learning is to minimize the generalization error by finding the right balance between model complexity and training data size and diversity. This is typically achieved through techniques such as cross-validation and regularization.

OVERFITTING
MODEL SELECTION
EVALUATION

Classification Errors

چند نوع خطا داریم

خطای داده های آموزشی مدل من پس از ساخته شدن روی خود داده های اموزشی چه خطایی داره؟

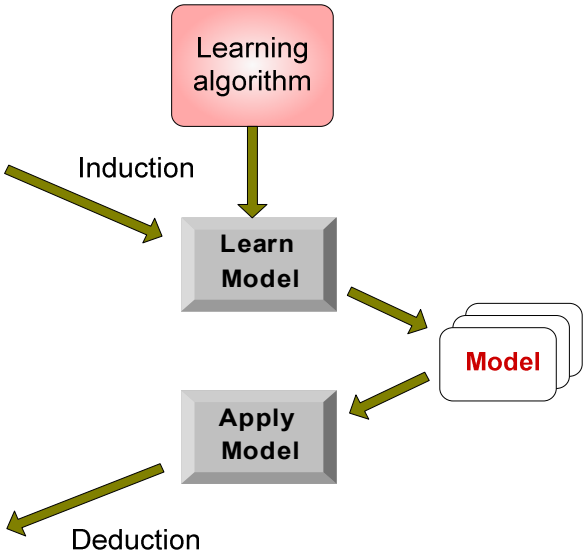
- **Training errors**: Errors committed on the training set
- **Test errors**: Errors committed on the test set
- **Generalization errors**: Expected error of a model over random selection of records from same distribution

خطاهای آموزشی: خطاهای مرتکب در مجموعه آموزشی
خطاهای تست: خطاهای مرتکب شده در مجموعه تست
خطاهای تعمیم: خطای مورد انتظار یک مدل در انتخاب تصادفی رکوردها از همان توزیع

خطای عمومی یا عام
ما دنبال این هستیم که یه تخمینی از خطای عام پیدا کنیم
خطای عام ینی به طور متوسط مدل ما چقدر میتونه وی کلسیفیکیشن خوب عمل کنه؟

Tid	Attrib1	Attrib2	Attrib3	Class
1	Yes	Large	125K	No
2	No	Medium	100K	No
3	No	Small	70K	No
4	Yes	Medium	120K	No
5	No	Large	95K	Yes
6	No	Medium	60K	No
7	Yes	Large	220K	No
8	No	Small	85K	Yes
9	No	Medium	75K	No
10	No	Small	90K	Yes

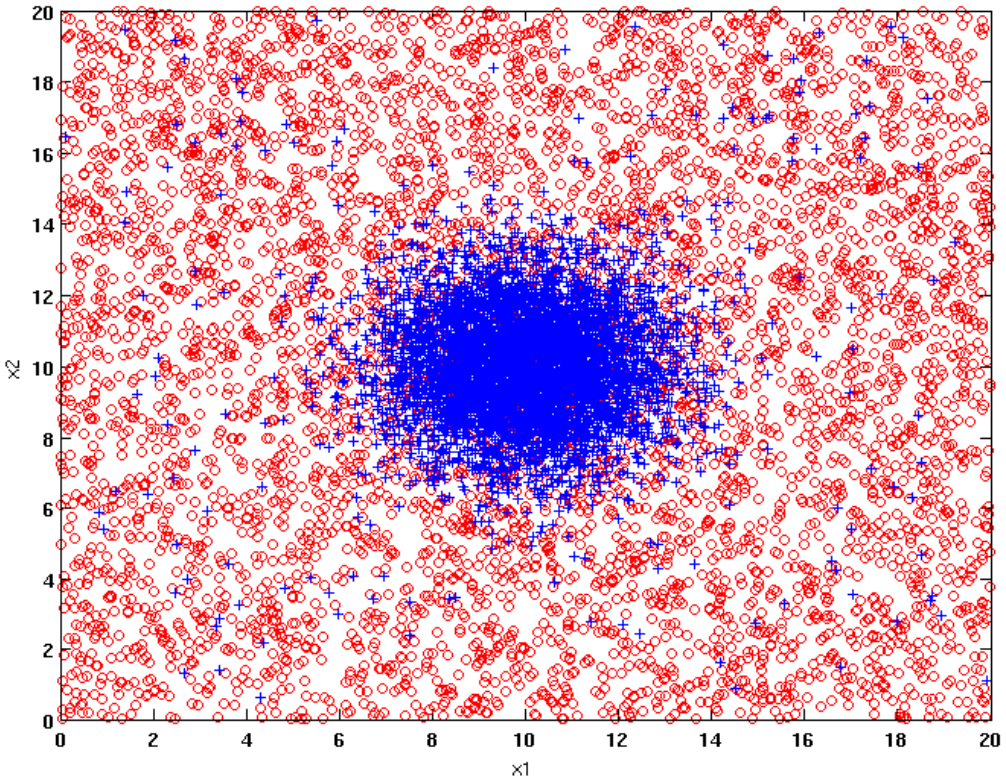
Training Set

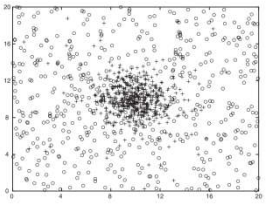


Tid	Attrib1	Attrib2	Attrib3	Class
11	No	Small	55K	?
12	Yes	Medium	80K	?
13	Yes	Large	110K	?
14	No	Small	95K	?
15	No	Large	67K	?

Test Set

Example Data Set





(b) Training set using 10% data.

Two class problem:

+ : 5400 instances

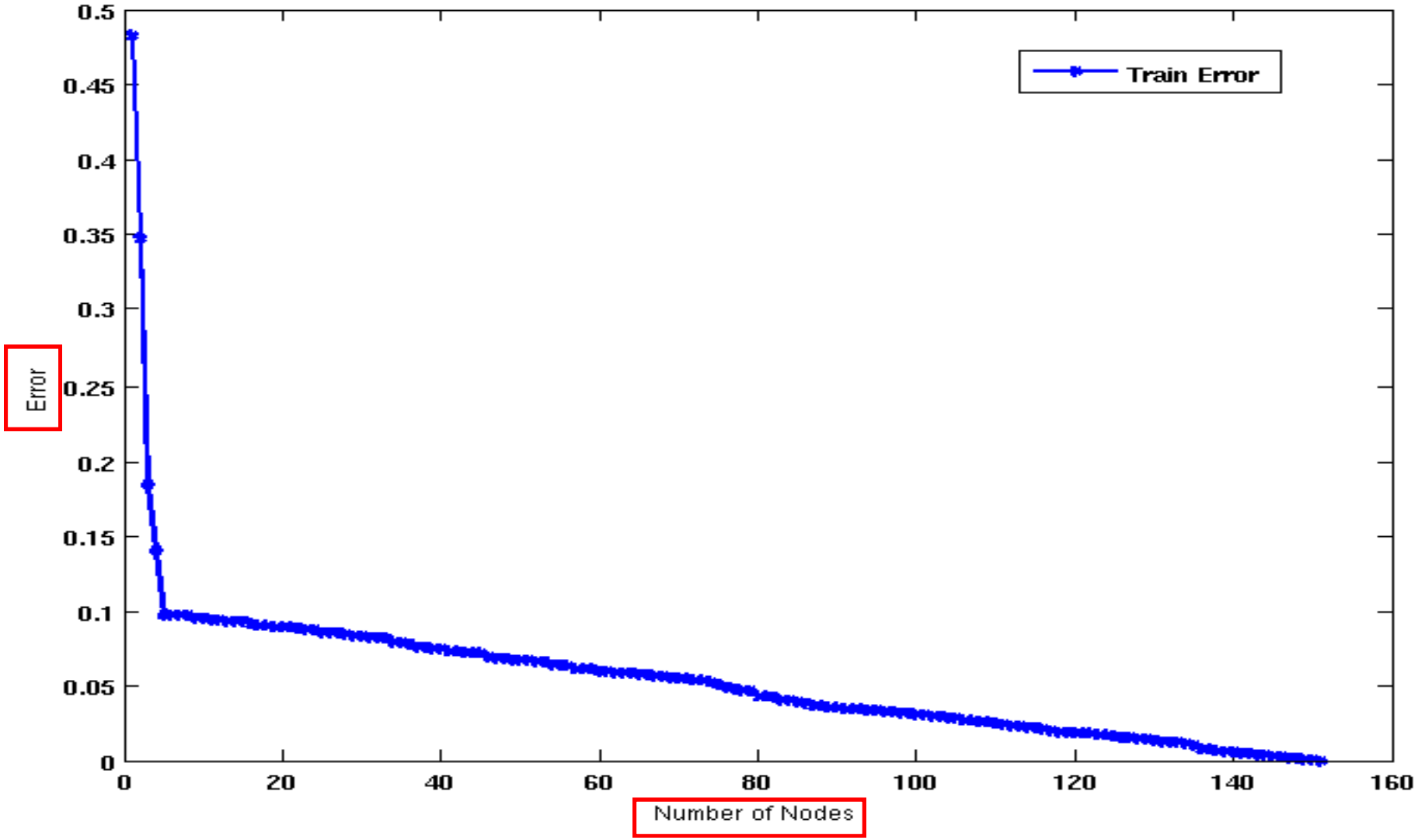
- 5000 instances generated from a Gaussian centered at (10,10)
- 400 noisy instances added

o : 5400 instances

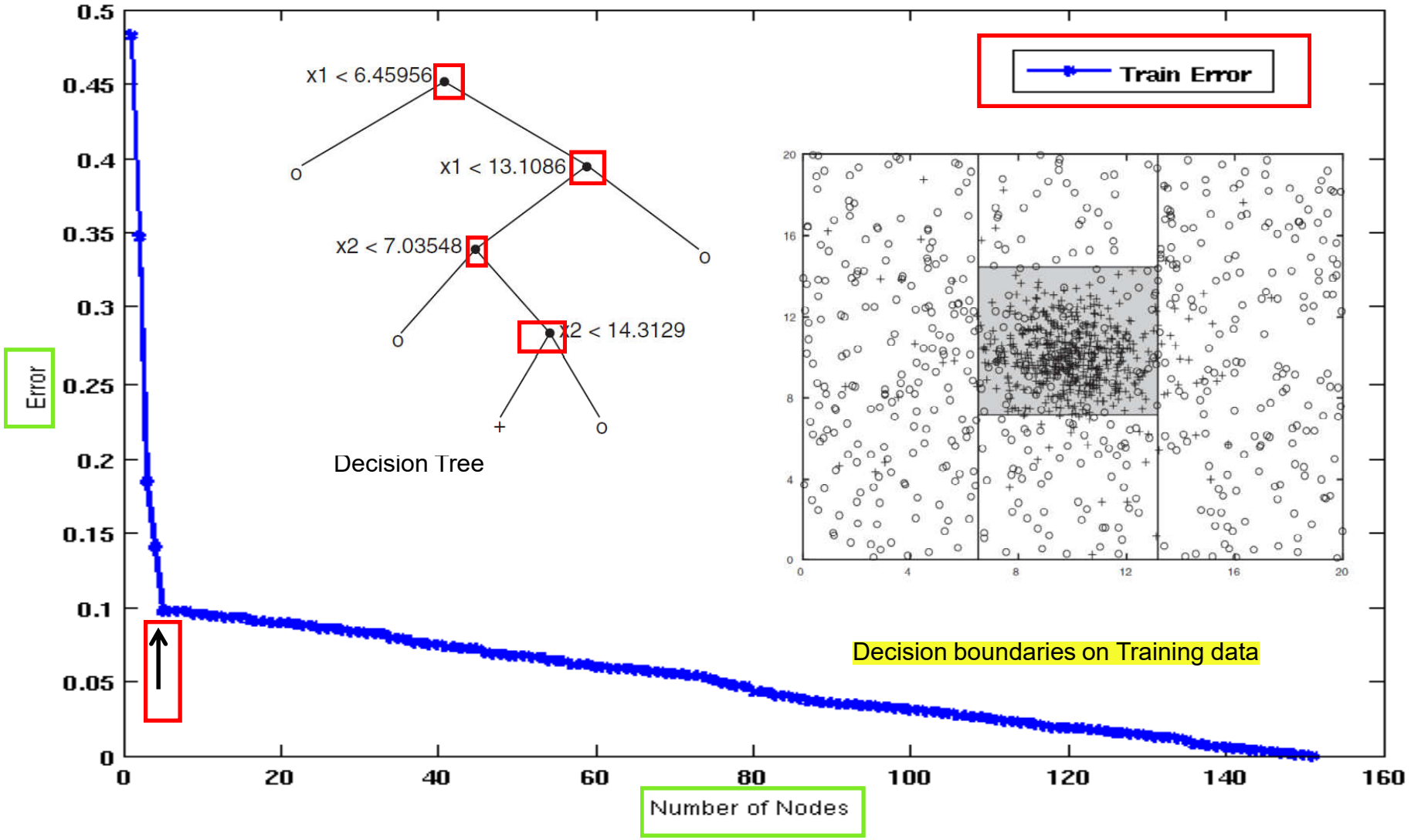
- Generated from a uniform distribution

10 % of the data used for training and 90% of the data used for testing

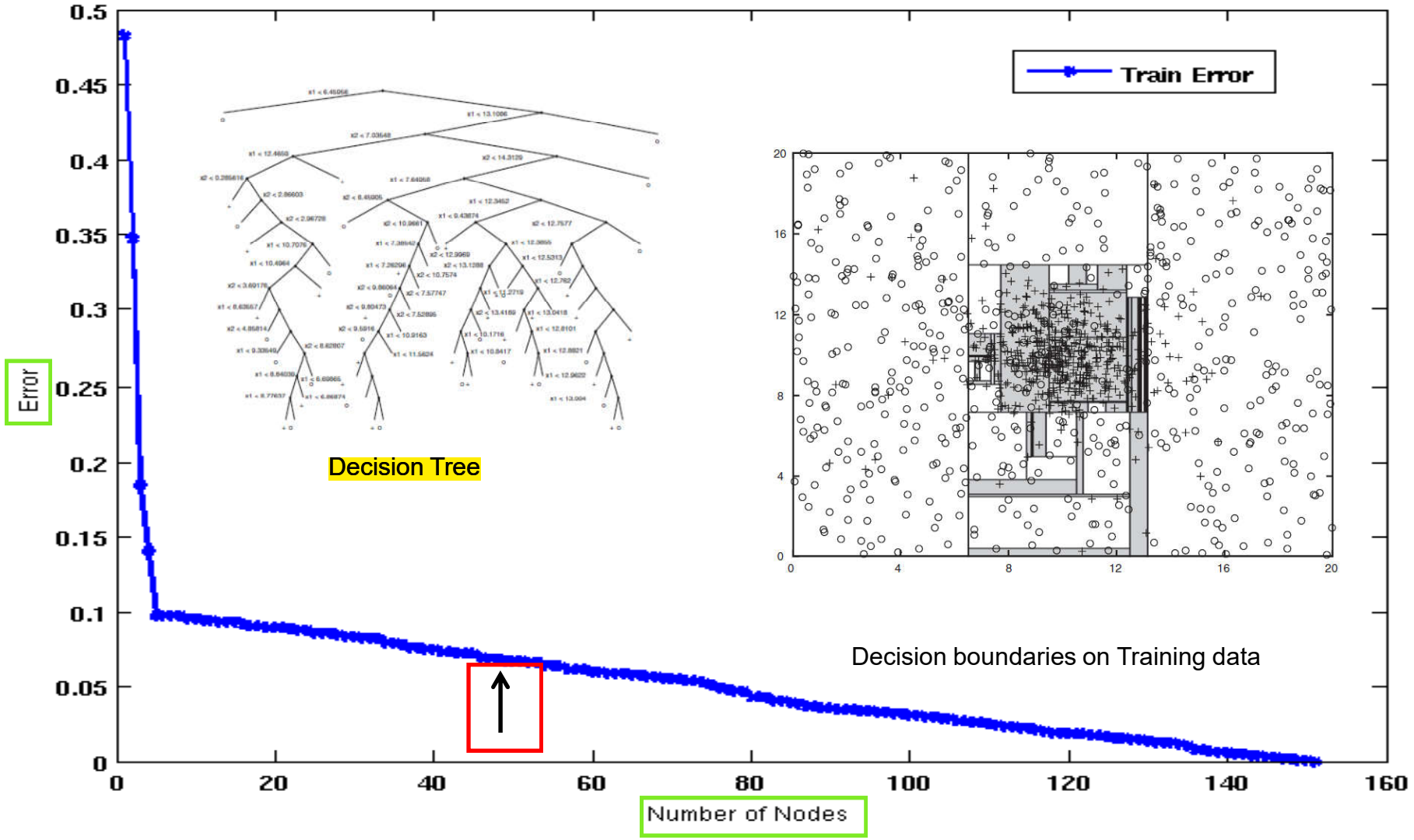
Increasing number of nodes in Decision Trees



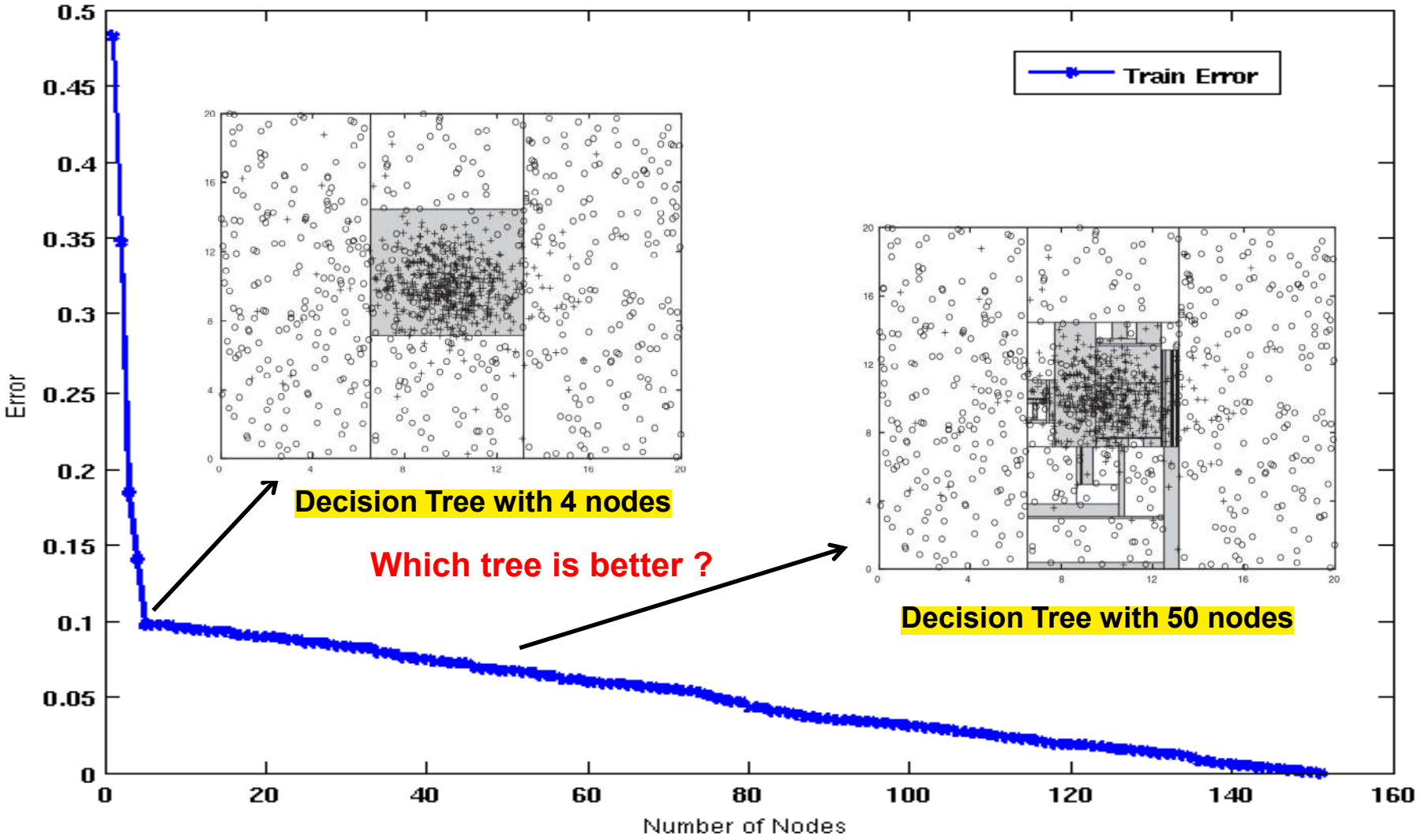
Decision Tree with 4 nodes



Decision Tree with 50 nodes

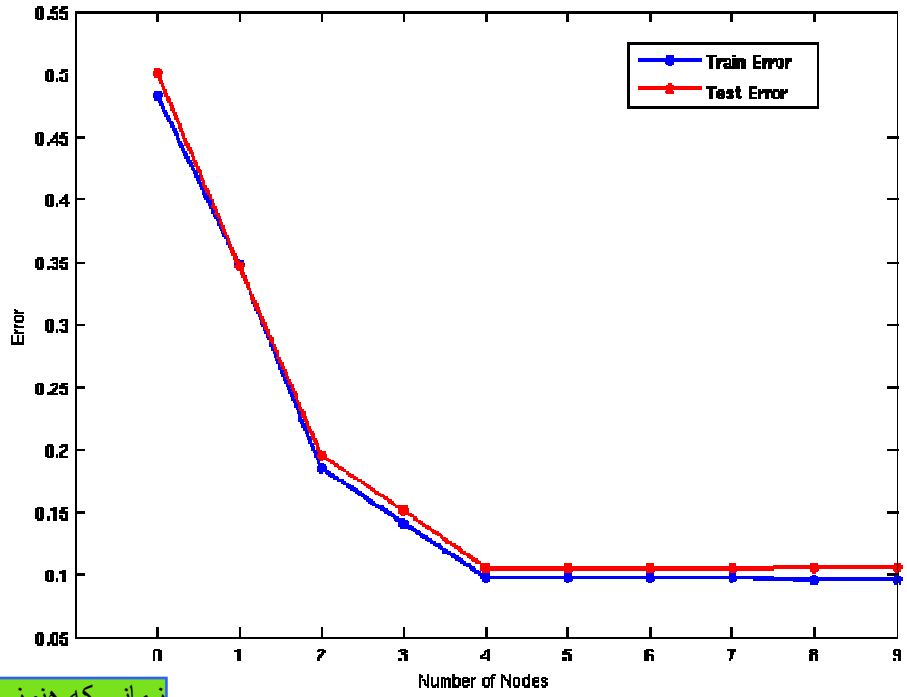


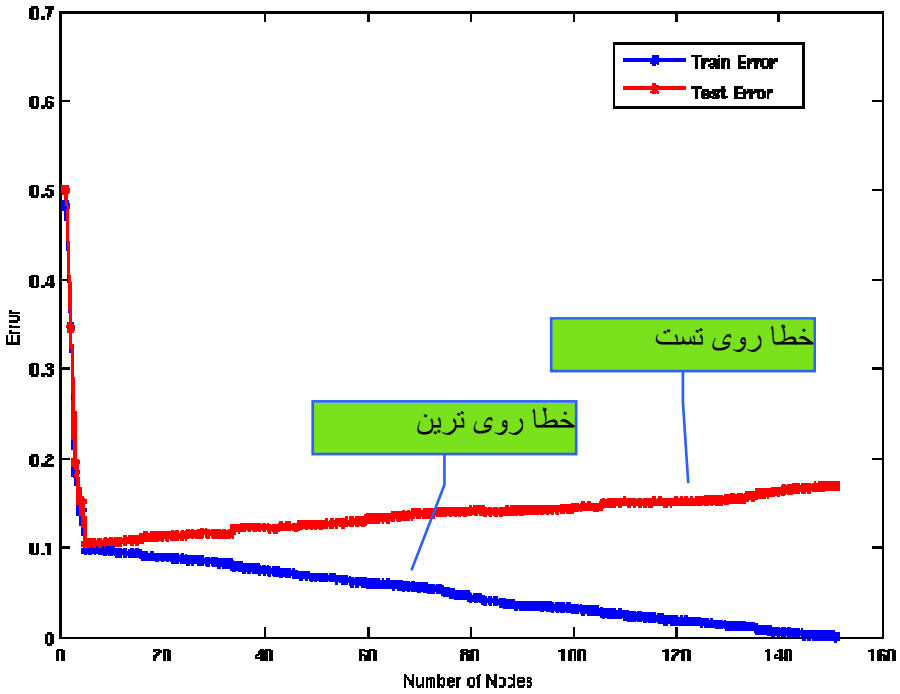
Which tree is better?



منظور همون درخته

Model Underfitting and Overfitting





زمانی که هنوز تقریبا چیزی یاد نگرفتیم از داد ها هم خطای اموزشمون بالاست هم تست

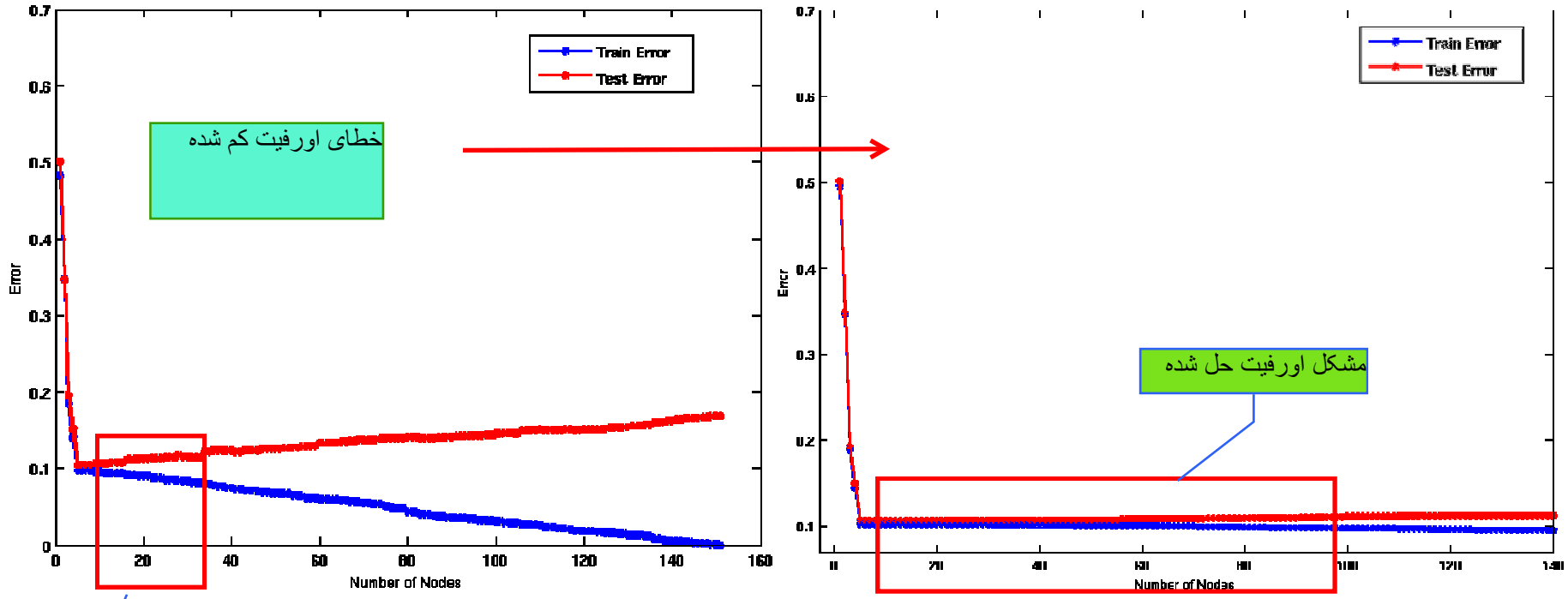
- As the model becomes more and more complex, test errors can start increasing even though training error may be decreasing

Underfitting: when model is too simple, both training and test errors are large

Overfitting: when model is too complex, training error is small but test error is large

در مدل های کلسیفیکیشن، تا یه جایی خطا روی داده های تست و ترین هردوشون کاهش پیدا میکنند ولی از یه جایی به بعد خطا روی داده های تست افزایش پیدا میکنه ولی خطای داده های ترین همچنان داره کم میشه =>>> اینجا جاییه که میگیم داریم دچار overfitمیشیم ینی داریم داده های ترین را حفظ میکنیم

Model Overfitting – Impact of Training Data Size



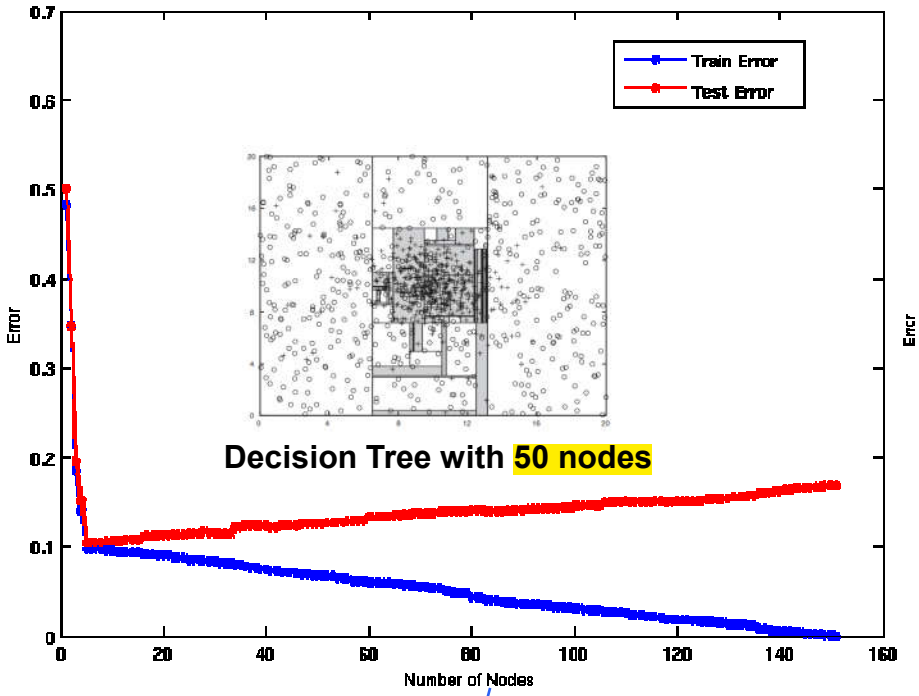
Using twice the number of data instances

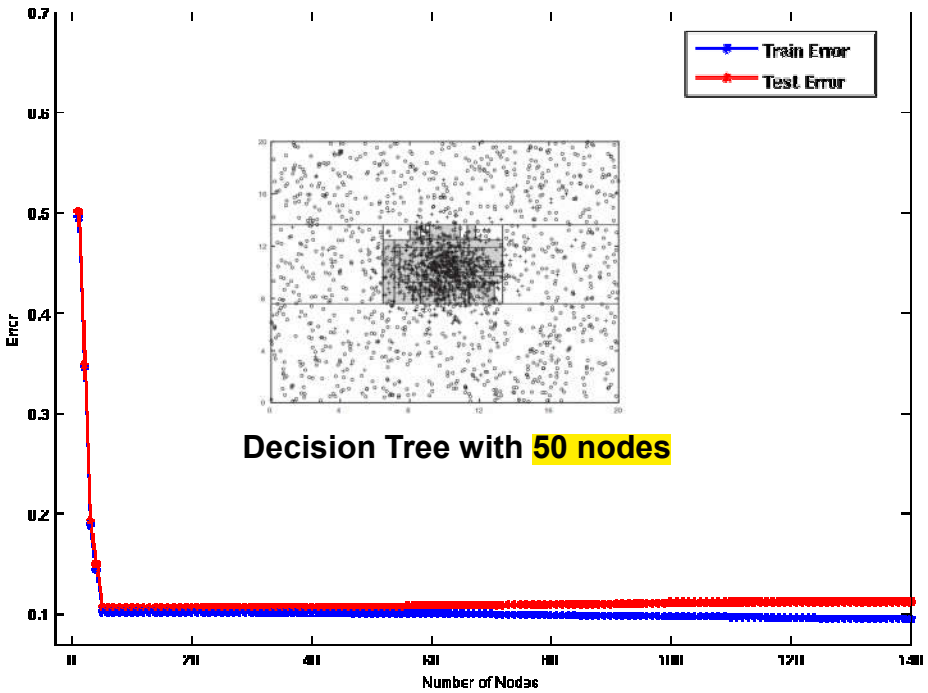
- Increasing the size of training data reduces the difference between training and testing errors at a given size of model

به ۲۰تا نود که میرسیم خطای ترین کم میشه همچنان ولی خطای تست افزایش پیدا میکنه

اگه به جای ۱۰درصدی که برای داده های اموزشی کنار گذاشتیم، درصد بیشتری را برای اموزش اختصاص بدیم چه اتفاقی میفته؟ الگوهایی که توی داده های اموزشی هستند بیشتر خودشان را نشان میدن و مدل دیرتر گیج میشه و احتمال اورفیت شدن کمتر میشه

Model Overfitting – Impact of Training Data Size





Using twice the number of data instances

- Increasing the size of training data reduces the difference between training and testing errors at a given size of model

وقتی داده های اموزشی کم باشه نویز ها خودشون را بیشتر نشون میدن و بیشتر روی مدل تاثیر میگذارن ینی بادیدن نمونه های مختلف سعی میکنه اون نمونه را مثل یک حالت خاص ببینه و براش یه ایف جداگانه بگذاره یا یک برنچ جداگانه براش بسازه

Reasons for Model Overfitting

مدل وقتی خیلی پیچیده بشه احتمال اینکه تک تک تصمیم گیری هارا غلط انجام بده خیلی زیاد میشه
منظور از پیچیدگی چیه؟ مثلا در درخت ها منظور یا شاخص پیچیدگی میتونه تعداد نودها باشه
در یک معادله مثلا تعداد پارامتر ها نشان دهنده ی پیچیدگی باشه

- Not enough training data
- High model complexity
  - Multiple Comparison Procedure

دلایل بیش از حد برازش مدل
داده های آموزشی کافی نیست
پیچیدگی مدل بالا
- روش مقایسه چندگانه

The impact of training data size on model overfitting can be significant. When the training set is too small relative to the complexity of the model and the variability of the data, overfitting is more likely to occur. Overfitting happens when a model is trained to fit the noise in the training data instead of the underlying patterns. This results in a high variance in the model's performance, meaning it will perform well on the training data but poorly on new data.

Increasing the size of the training set can help reduce the risk of overfitting by providing the model with more examples to learn from and better generalization of the underlying patterns. However, there is a point of diminishing returns where additional data may not provide any more benefit or may even hurt performance if the added data is noisy or irrelevant.

It's important to keep in mind that the impact of training data size on overfitting also depends on the complexity of the model being trained. A simpler model may require less data to generalize well, while a more complex model may need more data to avoid overfitting. Therefore, choosing an appropriate model complexity and having a sufficiently large and diverse training dataset are both important factors for building accurate machine learning models.

Effect of Multiple Comparison Procedure

- Consider the task of predicting whether stock market will rise/fall in the next 10 trading days

- Random guessing:

$$P(correct) = 0.5$$

- Make 10 random guesses in a row:

$$P(\#\,correct \geq 8) = \frac{\binom{10}{8} + \binom{10}{9} + \binom{10}{10}}{2^{10}} = 0.0547$$

Day 1	Up
Day 2	Down
Day 3	Down
Day 4	Up
Day 5	Down
Day 6	Down
Day 7	Up
Day 8	Up
Day 9	Up
Day 10	Down

Effect of Multiple Comparison Procedure

- Approach:
  - Get 50 analysts
  - Each analyst makes 10 random guesses
  - Choose the analyst that makes the most number of correct predictions

- Probability that at least one analyst makes at least 8 correct predictions

$$P(\#\,correct \geq 8) = 1 - (1 - 0.0547)^{50} = 0.9399$$

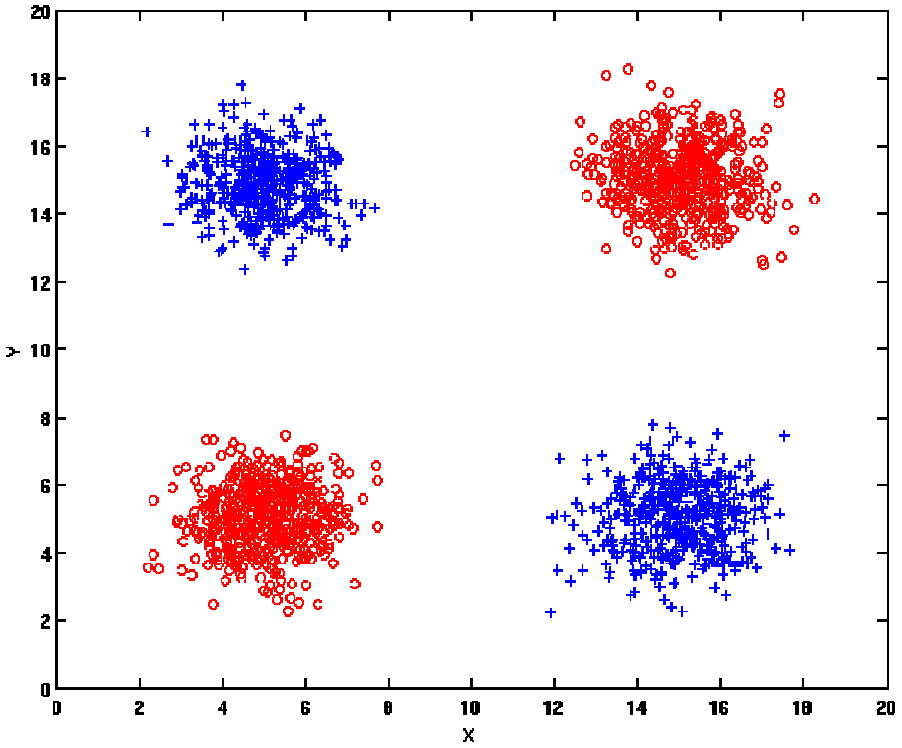
Effect of Multiple Comparison Procedure

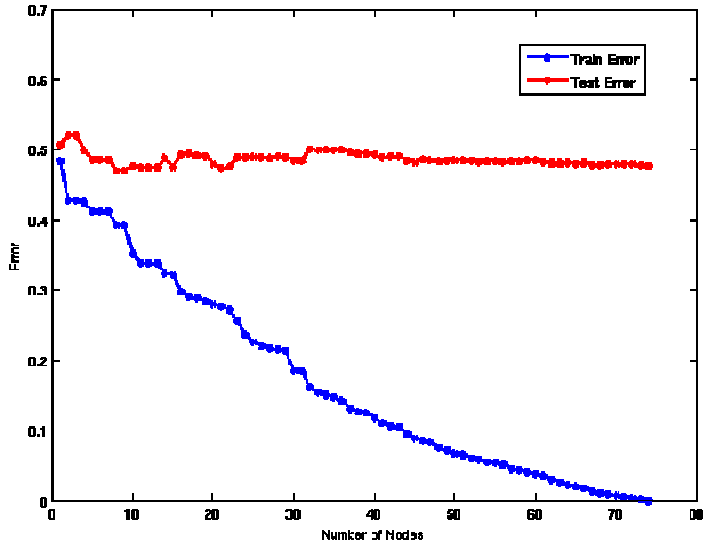
- Many algorithms employ the following greedy strategy:
  - Initial model: M
  - Alternative model: M' = M $\cup$ γ,
    where γ is a component to be added to the model
    (e.g., a test condition of a decision tree)
  - Keep M' if improvement, Δ(M,M') > α

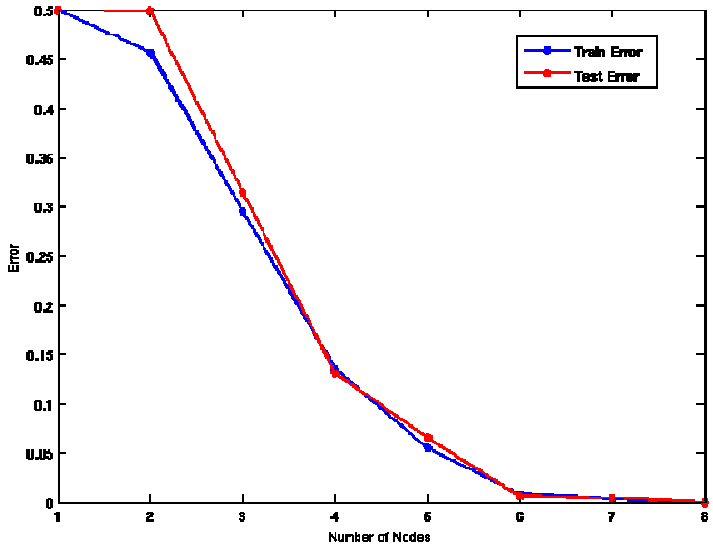
- Often times, γ is chosen from a set of alternative components, $\Gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_k\}$

- If many alternatives are available, one may inadvertently add irrelevant components to the model, resulting in model overfitting

Effect of Multiple Comparison - Example







Use additional 100 noisy variables generated from a uniform distribution along with X and Y as attributes.

Use 30% of the data for training and 70% of the data for testing

Using only X and Y as attributes

Notes on Overfitting

- Overfitting results in decision trees that are more complex than necessary

- Training error does not provide a good estimate of how well the tree will perform on previously unseen records

خطا روی داده های تستمون داره خوب عمل میکنه ولی روی داده هایی که مدل تاحالا ندیده چطوری عمل میکنه؟

- Need ways for estimating generalization errors(How?)

نکاتی در مورد Overfitting
برازش بیش از حد منجر به درخت های تصمیم می شود که پیچیده تر از حد لازم هستند
خطای آموزشی تخمین خوبی از عملکرد درخت در رکوردهای نادیده قبلی ارائه نمی دهد.
به روش هایی برای تخمین خطاهای تعمیم نیاز دارید (چگونه؟)

Model Selection

میخایم بین مدل ها انتخاب کنیم که کدومشون برای داده های ما بهتره؟ مثلا مدلی که ۴ تا نود داره بهتره یا اونی که ۵۰تا نود داره بهتره؟

۲تا رویکرد یا راه حل برای انتخاب مدل هست:
۱. رویکرد داده محور که یه مجموعه ی validation یا ارزیابی درنظر میگیریم
۲. رویکرد محاسباتی و فرمولی

- Purpose
  - ensure that model is not overly complex (to avoid overfitting)
  - Performed during model building

در حین یادگیری باید سریع راجع بش تصمیم بگیریم که کدوم مدل بهتره؟

- Need to estimate generalization error
  - Using Validation Set
  - Incorporating Model Complexity

هدف
اطمینان حاصل کنید که مدل بیش از حد پیچیده نیست (برای جلوگیری از برازش بیش از حد)
– در حین ساخت مدل اجرا می شود
نیاز به تخمین خطای تعمیم
– استفاده از Validation Set
– گنجاندن پیچیدگی مدل

Estimating generalization error is important to ensure that a machine learning model can perform well on new, unseen data. One common approach to estimating generalization error is to split the available data into training and validation sets. The training set is used to train the model, while the validation set is used to evaluate its performance and estimate its generalization error.

To do this, we typically train the model on the training set and then use the validation set to measure its performance. We can repeat this process multiple times, using different subsets of the data for training and validation, to get an estimate of the model's average performance and generalization error. This process is known as cross-validation.

Another approach to estimating generalization error is to use a holdout set, which is a small portion of the available data that is not used during training or validation. After we have trained and validated our model, we can use the holdout set to test its performance on completely new, unseen data. This gives us an estimate of the model's true generalization error.

Model Selection:

Using Validation Set

- Divide training data into two parts:
  - Training set:
    - use for model building
  - Validation set:
    - use for estimating generalization error
    - Note: validation set is not the same as test set

- Drawback:
  - Less data available for training

با استفاده از مجموعه اعتبار سنجی
داده های آموزشی را به دو قسمت تقسیم کنید:
– مجموعه آموزشی:
استفاده برای ساخت مدل
– مجموعه اعتبارسنجی:
استفاده برای تخمین خطای تعمیم
توجه: مجموعه اعتبارسنجی با مجموعه آزمایشی یکسان نیست
اشکال:
- داده های کمتری برای آموزش در دسترس است

انتخاب براساس پیچیدگی مدل
مدلی خوبه که کمتر پیچیده باشه مدلی که از همه ساده تره بهتره

Model Selection:

Incorporating Model Complexity

با توجه به دو مدل از خطاهای تعمیم مشابه، باید مدل ساده تر را بر مدل پیچیده تر ترجیح داد
- یک مدل پیچیده شانس بیشتری برای فیت شدن تصادفی دارد-

- Rationale: Occam's Razor
  - Given two models of similar generalization errors,

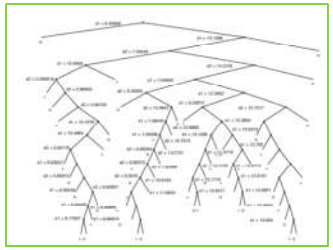
    one should prefer the simpler model over the more complex model

  - A complex model has a greater chance of being fitted accidentally

بنابراین، هنگام ارزیابی یک مدل باید پیچیدگی مدل را لحاظ کرد

  - Therefore, one should include model complexity when evaluating a model

اندازه گیری پیچیدگی مدل به کلسیفایر بستگی داره
مثلا توی درخت پیچیدگی را با عمقش مثلا میسنجیم یا تعداد برگ هاش



$$\text{Gen. Error(Model)} = \text{Train. Error(Model, Train. Data)} + \alpha \times \text{Complexity(Model)}$$

خطای اموزشی

جریمه درصورت افزایش پیچیدگی مدل

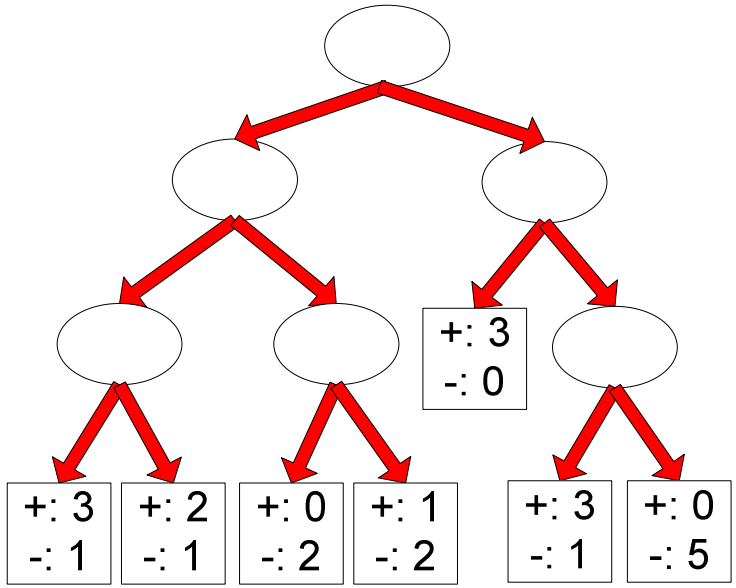
Estimating the Complexity of Decision Trees

- **Pessimistic Error Estimate** of decision tree T with k leaf nodes:

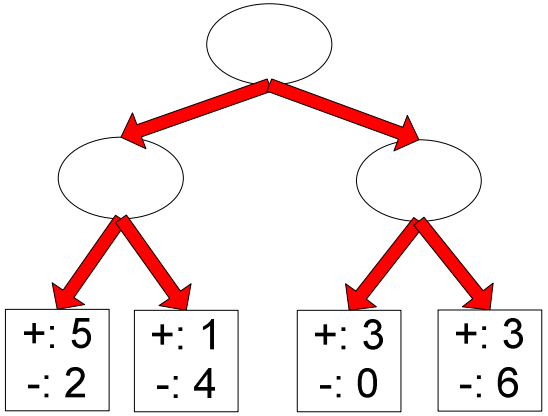
$$err_{gen}(T) = err(T) + \Omega \times \frac{k}{N_{train}}$$

  - err(T): error rate on all training records
  - Ω: trade-off hyper-parameter (similar to α)
    - Relative cost of adding a leaf node
  - k: number of leaf nodes
  - N_{train}: total number of training records

Estimating the Complexity of Decision Trees: Example



Decision Tree, T_L



Decision Tree, T_R

$e(T_L) = 4/24$

$e(T_R) = 6/24$

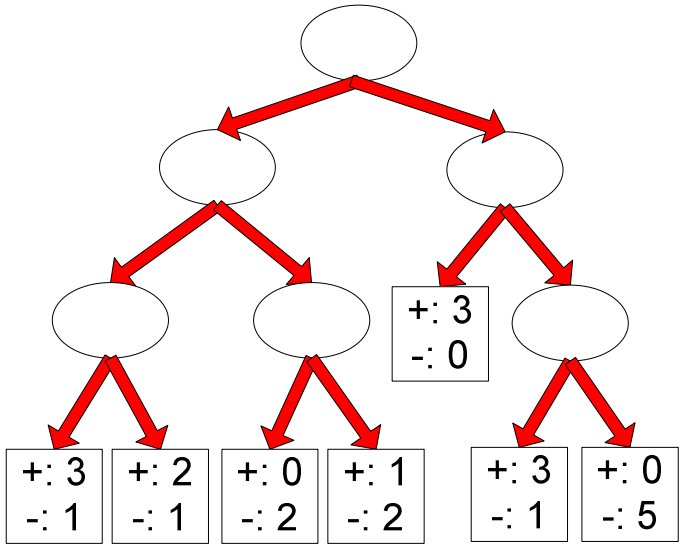
$\Omega = 1$

$e_{gen}(T_L) = 4/24 + 1*7/24 = 11/24 = 0.458$

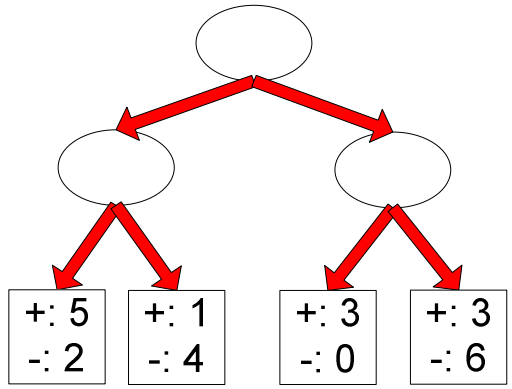
$e_{gen}(T_R) = 6/24 + 1*4/24 = 10/24 = 0.417$

Estimating the Complexity of Decision Trees

- Resubstitution Estimate:
  - Using training error as an optimistic estimate of generalization error
  - Referred to as optimistic error estimate



Decision Tree, T_L



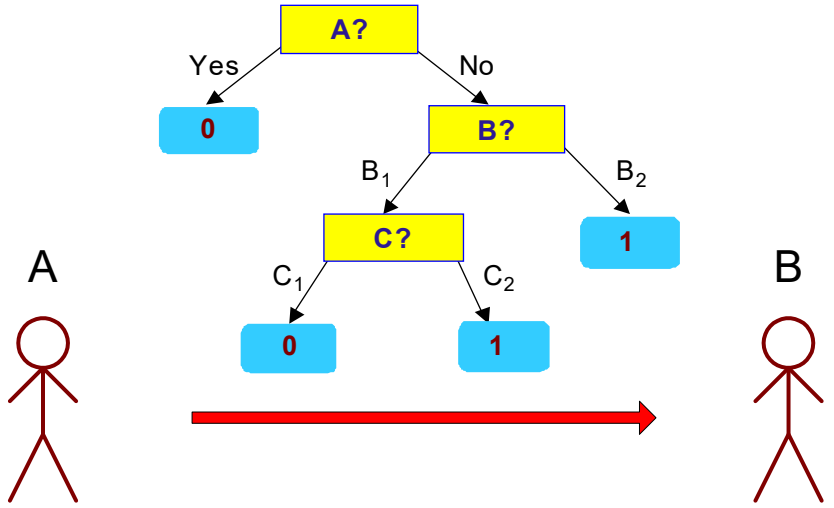
Decision Tree, T_R

$e(T_L) = 4/24$

$e(T_R) = 6/24$

Minimum Description Length (MDL)

X	y
X_1	1
X_2	0
X_3	0
X_4	1
…	…
X_n	1



X	y
X_1	?
X_2	?
X_3	?
X_4	?
…	…
X_n	?

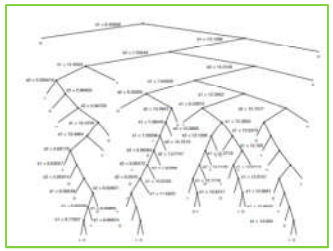
- Cost(Model,Data) = Cost(Data|Model) + α x Cost(Model)
  - Cost is the number of bits needed for encoding.
  - Search for the least costly model.
- Cost(Data|Model) encodes the misclassification errors.
- Cost(Model) uses node encoding (number of children) plus splitting condition encoding.

قبل از اینکه درخت بزرگ شه استاپ کنیم و پیش هرس کنیم

Model Selection for Decision Trees

مثلا ما از بزرگ شدن درختمون راضی نیستیم باید هرسش کنیم مثلا بعضی جاها خوب نیست همون اول کار درخت های کوچیک برای مدلمون بسازیم و بهتره یه درخت بزرگ بسازیم بعد هرسش کنیم

- Pre-Pruning (Early Stopping Rule)
  - Stop the algorithm before it becomes a fully-grown tree
  - Typical stopping conditions for a node:
    - Stop if all instances belong to the same class
    - Stop if all the attribute values are the same
  - More restrictive conditions:
    - Stop if number of instances is less than some user-specified threshold
    - Stop if class distribution of instances are independent of the available features (e.g., using χ^2 test)
    - Stop if expanding the current node does not improve impurity measures (e.g., Gini or information gain).
    - Stop if estimated generalization error falls below certain threshold



قبل از هرس (قانون توقف زودهنگام)
– الگوریتم را قبل از اینکه به درختی کاملاً رشد کرده تبدیل شود متوقف کنید
- شرایط توقف معمول برای یک گره: اگر همه نمونه ها به یک کلاس تعلق دارند، توقف کنید
اگر همه اتریبیوت ها مقادیر یکسان دارند، توقف کنید

- شرایط محدودتر: اگر تعداد نمونه ها کمتر از آستانه تعیین شده توسط کاربر باشد، متوقف شود
اگر توزیع کلاس نمونه ها مستقل از ویژگیهای موجود باشد، متوقف شود.
اگر گسترش گره فعلی معیارهای ناخالصی را بهبود نمی بخشد (به عنوان مثال، جینی یا افزایش اطلاعات) متوقف شود.
اگر خطای تعمیم تخمینی زیر آستانه معین قرار گرفت، توقف کنید

Model Selection for Decision Trees

اجازه بده درخت خیلی بزرگ شه بعد یال ها و برگهای اضافه رو هرس کن و حذفشون کن

- Post-pruning
  - Grow decision tree to its entirety
  - Subtree replacement
    - Trim the nodes of the decision tree in a bottom-up fashion
    - If generalization error improves after trimming, replace sub-tree by a leaf node
    - Class label of leaf node is determined from majority class of instances in the sub-tree

پس از هرس
- درخت تصمیم را به طور کامل رشد دهید
– جایگزینی درخت فرعی
گره های درخت تصمیم را به صورت پایین به بالا برش دهید
اگر خطای تعمیم پس از اصلاح بهبود یافت، درخت فرعی را با یک گره برگ جایگزین کنید
برچسب کلاس گره برگ از کلاس اکثر نمونه ها در زیر درخت تعیین می شود.

Example of Post-Pruning

misclassification error = 1- max(pi(t))
error = 1- max(20/30,10/30) = 1- 20/30 = 10/30

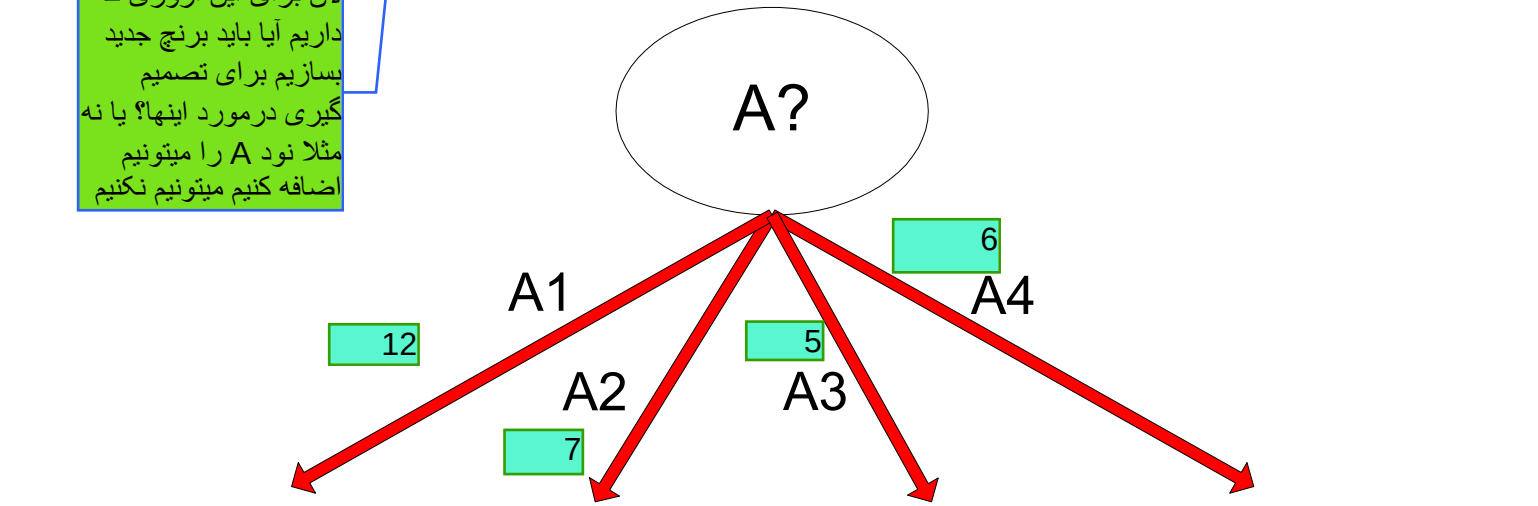
Training Error (Before splitting) = 10/30

Pessimistic error = (10 + 0.5)/30 = 10.5/30

Class = Yes	20
Class = No	10
Error = 10/30	

اگه عمده ی لیبل ها را درنظر بگیریم که ۲۰تا یس داریم پس کل را یس میگیریم پس خطا میشه 10/30
چون ۱۰تا از داده های اموزشمون را غلط گفتیم

لان برای این اروری که داریم آیا باید برنچ جدید بسازیم برای تصمیم گیری درمورد اینها؟ یا نه مثلا نود A را میتونیم اضافه کنیم میتونیم نکنیم



Class = Yes	8
Class = No	4

برچسب این شاخه چی میشه؟
yes
چون اکثریت یس هستند

Class = Yes	3
Class = No	4

برچسب این شاخه:no

Class = Yes	4
Class = No	1

label = yes

Class = Yes	5
Class = No	1

label = yes

Example of Post-Pruning

Class = Yes	20
Class = No	10
Error = 10/30	

Training Error (Before splitting) = 10/30

Pessimistic error = (10 + 0.5)/30 = 10.5/30

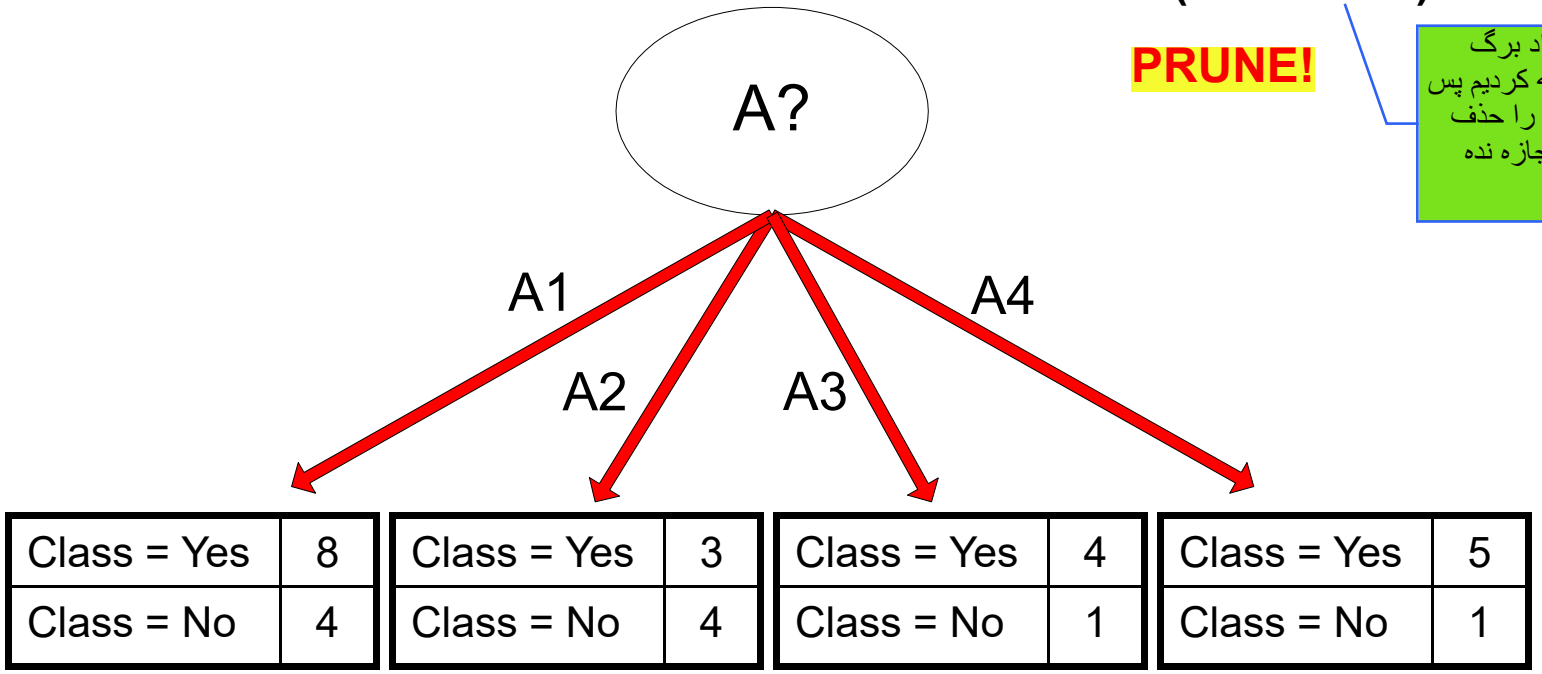
Training Error (After splitting) = 9/30

Pessimistic error (After splitting)

= (9 + 4 × 0.5)/30 = 11/30

PRUNE!

جریمه ی تعداد برگ
هایی که اضافه کردیم پس
میگیم این نود را حذف
کن یا اصلا اجازه نده
اضافه شه



Class = Yes	8
Class = No	4

Class = Yes	3
Class = No	4

Class = Yes	4
Class = No	1

Class = Yes	5
Class = No	1

Examples of Post-pruning

Decision Tree:

```
depth = 1 :
|   breadth > 7 : class 1
|   breadth <= 7 :
|   |   breadth <= 3 :
|   |   |   ImagePages > 0.375 : class 0
|   |   |   ImagePages <= 0.375 :
|   |   |   |   totalPages <= 6 : class 1
|   |   |   |   totalPages > 6 :
|   |   |   |   |   breadth <= 1 : class 1
|   |   |   |   |   breadth > 1 : class 0
|   |   width > 3 :
|   |   |   MultiIP = 0:
|   |   |   |   ImagePages <= 0.1333 : class 1
|   |   |   |   ImagePages > 0.1333 :
|   |   |   |   |   breadth <= 6 : class 0
|   |   |   |   |   breadth > 6 : class 1
|   |   |   MultiIP = 1:
|   |   |   |   TotalTime <= 361 : class 0
|   |   |   |   TotalTime > 361 : class 1
depth > 1 :
|   MultiAgent = 0:
|   |   depth > 2 : class 0
|   |   depth <= 2 :
|   |   |   MultiIP = 1: class 0
|   |   |   MultiIP = 0:
|   |   |   |   breadth <= 6 : class 0
|   |   |   |   breadth > 6 :
|   |   |   |   |   RepeatedAccess <= 0.0322 : class 0
|   |   |   |   |   RepeatedAccess > 0.0322 : class 1
|   MultiAgent = 1:
|   |   totalPages <= 81 : class 0
|   |   totalPages > 81 : class 1
```

Subtree Raising

Subtree Replacement

Simplified Decision Tree:

```
depth = 1 :
|   ImagePages <= 0.1333 : class 1
|   ImagePages > 0.1333 :
|   |   breadth <= 6 : class 0
|   |   breadth > 6 : class 1
depth > 1 :
|   MultiAgent = 0: class 0
|   MultiAgent = 1:
|   |   totalPages <= 81 : class 0
|   |   totalPages > 81 : class 1
```

Suppose we have a dataset of 100 patients and their corresponding symptoms and diagnoses. We split this dataset into a training set of 80 patients and a validation set of 20 patients.

We use the training set to build a decision tree with 5 levels, which achieves 95% accuracy on the training data. However, we suspect that this tree may be overfitting, so we decide to perform post-pruning.

To do this, we start at the bottom of the tree and evaluate the cost complexity of pruning each leaf node using an alpha parameter. For simplicity, let's say that each leaf node has a cost complexity of 1.

We calculate the total cost complexity of the original tree by summing the individual cost complexities of all the leaf nodes, which is equal to 10 (since there are 10 leaf nodes in the tree).

Next, we try out different values of alpha and evaluate the validation accuracy of the pruned tree for each alpha value. Let's say we get the following results:

Alpha = 0: Pruned tree still has 5 levels, validation accuracy = 92%
Alpha = 1: Pruned tree has 4 levels, validation accuracy = 93%
Alpha = 2: Pruned tree has 3 levels, validation accuracy = 94%
Alpha = 3: Pruned tree has 2 levels, validation accuracy = 92%
Alpha = 4: Pruned tree has 1 level, validation accuracy = 90%
Based on these results, we choose an alpha value of 2, which gives us the simplest tree with the highest validation accuracy (94%). This new pruned tree has 3 levels and is our final decision tree.

Note that in practice, we might use cross-validation to evaluate the accuracy of the pruned tree more robustly and avoid overfitting on the validation set.

Let's say we have a decision tree with 1000 nodes, and it was trained using a dataset containing 10,000 samples. We want to use post-pruning to reduce the size of the decision tree while maintaining its accuracy.

To do this, we would first split our dataset into two parts: a training set and a validation set. Let's use an 80-20 split, which means that 8,000 samples would be used for training, and 2,000 samples would be used for validation.

We would then train our decision tree using the training set and perform pruning on the resulting tree using the validation set. One common post-pruning technique is reduced error pruning, which works as follows:

Starting at the leaves of the decision tree, replace each subtree with its majority class.
Evaluate the accuracy of the pruned tree on the validation set.
If the accuracy has improved, keep the pruned tree. Otherwise, restore the original subtree.
We repeat steps 1-3 until we can no longer improve the accuracy on the validation set.

Let's say that after pruning, our decision tree has 800 nodes instead of 1000. We can then test the accuracy of the pruned tree on a test set, which is a separate dataset from the training and validation sets. This will give us an estimate of how well our decision tree will perform on new, unseen data.

differences between training set evaluation and testing set evaluation

Let's say we're building a machine learning model to classify images of cats and dogs. We have a dataset containing 10,000 images, with 5,000 images of cats and 5,000 images of dogs. We split the dataset into two parts: a training set containing 8,000 images (4,000 cats and 4,000 dogs) and a testing set containing 2,000 images (1,000 cats and 1,000 dogs).

During training, we use the training set to optimize the parameters of the model. For example, we might use an algorithm like logistic regression or a neural network, and we adjust the weights of the model until it produces accurate predictions on the training set.

After training, we want to evaluate the performance of the model on new, unseen data. This is where the testing set comes in. We use the testing set to measure how well our model generalizes to new data that it has not seen before. We can calculate metrics like accuracy, precision, recall, and F1 score based on the predictions the model makes on the testing set.

The key difference between training set evaluation and testing set evaluation is that training set evaluation measures how well the model fits the training data, while testing set evaluation measures how well the model generalizes to new, unseen data. It's important to perform both types of evaluation to ensure that the model is not overfitting to the training data and that it will perform well on real-world examples.

Model Evaluation

ارزیابی مدل

هدف:
- تخمین عملکرد کلسیفایر بر روی داده های دیده نشده قبلی (مجموعه تست)
Holdout
اعتبار سنجی متقابل

Purpose:

- To estimate performance of classifier on previously unseen data (test set)

- Holdout
- Cross validation

Holdout is a method for evaluating machine learning models that involves splitting the available data into two sets: a training set and a validation set (also called a holdout set or testing set).

In a holdout evaluation, we randomly split the available data into two parts, typically using a 70-30 or 80-20 split. The larger portion of the data is used to train the model, while the smaller portion is reserved for validation.

Once the model has been trained on the training set, it is evaluated on the validation set. The performance of the model on the validation set gives us an estimate of how well the model will perform on new, unseen data.

The advantage of using a holdout evaluation is that it provides an unbiased estimate of the model's performance, since the validation set was not used during training. This can help us avoid overfitting, which occurs when the model performs well on the training set but poorly on new data.

However, one potential disadvantage of holdout evaluation is that it may not be as statistically robust as other methods, such as cross-validation, especially if the dataset is small. In addition, the specific split of the data into training and validation sets can have a large impact on the estimated performance of the model. To address this issue, it's common to perform multiple rounds of holdout evaluation with different random splits of the data and average the results.

Overall, holdout evaluation is a useful and widely-used method for evaluating machine learning models, but it should be used in conjunction with other methods and best practices to obtain a more comprehensive performance assessment of the model.

Sure, here's an example of how to use holdout for evaluating a machine learning model in Python:

```python
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

# Load data
X, y = load_data()

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train a logistic regression model on the training set
model = LogisticRegression()
model.fit(X_train, y_train)

# Evaluate the model on the testing set
score = model.score(X_test, y_test)
print(f"Accuracy: {score}")
```

In this example, we first load our data (not shown). We then split the data into a training set and a testing set using `train_test_split` from the `sklearn.model_selection` module. Here, we're using 20% of the data as our testing set.

Next, we train a logistic regression model on the training set with `LogisticRegression` from the `sklearn.linear_model` module. Finally, we evaluate the model on the testing set by computing the accuracy with `score`.

In the example I provided, the testing set and the validation set are actually the same thing.

The `train_test_split()` function from scikit-learn is commonly used to split a dataset into two parts: a training set and a testing set. The testing set is then used to evaluate the performance of the model on unseen data after it has been trained on the training set.

However, in some cases, we may want to tune the hyperparameters of our model or perform feature selection using a third subset of the data, which is called a validation set. In this case, we would typically split the original dataset into three parts: a training set, a validation set, and a testing set.

To split the dataset into three parts instead of two, you can modify the code as follows:

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

# Load data
X, y = load_data()

# Split data into training, validation, and testing sets
X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_size=0.25, random_state=42)

# Train a logistic regression model on the training set
model = LogisticRegression()
model.fit(X_train, y_train)

# Evaluate the model on the validation set
val_score = model.score(X_val, y_val)
print(f"Validation Accuracy: {val_score}")

# Evaluate the model on the testing set
test_score = model.score(X_test, y_test)
print(f"Testing Accuracy: {test_score}")
```

In this modified example, we first split the data into a training/validation set (80% of the data) and a testing set (20% of the data). We then split the training/validation set further into a training set (60% of the original data) and a validation set (20% of the original data).

We train the logistic regression model on the training set and evaluate its performance on the validation set by computing the accuracy score. We use this score to tune the hyperparameters of our model or perform feature selection.

Finally, we evaluate the performance of the model on the testing set, which gives us an estimate of how well the model will generalize to new, unseen data.

Cross-validation is a technique used to evaluate the performance of a machine learning model by partitioning the data into multiple subsets (called folds), training the model on some of the folds, and testing it on the remaining fold. The process is repeated multiple times, with a different fold reserved for testing each time. This allows us to obtain more reliable estimates of the model's performance than a single train/test split would provide.
There are several types of cross-validation, but the most common one is k-fold cross-validation. Here's how it works:

The data is partitioned into k equal-sized folds.
For each fold i, the model is trained on the remaining k-1 folds.
The model is tested on the held-out fold i and the performance metric is recorded.
Steps 2-3 are repeated k times, with each fold being used exactly once as the test set.
The CV score is computed by averaging the performance metric over all k folds.
Here's an example of how to perform k-fold cross-validation in Python using scikit-learn:

```
from sklearn.model_selection import KFold
from sklearn.linear_model import LogisticRegression

# Load data
X, y = load_data()

# Define the number of folds
num_folds = 5

# Create a k-fold cross-validation object
kf = KFold(n_splits=num_folds)

# Initialize a list to store the validation scores
val_scores = []
```

```
# Train and validate the model on each fold
for train_idxs, val_idxs in kf.split(X):

    # Split the data into training and validation sets
    X_train, X_val = X[train_idxs], X[val_idxs]
    y_train, y_val = y[train_idxs], y[val_idxs]

    # Train a logistic regression model on the training set
    model = LogisticRegression()
    model.fit(X_train, y_train)

    # Evaluate the model on the validation set
    val_score = model.score(X_val, y_val)
    val_scores.append(val_score)

# Compute the mean validation score over all folds
mean_val_score = np.mean(val_scores)

print(f"Mean cross-validation score: {mean_val_score}")
```

In this example, we first load our data (not shown). We define the number of folds (in this case, 5) and create a KFold object using scikit-learn's KFold function. We then loop over each fold and split the data into training and validation sets, as before. However, instead of computing a single test score, we compute a validation score for each fold and store it in a list. Finally, we compute the mean validation score over all folds.

Stratified cross-validation is a type of cross-validation technique used in machine learning to evaluate the performance of a model. In this technique, the data is divided into several subsets or folds, ensuring that each fold has a representative proportion of the different classes found in the dataset.

For example, let's say we have a dataset with 1000 instances, where 600 are labeled as class A and 400 are labeled as class B. To perform stratified cross-validation, we would split the dataset into several folds, such as 5 or 10, while maintaining the same proportion of classes in each fold. So, if we were using 5 folds, we would ensure that each fold had 120 instances of class A (600/5) and 80 instances of class B (400/5).

This approach ensures that the model is trained and evaluated on data that is representative of the overall distribution of classes in the dataset, which can lead to more accurate and reliable performance metrics.

here is an example of performing stratified cross-validation in Python using the StratifiedKFold function from the sklearn.model_selection module:

```
from sklearn.model_selection import StratifiedKFold
import numpy as np

X = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10]])
y = np.array([0, 0, 1, 1, 1])

skf = StratifiedKFold(n_splits=3)
for train_index, test_index in skf.split(X, y):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
```

In this example, we're creating a dataset with 5 instances and two features, where the first two instances belong to class 0 and the remaining three instances belong to class 1. We're then using StratifiedKFold with n_splits=3 to perform stratified 3-fold cross-validation on the dataset.

The split method of StratifiedKFold returns the indices of the training and testing data for each fold. We're then splitting the data into training and testing sets based on these indices and printing them out for each fold.

Repeated cross-validation is a technique used in machine learning to obtain a more robust estimate of model performance by repeating the cross-validation process multiple times using different random partitions of the data.

In repeated cross-validation, the dataset is randomly split into training and testing sets, and the model is trained and evaluated on each split. This process is then repeated for a specified number of times, with different random splits each time. The results from each repetition are then averaged to obtain an overall performance estimate.

Here's an example of how to perform repeated cross-validation in Python using scikit-learn:

```python
from sklearn.model_selection import cross_val_score, RepeatedKFold
from sklearn.linear_model import LogisticRegression
import numpy as np

X = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10]])
y = np.array([0, 0, 1, 1, 1])

model = LogisticRegression()
cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)
scores = cross_val_score(model, X, y, scoring='accuracy', cv=cv, n_jobs=-1)

print('Mean Accuracy: %.3f' % np.mean(scores))
```

In this example, we're using RepeatedKFold with n_splits=10 and n_repeats=3 to perform 30-fold cross-validation (10 splits, 3 repeats) on the dataset. We're also specifying random_state=1 to ensure reproducibility of the results.

We're using logistic regression as our model and evaluating its accuracy using the accuracy metric. Finally, we're printing the mean accuracy score across all the repetitions.

cross_val_score is a function in scikit-learn that performs cross-validation to evaluate the performance of a machine learning model. It takes a model, input features, target variable, and a number of cross-validation parameters as input, and returns the performance score for each fold of the cross-validation.

The cross_val_score function works by splitting the data into k-folds (or more generally n-folds), where k is specified using the cv parameter. For each fold, it trains the model on k-1 folds of the data and evaluates its performance on the remaining fold, returning the performance score for that fold. This process is repeated k times, with each fold being used as the test set once, resulting in k performance scores.

These individual performance scores can then be aggregated in various ways, such as taking their average or computing their standard deviation, to obtain an overall estimate of the model's performance.

Here is an example of how to use cross_val_score in Python to evaluate the performance of a logistic regression model using 10-fold cross-validation:

```python
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
import numpy as np

X = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10]])
y = np.array([0, 0, 1, 1, 1])

model = LogisticRegression()
scores = cross_val_score(model, X, y, cv=10)

print('Cross-Validation Scores:', scores)
print('Mean Performance Score:', np.mean(scores))
```

In this example, we're using cross_val_score to perform 10-fold cross-validation on a logistic regression model trained on the dataset X with corresponding labels y. The function returns an array of 10 performance scores, one for each fold of the cross-validation. We're printing these scores and also computing their mean to obtain an overall estimate of the model's performance.

Model Evaluation:Holdout

- Holdout
  - Reserve k% for training and (100-k)% for testing
  - Random subsampling: repeated holdout

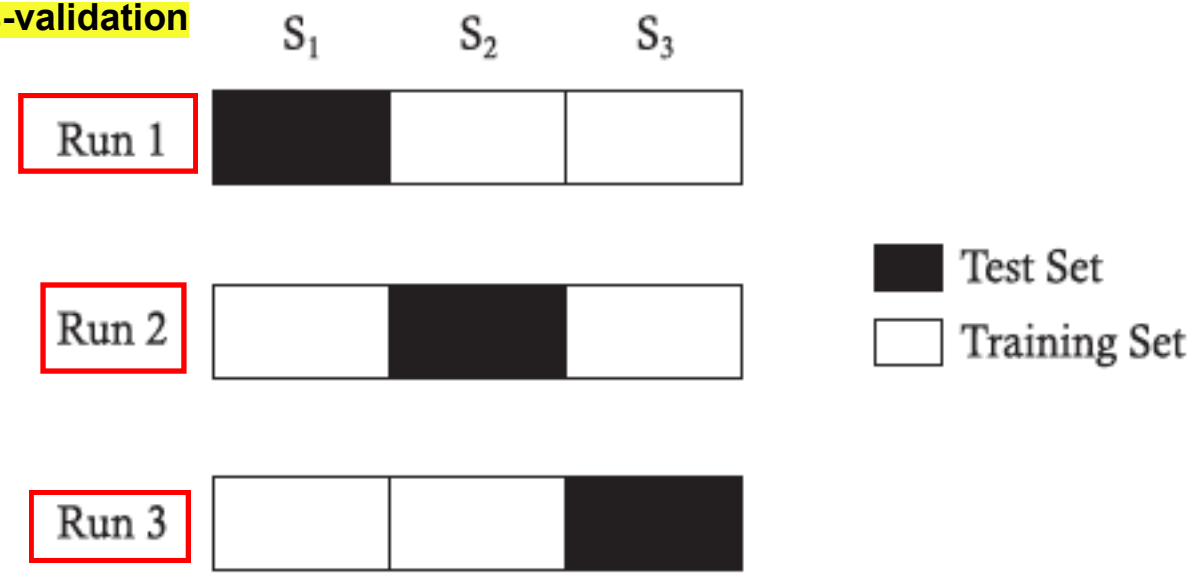
اینکه این k چی باشه رو
میشه به صورت رندم
انتخاب کرد چندین بار

Model Evaluation:Cross-validation

- Cross validation
  - Partition data into k disjoint subsets
  - k-fold: train on k-1 partitions, test on the remaining one
  - Leave-one-out: k=n

تعداد رکوردها یا تعداد ابجکت هامون

3-fold cross-validation



چون ۳تا پارتیشن داریم سه تا حالت خواهیم داشت بعد سه تا خطا بدست میاد که میشه بین این سه تا میانگین بگیریم

تغییرات در اعتبار سنجی متقابل

Variations on Cross-validation

اعتبارسنجی متقابل مکرر
- چند بار اعتبار سنجی متقابل را انجام دهید
– تخمینی از واریانس خطای تعمیم می دهد

مدام تکرار کنیم که یه مشکلی پیش میاد مثلا اگه داده هامون دو کلاسه باشن، برچسب هاشون متوازن نباشه ینی همون قدر که داده ی کلاس یک داریم داده ی کلاس منهای یک نداشته باشیم وقتی داریم داده هامون رو پارتیشن بندی میکنیم ممکنه عمده ش توی یک کلاس باشه و تعداد کمیش توی یه کلاس بره حتی ممکنه توی یک پارتیشن همه توی یک کلاس برن و توی اون پارتیشن اصلا کلاس دوم را نداشته باشیم

- Repeated cross-validation
  - Perform cross-validation a number of times
  - Gives an estimate of the variance of the generalization error

راه حل اینکه طوری پارتیشن هارا انتخاب کنیم که نسبت توزیع کلاس ها در پارتیشن ها رعایت شده باشه
نسبت هردوکلاس در پارتیشن ها رعایت بشه

- Stratified cross-validation

طبقه بندی شده

  - Guarantee the same percentage of class labels in training and test
  - Important when classes are imbalanced and the sample is small
- Use nested cross-validation approach for model selection and evaluation

اعتبار سنجی متقاطع طبقه بندی شده
- ضمانت درصد یکسان از برچسب کلاس در آموزش و آزمون
- زمانی که کلاس ها نامتعادل هستند و نمونه کوچک است مهم است

از روش اعتبارسنجی متقابل تو در تو برای انتخاب و ارزیابی مدل استفاده کنید