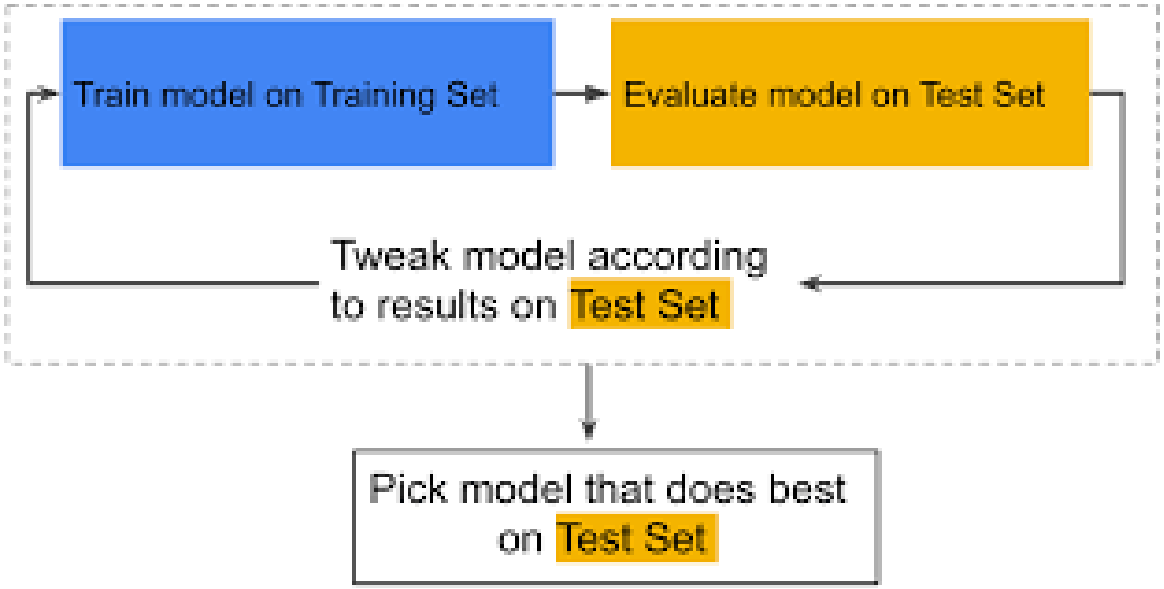
Classification Model Evaluation

Resources:
https://www.evidentlyai.com/classification-metrics/confusion-matrix

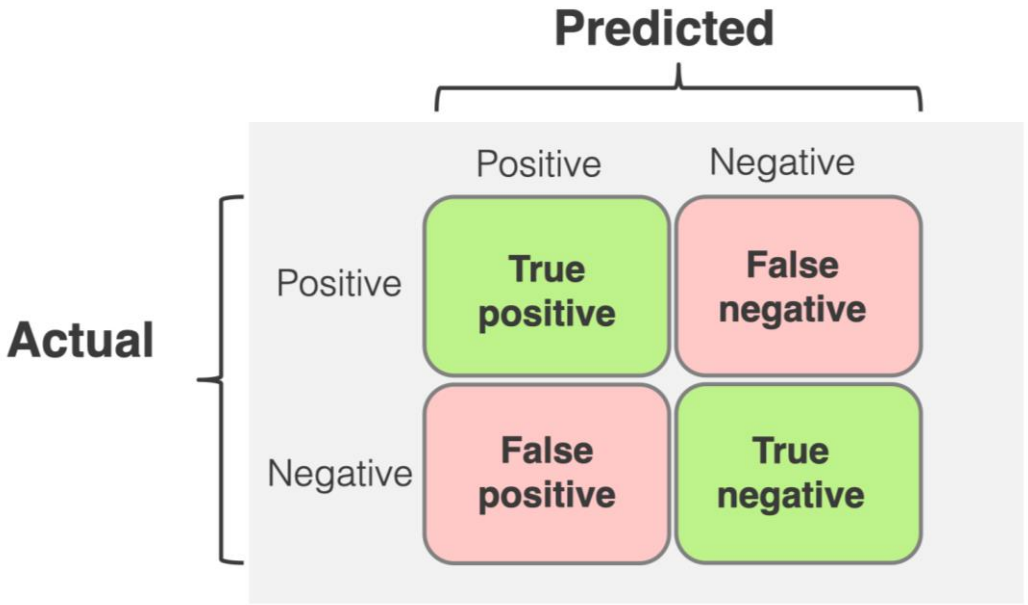
Defining Training and Test Set Splits



- The model is trained on the **training set** (aka the train set), typically 70 to 80% of the data
- The remaining 20 to 30% of data that is **unseen** by the model during the training phase is called the **test set**
- Test set serves mimics the real-world scenario of running the model to predict new data points

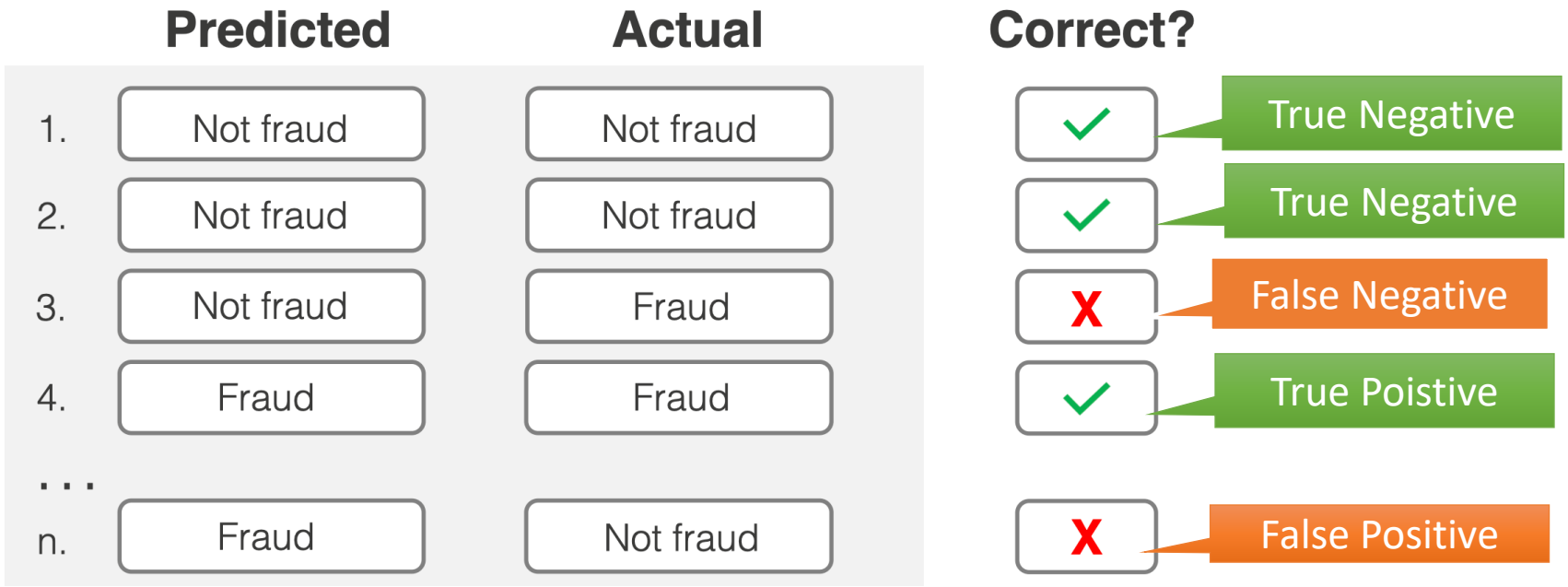
Evaluating Classifier Accuracy

- **What to Evaluate?**
  - Accuracy is measured in terms of error rate
  - Details of errors are shown in a confusion matrix



- A confusion matrix is a **table** that summarizes the performance of a classification model
- It shows the number of **correct predictions**: true positives (TP) and true negatives (TN)
- It also shows the **model errors**: false positives (FP) are “false alarms,” and false negatives (FN) are missed cases
- Using TP, TN, FP, and FN, you can calculate various classification quality metrics, such as precision and recall

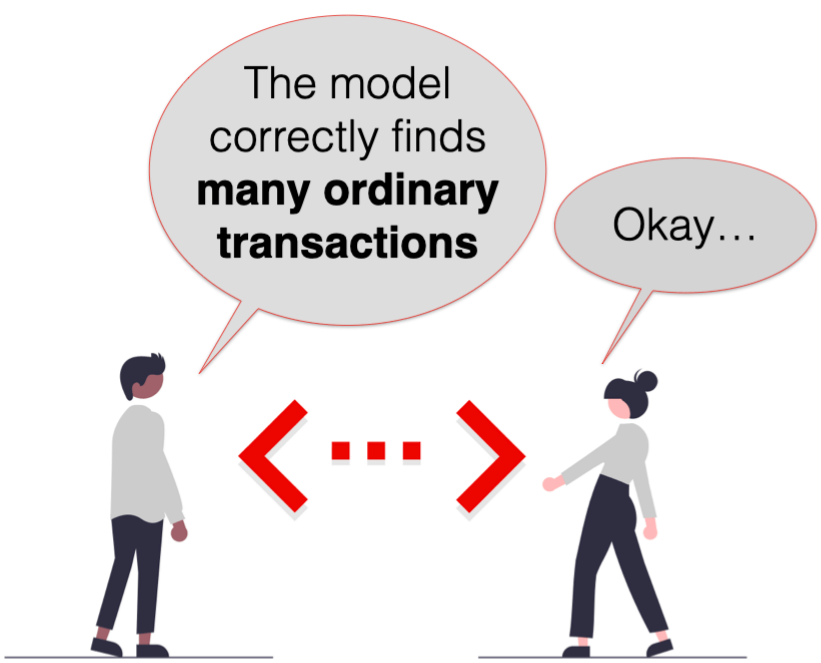
False Positive vs False Negative



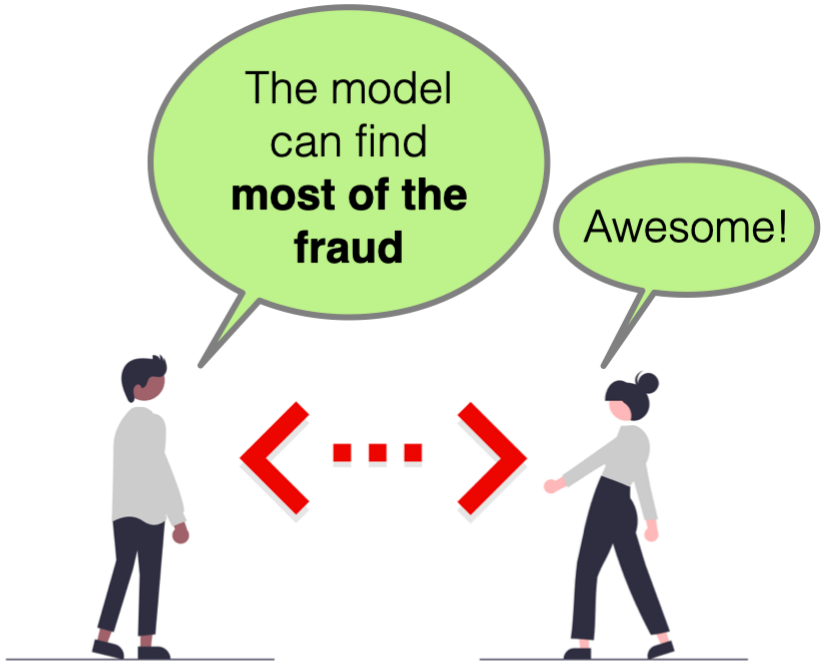
- There are two types of errors the model can make:
  - It can make a **false alarm** and label an ordinary transaction as fraudulent (False Positive)
  - It can **miss real fraud** and label a fraudulent transaction as ordinary (False Negative)

True Positive vs True Negative

- How well the model can identify fraudulent transactions rather than how often the model is right overall



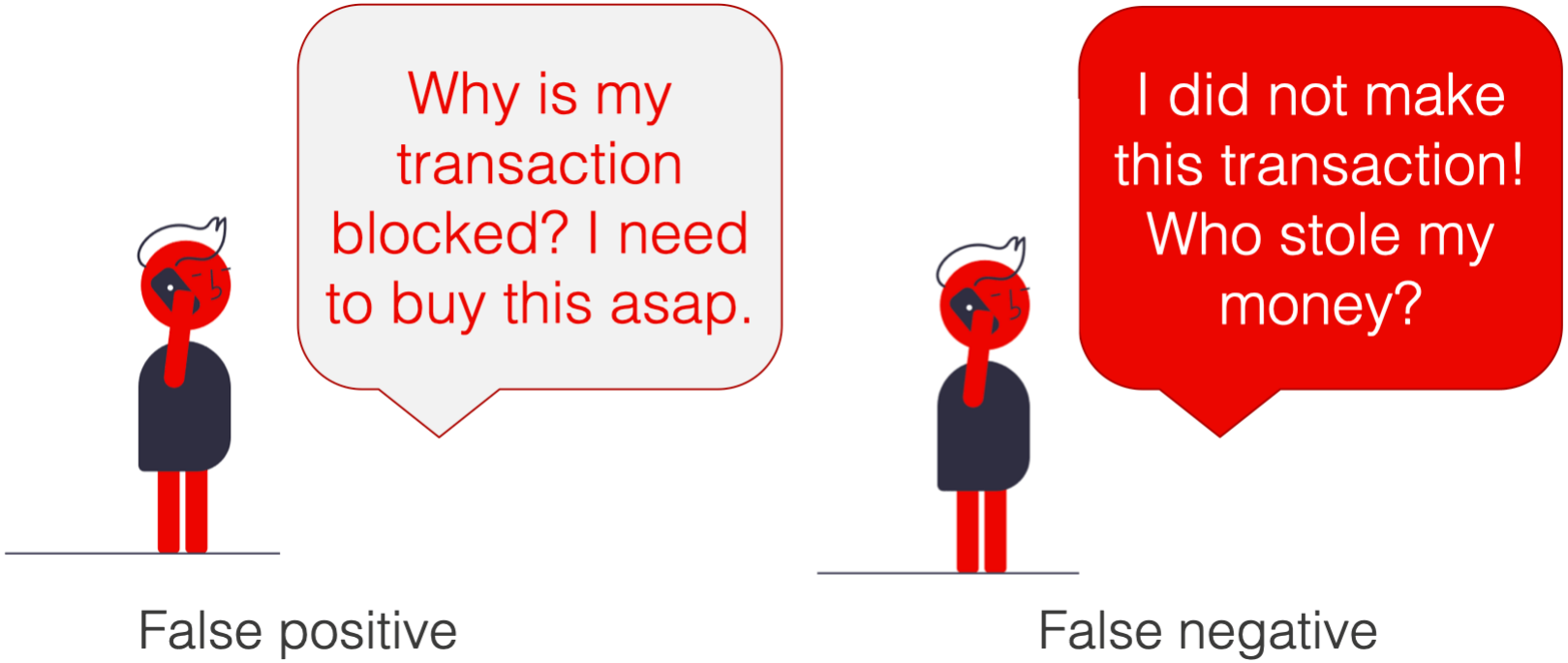
True negative



True positive

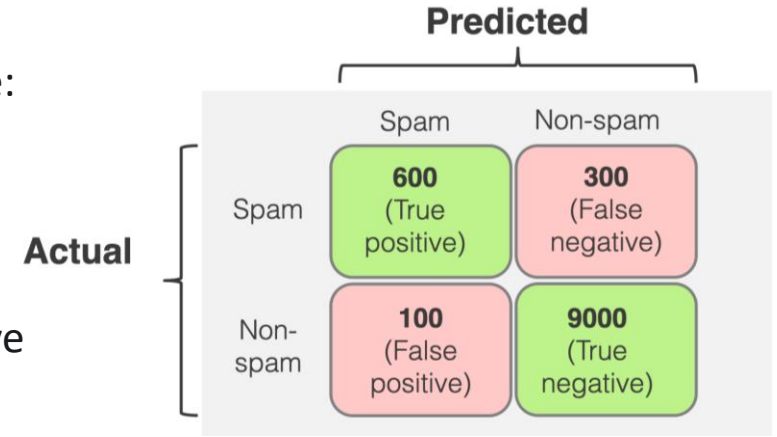
False Positive vs False Negative

- The distinction between false positives and negatives is important because the consequences of errors are different. You might consider one error less or more harmful than the other
  - In this scenario, False negative is more harmful

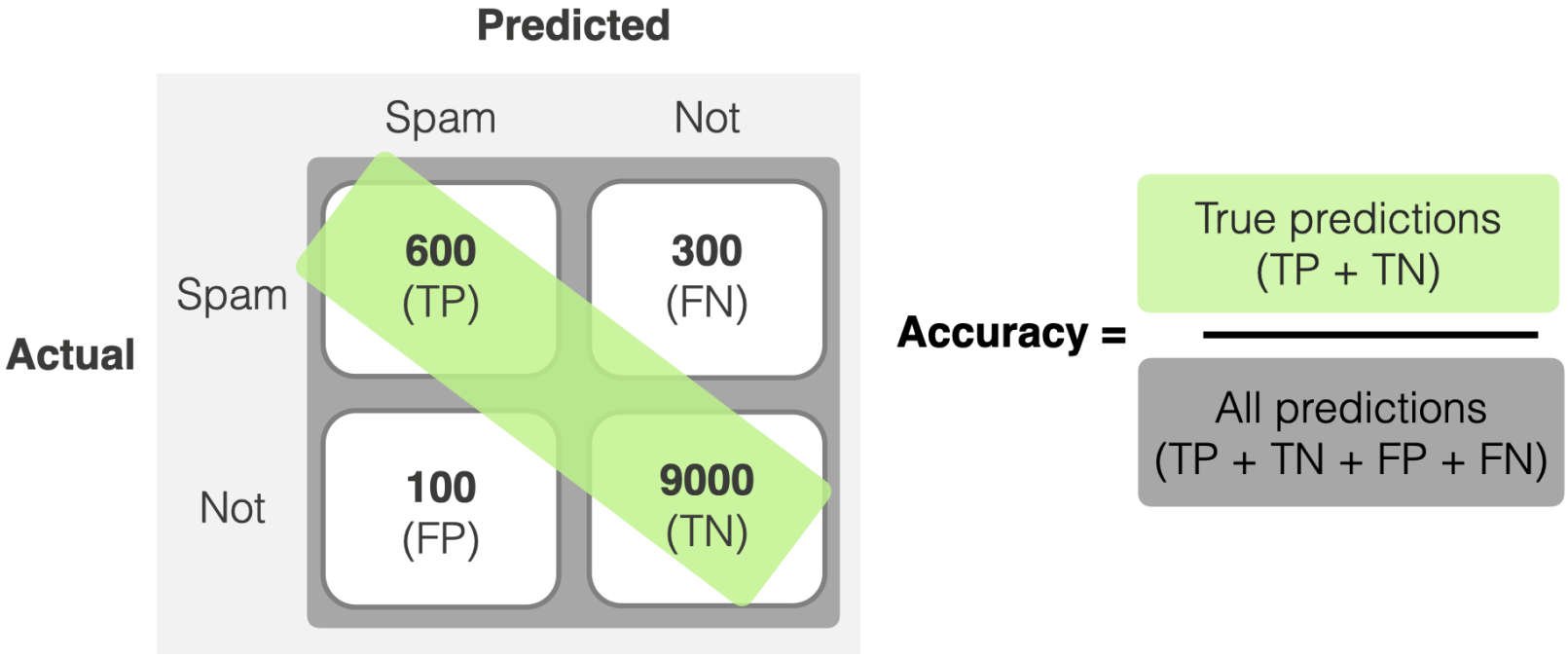


Spam Classifier

- **True Positive (TP)**
  - Number of correctly identified positive cases: the number of correctly predicted spam emails (true positives is 600)
- **True Negative (TN)**
  - It shows the number of correctly identified negative: the number of correctly predicted non-spam emails (true negatives is 9000)
- **False Positive (FP):**
  - It shows the number of incorrectly predicted positive cases: these are **false alarms**
  - This is the number of emails incorrectly labeled as spam (number of false positives is 100)
- **False Negative (FN):**
  - It shows the number of incorrectly predicted negative cases: these are **missed cases**
  - This is the number of missed spam emails that made their way into the primary inbox (false negatives is 300)

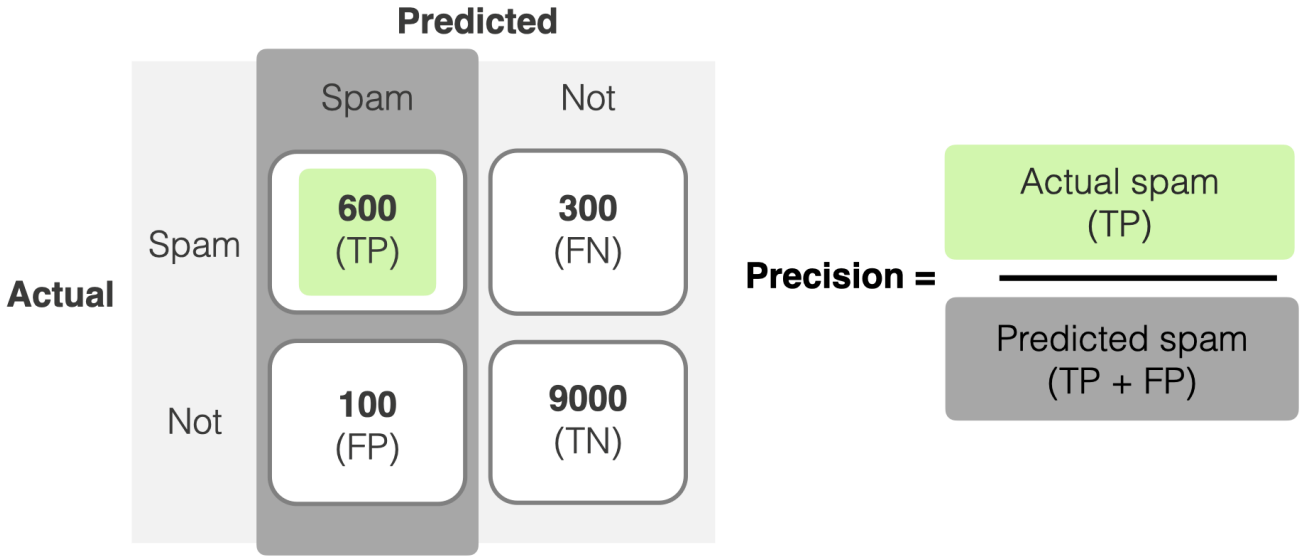


Accuracy



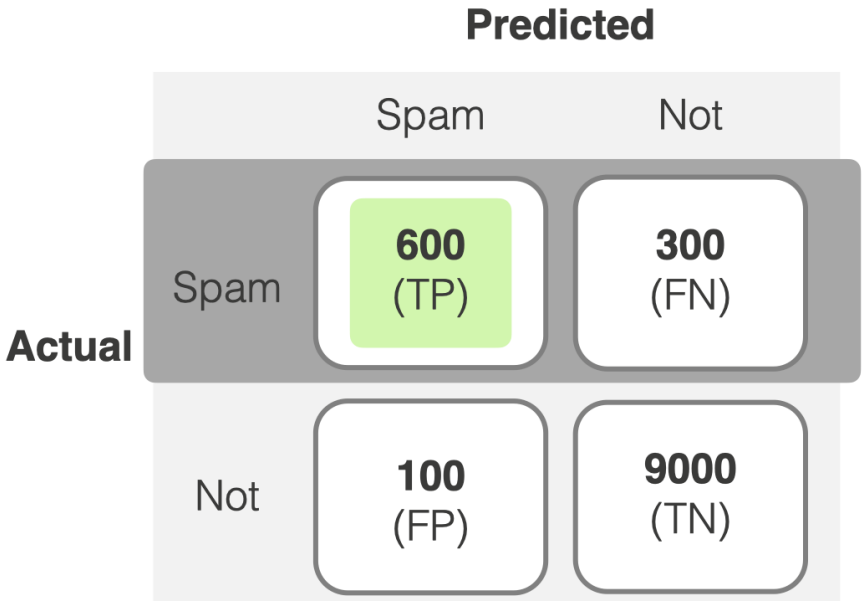
- Calculate accuracy by dividing all true predictions by the total number of predictions
- In our example, accuracy is (9000+600)/10000 = 0.96. The model was correct in 96% of cases
- However, accuracy can be misleading for imbalanced datasets when one class has significantly more samples (e.g., we have many non-spam emails: 9100 out of 10000 are regular emails)

Precision



- Precision is the % of **true positive predictions** in all positive predictions
  - It shows how often the model is right when it predicts the target class
- Calculate precision by dividing the **correctly identified positives** by the total number of positive predictions made by the model
- In our example, precision is 600/(600+100)= 0.86. When predicting "spam" the model was correct in 86% of cases
- Precision is a good metric when the **cost of false positives is high**
  - If you prefer to avoid sending good emails to spam folders, you might want to focus primarily on precision

Recall



Recall = Actual spam (TP) / All spam (TP + FN)

- Recall, or **true positive rate (TPR)** shows the % of true positive predictions made by the model out of all positive samples in the dataset
  - i.e., the recall shows % instances of the target class the model can find
- Calculate the recall by dividing the number of true positives by the total number of positive cases
- In our example, recall is 600/(600+300)= 0.67. The model correctly found 67% of spam emails. The other 33% made their way to the inbox
- Recall is a helpful metric when the **cost of false negatives is high**
  - E.g., you can optimize for recall for the classifier of Fraudulent transactions

F1 Score

$$\text{F1 Score} = \frac{2 \times (\text{Precision} \times \text{Recall})}{\text{Precision} + \text{Recall}}$$

- The F1 score is the harmonic mean of precision and recall. It provides a single metric that balances both precision and recall
- The F1 score reaches its best value at 1 (perfect precision and recall) and worst at 0.
- The F1 score is particularly useful in situations where there is an imbalance between the number of positive and negative instances in the dataset
  - It provides a more informative evaluation of the model's performance compared to accuracy alone, especially when the classes are unevenly distributed

Evaluating Classifier Accuracy

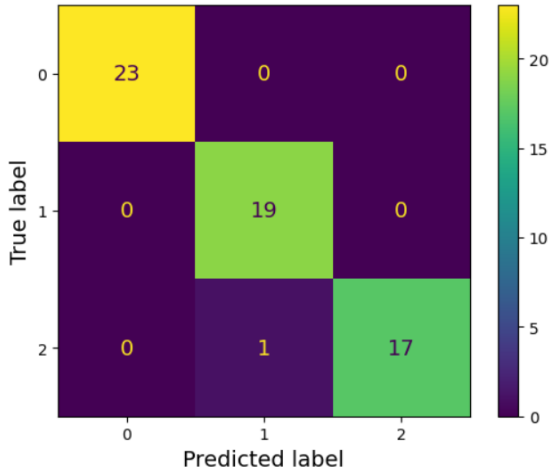
```
y_pred = tree_clf.predict(X_test)
print("Predicted Labels:", y_pred)  #Predicted labels the testing points
print("True Labels:     ", y_test) #True labels
print("Testing Accuracy:", metrics.accuracy_score(y_test, y_pred))  #1 mistake .. 59/60 = 0.983
```

```
Predicted Labels: [1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 2 0 0 0 0 1 0 0 2 1
 0 0 0 2 1 1 0 0 1 1 2 1 2 1 2 1 0 2 1 0 0 0 1]
True Labels:      [1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 2 0 0 0 0 1 0 0 2 1
 0 0 0 2 1 1 0 0 1 2 2 1 2 1 2 1 0 2 1 0 0 0 1]
Testing Accuracy: 0.9833333333333333
```

```
from sklearn.metrics import confusion_matrix
confusion_matrix(y_test, y_pred)  # 1 wrong prediction is made .. Accuracy
```

```
array([[23,  0,  0],
       [ 0, 19,  0],
       [ 0,  1, 17]], dtype=int64)
```

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
cm = confusion_matrix(y_test, y_pred)  # 1 wrong prediction is made .. Accuracy
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=tree_clf.classes_)
disp.plot()
plt.show()
```



Evaluating Classifier Accuracy

- **How to Evaluate?**
  - Normally, a separate independently sampled test set is used to measure accuracy of a classification model
  - Evaluation methods:
    - Holdout Method: divide data set (e.g., 60-40, or 2/3-1/3) as training and test sets

In [6]:
```
#Take 40% of the data as testing and the remaining 60% as training
X_train, X_test, y_train, y_test = train_test_split(X_iris, y_iris, test_size=0.4, random_state=42)
y_train = y_train.to_numpy()
y_test = y_test.to_numpy()
print("Training Data", X_train.shape)
print("Testing Data", X_test.shape)  #60 points for testing
```

```
Training Data (90, 2)
Testing Data (60, 2)
```

Decision tree trained on X_train

In [7]:
```
tree_clf = DecisionTreeClassifier(random_state=42)
tree_clf.fit(X_train, y_train)
# Accuracy on the training dataset using the score method
print("Training Accuracy:", tree_clf.score(X_train, y_train))
```

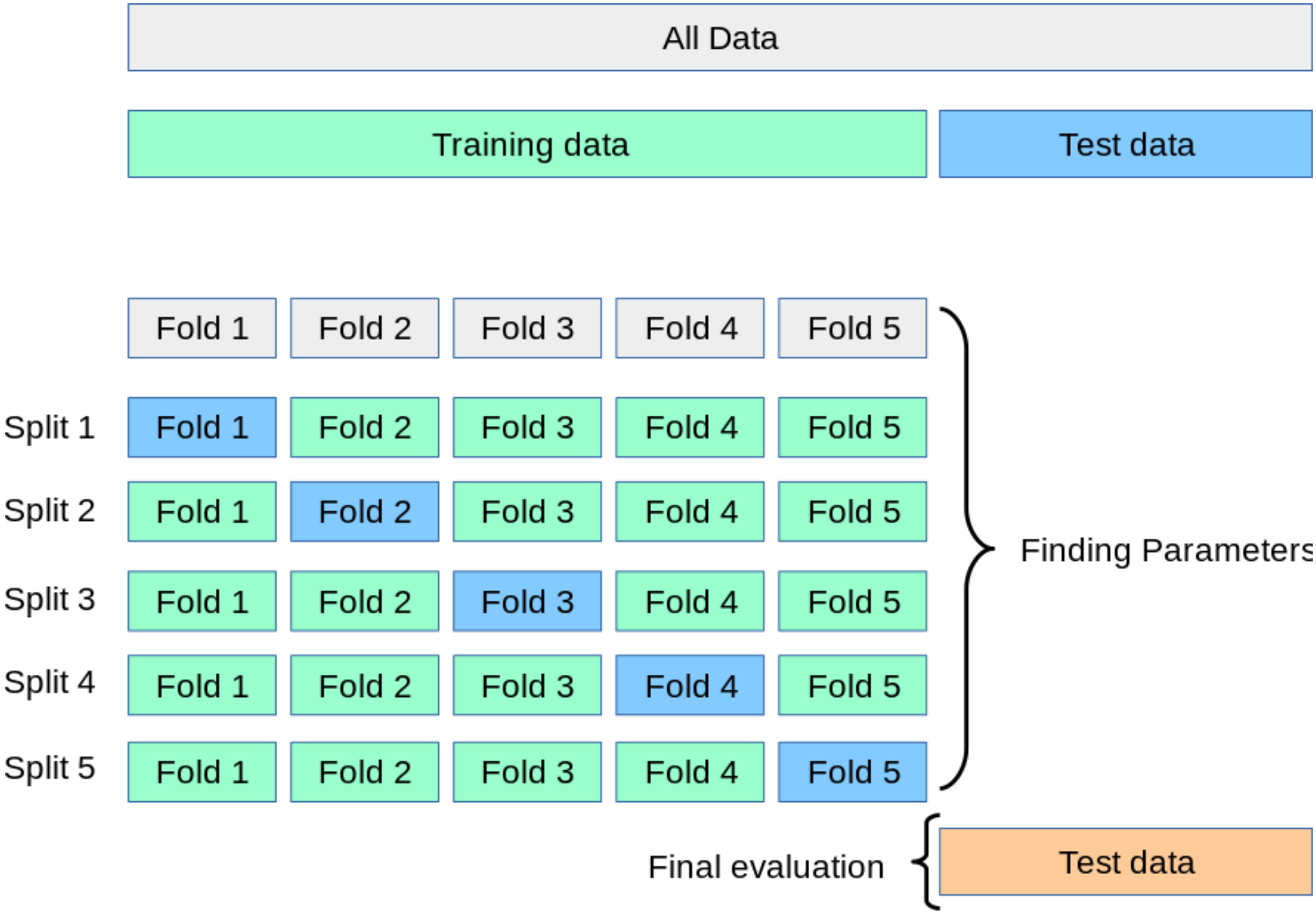
```
Training Accuracy: 0.9888888888888889
```

Making Prediction

In [8]:
```
y_pred = tree_clf.predict(X_test)
print("Predicted Labels:", y_pred)  #Predicted labels the testing points
print("True Labels:     ", y_test) #True labels
print("Testing Accuracy:", metrics.accuracy_score(y_test, y_pred))  #1 mistake .. 59/60 = 0.983
```

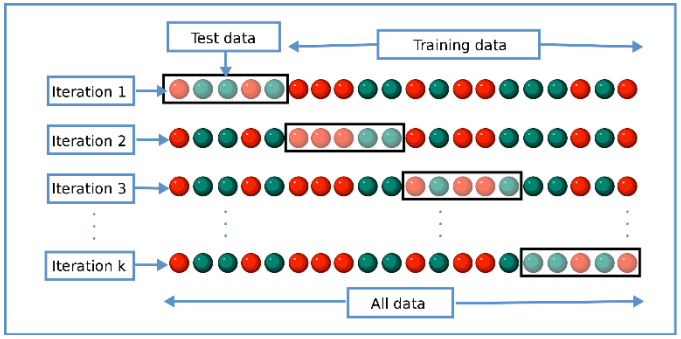
```
Predicted Labels: [1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1
 0 0 0 2 1 1 0 0 1 1 2 1 2 1 2 1 0 2 1 0 0 0 1]
True Labels:      [1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1
 0 0 0 2 1 1 0 0 1 2 2 1 2 1 2 1 0 2 1 0 0 0 1]
Testing Accuracy: 0.9833333333333333
```

Cross-validation



Evaluating Classifier Accuracy

- **How to Evaluate?**
  - Normally, a separate independently sampled test set is used to measure accuracy of a classification model
  - Evaluation methods:
    - Cross Validation: the data set is divided into k equal-size partitions. For each round of decision tree induction, one partition is used for testing and the rest used for training. After k rounds, the average error rate is used.
      - Leave-one-out: a special case for small size data sets.



Cross Validation

Every single point is used for testing once

In [12]:
```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(tree_clf, X_iris, y_iris, cv=5)
print(scores)

print("%0.2f accuracy with a standard deviation of %0.2f" % (scores.mean(), scores.std()))
```

```
[0.96666667 0.96666667 0.9        0.96666667 1.        ]
0.96 accuracy with a standard deviation of 0.03
```

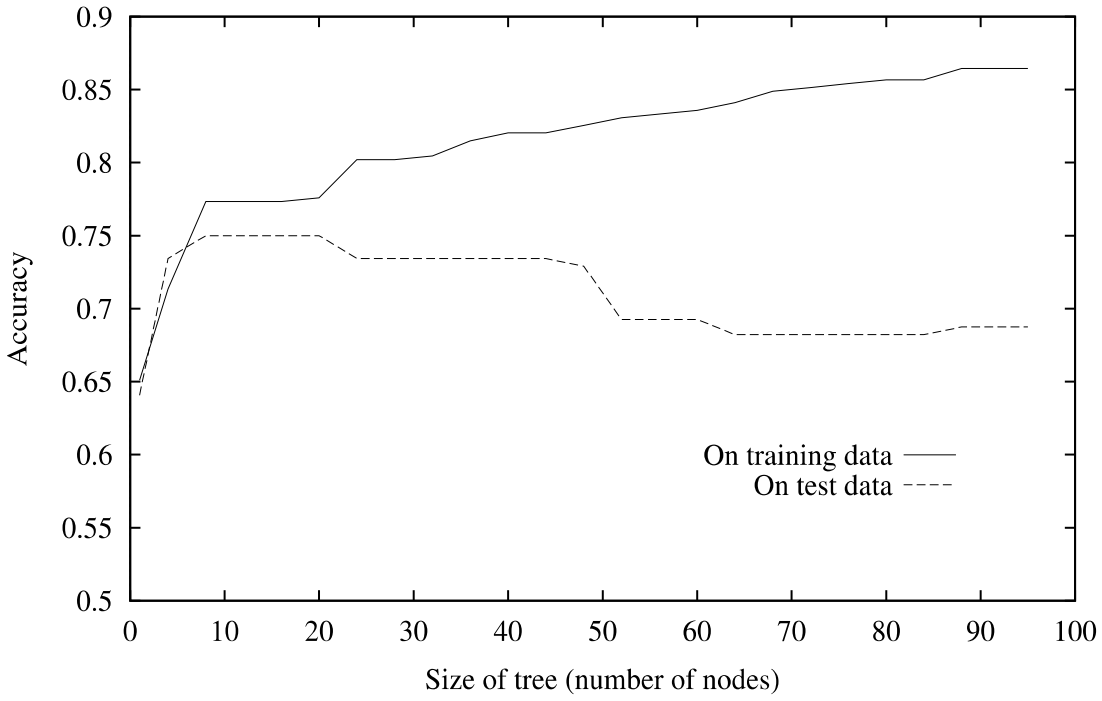
In [18]:
```
#50 cross validation. Each time, train on 147 and test on 3 samples
scores = cross_val_score(tree_clf, X_iris, y_iris, cv=50)
print(scores.round(2))

print("%0.2f accuracy with a standard deviation of %0.2f" % (scores.mean(), scores.std()))
```

```
[1.   1.   1.   1.   1.   1.   0.67 1.   1.   1.   1.   1.   1.   1.
 1.   1.   1.   1.   1.   0.67 0.67 1.   1.   1.   1.   1.   0.67 0.67
 1.   0.67 1.   1.   1.   0.67 1.   1.   1.   1.   0.67 1.   1.   1.
 1.   1.   1.   1.   1.   1.   1.   1.  ]
0.95 accuracy with a standard deviation of 0.12
```

Issues in Decision Tree Learning

- Homogeneous classification
  - All leaves are pure, that is have an entropy of 0
  - Leads to **over-fitting** of the training data

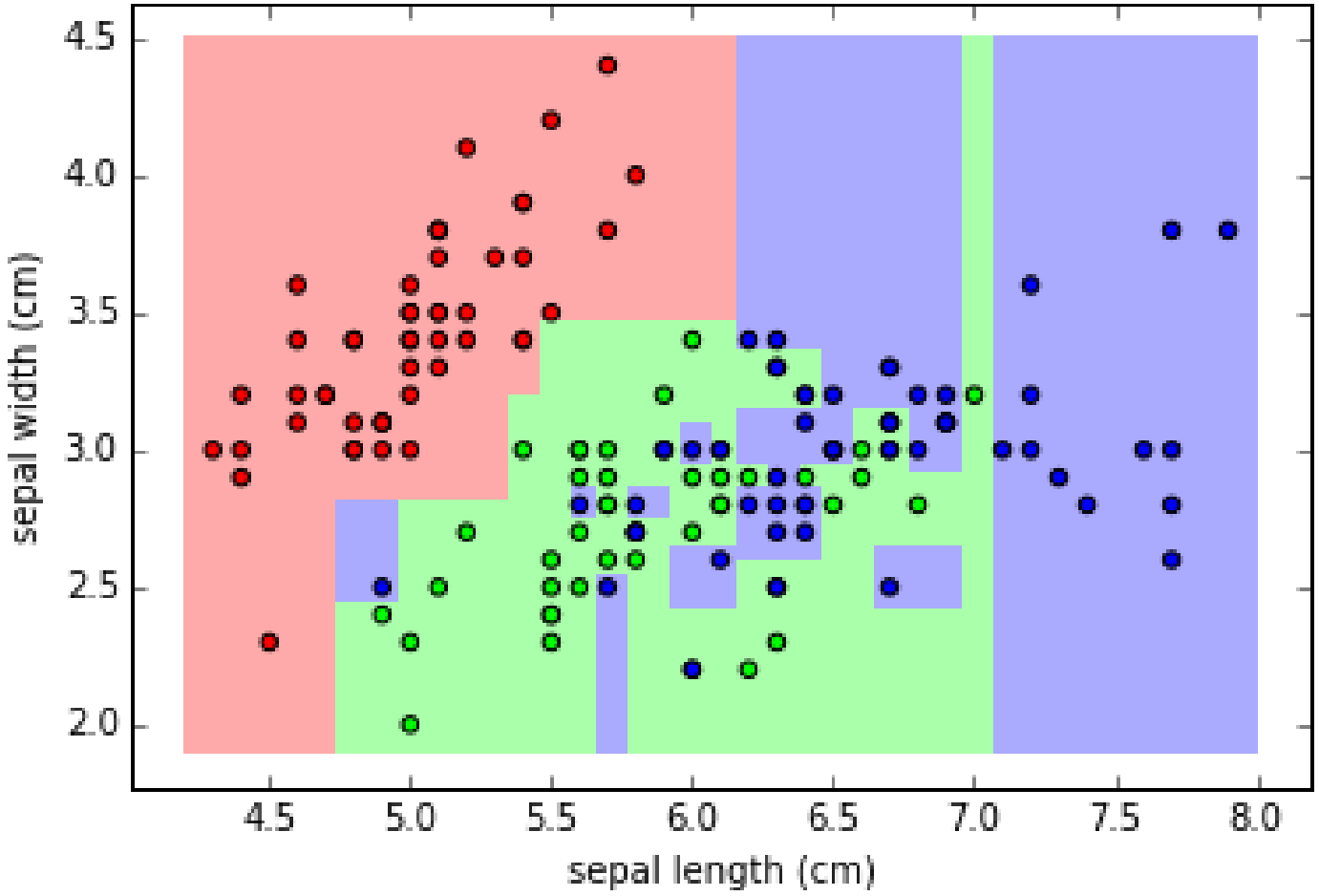


The Problem of Model Overfitting

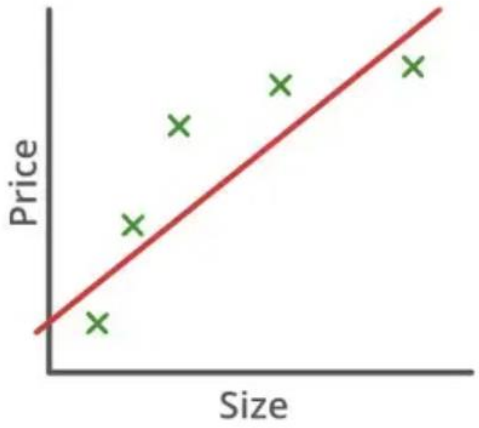
- **Problem Description**
  - All algorithms that build a classification model based on a finite set of training examples tend to have the problem of model overfitting:
    - the model induced from the training examples fits the training examples *too well*.
    - It reflects the specific features of the examples than features of actual data at large.
  - In decision tree induction, a fuller tree with more branches towards the leaf nodes may be built in order to classify those few examples.
  - Using such trees often results in misclassification
  - Causes of the problem include presence of noise data, lack of representative training examples, etc.

Overfitting

If we keep splitting we will be reducing the error, but…

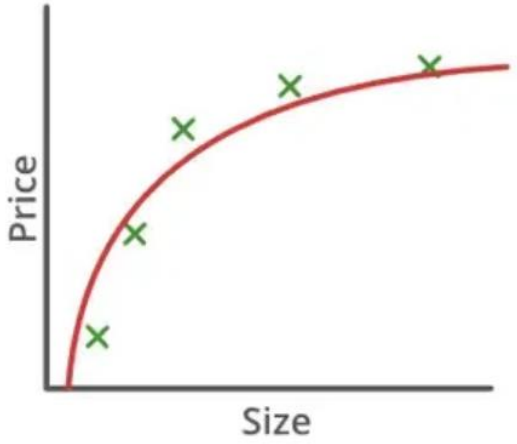


Underfitting vs. Overfitting



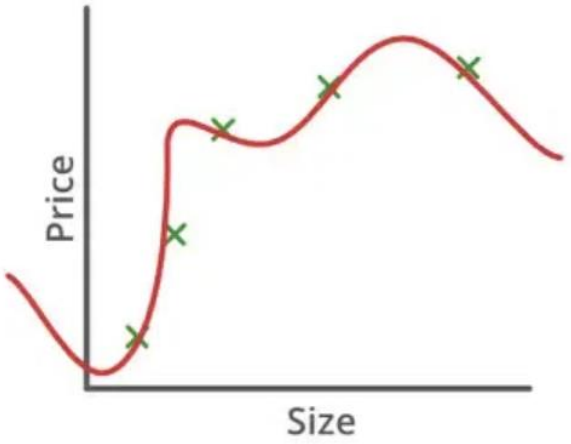
$\theta_0 + \theta_1 x$

High Bias
(Underfitting)



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

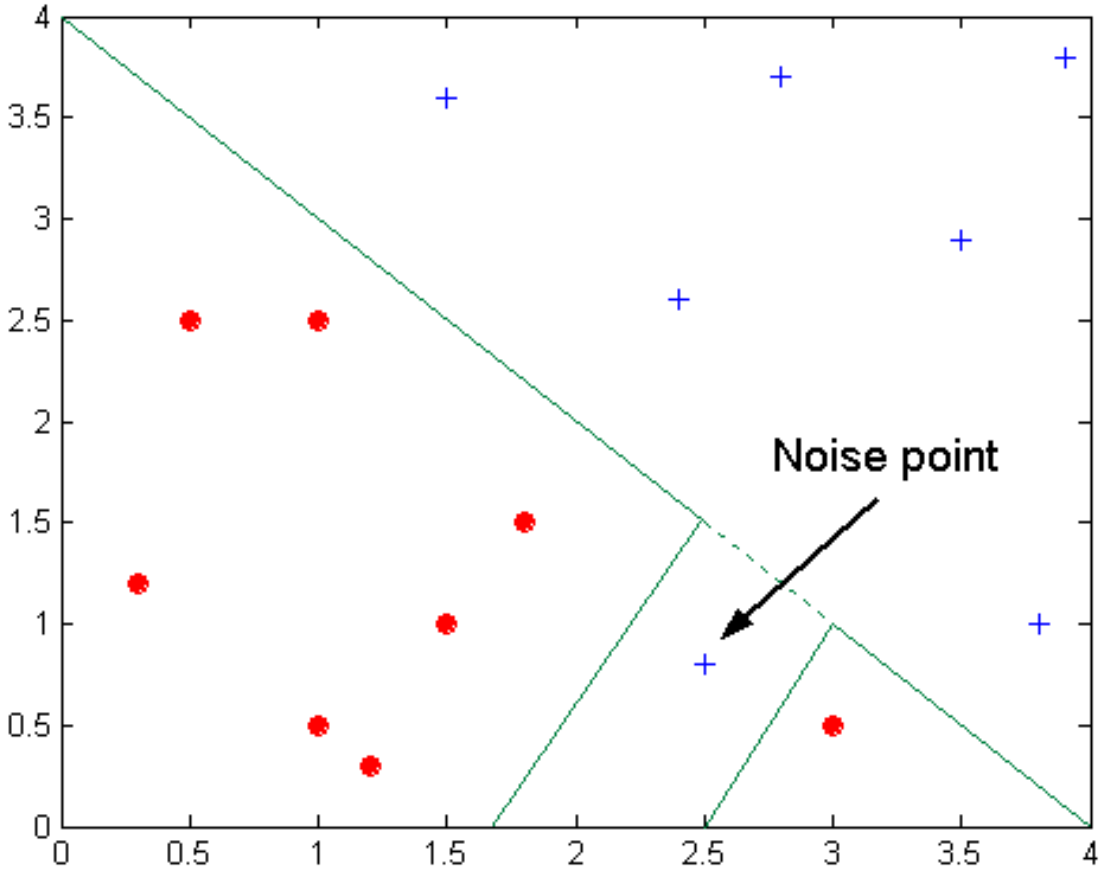
Low Bias, Low Variance
(Goodfitting)



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

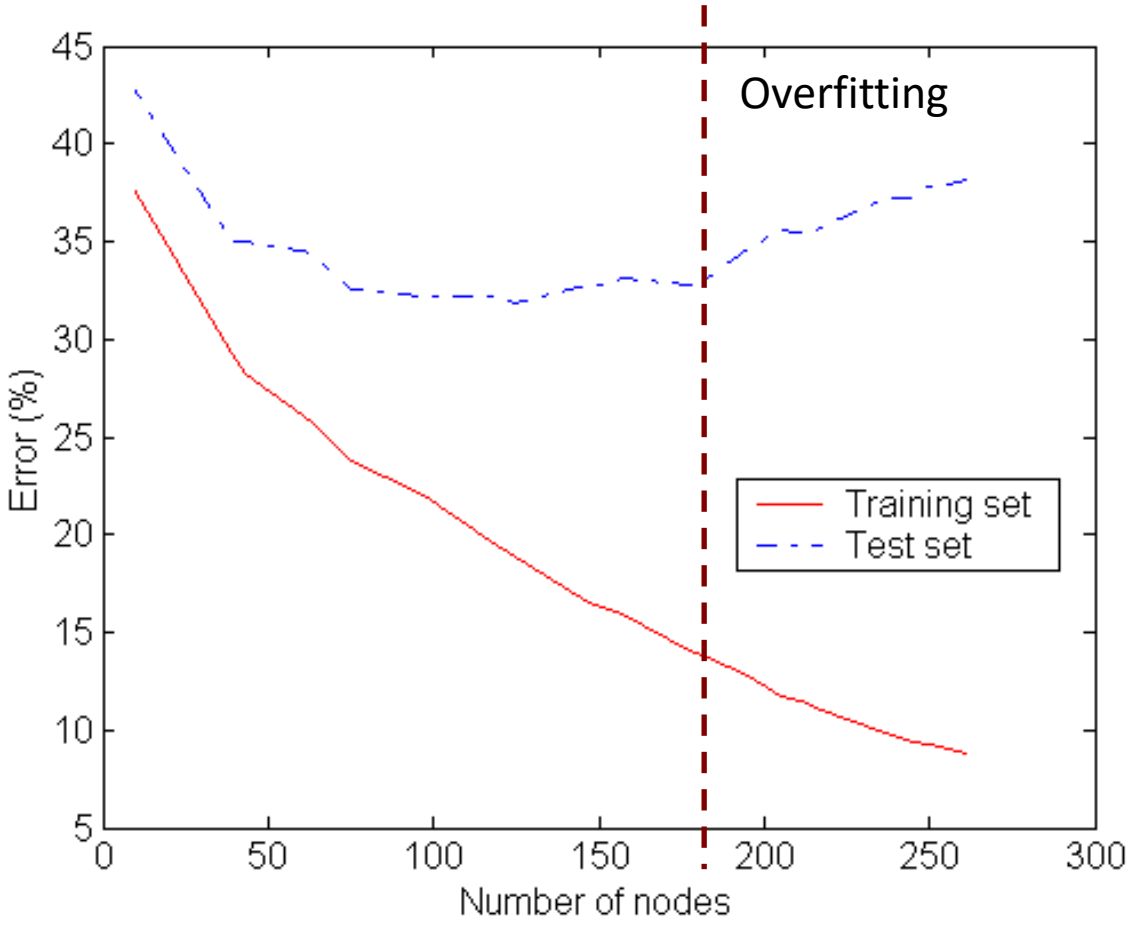
High Variance
(Overfitting)

Overfitting due to Noise



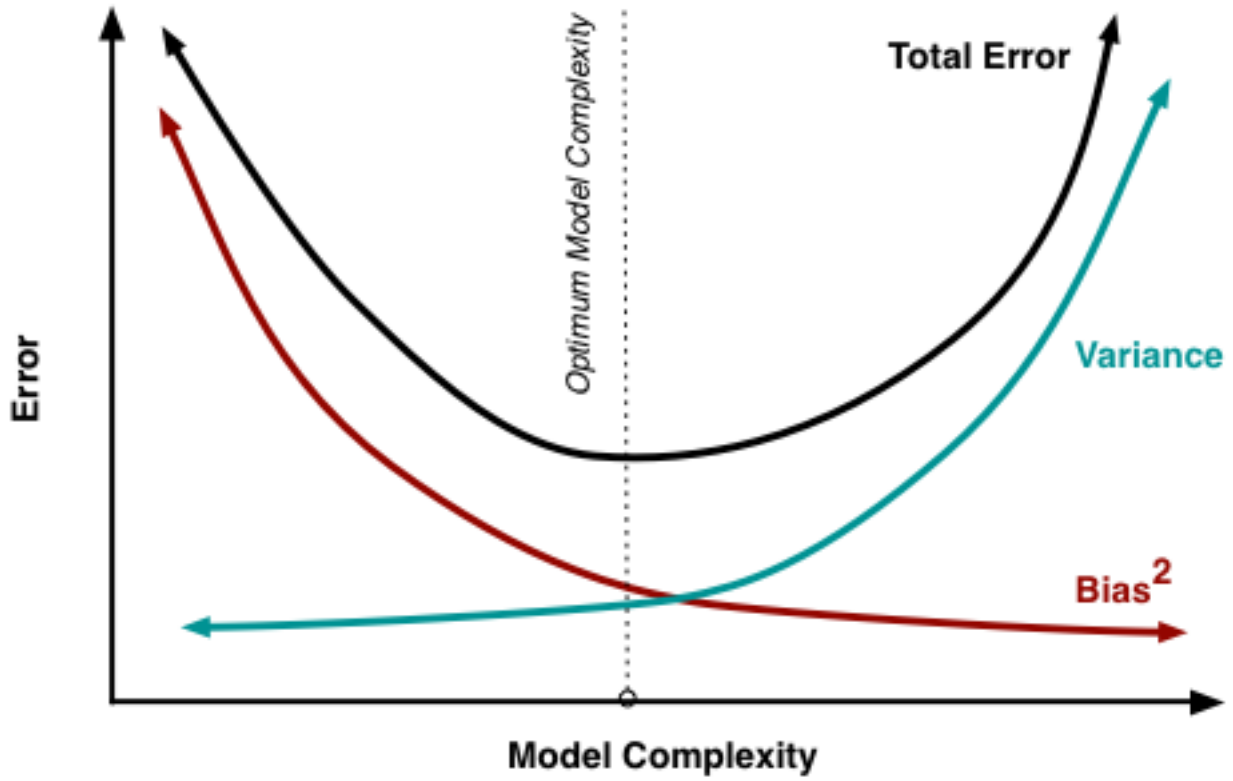
Decision boundary is distorted by noise point

Underfitting (High Bias) vs. Overfitting (High Variance)



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

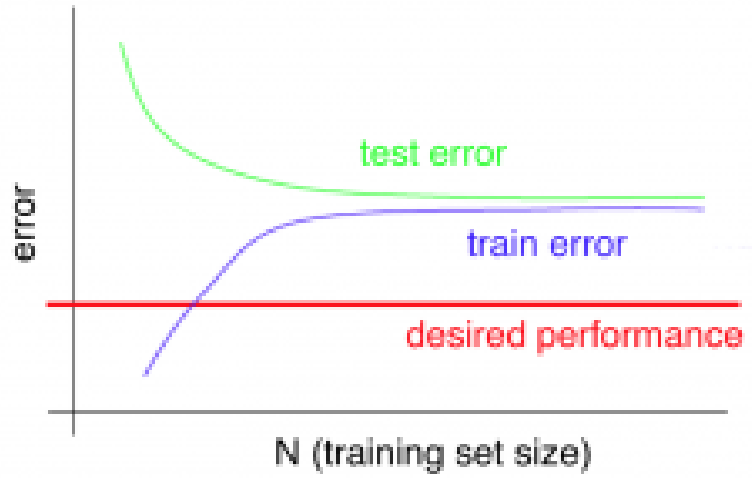
Trade-off: bias-variance errors



More on Bias vs. Variance

If a learning algorithm is suffering from **high bias**, getting more training data will not **(by itself)** help much.

Typical learning curve for high bias (at fixed model complexity)




```
#Generate data raning from 1000 to 5000. Split into training and testing.
#Build a decision tree of one node only
for i in range(1,6):
    X_moons_train, y_moons_train = make_moons(n_samples=1000*i, noise=0.3, random_state=42)
    X_moons_train, X_moons_test, y_moons_train, y_moons_test = train_test_split(X_moons, y_moons, test_size=0.5)
    tree_clf = DecisionTreeClassifier(max_depth=1,random_state=42)
    tree_clf.fit(X_moons_train, y_moons_train)
    print("Training Dataset", 1000*i/2, print("Testing Dataset", 1000*i/2))
    print("Training Accuracy", tree_clf.score(X_moons_train, y_moons_train))
    print("Testing Accuracy", tree_clf.score(X_moons_test, y_moons_test))
```

```
Testing Dataset 500.0
Training Dataset 500.0 None
Training Accuracy 0.8133333333333334
Testing Accuracy 0.7866666666666666
Testing Dataset 1000.0
Training Dataset 1000.0 None
Training Accuracy 0.8266666666666667
Testing Accuracy 0.84
Testing Dataset 1500.0
Training Dataset 1500.0 None
Training Accuracy 0.8266666666666667
Testing Accuracy 0.8
Testing Dataset 2000.0
Training Dataset 2000.0 None
Training Accuracy 0.8133333333333334
Testing Accuracy 0.8
Testing Dataset 2500.0
Training Dataset 2500.0 None
Training Accuracy 0.8666666666666667
Testing Accuracy 0.8
```

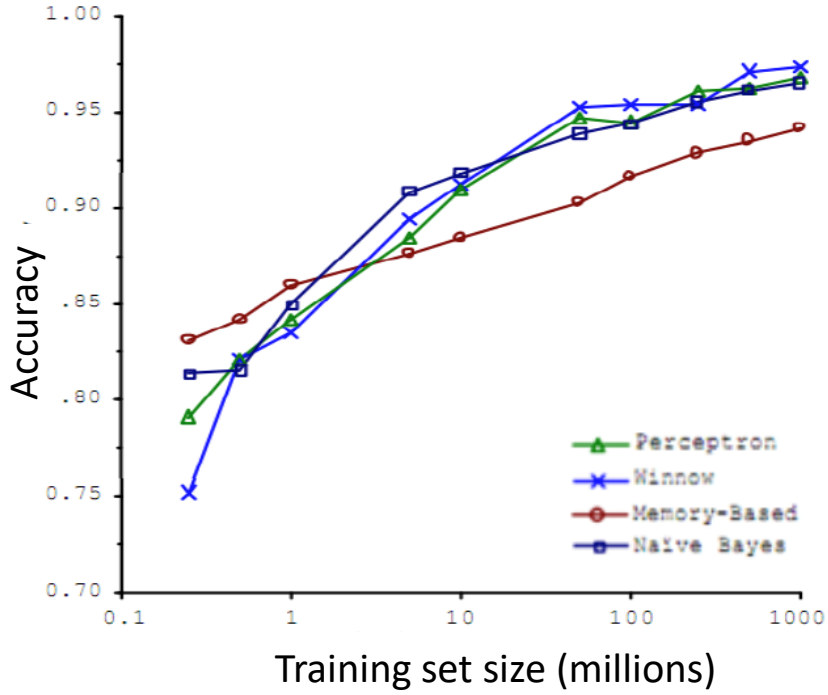
More on Bias vs. Variance

If a learning algorithm is suffering from **high variance**, getting more training data is likely to help.

Typical learning curve for high variance (at fixed model complexity)

Having more data…

Michele Banko and Eric Brill. 2001. **Scaling to very very large corpora for natural language disambiguation**. In Proceedings of the 39th Annual Meeting on Association for Computational Linguistics (ACL '01). Association for Computational Linguistics, USA, 26–33.



"It's not who has the best algorithm that wins. It's who has the most data."

Diagnosing ML Models

- **Simple model** → prone to underfitting (high bias & low variance).
  - It is computationally cheaper.
  - The model fits poorly consistently
  - Fixes:
    - Add features
      - to have sufficient information, e.g., Predict housing price from only size and no other features is hard even for human to do.
    - Build more complex model, e.g., larger trees, ANN with higher number of hidden layers, etc.

- **Complex model** → prone to overfitting (high variance – low bias).
  - It is computationally expensive.
  - Fixes:
    - More training examples
    - Smaller set of features
    - Build less complex models, e.g., shorter trees, ANN with less number of hidden layers, etc.

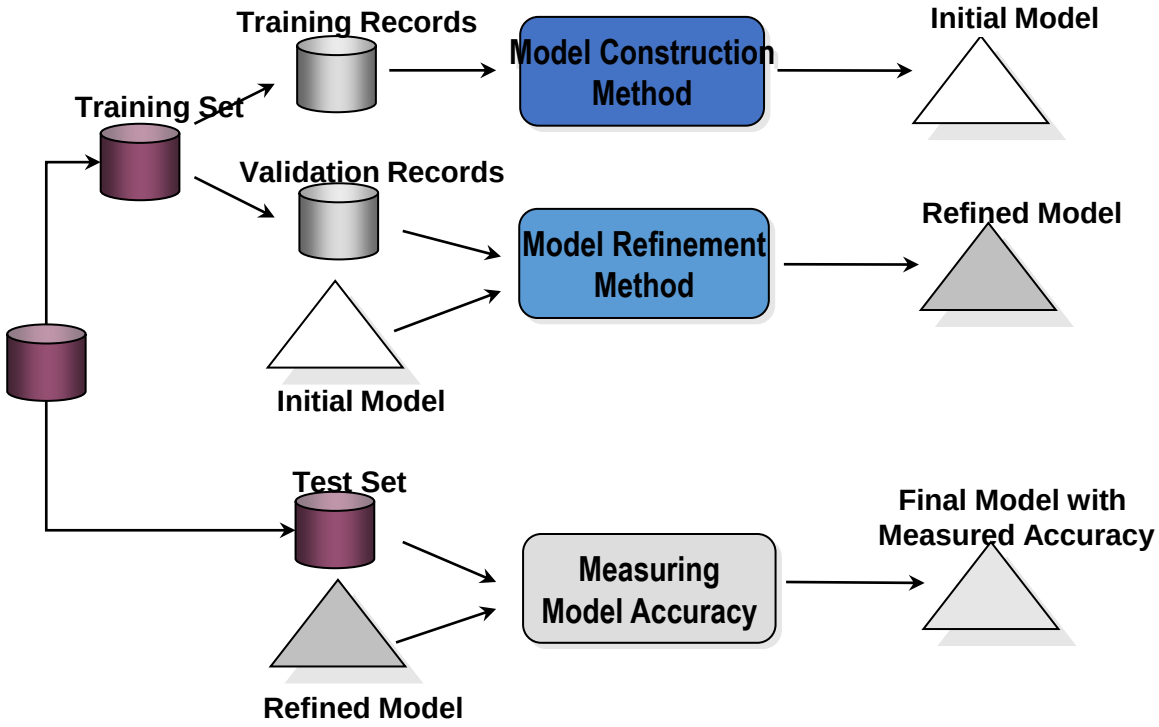
- In reality, we would want to choose a model somewhere in between, that can generalize well but also fits the data reasonably well.

Pruning: Simplifying a Decision Tree

- Tree Pruning: Remove subtrees for better generalization (decrease variance)
- **Approaches**
  - **Pre-pruning** (forward pruning): Early stopping rule: tree construction is halted at some stage
    - Typical stopping conditions for a node:
      - Stop if all instances belong to the same class
      - Stop if all the attribute values are the same
    - Do not split a node if this would result in the goodness measure (e.g., using χ^2 test, Gini, or information gain) falling below a threshold
      - Difficult to choose an appropriate threshold
    - Stop at a specified max. tree size (depth)
  - **Post-pruning** (backward pruning or just pruning): Grow the whole tree then "cut back" certain subtrees and replace them with leaf nodes, making the tree smaller and more robust
- Prepruning is faster, postpruning is more accurate (requires a separate pruning set)

Tree Pruning

- Use an independently sampled validation set to assist tree pruning



- Pruning methods:
  - **reduced error pruning**
  - cost complexity pruning
  - pessimistic pruning
  - …and many others

Tree Pruning

- **Reduced Error Pruning Method**

  1. Classify all validation examples using the tree. Note down errors at non-leaf nodes

  2. For every non-leaf node, count the number of errors if the subtree is replaced by *the best possible* leaf.

  3. Choose the subtree that has the largest reduction of the number of errors to prune, if any of its subtree do not reduce the number of errors. Repeat this step until any further pruning increases the number of errors.

  (a) Prune the tree even when there is a zero reduction in errors.
  (b) There may be a number of subtrees with the same error reduction. In this case, choose to prune the largest subtree.

Tree Pruning

- **Reduced Error Pruning (Example)**

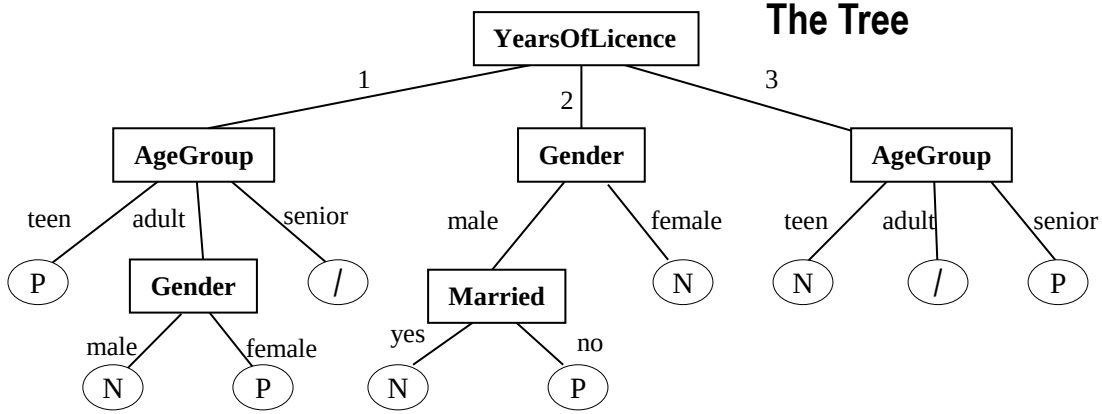
Training set

AgeGroup	Gender	Married	YearsOfLicience	Class
teen	male	no	1	P
teen	male	yes	1	P
teen	female	no	2	N
adult	male	no	2	P
adult	female	yes	2	N
adult	male	yes	1	N
teen	female	yes	2	N
teen	male	no	3	N
adult	female	yes	1	P
senior	male	yes	2	N
senior	female	yes	2	N
senior	male	no	3	P

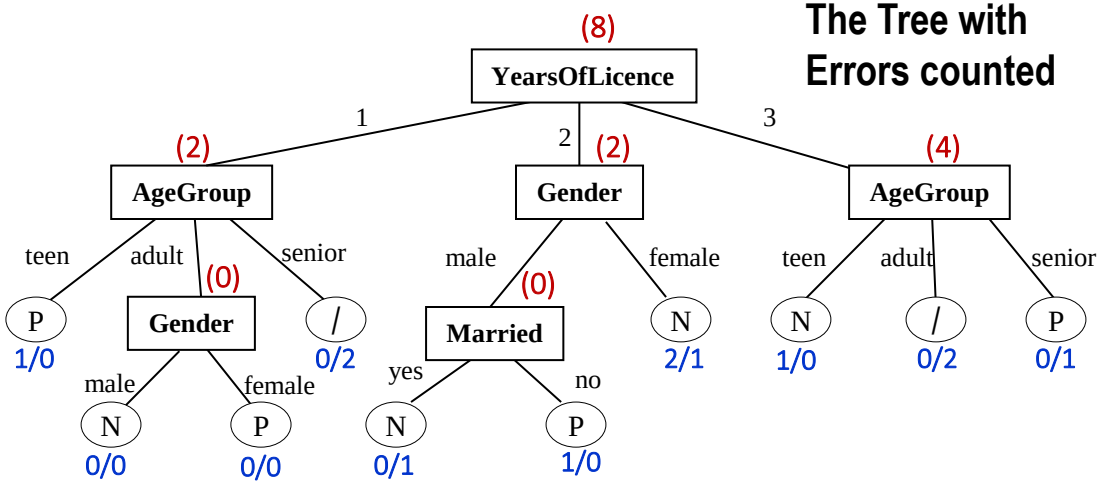
Validation set

AgeGroup	Gender	Married	YearsOfLicience	Class
teen	male	no	2	P
teen	male	no	3	P
teen	female	no	2	P
teen	female	yes	2	P
adult	female	no	3	N
adult	female	yes	3	N
adult	female	no	2	N
teen	female	yes	1	P
senior	male	yes	1	N
senior	female	yes	1	N
senior	female	yes	3	N
adult	male	yes	2	N

The Tree

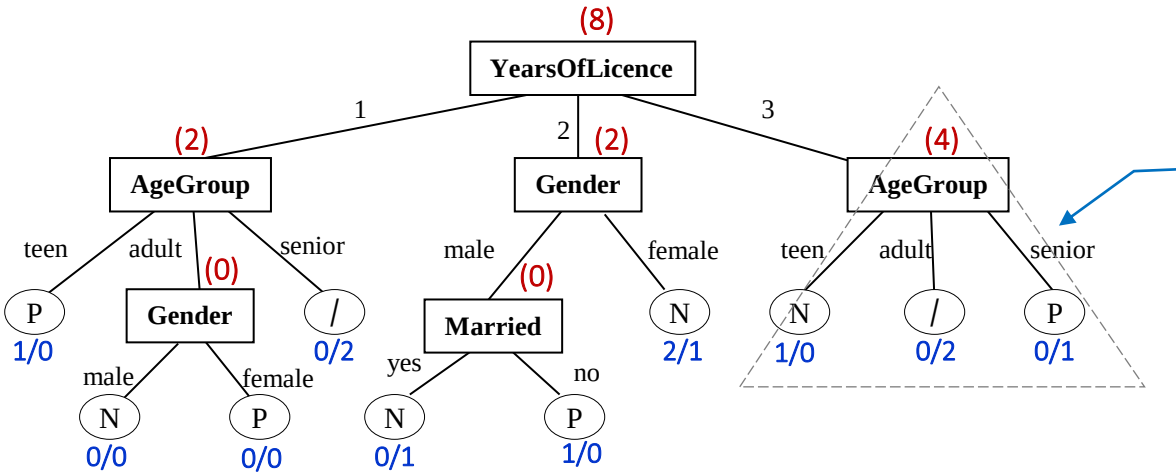


The Tree with Errors counted

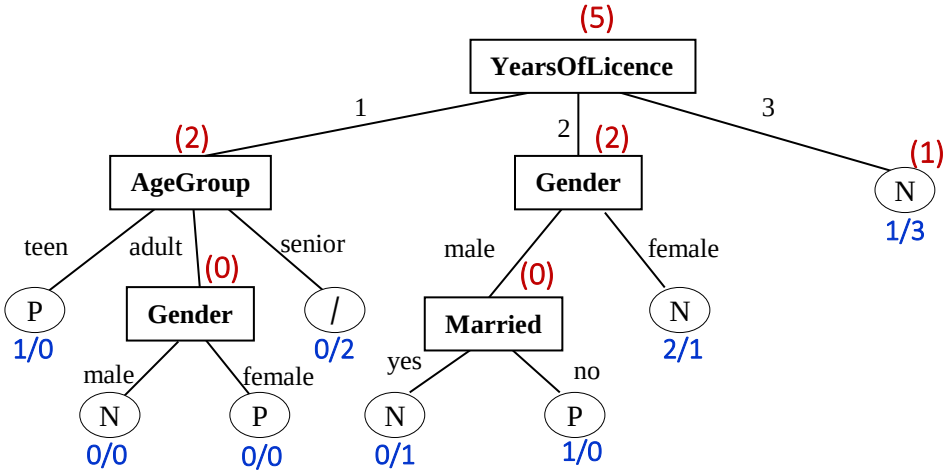


Tree Pruning

- **Reduced Error Pruning (Example)**

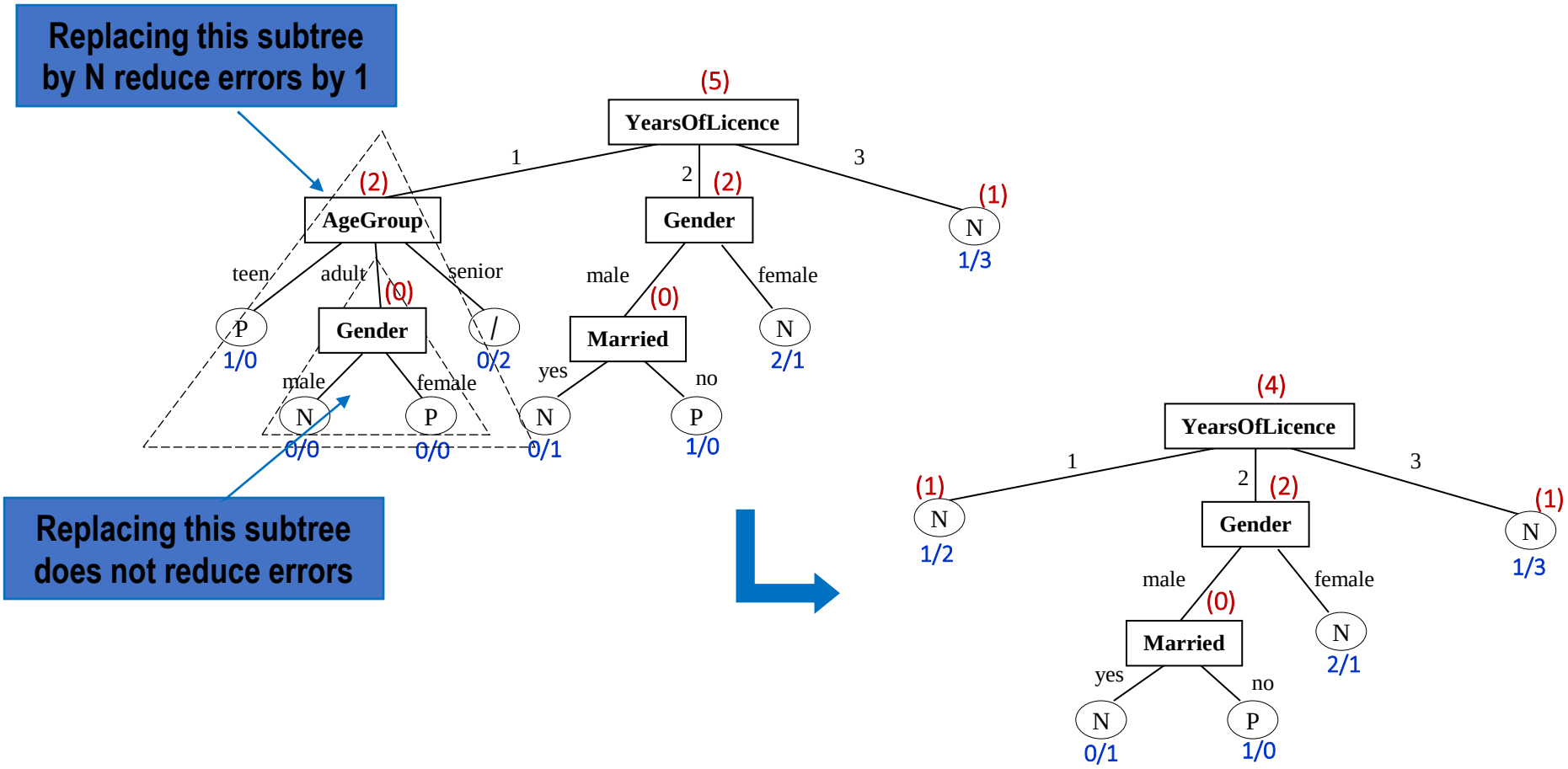


This subtree has the largest number of errors. Replacing it by N reduce errors from 4 to 1



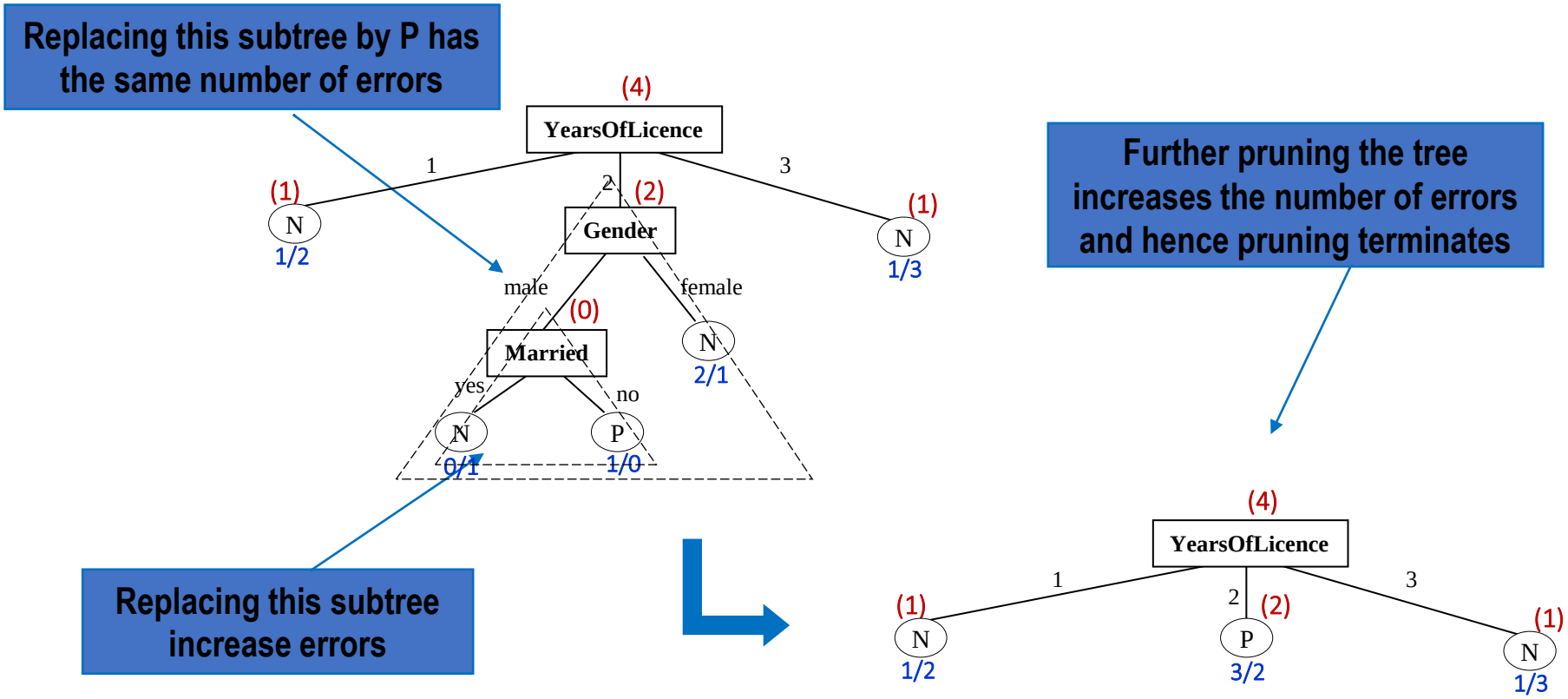
Tree Pruning

- **Reduced Error Pruning (Example)**



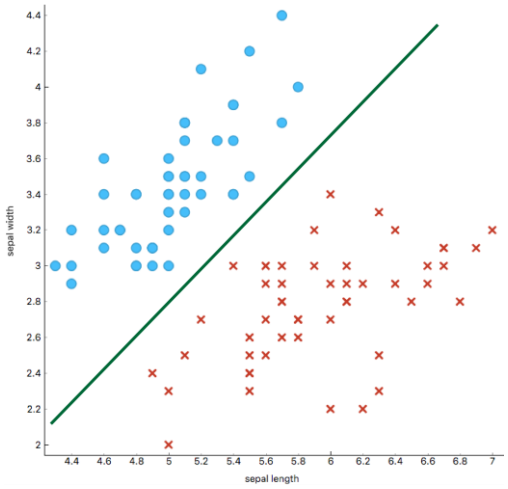
Tree Pruning

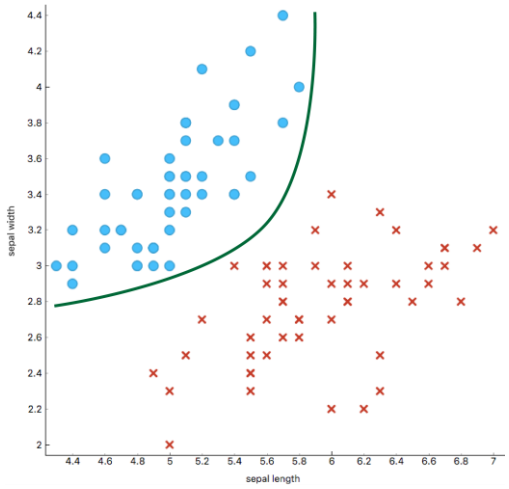
- **Reduced Error Pruning (Example)**

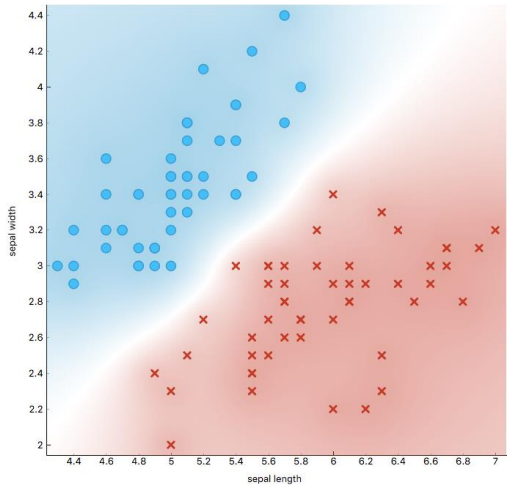


Occam's Razor

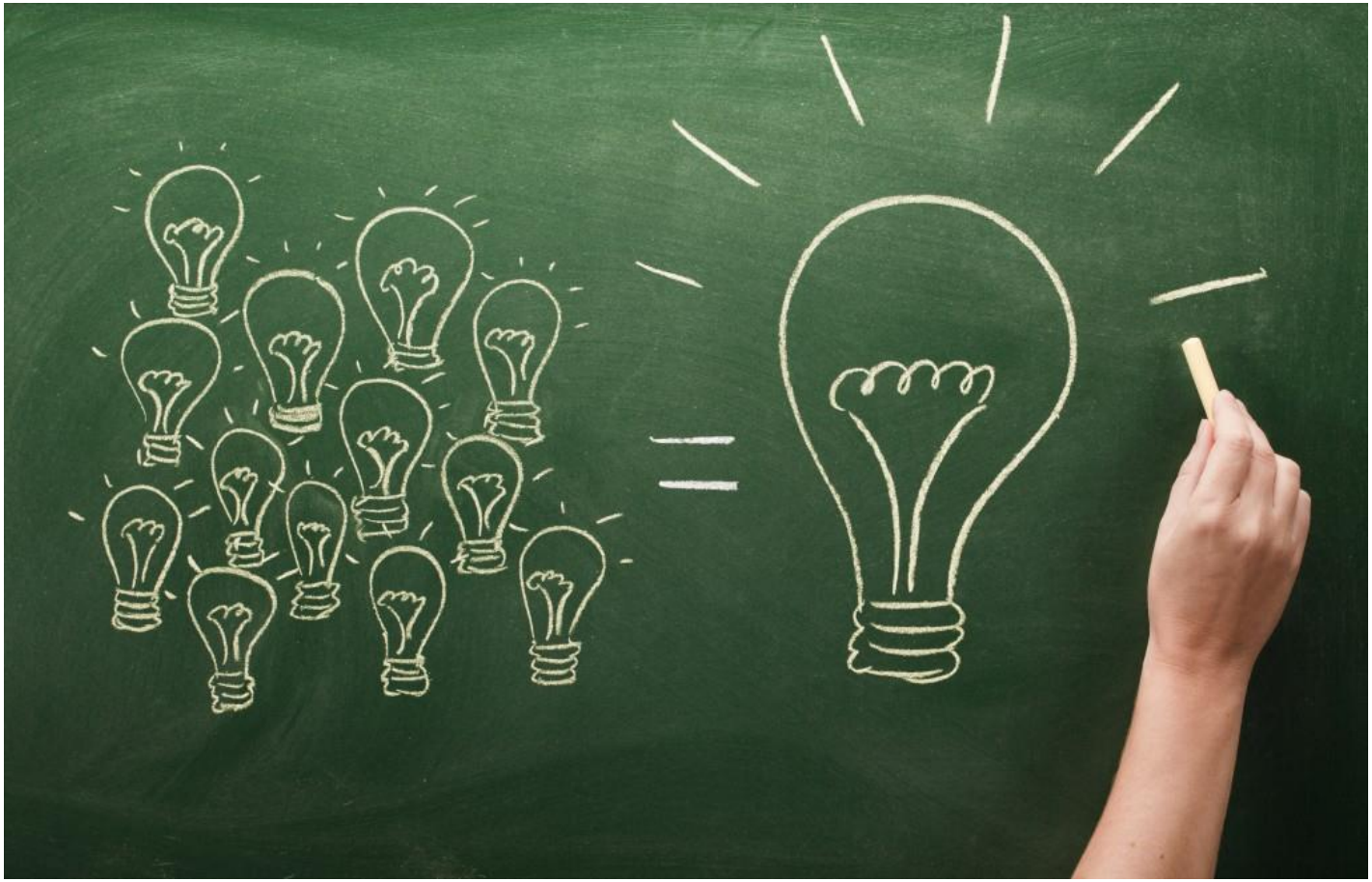
- The principle of preferring the **simplest hypothesis** that is (reasonably) consistent with the training data
  - Given two models of similar generalization errors, one should prefer the simpler model over the more complex model
  - For complex models, there is a greater chance that it was fitted accidentally by errors in data
  - Therefore, one should include model complexity when evaluating a model







Ensemble Learning: Power of the crowds



Ensemble methods

- A single decision tree does not perform well
- But, it is super fast
- What if we learn multiple trees?

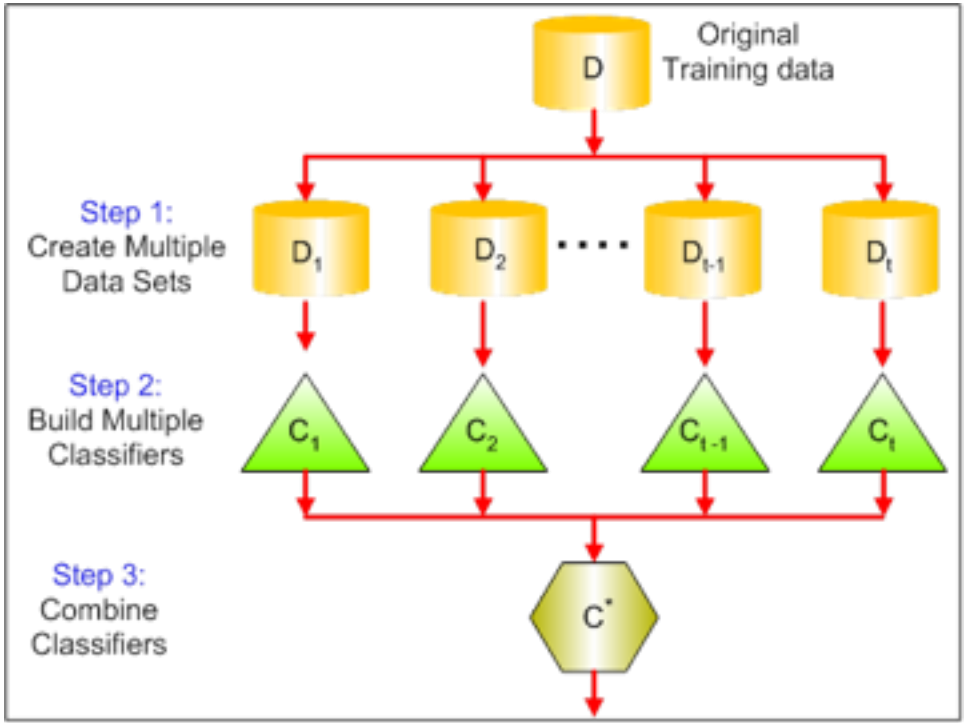
We need to make sure they do not all just learn the same

- Methods:
  - Bagging
  - Boosting
  - Stacking

Bagging

Bootstrap **agg**regat**ing:**

- Multiple realizations of the data
  - Multiple samples (sampling with replacement)
- Train multiple models (hundreds) each with a different data sample
- Calculate predictions multiple times and take the average/majority vote with equal weight
- **Random forest**: an implementation of bagging



- Pros:
  - Improve accuracy over single models; reduces overfitting (variance)
  - Normally uses one type of model; decision trees (no pruning) are popular
  - Easy to parallelize
- Cons:
  - Difficult to interpret the resulting model.

Bagging

One Single Tree

In [27]:
```
#One Single Tree
tree_clf = DecisionTreeClassifier(random_state=42)
tree_clf.fit(X_train, y_train)
print("Accuracy:",tree_clf.score(X_test, y_test))
```

```
Accuracy: 0.8716666666666667
```

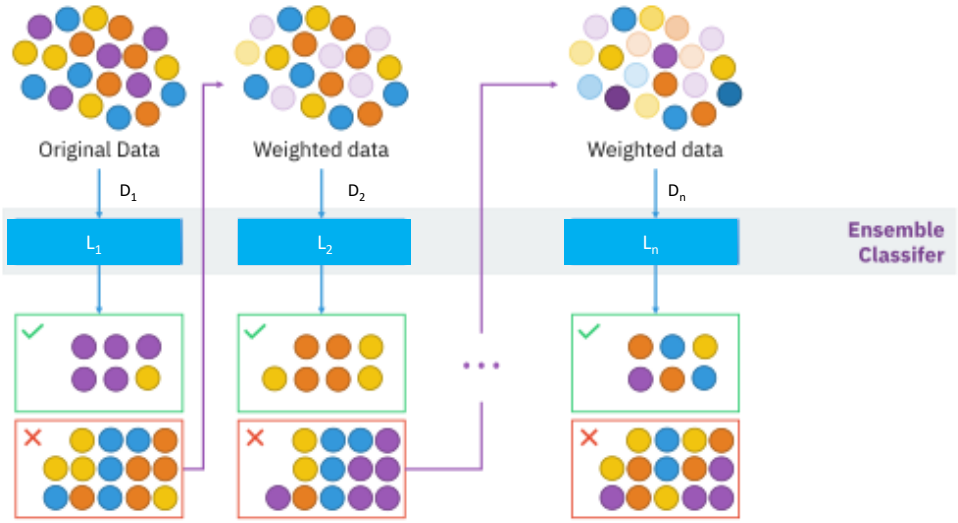
Random Forest

In [28]:
```
randomForest_clf = RandomForestClassifier(n_estimators=100, random_state=42,max_samples=480)
randomForest_clf.fit(X_train, y_train)
print("Accuracy:",randomForest_clf.score(X_test, y_test))
```

```
Accuracy: 0.9166666666666666
```

Boosting

- Incrementally building an ensemble
- Training each new model to emphasize the training instances that previous models mis-classified.
  - Equal weight is given to the sample training data (D_1) at the very starting round.
  - This data (D_1) is then given to a base learner (L_1).
  - The mis-classified instances by L_1 are assigned a weight higher than the correctly classified instances.
  - This boosted data (D_2) is then given to second base learner (L_2) and so on.
  - The results are then combined in the form of weighted voting, based on how well a learner performs.
- **Adaboost (**Adaptive Boosting**)**: an implementation of boosting
  - AdaBoost (with decision trees "stumps" as the weak learners) is often referred to as the best out-of-the-box classifier
- Pros:
  - Reduces bias, and also variance
  - In some cases, boosting has been shown to yield better accuracy than bagging
- Cons:
  - Tends to be more likely to over-fit the training data
  - Sensitive to noisy data and outliers



Stacking

- Stacking (a.k.a. Stacked Generalization): training a ML model to combine the predictions of several other models trained using the available data
  - Level-0 Models (Base-Models) are typically different (e.g. not all decision trees) and fit on the same dataset (e.g. instead of samples of the training dataset).
  - Level-1 Model (Meta-Model): learns how to best combine the predictions of the base models.
  - Typically yields performance better than any single one of the trained models