Classification Model Evaluation

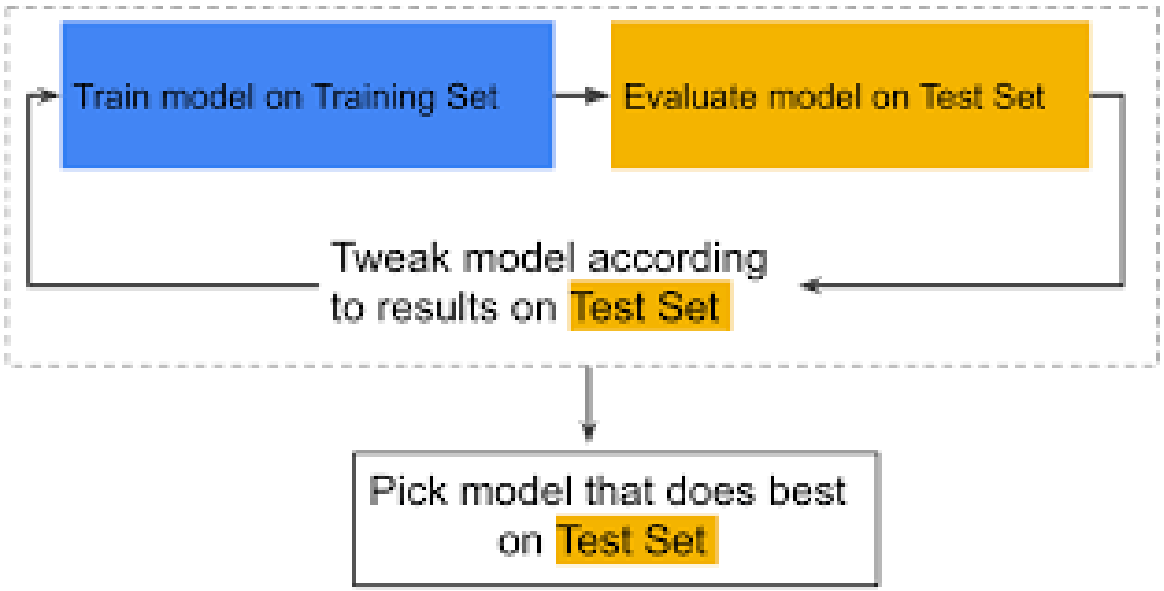
Outline

- ML Model Evaluation Metrics
- Underfitting & Overfitting Issues
- Tree Pruning
- Ensemble Learning

ML Model Evaluation Metrics

This part is adapted from this source

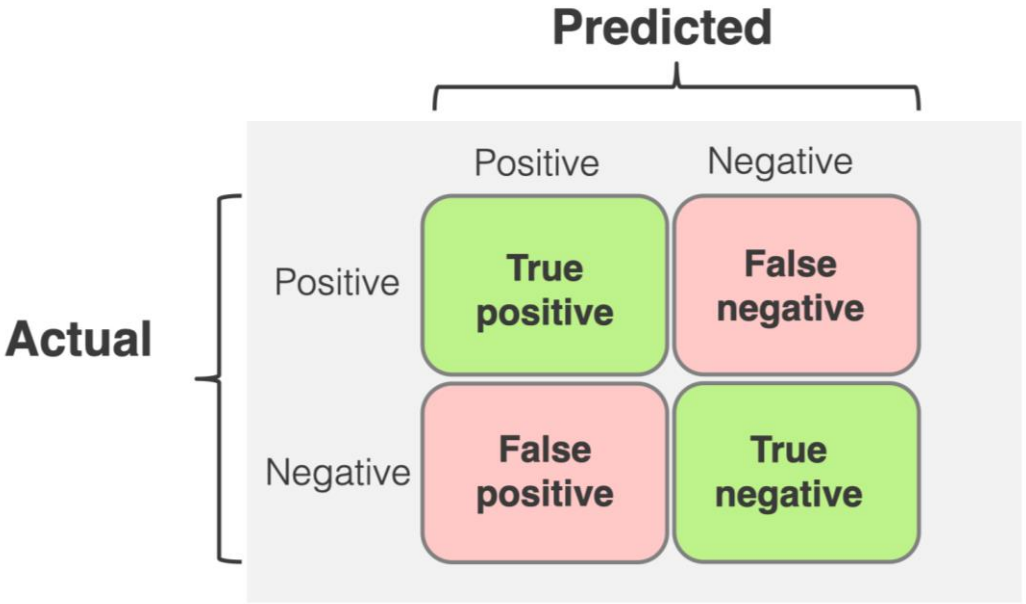
Defining Training and Test Set Splits



- 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

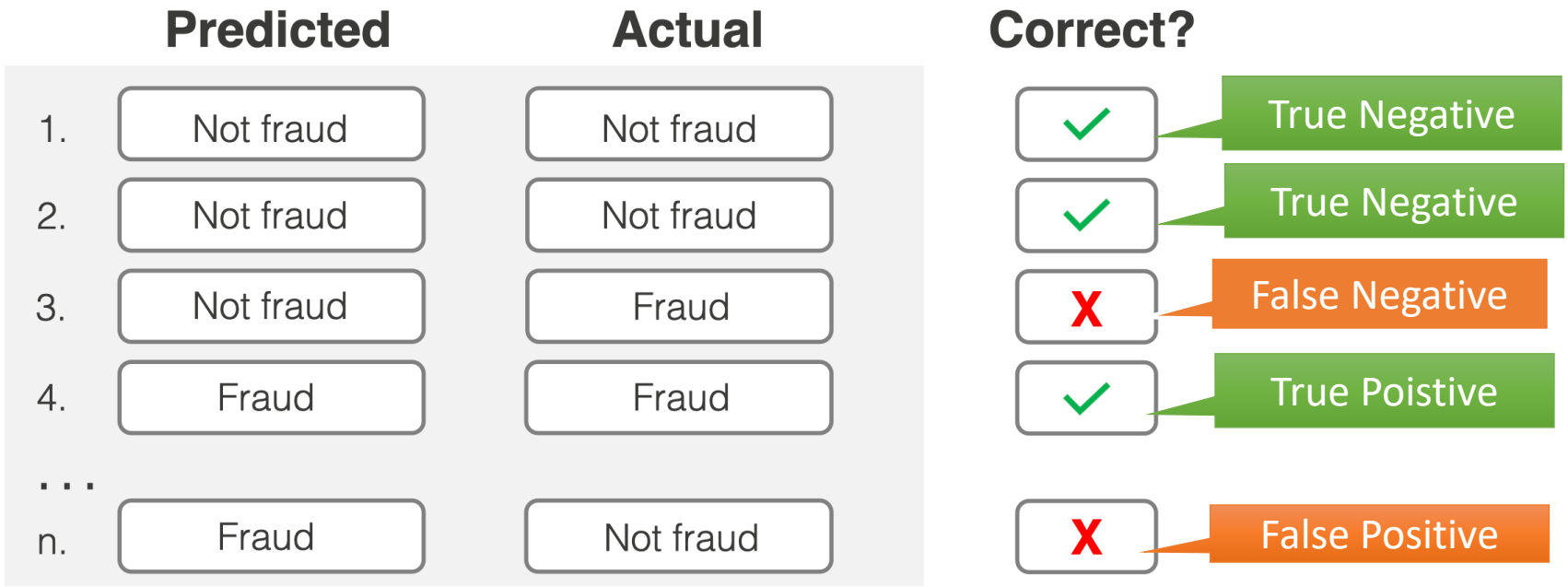
Evaluating Classifier Accuracy

- **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

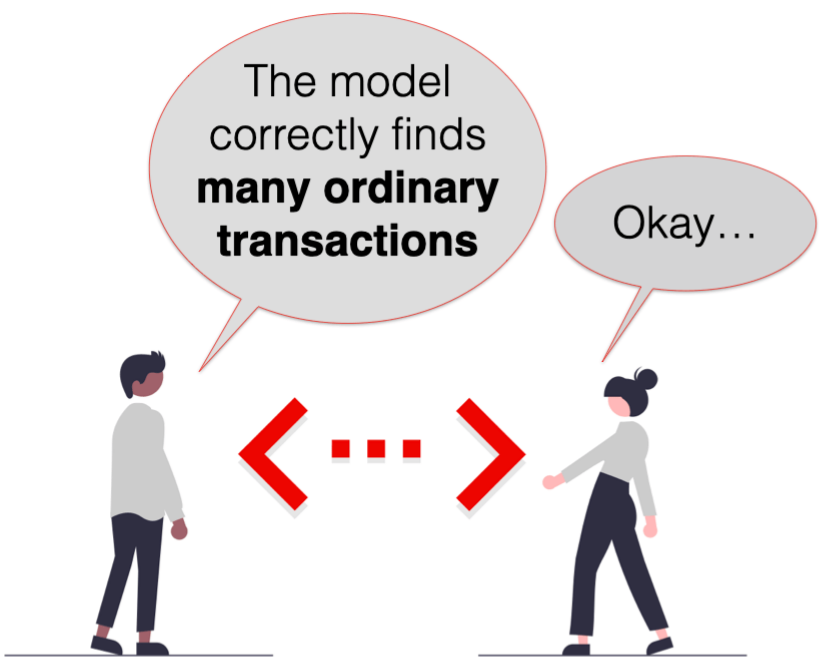
False Positive vs False Negative



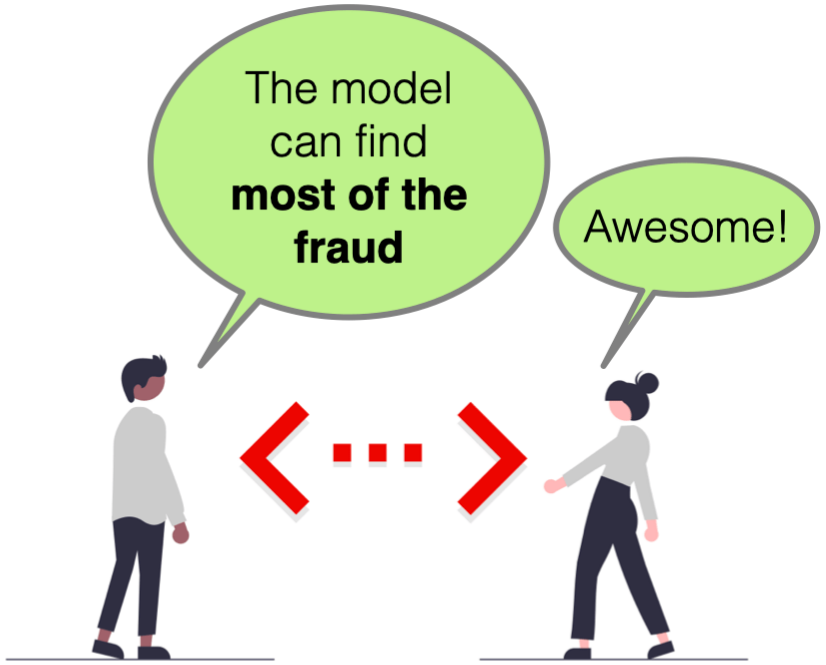
- There are two types of errors the model can make:
  - It can make a **false alarm** and label an ordinary transaction as fraudulent (False Positive)
  - It can **miss real fraud** and label a fraudulent transaction as ordinary (False Negative)

True Positive vs True Negative

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



True negative

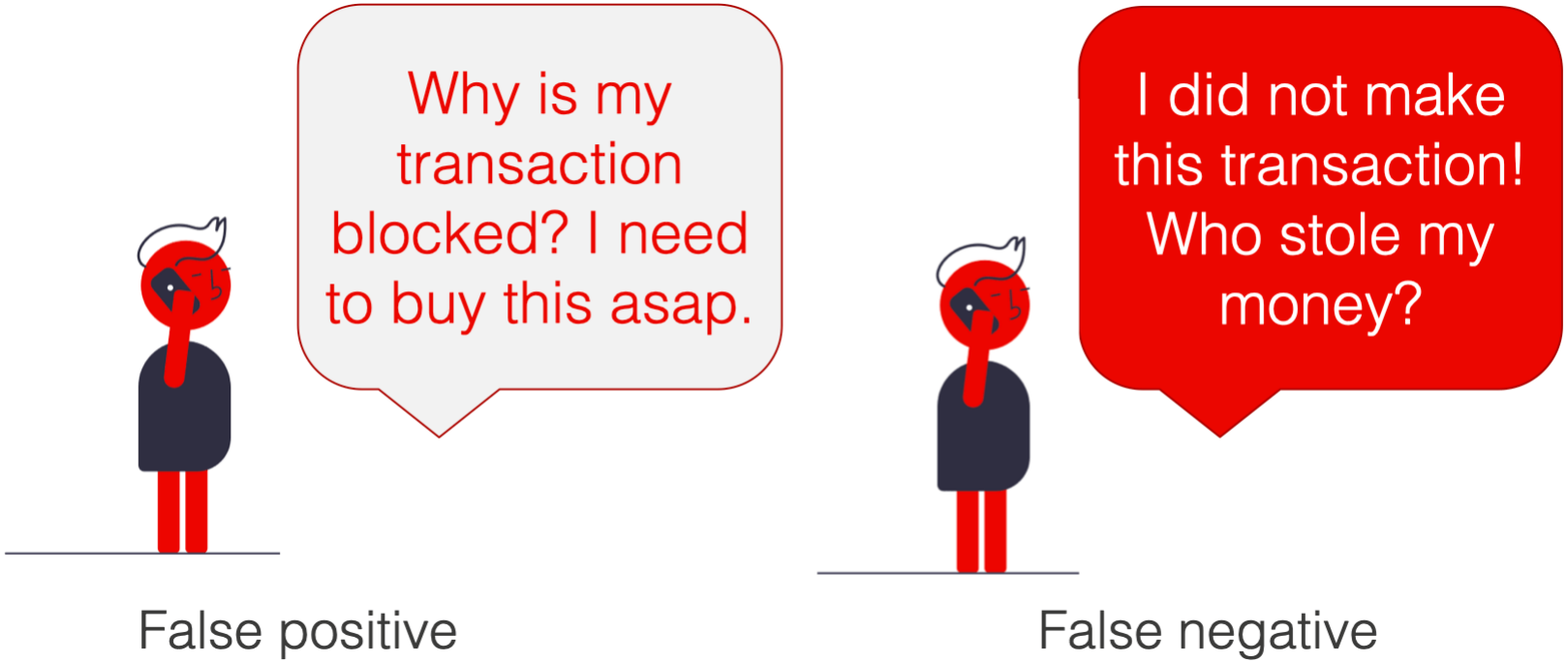


True positive

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

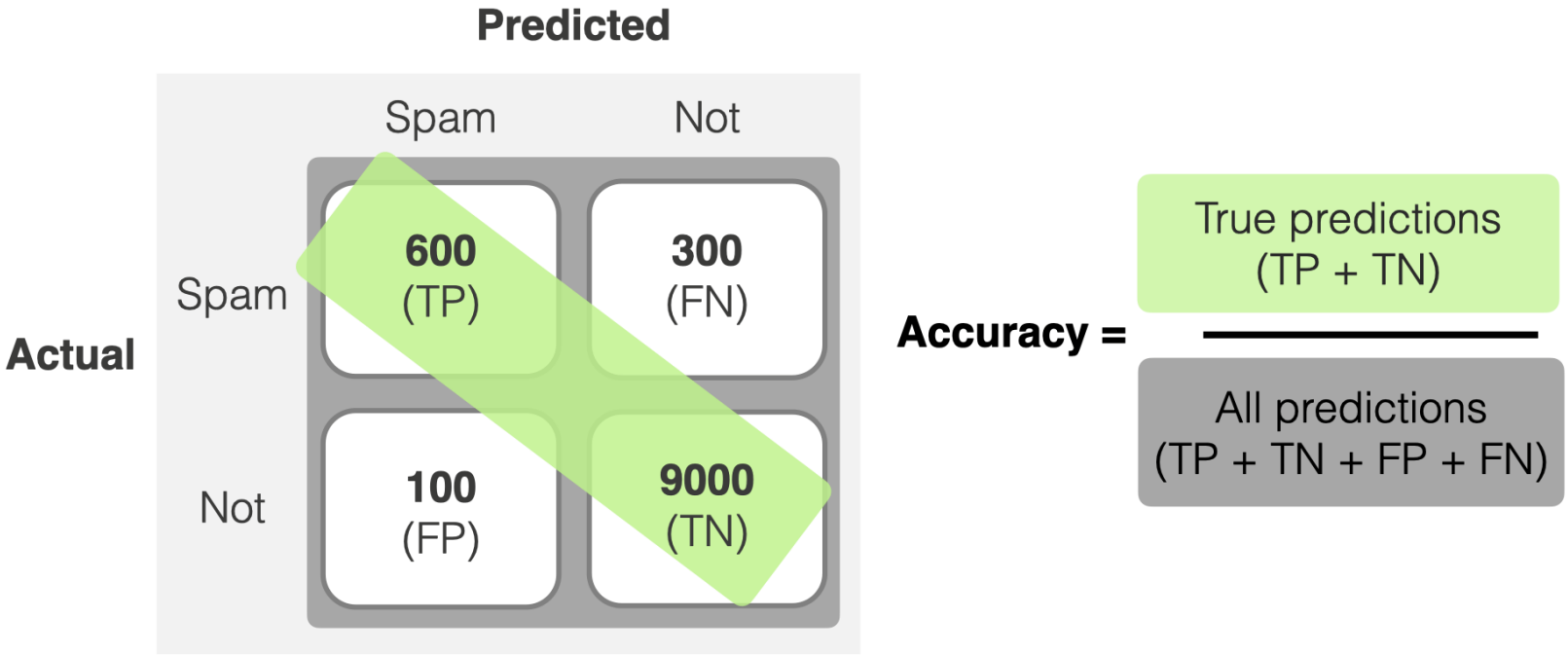


Spam Classifier

- **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)

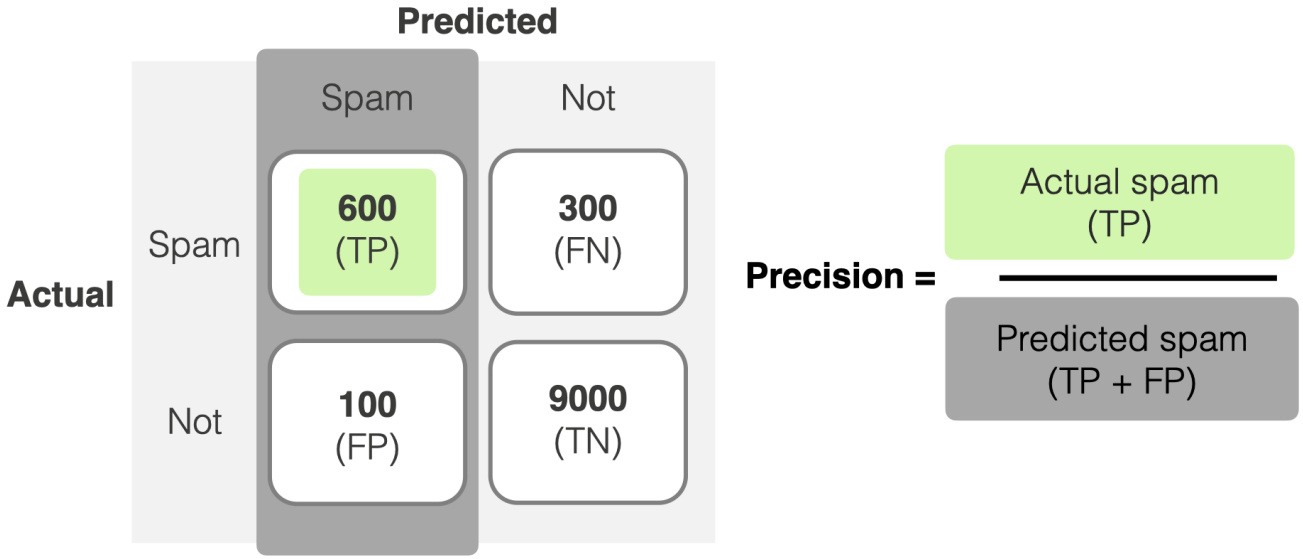
		Predicted: Spam	Predicted: Non-spam
Actual	Spam	**600** (True positive)	**300** (False negative)
Actual	Non-spam	**100** (False positive)	**9000** (True negative)

Accuracy



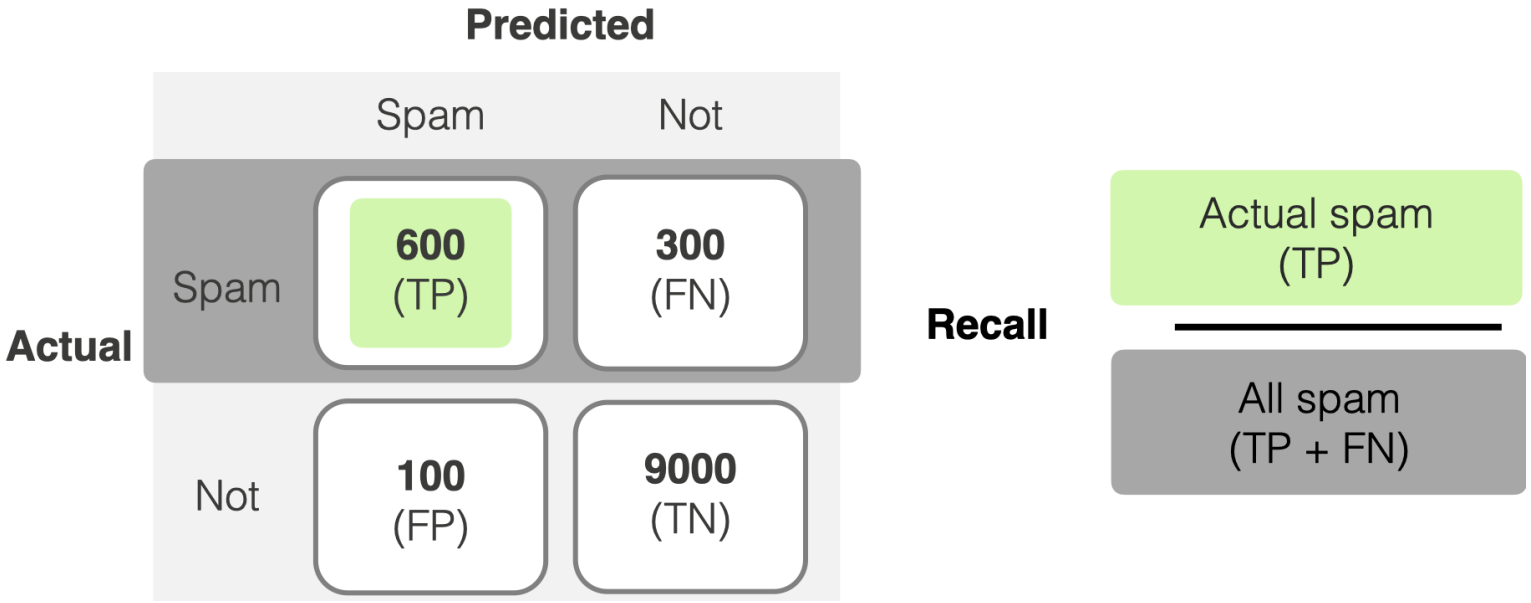
- Calculate accuracy by dividing all true predictions by the total number of predictions
- In our example, accuracy is (9000+600)/10000 = 0.96. The model was correct in 96% of cases
- However, accuracy can be misleading for imbalanced datasets when one class has significantly more samples (e.g., we have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

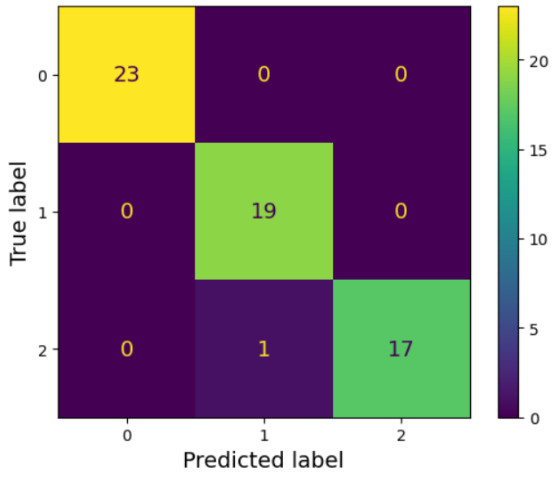
```
y_pred = tree_clf.predict(X_test)
print("Predicted Labels:", y_pred)  #Predicted labels the testing points
print("True Labels:     ", y_test) #True labels
print("Testing Accuracy:", metrics.accuracy_score(y_test, y_pred))  #1 mistake .. 59/60 = 0.983
```

```
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))
```

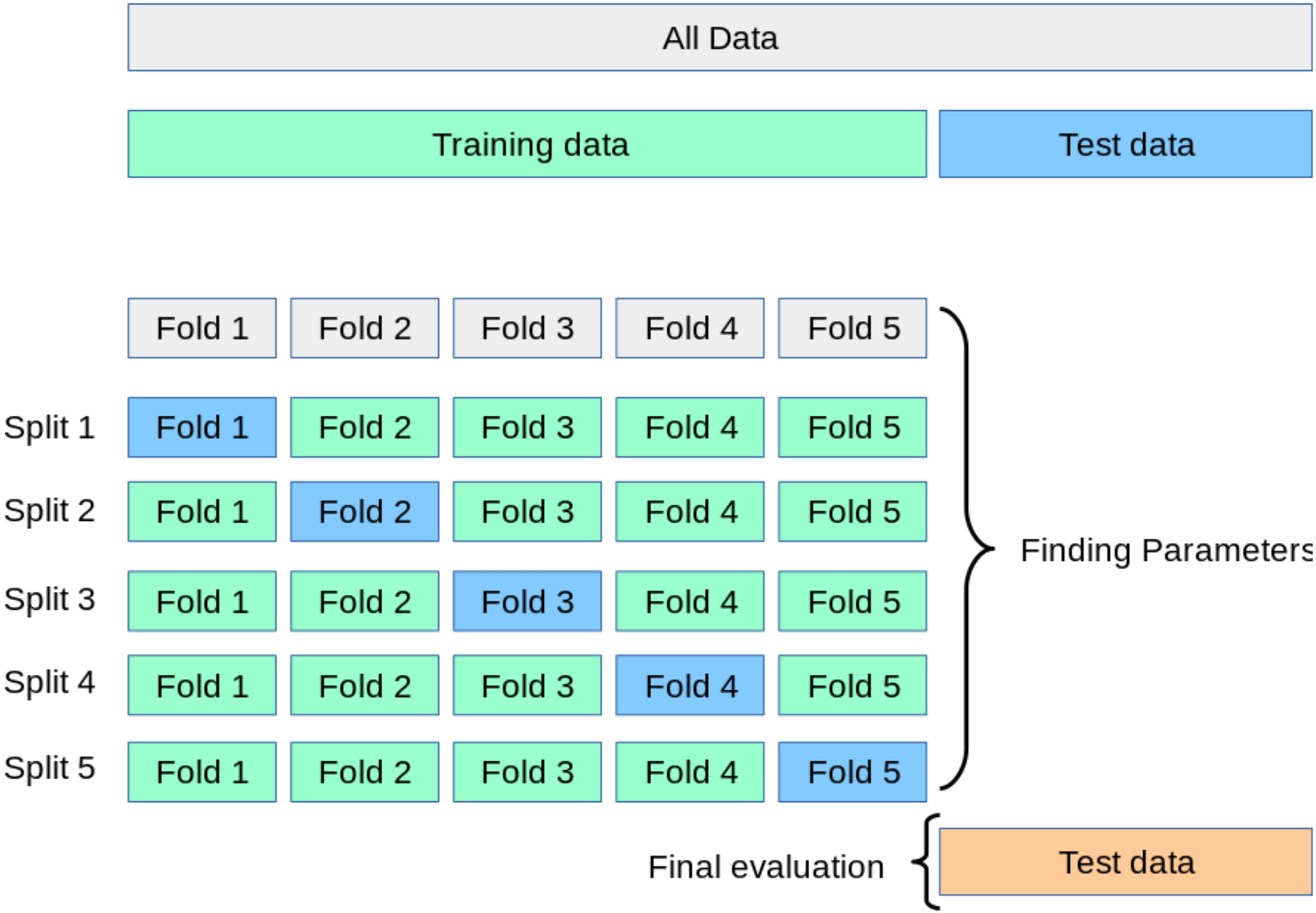
```
Training Accuracy: 0.9888888888888889
```

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
```

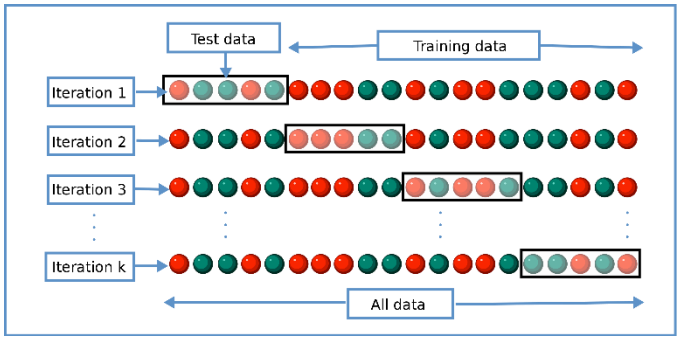
```
Predicted Labels: [1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1
 0 0 0 2 1 1 0 0 1 1 2 1 2 1 2 1 0 2 1 0 0 0 1]
True Labels:      [1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0 0 0 1 0 0 2 1
 0 0 0 2 1 1 0 0 1 2 2 1 2 1 2 1 0 2 1 0 0 0 1]
Testing Accuracy: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

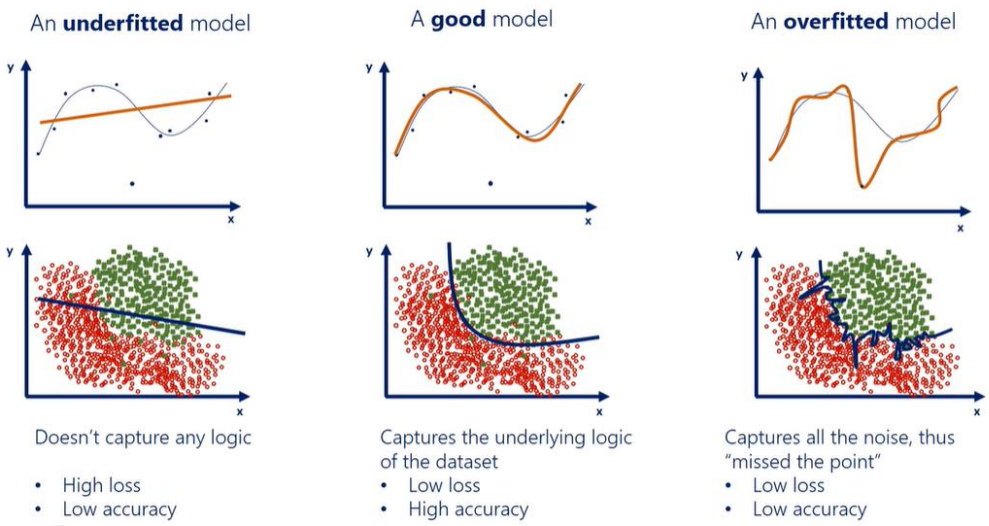
```
[0.96666667 0.96666667 0.9        0.96666667 1.        ]
0.96 accuracy with a standard deviation of 0.03
```

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()))
```

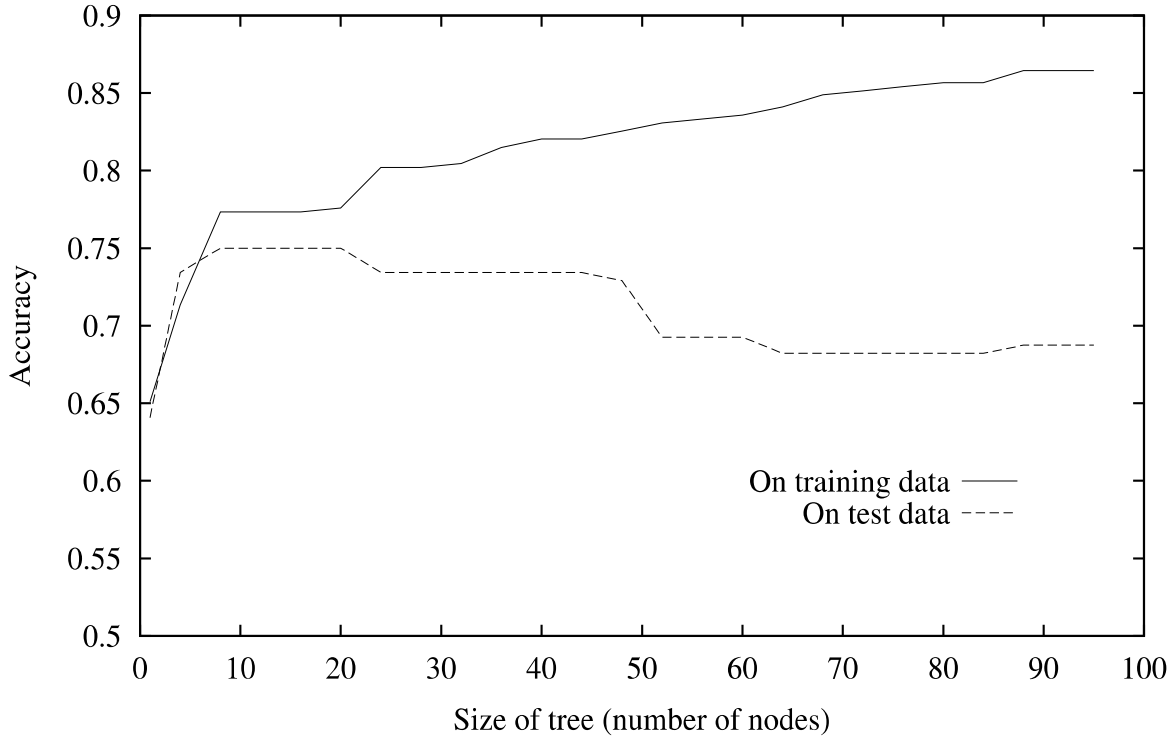
```
[1.   1.   1.   1.   1.   1.   0.67 1.   1.   1.   1.   1.   1.   1.
 1.   1.   1.   1.   1.   0.67 0.67 1.   1.   1.   1.   1.   0.67 0.67
 1.   0.67 1.   1.   1.   0.67 1.   1.   1.   1.   0.67 1.   1.   1.
 1.   1.   1.   1.   1.   1.   1.   1.  ]
0.95 accuracy with a standard deviation of 0.12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

Underfitting & Overfitting Issues



Issues in Decision Tree Learning

- Learning a tree that classifies the training data perfectly (with pure leaves) may lead to **over-fitting** of the training data (lower **generalization**)
  - There may be noise in the training data the tree is fitting
  - The algorithm might be making decisions based on very little data



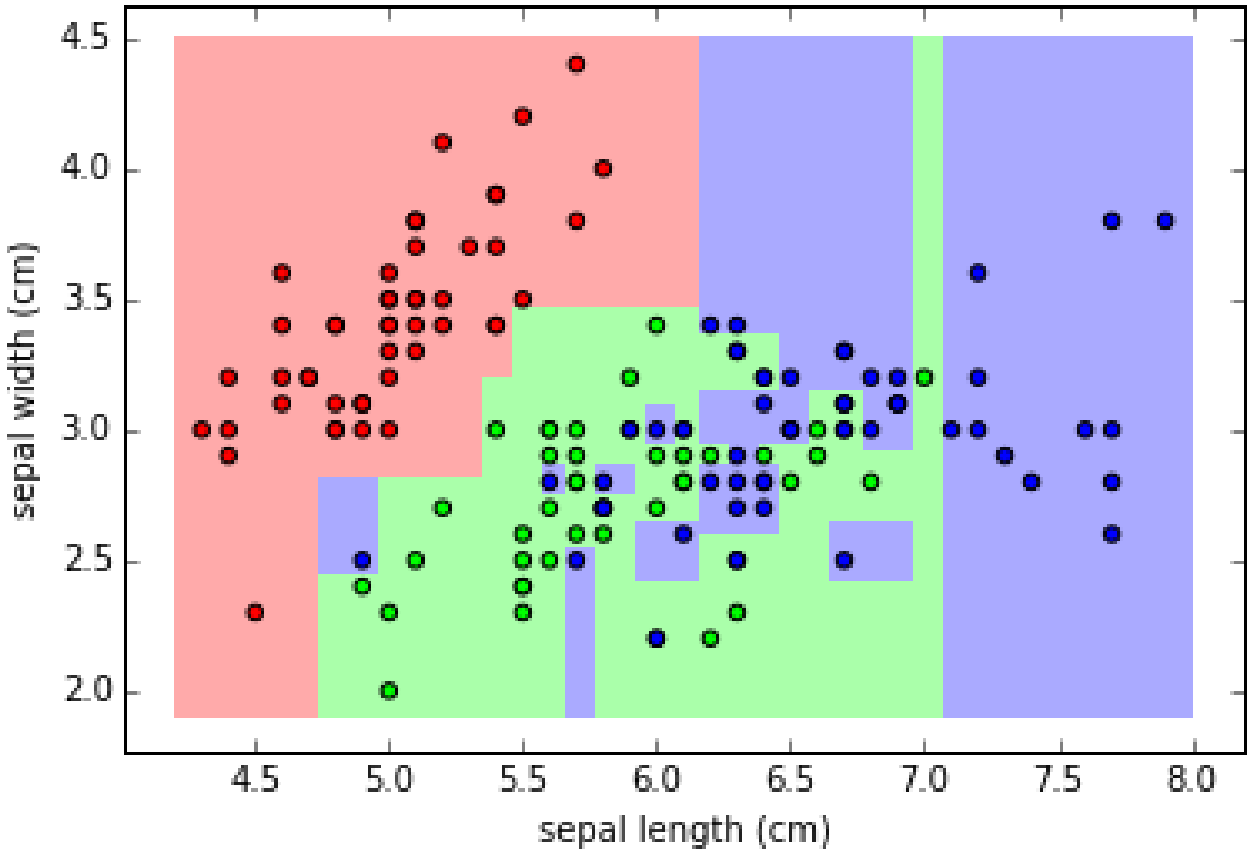
- A decision tree overfits the training data when its accuracy on the training data goes up but its accuracy on unseen test data goes down

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

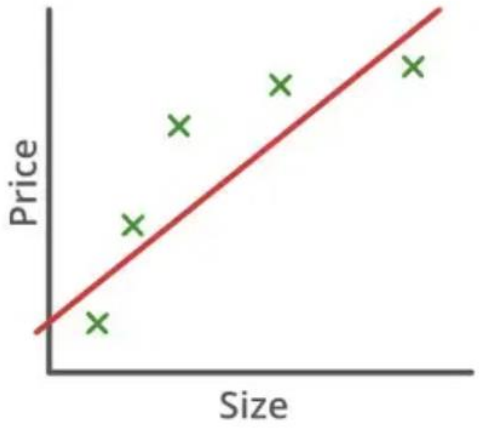
If we keep splitting, we will be reducing the error, but…



Causes overfitting => decision tree that is more complex than necessary

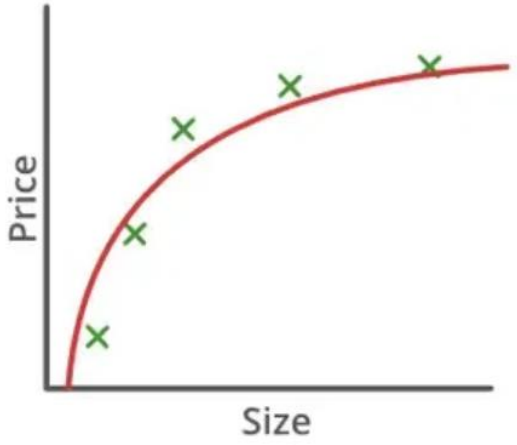
Result is in poor accuracy for unseen samples

Underfitting vs. Overfitting



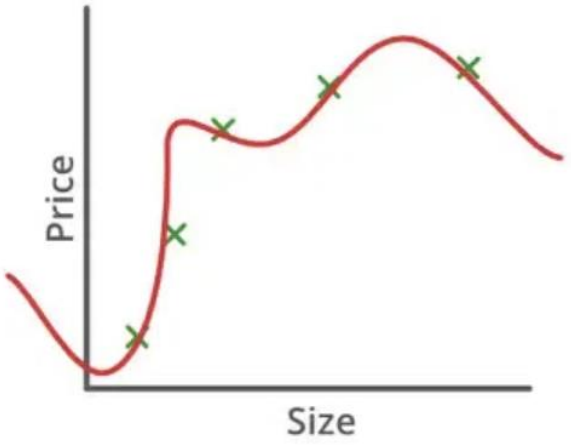
$\theta_0 + \theta_1 x$

High Bias
(Underfitting)



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

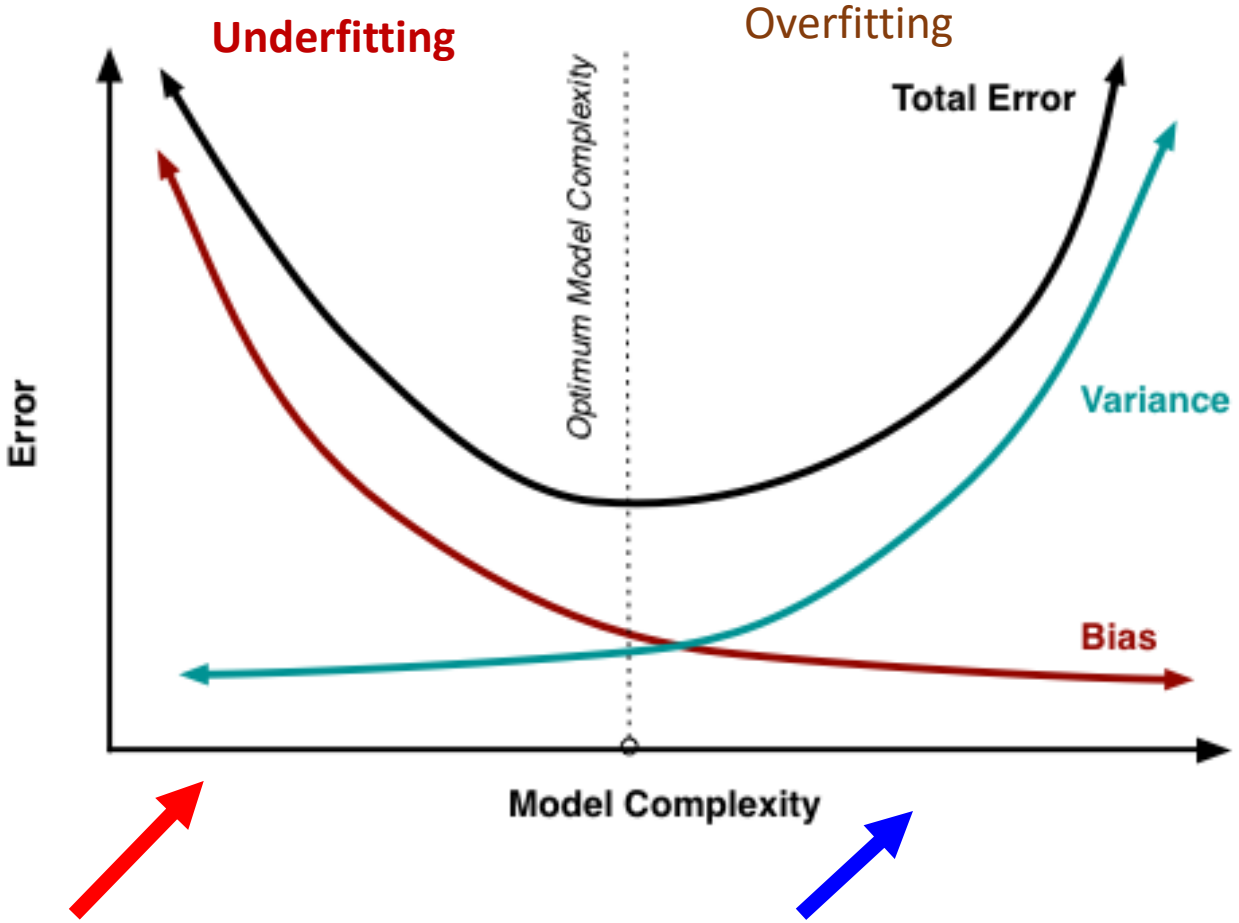
Low Bias, Low Variance
(Goodfitting)



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

High Variance
(Overfitting)

Underfitting and Overfitting



Simple models:
High bias and low variance

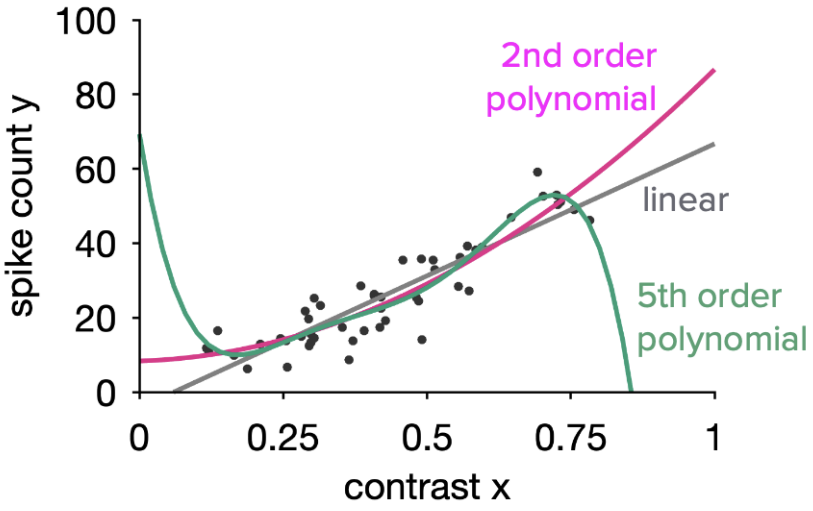
Complex models:
High variance and low bias

Trade-off: bias-variance errors

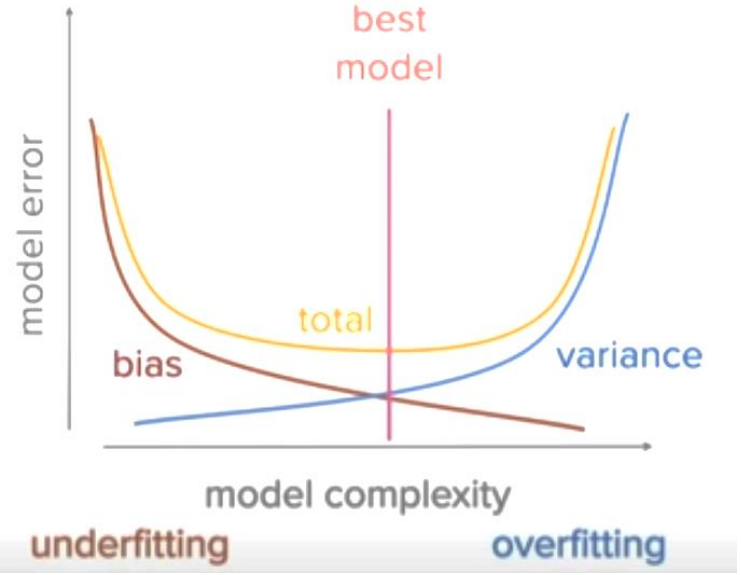
Bias-Variance tradeoff

Bias-variance Trade-off

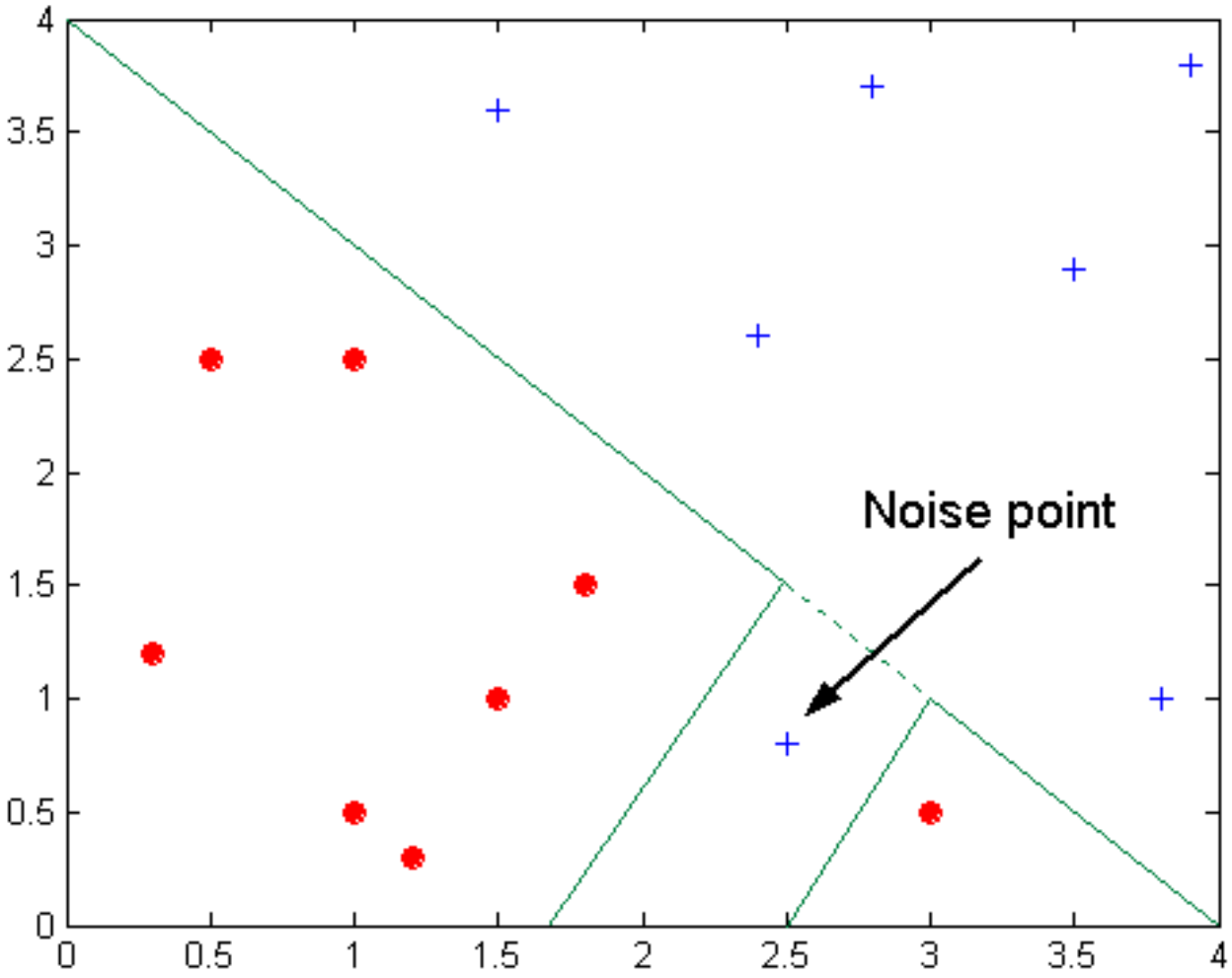
Total error = **bias** + **variance**



Best model: balances bias / variance

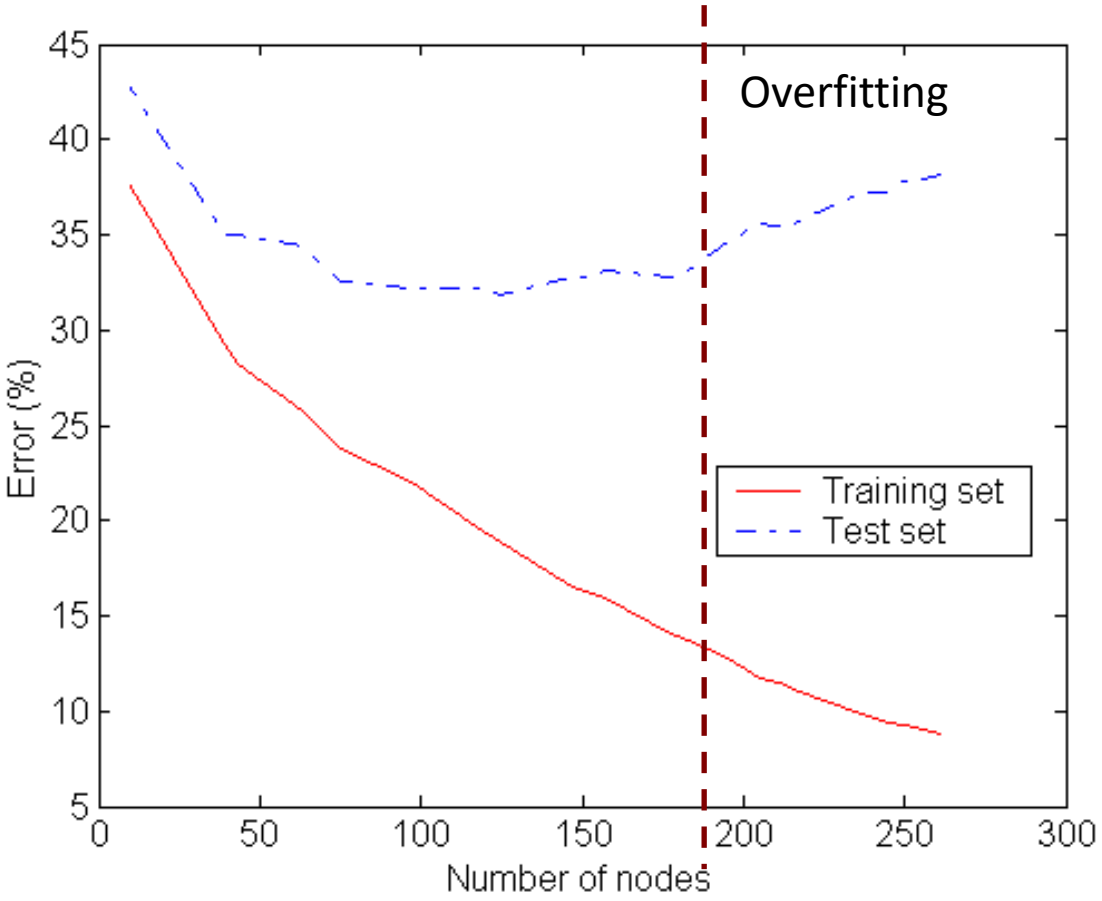


Overfitting due to Noise



Decision boundary is distorted by noise point

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

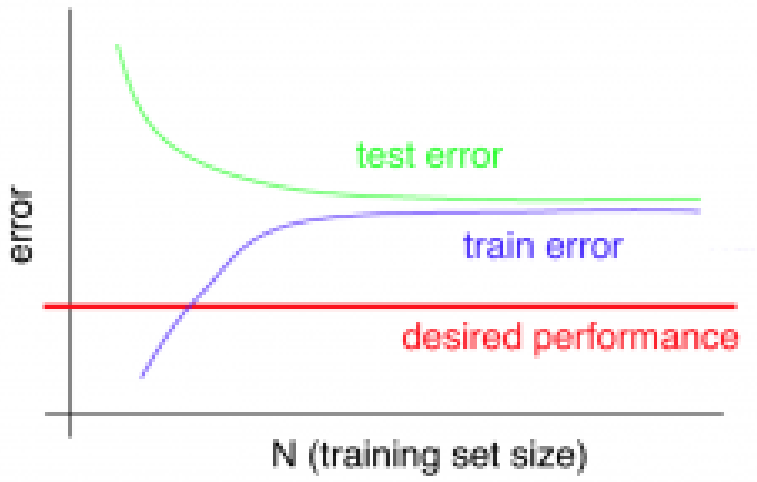


Underfitting: when model is too simple, both training and test errors are large

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.

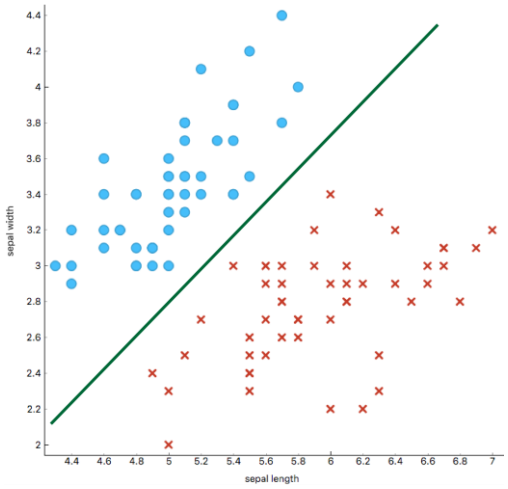
Typical learning curve for high variance (at fixed model complexity)

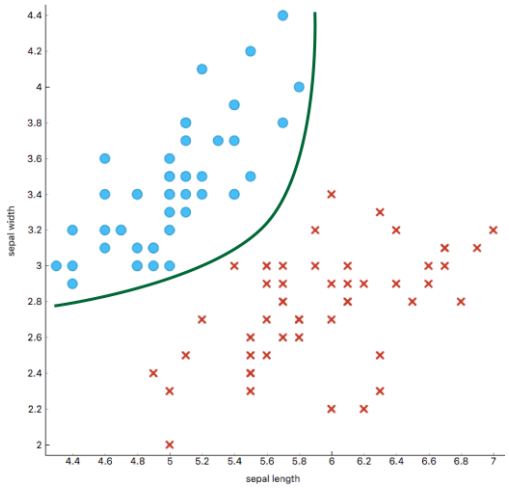
Diagnosing ML Models

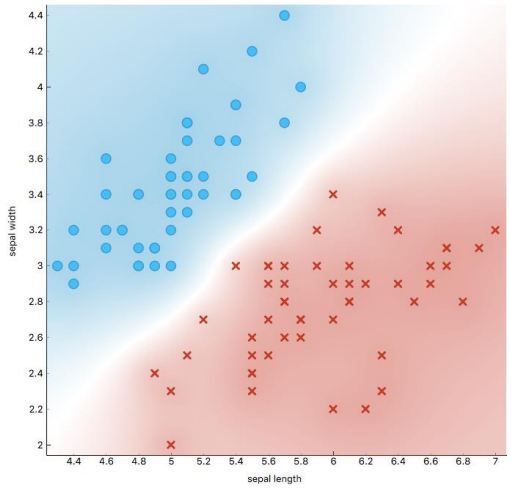
- **Simple model** → prone to underfitting (high bias & low variance).
  - It is computationally cheaper.
  - The model fits poorly consistently
  - Fixes:
    - Add features: to have sufficient information, e.g., Predict housing price from only size and no other features is hard even for human to do.
    - 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.

Occam's Razor Principle

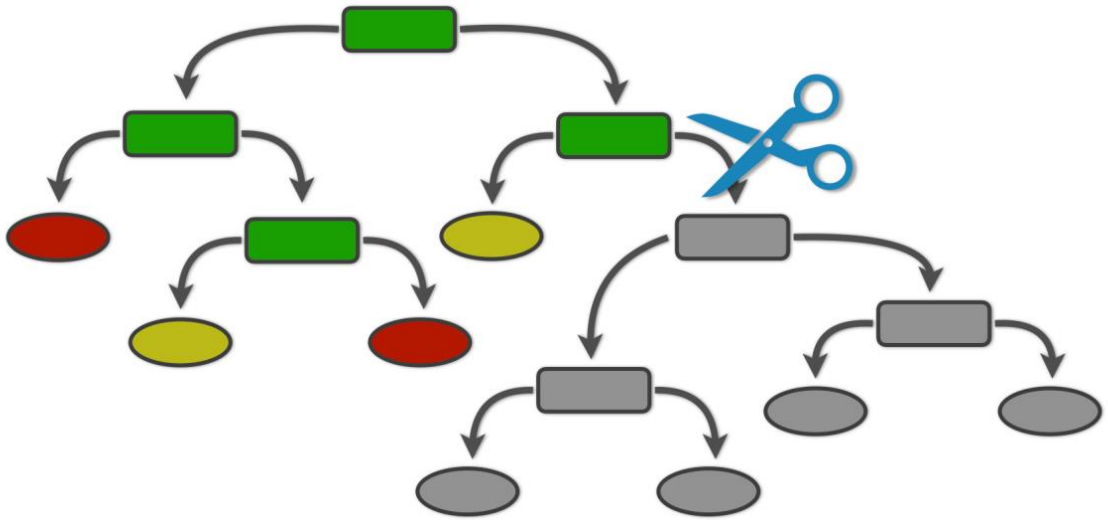
- In the context of evaluating ML models, Occam's Razor suggests that among competing models with similar predictive performance, **the simpler model should be preferred over the more complex model**
  - Simpler models are easier to understand, interpret, and maintain
  - Overly complex models may capture noise or irrelevant patterns in the training data (overfitting). Instead, select models that strike a balance between model complexity and generalization performance on unseen data
  - Therefore, one should include model complexity when evaluating a model







Tree Pruning



Pruning DT

- *Overfitting* the training data can be reduced by pruning the tree
  - Simplifying a Decision Tree making it smaller and more robust
- Two pruning strategies:
  - ***Pre-pruning***: stop building the tree before it is "complete"
  - ***Post-pruning***: build a full tree and then “cut back” certain subtrees and replace them with leaf nodes
- Pre-pruning is much faster but can stop prematurely, post-pruning is more accurate

Pre-pruning strategies

Stop building the tree (and make a leaf):

- At **max** tree depth
- When **number of instances in a leaf** is less than certain minimum
- When the change in Error rate, Gini index, or Entropy is below some threshold
- When **percentage of majority class** at a node if more than n%

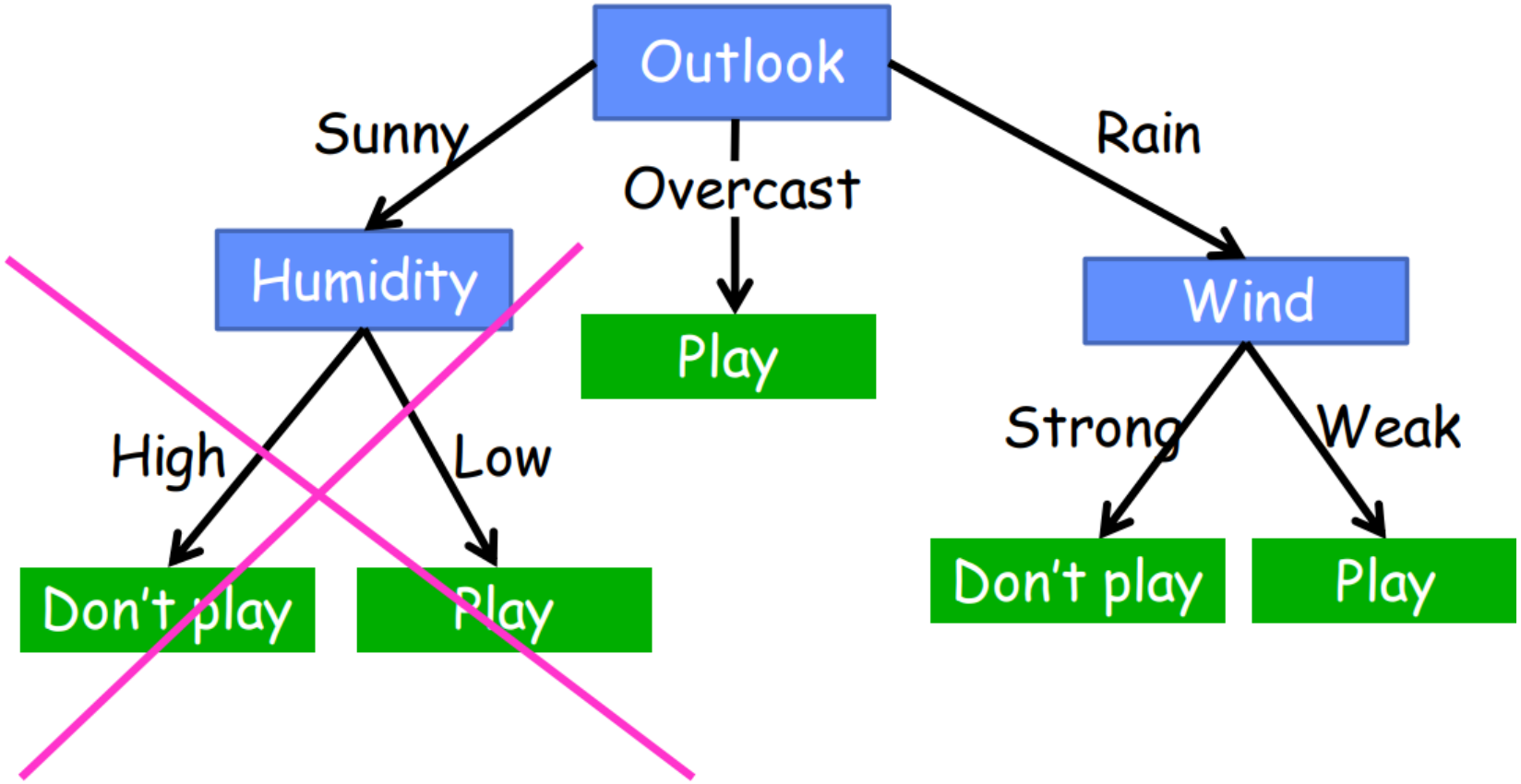
Post-pruning

- Build a full tree, then prune
- Pruning methods:
  - reduced error pruning
  - cost complexity pruning
  - pessimistic pruning
  - …and many others

Reduced Error Pruning Method

- Split data into train and validation set
- Repeat until pruning is harmful (i.e., decreases accuracy)
  - Remove each subtree and replace it with majority class and evaluate accuracy on the validation set
  - Remove subtree that leads to largest improvement in accuracy

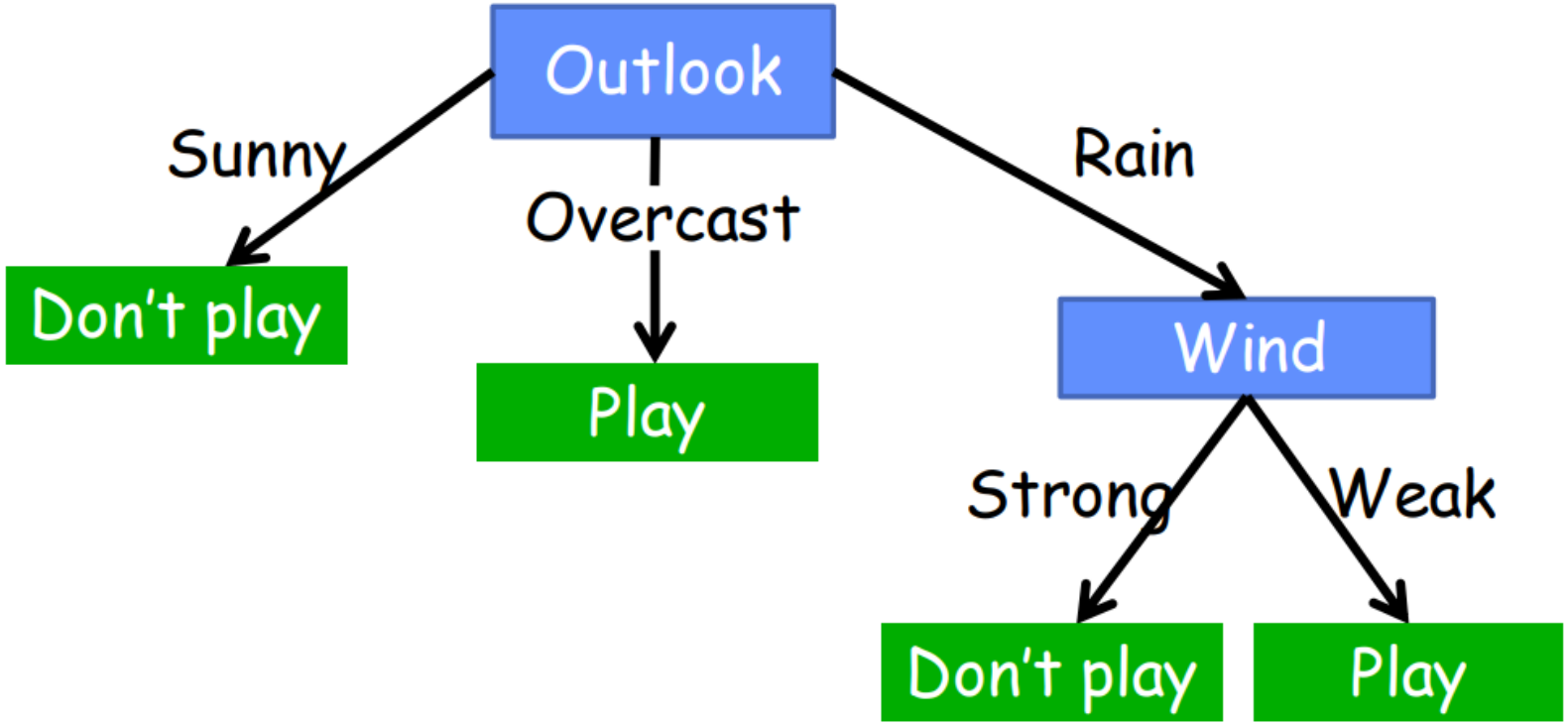
Reduced Error Pruning - Example



Validation set accuracy = 0.75

- Accuracy before pruning is 0.75

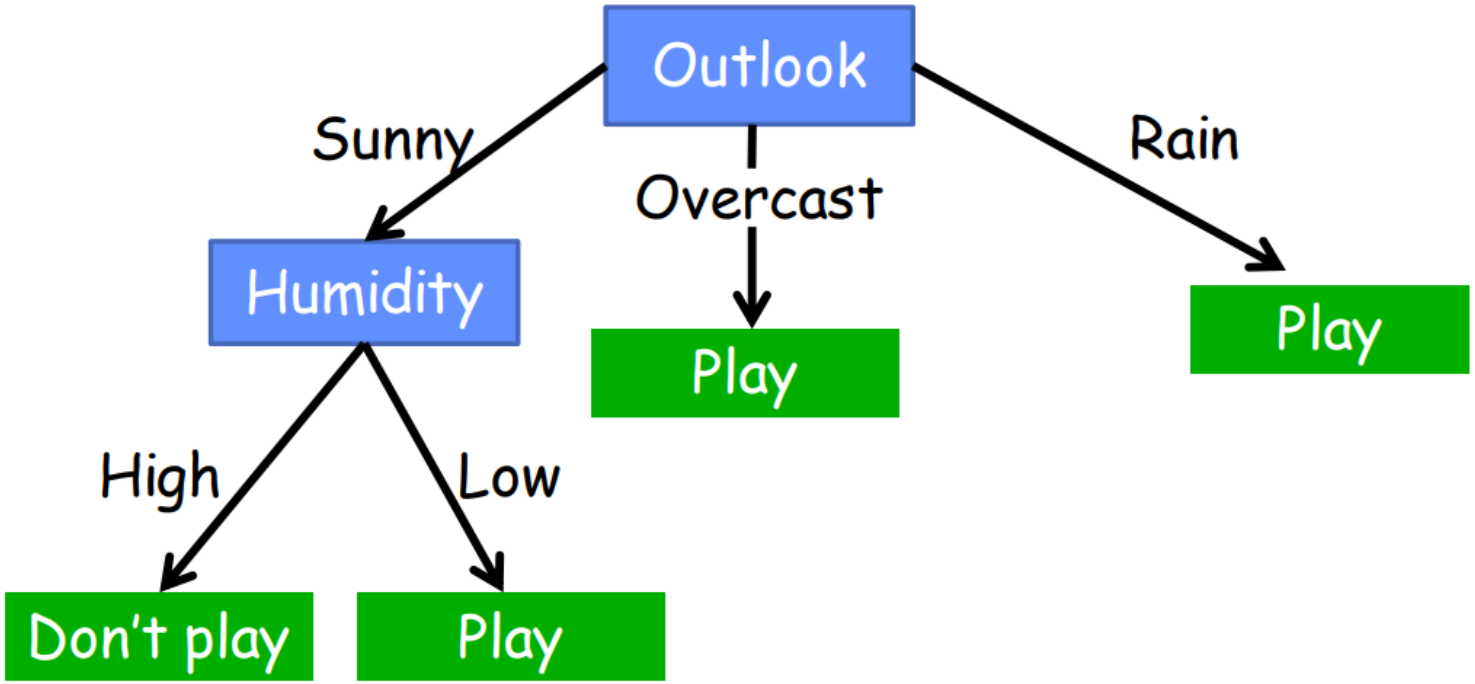
Reduced Error Pruning - Example



Validation set accuracy = 0.80

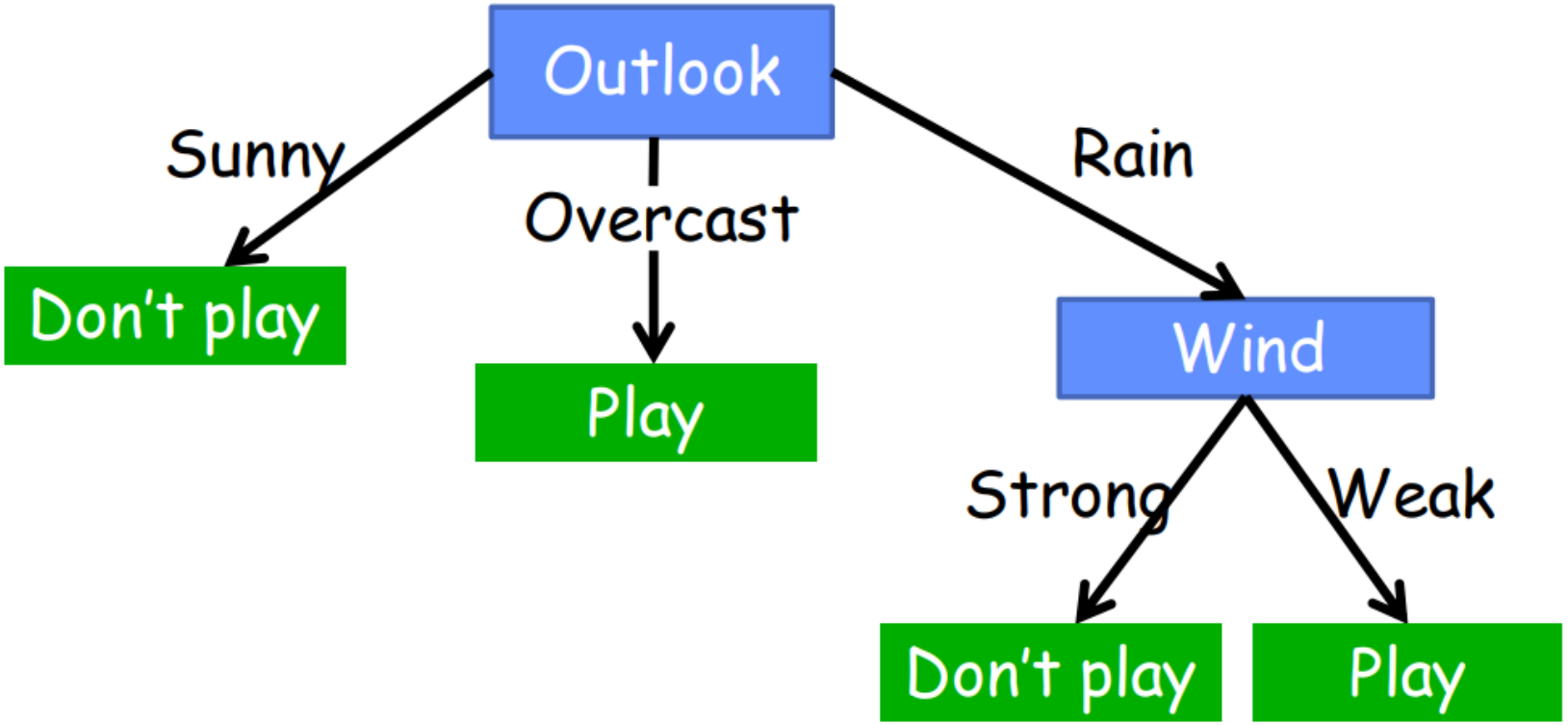
- Accuracy after pruning Humidity subtree is 0.8

Reduced Error Pruning - Example



Validation set accuracy = 0.70

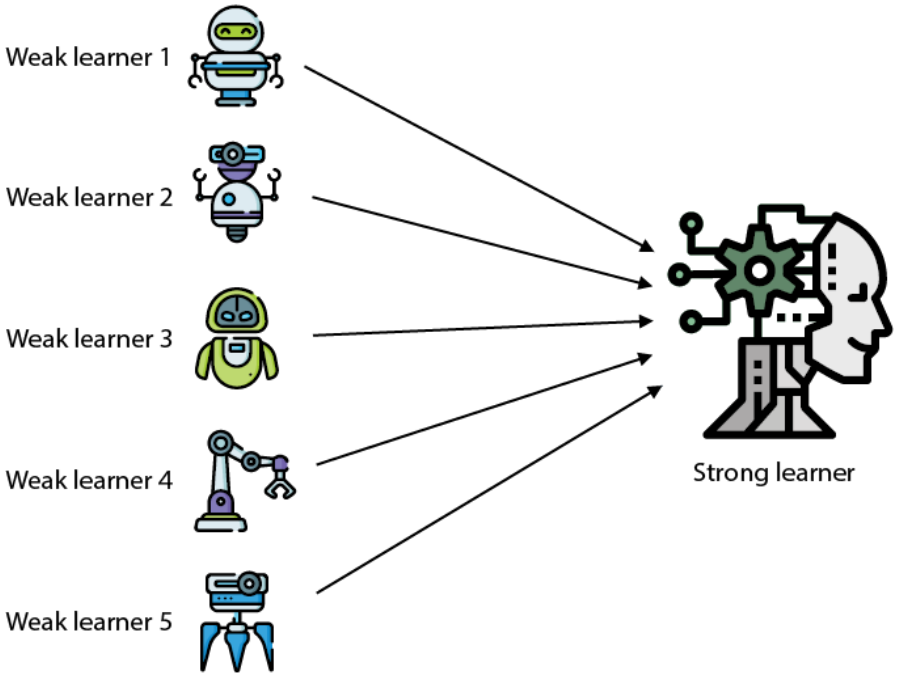
- Accuracy after pruning Wind subtree is 0.8



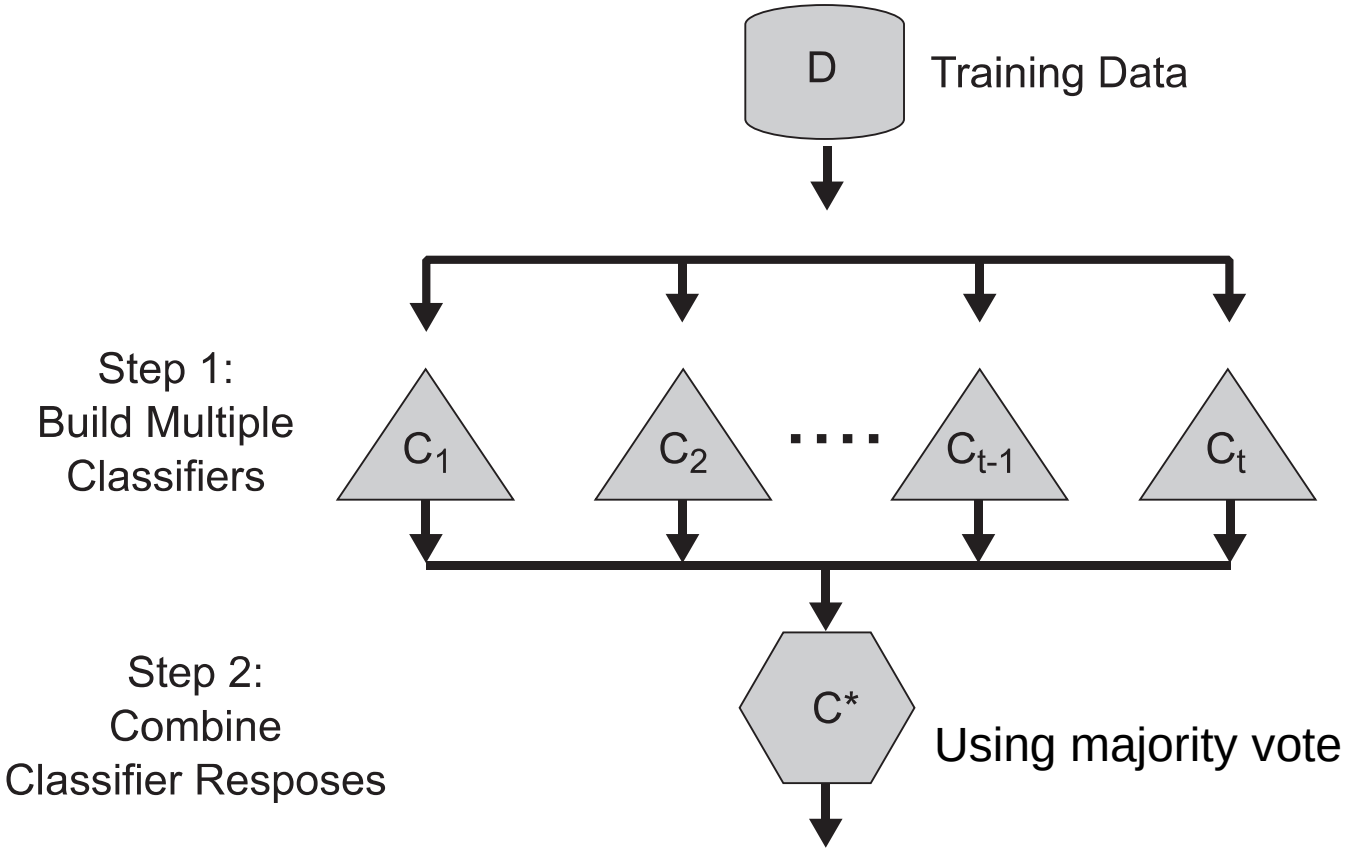
Use this as final tree

- Only prune Humidity subtree as this leads to the highest accuracy of 0.8

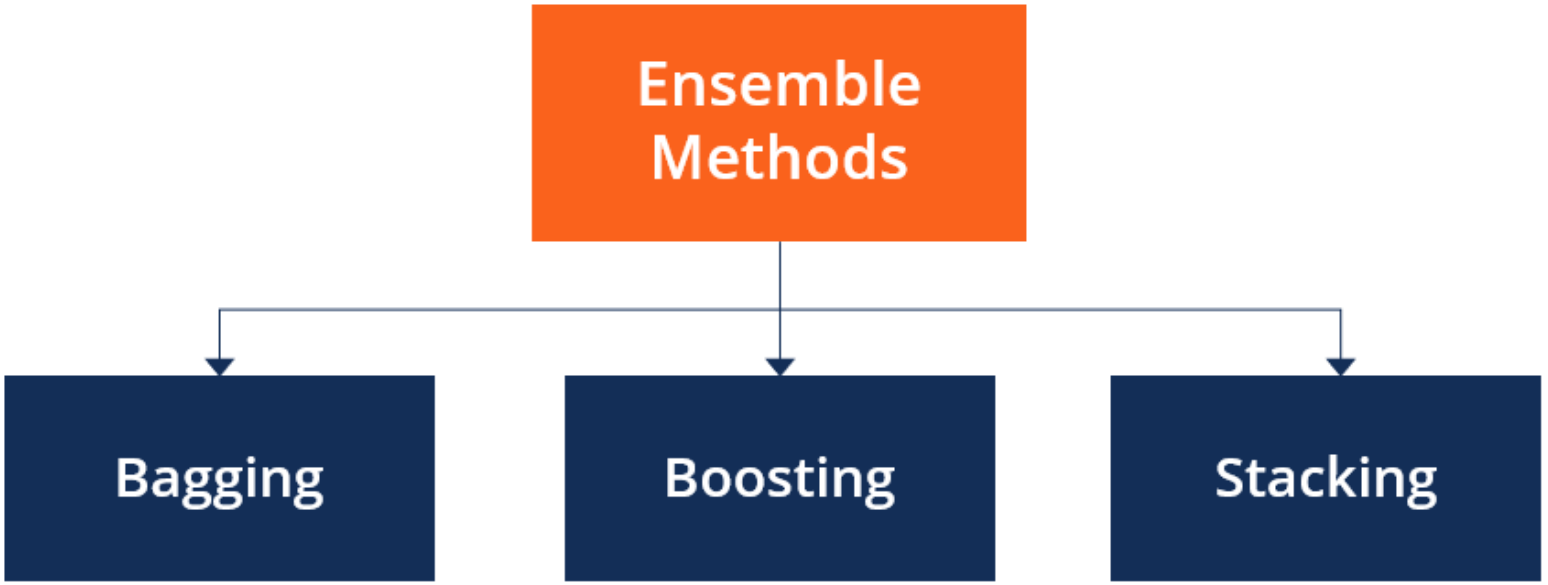
Ensemble Learning



General Approach of Ensemble Learning



- Basic Idea: Instead of learning one model, learn several and combine them
- Ensemble methods try to reduce the (overfitting) variance of complex models by *aggregating* responses of multiple base classifiers



Homogeneous Learners

Parallel Learning

e.g. Random Forest

Homogeneous Learners
Sequential Learning
e.g. Gradient Boosted Trees

Heterogeneous Learners
Parallel Learning

Bagging (Bootstrap AGGregatING)

Bootstrapping

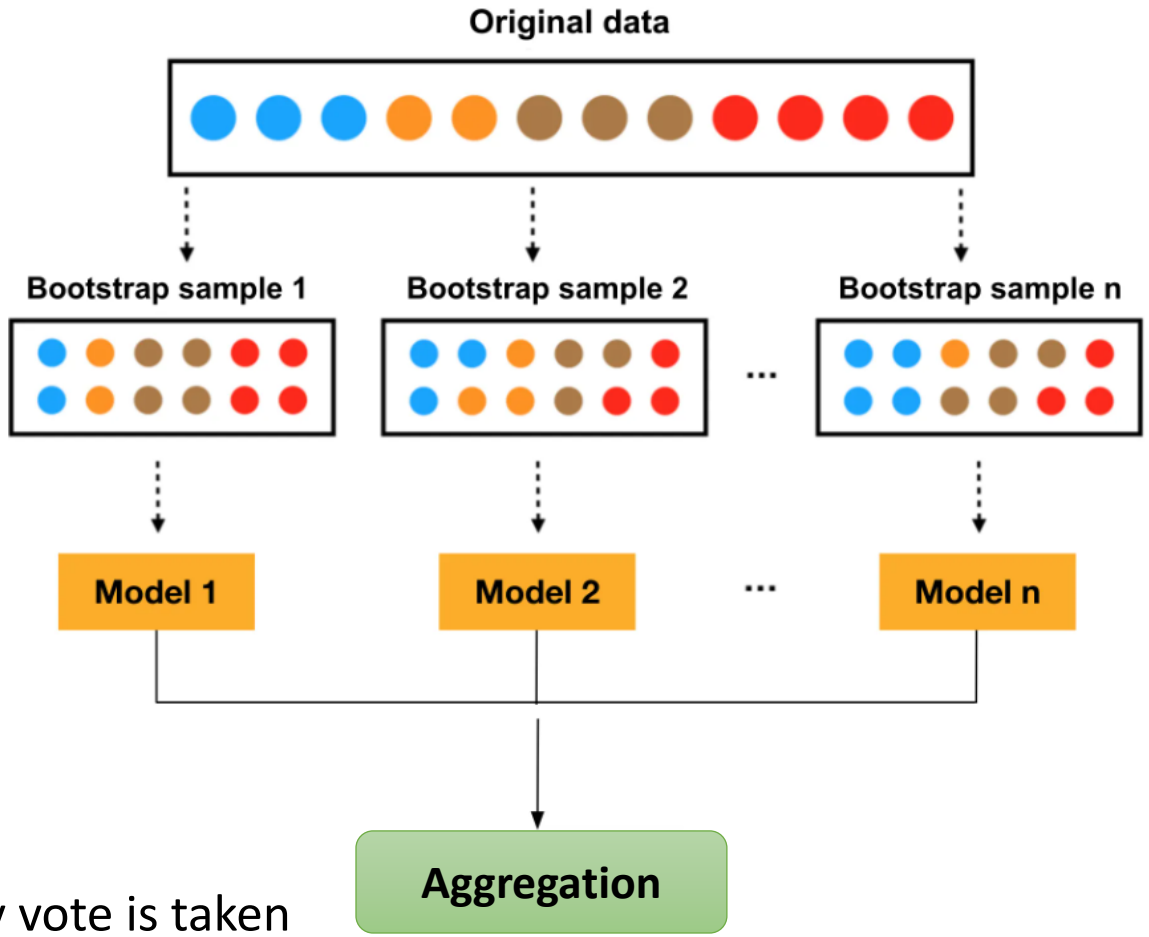
(sampling with replacement)

Model Training

(train a model on each bootstrap using one type of algorithm such as Decision Tree, Logistic Regression)

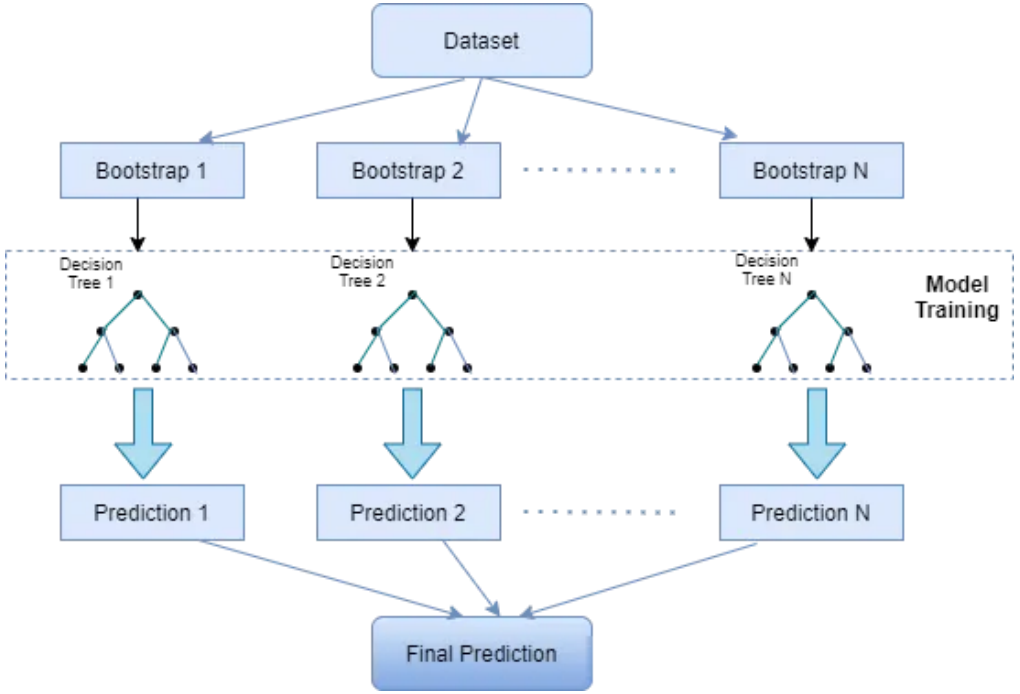
Aggregation

(Classification: Majority vote is taken
Regression: Average of outputs)



Random forest: an example implementation of bagging

- Construct multitude of decision trained on a random subset of the data and makes independent predictions
- The final prediction is determined by **aggregating** the predictions of individual trees, such as through a majority vote for classification tasks or averaging for regression tasks



- **Pros:**

- Improve accuracy over single models; reduces overfitting (variance)

- Easy to parallelize

- **Cons**: Difficult to interpret the resulting model

Random Forst

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

- Boosting allow **incrementally** building an ensemble
  - In each iteration, it sequentially train a **new model by focusing more on previously misclassified instances**
  - Initially, all **N** instances are assigned equal weights (equal chance for being selected for training)
  - The mis-classified instances are assigned a **weight higher** than the correctly classified instances

- Example 4 was misclassified in round 2

- Its weight is increased. Therefore, it is more likely to be chosen again in subsequent

Original Data	1	2	3	4	5	6	7	8	9	10
Boosting (Round 1)	7	3	2	8	7	9	4	10	6	3
Boosting (Round 2)	5	4	9	4	2	5	1	7	4	2
Boosting (Round 3)	4	4	8	10	4	5	4	6	3	4

Pros:

Reduces bias and variance

In some cases, boosting has been shown to yield better accuracy than bagging

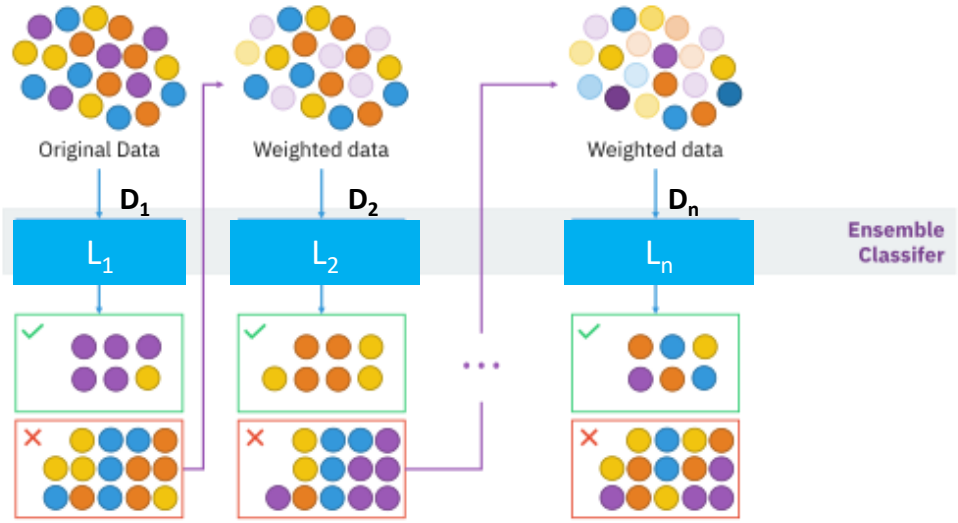
Cons:

Tends to be more likely to overfit the training data

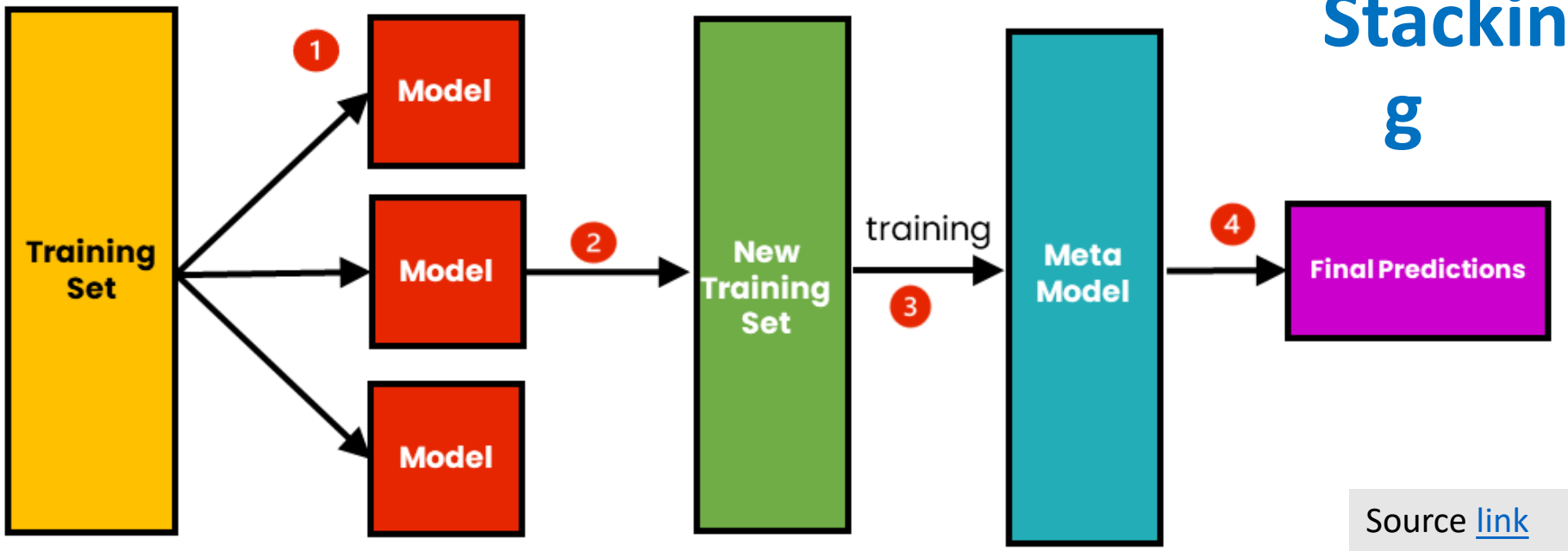
Sensitive to noisy data and outliers

Boosting

- Equal weight is given to the training data instances (D_1) at the starting round
  - (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 learner (L_2) and so on
- The results are then combined in the form of weighted voting, based on how well a learner performs



Stacking



Source link

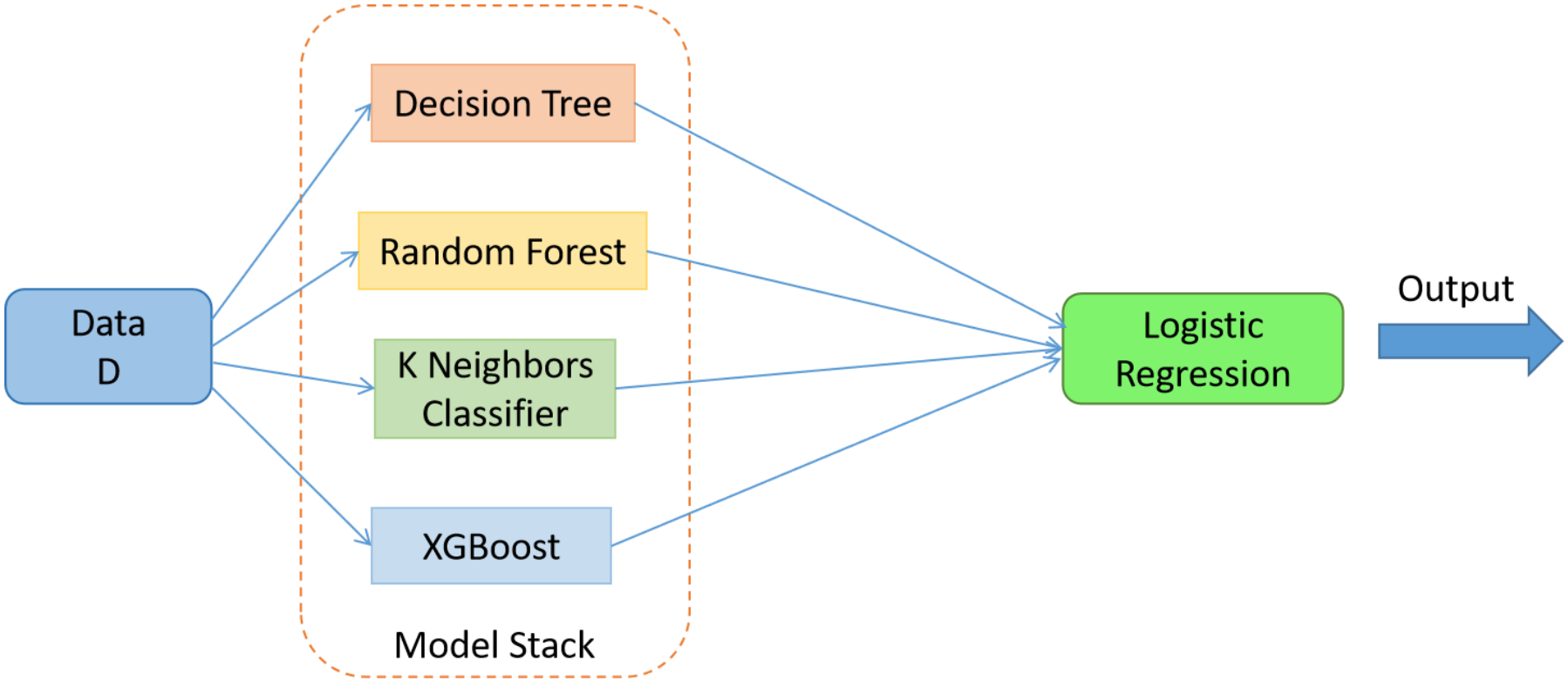
Stacking: training a ML model that combines the predictions of several base models trained using the same training data

1. **Base Models**: Train several diverse base models on the training data using different algorithms
2. **Generate Predictions**: Use each base model to make predictions on the validation set
3. **Meta-Model Training**: Combine the predictions from the base models and use them to train a meta-model (typically a simple model like logistic regression, or decision tree)
4. **Prediction**: Use the trained meta-model to make final predictions on unseen data

Stacking

- The main goal of stacking is to **improve the predictive performance** by leveraging the strengths of several base models, often yielding better performance than any single model in the ensemble.
- It can help the model capture non-linear patterns and interactions between features that might be missed by individual models.

Stacking Example



- Multiple machine learning algorithms (e.g., Decision Tree, XGBoost) are used as the base models trained on the same dataset
- Then we use the trained models to predict using the testing set
- The output of the base models on the testing set are then used to train the meta-model (Logistic regression in this example)