

# Topic 5

# Model Improvement

---

IT8302 APPLIED MACHINE LEARNING

# Learning Outcomes

---

- ❑ Apply cross-validation
  - Explain leave-one-out cross-validation
  - Explain k-fold cross-validation
- ❑ Apply hyperparameter tuning
  - Apply grid search cv
  - Apply random search cv
- ❑ Apply regularization
  - Curse of Dimensionality (CoD)
  - Explain L1 regularization
  - Explain L2 regularization
- ❑ Data Scaling (Feature Scaling)

# Learning Outcomes

---

- Apply ensemble learning
  - Explain bagging and random forest
  - Explain boosting and gradient boosted trees
  - Explain stacking

# Cross-Validation

---

# TRAINING/TEST DATA SPLIT

---

Talked about splitting data in training/test sets

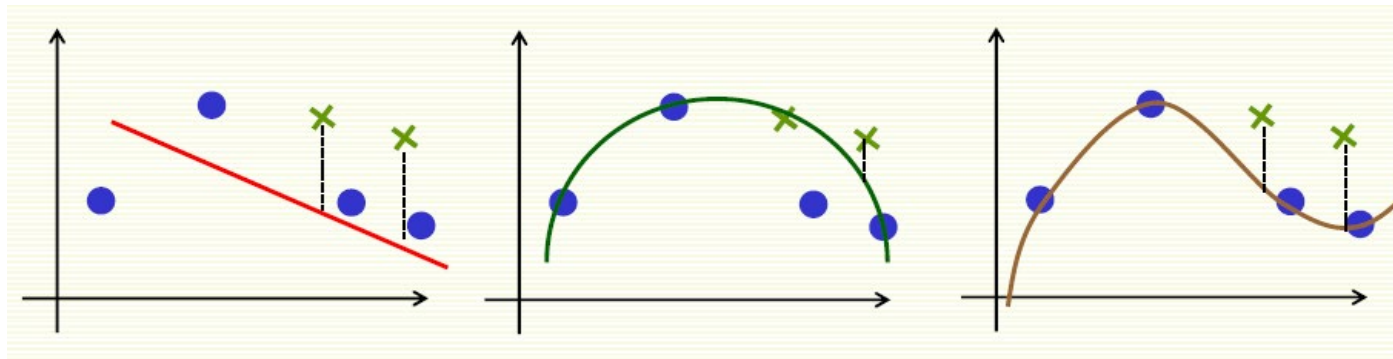
- training data is used to fit parameters
- test data is used to assess how classifier generalizes to new data

What if classifier has “non-tunable” parameters?

- a parameter is “non-tunable” if tuning (or training) it on the training data leads to overfitting

# TRAINING/TEST DATA SPLIT

---



What about test error? Seems appropriate

- degree 2 is the best model according to the test error

Except what do we report as the test error now?

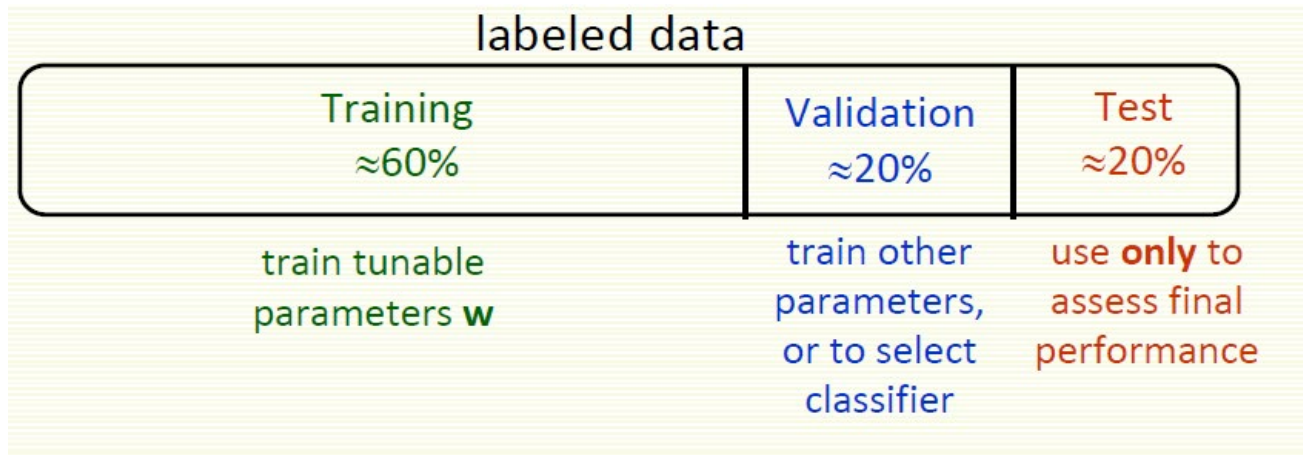
- Test error should be computed on data that was **not used for training at all**
- Here used “test” data for training, i.e. choosing model

# VALIDATION DATA

---

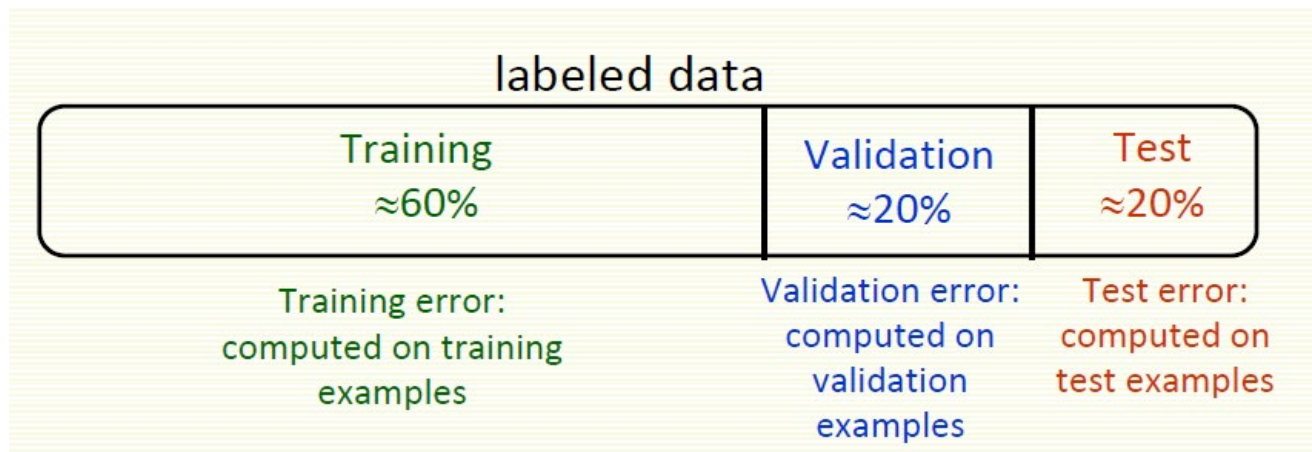
Same question when choosing among several classifiers

- our polynomial degree example can be looked at as choosing among 3 classifiers (degree 1, 2, or 3)
- Solution: split the labeled data into three parts



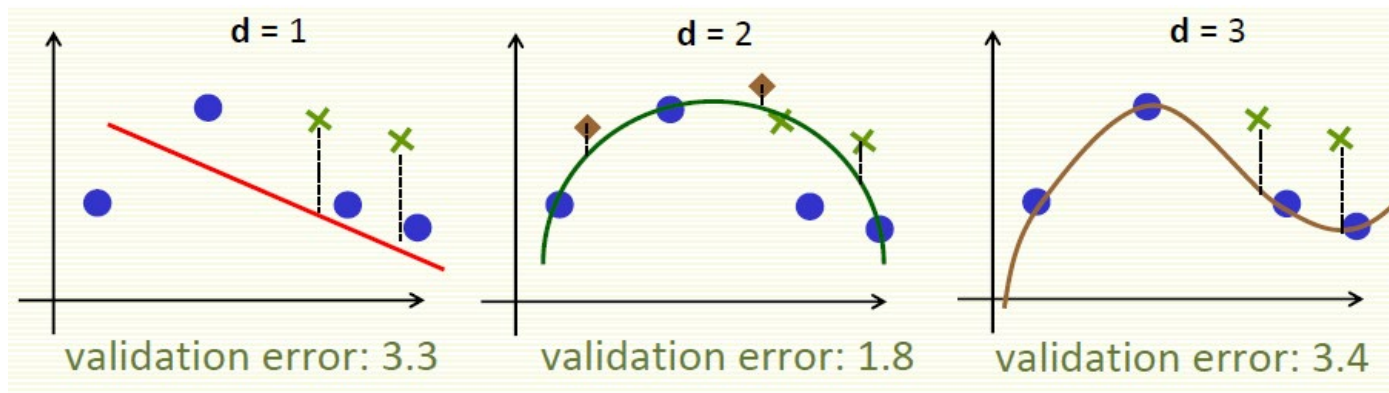
# TRAINING/ VALIDATION

---





# Training/Validation/Test Data



Training Data

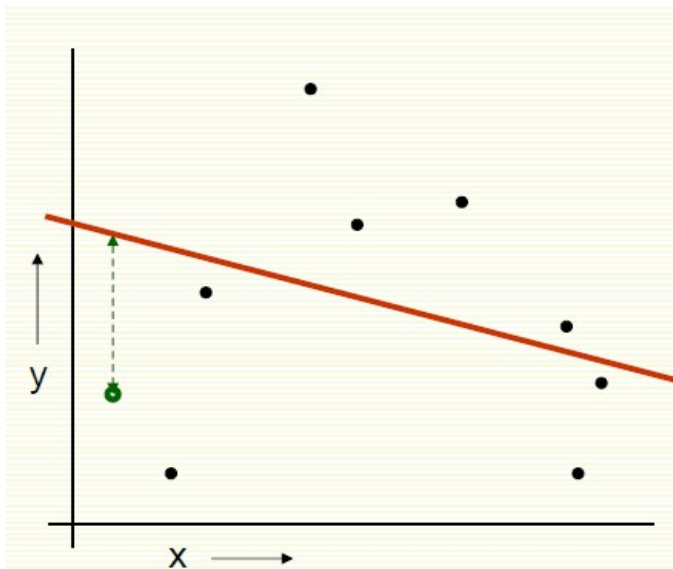
Validation Data

$d = 2$  is chosen

Test Data

1.3 test error computed for  $d = 2$

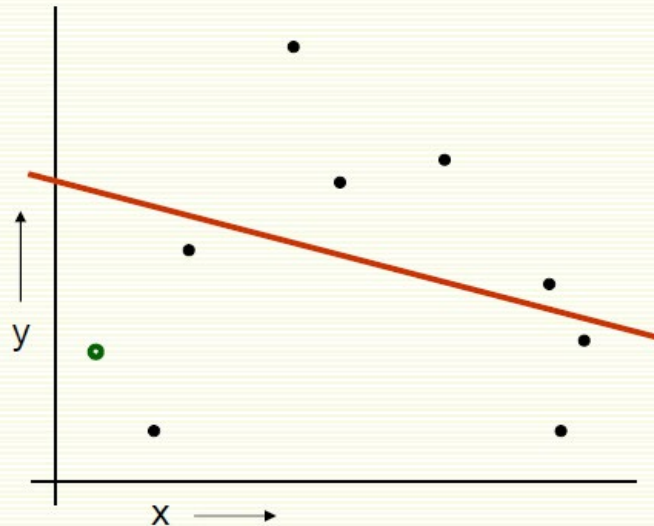
# LOOCV (Leave-one-out Cross Validation)



For  $k=1$  to  $n$

1. Let  $(\mathbf{x}^k, \mathbf{y}^k)$  be the  $k$ th example
2. Temporarily remove  $(\mathbf{x}^k, \mathbf{y}^k)$  from the dataset
3. Train on the remaining  $n-1$  examples
4. Note your error on  $(\mathbf{x}^k, \mathbf{y}^k)$

# LOOCV (Leave-one-out Cross Validation)



For  $k=1$  to  $n$

1. Let  $(\mathbf{x}^k, \mathbf{y}^k)$  be the  $k$ th example
2. Temporarily remove  $(\mathbf{x}^k, \mathbf{y}^k)$  from the dataset
3. Train on the remaining  $n-1$  examples
4. Note your error on  $(\mathbf{x}^k, \mathbf{y}^k)$

When you've done all points,  
report the mean error

# Which kind of Cross Validation?

---

	Downside	Upside
Test-set	may give unreliable estimate of future performance	cheap
Leave-one-out	expensive	doesn't waste data

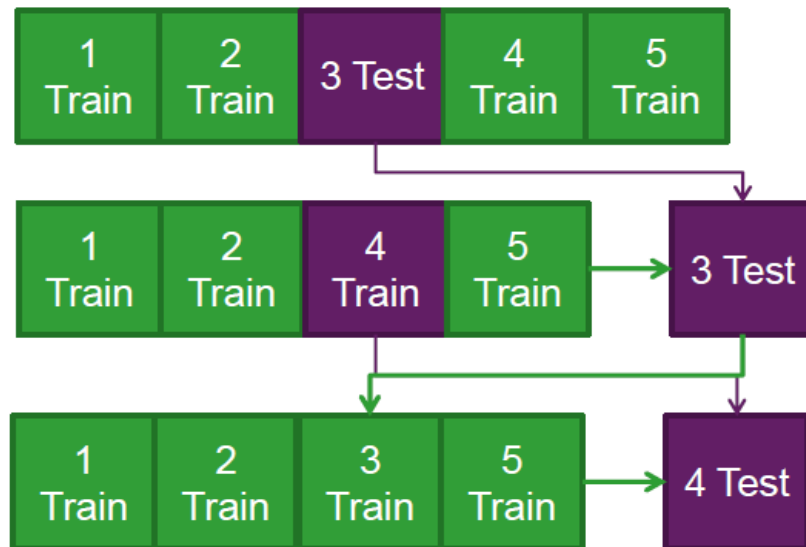
# K-FOLD CROSS VALIDATION

---

- Since data are often scarce, there might not be enough to set aside for a validation sample
- To work around this issue k-fold CV works as follows:
  1. Split the sample into  $k$  subsets of equal size
  2. For each fold estimate a model on all the subsets except one
  3. Use the left out subset to test the model, by calculating a CV metric of choice
  4. Average the CV metric across subsets to get the CV error
- This has the advantage of using all data for estimating the model, however finding a good value for  $k$  can be tricky

# K-fold Cross Validation Example

---



1. Split the data into 5 samples
2. Fit a model to the training samples and use the test sample to calculate a CV metric.
3. Repeat the process for the next sample, until all samples have been used to either train or test the model

# Which kind of Cross Validation?

---

	Downside	Upside
Test-set	may give unreliable estimate of future performance	cheap
Leave-one-out	expensive	doesn't waste data
10-fold	wastes 10% of the data, 10 times more expensive than test set	only wastes 10%, only 10 times more expensive instead of $n$ times
3-fold	wastes more data than 10-fold, more expensive than test set	slightly better than test-set
N-fold	Identical to Leave-one-out	

# Hyperparameter tuning

---



# Using sklearn GridSearchCV

---

[https://scikit-learn.org/stable/modules/generated/sklearn.model\\_selection.GridSearchCV.html](https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html)

[https://scikit-learn.org/stable/modules/grid\\_search.html#grid-search](https://scikit-learn.org/stable/modules/grid_search.html#grid-search)

- Hyper-parameters are parameters that are not directly learnt within estimators. In scikit-learn they are passed as arguments to the constructor of the estimator classes. Typical examples include C, kernel and gamma for Support Vector Classifier, alpha for Lasso, etc.
- It is possible and recommended to search the hyper-parameter space for the best cross validation score.
- GridSearchCV exhaustively **considers all parameter** combination

# Using sklearn RandomSearchCV

---

[https://scikit-learn.org/stable/modules/generated/sklearn.model\\_selection.RandomizedSearchCV.html](https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html)

- RandomizedSearchCV can **sample a given number** of candidates from a parameter space with a specified distribution.

# Tuning the hyperparameters

---

A search consists of:

- an estimator (regressor or classifier such as `sklearn.svm.SVC()`);
- a parameter space;
- a method for searching or sampling candidates;
- a cross-validation scheme; and
- a score function.

Specifically, to find the names and current values for all parameters for a given estimator, use: `estimator.get_params()`

```
# Grid Search for Algorithm Tuning
import numpy as np
from sklearn import datasets
from sklearn.linear_model import Ridge
from sklearn.model_selection import GridSearchCV
# load the diabetes datasets
dataset = datasets.load_diabetes()
# prepare a range of alpha values to test
alphas = np.array([1, 0.1, 0.01, 0.001, 0.0001, 0])
# create and fit a ridge regression model, testing each alpha
model = Ridge()
grid = GridSearchCV(estimator=model, param_grid=dict(alpha=alphas))
grid.fit(dataset.data, dataset.target)
print(grid)
# summarize the results of the grid search
print(grid.best_score_)
print(grid.best_estimator_.alpha)
```

```
# Randomized Search for Algorithm Tuning
import numpy as np
from scipy.stats import uniform as sp_rand
from sklearn import datasets
from sklearn.linear_model import Ridge
from sklearn.model_selection import RandomizedSearchCV
# load the diabetes datasets
dataset = datasets.load_diabetes()
# prepare a uniform distribution to sample for the alpha parameter
param_grid = {'alpha': sp_rand()}
# create and fit a ridge regression model, testing random alpha values
model = Ridge()
rsearch = RandomizedSearchCV(estimator=model,
                             param_distributions=param_grid, n_iter=100)
rsearch.fit(dataset.data, dataset.target)
print(rsearch)
# summarize the results of the random parameter search
print(rsearch.best_score_)
print(rsearch.best_estimator_.alpha)
```

# Regularization

---

# Curse of Dimensionality (CoD)


---

**Curse of Dimensionality** refers to a set of problems that arise when working with high-dimensional data. The dimension of a dataset corresponds to the number of attributes/features that exist in a dataset. A dataset with a large number of attributes, generally of the order of hundred or more, is referred to as high dimensional data.









1. If we have more features than observations than we run the risk of massively overfitting our model — this would generally result in terrible out of sample performance.
2. When we have too many features, observations become harder to cluster — believe it or not, too many dimensions causes every observation in your dataset to appear equidistant from all the others.

Source: <https://towardsdatascience.com/the-curse-of-dimensionality-50dc6e49aa1e>

# Curse of Dimensionality (CoD)

	Reddish	Bluish
	1	0
	1	0
	1	0
	1	0
	0	1
	0	1
	0	1

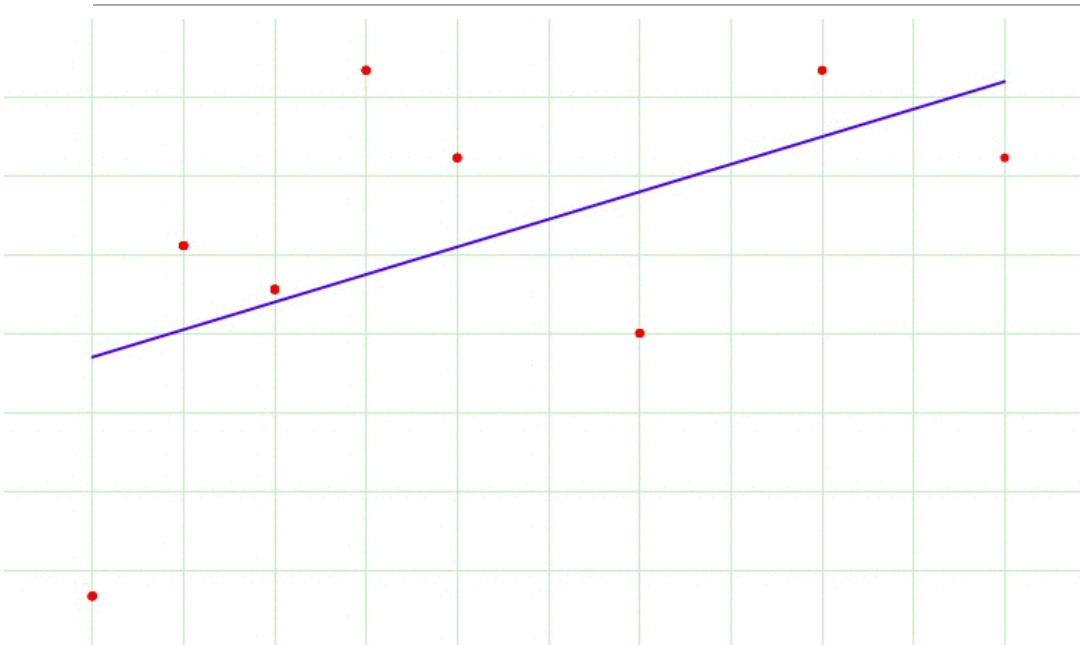
The algorithm can learn the relationship between color and reddish vs bluish

	Red	Maroon	Pink	Flamingo	Blue	Turquoise	Seaweed	Ocean
	1	0	0	0	0	0	0	0
	0	1	0	0	0	0	0	0
	0	0	1	0	0	0	0	0
	0	0	0	1	0	0	0	0
	0	0	0	0	1	0	0	0
	0	0	0	0	0	1	0	0
	0	0	0	0	0	0	1	0
	0	0	0	0	0	0	0	1

The algorithm cannot learn any pattern. Every row is unique



# Reducing overfitting



Overfitting happens when model learns signal as well as noise in the training data and wouldn't perform well on new data on which model wasn't trained on.

There are few ways you can avoid overfitting your model on training data like cross-validation sampling, reducing number of features, pruning, **regularization** etc.

# Regularization and CoD

---

**Regularization** is one way to avoid overfitting. However, in models where regularization is not applicable, such as decision trees and KNN, we can use feature selection and dimensionality reduction techniques to help us avoid the curse of dimensionality.

Essentially, **regularization** penalizes the classifier for paying attention to features that don't help much toward answering the question at hand, while favouring features that provide relevant information. This filters out the irrelevant and sparse sections of your data, leaving your classifier to only pay attention to the robust features of your dataset.

Source: <https://builtin.com/data-science/curse-dimensionality>

# Regularization

---

Regularization basically adds the penalty as model complexity increases. Regularization parameter (lambda) penalizes all the parameters except intercept so that model generalizes the data and won't overfit.

$$J(\theta) = \frac{1}{2m} \left[ \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^n \theta_j^2 \right]$$

$\min_{\theta} J(\theta)$

# L1 regularization

---

A regression model that uses L1 regularization technique is called Lasso Regression

Lasso Regression (Least Absolute Shrinkage and Selection Operator) adds “absolute value of magnitude” of coefficient as penalty term to the loss function.

$$\sum_{i=1}^n (Y_i - \sum_{j=1}^p X_{ij} \beta_j)^2 + \lambda \sum_{j=1}^p |\beta_j|$$

The key difference between these techniques is that Lasso shrinks the less important feature's coefficient to zero thus, removing some feature altogether. So, this works well for feature selection in case we have a huge number of features.

# L2 regularization

---

A regression model which uses L2 is called Ridge Regression

Ridge regression adds “squared magnitude” of coefficient as penalty term to the loss function. Here the highlighted part represents L2 regularization element.

$$\sum_{i=1}^n (y_i - \sum_{j=1}^p x_{ij} \beta_j)^2 + \lambda \sum_{j=1}^p \beta_j^2$$

Here, if lambda is zero then you can imagine we get back OLS. However, if lambda is very large then it will add too much weight and it will lead to under-fitting. Having said that it's important how lambda is chosen. This technique works very well to avoid over-fitting issue.

# Data Scaling (Feature Scaling)

---

# Why data or feature scaling?

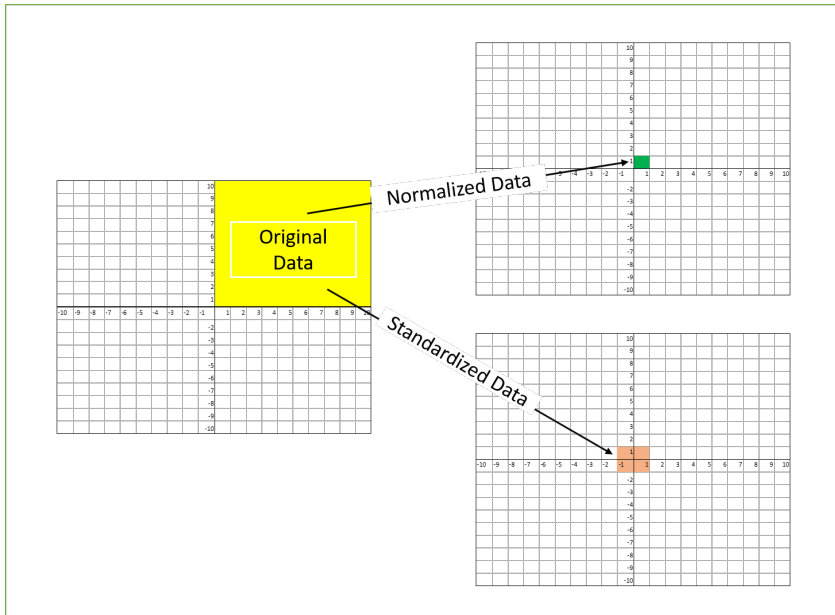
---

Machine learning algorithm just sees number — if there is a vast difference in the range say few ranging in thousands and few ranging in the tens, and it makes the underlying assumption that higher ranging numbers have superiority of some sort. So these **more significant number starts playing a more decisive role** while training the model.

The solution to this problem is to resize the numbers to a comparable range. Two common techniques are min-max scaling (normalization) and standardization.

source: <https://towardsdatascience.com/all-about-feature-scaling-bcc0ad75cb35>

# Feature Scaling



The most common techniques of feature scaling are **Normalization** and **Standardization**. Normalization is used when we want to bound our values between two numbers, typically, between [0,1] or [-1,1]. While Standardization transforms the data to have zero mean and a variance of 1, they make our data unitless.

$$x_{new} = \frac{x - x_{min}}{x_{max} - x_{min}} \quad \text{normalization}$$

$$x_{new} = \frac{x - \mu}{\sigma} \quad \text{standardization}$$



# Ensemble Learning

---

# Ensemble Learning

---

- Ensemble learning use multiple learning algorithms to obtain better predictive performance than could be obtained from any of the constituent learning algorithms alone.
- Common types of ensemble:
  - Bootstrap Aggregation (Bagging)
  - Boosting
  - Stacking

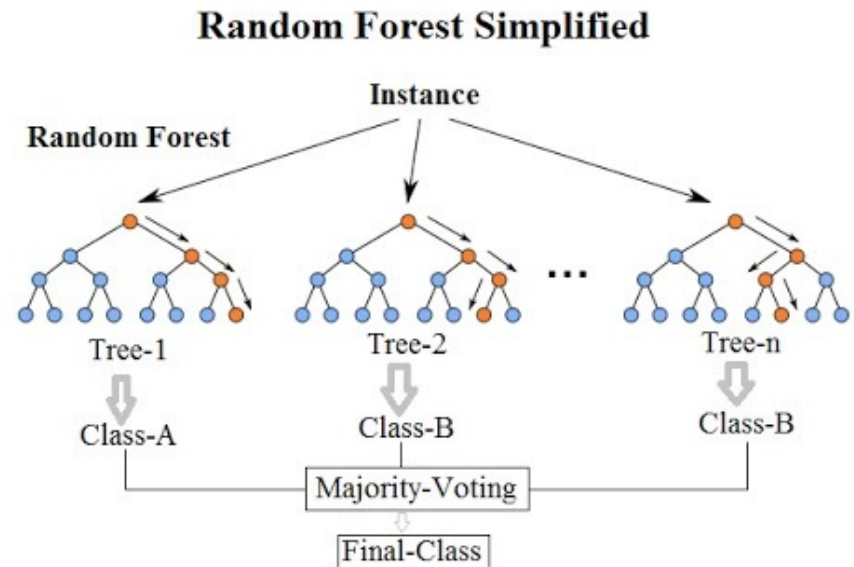
# Bagging

---

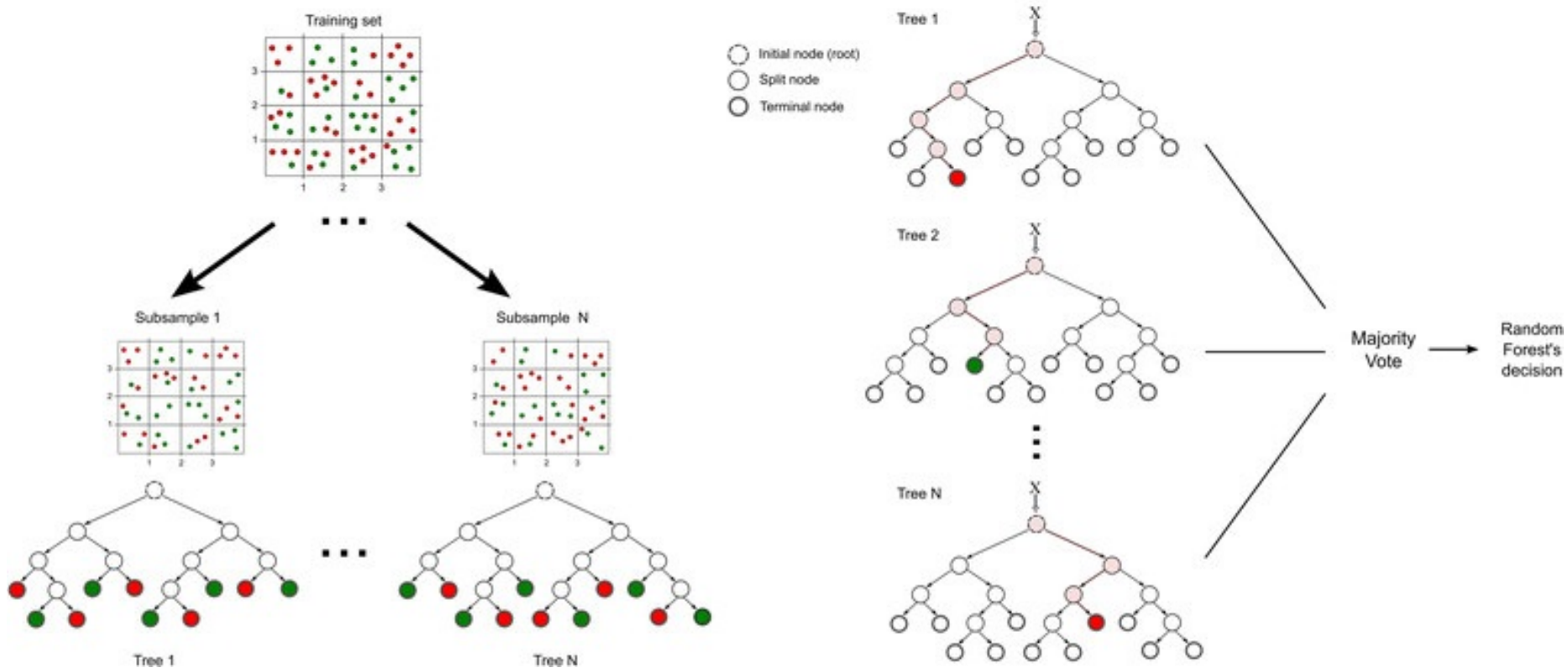
- Bootstrap aggregating, often abbreviated as **bagging**, involves having each model in the ensemble vote with equal weight. In order to promote model variance, bagging trains each model in the ensemble using a randomly drawn subset of the training set. As an example, the **random forest** algorithm combines random decision trees with bagging to achieve very high classification accuracy.
- In bagging the samples are generated in such a way that the samples are different from each other however **replacement is allowed**. Replacement means that an instance can occur in multiple samples multiple times or it can not appear in some samples at all. These samples are then given to multiple learners and then the results from each learner are combined in the form of voting.

# Random/Decision Forest

- Random/Decision forests (regression, two-class, and multiclass) are all based on decision trees.
- Because a feature space can be subdivided into arbitrarily small regions, it's easy to imagine dividing it finely enough to have one data point per region. This is an extreme example of overfitting.
- In order to avoid this, a **large set of trees** are constructed with special mathematical care taken that the trees are not correlated.
- The average of this "**decision forest**" is a tree that avoids overfitting. Decision forests can use a lot of memory.



# Random Forest (Visualization)



# Boosting

---

**Boosting** involves incrementally building an ensemble by training each new model instance to emphasize the training instances that previous models misclassified. In some cases, boosting has been shown to yield better accuracy than bagging, but it also tends to be more likely to overfit the training data. By far, the most common implementation of boosting is **Adaboost**, although some newer algorithms are reported to achieve better results.

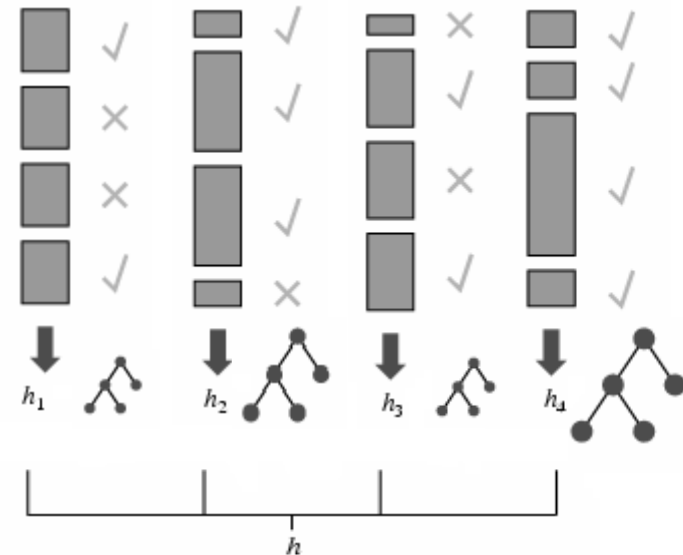
In Boosting, an equal weight (uniform probability distribution) is given to the sample training data (say D1) at the very starting round. This data (D1) is then given to a base learner (say L1). The misclassified instances by L1 are assigned a weight higher than the correctly classified instances, but keeping in mind that the total probability distribution will be equal to 1. This boosted data (say D2) is then given to second base learner (say L2) and so on. The results are then combined in the form of voting.

# Adaptive Boosting (AdaBoost)

Each rectangle corresponds to an example,  
with **weight proportional to its height**.

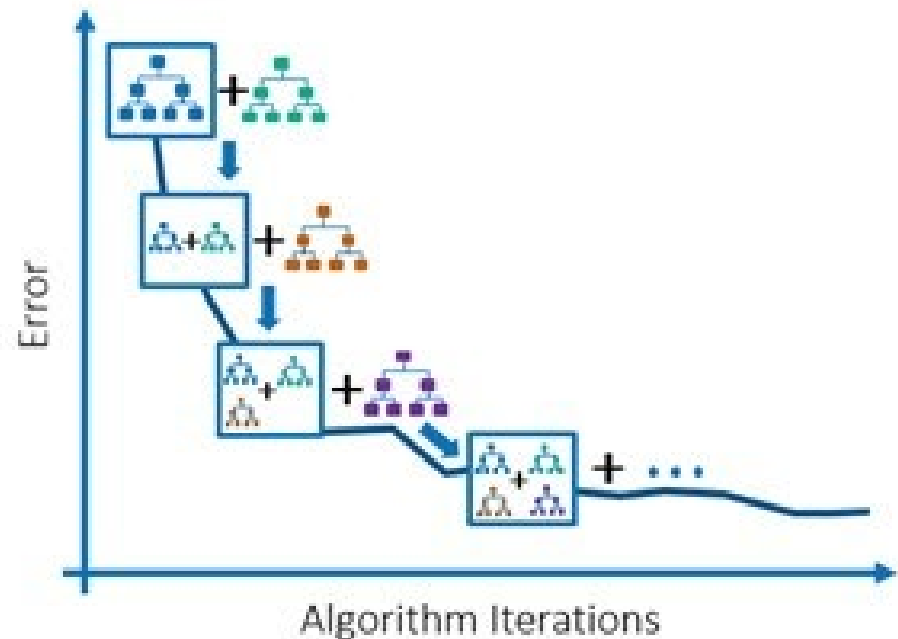
Crosses correspond to **misclassified** examples.

Size of decision tree indicates **the weight of that hypothesis** in the final ensemble.



# Gradient Boosting

- A boosted decision tree is an ensemble learning method in which the second tree corrects for the errors of the first tree, the third tree corrects for the errors of the first and second trees, and so forth.
- Generally, when properly configured, boosted decision trees are the easiest methods with which to get top performance on a wide variety of machine learning tasks.
- However, they are also one of the more memory-intensive learners.



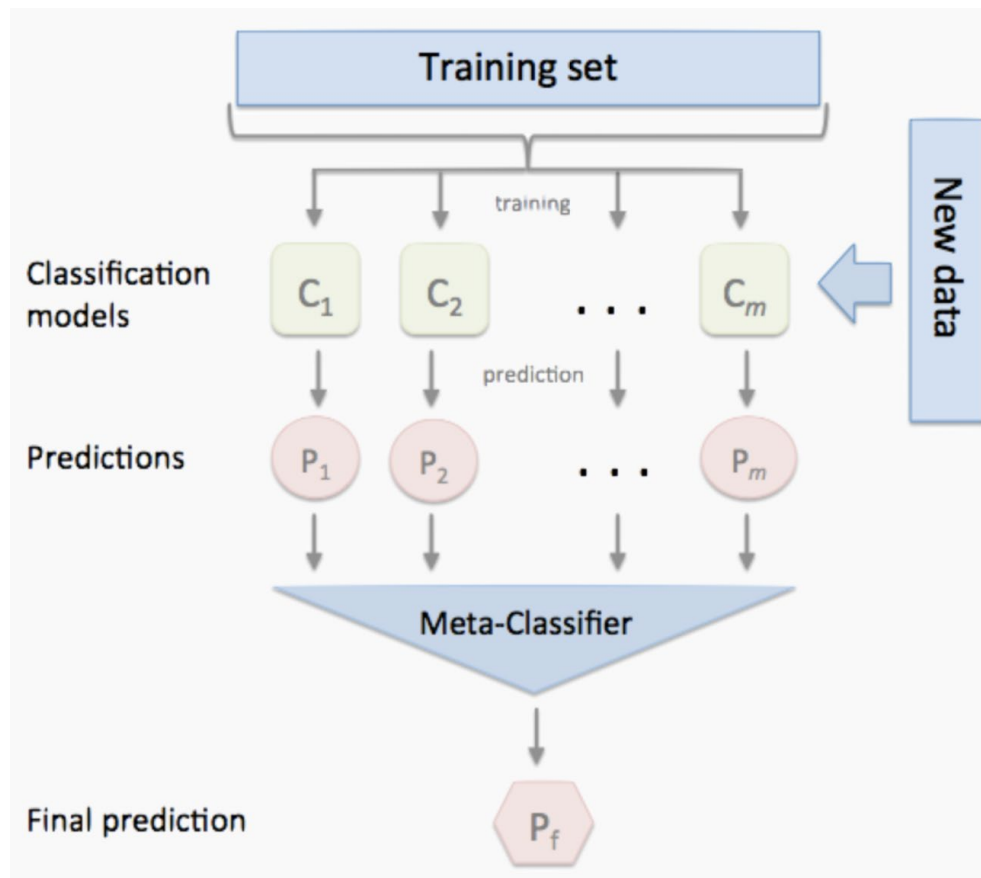


# Stacking

---

- **Stacking** (sometimes called stacked generalization) involves training a learning algorithm to combine the predictions of several other learning algorithms. First, all of the other algorithms are trained using the available data, then a combiner algorithm is trained to make a final prediction using all the predictions of the other algorithms as additional inputs.
- In practice, a logistic regression model is often used as the combiner.
- Stacking typically yields performance better than any single one of the trained models.

# Stacking: an Illustration



# Stacking with sklearn

---

- Unfortunately, Python does not have stacking algorithms implemented for you.
- So how can we do it?
- We can set it up by ‘manually’ fitting several base models, take the outputs of those models, and fitting the meta model on the outputs of those base models.
- It’s a model on models!

# Summary

---

What we have learnt:

- Cross-validation
- Hyperparameter tuning
- Regularization
- Ensemble Learning
  - Bagging
  - Boosting
  - Stacking