INT354: MACHINE LEARNING-I: NOTES

Unit VI:

The bias-complexity trade off: No Free Lunch Theorem, Error Decomposition, The VC-Dimension, The Rademacher Complexity, The Natarajan Dimension

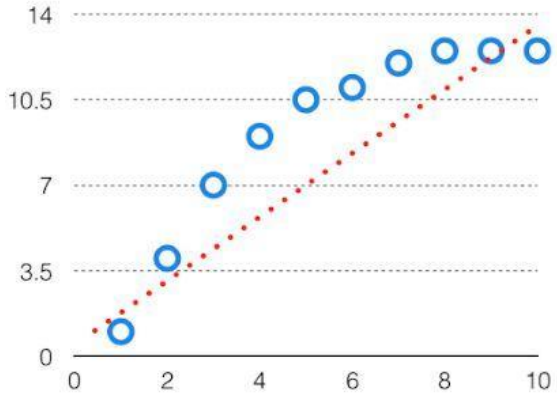
Algorithm-Independent machine Learning: Combining Classifiers, Majority Voting Classifier, Resampling for Estimating Statistics, Lack of Inherent Superiority of Classifier, Bagging and Boosting Classifier, Random Forest Classifier, Regressor, Support Vector Classifier and Regressor

The bias-complexity trade off:

It is important to understand prediction errors (bias and variance) when it comes to accuracy in any machine-learning algorithm. There is a tradeoff between a model's ability to minimize bias and variance which is referred to as the best solution for selecting a value of **Regularization** constant. A proper understanding of these errors would help to avoid the overfitting and underfitting of a data set while training the algorithm.

What is Bias?

The bias is known as the difference between the prediction of the values by the Machine Learning model and the correct value. Being high in biasing gives a large error in training as well as testing data. It recommended that an algorithm should always be low-biased to avoid the problem of underfitting. By high bias, the data predicted is in a straight line format, thus not fitting accurately in the data in the data set. Such fitting is known as the **Underfitting of Data**. This happens when the hypothesis is too simple or linear in nature. Refer to the graph given below for an example of such a situation.



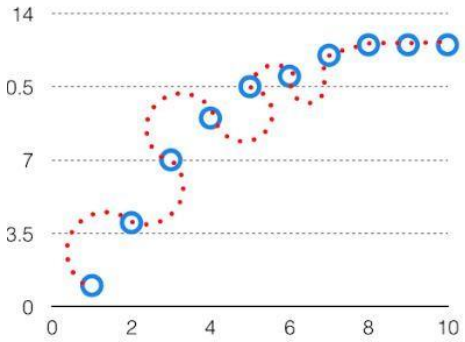
In such a problem, a hypothesis looks like follows.

$$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$$

What is Variance?

The variability of model prediction for a given data point which tells us the spread of our data is called the variance of the model. The model with high variance has a very complex fit to the training data and thus is not able to fit accurately on the data which it hasn't seen before. As a result, such models perform very well on training data but have high error rates on test data. When a model is high on variance, it is then said to as **Overfitting of Data**. Overfitting is fitting the training set accurately via complex curve and high order

hypothesis but is not the solution as the error with unseen data is high. While training a data model variance should be kept low. The high variance data looks as follows.



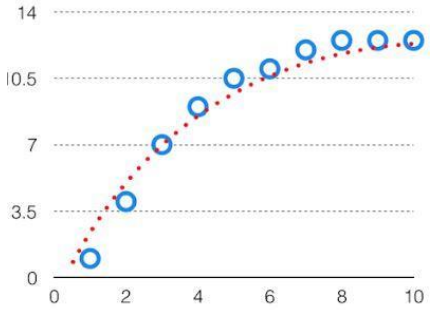
High Variance in the Model

In such a problem, a hypothesis looks like follows.

$$h_\theta(x) = g\left(\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4\right)$$

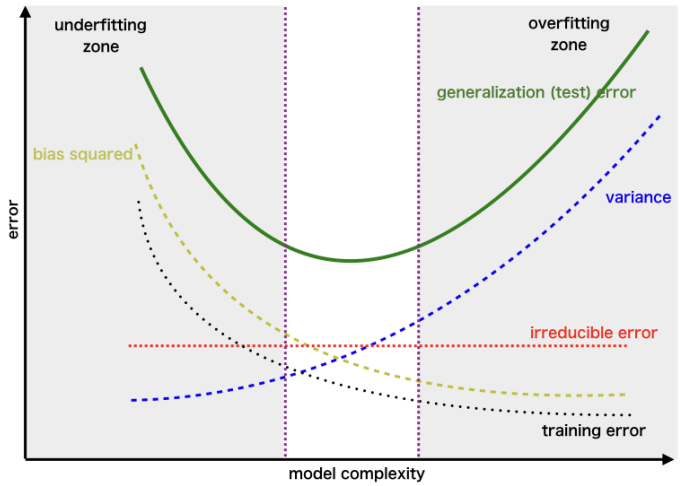
Bias Variance Tradeoff

If the algorithm is too simple (hypothesis with linear equation) then it may be on high bias and low variance condition and thus is error-prone. If algorithms fit too complex (hypothesis with high degree equation) then it may be on high variance and low bias. In the latter condition, the new entries will not perform well. Well, there is something between both of these conditions, known as a Trade-off or Bias Variance Trade-off. This tradeoff in complexity is why there is a tradeoff between bias and variance. An algorithm can't be more complex and less complex at the same time. For the graph, the perfect tradeoff will be like this.



We try to optimize the value of the total error for the model by using the Bias-Variance Tradeoff.

The best fit will be given by the hypothesis on the tradeoff point. The error to complexity graph to show trade-off is given as –



Region for the Least Value of Total Error

This is referred to as the best point chosen for the training of the algorithm which gives low error in training as well as testing data.

No Free Lunch Theorem:

What is No Free Lunch Theorem :
The No Free Lunch Theorem is often used in optimization and machine learning, with little comprehension of what it means or implies.
The theory asserts that when the performance of all optimization methods is averaged across all conceivable problems, they all perform equally well. It indicates that no one optimum optimization algorithm exists. Because of the strong link between optimization, search, and machine learning, there is no one optimum machine learning method for predictive modelling tasks like classification and regression.

They all agree on one point: there is no "best" algorithm for specific kinds of algorithms, since they all perform similarly on average. Mathematically, the computing cost of finding a solution is the same for any solution technique when averaged across all problems in the class. As a result, no solution provides a shortcut.

*There are **two No Free Lunch (NFL) theorems in general:** one for machine learning and one for search and optimization. These two theorems are connected and are frequently combined into a single general postulate (the folklore theorem).*

Although many other scholars have contributed to the collective writings on the No Free Lunch theorems, David Wolpert is the most well-known name connected with these studies. Surprisingly, the concept that may have inspired the NFL theorem was first offered by a 1700s philosopher. Yes, you read that correctly! A philosopher, not a mathematician or a statistician..

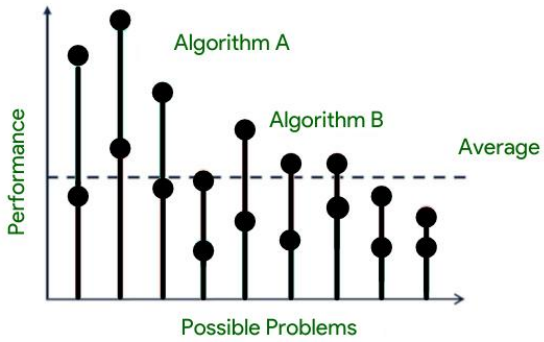


Figure 1. Understanding NFL.

David Hume, a Scottish philosopher, presented the issue of induction in the mid-1700s. This is a philosophical question about whether inductive reasoning leads to true knowledge.

Inductive reasoning is a type of thinking in which we make inferences about the world based on previous observations.

According to the "No Free Lunch" theory, there is no one model that works best for every situation. Because the assumptions of a great model for one issue may not hold true for another, it is typical in machine learning to attempt many models to discover the one that performs best for a specific problem. This is especially true in supervised learning, where validation or cross-validation is frequently used to compare the prediction accuracy of many models of various complexity in order to select the optimal model. A good model may also be trained using several methods — for example, linear regression can be learned using normal equations or gradient descent.

According to the "No Free Lunch" theorem, all optimization methods perform equally well when averaged over all optimization tasks without re-sampling. This fundamental theoretical notion has had the greatest impact on optimization, search, and supervised learning. The first theorem, No Free Lunch, was rapidly formulated, resulting in a series of research works, which defined a whole field of study with meaningful outcomes across different disciplines of science where the effective exploration of a search region is a vital and crucial activity.
In general, its usefulness is as important as the algorithm. An effective solution is created by matching the utility with the algorithm. If no good conditions for the objective function are known, and one is just working with a black box, no guarantee can be made that this or that method outperforms a (pseudo)random search.

A framework is being created to investigate the relationship between successful optimization algorithms and the issues they solve. A series of "no free lunch" (NFL) theorems are provided, establishing that any improved performance over one class of tasks is compensated by improved performance over another. These theorems provide a geometric explanation of what it means for an algorithm to be well matched to an optimization issue.

The NFL theorems are also applied to information-theoretic elements of optimization and benchmark measurements of performance.

There is no such thing as a free lunch, since adding alternatives to a project incurs both direct and opportunity expenses. As a result, incorporating actual alternatives may increase the original development cost. Direct costs are the expenses of additional development effort required to include certain flexibilities into the project's architecture. Opportunity costs are the expenses of not being able to do anything else (for example, add a feature) as a result of the time and effort spent on generating that flexibility.

Conclusion:
Machine learning models adhere to the Garbage in, Garbage out (GIGO) principle (i.e. Predictions rely on the data quality on which our model is trained). And a lot of study went into these theorems, and others may claim that this theorem does not apply in all instances. It is preferable that we concentrate on the aspects that will help us better comprehend the data and construct the best performing models.

*Ugly Duckling Theorem

While the No Free Lunch Theorem shows that in the absence of assumptions we should not prefer any learning or classification algorithm over another, an analogous theorem addresses features and patterns. Roughly speaking, the Ugly Duckling Theorem states that in the absence of assumptions there is no privileged or "best" feature representation, and that even the notion of similarity between patterns depends implicitly on assumptions which may or may not be correct.

Theorem 9.2 (Ugly Duckling) *Given that we use a finite set of predicates that enables us to distinguish any two patterns under consideration, the number of predicates shared by any two such patterns is constant and independent of the choice of those patterns. Furthermore, if pattern similarity is based on the total number of predicates shared by two patterns, then any two patterns are "equally similar." **

Error Decomposition

To answer this question we decompose the error of an $\mathrm{ERM}_{\mathcal{H}}$ predictor into two components as follows. Let h_S be an $\mathrm{ERM}_{\mathcal{H}}$ hypothesis. Then, we can write

$$L_{\mathcal{D}}(h_S) = \epsilon_{\mathrm{app}} + \epsilon_{\mathrm{est}} \quad \text{where :} \quad \epsilon_{\mathrm{app}} = \min_{h \in \mathcal{H}} L_{\mathcal{D}}(h), \quad \epsilon_{\mathrm{est}} = L_{\mathcal{D}}(h_S) - \epsilon_{\mathrm{app}}. \quad (5.7)$$

The Approximation Error – the minimum risk achievable by a predictor in the hypothesis class. This term measures how much risk we have because we restrict ourselves to a specific class, namely, how much *inductive bias* we have. The approximation error does not depend on the sample size and is determined by the hypothesis class chosen. Enlarging the hypothesis class can decrease the approximation error.

Under the realizability assumption, the approximation error is zero. In the agnostic case, however, the approximation error can be large.[1]

Error Decomposition:

Error Decomposition

To answer this question we decompose the error of an $\mathrm{ERM}_{\mathcal{H}}$ predictor into two components as follows. Let h_S be an $\mathrm{ERM}_{\mathcal{H}}$ hypothesis. Then, we can write

$$L_{\mathcal{D}}(h_S) = \epsilon_{\mathrm{app}} + \epsilon_{\mathrm{est}} \quad \text{where :} \quad \epsilon_{\mathrm{app}} = \min_{h \in \mathcal{H}} L_{\mathcal{D}}(h), \quad \epsilon_{\mathrm{est}} = L_{\mathcal{D}}(h_S) - \epsilon_{\mathrm{app}}. \quad (5.7)$$

The Approximation Error – the minimum risk achievable by a predictor in the hypothesis class. This term measures how much risk we have because we restrict ourselves to a specific class, namely, how much *inductive bias* we have. The approximation error does not depend on the sample size and is determined by the hypothesis class chosen. Enlarging the hypothesis class can decrease the approximation error.

Under the realizability assumption, the approximation error is zero. In the agnostic case, however, the approximation error can be large.[1]

Effect of hypothesis class size

As the hypothesis class size increases...

Approximation error decreases because:

taking min over larger set

Estimation error increases because:

harder to estimate something more complex

Estimation error analogy

Scenario 1: ask few people around

Is your name Joe?

Scenario 2: email all of Stanford

Is your name Joe?

people = hypotheses, questions = examples

- **The Estimation Error** – the difference between the approximation error and the error achieved by the ERM predictor. The estimation error results because the empirical risk (i.e., training error) is only an estimate of the true risk, and so the predictor minimizing the empirical risk is only an estimate of the predictor minimizing the true risk.

  The quality of this estimation depends on the training set size and on the size, or complexity, of the hypothesis class. As we have shown, for a finite hypothesis class, ϵ_{est} increases (logarithmically) with $|\mathcal{H}|$ and decreases with m. We can think of the size of $\mathcal{H}$ as a measure of its complexity. In future chapters we will define other complexity measures of hypothesis classes.

Error decomposition in machine learning refers to the process of breaking down the overall error or loss of a model into its constituent parts to better understand the sources of inaccuracies and identify areas for improvement. This is particularly useful for diagnosing model performance and making informed decisions on how to enhance it. The decomposition typically involves analyzing different components of the error, such as bias, variance, and irreducible error.

Here are the key components of error decomposition:

1. Bias Error (Underfitting): Bias error, also known as underfitting, occurs when a model is too simple to capture the underlying patterns in the data. This leads to a systematic error where the model consistently fails to predict the true values.

2. Variance Error (Overfitting): Variance error, or overfitting, occurs when a model is too complex and captures noise or random fluctuations in the training data. This results in a high sensitivity to variations in the training data but may perform poorly on new, unseen data.

3. Irreducible Error:Irreducible error represents the inherent uncertainty in the data that cannot be reduced regardless of how well the model is trained. It is the minimum achievable error and is a result of factors such as noise in the data, measurement errors, or inherent unpredictability.

The total error of a machine learning model can be expressed as the sum of these three components:

$$\text{Total Error} = \text{Bias}^2 + \text{Variance} + \text{Irreducible Error}$$

Understanding error decomposition is crucial for model evaluation and improvement:

- High Bias: Indicates that the model is too simple and unable to capture the underlying patterns in the data. Strategies to reduce bias include using more complex models or adding additional features.

- High Variance:Suggests that the model is too sensitive to the training data and may be overfitting. Solutions involve simplifying the model, using regularization techniques, or obtaining more diverse training data.

- Balancing Bias and Variance: Striking a balance between bias and variance is essential to achieve a model that generalizes well to unseen data.

Techniques like cross-validation, learning curves, and regularization can help in analyzing and mitigating bias and variance issues during the model development process. Error decomposition provides valuable insights into the performance of machine learning models and guides the selection of appropriate strategies for model improvement.

The VC-Dimension:

PAC and Agnostic Learning

- Finite H, assume target function c ∈ H

$$Pr(\exists h \in H,\ s.t.\ (error_{train}(h) = 0) \wedge (error_{true}(h) > \epsilon)\ ) \leq |H|e^{-\epsilon m}$$

  - Suppose we want this to be at most δ. Then *m* examples suffice:

$$m \geq \frac{1}{\epsilon}(\ln|H| + \ln(1/\delta))$$

- Finite H, agnostic learning: perhaps c *not* in H

$$P(\exists h \in H,\ |\epsilon(h) - \hat{\epsilon}(h)| > \gamma) \leqslant 2k\exp(-2\gamma^2 m)$$

  - ➔ $$m \geq \frac{1}{2\gamma^2}\log\frac{2k}{\delta}$$
  - with probability at least (1-δ) every h in H satisfies

$$\epsilon(\hat{h}) \leq \left(\min_{h\in H}\epsilon(h)\right) + 2\sqrt{\frac{1}{m}\log\frac{2k}{\delta}}$$

What if H is not finite?

- Can't use our result for infinite H

- Need some other measure of complexity for H – Vapnik-Chervonenkis (VC) dimension!

Shattering a Set of Instances

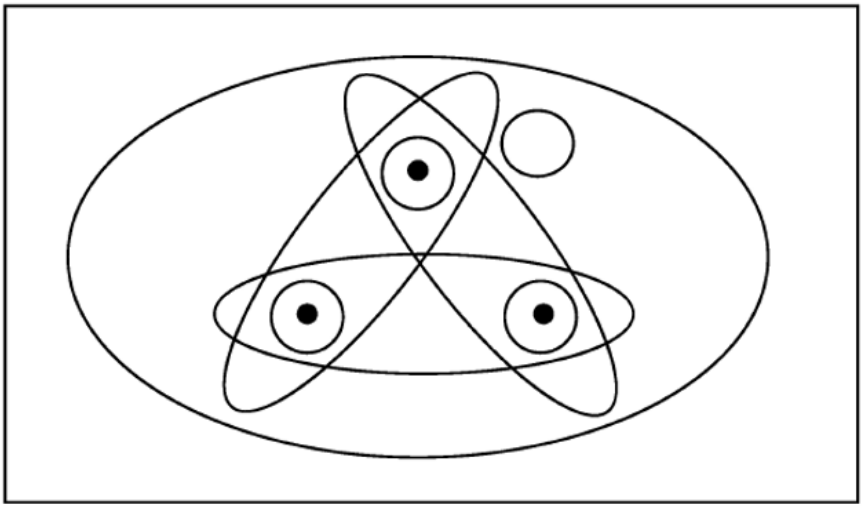
- *Definition*: Given a set $\mathcal{S} = \{x^{(1)}, \ldots, x^{(m)}\}$ (no relation to the training set) of points $x^{(i)} \in X$, we say that $\mathcal{H}$ shatters $\mathcal{S}$ if $\mathcal{H}$ can realize any labeling on $\mathcal{S}$.

  I.e., if for any set of labels $\{y^{(1)}, \ldots, y^{(d)}\}$, there exists some $h \in \mathcal{H}$ so that $h(x^{(i)}) = y^{(i)}$ for all $i = 1, \ldots, m$.

- There are 2^m different ways to separate the sample into two sub-samples (a dichotomy)

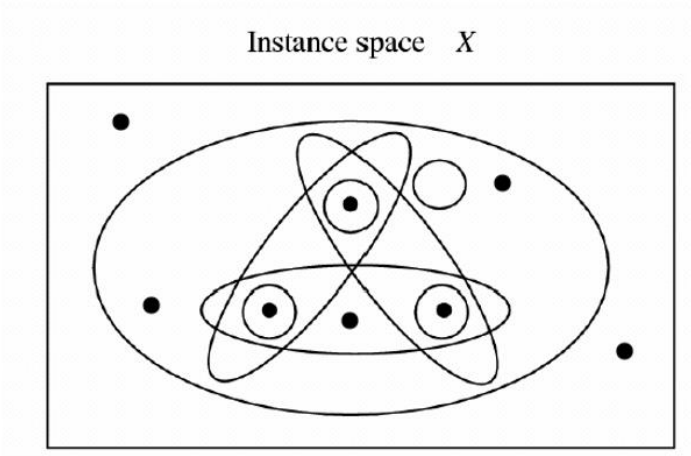
Three Instances Shattered

Instance space X



The Vapnik-Chervonenkis Dimension

- *Definition*: The **Vapnik-Chervonenkis dimension**, *VC*(*H*), of hypothesis space *H* defined over instance space *X* is the size of the *largest finite subset* of *X* shattered by *H* . If arbitrarily large finite sets of *X* can be shattered by *H* , then $VC(H) \equiv \infty$.



VC dimension: examples

Consider X = $\mathbb{R}$, want to learn c: X→{0,1}

What is VC dimension of

- Open intervals:

  H1: if x>a, then y=1 else y=0

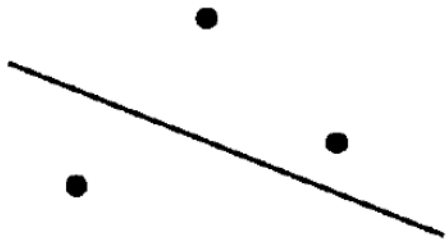
- Closed intervals:

  H2: if a<x<b, then y=1 else y=0

VC dimension: examples

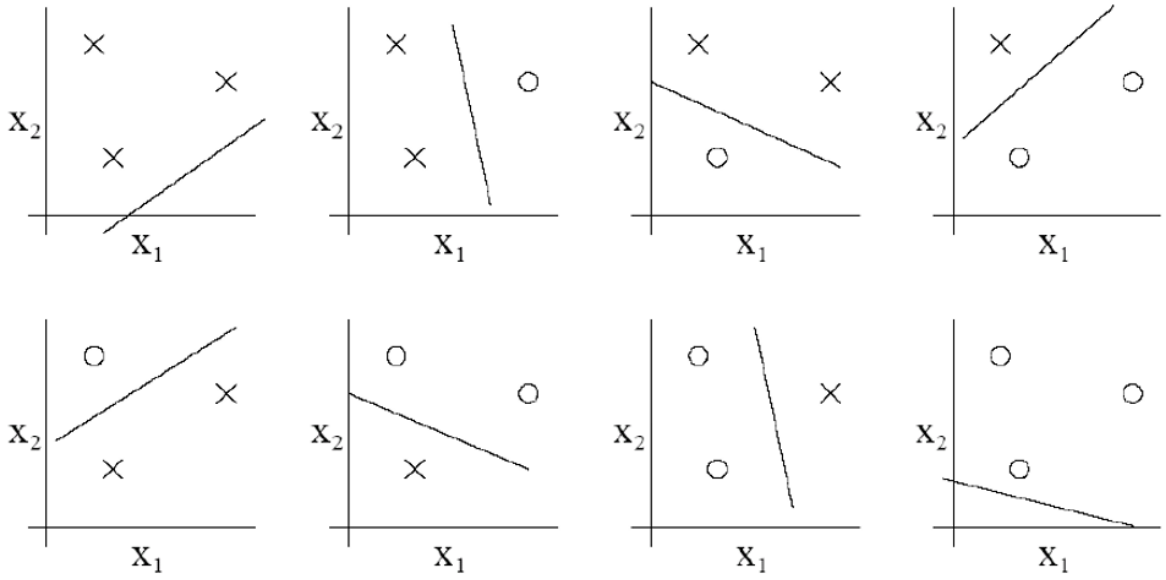
Consider X = $\mathbb{R}^2$, want to learn c: X→{0,1}

- What is VC dimension of lines in a plane?
  H= { ((wx+b)>0 → y=1) }

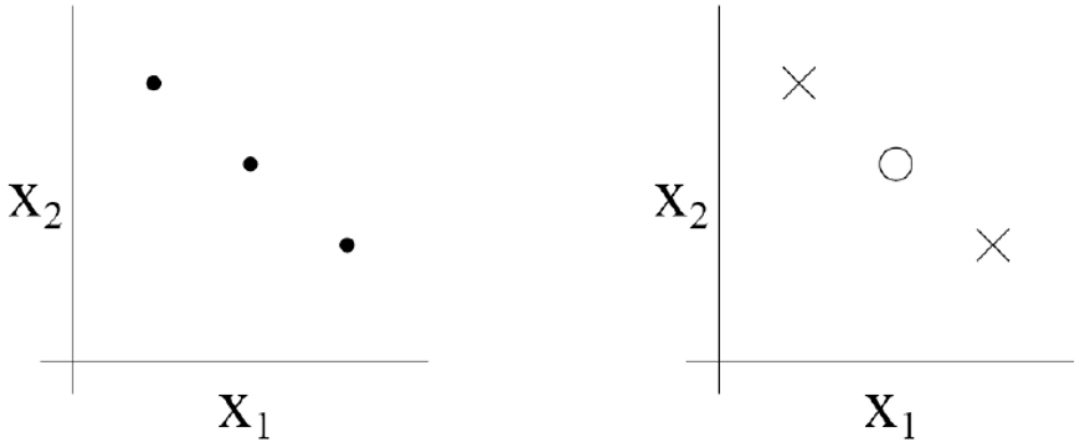


(a)

(b)



- For any of the eight possible labelings of these points, we can find a linear classier that obtains "zero training error" on them.
- Moreover, it is possible to show that there is no set of 4 points that this hypothesis class can shatter.



- The VC dimension of H here is 3 even though there may be sets of size 3 that it cannot shatter.
- under the definition of the VC dimension, in order to prove that VC(H) is at least *d*, we need to show only that there's at least one set of size d that H can shatter.

- **Theorem** Consider some set of *m* points in $\mathbb{R}^n$. Choose any one of the points as origin. Then the *m* points can be shattered by oriented hyperplanes if and only if the position vectors of the remaining points are linearly independent.

- **Corollary**: The VC dimension of the set of oriented hyperplanes in $\mathbb{R}^n$ is $n+1$.

  Proof: we can always choose $n + 1$ points, and then choose one of the points as origin, such that the position vectors of the remaining n points are linearly independent, but can never choose $n + 2$ such points (since no $n + 1$ vectors in $\mathbb{R}^n$ can be linearly independent).

Sample Complexity from VC Dimension

- How many randomly drawn examples suffice to ε-exhaust $VS_{H,S}$ with probability at least (1 - δ)?

ie., to guarantee that any hypothesis that perfectly fits the training data is probably (1-δ) approximately (ε) correct on testing data from the same distribution

$$m \geq \frac{1}{\varepsilon}(4\log_2(2/\delta) + 8VC(H)\log_2(13/\varepsilon))$$

Compare to our earlier results based on $|H|$:

$$m \geq \frac{1}{2\varepsilon^2}(\ln|H| + \ln(1/\delta))$$

The Vapnik-Chervonenkis dimension

- The Vapnik-Chervonenkis dimension of H is d if there exists such an S, |S|=d which it can shatter, but it cannot shatter any S for |S|=d+1 (If it can shatter any finite S then VCD=∞.)
- Theorem: Let **C** be a concept class,a and H a representation set for which VCD(H)=d. Let L be a learning algorithm that learns c∈**C** by getting a set S of training samples with |S|=m, and it outputs a hypothesis h∈H which is consistent with S. The learning of **C** over H is PAC learning if

$$m \geq c_0 \frac{1}{\varepsilon}\left(d \log \frac{1}{\varepsilon} + \log \frac{1}{\delta}\right)$$

(where c_0 is a proper constant)

- Remark: In contrary to the finite case, the bound obtained here is tight (that is, m samples are not only sufficient, but in certain cases they are necessary as well).

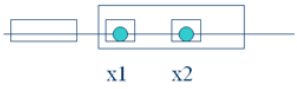
The Vapnik-Chervonenkis dimension

- Let's compare the bounds obtained for the finite and infinite cases:
- Finite case: $m \geq \frac{1}{\varepsilon}(\ln|H| + \ln(\frac{1}{\delta}))$
- Infinite case: $m \geq c_0 \frac{1}{\varepsilon}(d \log \frac{1}{\varepsilon} + \log \frac{1}{\delta})$

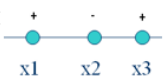
- The two formulas look quite similar, but the role of |H| is taken by the Vapnik-Chervonenkis dimension in the infinite case
  - Both formulas increase relatively slowly as a function of ε and δ, so in this sense these are not bad boundaries…

Examples of VC-dimension

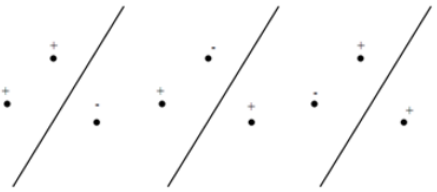
- Finite intervals over the line: VCD=2
  - VCD≥2, as **these** two points can be shattered: (=separated for all label configurations)



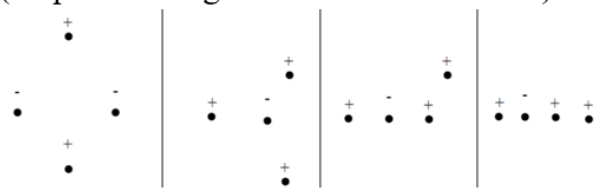
  - VCD<3, as **no** 3 points can be shattered:



- Separating the two classes by lines on the plane: VCD=3 (in d-dimensional space: VCD=d+1)
  - VCD ≥3, as these 3 points can be shattered: (all labeling configurations should be tried!)

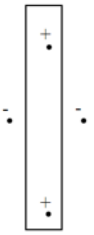


  - VCD<4, as no 4 points can be shattered: (all point arrangements should be tried!)

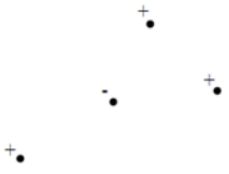


Examples of VC-dimension

- Axis-aligned rectangles on the plane: VCD=4
  - VCD≥4, as **these** 4 points can be shattered:



  - VCD<5, as **no** 5 points can be shattered:



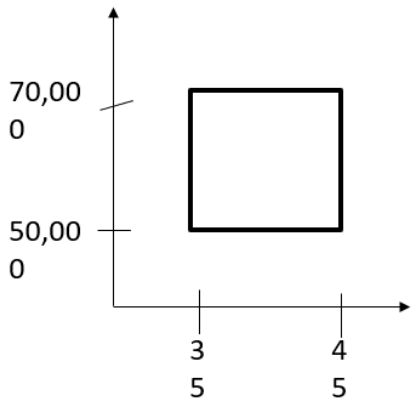
- Convex poligons on the plane: VCD=2d+1 (d is the number of vertices) (the book proofs only one of the directions)
- Conjunctions of literals over $\{0,1\}^n$: VCD=n (See Mitchell's book, only one direction is proved)

Examples

- Intervals of the real axis:
  - $Vcdim = 2, H[n] = O(n^2)$
- Rectangle with axis-parallel edges:
  - $Vcdim = 4, H[n] = O(n^4)$
- Union of 2 intervals of the real axis (Divide an orders set of numbers into two different intervals)
  - $Vcdim = 4, H[n] = O(n^4)$
- Convex polygons:
  - $Vcdim \to \infty, H[n] = 2^n$

Example

- Consider a database consisting of the salary and age for a random sample of the adult population in the United States.
- We are interested in using the database to answer the question:
- What fraction of the adult population in the US has:
- - age between 35 and 45
- - salary between 50,000$ and 70,000$?



Axis Aligned Rectangles

Let $\mathcal{H}$ be the class of axis aligned rectangles, formally

$$\mathcal{H} = \{h_{(a_1,a_2,b_1,b_2)} : a_1 \leq a_2 \text{ and } b_1$$

where

$$h_{(a_1,a_2,b_1,b_2)}(x_1, x_2) = \begin{cases} 1 & \text{if } a_1 \leq x_1 \leq a_2 \text{ a} \\ 0 & \text{otherwise} \end{cases}$$

OR

The Vapnik-Chervonenkis (VC) dimension is a measure of the capacity of a hypothesis set to fit different data sets. It was introduced by Vladimir Vapnik and Alexey Chervonenkis in the 1970s and has become a fundamental concept in statistical learning theory. The VC dimension is a measure of the complexity of a model, which can help us understand how well it can fit different data sets.

The VC dimension of a hypothesis set H is the largest number of points that can be shattered by H. A hypothesis set H shatters a set of points S if, for every possible labeling of the points in S, there exists a hypothesis in H that correctly classifies the points. In other words, a hypothesis set shatters a set of points if it can fit any possible labeling of those points.

Bounds of VC – Dimension

The VC dimension provides both upper and lower bounds on the number of training examples required to achieve a given level of accuracy. The upper bound on the number of training examples is logarithmic in the VC dimension, while the lower bound is linear.

Applications of VC – Dimension

The VC dimension has a wide range of applications in machine learning and statistics. For example, it is used to analyze the complexity of neural networks, support vector machines, and decision trees. The VC dimension can also be used to design new learning algorithms that are robust to noise and can generalize well to unseen data.

The VC dimension can be extended to more complex learning scenarios, such as multiclass classification and regression. The concept of the VC dimension can also be applied to other areas of computer science, such as computational geometry and graph theory.
Code Implementation for VC – Dimension
The VC dimension is a theoretical concept that cannot be directly computed from data. However, we can estimate the VC dimension for a given hypothesis set by counting the number of points that can be shattered by the set. In Python, we can implement a function that computes the VC dimension of a given hypothesis set using this approach.

The function takes a hypothesis set as its input and computes the VC dimension using the brute-force approach of checking all possible combinations of points and labels. It uses the itertools module to generate all possible combinations of points and labels and then checks if the hypothesis set can shatter each combination. The function returns the estimated VC dimension of the hypothesis set.

Let's illustrate the usage of this function with some examples:

Example 1:
Suppose we have a hypothesis set that consists of all linear functions of form f(x) = ax + b, where a and b are real numbers. We can define this hypothesis set in Python as follows:

```
import itertools



def vc_dimension(hypothesis_set):
    """
    Estimates the VC dimension of a hypothesis set using the brute-force approach.
    """
    n = 4
    while True:
        points = [(i, j) for i in range(n) for j in range(2)]
        shattered_sets = 0
        for combination in itertools.combinations(points, n):
            is_shattered = True
            for labeling in itertools.product([0, 1], repeat=n):
                hypotheses = [hypothesis_set(point) for point in
combination]
                if set(hypotheses) != set(labeling):
                    is_shattered = False
```

```
                    break
                if is_shattered:
                    shattered_sets += 1
                else:
                    break
            if not is_shattered:
                break
            n += 1
        return n-1 if shattered_sets == 2**n else n-2


# Example 1: linear function hypothesis set
def linear_function(point):
    x, y = point
    return int(y >= x)


print(vc_dimension(linear_function))
```

Output:
2

In example 1, the linear_function function implements a simple linear function hypothesis set that returns 1 if the y-coordinate of the input point is greater than or equal to the x-coordinate, and 0 otherwise. The vc_dimension function is then used to estimate the VC dimension of this hypothesis set, which is 2.

Example 2:
Suppose we have a hypothesis set that consists of all quadratic function of form $f(x) = ax^2 + bx + c$, where a, b, and c are real numbers. We can define this hypothesis set in Python as follows:

```
import itertools
```

```
def vc_dimension(hypothesis_set):
    """
    Estimates the VC dimension of a hypothesis set using the brute-force approach.
    """
    n = 5
    while True:
        points = [(i, j) for i in range(n) for j in range(2)]
        shattered_sets = 0
        for combination in itertools.combinations(points, n):
            is_shattered = True
            for labeling in itertools.product([0, 1], repeat=n):
                hypotheses = [hypothesis_set(point) for point in
combination]
                if set(hypotheses) != set(labeling):
                    is_shattered = False
                    break
            if is_shattered:
                shattered_sets += 1
            else:
                break
        if not is_shattered:
            break
        n += 1
    return n-1 if shattered_sets == 2**n else n-2


# Example 2: quadratic function hypothesis set
def quadratic_function(point):
    x, y = point
    return int(y >= x**2)
```

print(vc_dimension(quadratic_function))

Output:
3

In example 2, the quadratic_function function implements a more complex quadratic function hypothesis set that returns 1 if the y-coordinate of the input point is greater than or equal to the square of the x-coordinate, and 0 otherwise. The vc_dimension function is then used to estimate the VC dimension of this hypothesis set, which is 3.

Conclusion
The VC dimension is a fundamental concept in statistical learning theory that measures the complexity of a hypothesis set. It provides both upper and lower bounds on the number of training examples required to achieve a given level of accuracy. In Python, we can estimate the VC dimension of a given hypothesis set using a brute-force approach that checks all possible combinations of points and labels. The VC dimension has a wide range of applications in machine learning and statistics and can be extended to more complex learning scenarios.

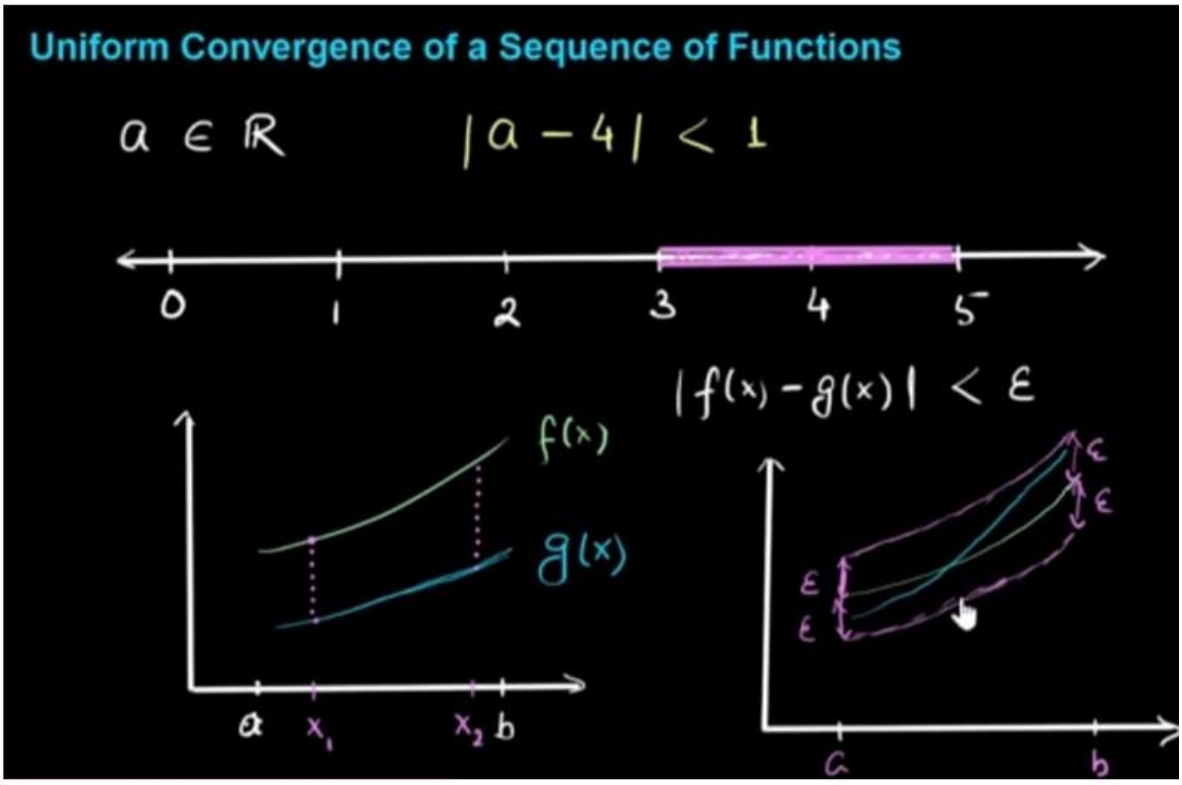
The Rademacher Complexity:

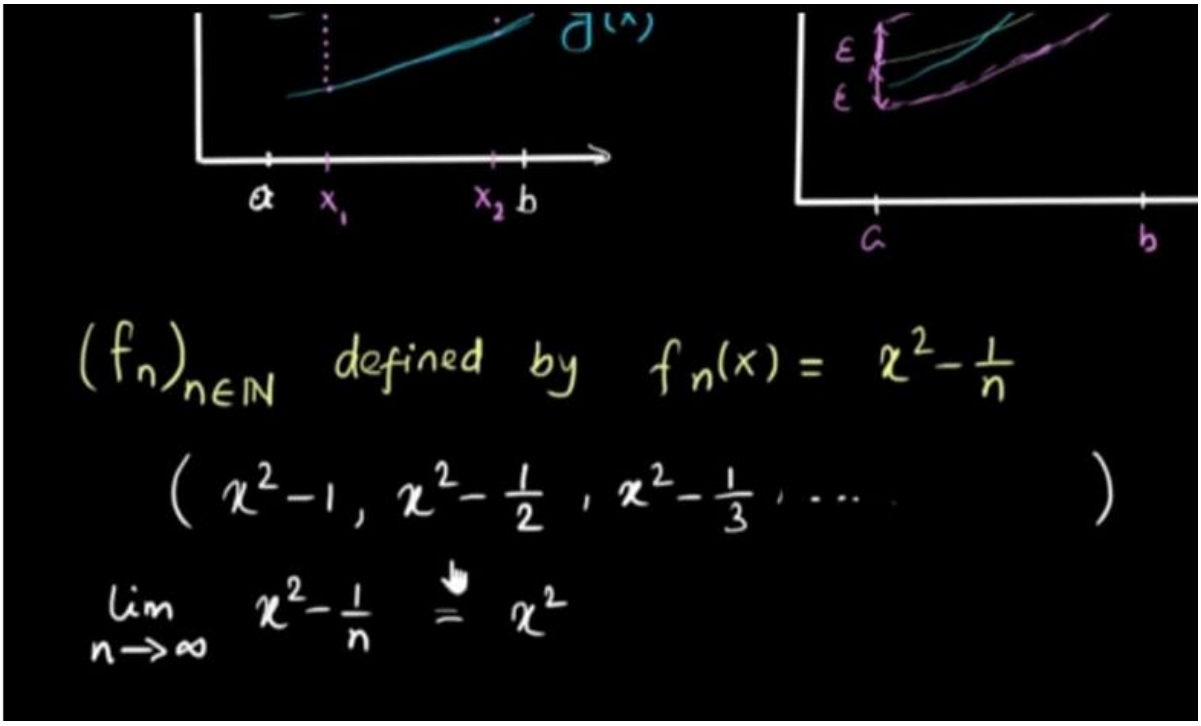
Uniform Convergence Is Sufficient for Learnability

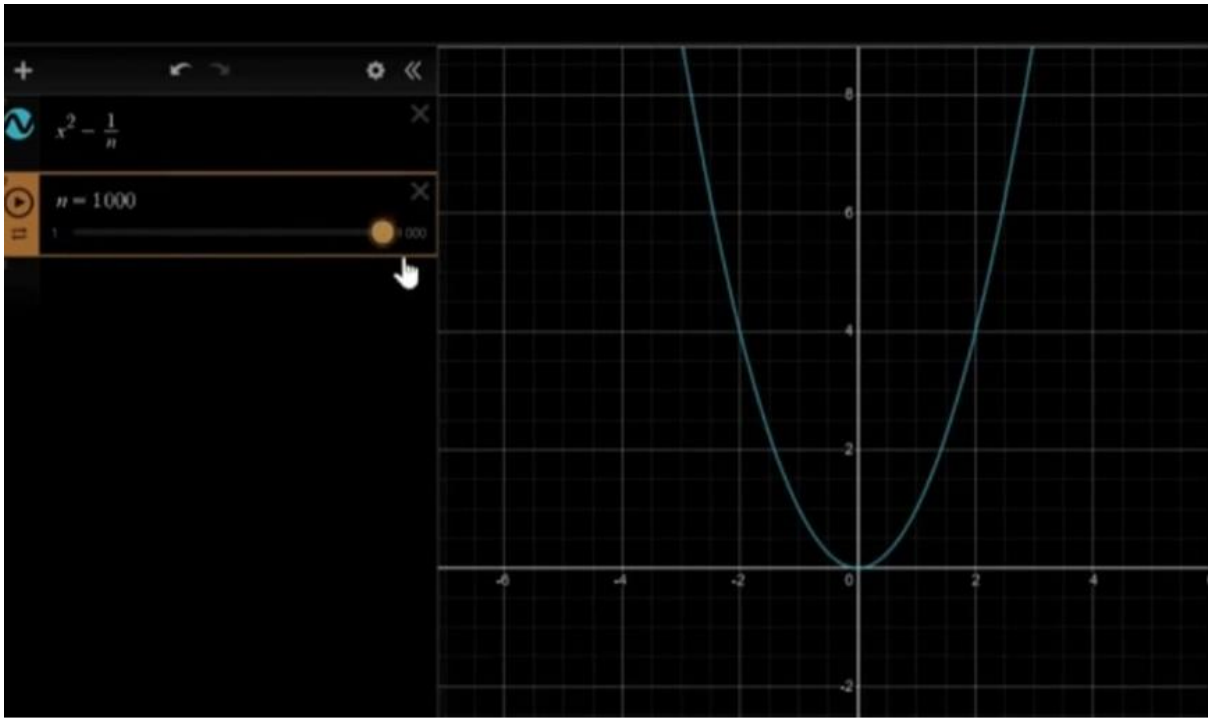
The idea behind the learning condition discussed in this chapter is very simple. Recall that, given a hypothesis class, $\mathcal{H}$, the ERM learning paradigm works as follows: Upon receiving a training sample, S, the learner evaluates the risk (or error) of each h in $\mathcal{H}$ on the given sample and outputs a member of $\mathcal{H}$ that minimizes this empirical risk. The hope is that an h that minimizes the empirical risk with respect to S is a risk minimizer (or has risk close to the minimum) with respect to the true data probability distribution as well. For that, it suffices to ensure that the empirical risks of all members of $\mathcal{H}$ are good approximations of their true risk. Put another way, we need that uniformly over all hypotheses in the hypothesis class, the empirical risk will be close to the true risk, as formalized in the following.

DEFINITION 4.1 (ϵ-representative sample) A training set S is called ϵ-representative (w.r.t. domain Z, hypothesis class $\mathcal{H}$, loss function ℓ, and distribution $\mathcal{D}$) if

$$\forall h \in \mathcal{H}, \quad |L_S(h) - L_{\mathcal{D}}(h)| \leq \epsilon.$$


Uniform Convergence of a Sequence of Functions
a ∈ ℝ
|a − 4| < 1
0 1 2 3 4 5
|f(x) − g(x)| < ε
f(x)
g(x)
a x1 x2 b
ε
ε
ε
ε
a
b


a x1 x2 b
ε
ε
a
b
(fn)n∈ℕ defined by fn(x) = x² − 1/n
(x² − 1 , x² − 1/2 , x² − 1/3 , ...)
lim n→∞ x² − 1/n = x²



The next simple lemma states that whenever the sample is $(\epsilon/2)$-representative, the ERM learning rule is guaranteed to return a good hypothesis.

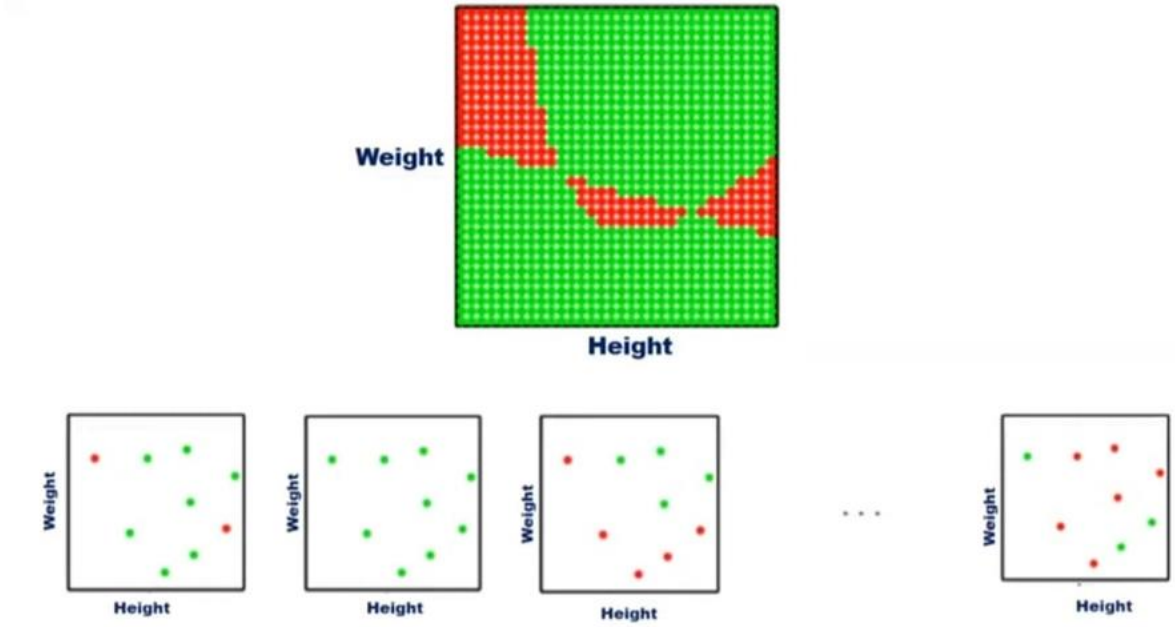
LEMMA 4.2 *Assume that a training set S is $\frac{\epsilon}{2}$-representative (w.r.t. domain Z, hypothesis class $\mathcal{H}$, loss function ℓ, and distribution $\mathcal{D}$). Then, any output of* $\mathrm{ERM}_{\mathcal{H}}(S)$, *namely, any* $h_S \in \operatorname{argmin}_{h \in \mathcal{H}} L_S(h)$, *satisfies*

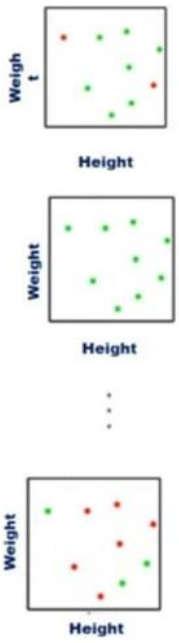
$$L_{\mathcal{D}}(h_S) \leq \min_{h \in \mathcal{H}} L_{\mathcal{D}}(h) + \epsilon.$$

DEFINITION 4.3 (Uniform Convergence) We say that a hypothesis class $\mathcal{H}$ has the *uniform convergence property* (w.r.t. a domain Z and a loss function ℓ) if there exists a function $m_{\mathcal{H}}^{\text{UC}} : (0,1)^2 \to \mathbb{N}$ such that for every $\epsilon, \delta \in (0,1)$ and for every probability distribution $\mathcal{D}$ over Z, if S is a sample of $m \geq m_{\mathcal{H}}^{\text{UC}}(\epsilon, \delta)$ examples drawn i.i.d. according to $\mathcal{D}$, then, with probability of at least $1 - \delta$, S is ϵ-representative.

Similar to the definition of sample complexity for PAC learning, the function $m_{\mathcal{H}}^{\text{UC}}$ measures the (minimal) sample complexity of obtaining the uniform convergence property, namely, how many examples we need to ensure that with probability of at least $1 - \delta$ the sample would be ϵ-representative.

The term *uniform* here refers to having a fixed sample size that works for all members of $\mathcal{H}$ and over all possible probability distributions over the domain.





Error(H1)	Error(h2)
0.04	0.04
0.03	0.035
0.09	0.039
0.06	0.06
0.025	0.025
0.049	0.059
0.04	0.04
0.03	0.03
0.05	0.55
0.043	0.043

ε= 0.05 δ =0.20

P (H1) = 8/10=0.80

P (H1) = 8/10=0.80 >=1- 0.20

Hence H1 is probably approximately correct

P(H2)= 7/10=0.70

P(H2)= 7/10=0.70 <1-0.20

Hence H2 is not probably approximately correct

COROLLARY 4.4 *If a class $\mathcal{H}$ has the uniform convergence property with a function $m_{\mathcal{H}}^{UC}$ then the class is agnostically PAC learnable with the sample complexity $m_{\mathcal{H}}(\epsilon, \delta) \leq m_{\mathcal{H}}^{UC}(\epsilon/2, \delta)$. Furthermore, in that case, the $\mathrm{ERM}_{\mathcal{H}}$ paradigm is a successful agnostic PAC learner for $\mathcal{H}$.*

Nonuniform Learnability

"Nonuniform learnability" allows the sample size to be nonuniform with respect to the different hypotheses with which the learner is competing. We say that a hypothesis h is (ϵ, δ)-competitive with another hypothesis h' if, with probability higher than $(1 - \delta)$,

$$L_{\mathcal{D}}(h) \leq L_{\mathcal{D}}(h') + \epsilon.$$

In PAC learnability, this notion of "competitiveness" is not very useful, as we are looking for a hypothesis with an absolute low risk (in the realizable case) or with a low risk compared to the minimal risk achieved by hypotheses in our class (in the agnostic case). Therefore, the sample size depends only on the accuracy and confidence parameters. In nonuniform learnability, however, we allow the sample size to be of the form $m_{\mathcal{H}}(\epsilon, \delta, h)$; namely, it depends also on the h with which we are competing.

DEFINITION 7.1 A hypothesis class $\mathcal{H}$ is *nonuniformly learnable* if there exist a learning algorithm, A, and a function $m_{\mathcal{H}}^{\text{NUL}} : (0,1)^2 \times \mathcal{H} \to \mathbb{N}$ such that, for every $\epsilon, \delta \in (0,1)$ and for every $h \in \mathcal{H}$, if $m \geq m_{\mathcal{H}}^{\text{NUL}}(\epsilon, \delta, h)$ then for every distribution $\mathcal{D}$, with probability of at least $1 - \delta$ over the choice of $S \sim \mathcal{D}^m$, it holds that

$$L_{\mathcal{D}}(A(S)) \leq L_{\mathcal{D}}(h) + \epsilon.$$

At this point it might be useful to recall the definition of agnostic PAC learnability (Definition 3.3):

A hypothesis class $\mathcal{H}$ is agnostically PAC learnable if there exist a learning algorithm, A, and a function $m_{\mathcal{H}} : (0,1)^2 \to \mathbb{N}$ such that, for every $\epsilon, \delta \in (0,1)$ and for every distribution $\mathcal{D}$, if $m \geq m_{\mathcal{H}}(\epsilon, \delta)$, then with probability of at least $1 - \delta$ over the choice of $S \sim \mathcal{D}^m$ it holds that

$$L_{\mathcal{D}}(A(S)) \leq \min_{h' \in \mathcal{H}} L_{\mathcal{D}}(h') + \epsilon.$$

Note that this implies that for every $h \in \mathcal{H}$

$$L_{\mathcal{D}}(A(S)) \leq L_{\mathcal{D}}(h) + \epsilon.$$

In both types of learnability, we require that the output hypothesis will be (ϵ, δ)-competitive with every other hypothesis in the class. But the difference between these two notions of learnability is the question of whether the sample size m may depend on the hypothesis h to which the error of $A(S)$ is compared. Note that that nonuniform learnability is a relaxation of agnostic PAC learnability. That is, if a class is agnostic PAC learnable then it is also nonuniformly learnable.

Characterizing Nonuniform Learnability

Our goal now is to characterize nonuniform learnability. In the previous chapter we have found a crisp characterization of PAC learnable classes, by showing that a class of binary classifiers is agnostic PAC learnable if and only if its VC-dimension is finite. In the following theorem we find a different characterization for nonuniform learnable classes for the task of binary classification.

THEOREM 7.2 *A hypothesis class $\mathcal{H}$ of binary classifiers is nonuniformly learnable if and only if it is a countable union of agnostic PAC learnable hypothesis classes.*

THEOREM 7.3 *Let $\mathcal{H}$ be a hypothesis class that can be written as a countable union of hypothesis classes, $\mathcal{H} = \bigcup_{n\in\mathbb{N}} \mathcal{H}_n$, where each $\mathcal{H}_n$ enjoys the uniform convergence property. Then, $\mathcal{H}$ is nonuniformly learnable.*

Structural Risk Minimization

So far, we have encoded our prior knowledge by specifying a hypothesis class $\mathcal{H}$, which we believe includes a good predictor for the learning task at hand. Yet another way to express our prior knowledge is by specifying preferences over hypotheses within $\mathcal{H}$. In the Structural Risk Minimization (SRM) paradigm, we do so by first assuming that $\mathcal{H}$ can be written as $\mathcal{H} = \bigcup_{n\in\mathbb{N}} \mathcal{H}_n$ and then specifying a weight function, $w : \mathbb{N} \to [0, 1]$, which assigns a weight to each hypothesis class, $\mathcal{H}_n$, such that a higher weight reflects a stronger preference for the hypothesis class. In this section we discuss how to learn with such prior knowledge. In the next section we describe a couple of important weighting schemes, including Minimum Description Length.

Concretely, let $\mathcal{H}$ be a hypothesis class that can be written as $\mathcal{H} = \bigcup_{n \in \mathbb{N}} \mathcal{H}_n$. For example, $\mathcal{H}$ may be the class of all polynomial classifiers where each $\mathcal{H}_n$ is the class of polynomial classifiers of degree n (see Example 7.1). Assume that for each n, the class $\mathcal{H}_n$ enjoys the uniform convergence property (see Definition 4.3 in Chapter 4) with a sample complexity function $m^{\text{UC}}_{\mathcal{H}_n}(\epsilon, \delta)$. Let us also define the function $\epsilon_n : \mathbb{N} \times (0,1) \to (0,1)$ by

$$\epsilon_n(m, \delta) = \min\{\epsilon \in (0,1) : m^{\text{UC}}_{\mathcal{H}_n}(\epsilon, \delta) \leq m\}. \tag{7.1}$$

In words, we have a fixed sample size m, and we are interested in the lowest possible upper bound on the gap between empirical and true risks achievable by using a sample of m examples.

From the definitions of uniform convergence and ϵ_n, it follows that for every m and δ, with probability of at least $1 - \delta$ over the choice of $S \sim \mathcal{D}^m$ we have that

$$\forall h \in \mathcal{H}_n, \quad |L_{\mathcal{D}}(h) - L_S(h)| \leq \epsilon_n(m, \delta). \tag{7.2}$$

Let $w : \mathbb{N} \to [0,1]$ be a function such that $\sum_{n=1}^{\infty} w(n) \leq 1$. We refer to w as a *weight function* over the hypothesis classes $\mathcal{H}_1, \mathcal{H}_2, \ldots$. Such a weight function can reflect the importance that the learner attributes to each hypothesis class, or some measure of the complexity of different hypothesis classes. If $\mathcal{H}$ is a finite union of N hypothesis classes, one can simply assign the same weight of $1/N$ to all hypothesis classes. This equal weighting corresponds to no a priori preference to any hypothesis class. Of course, if one believes (as prior knowledge) that a certain hypothesis class is more likely to contain the correct target function, then it should be assigned a larger weight, reflecting this prior knowledge. When $\mathcal{H}$ is a (countable) infinite union of hypothesis classes, a uniform weighting is not possible but many other weighting schemes may work. For example, one can choose $w(n) = \frac{6}{\pi^2 n^2}$ or $w(n) = 2^{-n}$. Later in this chapter we will provide another convenient way to define weighting functions using description languages.

The SRM rule follows a "bound minimization" approach. This means that the goal of the paradigm is to find a hypothesis that minimizes a certain upper bound on the true risk. The bound that the SRM rule wishes to minimize is given in the following theorem.

THEOREM 7.4 *Let $w : \mathbb{N} \to [0,1]$ be a function such that $\sum_{n=1}^{\infty} w(n) \leq 1$. Let $\mathcal{H}$ be a hypothesis class that can be written as $\mathcal{H} = \bigcup_{n \in \mathbb{N}} \mathcal{H}_n$, where for each n, $\mathcal{H}_n$ satisfies the uniform convergence property with a sample complexity function $m^{UC}_{\mathcal{H}_n}$. Let ϵ_n be as defined in Equation (7.1). Then, for every $\delta \in (0,1)$ and distribution $\mathcal{D}$, with probability of at least $1-\delta$ over the choice of $S \sim \mathcal{D}^m$, the following bound holds (simultaneously) for every $n \in \mathbb{N}$ and $h \in \mathcal{H}_n$.*

$$|L_{\mathcal{D}}(h) - L_S(h)| \leq \epsilon_n(m, w(n) \cdot \delta).$$

Therefore, for every $\delta \in (0,1)$ and distribution $\mathcal{D}$, with probability of at least $1-\delta$ it holds that

$$\forall h \in \mathcal{H}, \quad L_{\mathcal{D}}(h) \leq L_S(h) + \min_{n: h \in \mathcal{H}_n} \epsilon_n(m, w(n) \cdot \delta). \tag{7.3}$$

Applying the union bound over $n = 1, 2, \ldots$, we obtain that with probability of at least $1 - \sum_n \delta_n = 1 - \delta \sum_n w(n) \geq 1 - \delta$, the preceding holds for all n, which concludes our proof. □

Denote

$$n(h) = \min\{n : h \in \mathcal{H}_n\}, \tag{7.4}$$

The SRM paradigm searches for h that minimizes this bound, as formalized in the following pseudocode:

Structural Risk Minimization (SRM)

prior knowledge:
$\mathcal{H} = \bigcup_n \mathcal{H}_n$ where $\mathcal{H}_n$ has uniform convergence with $m^{\text{UC}}_{\mathcal{H}_n}$
$w : \mathbb{N} \to [0,1]$ where $\sum_n w(n) \leq 1$
define: ϵ_n as in Equation (7.1) ; $n(h)$ as in Equation (7.4)
input: training set $S \sim \mathcal{D}^m$, confidence δ
output: $h \in \operatorname{argmin}_{h \in \mathcal{H}} \left[L_S(h) + \epsilon_{n(h)}(m, w(n(h)) \cdot \delta) \right]$

THEOREM 7.5 *Let $\mathcal{H}$ be a hypothesis class such that $\mathcal{H} = \bigcup_{n \in \mathbb{N}} \mathcal{H}_n$, w each $\mathcal{H}_n$ has the uniform convergence property with sample complexity $m^{UC}_{\mathcal{H}_n}$ $w : \mathbb{N} \to [0,1]$ be such that $w(n) = \frac{6}{n^2 \pi^2}$. Then, $\mathcal{H}$ is nonuniformly learn using the SRM rule with rate*

$$m^{NUL}_{\mathcal{H}}(\epsilon, \delta, h) \leq m^{UC}_{\mathcal{H}_{n(h)}}\left(\epsilon/2, \frac{6\delta}{(\pi n(h))^2}\right).$$

Indeed, there is no inherent generalizability difference between hypotheses. The crucial aspect here is the dependency order between the initial choice of language (or, preference over hypotheses) and the training set. As we know from the basic Hoeffding's bound (Equation (4.2)), if we commit to any hypothesis *before* seeing the data, then we are guaranteed a rather small estimation error term $L_{\mathcal{D}}(h) \leq L_S(h) + \sqrt{\frac{\ln(2/\delta)}{2m}}$. Choosing a description language (or, equivalently, some weighting of hypotheses) is a weak form of committing to a hypothesis. Rather than committing to a single hypothesis, we spread out our commitment among many. As long as it is done independently of the training sample, our generalization bound holds. Just as the choice of a single hypothesis to be evaluated by a sample can be arbitrary, so is the choice of description language.

Other Notions of Learnability – Consistency

The notion of learnability can be further relaxed by allowing the needed sample sizes to depend not only on ϵ, δ, and h but also on the underlying data-generating probability distribution $\mathcal{D}$ (that is used to generate the training sample and to determine the risk). This type of performance guarantee is captured by the notion of *consistency*[1] of a learning rule.

DEFINITION 7.8 (Consistency) Let Z be a domain set, let $\mathcal{P}$ be a set of probability distributions over Z, and let $\mathcal{H}$ be a hypothesis class. A learning rule A is *consistent* with respect to $\mathcal{H}$ and $\mathcal{P}$ if there exists a function $m_{\mathcal{H}}^{\text{CON}} : (0,1)^2 \times \mathcal{H} \times \mathcal{P} \to \mathbb{N}$ such that, for every $\epsilon, \delta \in (0,1)$, every $h \in \mathcal{H}$, and every $\mathcal{D} \in \mathcal{P}$, if $m \geq m_{\mathcal{H}}^{\text{NUL}}(\epsilon, \delta, h, \mathcal{D})$ then with probability of at least $1 - \delta$ over the choice of $S \sim \mathcal{D}^m$ it holds that

$$L_{\mathcal{D}}(A(S)) \leq L_{\mathcal{D}}(h) + \epsilon.$$

Example 7.4 Consider the classification prediction algorithm `Memorize` defined as follows. The algorithm memorizes the training examples, and, given a test point x, it predicts the majority label among all labeled instances of x that exist in the training sample (and some fixed default label if no instance of x appears in the training set). It is possible to show (see Exercise 6) that the `Memorize` algorithm is universally consistent for every countable domain $\mathcal{X}$ and a finite label set $\mathcal{Y}$ (w.r.t. the zero-one loss).

Intuitively, it is not obvious that the `Memorize` algorithm should be viewed as a *learner*, since it lacks the aspect of generalization, namely, of using observed data to predict the labels of unseen examples. The fact that `Memorize` is a consistent algorithm for the class of all functions over any countable domain set therefore raises doubt about the usefulness of consistency guarantees. Furthermore, the sharp-eyed reader may notice that the "bad learner" we introduced in Chapter 2, which led to overfitting, is in fact the `Memorize` algorithm.

Discussing the Different Notions of Learnability

What Is the Risk of the Learned Hypothesis?

The first possible goal of deriving performance guarantees on a learning algorithm is bounding the risk of the output predictor. Here, both PAC learning and nonuniform learning give us an upper bound on the true risk of the learned hypothesis based on its empirical risk. Consistency guarantees do not provide such a bound. However, it is always possible to estimate the risk of the output predictor using a validation set (as will be described in Chapter 11).

How Many Examples Are Required to Be as Good as the Best Hypothesis in $\mathcal{H}$?

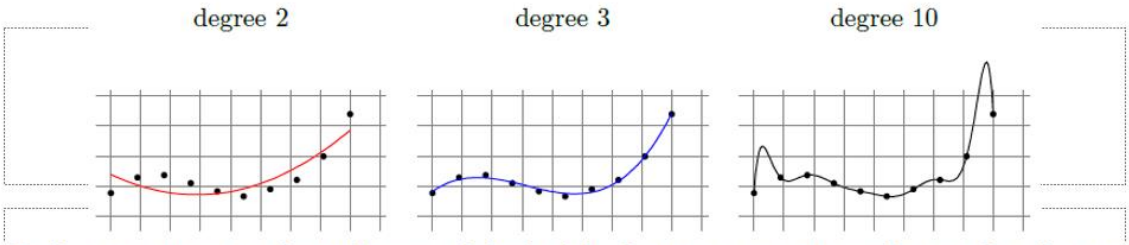
When approaching a learning problem, a natural question is how many examples we need to collect in order to learn it. Here, PAC learning gives a crisp answer. However, for both nonuniform learning and consistency, we do not know in advance how many examples are required to learn $\mathcal{H}$. In nonuniform learning this number depends on the best hypothesis in $\mathcal{H}$, and in consistency it also depends on the underlying distribution. In this sense, PAC learning is the only useful definition of learnability. On the flip side, one should keep in mind that even if the estimation error of the predictor we learn is small, its risk may still be large if $\mathcal{H}$ has a large approximation error. So, for the question "How many examples are required to be as good as the Bayes optimal predictor?" even PAC guarantees do not provide us with a crisp answer. This reflects the fact that the usefulness of PAC learning relies on the quality of our prior knowledge.

PAC guarantees also help us to understand what we should do next if our learning algorithm returns a hypothesis with a large risk, since we can bound the part of the error that stems from estimation error and therefore know how much of the error is attributed to approximation error. If the approximation error is large, we know that we should use a different hypothesis class. Similarly, if a nonuniform algorithm fails, we can consider a different weighting function over (subsets of) hypotheses. However, when a consistent algorithm fails, we have no idea whether this is because of the estimation error or the approximation error. Furthermore, even if we are sure we have a problem with the estimation error term, we do not know how many more examples are needed to make the estimation error small.

How to Learn? How to Express Prior Knowledge?

Maybe the most useful aspect of the theory of learning is in providing an answer to the question of "how to learn." The definition of PAC learning yields the limitation of learning (via the No-Free-Lunch theorem) and the necessity of prior knowledge. It gives us a crisp way to encode prior knowledge by choosing a hypothesis class, and once this choice is made, we have a generic learning rule – ERM. The definition of nonuniform learnability also yields a crisp way to encode prior knowledge by specifying weights over (subsets of) hypotheses of $\mathcal{H}$. Once this choice is made, we again have a generic learning rule – SRM. The SRM rule is also advantageous in model selection tasks, where prior knowledge is partial.

Consider the problem of fitting a one dimensional polynomial to data; namely, our goal is to learn a function, $h : \mathbb{R} \to \mathbb{R}$, and as prior knowledge we consider the hypothesis class of polynomials. However, we might be uncertain regarding which degree d would give the best results for our data set: A small degree might not fit the data well (i.e., it will have a large approximation error), whereas a high degree might lead to overfitting (i.e., it will have a large estimation error). In the following we depict the result of fitting a polynomial of degrees 2, 3, and 10 to the same training set.



It is easy to see that the empirical risk decreases as we enlarge the degree. Therefore, if we choose $\mathcal{H}$ to be the class of all polynomials up to degree 10 then the ERM rule with respect to this class would output a 10 degree polynomial and would overfit. On the other hand, if we choose too small a hypothesis class, say, polynomials up to degree 2, then the ERM would suffer from underfitting (i.e., a large approximation error). In contrast, we can use the SRM rule on the set of all polynomials, while ordering subsets of $\mathcal{H}$ according to their degree, and this will yield a 3rd degree polynomial since the combination of its empirical risk and the bound on its estimation error is the smallest. In other words, the SRM rule enables us to select the right model on the basis of the data itself. The price we pay for this flexibility (besides a slight increase of the estimation error relative to PAC learning w.r.t. the optimal degree) is that we do not know in advance how many examples are needed to compete with the best hypothesis in $\mathcal{H}$.

Unlike the notions of PAC learnability and nonuniform learnability, the definition of consistency does not yield a natural learning paradigm or a way to encode prior knowledge. In fact, in many cases there is no need for prior knowledge at all. For example, we saw that even the `Memorize` algorithm, which intuitively should not be called a learning algorithm, is a consistent algorithm for any class defined over a countable domain and a finite label set. This hints that consistency is a very weak requirement.

Which Learning Algorithm Should We Prefer?

One may argue that even though consistency is a weak requirement, it is desirable that a learning algorithm will be consistent with respect to the set of all functions from $\mathcal{X}$ to $\mathcal{Y}$, which gives us a guarantee that for enough training examples, we will always be as good as the Bayes optimal predictor. Therefore, if we have two algorithms, where one is consistent and the other one is not consistent, we should prefer the consistent algorithm. However, this argument is problematic for two reasons. First, maybe it is the case that for most "natural" distributions we will observe in practice that the sample complexity of the consistent algorithm will be so large so that in every practical situation we will not obtain enough examples to enjoy this guarantee. Second, it is not very hard to make any PAC or nonuniform learner consistent with respect to the class of all functions from $\mathcal{X}$ to $\mathcal{Y}$. Concretely, consider a countable domain, $\mathcal{X}$, a finite label set $\mathcal{Y}$, and a hypothesis class, $\mathcal{H}$, of functions from $\mathcal{X}$ to $\mathcal{Y}$. We can make any nonuniform learner for $\mathcal{H}$ be consistent with respect to the class of *all* classifiers from $\mathcal{X}$ to $\mathcal{Y}$ using the following simple trick: Upon receiving a training set, we will first run the nonuniform learner over the training set, and then we will obtain a bound on the true risk of the learned predictor. If this bound is small enough we are done. Otherwise, we revert to the `Memorize` algorithm. This simple modification makes the algorithm consistent with respect to all functions from $\mathcal{X}$ to $\mathcal{Y}$. Since it is easy to make any algorithm consistent, it may not be wise to prefer one algorithm over the other just because of consistency considerations.

The Runtime of Learning

So far in the book we have studied the statistical perspective of learning, namely, how many samples are needed for learning. In other words, we focused on the amount of information learning requires. However, when considering automated learning, computational resources also play a major role in determining the complexity of a task: that is, how much *computation* is involved in carrying out a learning task. Once a sufficient training sample is available to the learner, there is some computation to be done to extract a hypothesis or figure out the label of a given test instance. These computational resources are crucial in any practical application of machine learning. We refer to these two types of resources as the *sample complexity* and the *computational complexity*. In this chapter, we turn our attention to the computational complexity of learning.

The actual runtime (in seconds) of an algorithm depends on the specific machine the algorithm is being implemented on (e.g., what the clock rate of the machine's CPU is). To avoid dependence on the specific machine, it is common to analyze the runtime of algorithms in an asymptotic sense. For example, we say that the computational complexity of the merge-sort algorithm, which sorts a list of n items, is $O(n \log(n))$. This implies that we can implement the algorithm on any machine that satisfies the requirements of some accepted abstract model of computation, and the actual runtime in seconds will satisfy the following: there exist constants c and n_0, which can depend on the actual machine, such that, for any value of $n > n_0$, the runtime in seconds of sorting any n items will be at most $c\,n \log(n)$. It is common to use the term *feasible* or *efficiently computable* for tasks that can be performed by an algorithm whose running time is $O(p(n))$ for some polynomial function p. One should note that this type of analysis depends on defining what is the input size n of any instance to which the algorithm is expected to be applied. For "purely algorithmic" tasks, as discussed in the common computational complexity literature, this input size is clearly defined; the algorithm gets an input instance, say, a list to be sorted, or an arithmetic operation to be calculated, which has a well defined size (say, the number of bits in its representation). For machine learning tasks, the notion of an input size is not so clear. An algorithm aims to detect some pattern in a data set and can only access random samples of that data.

Computational Complexity of Learning

Recall that a learning algorithm has access to a domain of examples, Z, a hypothesis class, $\mathcal{H}$, a loss function, ℓ, and a training set of examples from Z that are sampled i.i.d. according to an unknown distribution $\mathcal{D}$. Given parameters ϵ, δ, the algorithm should output a hypothesis h such that with probability of at least $1 - \delta$,

$$L_{\mathcal{D}}(h) \leq \min_{h' \in \mathcal{H}} L_{\mathcal{D}}(h') + \epsilon.$$

As mentioned before, the actual runtime of an algorithm in seconds depends on the specific machine. To allow machine independent analysis, we use the standard approach in computational complexity theory. First, we rely on a notion of an abstract machine, such as a Turing machine (or a Turing machine over the reals (Blum, Shub & Smale 1989)). Second, we analyze the runtime in an asymptotic sense, while ignoring constant factors, thus the specific machine is not important as long as it implements the abstract machine. Usually, the asymptote is with respect to the size of the input to the algorithm. For example, for the merge-sort algorithm mentioned before, we analyze the runtime as a function of the number of items that need to be sorted.

In the context of learning algorithms, there is no clear notion of "input size." One might define the input size to be the size of the training set the algorithm receives, but that would be rather pointless. If we give the algorithm a very large number of examples, much larger than the sample complexity of the learning problem, the algorithm can simply ignore the extra examples. Therefore, a larger training set does not make the learning problem more difficult, and, consequently, the runtime available for a learning algorithm should not increase as we increase the size of the training set. Just the same, we can still analyze the runtime as a function of natural parameters of the problem such as the target accuracy, the confidence of achieving that accuracy, the dimensionality of the domain set, or some measures of the complexity of the hypothesis class with which the algorithm's output is compared.

To illustrate this, consider a learning algorithm for the task of learning axis aligned rectangles. A specific problem of learning axis aligned rectangles is derived by specifying ϵ, δ, and the dimension of the instance space. We can define a sequence of problems of the type "rectangles learning" by fixing ϵ, δ and varying the dimension to be $d = 2, 3, 4, \ldots$. We can also define another sequence of "rectangles learning" problems by fixing d, δ and varying the target accuracy to be $\epsilon = \frac{1}{2}, \frac{1}{3}, \ldots$. One can of course choose other sequences of such problems. Once a sequence of the problems is fixed, one can analyze the asymptotic runtime as a function of variables of that sequence.

Before we introduce the formal definition, there is one more subtlety we need to tackle. On the basis of the preceding, a learning algorithm can "cheat," by transferring the computational burden to the output hypothesis. For example, the algorithm can simply define the output hypothesis to be the function that stores the training set in its memory, and whenever it gets a test example x it calculates the ERM hypothesis on the training set and applies it on x. Note that in this case, our algorithm has a fixed output (namely, the function that we have just described) and can run in constant time. However, learning is still hard – the hardness is now in implementing the output classifier to obtain a label prediction. To prevent this "cheating," we shall require that the output of a learning algorithm must be applied to predict the label of a new example in time that does not exceed the runtime of training (that is, computing the output classifier from the input training sample). In the next subsection the advanced reader may find a formal definition of the computational complexity of learning.

Formal Definition:

The definition that follows relies on a notion of an underlying abstract machine, which is usually either a Turing machine or a Turing machine over the reals. We will measure the computational complexity of an algorithm using the number of "operations" it needs to perform, where we assume that for any machine that implements the underlying abstract machine there exists a constant c such that any such "operation" can be performed on the machine using c seconds.

DEFINITION 8.1 (The Computational Complexity of a Learning Algorithm) We define the complexity of learning in two steps. First we consider the computational complexity of a fixed learning problem (determined by a triplet $(Z, \mathcal{H}, \ell)$ – a domain set, a benchmark hypothesis class, and a loss function). Then, in the second step we consider the rate of change of that complexity along a sequence of such tasks.

1. Given a function $f : (0,1)^2 \to \mathbb{N}$, a learning task $(Z, \mathcal{H}, \ell)$, and a learning algorithm, $\mathcal{A}$, we say that $\mathcal{A}$ solves the learning task in time $O(f)$ if there exists some constant number c, such that for every probability distribution $\mathcal{D}$ over Z, and input $\epsilon,\ \delta \in (0,1)$, when $\mathcal{A}$ has access to samples generated i.i.d. by $\mathcal{D}$,
   - $\mathcal{A}$ terminates after performing at most $cf(\epsilon,\ \delta)$ operations
   - The output of $\mathcal{A}$, denoted $h_{\mathcal{A}}$, can be applied to predict the label of a new example while performing at most $cf(\epsilon,\ \delta)$ operations
   - The output of $\mathcal{A}$ is probably approximately correct; namely, with probability of at least $1-\delta$ (over the random samples $\mathcal{A}$ receives), $L_{\mathcal{D}}(h_{\mathcal{A}}) \leq \min_{h' \in \mathcal{H}} L_{\mathcal{D}}(h') + \epsilon$

2. Consider a sequence of learning problems, $(Z_n, \mathcal{H}_n, \ell_n)_{n=1}^{\infty}$, where problem n is defined by a domain Z_n, a hypothesis class $\mathcal{H}_n$, and a loss function ℓ_n.

   Let $\mathcal{A}$ be a learning algorithm designed for solving learning problems of this form. Given a function $g : \mathbb{N} \times (0,1)^2 \to \mathbb{N}$, we say that the runtime of $\mathcal{A}$ with respect to the preceding sequence is $O(g)$, if for all n, $\mathcal{A}$ solves the problem $(Z_n, \mathcal{H}_n, \ell_n)$ in time $O(f_n)$, where $f_n : (0,1)^2 \to \mathbb{N}$ is defined by $f_n(\epsilon, \delta) = g(n, \epsilon, \delta)$.

We say that $\mathcal{A}$ is an *efficient* algorithm with respect to a sequence $(Z_n, \mathcal{H}_n, \ell_n)$ if its runtime is $O(p(n, 1/\epsilon, 1/\delta))$ for some polynomial p.

From this definition we see that the question whether a general learning problem can be solved efficiently depends on how it can be broken into a sequence of specific learning problems. For example, consider the problem of learning a finite hypothesis class. As we showed in previous chapters, the ERM rule over $\mathcal{H}$ is guaranteed to (ϵ, δ)-learn $\mathcal{H}$ if the number of training examples is order of $m_{\mathcal{H}}(\epsilon, \delta) = \log(|\mathcal{H}|/\delta)/\epsilon^2$. Assuming that the evaluation of a hypothesis on an example takes a constant time, it is possible to implement the ERM rule in time $O(|\mathcal{H}|\, m_{\mathcal{H}}(\epsilon, \delta))$ by performing an exhaustive search over $\mathcal{H}$ with a training set of size $m_{\mathcal{H}}(\epsilon, \delta)$. For any fixed finite $\mathcal{H}$, the exhaustive search algorithm runs in polynomial time. Furthermore, if we define a sequence of problems in which $|\mathcal{H}_n| = n$, then the exhaustive search is still considered to be efficient. However, if we define a sequence of problems for which $|\mathcal{H}_n| = 2^n$, then the sample complexity is still polynomial in n but the computational complexity of the exhaustive search algorithm grows exponentially with n (thus, rendered inefficient).

Surrogate Loss Functions

As mentioned, and as we will see in the next chapters, convex problems can be learned effficiently. However, in many cases, the natural loss function is not convex and, in particular, implementing the ERM rule is hard.

As an example, consider the problem of learning the hypothesis class of half-spaces with respect to the $0-1$ loss. That is,

$$\ell^{0-1}(\mathbf{w},(\mathbf{x},y)) = \mathbb{1}_{[y\neq \text{sign}(\langle \mathbf{w},\mathbf{x}\rangle)]} = \mathbb{1}_{[y\langle \mathbf{w},\mathbf{x}\rangle \leq 0]}.$$

This loss function is not convex with respect to $\mathbf{w}$ and indeed, when trying to minimize the empirical risk with respect to this loss function we might encounter local minima (see Exercise 1). Furthermore, as discussed in Chapter 8, solving the ERM problem with respect to the $0-1$ loss in the unrealizable case is known to be NP-hard.

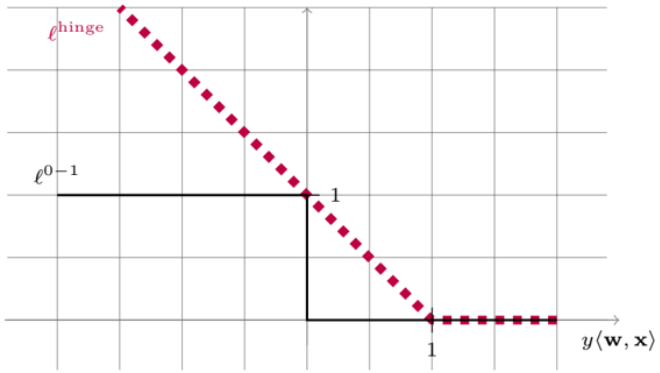
To circumvent the hardness result, one popular approach is to upper bound the nonconvex loss function by a convex surrogate loss function. As its name indicates, the requirements from a convex surrogate loss are as follows:

1. It should be convex.
2. It should upper bound the original loss.

For example, in the context of learning halfspaces, we can define the so-called *hinge* loss as a convex surrogate for the $0-1$ loss, as follows:

$$\ell^{\text{hinge}}(\mathbf{w},(\mathbf{x},y)) \stackrel{\text{def}}{=} \max\{0, 1 - y\langle \mathbf{w},\mathbf{x}\rangle\}.$$

Clearly, for all $\mathbf{w}$ and all $(\mathbf{x},y)$, $\ell^{0-1}(\mathbf{w},(\mathbf{x},y)) \leq \ell^{\text{hinge}}(\mathbf{w},(\mathbf{x},y))$.



Hence, the hinge loss satisfies the requirements of a convex surrogate loss function for the zero-one loss. An illustration of the functions ℓ^{0-1} and ℓ^{hinge} is given in the following.

- *Approximation error*: This is the term $\min_{\mathbf{w}\in\mathcal{H}} L_{\mathcal{D}}^{0-1}(\mathbf{w})$, which measures how well the hypothesis class performs on the distribution.

- *Estimation error*: This is the error that results from the fact that we only receive a training set and do not observe the distribution $\mathcal{D}$.

- *Optimization error*: This is the term $\left(\min_{\mathbf{w}\in\mathcal{H}} L_{\mathcal{D}}^{\text{hinge}}(\mathbf{w}) - \min_{\mathbf{w}\in\mathcal{H}} L_{\mathcal{D}}^{0-1}(\mathbf{w})\right)$ that measures the difference between the approximation error with respect to the surrogate loss and the approximation error with respect to the original loss. The optimization error is a result of our inability to minimize the training loss with respect to the original loss. The size of this error depends on the specific distribution of the data and on the specific surrogate loss we are using.

Rademacher Complexities

measures the rate of uniform convergence.

DEFINITION 26.1 (ϵ-Representative Sample) A training set S is called ϵ-representative (w.r.t. domain Z, hypothesis class $\mathcal{H}$, loss function ℓ, and distribution $\mathcal{D}$) if

$$\sup_{h \in \mathcal{H}} |L_{\mathcal{D}}(h) - L_S(h)| \leq \epsilon.$$

We have shown that if S is an $\epsilon/2$ representative sample then the ERM rule is ϵ-consistent, namely, $L_{\mathcal{D}}(\mathrm{ERM}_{\mathcal{H}}(S)) \leq \min_{h \in \mathcal{H}} L_{\mathcal{D}}(h) + \epsilon$.

To simplify our notation, let us denote

$$\mathcal{F} \stackrel{\text{def}}{=} \ell \circ \mathcal{H} \stackrel{\text{def}}{=} \{z \mapsto \ell(h, z) : h \in \mathcal{H}\},$$

and given $f \in \mathcal{F}$, we define

$$L_{\mathcal{D}}(f) = \mathop{\mathbb{E}}_{z \sim \mathcal{D}}[f(z)] \quad , \quad L_S(f) = \frac{1}{m} \sum_{i=1}^{m} f(z_i).$$

We define the *representativeness* of S with respect to $\mathcal{F}$ as the largest gap between the true error of a function f and its empirical error, namely,

$$\mathrm{Rep}_{\mathcal{D}}(\mathcal{F}, S) \stackrel{\text{def}}{=} \sup_{f \in \mathcal{F}} \big(L_{\mathcal{D}}(f) - L_S(f)\big). \tag{26.1}$$

Now, suppose we would like to estimate the representativeness of S using the sample S only. One simple idea is to split S into two disjoint sets, $S = S_1 \cup S_2$; refer to S_1 as a validation set and to S_2 as a training set. We can then estimate the representativeness of S by

$$\sup_{f \in \mathcal{F}} \big(L_{S_1}(f) - L_{S_2}(f)\big). \tag{26.2}$$

This can be written more compactly by defining $\boldsymbol{\sigma} = (\sigma_1, \ldots, \sigma_m) \in \{\pm 1\}^m$ to be a vector such that $S_1 = \{z_i : \sigma_i = 1\}$ and $S_2 = \{z_i : \sigma_i = -1\}$. Then, if we further assume that $|S_1| = |S_2|$ then Equation (26.2) can be rewritten as

$$\frac{2}{m} \sup_{f \in \mathcal{F}} \sum_{i=1}^{m} \sigma_i f(z_i). \tag{26.3}$$

The Rademacher complexity measure captures this idea by considering the expectation of the above with respect to a random choice of $\boldsymbol{\sigma}$. Formally, let $\mathcal{F} \circ S$ be the set of all possible evaluations a function $f \in \mathcal{F}$ can achieve on a sample S, namely,

$$\mathcal{F} \circ S = \{(f(z_1), \ldots, f(z_m)) : f \in \mathcal{F}\}.$$

Let the variables in $\boldsymbol{\sigma}$ be distributed i.i.d. according to $\mathbb{P}[\sigma_i = 1] = \mathbb{P}[\sigma_i = -1] = \frac{1}{2}$. Then, the Rademacher complexity of $\mathcal{F}$ with respect to S is defined as follows:

$$R(\mathcal{F} \circ S) \stackrel{\text{def}}{=} \frac{1}{m} \mathop{\mathbb{E}}_{\boldsymbol{\sigma} \sim \{\pm 1\}^m} \left[\sup_{f \in \mathcal{F}} \sum_{i=1}^{m} \sigma_i f(z_i) \right]. \tag{26.4}$$

More generally, given a set of vectors, $A \subset \mathbb{R}^m$, we define

$$R(A) \stackrel{\text{def}}{=} \frac{1}{m} \mathop{\mathbb{E}}_{\boldsymbol{\sigma}} \left[\sup_{\mathbf{a} \in A} \sum_{i=1}^{m} \sigma_i a_i \right]. \tag{26.5}$$

OR

1 Introduction

PAC learning guarantees were for finite hypothesis sets. However typical hypothesis sets in machine learning problems are infinite, e.g. set of all hyperplanes in SVM. We will generalize existing results and derive general learning guarantees for infinite hypothesis sets.

We will reduce the infinite hypothesis set to a finite set depending on the notion of complexity. First notion is *Rademacher complexity*, which is difficult to compute empirically for many hypothesis sets. We then study combinatorial notions of complexity, *growth function* and the *VC-dimension*. We relate Rademacher complexity to growth function, and then bound the growth function by the VC-dimension, which are easy to bound or compute in many cases.

2 Rademacher complexity

Consider a hypothesis set $H \subset \mathcal{Y}^{\mathcal{X}}$ and loss function $L : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$. Let $\mathcal{Z} = \mathcal{X} \times \mathcal{Y}$, then for each hypothesis $h \in H$, we can associate a function $g : \mathcal{Z} \to \mathbb{R}$ such that $g(x,y) = L(h(x),y)$ which captures the corresponding loss L. The family of loss function associated to hypothesis set H is defined as

$$G \triangleq \left\{ g \in \mathbb{R}^{\mathcal{Z}} : g(x,y) = L(h(x),y) \text{ for all } (x,y) \in \mathcal{X} \times \mathcal{Y}, h \in H \right\}.$$

The Rademacher complexity captures the richness of a family of functions by measuring the degree to which a hypothesis set can fit random noise.

Definition 2.1 (Rademacher random variable). A uniform random variable $X \in \{-1,1\}$ is called a **Rademacher random variable**.

For any $g \in G$ and m-sized sample $T \in \mathcal{Z}^m$, we denote by $g_T \triangleq (g(z_1), \ldots, g(z_m)) \in \mathbb{R}^m$.

Definition 2.2 (Empirical Rademacher complexity). Let $G \subseteq [a,b]^{\mathcal{Z}}$ be a family of functions and a fixed labeled sample $T = (z_1, \ldots, z_m) \in \mathcal{Z}^m$ of size m. Then, the **empirical Rademacher complexity** of G with respect to the labeled sample T is defined as

$$\hat{\mathcal{R}}_T(G) \triangleq \mathbb{E}\left[\sup_{g \in G} \frac{1}{m} \langle \sigma, g_T \rangle \right],$$

where $\sigma \in \{-1,1\}^m$, is an m-length vector of independent Rademacher variables.

Remark 1. The inner product $\langle \sigma, g_T \rangle$ measures the correlation of g_T with random noise σ, and the supremum over all $g \in G$ measures how well the hypothesis class H correlates with σ over the labeled sample T. This is a measure of richness/complexity of class G, since richer families can generate more g_T and better correlate with random noise on average.

Definition 2.3 (Rademacher complexity). Let D be the unknown fixed distribution according to which labeled sample T is drawn in an *i.i.d.* fashion. For any $m \in \mathbb{N}$, the **Rademacher complexity** of a family of loss functions G is mean of empirical Rademacher complexity for sample T, and denoted by

$$\mathcal{R}_m(G) \triangleq \mathbb{E}\hat{\mathcal{R}}_T(G).$$

Lemma 2.4. *Let $G \subseteq [0,1]^{\mathcal{Z}}$ be a family of functions. Then, for any $\delta > 0$, with probability at least $1 - \frac{\delta}{2}$*

$$\mathcal{R}_m(G) \leqslant \hat{\mathcal{R}}_T(G) + \sqrt{\frac{\ln \frac{2}{\delta}}{2m}}.$$

Proof. We observe that $\mathbb{E}\hat{\mathcal{R}}_T(G)$, and that $\hat{\mathcal{R}}_G$ satisfies the bounded difference property with bounding vector $\frac{1}{m}\mathbf{1}$. The result follows from the McDiarmid's inequality. □

Theorem 2.5. *Let $G \subseteq [0,1]^{\mathcal{Z}}$ be a family of functions. Then, for any $\delta > 0$, with probability at least $1-\delta$, both the inequalities hold for all $g \in G$*

$$\mathbb{E}g(z) \leqslant \frac{1}{m}\langle \mathbf{1}, g_T\rangle + 2\mathcal{R}_m(G) + \sqrt{\frac{\ln\frac{1}{\delta}}{2m}}, \qquad \mathbb{E}g(z) \leqslant \frac{1}{m}\langle \mathbf{1}, g_T\rangle + 2\hat{\mathcal{R}}_T(G) + 3\sqrt{\frac{\ln\frac{2}{\delta}}{2m}}.$$

Proof. For any labeled sample $T \in \mathcal{Z}^m$ and loss function $g \in G$, we denote the empirical average of g over labeled sample T as

$$\hat{\mathbb{E}}_T[g] \triangleq \frac{1}{m}\langle \mathbf{1}, g_T\rangle .$$

We consider the following function $\Phi : \mathcal{Z}^m \to \mathbb{R}$,

$$\Phi(T) \triangleq \sup_{g\in G}(\mathbb{E}g - \hat{\mathbb{E}}_T[g]).$$

Consider two samples T, T' differing at a single example z_m in T and z'_m in T'. Then, we can write

$$\Phi(T') - \Phi(T) \leqslant \sup_{g\in G}(\hat{\mathbb{E}}_{T'}[g] - \hat{\mathbb{E}}_T[g]) = \sup_{g\in G}\frac{g(z_m) - g(z'_m)}{m} \leqslant \frac{1}{m}.$$

Similarly, we can obtain $\Phi(T) - \Phi(T') \leqslant \frac{1}{m}$. Hence, the function Φ has the bounded difference property with bounding vector $\frac{1}{m}\mathbf{1}$. By McDiarmid's inequality, for any $\delta > 0$, with probability at least $1 - \frac{\delta}{2}$, we have

$$\Phi(T) \leqslant \mathbb{E}\Phi(T) + \sqrt{\frac{\ln\frac{2}{\delta}}{2m}}.$$

We next bound the mean of the $\Phi(T)$ by the difference of empirical average for samples T, T', sampled *i.i.d.* from the fixed unknown distribution D, by applying the Jensen's inequality to convex function supremum. We get

$$\mathbb{E}\Phi(T) = \mathbb{E}\left[\sup_{g\in G}(\mathbb{E}[g] - \hat{\mathbb{E}}_T[g])\right] = \mathbb{E}\left[\sup_{g\in G}\mathbb{E}\left[\hat{\mathbb{E}}_{T'}[g] - \hat{\mathbb{E}}_T[g]\right]\right] \leqslant \mathbb{E}\left[\sup_{g\in G}(\hat{\mathbb{E}}_{T'}[g] - \hat{\mathbb{E}}_T[g])\right].$$

Since T, T' are *i.i.d.*, the inner product $\langle \sigma, g_{T'} - g_T \rangle$ for *i.i.d.* Rademacher vector $\sigma \in \{-1,1\}^m$ has same distribution as $\langle \mathbf{1}, g_{T'} - g_T \rangle$. Therefore, we have

$$\mathbb{E}\Phi(T) \leqslant \mathbb{E}\left[\sup_{g\in G} \frac{1}{m} \langle \sigma, g_{T'} - g_T \rangle\right] \leqslant \mathbb{E}\left[\sup_{g\in G} \frac{1}{m} \langle \sigma, g_{T'} \rangle\right] + \mathbb{E}\left[\sup_{g\in G} \frac{1}{m} \langle -\sigma, g_T \rangle\right] = 2\mathcal{R}_m(G).$$

□

Lemma 2.6. *Let $\mathcal{Y} = \{-1,1\}$ and $\mathcal{Z} = \mathcal{X} \times \mathcal{Y}$, and the hypothesis set $H \subseteq \mathcal{Y}^{\mathcal{X}}$ be a family of functions and let G be the family of loss functions associated to the hypothesis set H for the zero-one loss, i.e.*

$$G = \left\{(x,y) \mapsto \mathbb{1}_{\{h(x)\neq y\}} : h \in H\right\}.$$

For any labeled sample $T \in \mathcal{Z}^m$, let $S = T_{\mathcal{X}}$ denote its projection over $\mathcal{X}$, i.e. $S = (x_1, \ldots, x_m) \in \mathcal{X}^m$. Then,

$$\hat{\mathcal{R}}_T(G) = \frac{1}{2}\hat{\mathcal{R}}_S(H).$$

Proof. For any sample $T = ((x_i, y_i) \in \mathcal{X} \times \mathcal{Y} : i \in [m])$ where $\mathcal{Y} = \{-1,1\}$, we have $\mathbb{1}_{\{h(x_i)\neq y_i\}} = \frac{1-y_i h(x_i)}{2}$. Therefore, we can write

$$\hat{\mathcal{R}}_T(G) = \mathbb{E}\left[\sup_{h\in H} \frac{1}{m} \sum_{i=1}^m \sigma_i \mathbb{1}_{\{h(x_i)\neq y_i\}}\right] = \mathbb{E}\left[\sup_{h\in H} \frac{1}{m} \sum_{i=1}^m \sigma_i \Big(\frac{1-y_i h(x_i)}{2}\Big)\right].$$

Since $\sum_{i=1}^m \sigma_i$ remains constant for all $h \in H$ and its mean is zero, we can ignore this term. Further, $\sigma \circ y = (\sigma_i y_i \in \mathcal{Y} : i \in [m])$ has same distribution as $\sigma = (\sigma_i \in \mathcal{Y} : i \in [m])$, and therefore

$$\hat{\mathcal{R}}_T(G) = \frac{1}{2}\mathbb{E}\left[\sup_{h\in H} \frac{1}{m} \langle -\sigma, y \circ h_S(x) \rangle\right] = \frac{1}{2}\mathbb{E}\left[\sup_{h\in H} \frac{1}{m} \langle \sigma, h_S(x) \rangle\right] = \frac{1}{2}\hat{\mathcal{R}}_S(H).$$

□

Theorem 2.7 (Rademacher complexity bounds – binary classification). *Let $H \subseteq \mathcal{X}^{\mathcal{Y}}$ be a family of functions for $\mathcal{Y} = \{?1, +1\}$ and let D be the fixed and unknown distribution over the input space $\mathcal{X}$. Then, for any $\delta > 0$, with probability at least $1?\delta$ over a sample S of size m drawn* i.i.d. *according to D, each of the following holds for any hypothesis $h \in H$*

$$R(h) \leqslant \hat{R}(h) + \mathcal{R}_m(H) + \sqrt{\frac{\ln \frac{1}{\delta}}{2m}}, \qquad R(h) \leqslant \hat{R}(h) + \hat{\mathcal{R}}_S(H) + 3\sqrt{\frac{\ln \frac{2}{\delta}}{2m}}.$$

Proof. The result follow from Theorem 2.5 and Lemma 2.6. □

Remark 2. The second learning bound is data dependent, and very useful if we can efficiently compute the empirical Rademacher complexity $\hat{\mathcal{R}}_S(H)$. Since σ and $-\sigma$ have the same distribution, we get

$$\hat{\mathcal{R}}_S(H) = \mathbb{E}\left[\sup_{h\in H} \frac{1}{m} \langle -\sigma, h_S \rangle\right] = -\mathbb{E}\left[\inf_{h\in H} \frac{1}{m} \langle \sigma, h_S \rangle\right].$$

for a fixed value of σ, computing $\inf_{h\in H} \frac{1}{m} \langle \sigma, h_S \rangle$ is equivalent to an *empirical risk minimization* problem, which is known to be computationally hard for some hypothesis sets.

A McDiarmid's inequality

Definition A.1 (Martingale difference). A sequence of random variables $(V_n \in \mathbb{R} : n \in \mathbb{N})$ is a **martingale difference sequence** with respect to a random sequence $(X_n \in \mathbb{R} : n \in \mathbb{N})$ if V_n is a function of $X_1, \ldots, X_n$ for all $n \in \mathbb{N}$, and

$$\mathbb{E}[V_{n+1} \mid X_1, \ldots, X_n] = 0.$$

Lemma A.2. *Let V and Z be random variables satisfying $\mathbb{E}[V \mid Z] = 0$ and $f(Z) \leqslant V \leqslant f(Z) + c$ for some function f and constant $c \geqslant 0$. Then, for all $t > 0$, we have*

$$\mathbb{E}[e^{sV} \mid Z] \leqslant e^{t^2 c^2/8}.$$

Proof. The result follows from Hoeffding's Lemma for conditional expectation given Z, where $[a,b] = [f(Z), f(Z) + c]$. □

Theorem A.3 (Azuma's inequality). *Let $(V_n : n \in \mathbb{N})$ be a martingale difference sequence with respect to the random variables $(X_n : n \in \mathbb{N})$ and assume that for all $n \in \mathbb{N}$ there is a constant $c_n \geqslant 0$ and random variable Z_n, which is a function of $X_1, \ldots, X_{i-1}$, that satisfy $Z_i \leqslant V_i \leqslant Z_i + c$. Defining $\sigma^2 \triangleq \sum_{i=1}^m c_i^2 = \|c\|_2^2$, we have for all $\varepsilon > 0$ and $m \in \mathbb{N}$,*

$$P\left\{\sum_{i=1}^m V_i \geqslant \varepsilon\right\} \leqslant e^{-2\varepsilon^2/\sigma^2}, \qquad P\left\{\sum_{i=1}^m V_i \leqslant -\varepsilon\right\} \leqslant e^{-2\varepsilon^2/\sigma^2}.$$

Proof. For any $k \in \mathbb{N}$, we can define $S_k \triangleq \sum_{i=1}^k V_i$, then by Chernoff bound, we have

$$P\{S_m \geqslant \varepsilon\} \leqslant e^{-t\varepsilon}\mathbb{E}[e^t S_m] = e^{-t\varepsilon}\mathbb{E}[e^t S_{m-1}\mathbb{E}[e^{tV_m}|X_1, \ldots, X_{m-1}]] \leqslant e^{-t\varepsilon}\mathbb{E}[e^{tS_{m-1}}]e^{t^2 c_m^2/8} \leqslant \exp\left(-t\varepsilon + \frac{t^2\sigma^2}{8}\right).$$

The result for the first part follows by taking $t^* = \frac{4\varepsilon}{\sigma^2}$. The second part can be proved similarly. □

Definition A.4 (Bounded difference property). A function $f : \mathcal{X}^m \to \mathbb{R}$ is said to have the **bounded difference property**, if for all $i \in [m]$ there exists a constant $c_i > 0$ such that for any $x, y \in \mathbb{R}^m$ differing only at the ith location, we have

$$|f(x) - f(y)| \leqslant c_i. \tag{1}$$

The vector $c \in \mathbb{R}_+^m$ is called the **bounding vector**.

Theorem A.5 (McDiarmid's inequality). *Let $f : \mathcal{X}^m$ be a function with the bounded difference property with bounding vector $c \in \mathbb{R}_+^m$, and $(X_i \in \mathcal{X} : i \in [m])$ be a set of m independent random variables. Denoting $f(S) \triangleq f(X_1, \ldots, X_m)$, for all $\varepsilon > 0$, we have*

$$P\{f(S) - \mathbb{E}f(S) \geqslant \varepsilon\} \leqslant e^{-2\varepsilon^2/\|c\|_2^2}, \qquad P\{f(S) - \mathbb{E}f(S) \leqslant -\varepsilon\} \leqslant e^{-2\varepsilon^2/\|c\|_2^2}.$$

Proof. It suffices to show that $f(S) - \mathbb{E}f(S) = \sum_{i=1}^{m} V_i$ for some martingale difference sequence $(V_i : i \in [m])$ with respect to the sequence $(X_i : i \in [m])$ and $Z_i \leqslant V_i \leqslant Z_i + c_i$ for some random variable Z_i a function of $X_1, \ldots, X_{i-1}$.

Let $V = f(S) - \mathbb{E}f(S)$, then we define such a sequence $(V_1, \ldots, V_m)$ as

$$V_k = \mathbb{E}[V \mid X_1, \ldots, X_k] - \mathbb{E}[V \mid X_1, \ldots, X_{k-1}],\ k \in [m], \qquad \sum_{k=1}^{m} V_k = V.$$

We can verify that $(V_i : i \in [m])$ is martingale difference equation, since V_k is a function of $X_1, \ldots, X_k$ and $\mathbb{E}[V_k | X_1, \ldots, X_{k-1}] = 0$ for each $k \in [m]$. Since $\mathbb{E}f(S)$ is not random, we can write

$$V_k = \mathbb{E}[f(S) \mid X_1, \ldots, X_k] - \mathbb{E}[f(S) \mid X_1, \ldots, X_{k-1}],$$

and define upper and lower bounds for V_k as

$$W_k \triangleq \sup_x \mathbb{E}[f(S) \mid X_1, \ldots, X_{k-1}, x] - \mathbb{E}[f(S) \mid X_1, \ldots, X_{k-1}], \quad U_k \triangleq \inf_x \mathbb{E}[f(S) \mid X_1, \ldots, X_{k-1}, x] - \mathbb{E}[f(S) \mid X_1, \ldots, X_{k-1}].$$

Then the result follows from the hypothesis (1), which implies that

$$W_k - U_k = \sup_{x,y \in \mathcal{X}} E[f(S) \mid X_1, \ldots, X_{k-1}, x] - E[f(S) \mid X_1, \ldots, X_{k-1}, y] \leqslant c_k.$$

The **Rademacher distribution** is distribution of discrete probability when a random variable has a half probability of being +1 and half probability of being -1. With the help of **`sympy.stats.Rademacher()`** method, we can create the random variable having randemacher distribution by using `sympy.stats.Rademacher()` method.

Syntax : *`sympy.stats.Rademacher(name)`* ***Return :*** *Return the rademacher distribution.*

Example #1 : In this example, we can see that by using `sympy.stats.Rademacher()` method, we are able to get the random variable of Rademacher distribution by using this method.

```
# Import sympy and Rademacher
from sympy.stats import Rademacher, density


# Using sympy.stats.Rademacher() method
X = Rademacher('X')
gfg = density(X).dict

print(gfg)
```

Output :

```
{1: 1/2, -1: 1/2}
```

Example #2 :

```
# Import sympy and Rademacher
from sympy.stats import Rademacher, density, P


# Using sympy.stats.Rademacher() method
X = Rademacher('X')
gfg = density(X).dict

print(P(gfg >= 0))
```

Output :

```
1/2
```

The Natarajan Dimension:

The Natarajan Dimension

Natarajan dimension, which is a generalization of the VC dimension to classes of multiclass predictors.

let $\mathcal{H}$ be a hypothesis class of multiclass predictors; namely, each $h \in \mathcal{H}$ is a function from $\mathcal{X}$ to $[k]$.

To define the Natarajan dimension, we first generalize the definition of shattering.

DEFINITION 29.1 (Shattering (Multiclass Version)) We say that a set $C \subset \mathcal{X}$ is shattered by $\mathcal{H}$ if there exist two functions $f_0, f_1 : C \to [k]$ such that

- For every $x \in C$, $f_0(x) \neq f_1(x)$.
- For every $B \subset C$, there exists a function $h \in \mathcal{H}$ such that

$$\forall x \in B, h(x) = f_0(x) \text{ and } \forall x \in C \setminus B, h(x) = f_1(x).$$

DEFINITION 29.2 (Natarajan Dimension) The Natarajan dimension of $\mathcal{H}$, denoted $\text{Ndim}(\mathcal{H})$, is the maximal size of a shattered set $C \subset \mathcal{X}$.

It is not hard to see that in the case that there are exactly two classes, $\text{Ndim}(\mathcal{H}) = \text{VCdim}(\mathcal{H})$. Therefore, the Natarajan dimension generalizes the VC dimension. We next show that the Natarajan dimension allows us to generalize the fundamental theorem of statistical learning from binary classification to multiclass classification.

OR

1 Backgrounds

Many tasks in statistical learning concern finding a good representation of the true relationship underlying the observations out of a perhaps huge family of functions. To this end, an intuitive and prominent approach is empirical risk minimization (ERM). Given i.i.d. observations $\{(X_i, Y_i)\}_{i=1}^n \in \mathcal{X} \times \mathcal{Y}$ and a loss function $\ell \colon \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}^+$, to learn a predictor $f \in \mathcal{F}$ for Y with the smallest loss, ERM selects the candidate with the smallest empirical prediction error:

$$\widehat{f} = \operatorname*{argmin}_{f \in \mathcal{F}} \widehat{L}_n(f), \quad \widehat{L}_n(f) := \frac{1}{n} \sum_{i=1}^{n} \ell(f(X_i), Y_i),$$

Here, it is hoped that $\widehat{L}_n(f)$ is a good estimate of the true prediction error $L(f) = \mathbb{E}[\ell(f(X), Y)]$. The complexity of the function class $\mathcal{F}$ is a crucial quantity that impacts the performance of such $\widehat{f}$. The more complex $\mathcal{F}$ is, the more likely it is that the smallest empirical risk occurs only by chance, and that the estimate for $\widehat{f}$ is far from its true value. Understanding the complexity of function classes to characterize the learning performance is a fundamental task in statistical learning theory.

1.1 Natarajan dimension

This paper studies the Natajaran dimension (Natarajan, 1989), a measure of the complexity in learning function classes for *multi-class classification* problems. It generalizes the well-known Vapnik-Chervonenkis (VC) dimension (Vapnik and Chervonenkis, 2015) for binary classification function classes. A closely related quantity, the graph dimension, is also defined below.

Definition 1.1. *Let $\mathcal{H}$ be a class of functions $h \colon \mathcal{X} \to \mathcal{Y}$, and let $S \subseteq \mathcal{X}$. We say that $\mathcal{H}$ G-shatters S if there exists an $f \colon S \to \mathcal{Y}$ such that for every $T \subseteq S$, there exsits a $g \in \mathcal{H}$ such that*

$$\forall x \in T,\ g(x) = f(x), \quad \text{and} \quad \forall x \in S \backslash T,\ g(x) \neq f(x).$$

We say that $\mathcal{H}$ N-shatters S if there exists $f_1, f_2 \colon S \to \mathcal{Y}$ such that $f_1(x) \neq f_2(x)$ for all $x \in S$, and for every $T \subseteq S$, there exists some $g \in \mathcal{H}$ such that

$$\forall x \in T,\ g(x) = f_1(x), \quad \text{and} \quad \forall x \in S \backslash T,\ g(x) = f_2(x).$$

The graph dimension of $\mathcal{H}$, denoted as $d_G(\mathcal{H})$, is the maximal cardinality of any set G-shattered by $\mathcal{H}$. The Natarajan dimension of $\mathcal{H}$, denoted as $d_N(\mathcal{H})$, is the maximal cardinality of any set N-shattered by $\mathcal{H}$.

Both the graph dimension and the Natarajan dimension coincide with the VC dimension for $\mathcal{Y} = \{0, 1\}$, and it is shown that $d_N(\mathcal{H}) \leq d_G(\mathcal{H}) \leq 4.67 \log_2(|\mathcal{Y}|) d_N(\mathcal{H})$ (Ben-David et al., 1992). In this work, we only focus on the Natarajan dimension, and the results on the graph dimension can be easily obtained.

Similar to the VC dimension, the Natarajan dimension can be used to characterize the generalization of ERM – more precisely, the true prediction error of the empirical risk minimizer – using multi-class classification function classes (Natarajan, 1989; Ben-David et al., 1992; Daniely et al., 2011). However, unlike the extensively studied VC dimension, results on upper bounds on the Natarajan dimensions are relatively rare; existing results only cover linear function classes and reduction tree classes (Daniely et al., 2011). This work provides two more instances, decision trees (and random forests) and neural networks (fully-connected ones with linear, binary and ReLU activations). We study tree-based function classes in Section 2 and neural networks in Section 3, while all proofs are in Section 4.

1.2 Growth functions

Our theoretical proof will follow the classical idea of bounding growth functions.

Definition 1.2. *Let $\mathcal{H}$ be a class of functions $h \colon \mathcal{X} \to \mathcal{Y}$. The growth function of $\mathcal{H}$ is defined as*

$$G(\mathcal{H}, n) := \max_{x_1, \ldots, x_n \in \mathcal{X}} \left| \left\{ \big(f(x_1), f(x_2), \ldots, f(x_n)\big) \colon f \in \mathcal{H} \right\} \right|,$$

which is the number of distinct realizations of $f \in \mathcal{H}$ on any n feature values, where for a finite set A, we write $|A|$ as the cardinality of A.

More specifically, following the definition of Natarajan dimension, the functions in $\mathcal{H}$ has at least $2^{d_N(\mathcal{H})}$ different configurations on $d_N(\mathcal{H})$ feature values in $\mathcal{X}$. This fact allows us to utilize $2^{d_N(\mathcal{H})} \leq G(\mathcal{H}, d_N(\mathcal{H}))$ to obtain proper upper bounds on $d_N(\mathcal{H})$ for the function classes $\mathcal{H}$ we study in this paper.

2 Tree-based function class

Decision trees and random forests are popular tree-based machine learning methods that could be used for multi-class classification. This section provides upper bounds on the Natarajan dimensions of these classes.

We first study the function class $\Pi_{L,d}^{\text{dtree}}$, each element of which is a depth-L d-class decision tree. A depth-L decision tree is a full binary tree, where each internal node v is associated with a feature $i_v \in \{1, \ldots, p\}$ and a threshold $\theta_v \in \mathbb{R}$, and each leaf node is associated with a class $k \in \{1, \ldots, d\}$. For input $x \in \mathbb{R}^p$, the output is obtained by traversing a path of length $L - 1$ from the root node to the leaf node. At each node v, if $x_{i_v} \leq \theta_v$ then we continue to its left child node, and to its right child node otherwise. The final classification is given by the class associated with the leaf node we arrive at.

Remark 2.1. *Daniely et al. (2012) also studies the Natarajan dimension for decision trees, however, under a different definition: they assume there is a bijection between the leaf nodes and the d classes, and the internal nodes are from a general class of binary functions. Instead, we allow multiple leaf nodes to represent the same class, but consider more restricted binary classification rules for the internal nodes. We will obtain different upper bounds with different proof techniques.*

The following theorem establishes upper bounds on the Natarajan dimension of $\Pi_{L,d}^{\text{dtree}}$.

Theorem 2.2. *The Natarajan dimension of $\Pi_{L,d}^{dtree}$ with inputs from $\mathbb{R}^p$ is no greater than $\mathcal{O}(L2^L \log(pd))$.*

We then consider the function class of random forests, denoted by $\Pi_{L,T,d}^{\text{forest}}$, each element of which is a random forest classifier $F(\cdot)$ consisting of T depth-L d-class decision trees $f_j(\cdot)$, $j = 1, \ldots, T$. Given any $x \in \mathbb{R}^p$, the output of a random forest is given by $F(x) = \text{argmax}_{1 \le k \le d} \sum_{j=1}^{T} \mathbf{1}\{f_j(x) = k\}$, the most-frequently predicted class among all T trees. Its Natarajan dimension can be bounded as follows.

Theorem 2.3. *The Natarajan dimension of $\Pi_{L,T,d}^{forest}$ with inputs from $\mathbb{R}^p$ is no greater than $\mathcal{O}(LT2^L \log(pd))$.*

As we discussed in Section 1.3, upper bounds on the VC dimension of decision trees with real-valued features appear to be very recent results (Leboeuf et al., 2022). In particular, the independent recent work of Leboeuf et al. (2022) shows that the VC dimension of binary decision trees with L_T leaves for p real-valued features and the same splitting rules as ours is $\mathcal{O}(L_T \log(L_t p))$. Since the number of leaves is $2^L - 1$ for a tree of depth L, our bound $\mathcal{O}(L2^L \log(pd))$ in Theorem 2.2 only pays a price of $\log(d)$ for d-class classification compared with the results in Leboeuf et al. (2022), while the rate in the number of leaves is the same.

3 Neural network function class

For a number d of actions, a multiple classification neural network has d outputs in the final layer and constructs a classification by taking the maximum over these outputs.

3.1 Neural network function class with binary and linear activations

We first consider $\Pi_{p,S}^{\text{binary}}$, a neural network function class with a fixed structure S of p parameters, where all activation functions are either binary or linear, which generalizes the setting in (Sontag et al., 1998).

The fixed structure S consists of L layers, where the ℓ-th layer has n_ℓ nodes, $\ell \in \{1, \ldots, L\}$. We denote the j-th node in layer ℓ as $\texttt{Node}_{\ell,j}$, and denote $\mathcal{N}_{\ell,j}$ as the set of nodes in layer $\ell - 1$ that are connected to $\texttt{Node}_{\ell,j}$, whose size is $m_{\ell,j} = |\mathcal{N}_{\ell,j}|$. There is one real-valued paremeter for each pair of connected nodes in adjacent layers. We define the set of parameters of the network as $w = \{w_{\ell,j,s}\}_{1 \le s \le m_{j,\ell}, 1 \le \ell < L}$ (excluding those weights for the last output layer), which we assume is of a size smaller than p.

Each element in $\Pi_{p,S}^{\text{binary}}$ is a feed-forward neural network; given any input $x \in \mathbb{R}^m$, it outputs $f(x; w)$ as follows. The input layer takes $x \in \mathbb{R}^m$ from m input nodes, each for one feature. In each hidden layer $\ell \in \{1, \ldots, L-1\}$, $\texttt{Node}_{\ell,j}$ performs a linear combination to the outputs from each node in $\mathcal{N}_{\ell,j}$ in the previous layer, using m_j parameters $w_{\ell,j,1}, \ldots, w_{j,m_j} \in \mathbb{R}$. To be specific, the output of $\texttt{Node}_{\ell,j}$ is

$$f_j^{(\ell)}(x) = \sigma\Big(\sum_{s \in \mathcal{N}_{\ell,j}} w_{\ell,j,s} \cdot f_s^{(\ell-1)}(x) \Big),$$

where $\sigma(\cdot)$ is either the binary activation $\sigma(z) = \mathbf{1}\{z > 0\}$ or the linear activation $\sigma(z) = z$ for $z \in \mathbb{R}$. The last layer has d nodes, and is fully-connected to the last hidden layer with an identity activation. The output of the neural network is

$$f(x; w) = \underset{1 \le k \le d}{\text{argmax}} \left\{ \sum_{s=1}^{n_{L-1}} w_{L,k,s} \cdot f_s^{(L-1)}(x) \right\}.$$

The following theorem provides upper bounds on Natarajan dimensions of such function classes.

Theorem 3.1. *The Natarajan dimension of $\Pi_{p,S}^{binary}$ described above is upper bounded by $\mathcal{O}(d \cdot p^2)$.*

Several seminal early works have established the VC dimension of neural networks. The textbook result in Shalev-Shwartz and Ben-David (2014) provides an upper bound of $\mathcal{O}(p \log p)$ for neural networks with p parameters and all binary activation functions $\sigma(z) = \mathbf{1}\{z > 0\}$. Sontag et al. (1998) provides an upper bound of $\mathcal{O}(p^2)$ for neural networks with p parameters and binary or linear activation functions as considered in this subsection. By comparing this two results, we see that the allowing for linear activation functions adds a factor of p to the VC dimension (from the rate of $p \log p$ to p^2). Also, moving from binary to d-class classification adds a multiplicative factor of d to the upper bound when comparing Theorem 3.1 with Sontag et al. (1998). This is because the d-output neural network structure leads to a power-d factor in the growth function using our current proof technique. It will be interesting to see whether such dependence on d can be further sharpened.

3.2 Neural network function class with ReLU activations

We now consider $\Pi^{\mathrm{ReLU}}_{p,S}$, the class of multiple classification neural networks with a given structure S, which contains at most p parameters in intermediate layers and d final outputs, and the activation functions are either binary, linear or ReLU, i.e., $\sigma(z) = z$ or $\sigma(z) = \mathbf{1}\{z > 0\}$ or $\sigma(z) = z\mathbf{1}\{z > 0\}$. The definition of structure S is the same as in the preceding subsection; the only difference is that the activation functions for internal nodes can now be more general.

The Natarajan dimension of this function class is upper bounded as in the following theorem.

Theorem 3.2. *The Natarajan dimension of $\Pi^{ReLU}_{p,S}$ described above is upper bounded by $\mathcal{O}(d \cdot p^2)$.*

Our bound in Theorem 3.2 is of the same order as that in Theorem 3.2. Intuitively, this is because the ReLU activation can be viewed as a combination of binary and linear activation, which does not significantly increase the growth function compared with networks with the latter two activation functions.

Our theoretical analysis for the Natarajan dimensions of neural network function class is largely inspired by the framework of Sontag et al. (1998), which expresses neural network outputs as depending on linear combinations of binary values and original features, where the linear cofficients are further polynomials of the parameters of a bounded degree.

Remark 3.3. *The price we pay for the number of classes d in neural networks is a multiplicative factor of d, which is higher than the $\log(d)$ factor for decision trees and random forests. Although we only provide an upper-bound analysis, such comparison potentially indicates that the complexity of multi-class classification with neural networks might be higher than that of tree-based function classes.*

4 Technical proofs

In this section, we provide the proofs for all results in this work.

4.1 Proof of Theorem 2.2

Given a set of inputs $\{x_1, \ldots, x_n\} \in \mathbb{R}^p$, we first bound the number of configurations of the output on $\{x_1, \ldots, x_n\}$ by a decision tree of depth L. Denote the growth function on $x_1, \ldots, x_n$ as

$$g(\Pi^{\mathrm{dtree}}_{L,d}, n \,|\, x_1, \ldots, x_n) = \Big|\big\{\big(f(x_1), f(x_2), \ldots, f(x_n)\big) : f \in \Pi^{\mathrm{dtree}}_{L,d}\big\}\Big|.$$

We sort and denote internal nodes as $v \in \{1, \ldots, V\}$, where $V = 2^{L-1} - 1$ is the total number of internal nodes. We denote the corresponding feature as i_v and the threshold as θ_v, which vary with the tree $f \in \Pi^{\mathrm{dtree}}_{L,d}$. We index the set of leaf nodes with $\{1, \ldots, 2^{L-1}\}$, and each leaf node $1 \le l \le 2^{L-1}$ represents a class $k_l \in \{1, \ldots, d\}$. In this way, given any input $x_i \in \mathbb{R}$, the output from a tree $f \in \Pi^{\mathrm{dtree}}_{L,d}$ is determined by the vector of queries $(\mathbf{1}\{x_{i,i_v} \le \theta_n\})_{v=1}^{V} \in \mathbb{R}^V$ at all the V internal nodes, as well as $\{k_l : 1 \le l \le 2^{L-1}\}$, the assignment of classes to all leaf nodes. To better represent the classification model, for a decision tree $f \in \Pi^{\mathrm{dtree}}_{L,d}$, we let $I_{f,v}(x) = \mathbf{1}\{x_{i,i_v} \le \theta_v\}$ be the query function at node v, $I_f(x) = (\mathbf{1}\{x_{i,i_v} \le \theta_n\})_{v=1}^{V}$ be the vector of queries at internal nodes, and $L(f) = (k_l)_{l=1}^{2^{L-1}}$ be the vector of leaf node assignments.

Since $f(x)$ is fully decided by $I_f(x)$ and $L(f)$, we obtain an upper bound on the growth function that

$$\begin{aligned} g(\Pi^{\mathrm{dtree}}_{L,d}, n \,|\, x_1, \ldots, x_n) &\le \Big|\big\{\big(I_f(x_1), I_f(x_2), \ldots, I_f(x_n), L(f)\big) \in \{0,1\}^{n\times V} \times \{1, \ldots, d\}^{2^{L-1}} : f \in \Pi^{\mathrm{dtree}}_{L,d}\big\}\Big| \\ &\le \Big|\big\{\big(I_f(x_1), I_f(x_2), \ldots, I_f(x_n)\big) \in \{0,1\}^{n\times V} : f \in \Pi^{\mathrm{dtree}}_{L,d}\big\}\Big| \\ &\quad \times \Big|\big\{L(f) \in \{1, \ldots, d\}^{2^{L-1}} : f \in \Pi^{\mathrm{dtree}}_{L,d}\big\}\Big|. \end{aligned}$$

The second term is upper bounded by the cardinality of the image as

$$\Big|\big\{L(f) \in \{1, \ldots, d\}^{2^{L-1}} : f \in \Pi^{\mathrm{dtree}}_{L,d}\big\}\Big| \le d^{2^{L-1}}.$$

For the first term, the total number of different configurations of I_f is upper bounded by the product of those of all $I_{f,v}$, namely,

$$\begin{aligned}
&\left|\left\{\left(I_f(x_1), I_f(x_2), \ldots, I_f(x_n)\right) \in \{0,1\}^{n\times V} \colon f \in \Pi^{\text{dtree}}_{L,d}\right\}\right| \\
&\le \prod_{v=1}^{V} \left|\left\{\left(I_{f,v}(x_1), I_{f,v}(x_2), \ldots, I_{f,v}(x_n)\right) \in \{0,1\}^{n} \colon f \in \Pi^{\text{dtree}}_{L,d}\right\}\right| \\
&= \left|\left\{\left(I(x_1), I(x_2), \ldots, I(x_n)\right) \in \{0,1\}^{n} \colon I \in \mathcal{I}\right\}\right|^{V},
\end{aligned}$$

since each query function $I_{f,v}$ belongs to the function class

$$\mathcal{I} = \left\{I \colon \mathbb{R}^p \to \{0,1\} \colon I(x) = \mathbf{1}\{x_i \le \theta\},\ i \in \{1,\ldots,p\},\ \theta \in \mathbb{R}\right\}.$$

Moreover, for any function $I \in \mathcal{I}$ of the form $I(x) = \mathbf{1}\{x_s \le \theta\}$ for some feature $s \in \{1,\ldots,p\}$, there are at most $n+1$ different classifications on the n samples: one could sort all possible classification results into a sequence of size $n+1$ (perhaps with recurring members), such that the ℓ-th of result classifies the ℓ sample with smallest x_s as positive and others as negative. Hence

$$\left|\left\{\left(I(x_1), I(x_2), \ldots, I(x_n)\right) \in \{0,1\}^{n} \colon I \in \mathcal{I}\right\}\right| \le p(n+1).$$

Putting this together, we have $g(\Pi^{\text{dtree}}_{L,d}, n \,|\, x_1, \ldots, x_n) \le (p(n+1))^{2^{L-1}-1} \cdot d^{2^{L-1}}$.

Now suppose $\{x_1, \ldots, x_N\}$ are N-shattered by $\Pi^{\text{dtree}}_{L,d}$. By definition, policies in $\Pi^{\text{dtree}}_{L,d}$ have at least 2^N different configurations on $\{x_1, \ldots, x_N\}$, leading to

$$2^N \le (p(N+1))^{2^{L-1}-1} \cdot d^{2^{L-1}}.$$

Taking logarithm of both sides yields $N \log 2 \le 2^L \log(pd) + 2^L \log N$, which further gives $N = \mathcal{O}(L2^L \log(pd))$. This proves the upper bound of Natarajan dimension of $\Pi^{\text{dtree}}_{L,d}$. □

4.2 Proof of Theorem 2.3

Suppose $\{x_1, \ldots, x_N\}$ is N-shattered by $\Pi^{\text{forest}}_{L,T,d}$. Since the output of a policy $f \in \Pi^{\text{forest}}_{L,T,d}$ is fully decided by the outputs of T decision trees $f_1, \ldots, f_T \in \Pi^{\text{dtree}}_{T,d}$, the number of different configurations on $\{x_1, \ldots, x_N\}$ by policies in $\Pi^{\text{forest}}_{L,T,d}$ is upper bounded by $g(\Pi^{\text{dtree}}_{L,d}, N \,|\, x_1, \ldots, x_N)^T$. Hence,

$$2^N \le g(\Pi^{\text{dtree}}_{L,d}, N)^T \le (p(N+1))^{T(2^{L-1}-1)} \cdot d^{T\cdot 2^{L-1}}.$$

This implies $N \le \mathcal{O}(LT2^L \log(pdT))$, which completes the proof of Theorem 2.2. □

4.3 Proof of Theorem 3.1

Firstly, we fix any $f \in \Pi_{p,S}^{\text{binary}}$. Note that the final output $f(x)$ is fully determined by the $d(d-1)/2$ binary functions $\{b_{k,k'}(\cdot)\colon 1 \le k < k; \le d\}$, where

$$b_{k,k'}(x) = \mathbf{1}\left\{ \sum_{s=1}^{n_{L-1}} \theta_{L,k,s} \cdot f_s^{(L-1)}(x) - \sum_{s=1}^{n_{L-1}} \theta_{L,k',s} \cdot f_s^{(L-1)}(x) > 0 \right\}.$$

For simplicity, we assume there are no ties among the k outputs; otherwise one could break the ties without changing our results.

We now proceed to analyze the outputs $f_j^{(\ell)}$ in the hidden layers for $\ell \le L-1$. Similar to the idea in Sontag et al. (1998), we will show that they can all be expressed as linear combinations of binary values and $x_1, \ldots, x_m$. To begin with, we note that the activation functions $\sigma(\cdot)$ are either binary $\sigma(z) = \mathbb{1}\{z > 0\}$ or linear $\sigma(z) = z$, hence the outputs at layer $\ell = 1$ are either binary-valued, or some linear function of x. That is, we either have $f_j^{(1)}(x) \in \{0, 1\}$, or $f_j^{(1)}(x) = x^\top \alpha_j^{(1)}$ for some $\alpha_j^{(1)} \in \mathbb{R}^m$. We let $f_s^{(\ell)}(x) = \sigma(g_s^{(\ell)}(x))$, i.e., we denote $g_s^{(\ell)}(x)$ as the intermediate output before applying the activation function for node s at layer j. By definition, $g_s^{(2)}(x)$ is some linear combination of $\{f_j^{(1)}(x)\}$, the node outputs at layer 1. Therefore, for any s, $g_s^{(2)}(x)$ is a linear combination of some binary values and some linear function of x. Letting $I(z) = \mathbf{1}\{z > 0\}$. We can write

$$\begin{aligned}
g_s^{(2)}(x) &= \sum_{j \in N_{2,s}, \sigma(z) = \mathbf{1}\{z>0\}} w_{2,s,j} I(x^\top \alpha_j^{(1)}) + \sum_{j \in N_{2,s}, \sigma(z)=z} w_{2,s,j} x^\top \alpha_j^{(1)} \\
&= \sum_{j \in N_{2,s}, \sigma(z) = \mathbf{1}\{z>0\}} w_{2,s,j} I(\theta_{1,j,1} x_1 + \theta_{1,j,1} x_2 + \cdots + \theta_{1,j,m} x_m) \\
&\qquad + w_{2,s,j} \left(\theta_{1,j,1} x_1 + \theta_{1,j,2} x_2 + \cdots + \theta_{1,j,m} x_m\right) \\
&:= \sum_{j \in N_{2,s}, \sigma(z) = \mathbf{1}\{z>0\}} \theta_{2,s,j} I(\theta_{1,j,1} x_1 + \theta_{1,j,1} x_2 + \cdots + \theta_{1,j,m} x_m) \\
&\qquad + \theta_{2,s,n_{2,j}-m+1} x_1 + \cdots \theta_{2,s,n_{2,s}} x_m
\end{aligned}$$

for some $\theta_{1,j,1}, \ldots, \theta_{1,j,m}$ and $\theta_{2,s,1}, \ldots, \theta_{2,s,n_{2,s}} \in \mathbb{R}$ that can be derived from the network structure and $\{w_{j,\ell,s}\}$, where $n_{2,j}$ are the total number of $\theta_{2,s,\cdot}$ needed in such expression. Following this rule, we can write

$$\begin{aligned}
g_s^{(3)}(x) &= w_{3,s,1} I\Big(\underbrace{\theta_{2,1,1} I(\cdots) + \theta_{2,1,2} I(\cdots) + \cdots + \theta_{2,1,n_{2,1}-m+1} x_1 + \cdots + \theta_{2,1,n_{2,1}} x_m}_{g_1^{(2)}(x) \text{ supposing the activation function at this node is } \sigma(z)=\mathbf{1}\{z>0\}} \Big) \\
&\quad + \underbrace{w_{3,s,2} I(\cdots)}_{\text{other } g_j^{(2)} \text{ with activation function } \sigma(z)=\mathbf{1}\{z>0\}} + \cdots + \underbrace{\cdots}_{\text{other } g_j^{(2)} \text{ with activation function } \sigma(z)=z} \\
&\quad + w_{3,s,m_{3,s}} \Big(\underbrace{\theta_{2,1,1} I(\cdots) + \theta_{2,1,2} I(\cdots) + \cdots + \theta_{2,1,n_{2,1}-m+1} x_1 + \cdots + \theta_{2,1,n_{2,1}} x_m}_{g_j^{(2)}(x) \text{ supposing the activation function at this node is } \sigma(z)=z} \Big) \\
&= \theta_{3,s,1} I(\cdots) + \theta_{3,s,2} I(\cdots) + \cdots + \theta_{3,s,n_{3,s}-m+1} x_1 + \cdots + \theta_{3,s,n_{3,s}} x_m
\end{aligned}$$

for some $\theta_{3,s,1}, \ldots, \theta_{3,s,n_{3,s}} \in \mathbb{R}$ and $n_{3,s}$ is the number of such coefficients. In the last expression, each $I(\cdots)$ represents a node with binary activation, and inside the argument is another linear combination of binary values and $x_1, \ldots, x_m$. So on and so forth, we have

$$\begin{aligned}
g_s^{(L-1)}(x) = \theta_{L-1,s,1} I\big(&\theta_{L-2,1,1} I(\cdots) + \theta_{L-2,1,2} I(\cdots) + \cdots \\
&+ \theta_{L-2,1,n_{L-1,1}-m+1} x_1 + \cdots + \theta_{L-2,1,n_{L-1,1}} x_m\big) + \theta_{L-1,s,2} I(\cdots) + \cdots \\
&+ \theta_{L-1,s,n_{L,s}-m+1} x_1 + \cdots + \theta_{L-1,s,n_{L,s}} x_m
\end{aligned}$$

for some $\theta_{3,s,1},\dots,\theta_{3,s,n_{3,s}} \in \mathbb{R}$ and $n_{3,s}$ is the number of such coefficients. In the last expression, each $I(\cdots)$ represents a node with binary activation, and inside the argument is another linear combination of binary values and $x_1,\dots,x_m$. So on and so forth, we have

$$\begin{aligned} g_s^{(L-1)}(x) = \theta_{L-1,s,1} I\big(&\theta_{L-2,1,1} I(\cdots) + \theta_{L-2,1,2} I(\cdots) + \cdots \\ &+ \theta_{L-2,1,n_{L-1,1}-m+1} x_1 + \cdots + \theta_{L-2,1,n_{L-1,1}} x_m\big) + \theta_{L-1,s,2} I(\cdots) + \cdots \\ &+ \theta_{L-1,s,n_{L,s}-m+1} x_1 + \cdots + \theta_{L-1,s,n_{L,s}} x_m \end{aligned}$$

for some $\theta_{L-1,s,1},\dots,\theta_{L-1,s,n_{L-1,s}} \in \mathbb{R}$. Put another way, the outputs $g_s^{(L-1)}(x)$ can be written as the linear combination of outputs of some binary functions and original features, and the arguments to these binary functions are again some linear combination of binary functions (computed from preceding layers) and the original features, and so on. Also, the coefficients $\theta_{\ell,j,s}$ are all polynomials of $x_1,\dots,x_m$ and $w_1,\dots,w_p$, whose degrees are no larger than p since there are at most p layers. Similarly, the function $b_{k,k'}(x)$ can be written as a binary function $b_{k,k'}(x) = I(c_{k,k'}(x))$, where $c_{k,k'}(x)$ is a linear combination of several binary functions decided by previous layers and the original features, for which the linear coefficients are polynomials of $x_1,\dots,x_m$, the original parameters $w_1,\dots,w_p$ and $\{w_{L,k,s}\}_{1\le k\le d, 1\le s\le n_{L-1}}$ of degree $\le p+1$.

Assume there are p_1 binary nodes in intermediate layers. Then those binary functions have at most 2^{p_1} configurations. Therefore, for any input $x, x' \in \mathbb{R}^m$ and parameters $w, w' \in \mathbb{R}^p$, we will have $f(x; w) = f(x'; w')$ if the set of binary functions

$$\begin{aligned} \mathcal{I} := \Big\{ &I(\theta_{\ell,j,1} b_1 + \cdots \theta_{\ell,j,p_1} b_{p_1} + \theta_{\ell,j,n_{\ell,j}-m+1} x_1 + \cdots + \theta_{\ell,j,n_{\ell,j}} x_m) : \\ &(b_1,\dots,b_{p_1}) \in \{0,1\}^{p_1}, 1 \le \ell \le L, 1 \le j \le n_\ell \Big\} \end{aligned}$$

take the same value, where $\theta_{\ell,j,s}$ are fixed polynomials of the parameters $w_1,\dots,w_p$, features $x_1,\dots,x_m$ and $\{w_{L,k,s}\}_{1\le k\le d, 1\le s\le n_{L-1}}$ of degree $\le p+1$. Note that there are at most $(d^2+p)2^p$ functions in $\mathcal{I}$, because for each configuration of $(b_1,\dots,b_{p_1})$, there are at most p binary nodes in the intermediate layers, and no more than d^2 comparisons among the d classes in the last layer. In other words,

$$\mathcal{I} = \Big\{ I\big(P_r(x, w_1,\dots,w_p, \{w_{L,k,s}\}_{1\le k\le d, 1\le s\le n_{L-1}})\big) : r = 1,\dots,R \Big\},$$

where $R \le (p+d^2)2^p$ and each $P_r(\cdot)$ is a polynomial with degree $\le p+1$. If we view the input x as fixed, the above set of functions is

$$\mathcal{I}(x) := \Big\{ I\big(P_r(x, w_1,\dots,w_p, \{w_{L,k,s}\}_{1\le k\le d, 1\le s\le n_{L-1}})\big) : r = 1,\dots,R \Big\},$$

and each $P_r(\cdot)$ is a polynomial of degree $\le p+1$ of no greater than $p(1+d)$ variables

$$\big\{ w_1,\dots,w_p, \{w_{L,k,s}\}_{1\le k\le d, 1\le s\le n_{L-1}} \big\},$$

since the parameters for the last layer is at most $p \cdot d$.

Let $x_1,\dots,x_N$ be N-shattered by the function class $\Pi_{p,S}^{\text{binary}}$, and let $w^{(1)},\dots,w^{(M)}$, $M = 2^N$ be the parameters that witness the shattering. Consider the classifications

$$\begin{pmatrix} f(x_1; w^{(1)}), & f(x_2; w^{(1)}), & \cdots, & f(x_N; w^{(1)}) \\ f(x_1; w^{(2)}), & f(x_2; w^{(2)}), & \cdots, & f(x_N; w^{(2)}) \\ \vdots & \vdots & & \vdots \\ f(x_1; w^{(M)}), & f(x_2; w^{(M)}), & \cdots, & f(x_N; w^{(M)}) \end{pmatrix}.$$

By the definition of N-shattering, every two rows in the above matrix are distinct. Thus for each $j \ne j'$, there exists some $i \in \{1,\dots,N\}$ such that $f(x_i; w^{(j)}) \ne f(x_i; w^{(j')})$. By above arguments, there exists some binary function in $\mathcal{I}$ that takes different values on $(x_i, w^{(j)})$ and $(x_i, w^{(j')})$. Then there must exist some $1 \le r \le R$ such that the signs of $P_r(x_i, w^{(j)})$ and $P_r(x_i, w^{(j')})$ are different. In other words, each $j \in \{1,\dots,M\}$ gives a unique configuration of the signs of the $N \cdot R$ polynomials in $\mathcal{I}(x_1),\dots,\mathcal{I}(x_N)$. The following lemma establishes an upper bound for the number of such configurations, which is a re-statement of Corollary 2.1 in Goldberg and Jerrum (1995).

Lemma 4.1. *Let $\{P_1, \ldots, P_{\widetilde{R}}\}$ be $\widetilde{R}$ polynomials of degree at most $\widetilde{p}$ in $\widetilde{n}$ real variables with $\widetilde{R} \geq \widetilde{n}$, then the number of different configurations of signs to the $\{P_1, \ldots, P_{\widetilde{R}}\}$ is at most $(8e\widetilde{p}\widetilde{R}/\widetilde{n})^{\widetilde{n}}$.*

Utilizing Lemma 4.1 with $\widetilde{R} = NR$, $\widetilde{p} = p + 1$ and $\widetilde{n} = p(1 + d)$, the number of different configurations on the $N \cdot R$ polynomials is upper bounded as

$$M = 2^N \leq \Big(\frac{8e(p+1) \cdot N(p+d^2) \cdot 2^p}{p(1+d)}\Big)^{p(1+d)}.$$

Taking logarithm we have

$$N \leq p(1+k) \log\big(8e \cdot (p+1)Nd \cdot 2^p\big),$$

hence $N \leq \mathcal{O}(dp^2)$, which completes the proof of Theorem 3.1. □

4.4 Proof of Theorem 3.2

The proof of Theorem 3.2 is similar to that of Theorem 3.1, except that we consider the configuration on signs of a slightly different set of polynomial functions. The formal definition of the structure is the same as the previous case, except that some of the activation functions are $\sigma(z) = z\mathbf{1}\{z > 0\}$. We use the same notations for nodes and weights as in the previous case.

For $\texttt{Node}_{j,\ell}$, we suppose its output is $f_j^{(\ell)}(x) = \sigma\big(g_j^{(\ell)}(x)\big)$, where $g_j^{(\ell)}(x)$ is the quantity before applying the activation function, i.e., the linear combination of the outputs of nodes in the preceding layer. If $\texttt{Node}_{j,\ell}$ has binary activation, then it appears in the formula for nodes in latter layers as 0 or 1 (see arguments in the proof of Theorem 3.1). If $\texttt{Node}_{j,\ell}$ has ReLU activation, then it appears as either 0 or $g_j^{(\ell)}(x)$, the linear combination itself. Recall that $I(z) = \mathbf{1}\{z > 0\}$. For any inputs $x, x' \in \mathbb{R}^m$ and parameters $w, w' \in \mathbb{R}^{p(1+d)}$ including parameters in the last layer, we have $f(x; w) = f(x'; w')$ if the set of binary functions

$$\mathcal{I} = \Big\{ I(\theta_{\ell,j,1}b_1 + \cdots \theta_{\ell,j,p_1}b_{p_1} + \theta_{\ell,j,n_{\ell,j}-m+1}x_1 + \cdots + \theta_{\ell,j,n_{\ell,j}}x_m) : \\ (b_1, \ldots, b_{p_1}) \in \{0,1\}^{p_1}, 1 \leq \ell \leq L, 1 \leq j \leq n_\ell \Big\}$$

all have the same sign on (x, w) and (x', w'). Here p_1 is the total number of binary and ReLU nodes, so that $p_1 \leq p$. Also, each $\theta_{\ell,j,s}$ is a polynomial in all entries of x, w of degree $\leq p + 1$. Viewing them as functions of parameters, the set of functions is

$$\mathcal{I}(x) := \Big\{ I\Big(P_r\big(x, w_1, \ldots, w_p, \{w_{L,k,s}\}_{1 \leq k \leq d, 1 \leq s \leq n_{L-1}}\big)\Big) : r = 1, \ldots, R \Big\},$$

Let $x_1, \ldots, x_N$ be N-shattered by the function class Π_p^{ReLU}, and let $w^{(1)}, \ldots, w^{(M)}$, $M = 2^N$ be the parameters that witness the shattering. Consider the classifications

$$\begin{pmatrix} f(x_1; w^{(1)}), & f(x_2; w^{(1)}), & \cdots, & f(x_N; w^{(1)}) \\ f(x_1; w^{(2)}), & f(x_2; w^{(2)}), & \cdots, & f(x_N; w^{(2)}) \\ \vdots & \vdots & & \vdots \\ f(x_1; w^{(M)}), & f(x_2; w^{(M)}), & \cdots, & f(x_N; w^{(M)}) \end{pmatrix}.$$

By the definition of N-shattering, every two rows in the above matrix are different. Thus for each $j \neq j'$, there exists some $i \in \{1, \ldots, N\}$ such that $f(x_i; w^{(j)}) \neq f(x_i; w^{(j')})$. By above arguments, there exists some binary function in $\mathcal{I}$ that takes different values on $(x_i, w^{(j)})$ and $(x_i, w^{(j')})$. Then there exists some $1 \leq r \leq R$ such that the signs of $P_r(x_i, w^{(j)})$ and $P_r(x_i, w^{(j')})$ are different. In other words, each $j \in \{1, \ldots, M\}$ gives a unique configuration of the signs of the $N \cdot R$ polynomials in $\mathcal{I}(x_1), \ldots, \mathcal{I}(x_N)$. Utilizing Lemma 4.1 again, we obtain the desired result. □

Algorithm-Independent machine Learning:

Mathematical Foundations that do not depend upon any particular classifier or learning algorithm used

Techniques used in conjunction with different learning algorithms or provide guidelines in their use – E.g., cross-validation and resampling

Combining Classifiers:

Component classifiers with discriminant functions

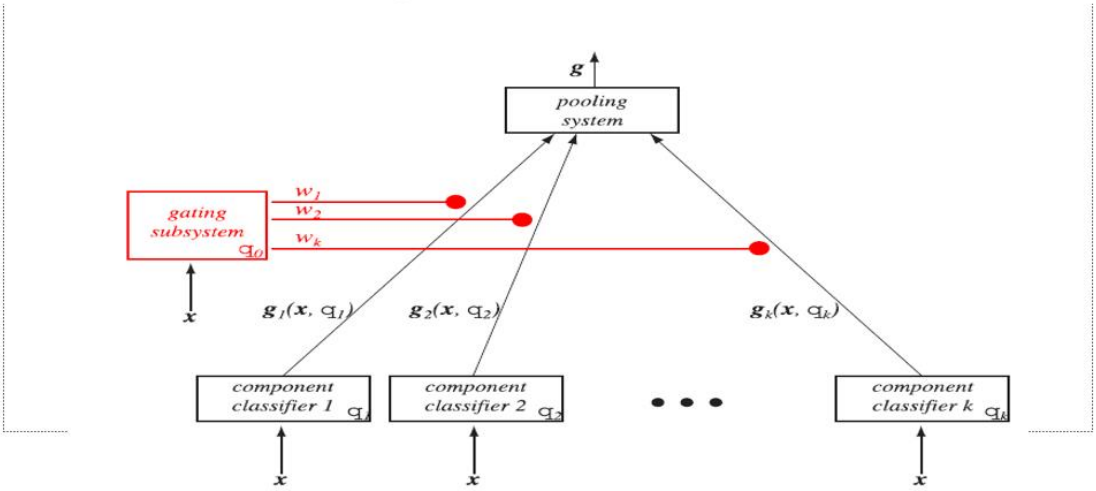
We assume that each pattern is produced by a *mixture model*, in which first some fundamental process or function indexed by r (where $1 \leq r \leq k$) is randomly chosen according to distribution $P(r|\mathbf{x}, \boldsymbol{\theta}_0^0)$ where $\boldsymbol{\theta}_0^0$ is a parameter vector. Next, the selected process r emits an output $\mathbf{y}$ (e.g., a category label) according to $P(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta}_r^0)$, where the parameter vector $\boldsymbol{\theta}_r^0$ describes the state of the process. (The superscript 0 indicates the properties of the generating model. Below, terms without this superscript refer to the parameters in a classifier.) The overall probability of producing output $\mathbf{y}$ is then the sum over all the processes according to:

$$P(\mathbf{y}|\mathbf{x}, \boldsymbol{\Theta}^0) = \sum_{r=1}^{k} P(r|\mathbf{x}, \boldsymbol{\eta}^0) P(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta}^0), \tag{54}$$

where $\boldsymbol{\Theta}^0 = [\boldsymbol{\theta}_0^0, \boldsymbol{\theta}_1^0, \ldots, \boldsymbol{\theta}_k^0]^t$ represents the vector of all relevant parameters. Equation 54 describes a *mixture density*, which could be discrete or continuous (Chap. ??).

Figure 9.19 shows the basic architecture of an ensemble classifier whose task is to classify a test pattern $\mathbf{x}$ into one of c categories; this architecture matches the assumed mixture model. A test pattern $\mathbf{x}$ is presented to each of the k component classifiers, each of which emits c scalar discriminant values, one for each category. The c discriminant values from component classifier r are grouped and marked $\mathbf{g}(\mathbf{x}, \boldsymbol{\theta}_r)$ in the figure, with

$$\sum_{j=1}^{c} g_{rj} = 1 \text{ for all } r. \tag{55}$$



All discriminant values from component classifier r are multiplied by a scalar weight w_r, governed by the *gating subsystem*, which has a parameter vector $\boldsymbol{\theta}_0$. Below we shall use the conditional mean of the mixture density, which can be calculated from Eq. 54

$$\boldsymbol{\mu} = \mathcal{E}[\mathbf{y}|\mathbf{x}, \boldsymbol{\Theta}] = \sum_{r=1}^{k} w_r \boldsymbol{\mu}_r \tag{56}$$

where $\boldsymbol{\mu}_r$ is the conditional mean associated with $P(\mathbf{y}|\mathbf{x}, \boldsymbol{\theta}_r^0)$.

The mixture-of-experts architecture is trained so that each component classifier models a corresponding process in the mixture model, and the gating subsystem models the mixing parameters $P(r|\mathbf{x}, \boldsymbol{\theta}_0^0)$ in Eq. 54. The goal is to find parameters that maximize the log-likelihood for n training patterns $\mathbf{x}^1, \ldots \mathbf{x}^n$ in set $\mathcal{D}$:

$$l(\mathcal{D}, \boldsymbol{\Theta}) = \sum_{i=1}^{n} ln \left(\sum_{r=1}^{k} P(r|\mathbf{x}^i, \boldsymbol{\theta}_0) P(\mathbf{y}^i|\mathbf{x}^i, \boldsymbol{\theta}_r) \right). \tag{57}$$

A straightforward approach is to use gradient descent on the parameters, where the derivatives are (Problem 43)

$$\frac{\partial l(\mathcal{D}, \boldsymbol{\Theta})}{\partial \boldsymbol{\mu}_r} = \sum_{i=1}^{n} P(r|\mathbf{y}^i, \mathbf{x}^i) \frac{\partial}{\partial \boldsymbol{\mu}_r} ln[P(\mathbf{y}^i|\mathbf{x}^i, \boldsymbol{\theta}_r)] \text{ for } r = 1, \ldots k \tag{58}$$

and

$$\frac{\partial l(\mathcal{D}, \boldsymbol{\Theta})}{\partial g_r} = \sum_{i=1}^{n} (P(r|\mathbf{y}^i, \mathbf{x}^i) - w_r^i). \tag{59}$$

Here $(P(r|\mathbf{y}^i, \mathbf{x}^i)$ is the posterior probability of process r conditional on the input and output being $\mathbf{x}^i$ and $\mathbf{y}^i$, respectively. Moreover, w_r^i is the prior probability $P(r|\mathbf{x}^i)$ that process r is chosen given the input is $\mathbf{x}^i$. Gradient descent according to Eq. 59 moves the prior probabilities to the posterior probabilities. The Expectation-Maximization (EM) algorithm can be used to train this architecture as well (Chap. **??**).

The final decision rule is simply to choose the category corresponding to the maximum discriminant value after the pooling system. An alternative, *winner-take-all* method is to use the decision of the single component classifier that is "most confident," i.e., has the largest single discriminant value g_{rj}. While the winner-take-all method is provably sub-optimal, it nevertheless is simple and can work well if the component classifiers are experts in separate regions of the input space.

Component classifiers without discriminant functions

Occasionally we seek to form an ensemble classifier from highly trained component classifiers, some of which might not themselves compute discriminant functions. For instance, we might have four component classifiers — a k-nearest-neighbor classifier, a decision tree, a neural network, and a rule-based system — all addressing the same problem. While a neural network would provide analog values for each of the c categories, the rule-based system would give only a single category label (i.e., a one-of-c representation) and the k-nearest neighbor classifier would give only rank order of the categories.

In order to integrate the information from the component classifiers we must convert the their outputs into discriminant values obeying the constraint of Eq. 55 so we can use the framework of Fig. 9.19. The simplest heuristics to this end are the following:

Analog If the outputs of a component classifier are analog values $\tilde{g}_i$, we can use the *softmax* transformation,

$$g_i = \frac{e^{\tilde{g}_i}}{\sum_{j=1}^{c} e^{\tilde{g}_i}}. \tag{60}$$

to convert them to values g_i.

Rank order If the output is a rank order list, we assume the discriminant function is linearly proportional to the rank order of the item on the list. Of course, the resulting g_i should then be properly normalized, and thus sum to 1.0.

One-of-c If the output is a one-of-c representation, in which a single category is identified, we let $g_j = 1$ for the j corresponding to the chosen category, and 0 otherwise.

Analog value		Rank order		One-of-c	
$\tilde{g}_i$	g_i	$\tilde{g}_i$	g_i	$\tilde{g}_i$	g_i
0.4	0.158	3rd	$4/21 = 0.194$	0	0
0.6	0.193	6th	$1/21 = 0.048$	1	1.0
0.9	0.260	5th	$2/21 = 0.095$	0	0
0.3	0.143	1st	$6/21 = 0.286$	0	0
0.2	0.129	2nd	$5/21 = 0.238$	0	0
0.1	0.111	4th	$3/21 = 0.143$	0	0

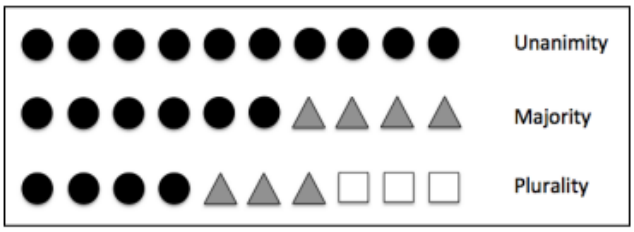
Once the outputs of the component classifiers have been converted to effective discriminant functions in this way, the component classifiers are themselves held fixed, but the gating network is trained as described in Eq. 59. This method is particularly useful when several highly trained component classifiers are pooled to form a single decision.

Majority Voting Classifier:

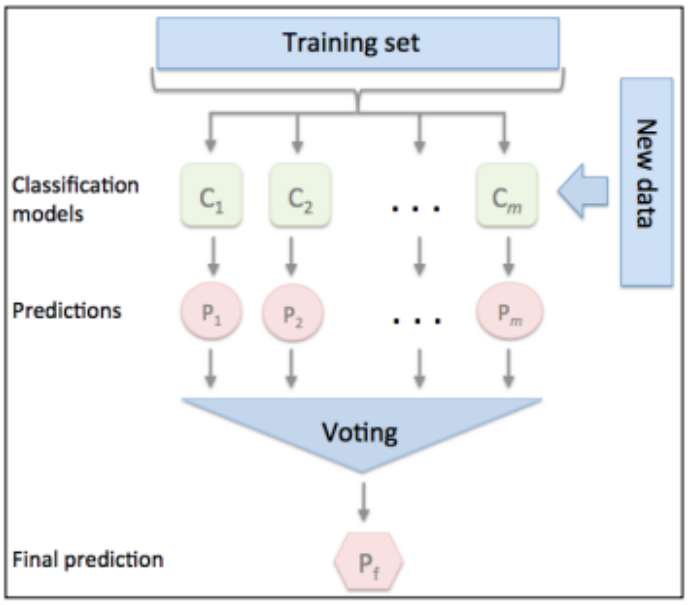
Learning with ensembles

The goal behind **ensemble methods** is to combine different classifiers into a meta-classifier that has a better generalization performance than each individual classifier alone. For example, assuming that we collected predictions from 10 experts, ensemble methods would allow us to strategically combine these predictions by the 10 experts to come up with a prediction that is more accurate and robust than the predictions by each individual expert. As we will see later in this chapter, there are several different approaches for creating an ensemble of classifiers. In this section, we will introduce a basic perception about how ensembles work and why they are typically recognized for yielding a good generalization performance.

In this chapter, we will focus on the most popular ensemble methods that use the **majority voting** principle. Majority voting simply means that we select the class label that has been predicted by the majority of classifiers, that is, received more than 50 percent of the votes. Strictly speaking, the term **majority vote** refers to binary class settings only. However, it is easy to generalize the majority voting principle to multi-class settings, which is called **plurality voting**. Here, we select the class label that received the most votes (mode). The following diagram illustrates the concept of majority and plurality voting for an ensemble of 10 classifiers where each unique symbol (triangle, square, and circle) represents a unique class label:



Using the training set, we start by training m different classifiers ($C_1,\ldots,C_m$). Depending on the technique, the ensemble can be built from different classification algorithms, for example, decision trees, support vector machines, logistic regression classifiers, and so on. Alternatively, we can also use the same base classification algorithm fitting different subsets of the training set. One prominent example of this approach would be the random forest algorithm, which combines different decision tree classifiers. The following diagram illustrates the concept of a general ensemble approach using majority voting:



To predict a class label via a simple majority or plurality voting, we combine the predicted class labels of each individual classifier C_j and select the class label $\hat{y}$ that received the most votes:

$$\hat{y} = mode\{C_1(\boldsymbol{x}), C_2(\boldsymbol{x}), \ldots, C_m(\boldsymbol{x})\}$$

For example, in a binary classification task where $class1 = -1$ and $class2 = +1$, we can write the majority vote prediction as follows:

$$C(\boldsymbol{x}) = sign\left[\sum_j^m C_j(\boldsymbol{x})\right] = \begin{cases} 1 \ if \sum_j C_j(\boldsymbol{x}) \geq 0 \\ -1 \ otherwise \end{cases}$$

To illustrate why ensemble methods can work better than individual classifiers alone, let's apply the simple concepts of combinatorics. For the following example, we make the assumption that all n base classifiers for a binary classification task have an equal error rate ε. Furthermore, we assume that the classifiers are independent and the error rates are not correlated. Under those assumptions, we can simply express the error probability of an ensemble of base classifiers as a probability mass function of a binomial distribution:

$$P(y \geq k) = \sum_k^n \binom{n}{k} \varepsilon^k (1-\varepsilon)^{n-k} = \varepsilon_{ensemble}$$

Here, $\binom{n}{k}$ is the binomial coefficient *n choose k*. In other words, we compute the probability that the prediction of the ensemble is wrong. Now let's take a look at a more concrete example of 11 base classifiers ($n = 11$) with an error rate of 0.25 ($\varepsilon = 0.25$):

$$P(y \geq k) = \sum_{k=6}^n \binom{11}{k} 0.25^k (1-\varepsilon)^{11-k} = 0.034$$

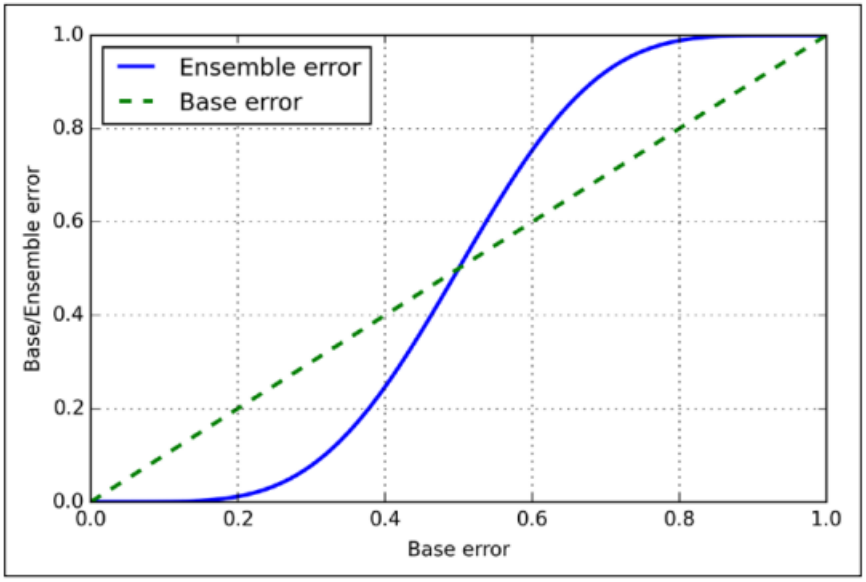
As we can see, the error rate of the ensemble (0.034) is much lower than the error rate of each individual classifier (0.25) if all the assumptions are met. Note that, in this simplified illustration, a 50-50 split by an even number of classifiers n is treated as an error, whereas this is only true half of the time. To compare such an idealistic ensemble classifier to a base classifier over a range of different base error rates, let's implement the probability mass function in Python:

```
>>> from scipy.misc import comb
>>> import math
>>> def ensemble_error(n_classifier, error):
...     k_start = math.ceil(n_classifier / 2.0)
...     probs = [comb(n_classifier, k) *
...              error**k *
...              (1-error)**(n_classifier - k)
...              for k in range(k_start, n_classifier + 1)]
...     return sum(probs)
>>> ensemble_error(n_classifier=11, error=0.25)
0.034327507019042969
```

After we've implemented the `ensemble_error` function, we can compute the ensemble error rates for a range of different base errors from 0.0 to 1.0 to visualize the relationship between ensemble and base errors in a line graph:

```
>>> import numpy as np
>>> error_range = np.arange(0.0, 1.01, 0.01)
>>> ens_errors = [ensemble_error(n_classifier=11, error=error)
...               for error in error_range]
>>> import matplotlib.pyplot as plt
>>> plt.plot(error_range, ens_errors,
...          label='Ensemble error',
...          linewidth=2)
>>> plt.plot(error_range, error_range,
...          linestyle='--', label='Base error',
...          linewidth=2)
>>> plt.xlabel('Base error')
>>> plt.ylabel('Base/Ensemble error')
>>> plt.legend(loc='upper left')
>>> plt.grid()
>>> plt.show()
```

As we can see in the resulting plot, the error probability of an ensemble is always better than the error of an individual base classifier as long as the base classifiers perform better than random guessing ($\varepsilon < 0.5$). Note that the y-axis depicts the base error (dotted line) as well as the ensemble error (continuous line):



Implementing a simple majority vote classifier

After the short introduction to ensemble learning in the previous section, let's start with a warm-up exercise and implement a simple ensemble classifier for majority voting in Python. Although the following algorithm also generalizes to multi-class settings via plurality voting, we will use the term *majority voting* for simplicity as is also often done in literature.

The algorithm that we are going to implement will allow us to combine different classification algorithms associated with individual weights for confidence. Our goal is to build a stronger meta-classifier that balances out the individual classifiers' weaknesses on a particular dataset. In more precise mathematical terms, we can write the weighted majority vote as follows:

$$\hat{y} = \arg\max_{i} \sum_{j=1}^{m} w_j \chi_A \left(C_j \left(\boldsymbol{x} \right) = i \right)$$

Here, w_j is a weight associated with a base classifier, C_j, $\hat{y}$ is the predicted class label of the ensemble, χ_A (Greek *chi*) is the characteristic function $[C_j(\boldsymbol{x}) = i \in A]$, and A is the set of unique class labels. For equal weights, we can simplify this equation and write it as follows:

$$\hat{y} = mode\{C_1(\boldsymbol{x}), C_2(\boldsymbol{x}), \ldots, C_m(\boldsymbol{x})\}$$

To better understand the concept of *weighting*, we will now take a look at a more concrete example. Let's assume that we have an ensemble of three base classifiers C_j ($j \in \{0,1\}$) and want to predict the class label of a given sample instance x. Two out of three base classifiers predict the class label 0, and one C_3 predicts that the sample belongs to class 1. If we weight the predictions of each base classifier equally, the majority vote will predict that the sample belongs to class 0:

$$C_1(x) \rightarrow 0,\ C_2(x) \rightarrow 0,\ C_3(x) \rightarrow 1$$

$$\hat{y} = mode\{0, 0, 1\} = 0$$

Now let's assign a weight of 0.6 to C_3 and weight C_1 and C_2 by a coefficient of 0.2, respectively.

$$\hat{y} = \arg\max_i \sum_{j=1}^{m} w_j \chi_A\left(C_j(\boldsymbol{x}) = i\right)$$

$$= \arg\max_i \left[0.2 \times i_0 + 0.2 \times i_0 + 0.6 \times i_1\right] = 1$$

More intuitively, since $3 \times 0.2 = 0.6$, we can say that the prediction made by C_3 has three times more weight than the predictions by C_1 or C_2, respectively. We can write this as follows:

$$\hat{y} = mode\{0, 0, 1, 1, 1\} = 1$$

To translate the concept of the weighted majority vote into Python code, we can use NumPy's convenient `argmax` and `bincount` functions:

```
>>> import numpy as np
>>> np.argmax(np.bincount([0, 0, 1],
...           weights=[0.2, 0.2, 0.6]))
1
```

As discussed in *Chapter 3, A Tour of Machine Learning Classifiers Using Scikit-learn*, certain classifiers in scikit-learn can also return the probability of a predicted class label via the `predict_proba` method. Using the predicted class probabilities instead of the class labels for majority voting can be useful if the classifiers in our ensemble are well calibrated. The modified version of the majority vote for predicting class labels from probabilities can be written as follows:

$$\hat{y} = \arg\max_i \sum_{j=1}^{m} w_j p_{ij}$$

Here, p_{ij} is the predicted probability of the *jth* classifier for class label *i*.

To continue with our previous example, let's assume that we have a binary classification problem with class labels $i \in \{0,1\}$ and an ensemble of three classifiers C_j ($j \in \{1,2,3\}$). Let's assume that the classifier C_j returns the following class membership probabilities for a particular sample $\boldsymbol{x}$:

$$C_1(\boldsymbol{x}) \rightarrow [0.9, 0.1],\ C_2(\boldsymbol{x}) \rightarrow [0.8, 0.2],\ C_3(\boldsymbol{x}) \rightarrow [0.4, 0.6]$$

We can then calculate the individual class probabilities as follows:

$$p(i_0 \mid \boldsymbol{x}) = 0.2 \times 0.9 + 0.2 \times 0.8 + 0.6 \times 0.4 = 0.58$$

$$p(i_1 \mid \boldsymbol{x}) = 0.2 \times 0.1 + 0.2 \times 0.2 + 0.6 \times 0.06 = 0.42$$

$$\hat{y} = \arg\max_i \left[p(i_0 \mid \boldsymbol{x}), p(i_1 \mid \boldsymbol{x}) \right] = 0$$

To implement the weighted majority vote based on class probabilities, we can again make use of NumPy using `numpy.average` and `np.argmax`:

```
>>> ex = np.array([[0.9, 0.1],
...                [0.8, 0.2],
...                [0.4, 0.6]])
>>> p = np.average(ex, axis=0, weights=[0.2, 0.2, 0.6])
>>> p
array([ 0.58,  0.42])
>>> np.argmax(p)
0
```

Putting everything together, let's now implement a `MajorityVoteClassifier` in Python:

```
from sklearn.base import BaseEstimator
from sklearn.base import ClassifierMixin
from sklearn.preprocessing import LabelEncoder
from sklearn.externals import six
from sklearn.base import clone
from sklearn.pipeline import _name_estimators
import numpy as np
import operator


class MajorityVoteClassifier(BaseEstimator,
                             ClassifierMixin):
    """ A majority vote ensemble classifier

    Parameters
    ----------
    classifiers : array-like, shape = [n_classifiers]
      Different classifiers for the ensemble

    vote : str, {'classlabel', 'probability'}
      Default: 'classlabel'
      If 'classlabel' the prediction is based on
      the argmax of class labels. Else if
      'probability', the argmax of the sum of
      probabilities is used to predict the class label
      (recommended for calibrated classifiers).

    weights : array-like, shape = [n_classifiers]
      Optional, default: None
      If a list of `int` or `float` values are

  provided, the classifiers are weighted by
  importance; Uses uniform weights if `weights=None`.

"""
def __init__(self, classifiers,
             vote='classlabel', weights=None):

    self.classifiers = classifiers
    self.named_classifiers = {key: value for
                              key, value in
                              _name_estimators(classifiers)}
    self.vote = vote
    self.weights = weights
```

```
                                        _name_estimators(classifiers)}
        self.vote = vote
        self.weights = weights

    def fit(self, X, y):
        """ Fit classifiers.

        Parameters
        ----------
        X : {array-like, sparse matrix},
            shape = [n_samples, n_features]
            Matrix of training samples.

        y : array-like, shape = [n_samples]
            Vector of target class labels.

        Returns
        -------
        self : object

        """
        # Use LabelEncoder to ensure class labels start
        # with 0, which is important for np.argmax
        # call in self.predict
        self.lablenc_ = LabelEncoder()
        self.lablenc_.fit(y)
        self.classes_ = self.lablenc_.classes_
        self.classifiers_ = []
        for clf in self.classifiers:
            fitted_clf = clone(clf).fit(X,
                                        self.lablenc_.transform(y))
            self.classifiers_.append(fitted_clf)
        return self
```

I added a lot of comments to the code to better understand the individual parts. However, before we implement the remaining methods, let's take a quick break and discuss some of the code that may look confusing at first. We used the parent classes `BaseEstimator` and `ClassifierMixin` to get some base functionality *for free*, including the methods `get_params` and `set_params` to set and return the classifier's parameters as well as the `score` method to calculate the prediction accuracy, respectively. Also note that we imported `six` to make the `MajorityVoteClassifier` compatible with Python 2.7.

Next we will add the `predict` method to predict the class label via majority vote based on the class labels if we initialize a new `MajorityVoteClassifier` object with `vote='classlabel'`. Alternatively, we will be able to initialize the ensemble classifier with `vote='probability'` to predict the class label based on the class membership probabilities. Furthermore, we will also add a `predict_proba` method to return the average probabilities, which is useful to compute the **Receiver Operator Characteristic area under the curve** (**ROC AUC**).

```
def predict(self, X):
    """ Predict class labels for X.

    Parameters
    ----------
    X : {array-like, sparse matrix},
        Shape = [n_samples, n_features]
        Matrix of training samples.

    Returns
    ----------
    maj_vote : array-like, shape = [n_samples]
        Predicted class labels.

    """
    if self.vote == 'probability':
        maj_vote = np.argmax(self.predict_proba(X),
                             axis=1)
    else:  # 'classlabel' vote

        #  Collect results from clf.predict calls
        predictions = np.asarray([clf.predict(X)
                                  for clf in
                                  self.classifiers_]).T

        maj_vote = np.apply_along_axis(
                       lambda x:
                       np.argmax(np.bincount(x,
```

```
                                        weights=self.weights)),
                        axis=1,
                        arr=predictions)
        maj_vote = self.lablenc_.inverse_transform(maj_vote)
        return maj_vote

    def predict_proba(self, X):
        """ Predict class probabilities for X.

        Parameters
        ----------
        X : {array-like, sparse matrix},
            shape = [n_samples, n_features]
            Training vectors, where n_samples is
            the number of samples and
            n_features is the number of features.

        Returns
        ----------
        avg_proba : array-like,
            shape = [n_samples, n_classes]
            Weighted average probability for
            each class per sample.

        """
        probas = np.asarray([clf.predict_proba(X)
                             for clf in self.classifi
        avg_proba = np.average(probas,
                               axis=0, weights=self.w
        return avg_proba

    def get_params(self, deep=True):
        """ Get classifier parameter names for GridSe
        if not deep:
            return super(MajorityVoteClassifier,
                         self).get_params(deep=False)
        else:
            out = self.named_classifiers.copy()
            for name, step in\
                    six.iteritems(self.named_classifi
                for key, value in six.iteritems(
                        step.get_params(deep=True)):
                    out['%s__%s' % (name, key)] = val
            return out
```

Also, note that we defined our own modified version of the `get_params` methods to use the `_name_estimators` function in order to access the parameters of individual classifiers in the ensemble. This may look a little bit complicated at first, but it will make perfect sense when we use grid search for hyperparameter-tuning in later sections.

Although our `MajorityVoteClassifier` implementation is very useful for demonstration purposes, I also implemented a more sophisticated version of the majority vote classifier in scikit-learn. It will become available as `sklearn.ensemble.VotingClassifier` in the next release version (v0.17).

Combining different algorithms for classification with majority vote

Now it is about time to put the `MajorityVoteClassifier` that we implemented in the previous section into action. But first, let's prepare a dataset that we can test it on. Since we are already familiar with techniques to load datasets from CSV files, we will take a shortcut and load the **Iris** dataset from scikit-learn's dataset module. Furthermore, we will only select two features, **sepal width** and **petal length**, to make the classification task more challenging. Although our `MajorityVoteClassifier` generalizes to multiclass problems, we will only classify flower samples from the two classes, **Iris-Versicolor** and **Iris-Virginica**, to compute the **ROC AUC**. The code is as follows:

```
>>> from sklearn import datasets
>>> from sklearn.cross_validation import train_test_split
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.preprocessing import LabelEncoder
>>> iris = datasets.load_iris()
>>> X, y = iris.data[50:, [1, 2]], iris.target[50:]
>>> le = LabelEncoder()
>>> y = le.fit_transform(y)
```

Note that scikit-learn uses the `predict_proba` method (if applicable) to compute the ROC AUC score. In *Chapter 3, A Tour of Machine Learning Classifiers Using Scikit-learn*, we saw how the class probabilities are computed in logistic regression models. In decision trees, the probabilities are calculated from a frequency vector that is created for each node at training time. The vector collects the frequency values of each class label computed from the class label distribution at that node. Then the frequencies are normalized so that they sum up to 1. Similarly, the class labels of the k-nearest neighbors are aggregated to return the normalized class label frequencies in the k-nearest neighbors algorithm. Although the normalized probabilities returned by both the decision tree and k-nearest neighbors classifier may look similar to the probabilities obtained from a logistic regression model, we have to be aware that these are actually not derived from probability mass functions.

Next we split the Iris samples into 50 percent training and 50 percent test data:

```
>>> X_train, X_test, y_train, y_test =\
...         train_test_split(X, y,
...                          test_size=0.5,
...                          random_state=1)
```

Using the training dataset, we now will train three different classifiers—a logistic regression classifier, a decision tree classifier, and a k-nearest neighbors classifier—and look at their individual performances via a 10-fold cross-validation on the training dataset before we combine them into an ensemble classifier:

```
>>> from sklearn.cross_validation import cross_val_score
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.tree import DecisionTreeClassifier
>>> from sklearn.neighbors import KNeighborsClassifier
>>> from sklearn.pipeline import Pipeline
>>> import numpy as np
>>> clf1 = LogisticRegression(penalty='l2',
...                           C=0.001,
...                           random_state=0)
>>> clf2 = DecisionTreeClassifier(max_depth=1,
...                               criterion='entropy',
...                               random_state=0)
>>> clf3 = KNeighborsClassifier(n_neighbors=1,
...                             p=2,
...                             metric='minkowski')
>>> pipe1 = Pipeline([['sc', StandardScaler()],
...                   ['clf', clf1]])

>>> pipe3 = Pipeline([['sc', StandardScaler()],
...                   ['clf', clf3]])
>>> clf_labels = ['Logistic Regression', 'Decision Tree', 'KNN']
>>> print('10-fold cross validation:\n')
>>> for clf, label in zip([pipe1, clf2, pipe3], clf_labels):
...     scores = cross_val_score(estimator=clf,
>>>                              X=X_train,
>>>                              y=y_train,
>>>                              cv=10,
>>>                              scoring='roc_auc')
>>>     print("ROC AUC: %0.2f (+/- %0.2f) [%s]"
...                % (scores.mean(), scores.std(), label))
```

The output that we receive, as shown in the following snippet, shows that the predictive performances of the individual classifiers are almost equal:

```
10-fold cross validation:

ROC AUC: 0.92 (+/- 0.20) [Logistic Regression]
ROC AUC: 0.92 (+/- 0.15) [Decision Tree]
ROC AUC: 0.93 (+/- 0.10) [KNN]
```

You may be wondering why we trained the logistic regression and k-nearest neighbors classifier as part of a **pipeline**. The reason behind it is that, as discussed in *Chapter 3, A Tour of Machine Learning Classifiers Using Scikit-learn*, both the logistic regression and k-nearest neighbors algorithms (using the Euclidean distance metric) are not scale-invariant in contrast with decision trees. Although the Iris features are all measured on the same scale (cm), it is a good habit to work with standardized features.

Now let's move on to the more exciting part and combine the individual classifiers for majority rule voting in our `MajorityVoteClassifier`:

```
>>> mv_clf = MajorityVoteClassifier(
...                  classifiers=[pipe1, clf2, pipe3])
>>> clf_labels += ['Majority Voting']
>>> all_clf = [pipe1, clf2, pipe3, mv_clf]
>>> for clf, label in zip(all_clf, clf_labels):
...     scores = cross_val_score(estimator=clf,
...                              X=X_train,
...                              y=y_train,
...                              cv=10,
...                              scoring='roc_auc')
...     print("Accuracy: %0.2f (+/- %0.2f) [%s]"
...                % (scores.mean(), scores.std(), label))
```

```
ROC AUC: 0.92 (+/- 0.20) [Logistic Regression]
ROC AUC: 0.92 (+/- 0.15) [Decision Tree]
ROC AUC: 0.93 (+/- 0.10) [KNN]
ROC AUC: 0.97 (+/- 0.10) [Majority Voting]
```

As we can see, the performance of the `MajorityVotingClassifier` has substantially improved over the individual classifiers in the 10-fold cross-validation evaluation.

Resampling for Estimating Statistic:

When we have a big dataset and excited to get started with analyzing it and building your machine learning model. Our machine gives an “out of memory” error while trying to load the dataset.

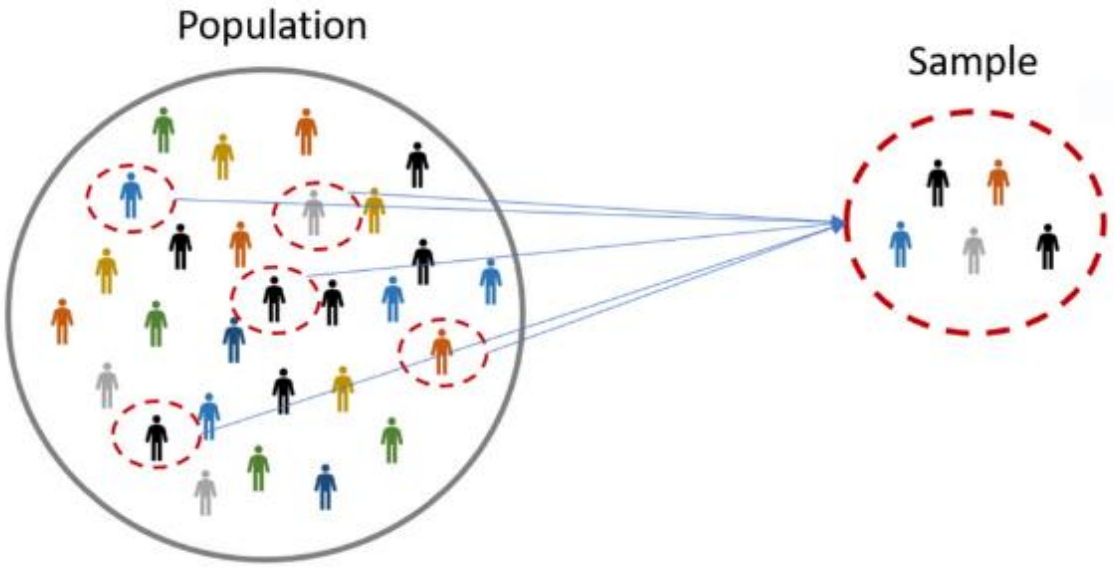
It’s happened to us most of the time when we have big dataset. Big dataset is one of the biggest hurdles we face in data science — dealing with massive amounts of data on computationally limited machines (of course we can resolve it with additional resource power).

So how can we overcome this problem? Is there a way to pick a subset of the data and analyze that — and that can be a good representation of the entire dataset?

Here comes the statistical approach to deal with bigger dataset called **"Sampling".**

"Sampling is a method that allows us to get information about the population based on the statistics from a subset of the population (sample), without having to investigate every individual"

Example: When you conduct research about a group of people, it's rarely possible to collect data from every person in that group. Instead, you select a sample. The sample is the group of individuals who will actually participate in the research.



Sampling from data. Ref Pic: link

Advantages and Challenges of Data Sampling

Sampling can be particularly useful with data sets that are too large to efficiently analyze in full — for example, in big data analytics applications or surveys. Identifying and analyzing a

representative sample is more efficient and cost-effective than surveying the entirety of the data or population.

There are many benefits to sampling compared to working with fuller or complete datasets, including reduced cost and greater speed.

An important consideration, though, is the size of the required data sample and the possibility of introducing a sampling error. In some cases, a small sample can reveal the most important information about a data set. In others, using a larger sample can increase the likelihood of accurately representing the data as a whole, even though the increased size of the sample may impede ease of manipulation and interpretation.

Sampling Framework (Steps):

In order to perform sampling, it requires that you carefully define your population and the method by which you will select (and possibly reject) observations to be a part of your data sample. This may very well be defined by the population parameters that you wish to estimate using the sample.

Some aspects to consider prior to collecting a data sample include:

- **Sample Goal**. The population property that you wish to estimate using the sample.
- **Population**. The scope or domain from which observations could theoretically be made.
- **Selection Criteria**. The methodology that will be used to accept or reject observations in your sample.

- **Sample Size**. The number of observations that will constitute the sample.

Steps involved in sampling framework:

Different steps of sampling

Step 1: *The first stage in the sampling process is to clearly define the target population.*

Step 2: Sampling Frame — It is a list of items or people forming a population from which the sample is taken.

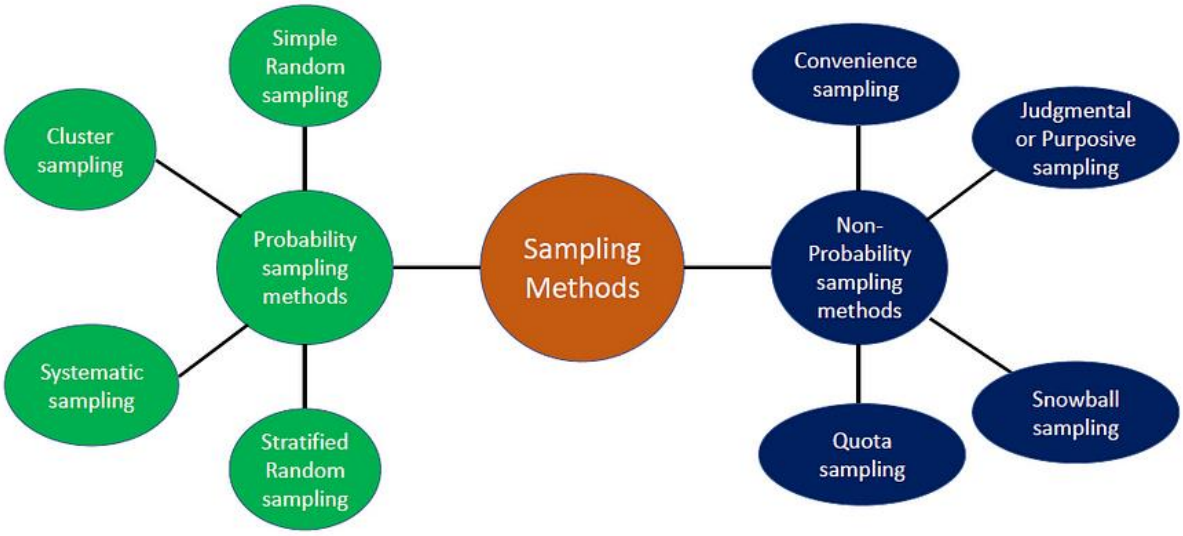
Step3: Generally, probability sampling methods are used because every vote has equal value and any person can be included in the sample irrespective of his caste, community, or religion. Different samples are taken from different regions all over the country.

Step 4: Sample Size — It is the number of individuals or items to be taken in a sample that would be enough to make inferences about the population with the desired level of accuracy and precision. Larger the sample size, more accurate our inference about the population would be. For the polls, agencies try to get as many people as possible of diverse backgrounds to be included in the sample as it would help in predicting the number of seats a political party can win.

Step 5: Once the target population, sampling frame, sampling technique, and sample size have been established, the next step is to **collect data from the sample**. In opinion polls, agencies generally put questions to the people, like which political party are they going to vote for or

has the previous party done any work, etc. Based on the answers, agencies try to interpret who the people of a constituency are going to vote for and approximately how many seats is a political party going to win. Pretty exciting work, right?!

Different Types of Sampling Techniques



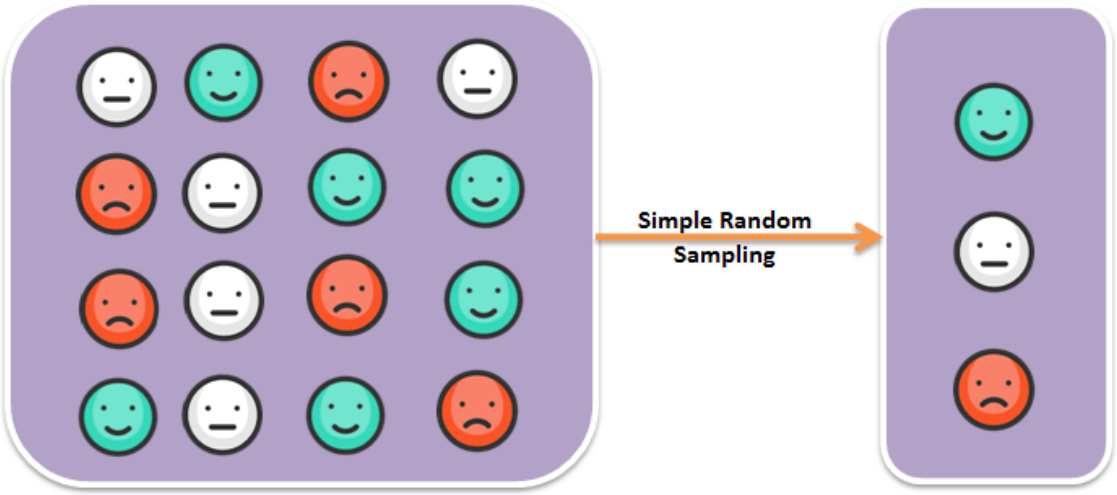
Different types of sampling methods

- **Probability Sampling:** In probability sampling, every element of the population has an equal chance of being selected. Probability sampling gives us the best chance to create a sample that is truly representative of the population
- **Non-Probability Sampling:** In non-probability sampling, all elements do not have an equal chance of being selected. Consequently, there is a significant risk of ending up with a non-representative sample which does not produce generalizable results

Types of Probability Sampling

1. Simple Random Sampling

This is a type of sampling technique you must have come across at some point. Here, every individual is chosen entirely by chance and each member of the population has an equal chance of being selected. **Simple random sampling reduces selection bias.** Simple random sampling reduces the chances of sampling error. Sampling error is lowest in this method out of all the methods.



Simple Random sampling . Ref: Link

2. Cluster Sampling

In a clustered sample, we use the subgroups of the population as the sampling unit rather than individuals. The population is divided into subgroups, known as clusters, and a whole cluster is randomly selected to be included in the study. **This type of sampling is used when we focus on a specific region or area.**

There are different types of cluster sampling — single stage, two stage and multi stage cluster sampling methods.

Example : In population survey clusters are identified and included in a sample based on demographic parameters like age, sex, location, etc. This makes it very simple for a survey creator to derive effective inference from the feedback.

Example of Cluster sampling. Ref: Link

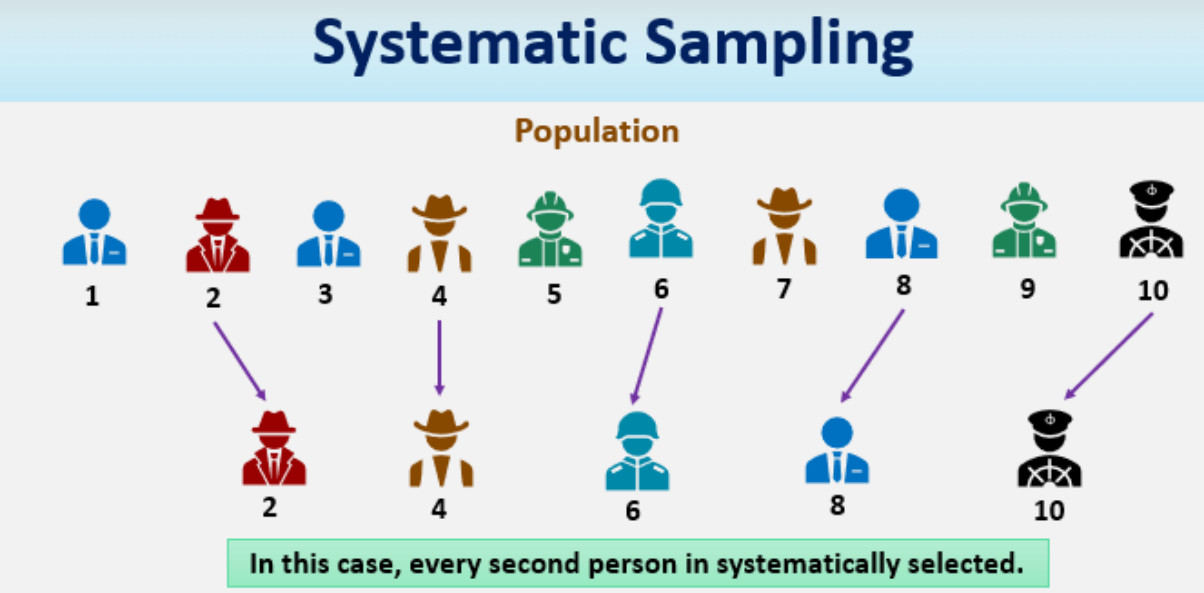
3. Systematic Sampling

Systematic sampling is a statistical method that researchers use to zero down on the desired population they want to research. Researchers calculate the sampling interval by dividing the entire population size by the desired sample size. Systematic sampling is an extended implementation of probability sampling in which each member of the group is selected at regular periods to form a sample.

Systematic sampling is defined as a probability sampling method where the researcher chooses elements from a target population by selecting a random starting point and selects sample members after a fixed ‘sampling interval.’

For example, in school, while selecting the captain of a sports team, most of our coaches asked us to call out numbers such as 1–5 (1-n) and the students with a random number decided by the coach. For instance, three would be called out to be the captains of different teams. It is a non-stressful selection process for both the coach and the players. There's an equal opportunity for every member of a population to be selected using this sampling technique.

For example, a researcher wants to choose 2000 people amongst the population of 10,000 people with the help of systematic sampling. He must enlist all the potential participants, and accordingly, a starting point will be selected. As soon as this list gets formed, every 5th person from the list would be selected as a participant, as 10,000/2000 = 5



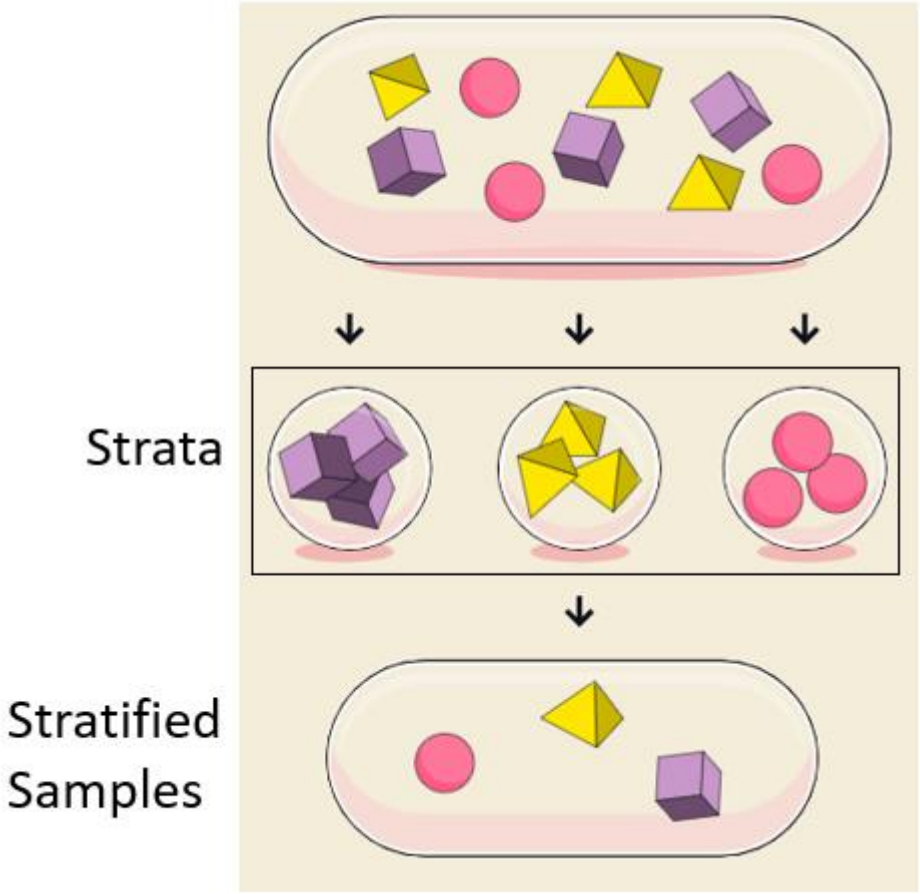
Systematic Sampling. Ref: Link

Here are the types of systematic sampling:

1. Systematic random sampling
2. Linear systematic sampling
3. Circular systematic sampling

4. Stratified random Sampling

In this type of sampling, we divide the population into subgroups (called strata) based on different traits like gender, category, etc. And then we select the sample(s) from these subgroups. We use this type of sampling when we want representation from all the subgroups of the population. However, stratified sampling requires proper knowledge of the characteristics of the population. For example, a researcher looking to analyze the characteristics of people belonging to different annual income divisions will create strata (groups) according to the annual family income. Eg — less than $20,000, $21,000 — $30,000, $31,000 to $40,000, $41,000 to $50,000, etc. By doing this, the researcher concludes the characteristics of people belonging to different income groups. Marketers can analyze which income groups to target and which ones to eliminate to create a roadmap that would bear fruitful results.



Stratified random Sampling. Ref Image link

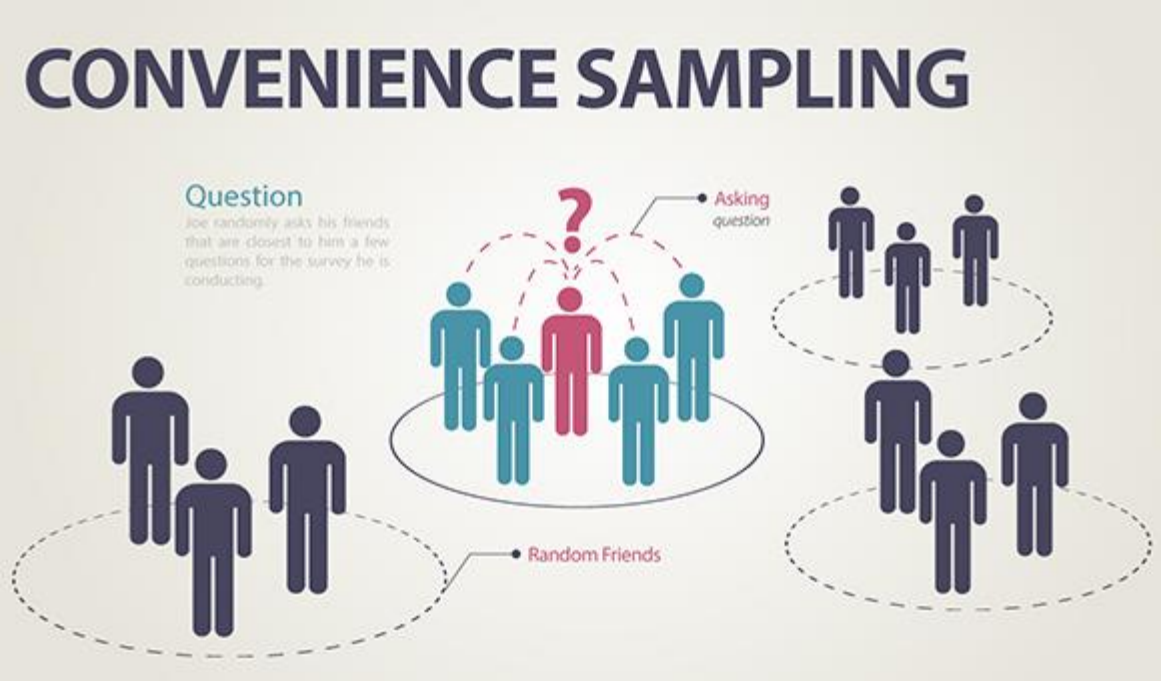
Types of Non Probability Sampling

1. Convenience Sampling

Convenience sampling (also known as availability sampling) method that relies on data collection from population members who are conveniently available to participate in study. Facebook polls or questions can be mentioned as a popular example for convenience sampling. Convenience sampling is a type of sampling where the first available primary data source will be used for the research without additional requirements. In other words, this sampling method involves getting participants wherever you can find them and typically wherever is convenient. In convenience sampling no inclusion criteria identified prior to the selection of subjects. All subjects are invited to participate. In business studies this method can be applied in order to gain initial primary data regarding specific issues such as perception of image of a particular brand or collecting opinions of perspective customers in relation to a new design of a product.

This method of data sampling is typically used when the availability of a sample is rare and expensive. It's also prone to bias, since the sample may not always represent the specific characteristics needed to be studied.

This method has the advantage of being easy to carry out at a relatively low cost in a timely manner. It also allows for gathering useful data and information from a less formal list, like the methods used in probability sampling. Convenience sampling is the preferred method for pilot studies and hypothesis generation.



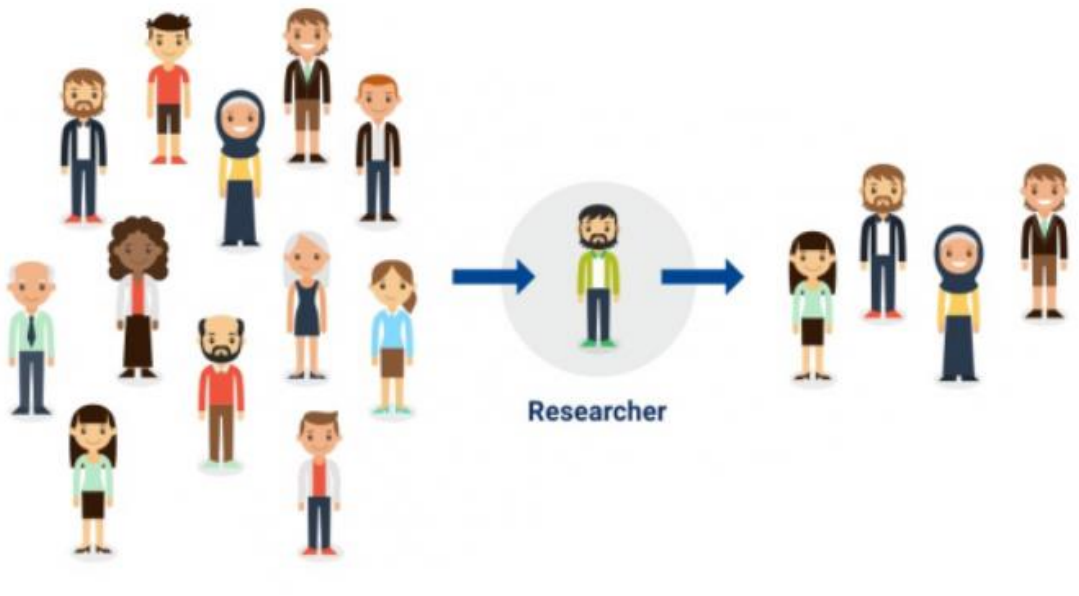
Convenience Sampling. Ref Link

2. Judgmental or Purposive or selective Sampling

Judgment sampling, which is also known as purposive or selective sampling, is based on the assessment of experts in the field when choosing who to ask to be included in the sample.

In this case, let's say you are selecting from a group of women aged 30–35, and the experts decide that only the women who have a college degree will be best suited to be included in the sample. This would be judgment sampling.

Judgment sampling takes less time than other methods, and since there's a smaller data set, researchers should conduct interviews and other hands-on collection techniques to ensure the right type of focus group. Since judgment sampling means researchers can go directly to the target population, there's an increased relevance of the entirety of the sample.



Judgmental or Purposive or selective Sampling. Ref Link

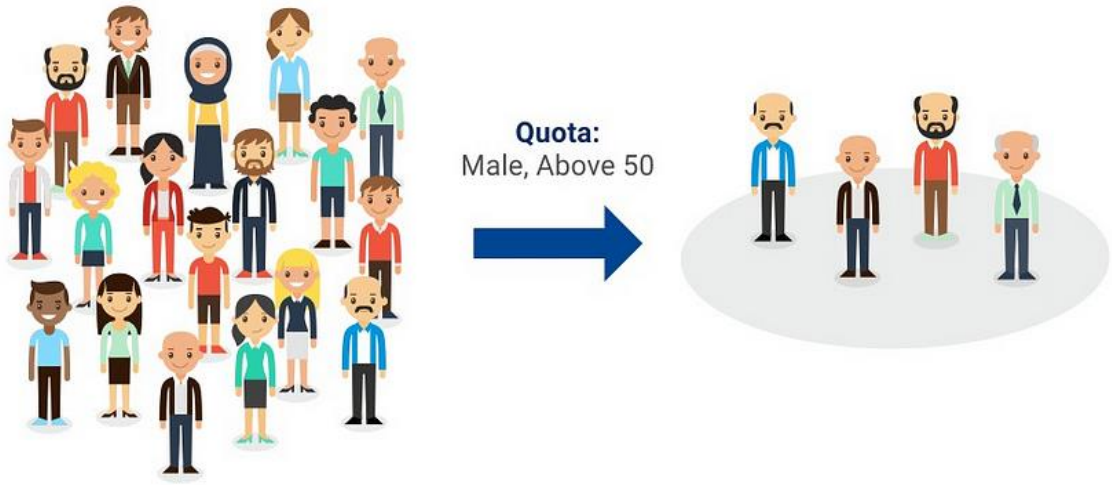
3. Snow ball Sampling

Existing people are asked to nominate further people known to them so that the sample increases in size like a rolling snowball. This method of sampling is effective when a sampling frame is difficult to identify. There is a significant risk of selection bias in snowball sampling, as the referenced individuals will share common traits with the person who recommends them.

Snowball Sampling. Ref Link

4. Quota Sampling

In this type of sampling, we choose items based on predetermined characteristics of the population. For example, it could be all women in a company, or it could be voters in the age range of 18–22 years in a region, etc. This is a way of collecting samples in a fast way but leaves space for bias.



Quota Sampling. Ref Link

Understanding data sampling errors

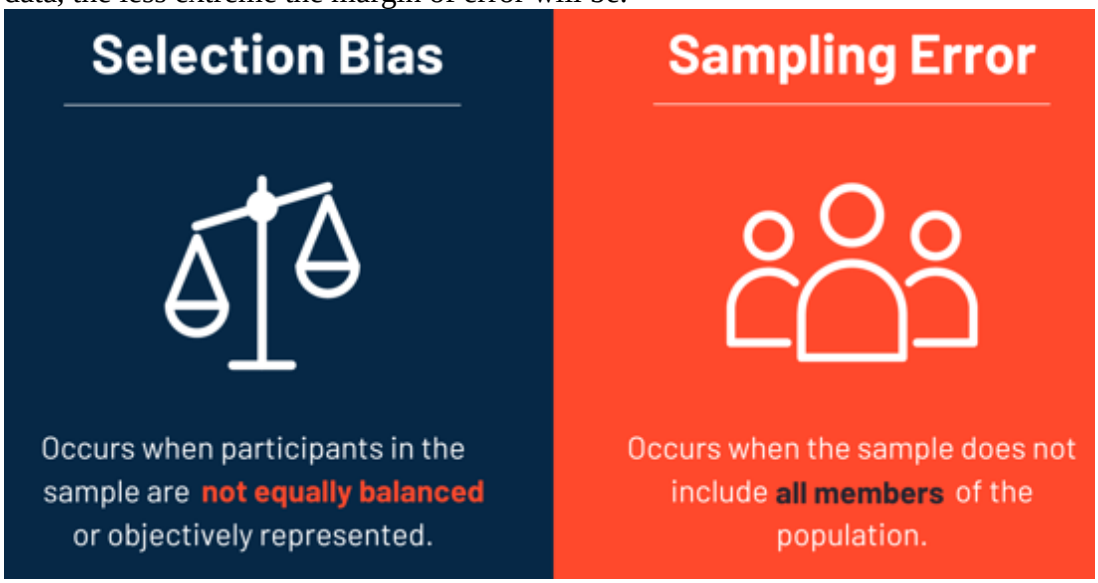
When data sampling occurs, it requires those involved to make statistical conclusions about the population from a series of observations.

Because these observations often come from estimations or generalizations, errors are bound to occur. The two main types of errors that occur when performing data sampling are:

1. **Selection bias**: The bias that's introduced by the selection of individuals to be part of the sample that isn't random. Therefore, the sample cannot be representative of the population that is looking to be analyzed.

2. **Sampling error**: The statistical error that occurs when the researcher doesn't select a sample that represents the entire population of data. When this happens, the results found in the sample don't represent the results that would have been obtained from the entire population.

The only way to 100% eliminate the chance of a sampling error is to test 100% of the population. Of course, this is usually impossible. However, the larger the sample size in your data, the less extreme the margin of error will be.



Introduction

Questions:

- If we are interested solely in the generalization performance, are there any reasons to prefer one classifier or learning algorithm over another?
- If we make no prior assumptions about the nature of the classification task, can we expect any classification method to be superior or inferior overall?
- Can we even find an algorithm that is overall superior to (or inferior to) random guessing?

No Free Lunch Theorem

No Free Lunch Theorem answers: ***NO***

No **pattern classification method** is inherently superior to any other, or even to random guessing

Nature of the problem, prior distribution, data distribution, amount of training data, cost or reward functions, ... are the aspects that determine which form of classifier should provide the best performance.

The ***off-training set error*** (expected value)— the error on points not in the training set — is a good measure for distinguishing algorithms.

No Free Lunch Theorem shows that in the absence of assumptions we should not prefer any learning or classification algorithm over another

Theorem 9.1 (No Free Lunch) *For any two learning algorithms $P_1(h|\mathcal{D})$ and $P_2(h|\mathcal{D})$, the following are true, independent of the sampling distribution $P(\mathbf{x})$ and the number n of training points:*

1. *Uniformly averaged over all target functions F, $\mathcal{E}_1(E|F,n) - \mathcal{E}_2(E|F,n) = 0$;*
2. *For any fixed training set $\mathcal{D}$, uniformly averaged over F, $\mathcal{E}_1(E|F,\mathcal{D}) - \mathcal{E}_2(E|F,\mathcal{D}) = 0$;*
3. *Uniformly averaged over all priors $P(F)$, $\mathcal{E}_1(E|n) - \mathcal{E}_2(E|n) = 0$;*
4. *For any fixed training set $\mathcal{D}$, uniformly averaged over $P(F)$, $\mathcal{E}_1(E|\mathcal{D}) - \mathcal{E}_2(E|\mathcal{D}) = 0$.**

*Ugly Ducking Theorem

In the absence of prior information, is there a principled reason to judge any two distinct patterns as more or less similar than two other distinct patterns?

The *Ugly Duckling Theorem* states that in the absence of assumptions there is no privileged or "best" **feature representation**, and that even the notion of similarity between patterns depends implicitly on assumptions which may or may not be correct.

Find a principled measure the similarity between two patterns, given some representation: the number of predicates (rather than the number of features) the patterns share.

The Theorem forces us to acknowledge that even the apparently simple notion of similarity between patterns is fundamentally based on implicit assumptions about the problem domain

Theorem 9.2 (Ugly Duckling) *Given that we use a finite set of predicates that enables us to distinguish any two patterns under consideration, the number of predicates shared by any two such patterns is constant and independent of the choice of those patterns. Furthermore, if pattern similarity is based on the total number of predicates shared by two patterns, then any two patterns are "equally similar."* *

Minimum Description Length

Algorithmic complexity — also known as Kolmogorov complexity, algorithmic entropy, ... — seeks to quantify an inherent complexity of a binary string (we shall assume both classifiers and patterns are described by such strings).

The *minimum description length* (MDL) principle states that we should minimize the sum of the model's algorithmic complexity and the description of the training data D with respect to that model, i.e.,

$$K(h, \mathcal{D}) = K(h) + K(\mathcal{D} \text{ using } h). \tag{8}$$

Thus we seek the model $h*$ that obeys $h^* = \arg\min_h K(h, D)$

It can be shown theoretically that classifiers designed with a minimum description length principle are guaranteed to converge to the ideal or true model *in the limit of more and more data*.

However, such derivations cannot prove that the principle leads to superior performance in the *finite data case*; to do so would violate the No Free Lunch Theorems.

The minimum description length principle states that simple models (small $K(h)$) are to be preferred, and thus amounts to a bias toward "*simplicity*".

It is found empirically that classifiers designed using the minimum description length principle work well in many problems.

Overfitting avoidance and Occam's razor

To avoid overfitting can be applied regularization, pruning, inclusion of penalty terms, minimizing a description length, and so on.

The *No Free Lunch* results throw such techniques into question: If there are no problem-independent reasons to prefer one algorithm over another, why is overfitting avoidance nearly universally advocated? (but frequent empirical "successes")

Occam's razor: in pattern recognition, one should not use classifiers that are more complicated than are necessary, where "necessary" is determined by the quality of fit to the training data.

The frequent empirical "successes" of Occam's razor imply that the classes of problems addressed have certain properties:

What might be the reason we explore problems that tend to favor simpler classifiers?

Principle of satisficing:

- Human cognition: through evolution, we have had strong selection pressure on our pattern recognition apparatuses to be computationally simple (require fewer neurons, less time, ...).
- Pattern recognition: Design methodology itself imposes a bias toward "simple" classifiers; we generally stop searching for a design when the classifier is "good enough".

Bagging and Boosting Classifier:

Bagging

Bagging — a name derived from "bootstrap aggregation" — uses multiple versions of a training set, each created by drawing $n' < n$ samples from $\mathcal{D}$ with replacement. Each of these bootstrap data sets is used to train a different *component classifier* and the final classification decision is based on the vote of each component classifier.* Traditionally the component classifiers are of the same general form — i.e., all hidden Markov models, or all neural networks, or all decision trees — merely the final parameter values differ among them due to their different sets of training patterns.

A classifier/learning algorithm combination is informally called *unstable* if "small" changes in the training data lead to significantly different classifiers and relatively "large" changes in accuracy.

In general, bagging improves recognition for unstable classifiers since it effectively averages over such discontinuities. There are no convincing theoretical derivations or simulation studies showing that bagging will help all stable classifiers, however.

Bagging is our first encounter with multiclassifier systems, where a f classifier is based on the outputs of a number of component classifiers. Th cision rule in bagging — a simple vote among the component classifiers — elementary method of pooling or integrating the outputs of the component

Boosting

The goal of boosting is to improve the accuracy of any given learning algorithm. In boosting we first create a classifier with accuracy on the training set greater than average, and then add new component classifiers to form an ensemble whose joint decision rule has arbitrarily high accuracy on the training set. In such a case we say the classification performance has been "boosted." In overview, the technique trains successive component classifiers with a subset of the training data that is "most informative" given the current set of component classifiers. Classification of a test point $\mathbf{x}$ is based on the outputs of the component classifiers, as we shall see.

For definiteness, consider creating three component classifiers for a two-category problem through boosting. First we randomly select a set of $n_1 < n$ patterns from the full training set $\mathcal{D}$ (without replacement); call this set $\mathcal{D}_1$. Then we train the first classifier, C_1, with $\mathcal{D}_1$. Classifier C_1 need only be a *weak learner*, i.e., have accuracy only slightly better than chance. (Of course, this is the minimum requirement; a weak learner could have high accuracy on the training set. In that case the benefit

of boosting will be small.) Now we seek a second training set, $\mathcal{D}_2$, that is the "most informative" given component classifier C_1. Specifically, half of the patterns in $\mathcal{D}_2$ should be correctly classified by C_1, half incorrectly classified by C_1 (Problem 29). Such an informative set $\mathcal{D}_2$ is created as follows: we flip a fair coin. If the coin is heads, we select remaining samples from $\mathcal{D}$ and present them, one by one to C_1 until C_1 misclassifies a pattern. We add this misclassified pattern to $\mathcal{D}_2$. Next we flip the coin again. If heads, we continue through $\mathcal{D}$ to find another pattern misclassified by C_1 and add it to $\mathcal{D}_2$ as just described; if tails we find a pattern which C_1 classifies correctly. We continue until no more patterns can be added in this manner. Thus half of the patterns in $\mathcal{D}_2$ are correctly classified by C_1, half are not. As such $\mathcal{D}_2$ provides information complementary to that represented in C_1. Now we train a second component classifier C_2 with $\mathcal{D}_2$.

Next we seek a third data set, $\mathcal{D}_3$, which is not well classified by the combined system C_1 and C_2. We randomly select a training pattern from those remaining in $\mathcal{D}$, and classify that pattern with C_1 and with C_2. If C_1 and C_2 disagree, we add this pattern to the third training set $\mathcal{D}_3$; otherwise we ignore the pattern. We continue adding informative patterns to $\mathcal{D}_3$ in this way; thus $\mathcal{D}_3$ contains those not well represented by the combined decisions of C_1 and C_2. Finally, we train the last component classifier, C_3, with the patterns in $\mathcal{D}_3$.

AdaBoost

There are a number of variations on basic boosting. The most popular, AdaBoost — from "adaptive" boosting — allows the designer to continue adding weak learners until some desired low training error has been achieved. In AdaBoost each training pattern receives a weight which determines its probability of being selected for a training set for an individual component classifier. If a training pattern is accurately classified, then its chance of being used again in a subsequent component classifier is reduced; conversely, if the pattern is not accurately classified, then its chance of being used again is raised. In this way, AdaBoost "focuses in" on the informative or "difficult" patterns. Specifically, we initialize these weights across the training set to to be uniform. On each iteration k, we draw a training set at random according to these weights, and train component classifier C_k on the patterns selected. Next we increase weights of training patterns misclassified by C_k and decrease weights of the patterns correctly classified by C_k. Patterns chosen according to this new distribution are used to train the next classifier, C_{k+1}, and the process is iterated.

Algorithm 1 (AdaBoost)

1 **begin initialize** $\mathcal{D} = \{\mathbf{x}^1, y_1, \mathbf{x}^2, y_2, \ldots, \mathbf{x}^n, y_n\}, k_{max}, W_1(i) = 1/n, i = 1, \ldots, n$
2 $k \leftarrow 0$
3 **do** $k \leftarrow k + 1$
4 Train weak learner C_k using $\mathcal{D}$ sampled according to distribution $W_k(i)$
5 $E_k \leftarrow$ Training error of C_k measured on $\mathcal{D}$ using $W_k(i)$
6 $\alpha_k \leftarrow \frac{1}{2}\ln[(1 - E_k)/E_k]$

7 $$W_{k+1}(i) \leftarrow \frac{W_k(i)}{Z_k} \times \begin{cases} e^{-\alpha_k} & \text{if } h_k(\mathbf{x}^i) = y_i \text{ (correctly classified)} \\ e^{\alpha_k} & \text{if } h_k(\mathbf{x}^i) \neq y_i \text{ (incorrectly classified)} \end{cases}$$

8 **until** $k = k_{max}$
9 **return** C_k and α_k for $k = 1$ to k_{max} (ensemble of classifiers with weights)
10 **end**

Example ADA-Boost

Sample indices	x	y	Weights	$\hat{y}$(x <= 3.0)?	Correct?	Updated weights
1	1.0	1	0.1	1	Yes	0.072
2	2.0	1	0.1	1	Yes	0.072
3	3.0	1	0.1	1	Yes	0.072
4	4.0	-1	0.1	-1	Yes	0.072
5	5.0	-1	0.1	-1	Yes	0.072
6	6.0	-1	0.1	-1	Yes	0.072
7	7.0	1	0.1	-1	Yes	0.167
8	8.0	1	0.1	-1	Yes	0.167
9	9.0	1	0.1	-1	Yes	0.167
10	10.0	-1	0.1	-1	Yes	0.072

$$\varepsilon = 0.1\times 0 + 0.1\times 0 + 0.1\times 0 + 0.1\times 0 + 0.1\times 0 + 0.1\times 0 + 0.1\times 0 + 0.1\times 0 + 0.1\times 0 = \frac{3}{10} = 0.3$$

Next we compute the coefficient α_j (shown in step 6), which is later used in step 7 to update the weights as well as for the weights in majority vote prediction (step 10):

$$\alpha_j = \frac{0.5\log(1-\varepsilon)}{\varepsilon} \approx 0.424$$

$$\boldsymbol{w} := \boldsymbol{w} \times \exp\left(-\alpha_j \times \hat{\boldsymbol{y}} \times \boldsymbol{y}\right)$$

$$0.1\times\exp\left(-0.424\times 1\times 1\right) \approx 0.066$$

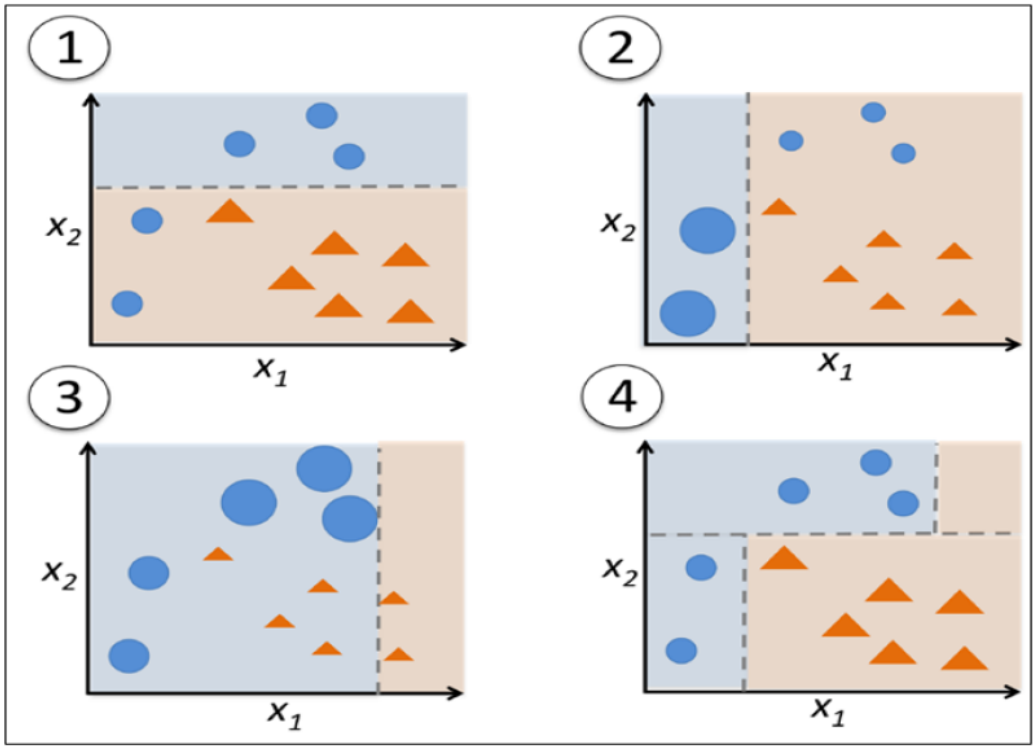
$$0.1\times\exp\left(-0.424\times 1\times(-1)\right) \approx 0.153$$

$$0.1\times\exp\left(-0.424\times(-1)\times(1)\right) \approx 0.153$$

$$\boldsymbol{w} := \frac{\boldsymbol{w}}{\sum_i w_i}$$

Here, $\sum_i w_i = 7\times 0.065 + 3\times 0.153 = 0.914$.

Example



Estimating and comparing classifiers

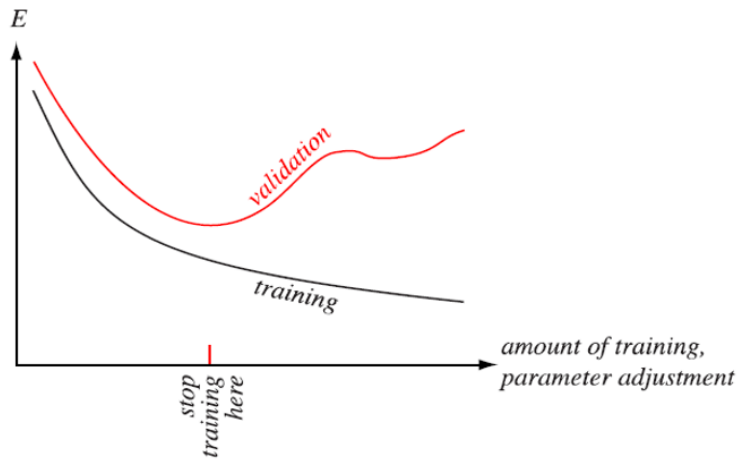
There are at least two reasons for wanting to know the generalization rate of a classifier on a given problem. One is to see if the classifier performs well enough to be useful; another is to compare its performance with that of a competing design. Estimating the final generalization performance invariably requires making assumptions about the classifier or the problem or both, and can fail if the assumptions are not valid. We should stress, then, that all the following methods are heuristic. Indeed, if there were a foolproof method for choosing which of two classifiers would generalize better on an arbitrary new problem, we could incorporate such a method into the learning and violate the No Free Lunch Theorem. Occasionally our assumptions are explicit (as in parametric models), but more often than not they are implicit and difficult to identify or relate to the final estimation (as in empirical methods).

Parametric models

One approach to estimating the generalization rate is to compute it from the assumed parametric model. For example, in the two-class multivariate normal case, we might estimate the probability of error using the Bhattacharyya or Chernoff bounds (Chap **??**), substituting estimates of the means and the covariance matrix for the unknown parameters. However, there are three problems with this approach. First, such an error estimate is often overly optimistic; characteristics that make the training samples peculiar or unrepresentative will not be revealed. Second, we should always suspect the validity of an assumed parametric model; a performance evaluation based on the same model cannot be believed unless the evaluation is unfavorable. Finally, in more general situations where the distributions are not simple it is very difficult to compute the error rate exactly, even if the probabilistic structure is known completely.

Cross validation

In cross validation we randomly split the set of labeled training samples $\mathcal{D}$ into two parts: one is used as the traditional training set for adjusting model parameters in the classifier. The other set — the *validation set* — is used to estimate the generalization error. Since our ultimate goal is low generalization error, we train the classifier until we reach a minimum of this validation error, as sketched in Fig. 9.9. It is essential that the validation (or the test) set not include points used for training the parameters in the classifier — a methodological error known as "testing on the training set."*



Cross validation can be applied to virtually every classification method, where the specific form of learning or parameter adjustment depends upon the general training method. For example, in neural networks of a fixed topology (Chap. **??**), the amount of training is the number of epochs or presentations of the training set. Alternatively, the number of hidden units can be set via cross validation. Likewise, the width of the Gaussian window in Parzen windows (Chap. **??**), and an optimal value of k in the k-nearest neighbor classifier (Chap. **??**) can be set by cross validation.

Cross validation is heuristic and need not (indeed cannot) give improved classifiers in every case. Nevertheless, it is extremely simple and for many real-world problems is found to improve generalization accuracy. There are several heuristics for choosing the portion γ of $\mathcal{D}$ to be used as a validation set ($0 < \gamma < 1$). Nearly always, a smaller portion of the data should be used as validation set ($\gamma < 0.5$) because the validation set is used merely to set a *single* global property of the classifier (i.e., when to stop adjusting parameters) rather than the large number of classifier parameters learned using the training set. If a classifier has a large number of free parameters or degrees of freedom, then a larger portion of $\mathcal{D}$ should be used as a training set, i.e., γ should be reduced. A traditional default is to split the data with $\gamma = 0.1$, which has proven effective in many applications. Finally, when the number of degrees of freedom in the classifier is small compared to the number of training points, the predicted generalization error is relatively insensitive to the choice of γ.

A simple generalization of the above method is *m-fold cross validation*. Here the training set is randomly divided into m disjoint sets of equal size n/m, where n is again the total number of patterns in $\mathcal{D}$. The classifier is trained m times, each time with a different set held out as a validation set. The estimated performance is the mean of these m errors. In the limit where $m = n$, the method is in effect the leave-one-out approach to be discussed in Sect. 9.6.3.

We emphasize that cross validation is a heuristic and need not work on every problem. Indeed, there are problems for which *anti-cross validation* is effective — halting the adjustment of parameters when the validation error is the first local *maximum*. As such, in any particular problem designers must be prepared to explore different values of γ, and possibly abandon the use of cross validation altogether if performance cannot be improved (Computer exercise 5).

Cross validation is, at base, an empirical approach that tests the classifier experimentally. Once we train a classifier using cross validation, the validation error gives an estimate of the accuracy of the final classifier on the unknown test set. If the true but unknown error rate of the classifier is p, and if k of the n' independent, randomly drawn test samples are misclassified, then k has the binomial distribution

$$P(k) = \binom{n'}{k} p^k (1-p)^{n'-k}. \tag{38}$$

Thus, the fraction of test samples misclassified is exactly the maximum likelihood estimate for p (Problem 39):

$$\hat{p} = \frac{k}{n'}. \tag{39}$$

Jackknife and bootstrap estimation of classification accuracy

A method for comparing classifiers closely related to cross validation is to use the jackknife or bootstrap estimation procedures (Sects. 9.4.1 & 9.4.2). The application of the jackknife approach to classification is straightforward. We estimate the accuracy of a given algorithm by training the classifier n separate times, each time using the training set $\mathcal{D}$ from which a different single training point has been deleted. This is merely the $m = n$ limit of m-fold cross validation. Each resulting classifier is tested on the single deleted point and the jackknife estimate of the accuracy is then simply the mean of these leave-one-out accuracies. Here the computational complexity may be very high, especially for large n (Problem 28).

The jackknife, in particular, generally gives good estimates, since each of the the n classifiers is quite similar to the classifier being tested (differing solely due to a single training point). Likewise, the jackknife estimate of the variance of this estimate is given by a simple generalization of Eq. 32. A particular benefit of the jackknife approach is that it can provide measures of confidence or statistical significance in the comparison between two classifier designs. Suppose trained classifier C_1 has an accuracy of 80% while C_2 has accuracy of 85%, as estimated by the jackknife procedure. Is C_2 really better than C_1? To answer this, we calculate the jackknife estimate of the variance of the classification accuracies and use traditional hypothesis testing to see if C_1's apparent superiority is statistically significant (Fig. 9.11).

There are several ways to generalize the bootstrap method to the problem of estimating the accuracy of a classifier. One of the simplest approaches is to train B classifiers, each with a different bootstrap data set, and test on other bootstrap data sets. The bootstrap estimate of the classifier accuracy is simply the mean of these bootstrap accuracies. In practice, the high computational complexity of bootstrap estimation of classifier accuracy is rarely worth possible improvements in that estimate. In Sect. 9.5.1 we shall discuss bagging, a useful modification of bootstrap estimation.

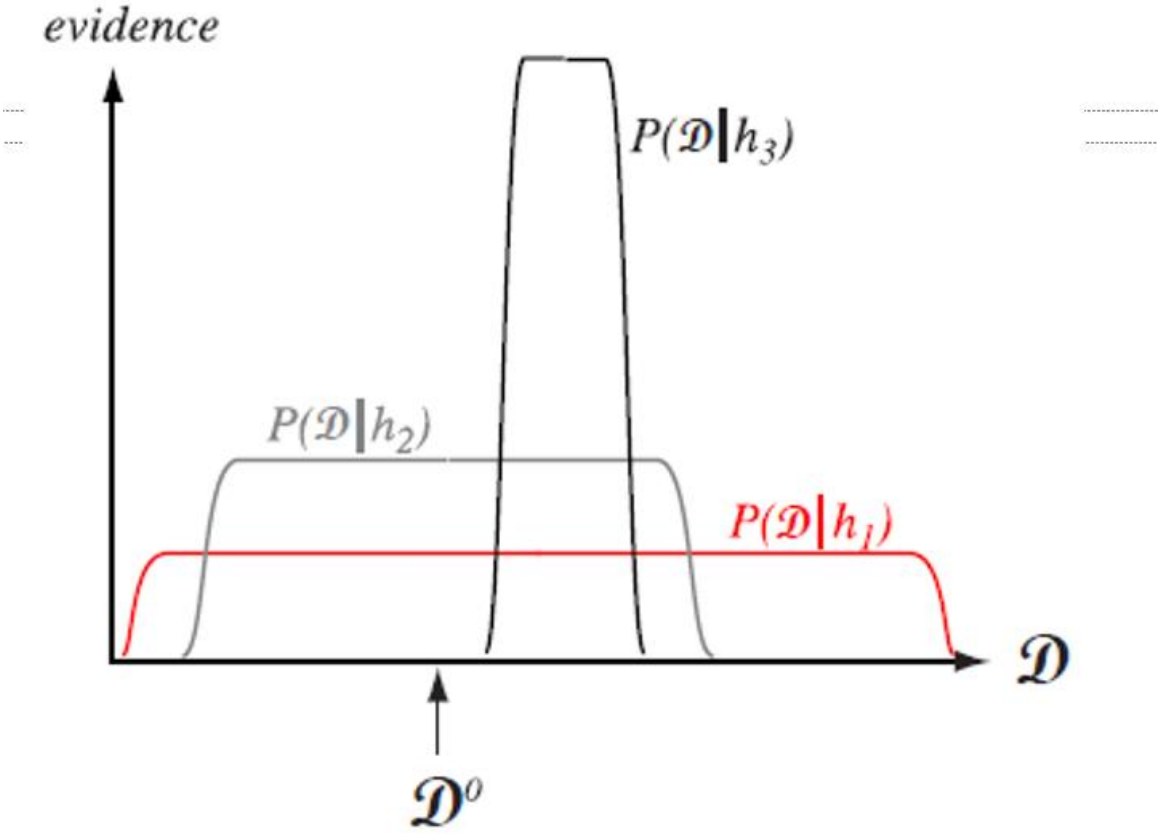
Maximum-likelihood model comparison

Recall first the maximum-likelihood parameter estimation methods discussed in Chap. **??**. Given a model with unknown parameter vector $\boldsymbol{\theta}$, we find the value $\hat{\boldsymbol{\theta}}$ which maximizes the probability of the training data, i.e., $p(\mathcal{D}|\hat{\boldsymbol{\theta}})$. Maximum-likelihood *model comparison* or maximum-likelihood *model selection* — sometimes called ML-II — is a direct generalization of those techniques. The goal here is to choose the *model* that best explains the training data, in a way that will become clear below.

We again let $h_i \in \mathcal{H}$ represent a candidate hypothesis or model (assumed discrete for simplicity), and $\mathcal{D}$ the training data. The posterior probability of any given model is given by Bayes' rule:

$$P(h_i|\mathcal{D}) = \frac{P(\mathcal{D}|h_i)P(h_i)}{p(\mathcal{D})} \propto P(\mathcal{D}|h_i)P(h_i), \tag{40}$$

where we will rarely need the normalizing factor $p(\mathcal{D})$. The data-dependent term, $P(\mathcal{D}|h_i)$, is the *evidence* for h_i; the second term, $P(h_i)$, is our subjective prior over the space of hypotheses — it rates our confidence in different models even before the data arrive. In practice, the data-dependent term dominates in Eq. 40, and hence the priors $P(h_i)$ are often neglected in the computation. In maximum-likelihood model comparison, we find the maximum likelihood parameters for each of the candidate models, calculate the resulting likelihoods, and select the model with the largest such likelihood in Eq. 40 (Fig. 9.12).



The evidence for h_i, i.e., $P(\mathcal{D}|h_i)$, was ignored in a maximum-likelihood setting of parameters $\hat{\boldsymbol{\theta}}$; nevertheless it is the central term in our comparison of models. As mentioned, in practice the evidence term in Eq. 40 dominates the prior term, and it is traditional to ignore such priors, which are often highly subjective or problematic anyway (Problem 38, Computer exercise 7). This procedure represents an inherent bias towards simple models (small $\Delta\boldsymbol{\theta}$); models that are overly complex (large $\Delta\boldsymbol{\theta}$) are automatically self-penalizing where "overly complex" is a data-dependent concept.

In the general case, the full integral of Eq. 41 is too difficult to calculate analytically or even numerically. Nevertheless, if $\boldsymbol{\theta}$ is k-dimensional and the posterior can be assumed to be a Gaussian, then the Occam factor can be calculated directly (Problem 37), yielding:

$$P(\mathcal{D}|h_i) \simeq \underbrace{P(\mathcal{D}|\hat{\boldsymbol{\theta}}, h_i)}_{\substack{\text{best fit} \\ \text{likelihood}}} \underbrace{p(\hat{\boldsymbol{\theta}}|h_i)(2\pi)^{k/2}|\mathbf{H}|^{-1/2}}_{\text{Occam factor}}. \tag{44}$$

where

$$\mathbf{H} = \frac{\partial^2 \ln p(\boldsymbol{\theta}|\mathcal{D}, h_i)}{\partial \boldsymbol{\theta}^2} \tag{45}$$

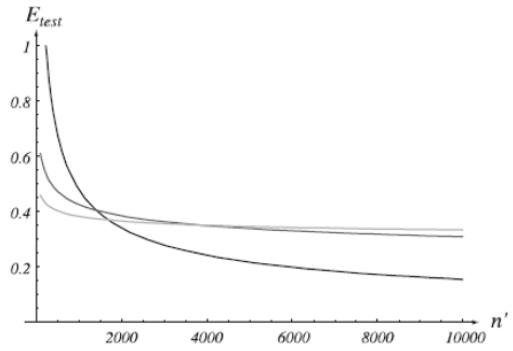
Predicting final performance from learning curves

Training on very large data sets can be computationally intensive, requiring days, weeks or even months on powerful machines. If we are exploring and comparing several different classification techniques, the total training time needed may be unacceptably long. What we seek, then, is a method to compare classifiers without the need of training all of them fully on the complete data set. If we can determine the most promising model quickly and efficiently, we need then only train this model fully.

One method is to use a classifier's performance on a relatively *small* training set to predict its performance on the ultimate large training set. Such performance is revealed in a type of learning curve in which the test error is plotted versus the size of the training set. Figure 9.15 shows the error rate on an independent test set after the classifier has been fully trained on $n' \leq n$ points in the training set. (Note that in this form of learning curve the training error decreases monotonically and does not show "overtraining" evident in curves such as Fig. 9.9.)

For many real-world problems, such learning curves decay monotonically and can be adequately described by a power-law function of the form

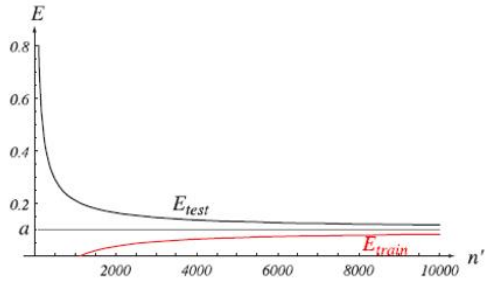
$$E_{test} = a + b/n'^{\alpha} \tag{49}$$



where a, b and $\alpha \geq 1$ depend upon the task and the classifier. In the limit of very large n', the training error equals the test error, since both the training and test sets represent the full problem space. Thus we also model the training error as a power-law function, having the same asymptotic error,

$$E_{train} = a - c/n'^{\beta}. \tag{50}$$

If the classifier is sufficiently powerful, this asymptotic error, a, is equal to the Bayes error. Furthermore, such a powerful classifier can learn perfectly the small training sets and thus the training error (measured on the n' points) will vanish at small n', as shown in Fig. 9.16.



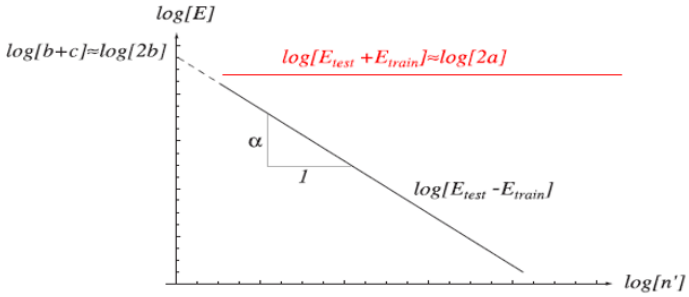
Now we seek to estimate the asymptotic error, a, from the training and test errors on small and intermediate size training sets. From Eqs. 49 & 50 we find:

$$\begin{aligned} E_{test} + E_{train} &= 2a + \frac{b}{n'^{\alpha}} - \frac{c}{n'^{\beta}} \\ E_{test} - E_{train} &= \frac{b}{n^{\alpha}} + \frac{c}{n^{\beta}}. \end{aligned} \tag{51}$$

If we make the assumption of $\alpha = \beta$ and $b = c$, then Eq. 51 reduces to

$$\begin{aligned} E_{test} + E_{train} &= 2a \\ E_{test} - E_{train} &= \frac{2b}{n'^{\alpha}}. \end{aligned} \tag{52}$$

Given this assumption, it is a simple matter to measure the training and test errors for small and intermediate values of n', plot them on a log-log scale and estimate a, as shown in Fig. 9.17. Even if the approximations $\alpha = \beta$ and $b = c$ do not hold in practice, the difference $E_{test} - E_{train}$ nevertheless still forms a straight line on a log-log plot and the sum, $s = b + c$, can be found from the height of the $log[E_{test} + E_{train}]$ curve. The weighted sum $cE_{test} + bE_{train}$ will be a straight line for some empirically set values of b and c, constrained to obey $b + c = s$, enabling a to be estimated (Problem 41). Once a has been estimated for each in the set of candidate classifiers, the one with the lowest a is chosen and must be trained on the full training set $\mathcal{D}$.



Random Forest Classifier:

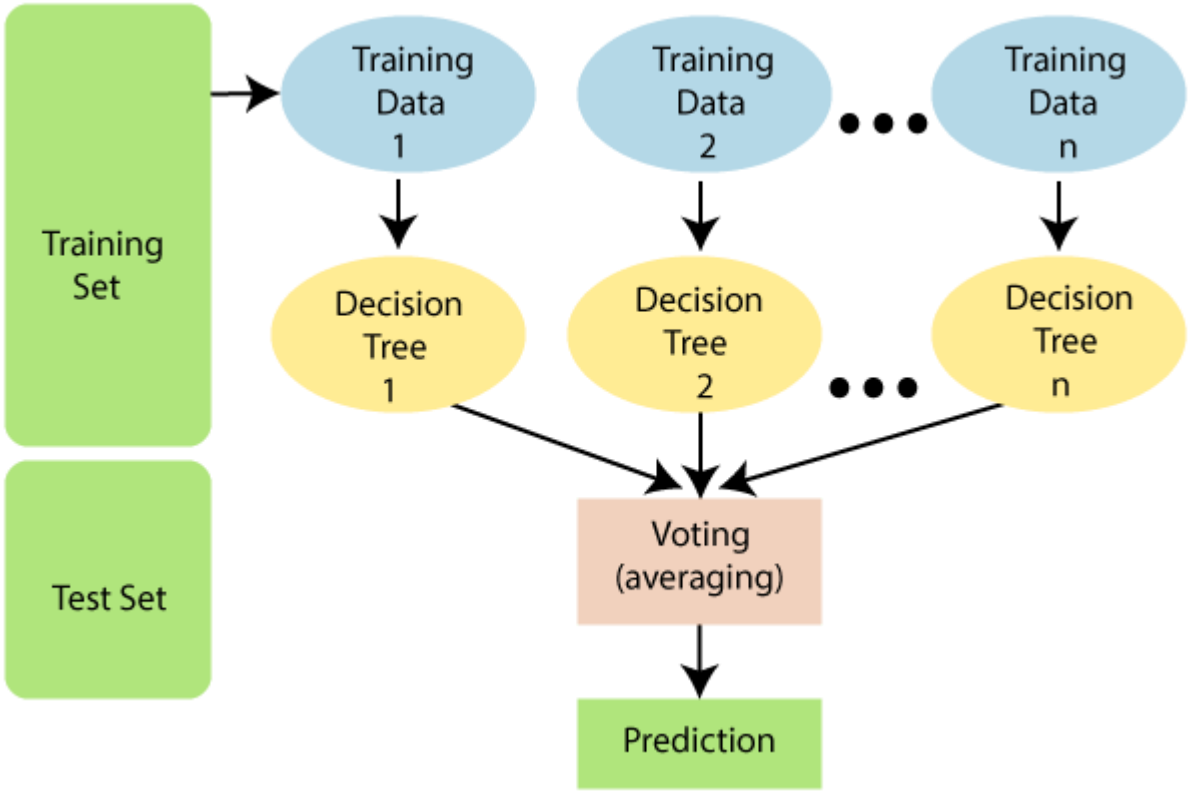
Random Forest Algorithm

Random Forest is a popular machine learning algorithm that belongs to the supervised learning technique. It can be used for both Classification and Regression problems in ML. It is based on the concept of **ensemble learning,** which is a process of *combining multiple classifiers to solve a complex problem and to improve the performance of the model.*

As the name suggests, ***"Random Forest is a classifier that contains a number of decision trees on various subsets of the given dataset and takes the average to improve the predictive accuracy of that dataset."*** Instead of relying on one decision tree, the random forest takes the prediction from each tree and based on the majority votes of predictions, and it predicts the final output.

The greater number of trees in the forest leads to higher accuracy and prevents the problem of overfitting.

The below diagram explains the working of the Random Forest algorithm:



Note: To better understand the Random Forest Algorithm, you should have knowledge of the Decision Tree Algorithm.

Assumptions for Random Forest

Since the random forest combines multiple trees to predict the class of the dataset, it is possible that some decision trees may predict the correct output, while others may not. But together, all the trees predict the correct output. Therefore, below are two assumptions for a better Random forest classifier:

- There should be some actual values in the feature variable of the dataset so that the classifier can predict accurate results rather than a guessed result.
- The predictions from each tree must have very low correlations.

Why use Random Forest?

Below are some points that explain why we should use the Random Forest algorithm:

<=”” li=”” style="user-select: auto;">

- It takes less training time as compared to other algorithms.
- It predicts output with high accuracy, even for the large dataset it runs efficiently.
- It can also maintain accuracy when a large proportion of data is missing.

How does Random Forest algorithm work?

Random Forest works in two-phase first is to create the random forest by combining N decision tree, and second is to make predictions for each tree created in the first phase.

The Working process can be explained in the below steps and diagram:

Step-1: Select random K data points from the training set.

Step-2: Build the decision trees associated with the selected data points (Subsets).

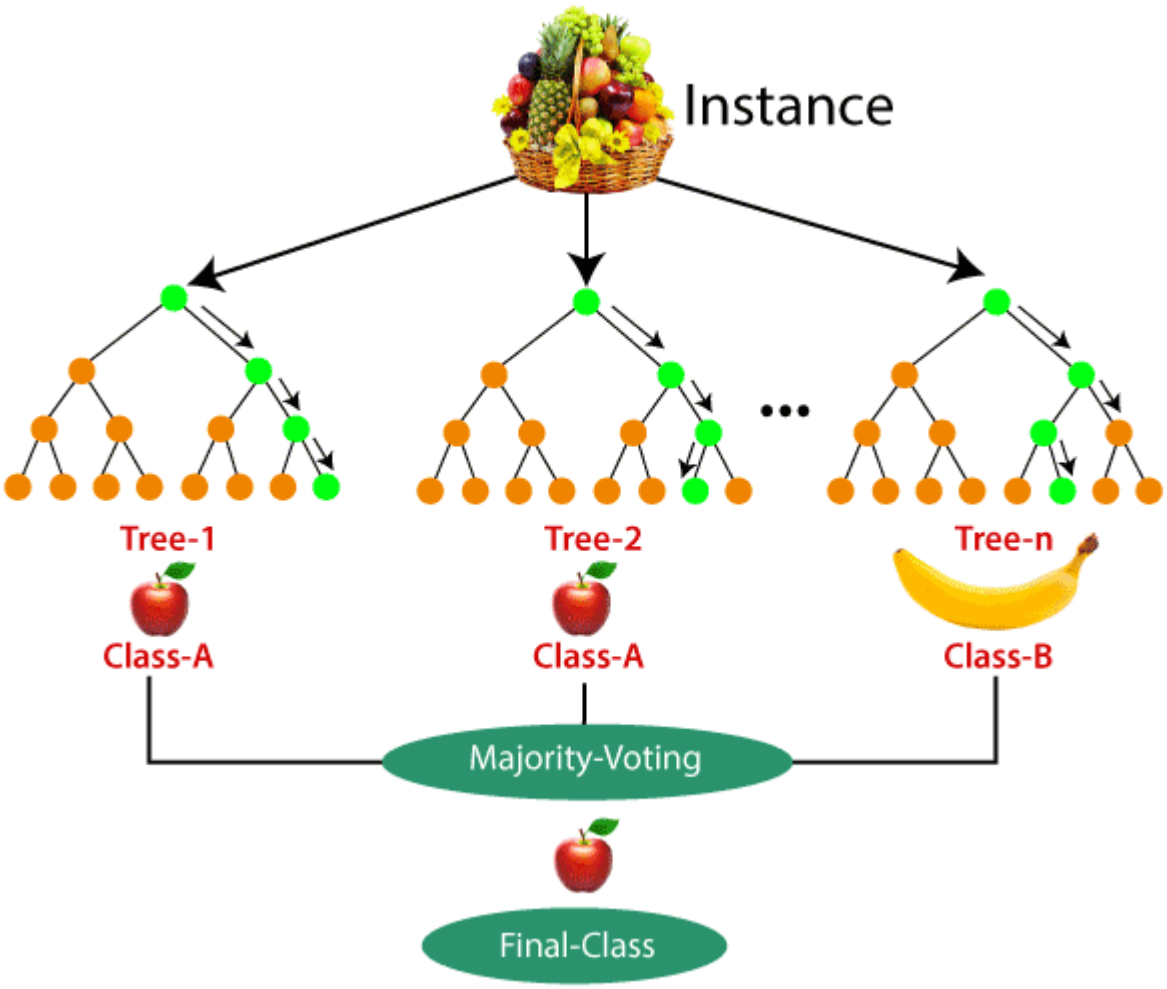
Step-3: Choose the number N for decision trees that you want to build.

Step-4: Repeat Step 1 & 2.

Step-5: For new data points, find the predictions of each decision tree, and assign the new data points to the category that wins the majority votes.

The working of the algorithm can be better understood by the below example:

Example: Suppose there is a dataset that contains multiple fruit images. So, this dataset is given to the Random forest classifier. The dataset is divided into subsets and given to each decision tree. During the training phase, each decision tree produces a prediction result, and when a new data point occurs, then based on the majority of results, the Random Forest classifier predicts the final decision. Consider the below image:



Applications of Random Forest

There are mainly four sectors where Random forest mostly used:

1. **Banking:** Banking sector mostly uses this algorithm for the identification of loan risk.
2. **Medicine:** With the help of this algorithm, disease trends and risks of the disease can be identified.
3. **Land Use:** We can identify the areas of similar land use by this algorithm.
4. **Marketing:** Marketing trends can be identified using this algorithm.

Advantages of Random Forest

- Random Forest is capable of performing both Classification and Regression tasks.
- It is capable of handling large datasets with high dimensionality.
- It enhances the accuracy of the model and prevents the overfitting issue.

Disadvantages of Random Forest

- Although random forest can be used for both classification and regression tasks, it is not more suitable for Regression tasks.

Python Implementation of Random Forest Algorithm

Now we will implement the Random Forest Algorithm tree using Python. For this, we will use the same dataset "user_data.csv", which we have used in previous classification models. By using the same dataset, we can compare the Random Forest classifier with other classification models such as Decision tree Classifier, KNN, SVM, Logistic Regression, etc.

Implementation Steps are given below:

- Data Pre-processing step
- Fitting the Random forest algorithm to the Training set
- Predicting the test result
- Test accuracy of the result (Creation of Confusion matrix)
- Visualizing the test set result.

1.Data Pre-Processing Step:

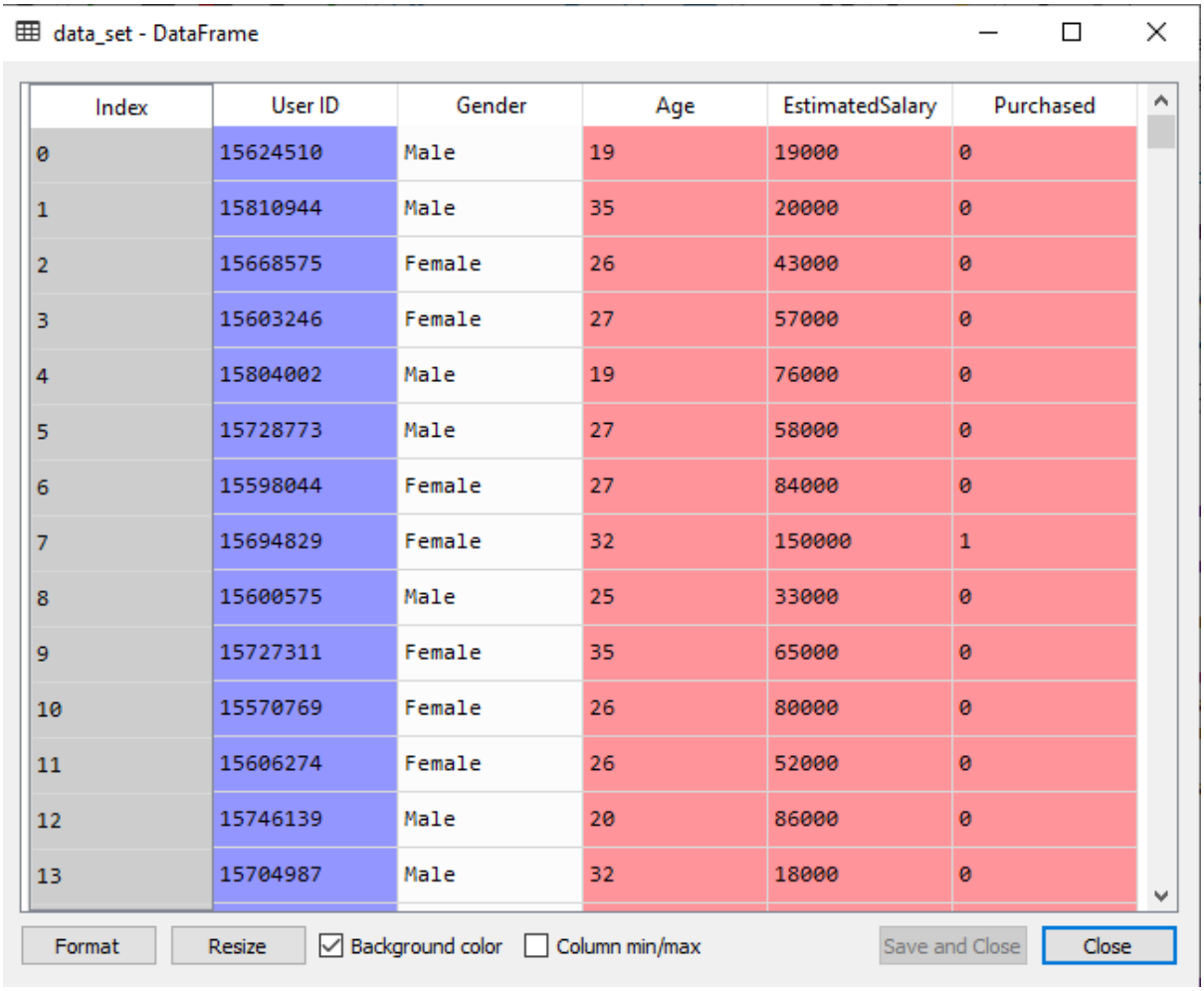
Below is the code for the pre-processing step:

```
1. # importing libraries
2. import numpy as nm
3. import matplotlib.pyplot as mtp
4. import pandas as pd
5.
6. #importing datasets
7. data_set= pd.read_csv('user_data.csv')
8.
9. #Extracting Independent and dependent Variable
10. x= data_set.iloc[:, [2,3]].values
11. y= data_set.iloc[:, 4].values
12.
13. # Splitting the dataset into training and test set.
14. from sklearn.model_selection import train_test_split
```

```
15. x_train, x_test, y_train, y_test= train_test_split(x, y, test_size= 0.25, random_state=0)


16.
17. #feature Scaling
18. from sklearn.preprocessing import StandardScaler
19. st_x= StandardScaler()
20. x_train= st_x.fit_transform(x_train)
21. x_test= st_x.transform(x_test)
```

In the above code, we have pre-processed the data. Where we have loaded the dataset, which is given as:



Index	User ID	Gender	Age	EstimatedSalary	Purchased
0	15624510	Male	19	19000	0
1	15810944	Male	35	20000	0
2	15668575	Female	26	43000	0
3	15603246	Female	27	57000	0
4	15804002	Male	19	76000	0
5	15728773	Male	27	58000	0
6	15598044	Female	27	84000	0
7	15694829	Female	32	150000	1
8	15600575	Male	25	33000	0
9	15727311	Female	35	65000	0
10	15570769	Female	26	80000	0
11	15606274	Female	26	52000	0
12	15746139	Male	20	86000	0
13	15704987	Male	32	18000	0

2. Fitting the Random Forest algorithm to the training set:

Now we will fit the Random forest algorithm to the training set. To fit it, we will import the **RandomForestClassifier** class from the **sklearn.ensemble** library. The code is given below:

```
1. #Fitting Decision Tree classifier to the training set
2. from sklearn.ensemble import RandomForestClassifier
3. classifier= RandomForestClassifier(n_estimators= 10, criterion="entropy")
4. classifier.fit(x_train, y_train)
```

In the above code, the classifier object takes below parameters:

- **n_estimators=** The required number of trees in the Random Forest. The default value is 10. We can choose any number but need to take care of the overfitting issue.
- **criterion=** It is a function to analyze the accuracy of the split. Here we have taken "entropy" for the information gain.

Output:

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='entropy',
                       max_depth=None, max_features='auto', max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=10,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
```

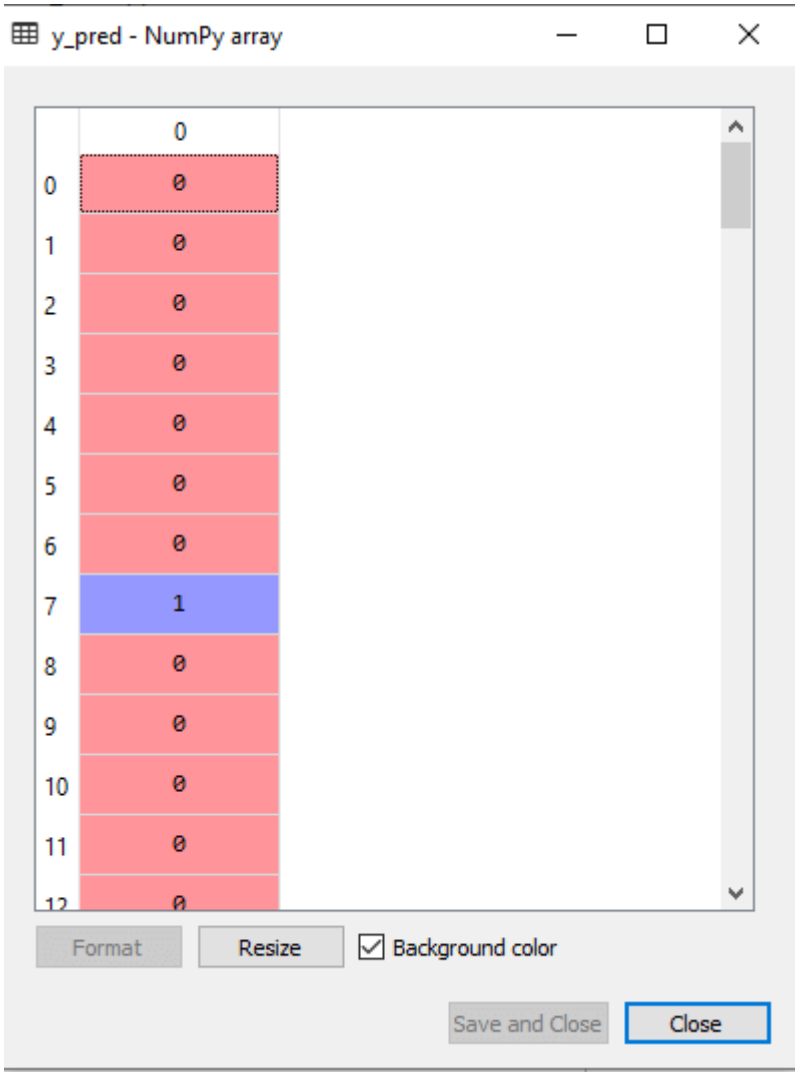
3. Predicting the Test Set result

Since our model is fitted to the training set, so now we can predict the test result. For prediction, we will create a new prediction vector y_pred. Below is the code for it:

```
1. #Predicting the test set result
2. y_pred= classifier.predict(x_test)
```

Output:

The prediction vector is given as:



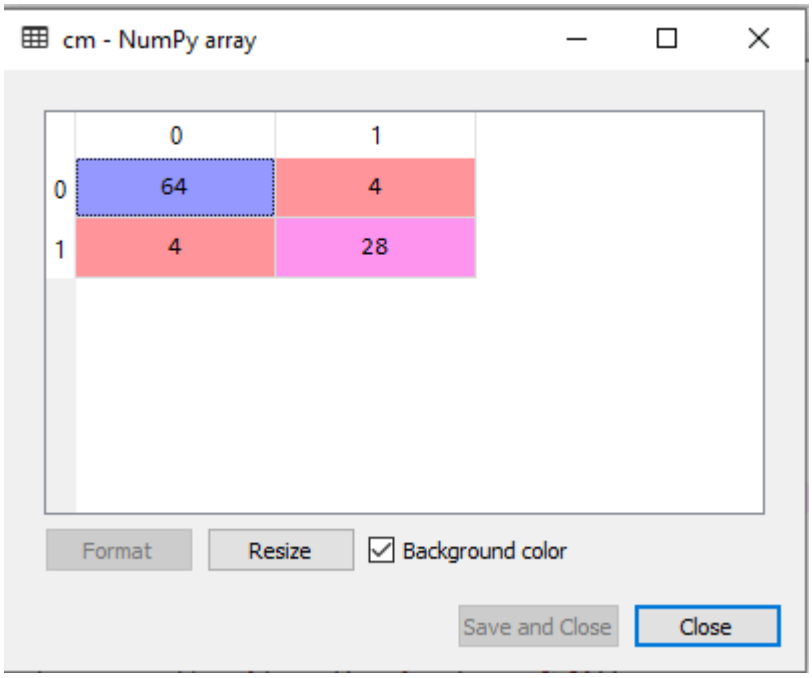
By checking the above prediction vector and test set real vector, we can determine the incorrect predictions done by the classifier.

4. Creating the Confusion Matrix

Now we will create the confusion matrix to determine the correct and incorrect predictions. Below is the code for it:

```
1. #Creating the Confusion matrix
2. from sklearn.metrics import confusion_matrix
3. cm= confusion_matrix(y_test, y_pred)
```

Output:



As we can see in the above matrix, there are **4+4= 8 incorrect predictions** and **64+28= 92 correct predictions.**

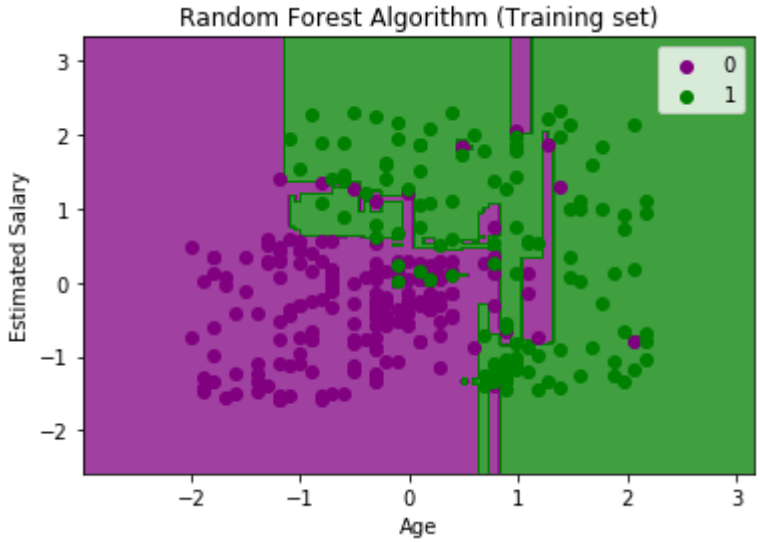
5. Visualizing the training Set result

Here we will visualize the training set result. To visualize the training set result we will plot a graph for the Random forest classifier. The classifier will predict yes or No for the users who have either Purchased or Not purchased the SUV car as we did in Logistic Regression. Below is the code for it:

```
1. from matplotlib.colors import ListedColormap
2. x_set, y_set = x_train, y_train
3. x1, x2 = nm.meshgrid(nm.arange(start = x_set[:, 0].min() - 1, stop = x_set[:, 0].max()
   + 1, step  =0.01),
4. nm.arange(start = x_set[:, 1].min() - 1, stop = x_set[:, 1].max() + 1, step = 0.01))
5. mtp.contourf(x1, x2, classifier.predict(nm.array([x1.ravel(), x2.ravel()]).T).reshape(x1
   .shape),
6. alpha = 0.75, cmap = ListedColormap(('purple','green' )))
7. mtp.xlim(x1.min(), x1.max())
8. mtp.ylim(x2.min(), x2.max())
9. for i, j in enumerate(nm.unique(y_set)):
10.     mtp.scatter(x_set[y_set == j, 0], x_set[y_set == j, 1],
11.         c = ListedColormap(('purple', 'green'))(i), label = j)
12. mtp.title('Random Forest Algorithm (Training set)')
```

```
13. mtp.xlabel('Age')
14. mtp.ylabel('Estimated Salary')
15. mtp.legend()
16. mtp.show()
```

Output:



The above image is the visualization result for the Random Forest classifier working with the training set result. It is very much similar to the Decision tree classifier. Each data point corresponds to each user of the user_data, and the purple and green regions are the prediction regions. The purple region is classified for the users who did not purchase the SUV car, and the green region is for the users who purchased the SUV.

So, in the Random Forest classifier, we have taken 10 trees that have predicted Yes or NO for the Purchased variable. The classifier took the majority of the predictions and provided the result.

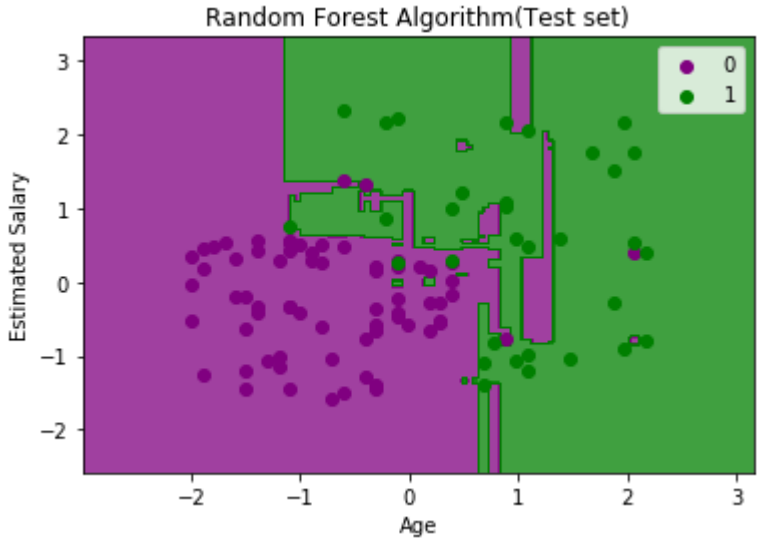
6. Visualizing the test set result

Now we will visualize the test set result. Below is the code for it:

```
1. #Visulaizing the test set result
2. from matplotlib.colors import ListedColormap
3. x_set, y_set = x_test, y_test
4. x1, x2 = nm.meshgrid(nm.arange(start = x_set[:, 0].min() - 1, stop = x_set[:, 0].max()
   + 1, step  =0.01),
5. nm.arange(start = x_set[:, 1].min() - 1, stop = x_set[:, 1].max() + 1, step = 0.01))
```

```
6. mtp.contourf(x1, x2, classifier.predict(nm.array([x1.ravel(), x2.ravel()]).T).reshape(x1
   .shape),
7. alpha = 0.75, cmap = ListedColormap(('purple','green' )))
8. mtp.xlim(x1.min(), x1.max())
9. mtp.ylim(x2.min(), x2.max())
10. for i, j in enumerate(nm.unique(y_set)):
11.     mtp.scatter(x_set[y_set == j, 0], x_set[y_set == j, 1],
12.         c = ListedColormap(('purple', 'green'))(i), label = j)
13. mtp.title('Random Forest Algorithm(Test set)')
14. mtp.xlabel('Age')
15. mtp.ylabel('Estimated Salary')
16. mtp.legend()
17. mtp.show()
```

Output:



The above image is the visualization result for the test set. We can check that there is a minimum number of incorrect predictions (8) without the Overfitting issue. We will get different results by changing the number of trees in the classifier.

Regressor:

Regression analysis is a statistical method to model the relationship between a dependent (target) and independent (predictor) variables with one or more independent variables. More specifically, Regression analysis helps us to understand how the value of the dependent variable is changing corresponding to an independent variable when other independent

variables are held fixed. It predicts continuous/real values such as **temperature, age, salary, price,** etc.

We can understand the concept of regression analysis using the below example:

Example: Suppose there is a marketing company A, who does various advertisement every year and get sales on that. The below list shows the advertisement made by the company in the last 5 years and the corresponding sales:

Advertisement	Sales
$90	$1000
$120	$1300
$150	$1800
$100	$1200
$130	$1380
$200	??

Now, the company wants to do the advertisement of $200 in the year 2019 **and wants to know the prediction about the sales for this year**. So to solve such type of prediction problems in machine learning, we need regression analysis.

Regression is a supervised learning technique which helps in finding the correlation between variables and enables us to predict the continuous output variable based on the one or more predictor variables. It is mainly used for **prediction, forecasting, time series modeling, and determining the causal-effect relationship between variables**.

In Regression, we plot a graph between the variables which best fits the given datapoints, using this plot, the machine learning model can make predictions about the data. In simple words, ***"Regression shows a line or curve that passes through all the datapoints on target-predictor graph in such a way that the vertical distance between the datapoints and the regression line is minimum."*** The distance between datapoints and line tells whether a model has captured a strong relationship or not.

Some examples of regression can be as:

- Prediction of rain using temperature and other factors

- Determining Market trends
- Prediction of road accidents due to rash driving.

Terminologies Related to the Regression Analysis:

- **Dependent Variable:** The main factor in Regression analysis which we want to predict or understand is called the dependent variable. It is also called **target variable**.
- **Independent Variable:** The factors which affect the dependent variables or which are used to predict the values of the dependent variables are called independent variable, also called as a **predictor**.
- **Outliers:** Outlier is an observation which contains either very low value or very high value in comparison to other observed values. An outlier may hamper the result, so it should be avoided.
- **Multicollinearity:** If the independent variables are highly correlated with each other than other variables, then such condition is called Multicollinearity. It should not be present in the dataset, because it creates problem while ranking the most affecting variable.
- **Underfitting and Overfitting:** If our algorithm works well with the training dataset but not well with test dataset, then such problem is called **Overfitting**. And if our algorithm does not perform well even with training dataset, then such problem is called **underfitting**.

Why do we use Regression Analysis?

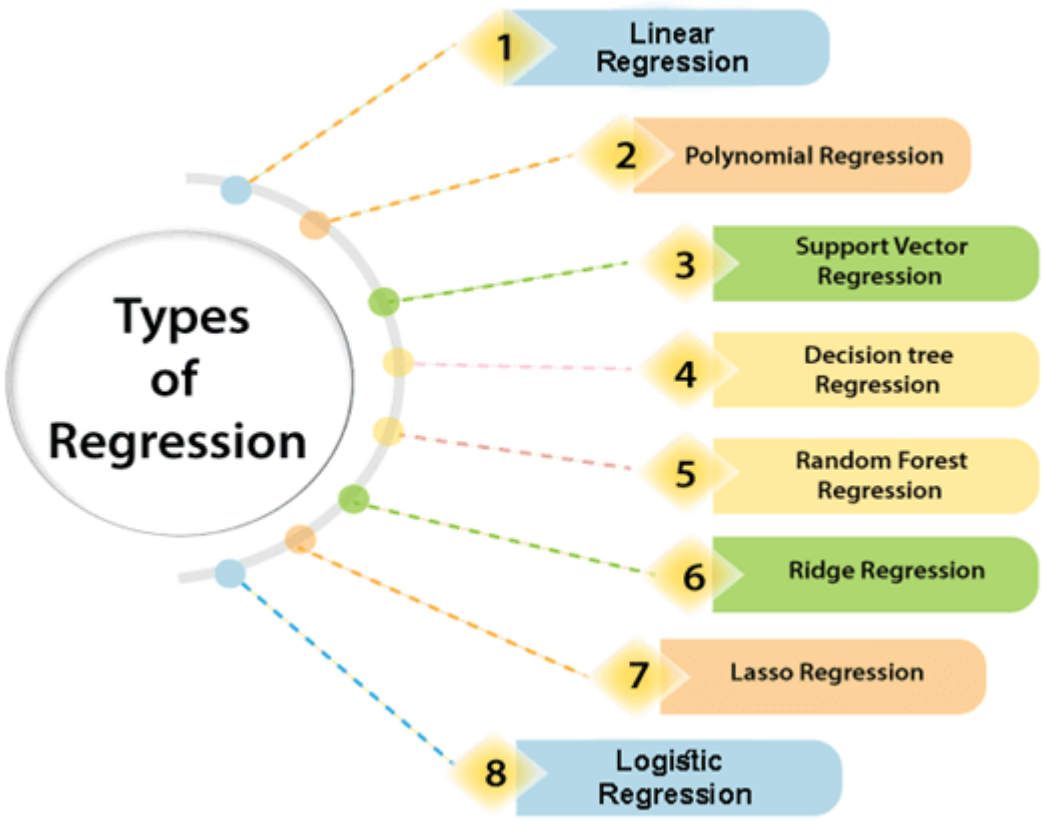
As mentioned above, Regression analysis helps in the prediction of a continuous variable. There are various scenarios in the real world where we need some future predictions such as weather condition, sales prediction, marketing trends, etc., for such case we need some technology which can make predictions more accurately. So for such case we need Regression analysis which is a statistical method and used in machine learning and data science. Below are some other reasons for using Regression analysis:

- Regression estimates the relationship between the target and the independent variable.
- It is used to find the trends in data.
- It helps to predict real/continuous values.
- By performing the regression, we can confidently determine the **most important factor, the least important factor, and how each factor is affecting the other factors**.

Types of Regression

There are various types of regressions which are used in data science and machine learning. Each type has its own importance on different scenarios, but at the core, all the regression methods analyze the effect of the independent variable on dependent variables. Here we are discussing some important types of regression which are given below:

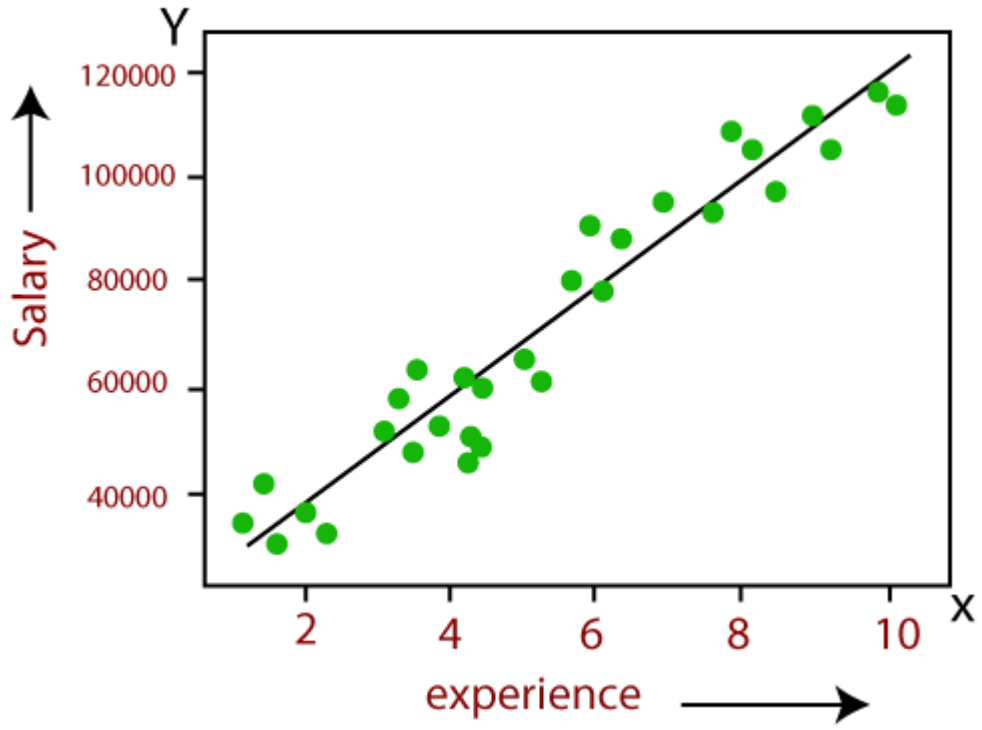
- **Linear Regression**
- **Logistic Regression**
- **Polynomial Regression**
- **Support Vector Regression**
- **Decision Tree Regression**
- **Random Forest Regression**
- **Ridge Regression**
- **Lasso Regression:**



Linear Regression:

- Linear regression is a statistical regression method which is used for predictive analysis.
- It is one of the very simple and easy algorithms which works on regression and shows the relationship between the continuous variables.

- It is used for solving the regression problem in machine learning.
- Linear regression shows the linear relationship between the independent variable (X-axis) and the dependent variable (Y-axis), hence called linear regression.
- If there is only one input variable (x), then such linear regression is called **simple linear regression**. And if there is more than one input variable, then such linear regression is called **multiple linear regression**.
- The relationship between variables in the linear regression model can be explained using the below image. Here we are predicting the salary of an employee on the basis of **the year of experience**.



- Below is the mathematical equation for Linear regression:

1. Y= aX+b

Here, Y = dependent variables (target variables), X= Independent variables (predictor variables), a and b are the linear coefficients

Some popular applications of linear regression are:

- **Analyzing trends and sales estimates**

- **Salary forecasting**
- **Real estate prediction**
- **Arriving at ETAs in traffic.**

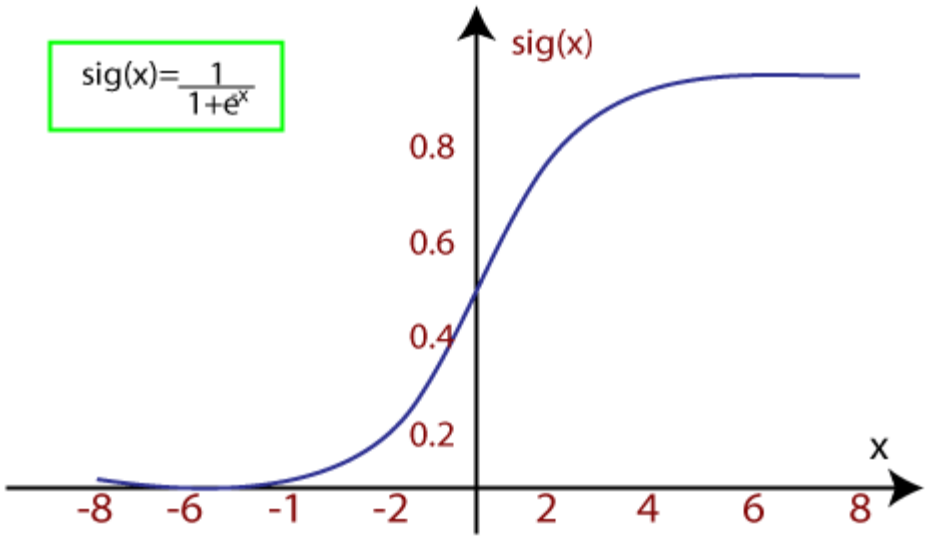
Logistic Regression:

- Logistic regression is another supervised learning algorithm which is used to solve the classification problems. In **classification problems**, we have dependent variables in a binary or discrete format such as 0 or 1.
- Logistic regression algorithm works with the categorical variable such as 0 or 1, Yes or No, True or False, Spam or not spam, etc.
- It is a predictive analysis algorithm which works on the concept of probability.
- Logistic regression is a type of regression, but it is different from the linear regression algorithm in the term how they are used.
- Logistic regression uses **sigmoid function** or logistic function which is a complex cost function. This sigmoid function is used to model the data in logistic regression. The function can be represented as:

$$f(x)= \frac{1}{1+e^{-x}}$$

- f(x)= Output between the 0 and 1 value.
- x= input to the function
- e= base of natural logarithm.

When we provide the input values (data) to the function, it gives the S-curve as follows:



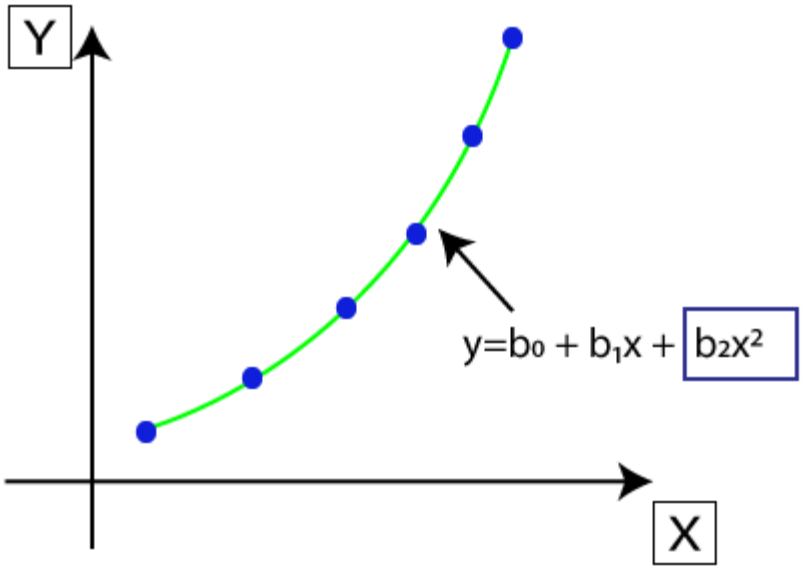
- It uses the concept of threshold levels, values above the threshold level are rounded up to 1, and values below the threshold level are rounded up to 0.

There are three types of logistic regression:

- **Binary(0/1, pass/fail)**
- **Multi(cats, dogs, lions)**
- **Ordinal(low, medium, high)**

Polynomial Regression:

- Polynomial Regression is a type of regression which models the **non-linear dataset** using a linear model.
- It is similar to multiple linear regression, but it fits a non-linear curve between the value of x and corresponding conditional values of y.
- Suppose there is a dataset which consists of datapoints which are present in a non-linear fashion, so for such case, linear regression will not best fit to those datapoints. To cover such datapoints, we need Polynomial regression.
- I**n Polynomial regression, the original features are transformed into polynomial features of given degree and then modeled using a linear model.** Which means the datapoints are best fitted using a polynomial line.



- The equation for polynomial regression also derived from linear regression equation that means Linear regression equation $Y= b_0+ b_1x$, is transformed into Polynomial regression equation $Y= b_0+b_1x+ b_2x^2+ b_3x^3+.....+ b_nx^n$.
- Here Y is the **predicted/target output, b_0, b_1,... b_n are the regression coefficients**. x is our **independent/input variable**.
- The model is still linear as the coefficients are still linear with quadratic

Note: ***This is different from Multiple Linear regression in such a way that in Polynomial regression, a single element has different degrees instead of multiple variables with the same degree.***

Support Vector Regression:

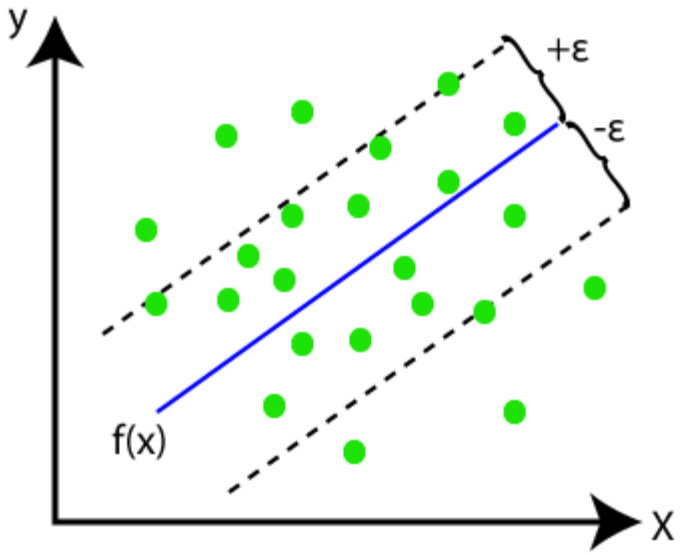
Support Vector Machine is a supervised learning algorithm which can be used for regression as well as classification problems. So if we use it for regression problems, then it is termed as Support Vector Regression.

Support Vector Regression is a regression algorithm which works for continuous variables. Below are some keywords which are used in **Support Vector Regression**:

- **Kernel:** It is a function used to map a lower-dimensional data into higher dimensional data.
- **Hyperplane:** In general SVM, it is a separation line between two classes, but in SVR, it is a line which helps to predict the continuous variables and cover most of the datapoints.
- **Boundary line:** Boundary lines are the two lines apart from hyperplane, which creates a margin for datapoints.

- **Support vectors:** Support vectors are the datapoints which are nearest to the hyperplane and opposite class.

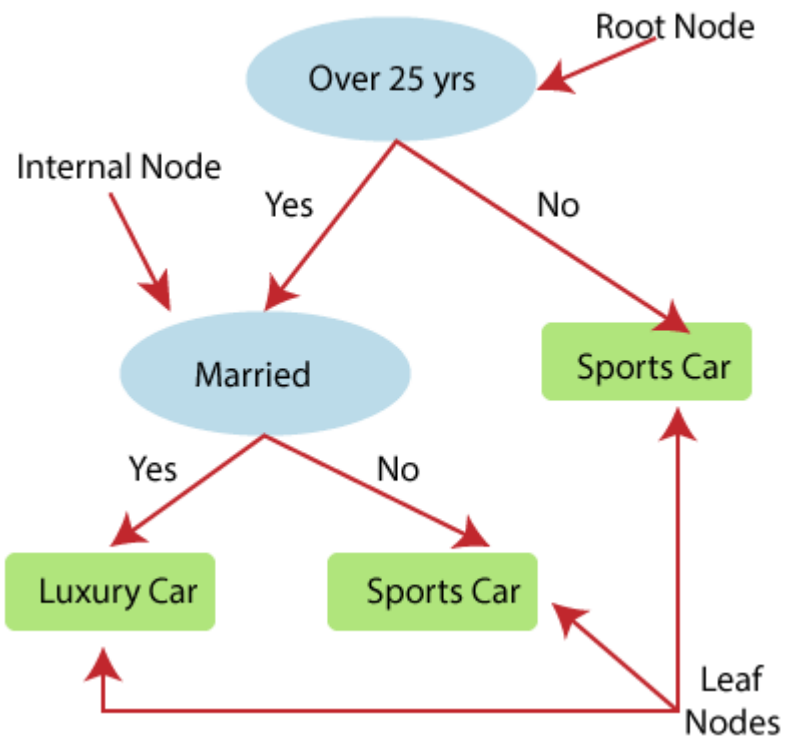
In SVR, we always try to determine a hyperplane with a maximum margin, so that maximum number of datapoints are covered in that margin. ***The main goal of SVR is to consider the maximum datapoints within the boundary lines and the hyperplane (best-fit line) must contain a maximum number of datapoints***. Consider the below image:



Here, the blue line is called hyperplane, and the other two lines are known as boundary lines.

Decision Tree Regression:

- Decision Tree is a supervised learning algorithm which can be used for solving both classification and regression problems.
- It can solve problems for both categorical and numerical data
- Decision Tree regression builds a tree-like structure in which each internal node represents the "test" for an attribute, each branch represent the result of the test, and each leaf node represents the final decision or result.
- A decision tree is constructed starting from the root node/parent node (dataset), which splits into left and right child nodes (subsets of dataset). These child nodes are further divided into their children node, and themselves become the parent node of those nodes. Consider the below image:

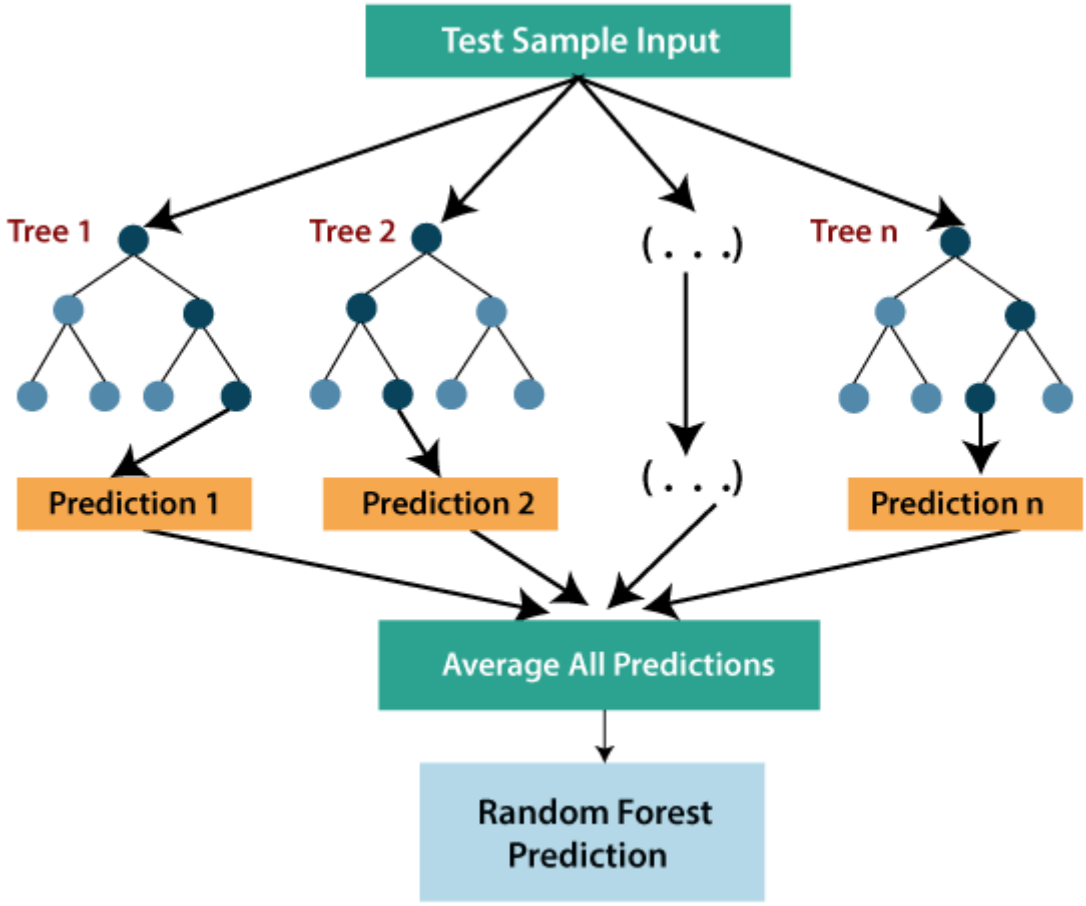


Above image showing the example of Decision Tee regression, here, the model is trying to predict the choice of a person between Sports cars or Luxury car.

- Random forest is one of the most powerful supervised learning algorithms which is capable of performing regression as well as classification tasks.
- The Random Forest regression is an ensemble learning method which combines multiple decision trees and predicts the final output based on the average of each tree output. The combined decision trees are called as base models, and it can be represented more formally as:

$g(x)= f_0(x)+ f_1(x)+ f_2(x)+....$

- Random forest uses **Bagging or Bootstrap Aggregation** technique of ensemble learning in which aggregated decision tree runs in parallel and do not interact with each other.
- With the help of Random Forest regression, we can prevent Overfitting in the model by creating random subsets of the dataset.



Ridge Regression:

- Ridge regression is one of the most robust versions of linear regression in which a small amount of bias is introduced so that we can get better long term predictions.
- The amount of bias added to the model is known as **Ridge Regression penalty**. We can compute this penalty term by multiplying with the lambda to the squared weight of each individual features.
- The equation for ridge regression will be:

$$L(x, y) = \text{Min}\left(\sum_{i=1}^{n}(y_i - w_i x_i)^2 + \lambda \sum_{i=1}^{n}(w_i)^2\right)$$

- A general linear or polynomial regression will fail if there is high collinearity between the independent variables, so to solve such problems, Ridge regression can be used.
- Ridge regression is a regularization technique, which is used to reduce the complexity of the model. It is also called as **L2 regularization**.
- It helps to solve the problems if we have more parameters than samples.

Lasso Regression:

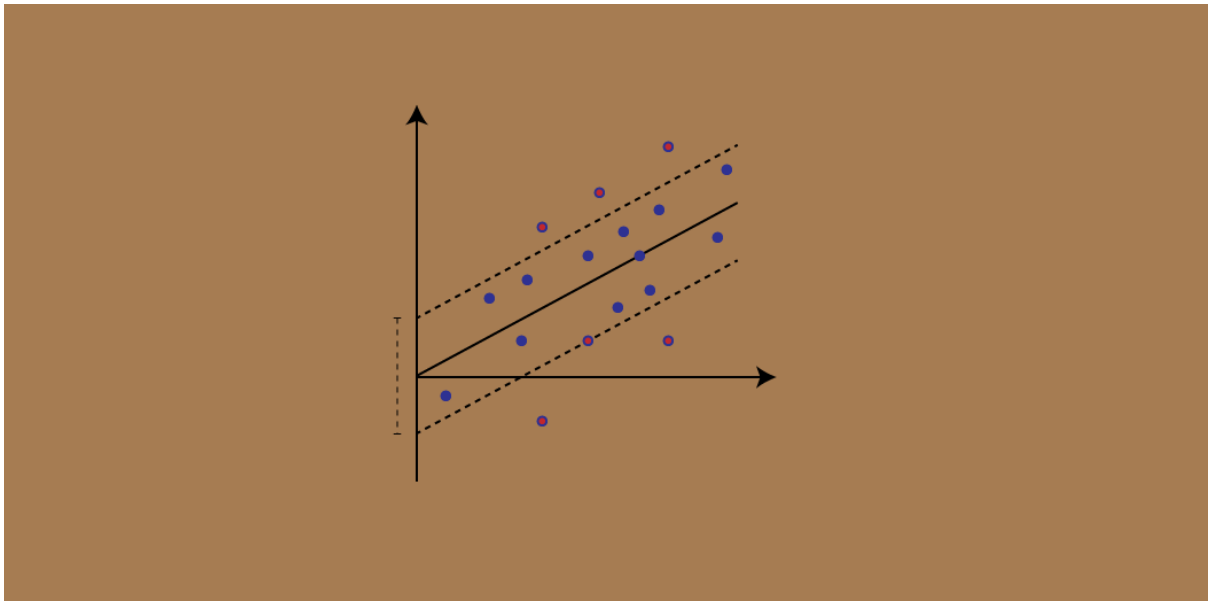
- Lasso regression is another regularization technique to reduce the complexity of the model.
- It is similar to the Ridge Regression except that penalty term contains only the absolute weights instead of a square of weights.
- Since it takes absolute values, hence, it can shrink the slope to 0, whereas Ridge Regression can only shrink it near to 0.
- It is also called as **L1 regularization**. The equation for Lasso regression will be:

$$L(x, y) = \text{Min}\left(\sum_{i=1}^{n} (y_i - w_i x_i)^2 + \lambda \sum_{i=1}^{n} |w_i| \right)$$

Support Vector Classifier and Regressor:

Support Vector Machines (SVM) are popularly and widely used for classification problems in machine learning. I've often relied on this not just in machine learning projects but when I want a quick result in a hackathon.

But SVM for regression analysis? I hadn't even considered the possibility for a while! And even now when I bring up "Support Vector Regression" in front of machine learning beginners, I often get a bemused expression. I understand – most courses and experts don't even mention Support Vector Regression (SVR) as a machine learning algorithm.



But SVR has its uses as you'll see in this tutorial. We will first quickly understand what SVM is, before diving into the world of Support Vector Regression in machine learning and how to implement it in Python!

Note: You can learn about Support Vector Machines and Regression problems in course format here (it's free!):

- *Support Vector Machine (SVM) in Python and R*
- *Fundamentals of Regression Analysis*

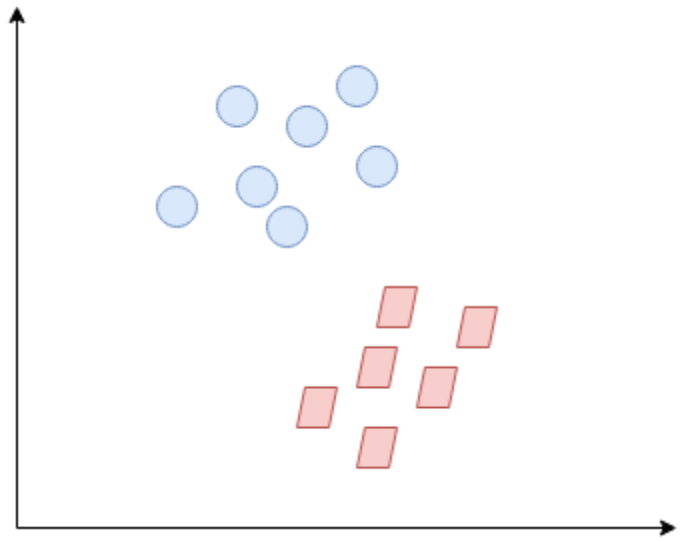
Here's what we'll cover in this Support Vector Regression tutorial:

- What is a Support Vector Machine (SVM)?
- Hyperparameters of the Support Vector Machine Algorithm
- Introduction to Support Vector Regression (SVR)
- Implementing Support Vector Regression in Python

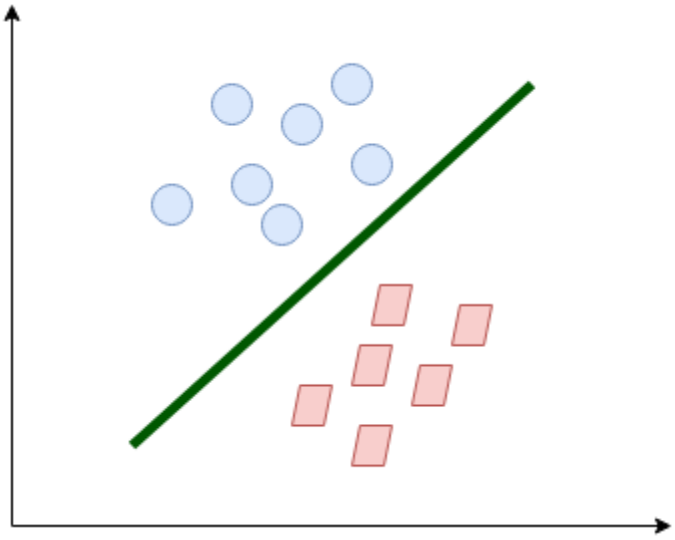
What is a Support Vector Machine (SVM)?

A Support Vector Machine (SVM) is a supervised machine learning algorithm used for classification and regression tasks. SVM works by finding a hyperplane in a high-dimensional space that best separates data into different classes. It aims to maximize the margin (the distance between the hyperplane and the nearest data points of each class) while minimizing classification errors. SVM can handle both linear and non-linear classification problems by using various kernel functions. It's widely used in tasks such as image classification, text categorization, and more.

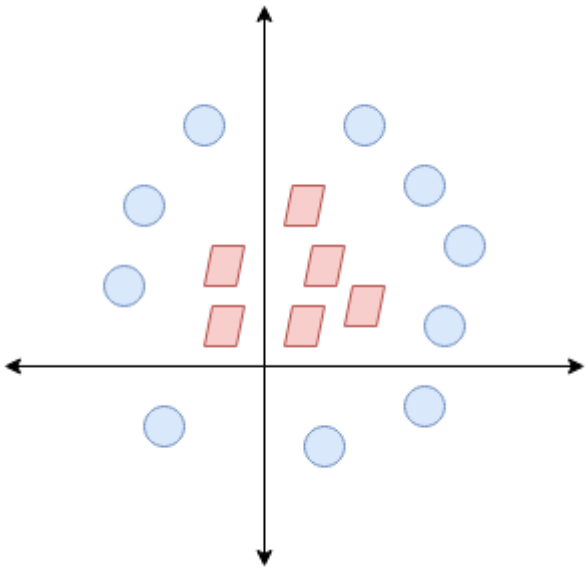
So what exactly is Support Vector Machine (SVM)? We'll start by understanding SVM in simple terms. Let's say we have a plot of two label classes as shown in the figure below:



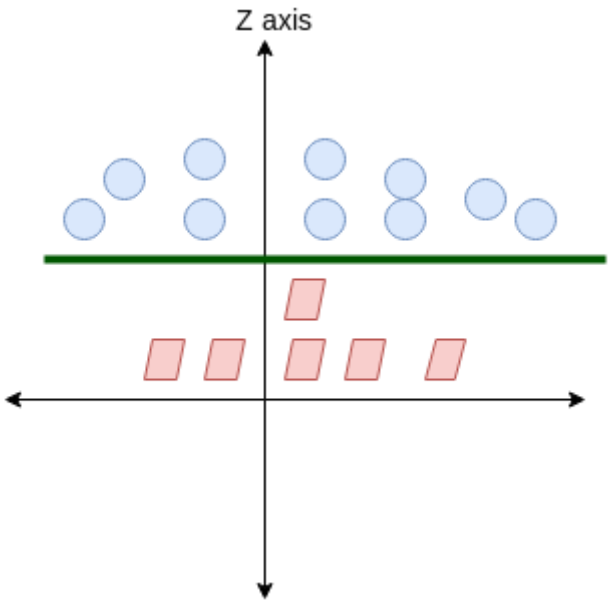
Can you decide what the separating line will be? You might have come up with this:



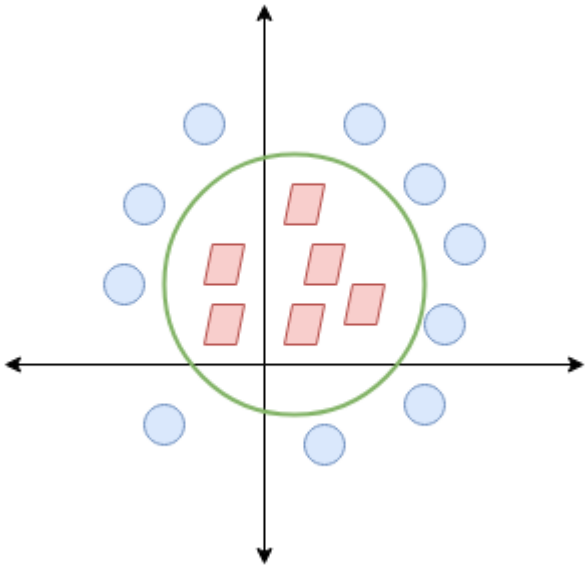
The line fairly separates the classes. This is what SVM essentially does – **simple class separation.** Now, what is the data was like this:



Here, we don't have a simple line separating these two classes. So we'll extend our dimension and introduce a new dimension along the z-axis. We can now separate these two classes:



When we transform this line back to the original plane, it maps to the circular boundary as I've shown here:

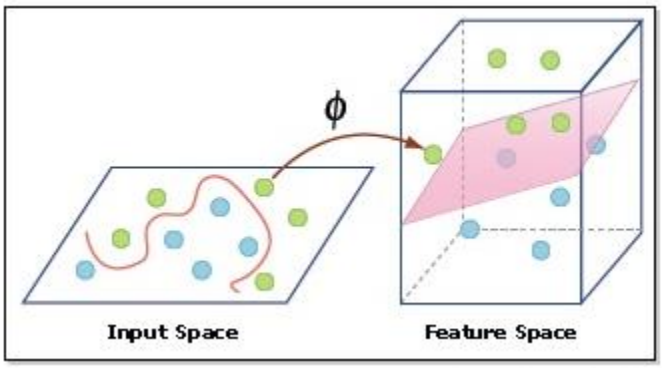


This is exactly what SVM does! It tries to find a line/hyperplane (in multidimensional space) that separates these two classes. Then it classifies the new point depending on whether it lies on the positive or negative side of the hyperplane depending on the classes to predict.

Hyperparameters of the Support Vector Machine (SVM) Algorithm

There are a few important parameters of SVM that you should be aware of before proceeding further:

- **Kernel:** A kernel helps us find a hyperplane in the higher dimensional space without increasing the computational cost. Usually, the computational cost will increase if the dimension of the data increases. This increase in dimension is required when we are unable to find a separating hyperplane in a given dimension and are required to move in a higher dimension:



- **Hyperplane:** This is basically a separating line between two data classes in SVM. But in Support Vector Regression, this is the line that will be used to predict the continuous output
- **Decision Boundary**: A decision boundary can be thought of as a demarcation line (for simplification) on one side of which lie positive examples and on the other side lie the negative examples. On this very line, the examples may be classified as either positive or negative. This same concept of SVM will be applied in Support Vector Regression as well

To understand SVM from scratch, I recommend this tutorial: Understanding Support Vector Machine(SVM) algorithm from examples.

Introduction to Support Vector Regression (SVR)

Support Vector Regression (SVR) is a type of machine learning algorithm used for regression analysis. The goal of SVR is to find a function that approximates the relationship between the input variables and a continuous target variable, while minimizing the prediction error.

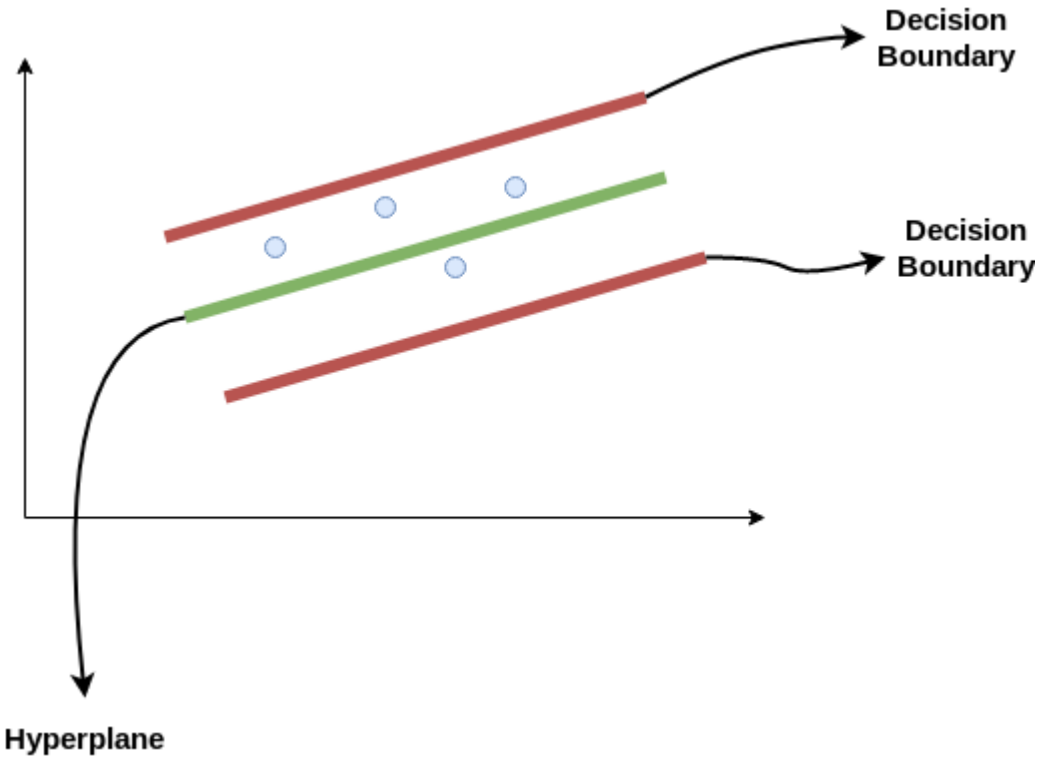
Unlike Support Vector Machines (SVMs) used for classification tasks, SVR seeks to find a hyperplane that best fits the data points in a continuous space. This is achieved by mapping the input variables to a high-dimensional feature space and finding the hyperplane that maximizes the margin (distance) between the hyperplane and the closest data points, while also minimizing the prediction error.

SVR can handle non-linear relationships between the input variables and the target variable by using a kernel function to map the data to a higher-dimensional space. This makes it a powerful tool for regression tasks where there may be complex relationships between the input variables and the target variable.

Support Vector Regression (SVR) uses the same principle as SVM, but for regression problems. Let's spend a few minutes understanding the idea behind SVR.

The Idea Behind Support Vector Regression

The problem of regression is to find a function that approximates mapping from an input domain to real numbers on the basis of a training sample. So let's now dive deep and understand how SVR works actually.



Consider these two red lines as the decision boundary and the green line as the hyperplane. **Our objective, when we are moving on with SVR, is to basically consider the points that are within the decision boundary line.** Our best fit line is the hyperplane that has a maximum number of points.

The first thing that we'll understand is what is the decision boundary (the danger red line above!). Consider these lines as being at any distance, say 'a', from the hyperplane. So, these are the lines that we draw at distance '+a' and '-a' from the hyperplane. This 'a' in the text is basically referred to as epsilon.

Assuming that the equation of the hyperplane is as follows:

Y = wx+b (equation of hyperplane)

Then the equations of decision boundary become:

wx+b= +a

wx+b= -a

Thus, any hyperplane that satisfies our SVR should satisfy:

-a < Y- wx+b < +a

Our main aim here is to decide a decision boundary at 'a' distance from the original hyperplane such that data points closest to the hyperplane or the support vectors are within that boundary line.

Hence, we are going to take only those points that are within the decision boundary and have the least error rate, or are within the Margin of Tolerance. This gives us a better fitting model.

Implementing Support Vector Regression (SVR) in Python

Time to put on our coding hats! In this section, we'll understand the use of Support Vector Regression with the help of a dataset. Here, we have to predict the salary of an employee given a few independent variables. A classic HR analytics project!

	User ID	Gender	Age	EstimatedSalary	Purchased
0	15624510	Male	19	19000	0
1	15810944	Male	35	20000	0
2	15668575	Female	26	43000	0
3	15603246	Female	27	57000	0
4	15804002	Male	19	76000	0

Step 1: Importing the libraries

```
import numpy as np
import matplotlib.pyplot as plt
```

```
import pandas as pd
```

view rawsvr1.py hosted with ♥ by GitHub

Step 2: Reading the dataset

```
dataset = pd.read_csv('Position_Salaries.csv')
X = dataset.iloc[:, 1:2].values
y = dataset.iloc[:, 2].values
```

view rawsv2.py hosted with ♥ by GitHub

Step 3: Feature Scaling

A real-world dataset contains features that vary in magnitudes, units, and range. I would suggest performing normalization when the scale of a feature is irrelevant or misleading.

Feature Scaling basically helps to normalize the data within a particular range. Normally several common class types contain the feature scaling function so that they make feature scaling automatically. However, the SVR class is not a commonly used class type so we should perform feature scaling using Python.

```
from sklearn.preprocessing import StandardScaler
sc_X = StandardScaler()
sc_y = StandardScaler()
X = sc_X.fit_transform(X)
y = sc_y.fit_transform(y)
```

view rawsvr3.py hosted with ♥ by GitHub

Step 4: Fitting SVR to the dataset

```
from sklearn.svm import SVR
regressor = SVR(kernel = 'rbf')
regressor.fit(X, y)
```

view rawsvr4.py hosted with ♥ by GitHub

Kernel is the most important feature. There are many types of kernels – linear, Gaussian, etc. Each is used depending on the dataset. To learn more about this, read this: Support Vector Machine (SVM) in Python and R

Step 5. Predicting a new result

```
y_pred = regressor.predict(6.5)
y_pred = sc_y.inverse_transform(y_pred)
```

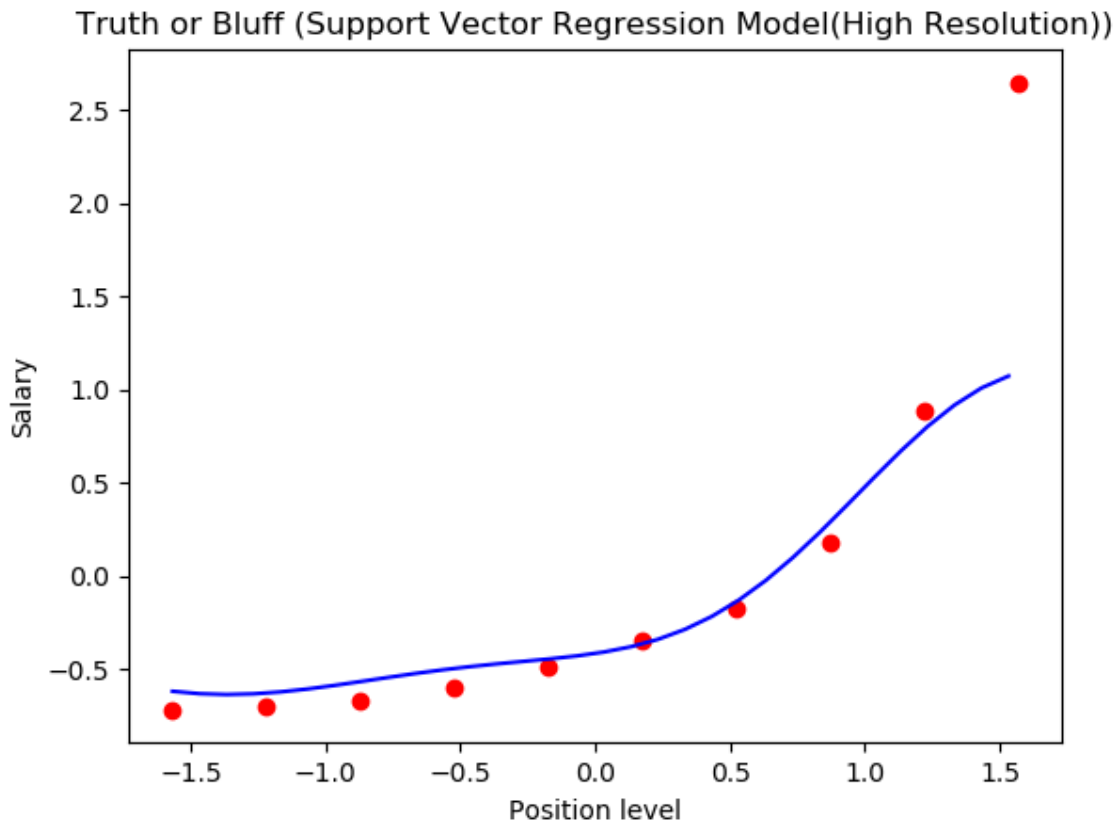
view rawsvr5.py hosted with ♥ by GitHub

So, the prediction for y_pred(6, 5) will be 170,370.

Step 6. Visualizing the SVR results (for higher resolution and smoother curve)

```
X_grid = np.arange(min(X), max(X), 0.01) #this step required because data is feature scaled.
X_grid = X_grid.reshape((len(X_grid), 1))
plt.scatter(X, y, color = 'red')
plt.plot(X_grid, regressor.predict(X_grid), color = 'blue')
plt.title('Truth or Bluff (SVR)')
plt.xlabel('Position level')
plt.ylabel('Salary')
plt.show()
```

view rawsvr6.py hosted with ♥ by GitHub



This is what we get as output- the best fit line that has a maximum number of points.

Quite accurate!