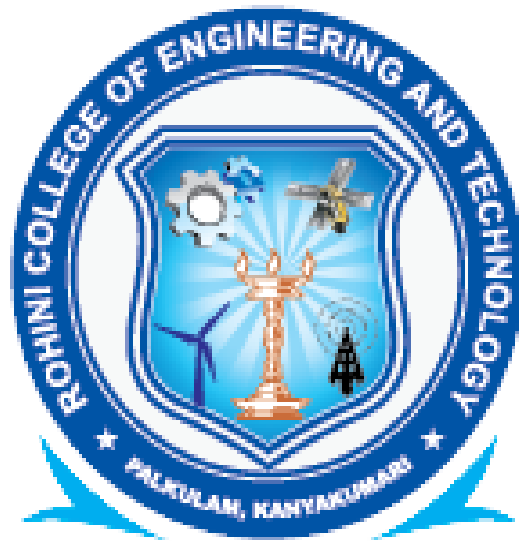# ROHINI COLLEGE OF ENGINEERING AND TECHNOLOGY

Department of Electronics and Communication Engineering

ADD ON MACHINE LEARNING TECHNIQUES

COURSE MATERIAL

Machine Learning

Objectives:

- Acquire theoretical Knowledge on setting hypothesis for pattern recognition.

- Apply suitable machine learning techniques for data handling and to gain knowledge from it.

- Evaluate the performance of algorithms and to provide solution for various real world applications.

UNIT I:
Introduction to Machine Learning

Introduction ,Components of Learning , Learning Models , Geometric Models, Probabilistic Models, Logic Models, Grouping and Grading, Designing a Learning System, Types of Learning, Supervised, Unsupervised, Reinforcement, Perspectives and Issues, Version Spaces, PAC Learning, VC Dimension.

UNIT II:
Supervised and Unsupervised Learning

Decision Trees: ID3, Classification and Regression Trees, Regression: Linear Regression, Multiple Linear Regression, Logistic Regression, Neural Networks: Introduction, Perception, Multilayer Perception, Support Vector Machines: Linear and Non-Linear, Kernel Functions, K Nearest Neighbors.
Introduction to clustering, K-means clustering, K-Mode Clustering.

UNIT III:
Ensemble and Probabilistic Learning

Model Combination Schemes, Voting, Error-Correcting Output Codes, Bagging: Random Forest Trees, Boosting: Adaboost, Stacking.
Gaussian mixture models - The Expectation-Maximization (EM) Algorithm, Information Criteria, Nearest neighbour methods - Nearest Neighbour Smoothing, Efficient Distance Computations: the KD-Tree, Distance Measures.

UNIT IV:
Reinforcement Learning and Evaluating Hypotheses

Introduction, Learning Task, Q Learning, Non deterministic Rewards and actions, temporal-difference learning, Relationship to Dynamic Programming, Active reinforcement learning, Generalization in reinforcement learning.
Motivation, Basics of Sampling Theory: Error Estimation and Estimating Binomial Proportions, The Binomial Distribution, Estimators, Bias, and Variance

UNIT V:
Genetic Algorithms: Motivation, Genetic Algorithms: Representing Hypotheses, Genetic Operator, Fitness Function and Selection, An Illustrative Example, Hypothesis Space Search, Genetic Programming, Models of Evolution and Learning: Lamarkian Evolution, Baldwin Effect, Parallelizing Genetic Algorithms.

TEXT BOOKS:

1. Ethem Alpaydin, "Introduction to Machine Learning", MIT Press, Prentice Hall of India, 3rd Edition2014.
2. Mehryar Mohri, Afshin Rostamizadeh, Ameet Talwalkar " Foundations of Machine Learning", MIT Press,2012.
3. Tom Mitchell, "Machine Learning", McGraw Hill, 3rdEdition, 1997.
4. MACHINE LEARNING - An Algorithmic Perspective, Second Edition, Stephen Marsland, 2015.

REFERENCE BOOKS:
1. CharuC.Aggarwal,"DataClassificationAlgorithmsandApplications",CRCPress,2014.
2. Charu C. Aggarwal, "DATA CLUSTERING Algorithms and Applications", CRC Press, 2014.
3. Kevin P. Murphy "Machine Learning: A Probabilistic Perspective", The MIT Press, 2012
4. Jiawei Han and Micheline Kambers and JianPei, "Data Mining Concepts andTechniques",3rd edition, Morgan Kaufman Publications, 2012.

OUTCOMES:
1. Recognize the characteristics of Machine Learning techniques that enable to solve real world problems
2. Recognize the characteristics of machine learning strategies
3. Apply various supervised learning methods to appropriate problems
4. Identify and integrate more than one techniques to enhance the performance of learning
5. Create probabilistic and unsupervised learning models for handling unknown pattern
6. Analyze the co-occurrence of data to find interesting frequent patterns

INDEX

UNIT I
Introduction to Machine Learning

<u>**1.** **Introduction**</u>

What Is Machine Learning?

Machine learning is programming computers to optimize a performance criterion using example data or past experience. We have a model defined up to some parameters, and learning is the execution of a computer program to optimize the parameters of the model using the training data or past experience. The model may be *predictive* to make predictions in the future, or *descriptive* to gain knowledge from data, or both.

Arthur Samuel, an early American leader in the field of computer gaming and artificial intelligence, coined the term "Machine Learning" in 1959 while at IBM. He defined machine learning as "the field of study that gives computers the ability to learn without being explicitly programmed." However, there is no universally accepted definition for machine learning. Different authors define the term differently.

Definition of learning

Definition

A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks T, as measured by P, improves with experience E.

Examples

i) Handwriting recognition learning problem
- Task T: Recognising and classifying handwritten words within images
- Performance P: Percent of words correctly classified
- Training experience E: A dataset of handwritten words with given classifications

ii) A robot driving learning problem
- Task T: Driving on highways using vision sensors
- Performance measure P: Average distance traveled before an error
- training experience: A sequence of images and steering commands recorded while observing a human driver

iii) A chess learning problem
- Task T: Playing chess
- Performance measure P: Percent of games won against opponents
- Training experience E: Playing practice games against itself

Definition

A computer program which learns from experience is called a machine learning program or simply a learning program. Such a program is sometimes also referred to as a learner.

<u>Components of Learning</u>

Basic components of learning process

The learning process, whether by a human or a machine, can be divided into four components, namely, data storage, abstraction, generalization and evaluation. Figure 1.1 illustrates the variouscomponents and the steps involved in the learning process.
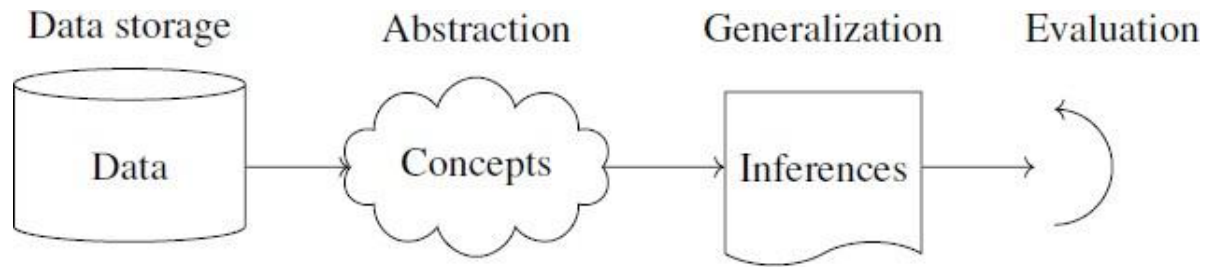
Figure 1.1: Components of learning process

1. Data storage

Facilities for storing and retrieving huge amounts of data are an important component of the learning process. Humans and computers alike utilize data storage as a foundation for advanced reasoning.

• In a human being, the data is stored in the brain and data is retrieved using electrochemical signals.
• Computers use hard disk drives, flash memory, random access memory and similar devices to store data and use cables and other technology to retrieve data.

2. Abstraction

The second component of the learning process is known as abstraction.

Abstraction is the process of extracting knowledge about stored data. This involves creating general concepts about the data as a whole. The creation of knowledge involves application of known models and creation of new models.

The process of fitting a model to a dataset is known as training. When the model has been trained, the data is transformed into an abstract form that summarizes the original information.

3. Generalization

The third component of the learning process is known as generalisation.

The term generalization describes the process of turning the knowledge about stored data into a form that can be utilized for future action. These actions are to be carried out on tasks that are similar, but not identical, to those what have been seen before. In generalization, the goal is to discover those properties of the data that will be most relevant to future tasks.

4. Evaluation

Evaluation is the last component of the learning process.

It is the process of giving feedback to the user to measure the utility of the learned knowledge. This feedback is then utilised to effect improvements in the whole learning process

Applications of machine learning

Application of machine learning methods to large databases is called data mining. In data mining, a large volume of data is processed to construct a simple model with valuable use, for example, having
high predictive accuracy.

The following is a list of some of the typical applications of machine learning.
1. In retail business, machine learning is used to study consumer behaviour.
2. In finance, banks analyze their past data to build models to use in credit applications, fraud detection, and the stock market.
3. In manufacturing, learning models are used for optimization, control, and troubleshooting.

2

4. In medicine, learning programs are used for medical diagnosis.
5. In telecommunications, call patterns are analyzed for network optimization and maximizing the quality of service.
6. In science, large amounts of data in physics, astronomy, and biology can only be analyzed fast enough by computers. The World Wide Web is huge; it is constantly growing and searching for relevant information cannot be done manually.
7. In artificial intelligence, it is used to teach a system to learn and adapt to changes so that the system designer need not foresee and provide solutions for all possible situations.
8. It is used to find solutions to many problems in vision, speech recognition, and robotics.
9. Machine learning methods are applied in the design of computer-controlled vehicles to steer correctly when driving on a variety of roads.
10. Machine learning methods have been used to develop programmes for playing games such as chess, backgammon and Go.

Learning Models

Machine learning is concerned with using the right features to build the right models that achieve the right tasks. The basic idea of Learning models has divided into three categories.
For a given problem, the collection of all possible outcomes represents the **sample space or instance space**.

- Using a Logical expression. (**Logical models**)
- Using the Geometry of the instance space. (**Geometric models)**
- Using Probability to classify the instance space. (**Probabilistic models**)
- Grouping and Grading

Logical models

Logical models use a logical expression to divide the instance space into segments and hence construct grouping models. A **logical expression** is an expression that returns a Boolean value, i.e., a True or False outcome. Once the data is grouped using a logical expression, the data is divided into homogeneous groupings for the problem we are trying to solve. For example, for a classification problem, all the instances in the group belong to one class.

There are mainly two kinds of logical models: **Tree models** and **Rule models**.

Rule models consist of a collection of implications or IF-THEN rules. For tree-based models, the 'if-part' defines a segment and the 'then-part' defines the behaviour of the model for this segment. Rule models follow the same reasoning.

Logical models and Concept learning

To understand logical models further, we need to understand the idea of **Concept Learning**. Concept Learning involves learning logical expressions or concepts from examples. The idea of Concept Learning fits in well with the idea of Machine learning, i.e., inferring a general function from specific training examples. Concept learning forms the basis of both tree-based and rule-based models. More formally, Concept Learning involves acquiring the definition of a general category from a given set of positive and negative training examples of the category. A Formal Definition for Concept Learning is "***The inferring of a Boolean-valued function from training examples of its input and output.***" In concept learning, we only learn a description for the positive class and label everything that doesn't satisfy that description as negative.

The following example explains this idea in more detail.

A Concept Learning Task – Enjoy Sport
Training Examples

Example	Sky	AirTemp	Humidity	Wind	Water	Forecast	EnjoySport
1	Sunny	Warm	Normal	Strong	Warm	Same	YES
2	Sunny	Warm	High	Strong	Warm	Same	YES
3	Rainy	Cold	High	Strong	Warm	Change	NO
4	Sunny	Warm	High	Strong	Warm	Change	YES

ATTRIBUTES   **CONCEPT**

A Concept Learning Task called "Enjoy Sport" as shown above is defined by a set of data from some example days. Each data is described by six attributes. The task is to learn to predict the value of Enjoy Sport for an arbitrary day based on the values of its attribute values. The problem can be represented by a **series of hypotheses**. Each hypothesis is described by a conjunction of constraints on the attributes. The training data represents a set of positive and negative examples of the target function. In the example above, each hypothesis is a vector of six constraints, specifying the values of the six attributes – Sky, AirTemp, Humidity, Wind, Water, and Forecast. The training phase involves learning the set of days (as a conjunction of attributes) for which Enjoy Sport = yes.

Thus, the problem can be formulated as:

- Given instances X which represent a set of all possible days, each described by the attributes:
    - Sky – (values: Sunny, Cloudy, Rainy),
    - AirTemp – (values: Warm, Cold),
    - Humidity – (values: Normal, High),
    - Wind – (values: Strong, Weak),
    - Water – (values: Warm, Cold),
    - Forecast – (values: Same, Change).

Try to identify a function that can predict the target variable Enjoy Sport as yes/no, i.e., 1 or 0.

Geometric models

In the previous section, we have seen that with logical models, such as decision trees, a logical expression is used to partition the instance space. Two instances are similar when they end up in the same logical segment. In this section, we consider models that define similarity by considering the geometry of the instance space. In Geometric models, features could be described as points in two dimensions (*x*- and *y*-axis) or a three-dimensional space (*x*, *y*, and *z*). Even when features are not

intrinsically geometric, they could be modelled in a geometric manner (for example, temperature as a function of time can be modelled in two axes). In geometric models, there are two ways we could impose similarity.

- We could use geometric concepts like **lines or planes to segment (classify)** the instance space. These are called **Linear models**.
- Alternatively, we can use the geometric notion of distance to represent similarity. In this case, if two points are close together, they have similar values for features and thus can be classed as similar. We call such models as **Distance-based models**.

Linear models

Linear models are relatively simple. In this case, the function is represented as a linear combination of its inputs. Thus, if x_1 and x_2 are two scalars or vectors of the same dimension and a and b are arbitrary scalars, then $ax_1 + bx_2$ represents a linear combination of x_1 and x_2. In the simplest case where $f(x)$ represents a straight line, we have an equation of the form $f(x)$ = $mx + c$ where c represents the intercept and m represents the slope.
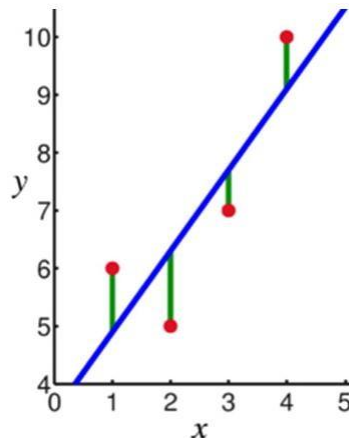

Linear models are **parametric**, which means that they have a fixed form with a small number of numeric parameters that need to be learned from data. For example, in $f(x) = mx + c$, m and c are the parameters that we are trying to learn from the data. This technique is different from tree or rule models, where the structure of the model (e.g., which features to use in the tree, and where) is not fixed in advance.

Linear models are **stable**, i.e., small variations in the training data have only a limited impact on the learned model. In contrast, **tree models tend to vary more with the training data**, as the choice of a different split at the root of the tree typically means that the rest of the tree is different as well. As a result of having relatively few parameters, Linear models have **low variance and high bias**. This implies that **Linear models are less likely to overfit the training data** than some other models. However, they are more likely to underfit. For example, if we want to learn the boundaries between countries based on labelled data, then linear models are not likely to give a good approximation.

Distance-based models

Distance-based models are the second class of Geometric models. Like Linear models, distance-based models are based on the geometry of data. As the name implies, distance-based models work on the concept of distance. In the context of Machine learning, the concept of distance is not based on merely the physical distance between two points. Instead, we could think of the distance between two points considering the **mode of transport** between two points. Travelling between two cities by plane

covers less distance physically than by train because a plane is unrestricted. Similarly, in chess, the concept of distance depends on the piece used – for example, a Bishop can move diagonally. Thus, depending on the entity and the mode of travel, the concept of distance can be experienced differently. The distance metrics commonly used are **Euclidean**, **Minkowski**, **Manhattan**, and **Mahalanobis**.
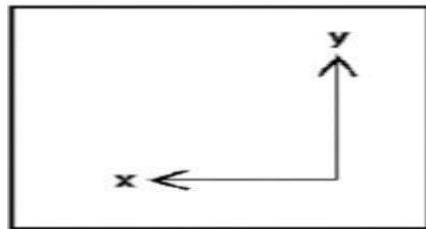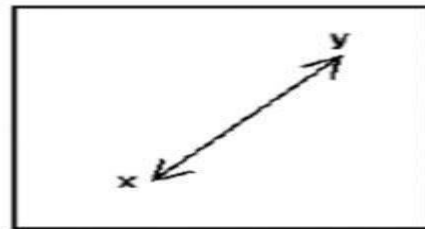
Distance is applied through the concept of **neighbours and exemplars**. Neighbours are points in proximity with respect to the distance measure expressed through exemplars. Exemplars are either **centroids** that find a centre of mass according to a chosen distance metric or **medoids** that find the most centrally located data point. The most commonly used centroid is the arithmetic mean, which minimises squared Euclidean distance to all other points.

Notes:
- The **centroid** represents the geometric centre of a plane figure, i.e., the arithmetic mean position of all the points in the figure from the centroid point. This definition extends to any object in *n*-dimensional space: its centroid is the mean position of all the points.
- **Medoids** are similar in concept to means or centroids. Medoids are most commonly used on data when a mean or centroid cannot be defined. They are used in contexts where the centroid is not representative of the dataset, such as in image data.

Examples of distance-based models include the **nearest-neighbour** models, which use the training data as exemplars – for example, in classification. The **K-means clustering** algorithm also uses exemplars to create clusters of similar data points.

Probabilistic models
The third family of machine learning algorithms is the probabilistic models. We have seen before that the k-nearest neighbour algorithm uses the idea of distance (e.g., Euclidian distance) to classify entities, and logical models use a logical expression to partition the instance space. In this section, we see how the **probabilistic models use the idea of probability to classify new entities.**

Probabilistic models see features and target variables as random variables. The process of modelling represents and **manipulates the level of uncertainty** with respect to these variables. There are two types of probabilistic models: **Predictive and Generative**. Predictive probability models use the idea of a **conditional probability** distribution $P(Y|X)$ from which Y can be predicted from X. Generative models estimate the **joint distribution** $P(Y, X)$. Once we know the joint distribution for the generative models, we can derive any conditional or marginal distribution involving the same variables. Thus, the generative model is capable of creating new data points and their labels, knowing the joint probability distribution. The joint distribution looks for a relationship between two variables. Once this relationship is inferred, it is possible to infer new data points.
Naïve Bayes is an example of a probabilistic classifier.

We can do this using the **Bayes rule** defined as

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

The Naïve Bayes algorithm is based on the idea of **Conditional Probability. Conditional probability is based on finding the** probability that something will happen, *given that something else* has already happened. The task of the algorithm then is to look at the evidence and to determine the likelihood of a specific class and assign a label accordingly to each entity.

Some broad categories of models:

| Geometric models	Probabilistic models	Logical models
E.g. K-nearest neighbors, linear regression, support vector machine, logistic regression, …	Naïve Bayes, Gaussian process regression, conditional random field, …	Decision tree, random forest, …

Grouping and Grading

Grading vs grouping is an orthogonal categorization to geometric-probabilistic-logical-compositional.

- Grouping models break the instance space up into groups or segments and in each segment apply a very simple method (such as majority class).
    - E.g. decision tree, KNN.
- Grading models form one global model over the instance space.
    - E.g. Linear classifiers – Neural networks

Designing a Learning System

For any learning system, we must be knowing the three elements — **T (Task)**, **P (Performance Measure)**, and **E (Training Experience)**. At a high level, the process of learning system looks as below.
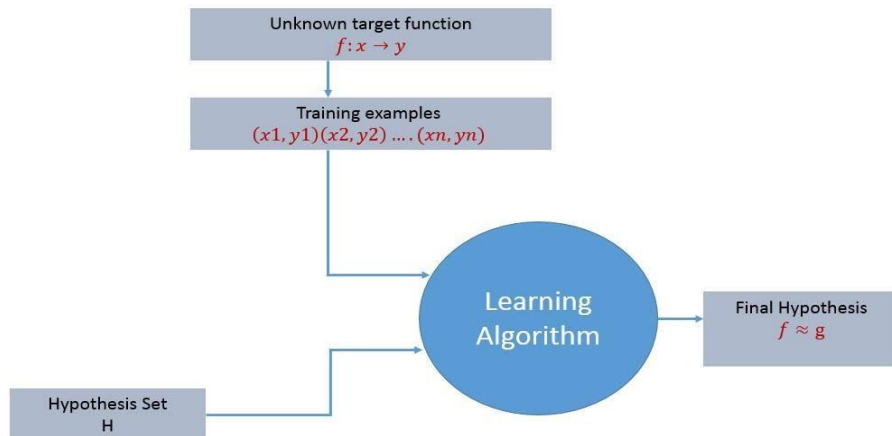
The learning process starts with task T, performance measure P and training experience E and objective are to find an unknown target function. The target function is an exact knowledge to be learned from the training experience and its unknown. For example, in a case of credit approval, the learning system will have customer application records as experience and task would be to classify whether the given customer application is eligible for a loan. So in this case, the training examples can be represented as

(x1,y1)(x2,y2)..(xn,yn) where X represents customer application details and y represents the status of credit approval.

With these details, what is that exact knowledge to be learned from the training experience?

So the target function to be learned in the credit approval learning system is a mapping function f:X →y. This function represents the exact knowledge defining the relationship between input variable X and output variable y.

Design of a learning system

Just now we looked into the learning process and also understood the goal of the learning. When we want to design a learning system that follows the learning process, we need to consider a few design choices. The design choices will be to decide the following key components:

1. **Type of training experience**
2. **Choosing the Target Function**
3. **Choosing a representation for the Target Function**
4. **Choosing an approximation algorithm for the Target Function**
5. **The final Design**

We will look into the game - checkers learning problem and apply the above design choices. For a checkers learning problem, the three elements will be,

   1. Task T: To play checkers
   2. Performance measure P: Total percent of the game won in the tournament.
   3. Training experience E: A set of games played against itself

Type of training experience
   During the design of the checker's learning system, the type of training experience available for a learning system will have a significant effect on the success or failure of the learning.

1. **Direct or Indirect training experience —** In the case of direct training experience, an individual board states and correct move for each board state are given. In case of indirect training experience, the move sequences for a game and the final result (win, loss or draw) are given for a number of games. How to assign credit or blame to individual moves is the credit assignment problem.

2. **Teacher or Not —** Supervised — The training experience will be labeled, which means, all the board states will be labeled with the correct move. So the learning takes place in the presence of a supervisor or a teacher. Unsupervised — The training experience will be unlabeled, which means, all the board states will not have the moves. So the learner generates random games and plays against itself with no supervision or teacher involvement.

Semi-supervised — Learner generates game states and asks the teacher for help in finding the correct move if the board state is confusing.

3. **Is the training experience good —** Do the training examples represent the distribution of examples over which the final system performance will be measured? Performance is best when training examples and test examples are from the same/a similar distribution.

The checker player learns by playing against oneself. Its experience is indirect. It may not encounter moves that are common in human expert play. Once the proper training experience is available, the next design step will be choosing the Target Function.
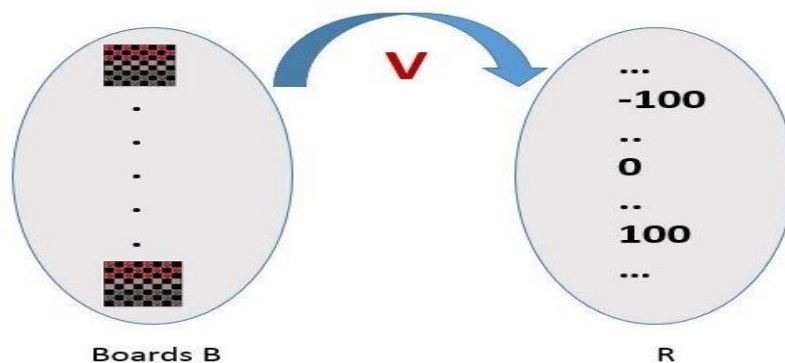
Choosing the Target Function
When you are playing the checkers game, at any moment of time, you make a decision on choosing the best move from different possibilities. You think and apply the learning that you have gained from the experience. Here the learning is, for a specific board, you move a checker such that your board state tends towards the winning situation. Now the same learning has to be defined in terms of the target function.

Here there are 2 considerations — direct and indirect experience.

- **During the direct experience**, the checkers learning system, it needs only to learn how to choose the best move among some large search space. We need to find a target function that will help us choose the best move among alternatives. Let us call this function ChooseMove and use the notation **ChooseMove : B →M** to indicate that this function accepts as input any board from the set of legal board states B and produces as output some move from the set of legal moves M.
- **When there is an indirect experience**, it becomes difficult to learn such function. How about assigning a real score to the board state.

So the function be **V : B →R** indicating that this accepts as input any board from the set of legal board states B and produces an output a real score. This function assigns the higher scores to better board states.

Boards B   R

If the system can successfully learn such a target function V, then it can easily use it to select the best move from any board position.

Let us therefore define the target value V(b) for an arbitrary board state b in B, as follows:

1. if b is a final board state that is won, then V(b) = 100
2. if b is a final board state that is lost, then V(b) = -100
3. if b is a final board state that is drawn, then V(b) = 0
4. if b is a not a final state in the game, then V (b) = V (b'), where b' is the best final board state that can be achieved starting from b and playing optimally until the end of the game.

The (4) is a recursive definition and to determine the value of V(b) for a particular board state, it performs the search ahead for the optimal line of play, all the way to the end of the game. So this definition is not efficiently computable by our checkers playing program, we say that it is a nonoperational definition.

The goal of learning, in this case, is to discover an operational description of V ; that is, a description that can be used by the checkers-playing program to evaluate states and select moves within realistic time bounds.
It may be very difficult in general to learn such an operational form of V perfectly. We expect learning algorithms to acquire only some approximation to the target function ^V.

Choosing a representation for the Target Function

Now that we have specified the ideal target function V, we must choose a representation that the learning program will use to describe the function ^V that it will learn. As with earlier design choices, we again have many options. We could, for example, allow the program to represent using a large table with a distinct entry specifying the value for each distinct board state. Or we could allow it to represent using a collection of rules that match against features of the board state, or a quadratic polynomial function of predefined board features, or an artificial
neural network. In general, this choice of representation involves a crucial tradeoff. On one hand, we wish to pick a very expressive representation to allow representing as close an approximation as possible to the ideal target function V.

On the other hand, the more expressive the representation, the more training data the program will require in order to choose among the alternative hypotheses it can represent. To keep the discussion brief, let us choose a simple representation:
for any given board state, the function ^V will be calculated as a linear combination of the following board features:

- $x_1(b)$ — number of black pieces on board b
- $x_2(b)$ — number of red pieces on b
- $x_3(b)$ — number of black kings on b
- $x_4(b)$ — number of red kings on b
- $x_5(b)$ — number of red pieces threatened by black (i.e., which can be taken on black's next turn)
- $x_6(b)$ — number of black pieces threatened by red

^V = w_0 + $w_1 \cdot x_1(b)$ + $w_2 \cdot x_2(b)$ + $w_3 \cdot x_3(b)$ + $w_4 \cdot x_4(b)$ +$w_5 \cdot x_5(b)$ + $w_6 \cdot x_6(b)$

Where w_0 through w_6 are numerical coefficients or weights to be obtained by a learning algorithm.
Weights w_1 to w_6 will determine the relative importance of different board features.

Specification of the Machine Learning Problem at this time — Till now we worked on choosing the type of training experience, choosing the target function and its representation. The checkers learning task can be summarized as below.

- **Task T : Play Checkers**
- **Performance Measure : % of games won in world tournament**
- **Training Experience E : opportunity to play against itself**
- **Target Function : V : Board → R**
- **Target Function Representation : ^V = w0 + w1 · x1(b) + w2 · x2(b) + w3 · x3(b) + w4 · x4(b) +w5 · x5(b) + w6 · x6(b)**

The first three items above correspond to the specification of the learning task,whereas the final two items constitute design choices for the implementation of the learning program.

Choosing an approximation algorithm for the Target Function

Generating training data —

To train our learning program, we need a set of training data, each describing a specific board state b and the training value V_train (b) for b. Each training example is an ordered pair <b,V_train(b)>

For example, a training example may be <(x1 = 3, x2 = 0, x3 = 1, x4 = 0, x5 = 0, x6 = 0), +100">. This is an example where black has won the game since x2 = 0 or red has no remaining pieces. However, such clean values of V_train (b) can be obtained only for board value b that are clear win, loss or draw.

In above case, assigning a training value V_train(b) for the specific boards b that are clean win, loss or draw is direct as they are direct training experience. But in the case of indirect training experience, assigning a training value V_train(b) for the intermediate boards is difficult. In such case, the training values are updated using temporal difference learning. **Temporal difference (TD) learning is a concept central to reinforcement learning, in which learning happens through the iterative correction of your estimated returns towards a more accurate target return.**

Let Successor(b) denotes the next board state following b for which it is again the program's turn to move. ^V is the learner's current approximation to V. Using these information, assign the training value of V_train(b) for any intermediate board state b as below :

$$V\_train(b) \leftarrow \text{^V(Successor(b))}$$

Adjusting the weights

Now its time to define the learning algorithm for choosing the weights and best fit the set of training examples. One common approach is to define the best hypothesis as that which minimizes the squared error E between the training values and the values predicted by the hypothesis ^V.

$$E \equiv \sum_{\langle b, V_{train}(b) \rangle \in training\ examples} (V_{train}(b) - \widehat{V}(b))^2$$

The learning algorithm should incrementally refine weights as more training examples become available and it needs to be robust to errors in training data Least Mean Square (LMS) training rule is the one training algorithm that will adjust weights a small amount in the direction that reduces the error.

The LMS algorithm is defined as follows:

For each training example b
1. Compute $error(b) = V_{train}(b) - \hat{V}(b)$
2. for each board feature x_i, update weight w_i
$$w_i \leftarrow w_i + \mu \cdot error(b) \cdot x_i$$
Here μ is a small constant (e.g., 0.1) that moderates the size of the weight update

Final Design for Checkers Learning system

The final design of our checkers learning system can be naturally described by four distinct program modules that represent the central components in many learning systems.

1. The performance System — Takes a new board as input and outputs a trace of the game it played against itself.
2. The Critic — Takes the trace of a game as an input and outputs a set of training examples of the target function.
3. The Generalizer — Takes training examples as input and outputs a hypothesis that estimates the target function. Good generalization to new cases is crucial.
4. The Experiment Generator — Takes the current hypothesis (currently learned function) as input and outputs a new problem (an initial board state) for the performance system to explore.
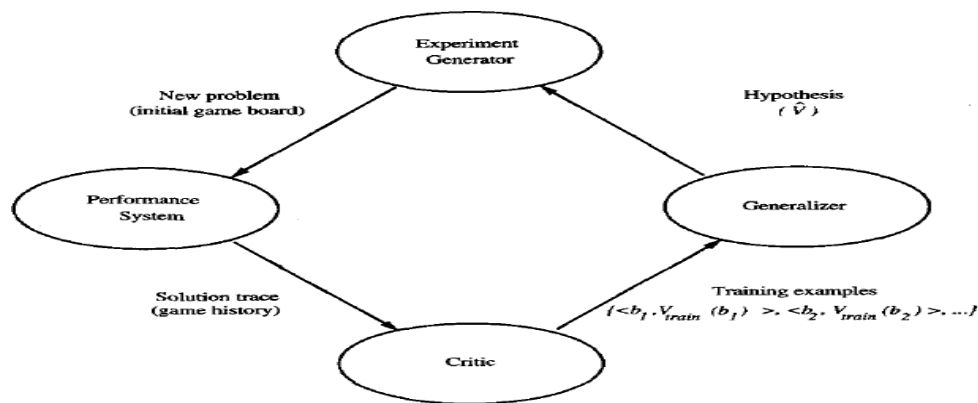
Final design of the checkers learning program.

Types of Learning

In general, machine learning algorithms can be classified into three types.
- Supervised learning
- Unsupervised learning
- Reinforcement learning

Supervised learning

A training set of examples with the correct responses (targets) is provided and, based on this training set, the algorithm generalises to respond correctly to all possible inputs. This is also called learning from exemplars. Supervised learning is the machine learning task of learning a function that maps an input to an output based on example input-output pairs.

In supervised learning, each example in the training set is a pair consisting of an input object (typically a vector) and an output value. A supervised learning algorithm analyzes the training data and produces a function, which can be used for mapping new examples. In the optimal case, the function will correctly determine the class labels for unseen instances. Both classification and regression

12

problems are supervised learning problems. A wide range of supervised learning algorithms are available, each with its strengths and weaknesses. There is no single learning algorithm that works best on all supervised learning problems.
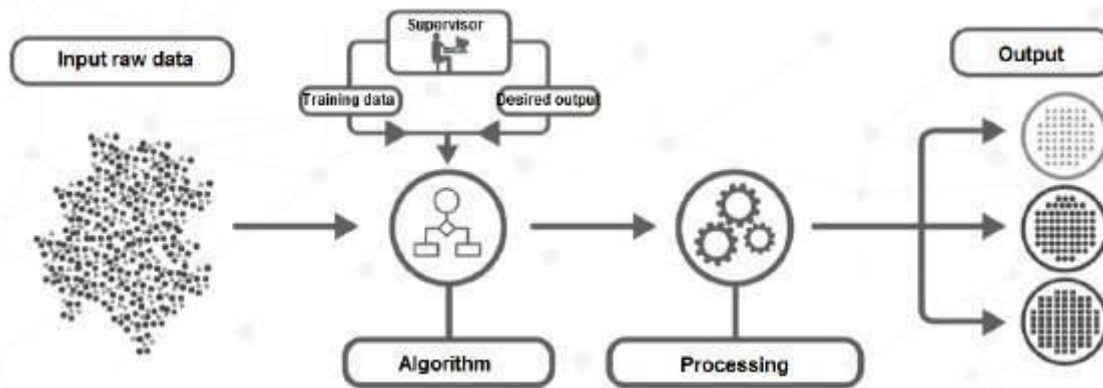

Figure 1.4: Supervised learning

Remarks

A "supervised learning" is so called because the process of an algorithm learning from the training dataset can be thought of as a teacher supervising the learning process. We know the correct answers (that is, the correct outputs), the algorithm iteratively makes predictions on the training data and is corrected by the teacher. Learning stops when the algorithm achieves an acceptable level of performance.

Example

Consider the following data regarding patients entering a clinic. The data consists of the gender and age of the patients and each patient is labeled as "healthy" or "sick".

gender	age	label
M	48	sick
M	67	sick
F	53	healthy
M	49	healthy
F	34	sick
M	21	healthy

Unsupervised learning

Correct responses are not provided, but instead the algorithm tries to identify similarities between the inputs so that inputs that have something in common are categorised together. The statistical approach to unsupervised learning is
known as density estimation.

Unsupervised learning is a type of machine learning algorithm used to draw inferences from datasets consisting of input data without labeled responses. In unsupervised learning algorithms, a classification or categorization is not included in the observations. There are no output values and so there is no estimation of functions. Since the examples given to the learner are unlabeled, the accuracy of the structure that is output by the algorithm cannot be evaluated. The most common unsupervised learning method is cluster analysis, which is used for exploratory data analysis to find hidden patterns

13

or grouping in data.

Example

Consider the following data regarding patients entering a clinic. The data consists of the gender and age of the patients.

gender	age
M	48
M	67
F	53
M	49
F	34
M	21

Based on this data, can we infer anything regarding the patients entering the clinic?

Reinforcement learning

This is somewhere between supervised and unsupervised learning. The algorithm gets told when the answer is wrong, but does not get told how to correct it. It has to explore and try out different possibilities until it works out how to get the answer right. Reinforcement learning is sometime called learning with a critic because of this monitor that scores the answer, but does not suggest improvements.

Reinforcement learning is the problem of getting an agent to act in the world so as to maximize its rewards. A learner (the program) is not told what actions to take as in most forms of machine learning, but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situations and, through that, all subsequent rewards.

Example

Consider teaching a dog a new trick: we cannot tell it what to do, but we can reward/punish it if it does the right/wrong thing. It has to find out what it did that made it get the reward/punishment. We can use a similar method to train computers to do many tasks, such as playing backgammon or chess, scheduling jobs, and controlling robot limbs. Reinforcement learning is different from supervised learning. Supervised learning is learning from examples provided by a knowledgeable expert.

PERSPECTIVES AND ISSUES IN MACHINE LEARNING

Perspectives in Machine Learning

One useful perspective on machine learning is that it involves searching a very large space of possible hypotheses to determine one that best fits the observed data and any prior knowledge held by the learner.

For example, consider the space of hypotheses that could in principle be output by the above checkers learner. This hypothesis space consists of all evaluation functions that can be represented by some choice of values for the weights *wo* through *w6.* The learner's task is thus to search through this vast space to locate the hypothesis that is most consistent with the available training examples. The LMS algorithm for fitting weights achieves this goal by iteratively tuning the weights, adding a correction to each weight each time the hypothesized evaluation function predicts a value that differs from the training value. This algorithm works well when the hypothesis representation considered by the learner defines a continuously parameterized space of potential hypotheses.

Many of the chapters in this book present algorithms that search a hypothesis space defined by some underlying representation (e.g., linear functions, logical descriptions, decision trees, artificial neural networks). These different hypothesis representations are appropriate for learning different kinds of target functions. For each of these hypothesis representations, the corresponding learning algorithm takes advantage of a different underlying structure to organize the search through the hypothesis space.

Throughout this book we will return to this perspective of learning as a search problem in order to characterize learning methods by their search strategies and by the underlying structure of the search spaces they explore. We will also find this viewpoint useful in formally analyzing the relationship between the size of the hypothesis space to be searched, the number of training examples available, and the confidence we can have that a hypothesis consistent with the training data will correctly generalize to unseen examples.

Issues in Machine Learning
Our checkers example raises a number of generic questions about machine learning. The field of machine learning, and much of this book, is concerned with answering questions such as the following:

- What algorithms exist for learning general target functions from specific training examples? In what settings will particular algorithms converge to the desired function, given sufficient training data? Which algorithms perform best for which types of problems and representations?
- How much training data is sufficient? What general bounds can be found to relate the confidence in learned hypotheses to the amount of training experience and the character of the learner's hypothesis space?
- When and how can prior knowledge held by the learner guide the process of generalizing from examples? Can prior knowledge be helpful even when it is only approximately correct?
- What is the best strategy for choosing a useful next training experience, and how does the choice of this strategy alter the complexity of the learning problem?
- What is the best way to reduce the learning task to one or more function approximation problems? Put another way, what specific functions should the system attempt to learn? Can this process itself be automated?
- How can the learner automatically alter **its** representation to improve its ability to represent and learn the target function?

Version Spaces
Definition (Version space). *A concept is* complete *if it covers all positive examples.*

A concept is consistent *if it covers none of the negative examples. The* version space *is the set of all complete and consistent concepts. This set is convex and is fully defined by its least and most general elements.*

The key idea in the CANDIDATE-ELIMINATION algorithm is to output a description of the set of all *hypotheses consistent with the training examples*

Representation
The Candidate – Elimination algorithm finds all describable hypotheses that are consistent with the

observed training examples. In order to define this algorithm precisely, we begin with a few basic definitions. First, let us say that a hypothesis is **consistent** with the training examples if it correctly classifies these examples.

Definition: A hypothesis h is **consistent** with **a** set of training examples **D** if **and** only if **h(x)** = **c(x)** for each example (x, **c(x))** in **D.**

$$Consistent\ (h,\ D) \equiv (\forall\ \langle x,\ c(x) \rangle \in D)\ h(x) = c(x))$$

Note difference between definitions of *consistent* and *satisfies*

- An example *x* is said to **satisfy** hypothesis **h** when **h(x)** = 1, regardless of whether *x* is a positive or negative example of the target concept.

- An example *x* is said to **consistent** with hypothesis **h** iff **h(x)** = **c(x)**

Definition: version space- The version space, denoted *V SH, D* with respect to hypothesis space *H* and training examples D, is the subset of hypotheses from *H* consistent with the training examples in D

$$V S_{H,\ D} \equiv \{h \in H \mid Consistent\ (h,\ D)\}$$

The LIST-THEN-ELIMINATION algorithm
The LIST-THEN-ELIMINATE algorithm first initializes the version space to contain all hypotheses in H and then eliminates any hypothesis found inconsistent with any training example.

1. *VersionSpace c* a list containing every hypothesis in H
2. For each training example, (x, c(x)) remove from *VersionSpace* any hypothesis h for which h(x) ≠ c(x)
3. Output the list of hypotheses in *VersionSpace*

- List-Then-Eliminate works in principle, so long as version space is finite.
- However, since it requires exhaustive enumeration of all hypotheses in practice it is not feasible.

A More Compact Representation for Version Spaces
The version space is represented by its most general and least general members. These members form general and specific boundary sets that delimit the version space within the partially ordered hypothesis space.
Definition: The **general boundary** G, with respect to hypothesis space *H* and training data *D,* is the set of maximally general members of *H* consistent with *D*

$$G \equiv \{g \in H \mid Consistent\ (g,\ D) \wedge (\neg \exists g' \in H)[(g' >_g g) \wedge Consistent(g',\ D)]\}$$

Definition: The **specific boundary** S, with respect to hypothesis space *H* and training data *D,* is the set of minimally general (i.e., maximally specific) members of *H* consistent with *D.*

$$S \equiv \{s \in H \mid Consistent\ (s,\ D) \wedge (\neg \exists s' \in H)[(s >_g s') \wedge Consistent(s',\ D)]\}$$

Theorem: Version Space representation theorem
Theorem: Let X be an arbitrary set of instances and Let H be a set of Boolean-valued hypotheses defined over X. Let c: X →{O, 1} be an arbitrary target concept defined over X, and let D be an arbitrary set of training examples

$\{(x, c(x)))$. For all X, H, c, and D such that S and G are well defined,

$$VS_{H,D} = \{\, h \in H \mid (\exists s \in S\,)\,(\exists g \in G\,)\,(\, g \geq_g h \geq_g s \,) \}$$

<u>To Prove:</u>

1. Every h satisfying the right hand side of the above expression is in $VS_{H,D}$

2. Every member of $VS_{H,D}$ satisfies the right-hand side of the expression

<u>Sketch of proof:</u>

1. let g, h, s be arbitrary members of G, H, S respectively with $g \geq_g h \geq_g s$

- By the definition of *S,* s must be satisfied by all positive examples in D. Because $h \geq_g s,$ h must also be satisfied by all positive examples in D.

- By the definition of *G,* g cannot be satisfied by any negative example in D, and because $g \geq_g h$ h cannot be satisfied by any negative example in D. Because h is satisfied by all positive examples in D and by no negative examples in D, h is consistent with D, and therefore h is a member of $VS_{H,D}$.

2. It can be proven by assuming some h in $VS_{H,D}$, that does not satisfy the right-hand side of the expression, then showing that this leads to an inconsistency

CANDIDATE-ELIMINATION Learning Algorithm

The CANDIDATE-ELIMINTION algorithm computes the version space containing all hypotheses from H that are consistent with an observed sequence of training examples.

Initialize G to the set of maximally general hypotheses in H Initialize S to the set of maximally specific hypotheses in H For each training example d, do
- If d is a positive example
    - Remove from G any hypothesis inconsistent with d
    - For each hypothesis s in S that is not consistent with d
        - Remove s from S
        - Add to S all minimal generalizations h of s such that
            - h is consistent with d, and some member of G is more general than h
        - Remove from S any hypothesis that is more general than another hypothesis in S

- If d is a negative example
    - Remove from S any hypothesis inconsistent with d
    - For each hypothesis g in G that is not consistent with d
        - Remove g from G

- Add to G all minimal specializations h of g such that
    - h is consistent with d, and some member of S is more specific than h
- Remove from G any hypothesis that is less general than another hypothesis in G

CANDIDATE- ELIMINTION algorithm using version spaces

An Illustrative Example

Example	Sky	AirTemp	Humidity	Wind	Water	Forecast	EnjoySport
1	Sunny	Warm	Normal	Strong	Warm	Same	Yes
2	Sunny	Warm	High	Strong	Warm	Same	Yes
3	Rainy	Cold	High	Strong	Warm	Change	No
4	Sunny	Warm	High	Strong	Cool	Change	Yes

CANDIDATE-ELIMINTION algorithm begins by initializing the version space to the set of all hypotheses in H;

Initializing the G boundary set to contain the most general hypothesis in H
$$G_0 \ \langle ?, ?, ?, ?, ?, ? \rangle$$

Initializing the S boundary set to contain the most specific (least general) hypothesis
$$S_0 \ \langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$$

- When the first training example is presented, the CANDIDATE-ELIMINTION algorithm checks the S boundary and finds that it is overly specific and it fails to cover the positive example.
- The boundary is therefore revised by moving it to the least more general hypothesis that covers this new example
- No update of the G boundary is needed in response to this training example because G_0 correctly covers this example

For training example d,

⟨Sunny, Warm, Normal, Strong, Warm, Same⟩ +

S_0 ⟨∅, ∅, ∅, ∅, ∅. ∅⟩

S_1 ⟨Sunny, Warm, Normal, Strong, Warm, Same⟩
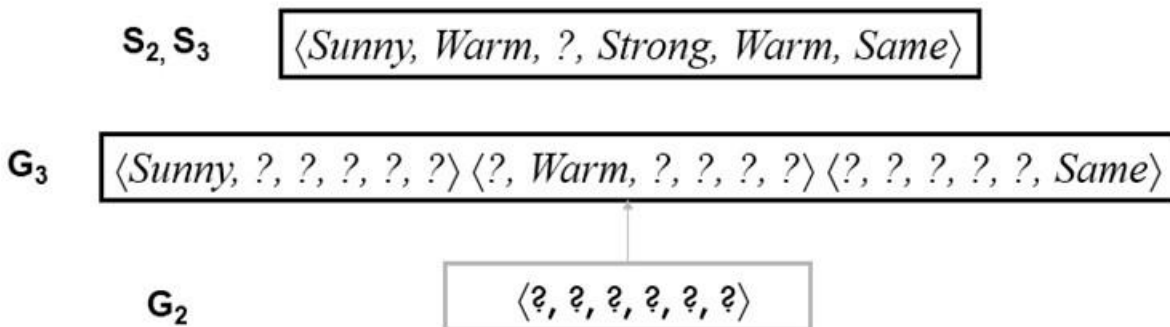
G_0, G_1 ⟨?, ?, ?, ?, ?, ?⟩
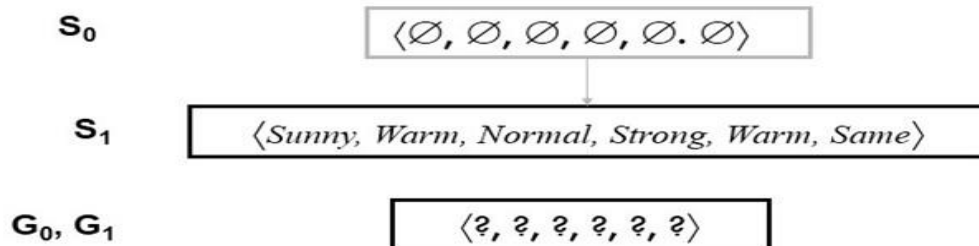
- When the second training example is observed, it has a similar effect of generalizing S further to S_2, leaving G again unchanged i.e., $G_2 = G_1 = G_0$

18

For training example d,

$\langle$ Sunny, Warm, High, Strong, Warm, Same $\rangle$ +

S_1 $\langle$ Sunny, Warm, Normal, Strong, Warm, Same $\rangle$

S_2 $\langle$ Sunny, Warm, ?, Strong, Warm, Same $\rangle$

G_1, G_2 $\langle$?, ?, ?, ?, ?, ? $\rangle$
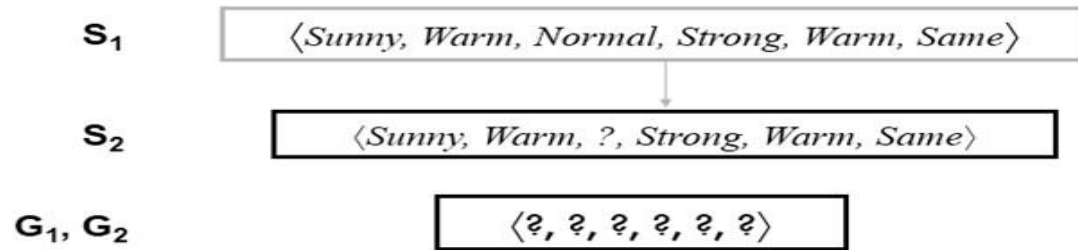
- Consider the third training example. This negative example reveals that the G boundary of the version space is overly general, that is, the hypothesis in G incorrectly predicts that this new example is a positive example.
- The hypothesis in the G boundary must therefore be specialized until it correctly classifies this new negative example.

For training example d,

$\langle$ Rainy, Cold, High, Strong, Warm, Change $\rangle$ −

S_2, S_3 $\langle$ Sunny, Warm, ?, Strong, Warm, Same $\rangle$

G_3 $\langle$ Sunny, ?, ?, ?, ?, ? $\rangle$ $\langle$?, Warm, ?, ?, ?, ? $\rangle$ $\langle$?, ?, ?, ?, ?, Same $\rangle$

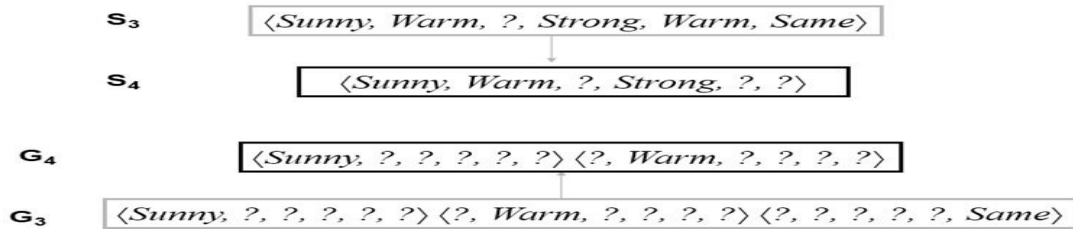G_2 $\langle$?, ?, ?, ?, ?, ? $\rangle$

Given that there are six attributes that could be specified to specialize G2, why are there only three new hypotheses in G3?

> For example, the hypothesis h = (?, ?, Normal, ?, ?, ?) is a minimal specialization of G2 that correctly labels the new example as a negative example, but it is not included in G3. The reason this hypothesis is excluded is that it is inconsistent with the previously encountered positive examples

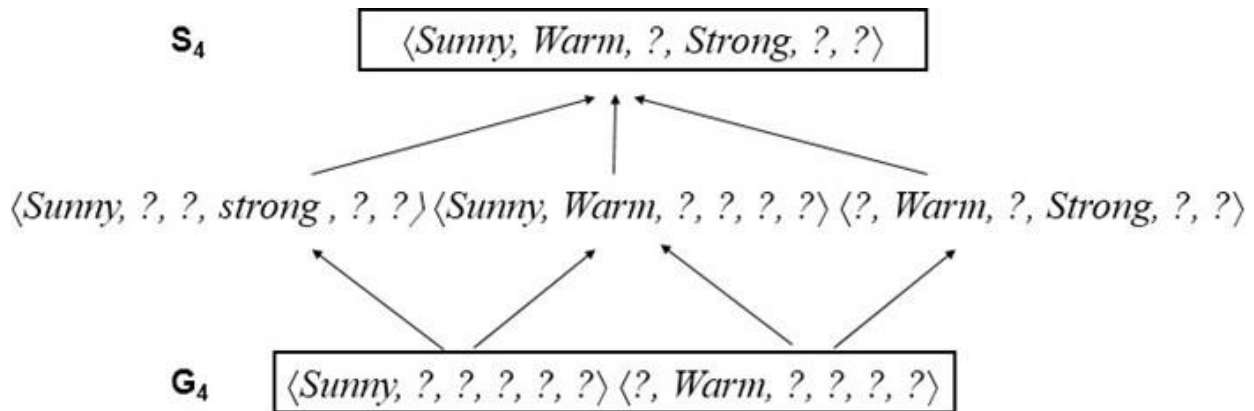Consider the fourth training example.

For training example d,

$$\langle Sunny, Warm, High, Strong, Cool Change \rangle +$$

S_3 $\langle Sunny, Warm, ?, Strong, Warm, Same \rangle$

S_4 $\langle Sunny, Warm, ?, Strong, ?, ? \rangle$

G_4 $\langle Sunny, ?, ?, ?, ?, ? \rangle \langle ?, Warm, ?, ?, ?, ? \rangle$

G_3 $\langle Sunny, ?, ?, ?, ?, ? \rangle \langle ?, Warm, ?, ?, ?, ? \rangle \langle ?, ?, ?, ?, ?, Same \rangle$

- This positive example further generalizes the S boundary of the version space. It also results in removing one member of the G boundary, because this member fails to cover the new positive example

After processing these four examples, the boundary sets S4 and G4 delimit the version space of all hypotheses consistent with the set of incrementally observed training examples.

S_4 $\langle Sunny, Warm, ?, Strong, ?, ? \rangle$

$\langle Sunny, ?, ?, strong, ?, ? \rangle \langle Sunny, Warm, ?, ?, ?, ? \rangle \langle ?, Warm, ?, Strong, ?, ? \rangle$

G_4 $\langle Sunny, ?, ?, ?, ?, ? \rangle \langle ?, Warm, ?, ?, ?, ? \rangle$

Probably approximately correct learning

In computer science, computational learning theory (or just learning theory) is a subfield of artificial intelligence devoted to studying the design and analysis of machine learning algorithms. In computational learning theory, probably approximately correct learning (PAC learning) is a framework for mathematical analysis of machine learning algorithms. It was proposed in 1984 by Leslie Valiant.

In this framework, the learner (that is, the algorithm) receives samples and must select a hypothesis from a certain class of hypotheses. The goal is that, with high probability (the "probably" part), the selected hypothesis will have low generalization error (the "approximately correct" part). In this section we first give an informal definition of PAC-learnability. After introducing a few nore notions, we give a more formal, mathematically oriented, definition of PAC-learnability. At the end, we mention one of the applications of PAC-learnability.

PAC-learnability
To define PAC-learnability we require some specific terminology and related notations.

- Let X be a set called the instance space which may be finite or infinite. For example, X may be the set of all points in a plane.
- A concept class C for X is a family of functions c : X → {0; 1}. A member of C is called a concept. A concept can also be thought of as a subset of X. If C is a subset of X, it defines a unique function μ_c : X → {0; 1} as follows:

$$\mu_C(x) = \begin{cases} 1 & \text{if } x \in C \\ 0 & \text{otherwise} \end{cases}$$

- A hypothesis h is also a function h : X → {0; 1}. So, as in the case of concepts, a hypothesis can also be thought of as a subset of X. H will denote a set of hypotheses.
- We assume that F is an arbitrary, but fixed, probability distribution over X.
- Training examples are obtained by taking random samples from X. We assume that the samples are randomly generated from X according to the probability distribution F.

Now, we give below an informal definition of PAC-learnability.

Definition (informal)
Let X be an instance space, C a concept class for X, h a hypothesis in C and F an arbitrary, but fixed, probability distribution. The concept class C is said to be PAC-learnable if there is an algorithm A which, for samples drawn with any probability distribution F and any concept c ∈ C, will with high probability produce a hypothesis h ∈ C whose error is small.

Examples

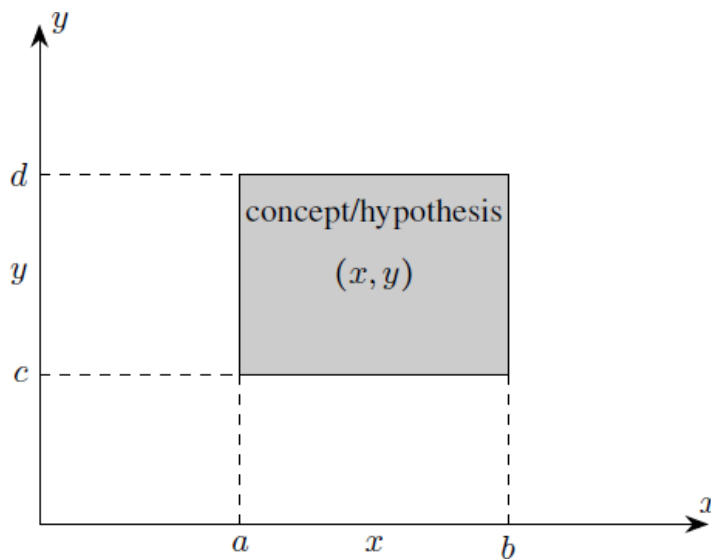To illustrate the definition of PAC-learnability, let us consider some concrete examples.

Figure : An axis-aligned rectangle in the Euclidean plane

Example

- Let the instance space be the set X of all points in the Euclidean plane. Each point is represented by its coordinates (x; y). So, the dimension or length of the instances is 2.
- Let the concept class C be the set of all "axis-aligned rectangles" in the plane; that is, the set of all rectangles whose sides are parallel to the coordinate axes in the plane (see Figure).
- Since an axis-aligned rectangle can be defined by a set of inequalities of the following form having four parameters

    a ≤ x ≤ b, c ≤ y ≤ d

    the size of a concept is 4.
- We take the set H of all hypotheses to be equal to the set C of concepts, H = C.

Given a set of sample points labeled positive or negative, let L be the algorithm which outputs the hypothesis defined by the axis-aligned rectangle which gives the tightest fit to the positive examples (that is, that rectangle with the smallest area that includes all of the positive examples and none of the negative examples) (see Figure bleow).
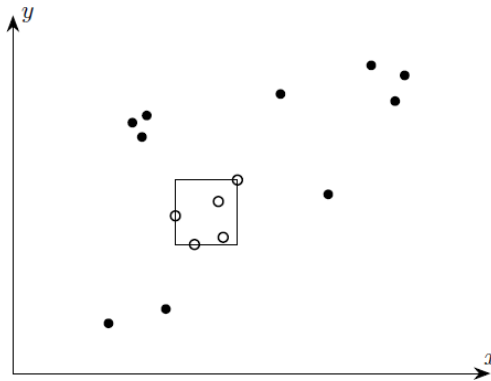

Figure : Axis-aligned rectangle which gives the tightest fit to the positive examples

It can be shown that, in the notations introduced above, the concept class C is PAC-learnable by the algorithm L using the hypothesis space H of all axis-aligned rectangles.

Vapnik-Chervonenkis (VC) dimension
The concepts of Vapnik-Chervonenkis dimension (VC dimension) and probably approximate correct (PAC) learning are two important concepts in the mathematical theory of learnability and hence are mathematically oriented. The former is a measure of the capacity (complexity, expressive power, richness, or flexibility) of a space of functions that can be learned by a classification algorithm. It was originally defined by Vladimir Vapnik and Alexey Chervonenkis in 1971. The latter is a framework for the mathematical analysis of learning algorithms. The goal is to check whether the probability for a selected hypothesis to be approximately correct is very high. The notion of PAC
learning was proposed by Leslie Valiant in 1984.

V-C dimension
Let H be the hypothesis space for some machine learning problem. The Vapnik-Chervonenkis dimension of H, also called the VC dimension of H, and denoted by V C(H), is a measure of the complexity (or, capacity, expressive power, richness, or flexibility) of the space H. To define the VC dimension we require the notion of the shattering of a set of instances.

Shattering of a set

Let D be a dataset containing N examples for a binary classification problem with class labels 0 and 1. Let H be a hypothesis space for the problem. Each hypothesis h in H partitions D into two disjoint subsets as follows:

$\{x \in D \mid h(x) = 0\}$ and $\{x \in D \mid h(x) = 1\}$.

Such a partition of S is called a "dichotomy" in D. It can be shown that there are 2^N possible dichotomies in D. To each dichotomy of D there is a unique assignment of the labels "1" and "0" to the elements of D. Conversely, if S is any subset of D then, S defines a unique hypothesis h as follows:

$$h(x) = \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{otherwise} \end{cases}$$

Thus to specify a hypothesis h, we need only specify the set {x ∈ D | h(x) = 1}. Figure 3.1 shows all possible dichotomies of D if D has three elements. In the figure, we have shown only one of the two sets in a dichotomy, namely the set {x ∈ D | h(x) = 1}.The circles and ellipses represent such sets.
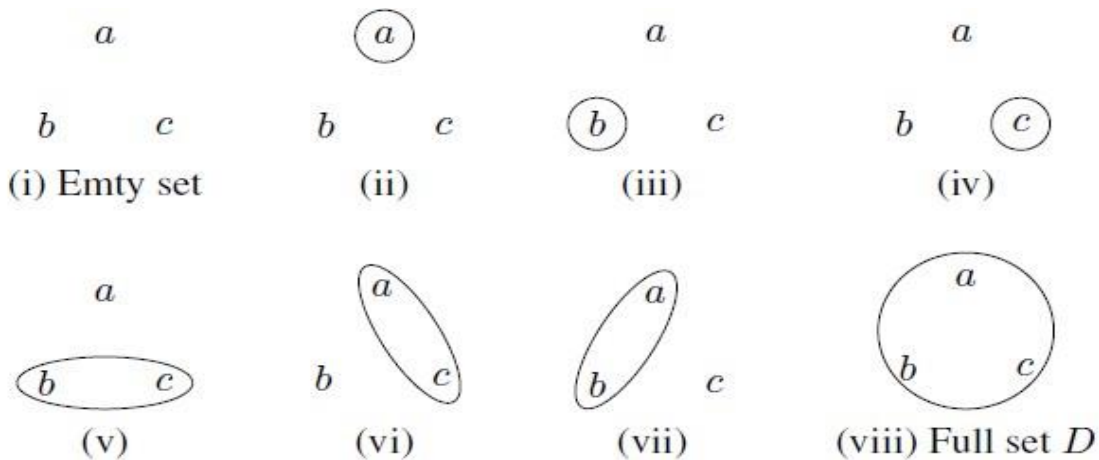

Figure 3.1: Different forms of the set $\{x \in S : h(x) = 1\}$ for $D = \{a, b, c\}$

Definition

A set of examples D is said to be shattered by a hypothesis space H if and only if for every dichotomy of D there exists some hypothesis in H consistent with the dichotomy of D.

The following example illustrates the concept of Vapnik-Chervonenkis dimension.

Example

In figure , we see that an axis-aligned rectangle can shatter four points in two dimensions. Then VC(H), when H is the hypothesis class of axis-aligned rectangles in two dimensions, is four. In calculating the VC dimension, it is enough that we find four points that can be shattered; it is not necessary that we be able to shatter any four points in two dimensions.
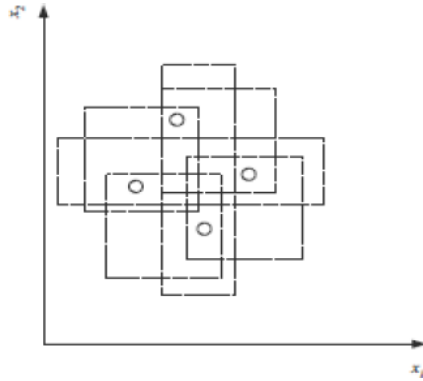
Fig: An axis-aligned rectangle can shattered four points. Only rectangle covering two points are shown.

VC dimension may seem pessimistic. It tells us that using a rectangle as our hypothesis class, we can learn only datasets containing four points and not more.

Unit II
Supervised and Unsupervised Learning

Topics: Decision Trees: ID3, Classification and Regression Trees, Regression: Linear Regression, Multiple Linear Regression, Logistic Regression, Neural Networks: Introduction, Perception, Multilayer Perception, Support Vector Machines: Linear and Non-Linear, Kernel Functions, K Nearest Neighbors. Introduction to clustering, K-means clustering, K-Mode Clustering.

Decision Tree

Introduction Decision Trees are a type of Supervised Machine Learning (that is you explain what the input is and what the corresponding output is in the training data) where the data is continuously split according to a certain parameter. The tree can be explained by two entities, namely **decision nodes and leaves**. The leaves are the decisions or the final outcomes. And the decision nodes are where the data is split.
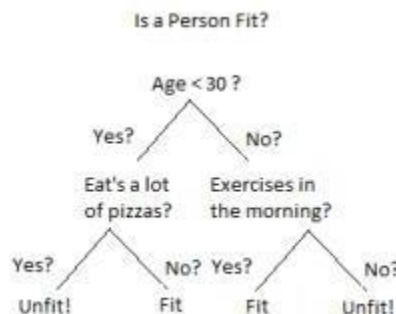
An example of a decision tree can be explained using above binary tree. Let's say you want to predict whether a person is fit given their information like age, eating habit, and physical activity, etc. The

decision nodes here are questions like 'What's the age?', 'Does he exercise?', and 'Does he eat a lot of pizzas'? And the leaves, which are outcomes like either 'fit', or 'unfit'. In this case this was a binary classification problem (a yes no type problem). There are two main types of Decision Trees:

1. **Classification trees** (Yes/No types)

What we have seen above is an example of classification tree, where the outcome was a variable like 'fit' or 'unfit'. Here the decision variable is **Categorical**.

2. **Regression trees** (Continuous data types)

Here the decision or the outcome variable is **Continuous**, e.g. a number like 123. **Working** Now that we know what a Decision Tree is, we'll see how it works internally. There are many algorithms out there which construct Decision Trees, but one of the best is called as **ID3 Algorithm**. ID3 Stands for **Iterative Dichotomiser 3**. Before discussing the ID3 algorithm, we'll go through few definitions. **Entropy** Entropy, also called as Shannon Entropy is denoted by H(S) for a finite set S, is the measure of the amount of uncertainty or randomness in data.

$$H(S) = \sum_{x \in X} p(x) \log_2 \frac{1}{p(x)}$$

Intuitively, it tells us about the predictability of a certain event. Example, consider a coin toss whose probability of heads is 0.5 and probability of tails is 0.5. Here the entropy is the highest possible, since there's no way of determining what the outcome might be. Alternatively, consider a coin which has heads on both the sides, the entropy of such an event can be predicted perfectly since we know beforehand that it'll always be heads. In other words, this event has **no randomness** hence it's entropy is zero. In particular, lower values imply less uncertainty while higher values imply high uncertainty. **Information Gain** Information gain is also called as Kullback-Leibler divergence denoted by IG(S,A) for a set S is the effective change in entropy after deciding on a particular attribute A. It measures the relative change in entropy with respect to the independent variables

$$IG(S,A) = H(S) - H(S,A)$$

Alternatively,

$$IG(S,A) = H(S) - \sum_{i=0}^{n} P(x) * H(x)$$

where IG(S, A) is the information gain by applying feature A. H(S) is the Entropy of the entire set, while the second term calculates the Entropy after applying the feature A, where P(x) is the probability of event x. Let's understand this with the help of an example Consider a piece of data collected over the course of 14 days where the features are Outlook, Temperature, Humidity, Wind and the outcome variable is whether Golf was played on the day. Now, our job is to build a predictive model which takes in above 4 parameters and predicts whether Golf will be played on the day. We'll build a decision tree to do that using **ID3 algorithm.**

Day	Outlook	Temperature	Humidity	Wind	Play Golf
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes

D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

ID3

ID3 Algorithm will perform following tasks recursively

1. Create root node for the tree
2. If all examples are positive, return leaf node „positive"
3. Else if all examples are negative, return leaf node „negative"
4. Calculate the entropy of current state H(S)
5. For each attribute, calculate the entropy with respect to the attribute „x" denoted by H(S, x)
6. Select the attribute which has maximum value of IG(S, x)
7. Remove the attribute that offers highest IG from the set of attributes
8. Repeat until we run out of all attributes, or the decision tree has all leaf nodes.

Now we"ll go ahead and grow the decision tree. The initial step is to calculate H(S), the Entropy of the current state. In the above example, we can see in total there are 5 No"s and 9 Yes"s.

Yes	No	Total
9	5	14

$$Entropy(S) = \sum_{x \in X} p(x) \log_2 \frac{1}{p(x)}$$

$$Entropy(S) = -\left(\frac{9}{14}\right) \log_2 \left(\frac{9}{14}\right) - \left(\frac{5}{14}\right) \log_2 \left(\frac{5}{14}\right)$$

$$= 0.940$$

where „x" are the possible values for an attribute. Here, attribute „Wind" takes two possible values in the sample data, hence x = {Weak, Strong} we"ll have to calculate:

1. $H(S_{weak})$
2. $H(S_{strong})$
3. $P(S_{weak})$
4. $P(S_{strong})$
5. $H(S) = 0.94$ which we had already calculated in the previous example

Amongst all the 14 examples we have **8 places where the wind is weak and 6 where the wind is Strong**.

Wind = Weak	Wind = Strong	Total
8	6	14

$$P(S_{weak}) = \frac{Number\ of\ Weak}{Total}$$

$$= \frac{8}{14}$$

$$P(S_{strong}) = \frac{Number\ of\ Strong}{Total}$$

$$= \frac{6}{14}$$

Now out of the 8 Weak examples, 6 of them were „Yes" for Play Golf and 2 of them were „No" for „Play Golf". So, we have,

$$Entropy(S_{weak}) = -\left(\frac{6}{8}\right)\log_2\left(\frac{6}{8}\right) - \left(\frac{2}{8}\right)\log_2\left(\frac{2}{8}\right)$$

$$= 0.811$$

Similarly, out of 6 Strong examples, we have **3 examples where the outcome was „Yes" for Play Golf and 3 where we had „No" for Play Golf**.

$$Entropy(S_{strong}) = -\left(\frac{3}{6}\right)\log_2\left(\frac{3}{6}\right) - \left(\frac{3}{6}\right)\log_2\left(\frac{3}{6}\right)$$

$$= 1.000$$

Remember, here half items belong to one class while other half belong to other. Hence we have perfect randomness. Now we have all the pieces required to calculate the Information Gain,

$$IG(S, Wind) = H(S) - \sum_{i=0}^{n} P(x) * H(x)$$

$$IG(S, Wind) = H(S) - P(S_{weak}) * H(S_{weak}) - P(S_{strong}) * H(S_{strong})$$

$$= 0.940 - \left(\frac{8}{14}\right)(0.811) - \left(\frac{6}{14}\right)(1.00)$$

$$= 0.048$$

Which tells us the Information Gain by considering „Wind" as the feature and give us information gain of **0.048**. Now we must similarly calculate the Information Gain for all the features.

$$IG(S, Outlook) = 0.246$$

$$IG(S, Temperature) = 0.029$$

$$IG(S, Humidity) = 0.151$$

$$IG(S, Wind) = 0.048\ \text{(Previous example)}$$

We can clearly see that IG(S, Outlook) has the highest information gain of 0.246, **hence we chose Outlook attribute as the root node**. At this point, the decision tree looks like.
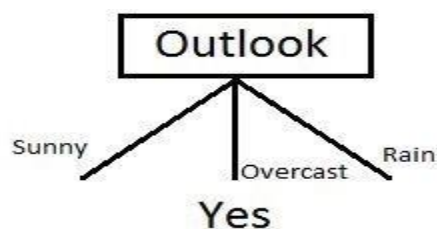
Here we observe that whenever the outlook is Overcast, Play Golf is always 'Yes', it's no coincidence by any chance, the simple tree resulted because of **the highest information gain is given by the attribute Outlook**. Now how do we proceed from this point? We can simply apply **recursion**, you might want to look at the algorithm steps described earlier. Now that we've used Outlook, we've got three of them remaining Humidity, Temperature, and Wind. And, we had three possible values of Outlook: Sunny, Overcast, Rain. Where the Overcast node already ended up having leaf node 'Yes', so we're left with two subtrees to compute: Sunny and Rain.

Next step would be computing $H(S_{sunny})$.

Table where the value of Outlook is Sunny looks like:

Temperature	Humidity	Wind	Play Golf
Hot	High	Weak	No
Hot	High	Strong	No
Mild	High	Weak	No
Cool	Normal	Weak	Yes
Mild	Normal	Strong	Yes

$$H(S_{sunny}) = \left(\frac{3}{5}\right)\log_2\left(\frac{3}{5}\right) - \left(\frac{2}{5}\right)\log_2\left(\frac{2}{5}\right) = 0.96$$

As we can see the **highest Information Gain is given by Humidity**. Proceeding in the same way with

$$S_{rain}$$

will give us Wind as the one with highest information gain. The final Decision Tree looks something like this. The final Decision Tree looks something like this.
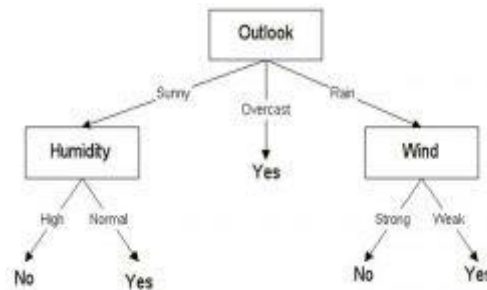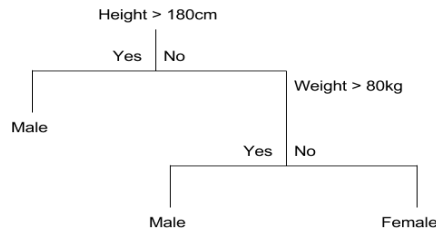

Classification and Regression Trees
      Classification Trees

      A classification tree is an algorithm where the target variable is fixed or categorical. The algorithm is then used to identify the "class" within which a target variable would most likely fall.

An example of a classification-type problem would be determining who will or will not subscribe to a digital platform; or who will or will not graduate from high school.

These are examples of simple binary classifications where the categorical dependent variable can assume only one of two, mutually exclusive values. In other cases, you might have to predict among a number of different variables. For instance, you may have to predict which type of smartphone a consumer may decide to purchase.

In such cases, there are multiple values for the categorical dependent variable. Here's what a classic classification tree looks like

Regression Trees

A regression tree refers to an algorithm where the target variable is and the algorithm is used to predict it's value. As an example of a regression type problem, you may want to predict the selling prices of a residential house, which is a continuous dependent variable.

This will depend on both continuous factors like square footage as well as categorical factors like the style of home, area in which the property is located and so on.
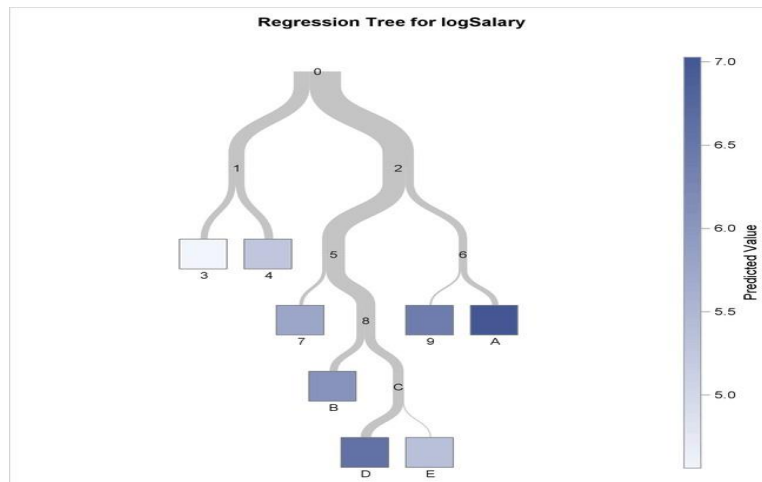
When to use Classification and Regression Trees

Classification trees are used when the dataset needs to be split into classes which belong to the response variable. In many cases, the classes Yes or No.

In other words, they are just two and mutually exclusive. In some cases, there may be more than two classes in which case a variant of the classification tree algorithm is used.

Regression trees, on the other hand, are used when the response variable is continuous. For instance, if the response variable is something like the price of a property or the temperature of the day, a regression tree is used.
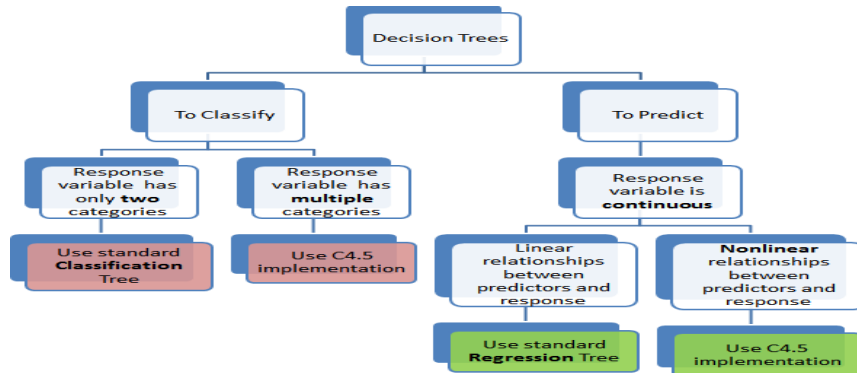
In other words, regression trees are used for prediction-type problems while classification trees are used for classification-type problems.

How Classification and Regression Trees Work

A classification tree splits the dataset based on the homogeneity of data. Say, for instance, there are two variables; income and age; which determine whether or not a consumer will buy a particular kind of phone.

If the training data shows that 95% of people who are older than 30 bought the phone, the data gets split there and age becomes a top node in the tree. This split makes the data "95% pure". Measures of impurity like entropy or Gini index are used to quantify the homogeneity of the data when it comes to classification trees.

In a regression tree, a regression model is fit to the target variable using each of the independent variables. After this, the data is split at several points for each independent variable.

At each such point, the error between the predicted values and actual values is squared to get "A Sum of Squared Errors" (SSE). The SSE is compared across the variables and the variable or point which has the lowest SSE is chosen as the split point. This process is continued recursively.

Decision Trees

- To Classify
  - Response variable has only **two** categories → Use standard **Classification** Tree
  - Response variable has **multiple** categories → Use C4.5 implementation
- To Predict
  - Response variable is **continuous**
    - Linear relationships between predictors and response → Use standard **Regression** Tree
    - **Nonlinear** relationships between predictors and response → Use C4.5 implementation

Advantages of Classification and Regression Trees

The purpose of the analysis conducted by any classification or regression tree is to create a set of if-else conditions that allow for the accurate prediction or classification of a case.

(i) The Results are Simplistic

The interpretation of results summarized in classification or regression trees is usually fairly simple. The simplicity of results helps in the following ways.

- It allows for the rapid classification of new observations. That's because it is much simpler to evaluate just one or two logical conditions than to compute scores using complex nonlinear equations for each group.
- It can often result in a simpler model which explains why the observations are either classified or predicted in a certain way. For instance, business problems are much easier to explain with if-then statements than with complex nonlinear equations.

(ii) Classification and Regression Trees are Nonparametric & Nonlinear

The results from classification and regression trees can be summarized in simplistic if-then conditions. This negates the need for the following implicit assumptions.

- The predictor variables and the dependent variable are linear.
- The predictor variables and the dependent variable follow some specific nonlinear link function.
- The predictor variables and the dependent variable are monotonic.

Since there is no need for such implicit assumptions, classification and regression tree methods are well suited to data mining. This is because there is very little knowledge or assumptions that can be made beforehand about how the different variables are related.

As a result, classification and regression trees can actually reveal relationships between these variables that would not have been possible using other techniques.

(iii) Classification and Regression Trees Implicitly Perform Feature Selection

Feature selection or variable screening is an important part of analytics. When we use decision trees, the top few nodes on which the tree is split are the most important variables within the set. As a result, feature selection gets performed automatically and we don't need to do it again.

Limitations of Classification and Regression Trees

Classification and regression tree tutorials, as well as classification and regression tree ppts, exist in abundance. This is a testament to the popularity of these decision trees and how frequently they are used. However, these decision trees are not without their disadvantages.

There are many classification and regression trees examples where the use of a decision tree has not led to the optimal result. Here are some of the limitations of classification and regression trees.

(i) Overfitting

Overfitting occurs when the tree takes into account a lot of noise that exists in the data and comes up with an inaccurate result.

(ii) High variance

In this case, a small variance in the data can lead to a very high variance in the prediction, thereby affecting the stability of the outcome.

(iii) Low bias

A decision tree that is very complex usually has a low bias. This makes it very difficult for the model to incorporate any new data.

What is a CART in Machine Learning?

A Classification and Regression Tree (CART) is a predictive algorithm used in machine learning. It explains how a target variable's values can be predicted based on other values.

It is a decision tree where each fork is a split in a predictor variable and each node at the end has a prediction for the target variable.

The CART algorithm is an important decision tree algorithm that lies at the foundation of machine learning. Moreover, it is also the basis for other powerful machine learning algorithms like bagged decision trees, random forest and boosted decision trees.

Summing up

The Classification and regression tree (CART) methodology is one of the oldest and most fundamental algorithms. It is used to predict outcomes based on certain predictor variables.

They are excellent for data mining tasks because they require very little data pre-processing. Decision tree models are easy to understand and implement which gives them a strong advantage when compared to other analytical models.

Regression

Regression Analysis in Machine learning

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:
- o Prediction of rain using temperature and other factors
- o Determining Market trends
- o Prediction of road accidents due to rash driving.

Terminologies Related to the Regression Analysis:
- o **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**.
- o **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**.
- o **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.
- o **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.
- o **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:

o Regression estimates the relationship between the target and the independent variable.
o It is used to find the trends in data.
o It helps to predict real/continuous values.
o 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:
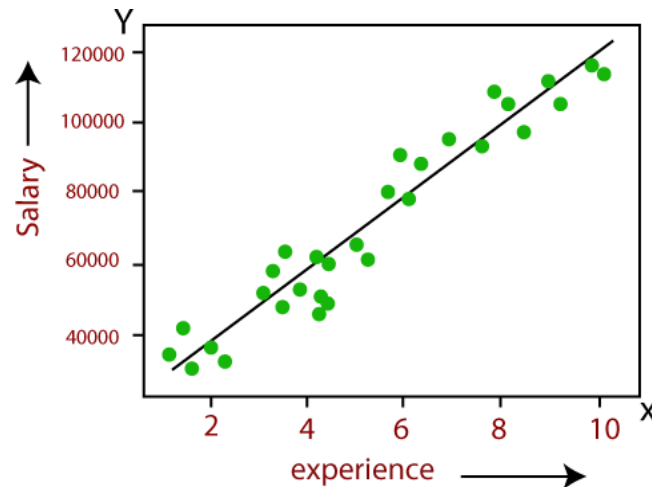
o **Linear Regression**
o **Logistic Regression**
o **Polynomial Regression**
o **Support Vector Regression**
o **Decision Tree Regression**
o **Random Forest Regression**
o **Ridge Regression**
o **Lasso Regression**
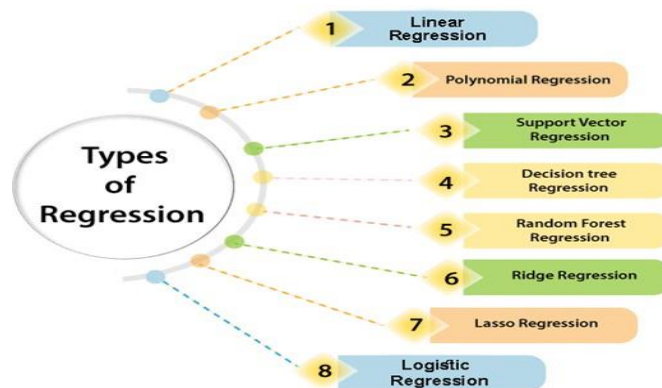
Linear Regression:
o Linear regression is a statistical regression method which is used for predictive analysis.
o It is one of the very simple and easy algorithms which works on regression and shows the relationship between the continuous variables.
o It is used for solving the regression problem in machine learning.
o Linear regression shows the linear relationship between the independent variable (X-axis) and the dependent variable (Y-axis), hence called linear regression.

- o 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**.
- o 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:

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:
- o Analyzing trends and sales estimates

- o Salary forecasting

- o Real estate prediction

- o Arriving at ETAs in traffic.

Logistic Regression:
- o 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.

- o Logistic regression algorithm works with the categorical variable such as 0 or 1, Yes or No, True or False, Spam or not spam, etc.

- o It is a predictive analysis algorithm which works on the concept of probability.

- o Logistic regression is a type of regression, but it is different from the linear regression algorithm in the term how they are used.

- o 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:

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

o f(x)= Output between the 0 and 1 value.

o x= input to the function

o e= base of natural logarithm.

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

$$sig(x)=\frac{1}{1+e^x}$$
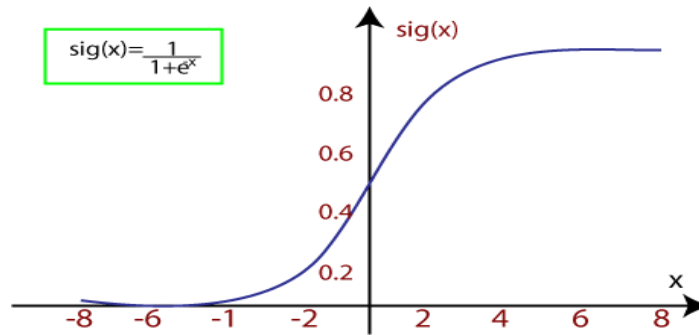
o It uses the concept of threshold levels, values above the threshold level are rounded up to 1, and values below the threshold level are rounded up to 0.

There are three types of logistic regression:
o **Binary(0/1, pass/fail)**

o **Multi(cats, dogs, lions)**

o **Ordinal(low, medium, high)**

Linear Regression in Machine Learning

Linear regression is one of the easiest and most popular Machine Learning algorithms. It is a statistical method that is used for predictive analysis. Linear regression makes predictions for continuous/real or numeric variables such as **sales, salary, age, product price,** etc.

Linear regression algorithm shows a linear relationship between a dependent (y) and one or more independent (y) variables, hence called as linear regression. Since linear regression shows the linear relationship, which means it finds how the value of the dependent variable is changing according to the value of the independent variable.

The linear regression model provides a sloped straight line representing the relationship between the variables. Consider the below image:
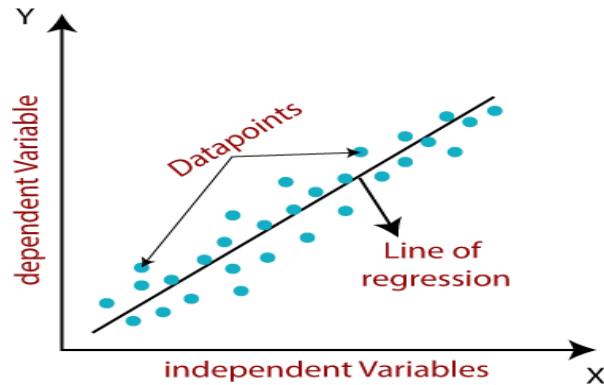
Mathematically, we can represent a linear regression as:

$$y = a_0 + a_1 x + \varepsilon$$

Here,
Y= Dependent Variable (Target Variable)
X= Independent Variable (predictor Variable)
a_0= intercept of the line (Gives an additional degree of freedom)
a_1 = Linear regression coefficient (scale factor to each input value).
ε = random error
The values for x and y variables are training datasets for Linear Regression model representation.

Types of Linear Regression

Linear regression can be further divided into two types of the algorithm:
- o **Simple Linear Regression:**
  If a single independent variable is used to predict the value of a numerical dependent variable, then such a Linear Regression algorithm is called Simple Linear Regression.

- o **Multiple Linear regression:**
  If more than one independent variable is used to predict the value of a numerical dependent variable, then such a Linear Regression algorithm is called Multiple Linear Regression.

Linear Regression Line:
A linear line showing the relationship between the dependent and independent variables is called a **regression line**.
A regression line can show two types of relationship:
- o **Positive Linear Relationship:**
  If the dependent variable increases on the Y-axis and independent variable increases on X-axis, then such a relationship is termed as a Positive linear relationship.
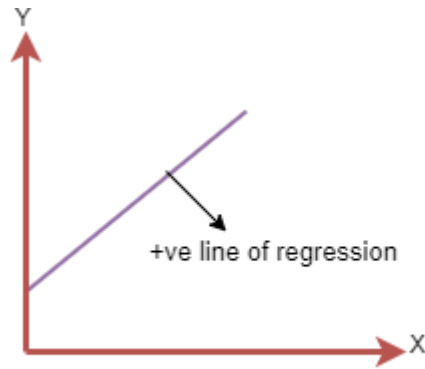
The line equation will be: $Y = a_0 + a_1X$

- o **Negative Linear Relationship:**
  If the dependent variable decreases on the Y-axis and independent variable increases on the X-axis, then such a relationship is called a negative linear relationship.
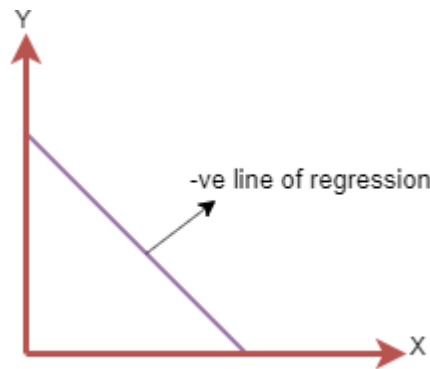
The line of equation will be: $Y = -a_0 + a_1X$

Finding the best fit line:

When working with linear regression, our main goal is to find the best fit line that means the error between predicted values and actual values should be minimized. The best fit line will have the least error.

The different values for weights or the coefficient of lines (a_0, a_1) gives a different line of regression, so we need to calculate the best values for a_0 and a_1 to find the best fit line, so to calculate this we use cost function.

Cost function-

- o The different values for weights or coefficient of lines (a_0, a_1) gives the different line of regression, and the cost function is used to estimate the values of the coefficient for the best fit line.

- o Cost function optimizes the regression coefficients or weights. It measures how a linear regression model is performing.

- o We can use the cost function to find the accuracy of the **mapping function**, which maps the input variable to the output variable. This mapping function is also known as **Hypothesis function**.

For Linear Regression, we use the **Mean Squared Error (MSE)** cost function, which is the average of squared error occurred between the predicted values and actual values. It can be written as:

For the above linear equation, MSE can be calculated as:

$$MSE = 1\frac{1}{N} \sum_{i=1}^{n}(y_i - (a_1 x_i + a_0))^2$$

Where,

N=Total number of observation

Yi = Actual value

$(a1x_i+a_0)$= Predicted value.

Residuals: The distance between the actual value and predicted values is called residual. If the observed points are far from the regression line, then the residual will be high, and so cost function will high. If the scatter points are close to the regression line, then the residual will be small and hence the cost function.

Gradient Descent:

o Gradient descent is used to minimize the MSE by calculating the gradient of the cost function.

o A regression model uses gradient descent to update the coefficients of the line by reducing the cost function.

o It is done by a random selection of values of coefficient and then iteratively update the values to reach the minimum cost function.

Model Performance:

The Goodness of fit determines how the line of regression fits the set of observations. The process of finding the best model out of various models is called **optimization**. It can be achieved by below method:

1. R-squared method:

o R-squared is a statistical method that determines the goodness of fit.

o It measures the strength of the relationship between the dependent and independent variables on a scale of 0-100%.

o The high value of R-square determines the less difference between the predicted values and actual values and hence represents a good model.

o It is also called a **coefficient of determination,** or **coefficient of multiple determination** for multiple regression.

o It can be calculated from the below formula:

$$R\text{-squared} = \frac{Explained\ variation}{Total\ Variation}$$

Assumptions of Linear Regression

Below are some important assumptions of Linear Regression. These are some formal checks while building a Linear Regression model, which ensures to get the best possible result from the given dataset.

o **Linear relationship between the features and target:**
Linear regression assumes the linear relationship between the dependent and independent variables.

o **Small or no multicollinearity between the features:**
Multicollinearity means high-correlation between the independent variables. Due to multicollinearity, it may difficult to find the true relationship between the predictors and target variables. Or we can say, it is

38

difficult to determine which predictor variable is affecting the target variable and which is not. So, the model assumes either little or no multicollinearity between the features or independent variables.

- o **Homoscedasticity Assumption:**
  Homoscedasticity is a situation when the error term is the same for all the values of independent variables. With homoscedasticity, there should be no clear pattern distribution of data in the scatter plot.

- o **Normal distribution of error terms:**
  Linear regression assumes that the error term should follow the normal distribution pattern. If error terms are not normally distributed, then confidence intervals will become either too wide or too narrow, which may cause difficulties in finding coefficients.
  It can be checked using the **q-q plot**. If the plot shows a straight line without any deviation, which means the error is normally distributed.

- o **No autocorrelations:**
  The linear regression model assumes no autocorrelation in error terms. If there will be any correlation in the error term, then it will drastically reduce the accuracy of the model. Autocorrelation usually occurs if there is a dependency between residual errors.

Simple Linear Regression in Machine Learning

Simple Linear Regression is a type of Regression algorithms that models the relationship between a dependent variable and a single independent variable. The relationship shown by a Simple Linear Regression model is linear or a sloped straight line, hence it is called Simple Linear Regression.

The key point in Simple Linear Regression is that the *dependent variable must be a continuous/real value*. However, the independent variable can be measured on continuous or categorical values.

Simple Linear regression algorithm has mainly two objectives:

- o **Model the relationship between the two variables.** Such as the relationship between Income and expenditure, experience and Salary, etc.

- o **Forecasting new observations.** Such as Weather forecasting according to temperature, Revenue of a company according to the investments in a year, etc.

Simple Linear Regression Model:

The Simple Linear Regression model can be represented using the below equation:

$y = a_0 + a_1 x + \varepsilon$

Where,
   a_0 = It is the intercept of the Regression line (can be obtained putting $x = 0$)
   a_1 = It is the slope of the regression line, which tells whether the line is increasing or decreasing.
   ε = The error term. (For a good model it will be negligible)

Multiple Linear Regressions

In the previous topic, we have learned about Simple Linear Regression, where a single Independent/Predictor(X) variable is used to model the response variable (Y). But there may be various cases in which the response variable is affected by more than one predictor variable; for such cases, the Multiple Linear Regression algorithm is used.

Moreover, Multiple Linear Regression is an extension of Simple Linear regression as it takes more than one predictor variable to predict the response variable.

We can define it as:
"*Multiple Linear Regression is one of the important regression algorithms which models the linear relationship between a single dependent continuous variable and more than one independent variable.*"

Example:
Prediction of CO_2 emission based on engine size and number of cylinders in a car.

Some key points about MLR:
- For MLR, the dependent or target variable(Y) must be the continuous/real, but the predictor or independent variable may be of continuous or categorical form.

- Each feature variable must model the linear relationship with the dependent variable.

- MLR tries to fit a regression line through a multidimensional space of data-points.

MLR equation:
In Multiple Linear Regression, the target variable(Y) is a linear combination of multiple predictor variables $x_1, x_2, x_3, ...,x_n$. Since it is an enhancement of Simple Linear Regression, so the same is applied for the multiple linear regression equation, the equation becomes:

$Y = b_0 + b_1 x_1 + b_2 x_2 + b_3 x_3 + ...$ bnxn ....(a)

Where,
Y= Output/Response variable
$b_0, b_1, b_2, b_3, b_n ...$ = Coefficients of the model.
$x_1, x_2, x_3, x_4,...$ = Various Independent/feature variable

Assumptions for Multiple Linear Regression:
- A linear relationship should exist between the Target and predictor variables.

- The regression residuals must be normally distributed.

- MLR assumes little or no multicollinearity (correlation between the independent variable) in data.

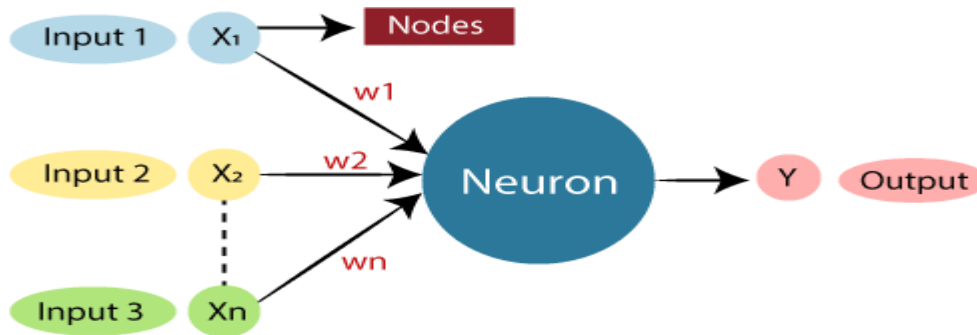**Neural Networks (ANN -** Artificial Neural Network**)**

Introduction
The term "Artificial Neural Network" is derived from Biological neural networks that develop the structure of a human brain. Similar to the human brain that has neurons interconnected to one another, artificial neural networks also have neurons that are interconnected to one another in various layers of the networks. These neurons are known as nodes.
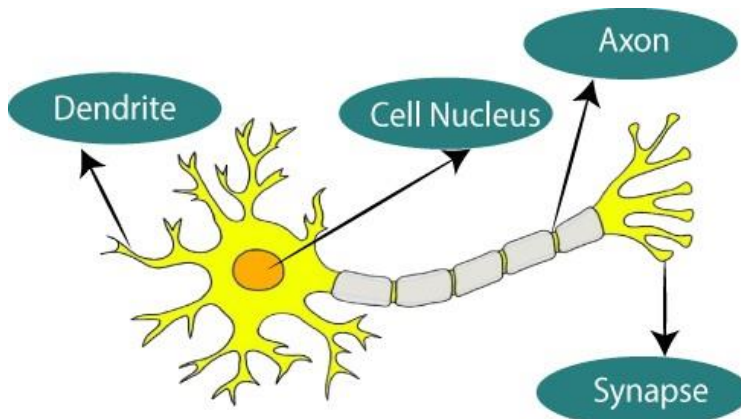
The given figure illustrates the typical diagram of Biological Neural Network.

The typical Artificial Neural Network looks something like the given figure.

Dendrites from Biological Neural Network represent inputs in Artificial Neural Networks, cell nucleus represents Nodes, synapse represents Weights, and Axon represents Output.

Relationship between Biological neural network and artificial neural network:

| Biological Neural Network	Artificial Neural Network
Dendrites	Inputs
Cell nucleus	Nodes
Synapse	Weights
Axon	Output

An **Artificial Neural Network** in the field of **Artificial intelligence** where it attempts to mimic the network of neurons makes up a human brain so that computers will have an option to understand things and make decisions in a human-like manner. The artificial neural network is designed by programming computers to behave simply like interconnected brain cells.

There are around 1000 billion neurons in the human brain. Each neuron has an association point somewhere in the range of 1,000 and 100,000. In the human brain, data is stored in such a manner as to be distributed, and we can

extract more than one piece of this data when necessary from our memory parallelly. We can say that the human brain is made up of incredibly amazing parallel processors.

We can understand the artificial neural network with an example, consider an example of a digital logic gate that takes an input and gives an output. "OR" gate, which takes two inputs. If one or both the inputs are "On," then we get "On" in output. If both the inputs are "Off," then we get "Off" in output. Here the output depends upon input. Our brain does not perform the same task. The outputs to inputs relationship keep changing because of the neurons in our brain, which are "learning."

The architecture of an artificial neural network:
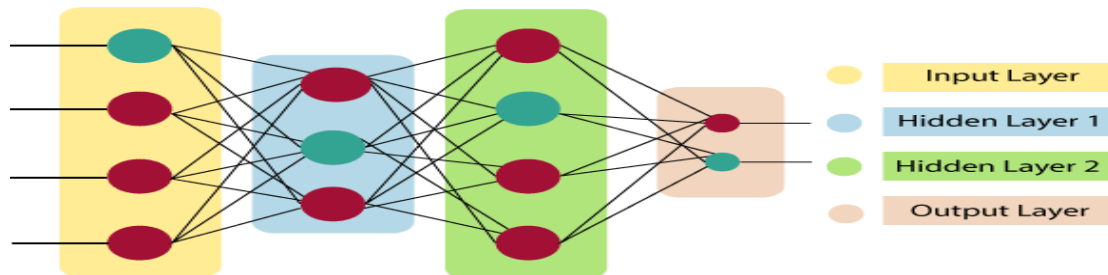
Input Layer:
As the name suggests, it accepts inputs in several different formats provided by the programmer.

Hidden Layer:
The hidden layer presents in-between input and output layers. It performs all the calculations to find hidden features and patterns.

Output Layer:
The input goes through a series of transformations using the hidden layer, which finally results in output that is conveyed using this layer.

The artificial neural network takes input and computes the weighted sum of the inputs and includes a bias. This computation is represented in the form of a transfer function.

$$\sum_{i=1}^{n} Wi * Xi + b$$

It determines weighted total is passed as an input to an activation function to produce the output. Activation functions choose whether a node should fire or not. Only those who are fired make it to the output layer. There are distinctive activation functions available that can be applied upon the sort of task we are performing.

Advantages of Artificial Neural Network (ANN)

Parallel processing capability:
Artificial neural networks have a numerical value that can perform more than one task simultaneously.

Storing data on the entire network:
Data that is used in traditional programming is stored on the whole network, not on a database. The disappearance of a couple of pieces of data in one place doesn't prevent the network from working.

Capability to work with incomplete knowledge:
After ANN training, the information may produce output even with inadequate data. The loss of

performance here relies upon the significance of missing data.

Having a memory distribution:
For ANN is to be able to adapt, it is important to determine the examples and to encourage the network according to the desired output by demonstrating these examples to the network. The succession of the network is directly proportional to the chosen instances, and if the event can't appear to the network in all its aspects, it can produce false output.

Having fault tolerance:
Extortion of one or more cells of ANN does not prohibit it from generating output, and this feature makes the network fault-tolerance.

Disadvantages of Artificial Neural Network:

Assurance of proper network structure:
There is no particular guideline for determining the structure of artificial neural networks. The appropriate network structure is accomplished through experience, trial, and error.

Unrecognized behavior of the network:
It is the most significant issue of ANN. When ANN produces a testing solution, it does not provide insight concerning why and how. It decreases trust in the network.

Hardware dependence:
Artificial neural networks need processors with parallel processing power, as per their structure. Therefore, the realization of the equipment is dependent.

Difficulty of showing the issue to the network:
ANNs can work with numerical data. Problems must be converted into numerical values before being introduced to ANN. The presentation mechanism to be resolved here will directly impact the performance of the network. It relies on the user's abilities.

The duration of the network is unknown:
The network is reduced to a specific value of the error, and this value does not give us optimum results.
"*Science artificial neural networks that have steeped into the world in the mid-20th century are exponentially developing. In the present time, we have investigated the pros of artificial neural networks and the issues encountered in the course of their utilization. It should not be overlooked that the cons of ANN networks, which are a flourishing science branch, are eliminated individually, and their pros are increasing day by day. It means that artificial neural networks will turn into an irreplaceable part of our lives progressively important.*"

How do artificial neural networks work?
Artificial Neural Network can be best represented as a weighted directed graph, where the artificial neurons form the nodes. The association between the neurons outputs and neuron inputs can be viewed as the directed edges with weights. The Artificial Neural Network receives the input signal from the external source in the form of a pattern and image in the form of a vector. These inputs are then mathematically assigned by the notations $x(n)$ for every n number of inputs.
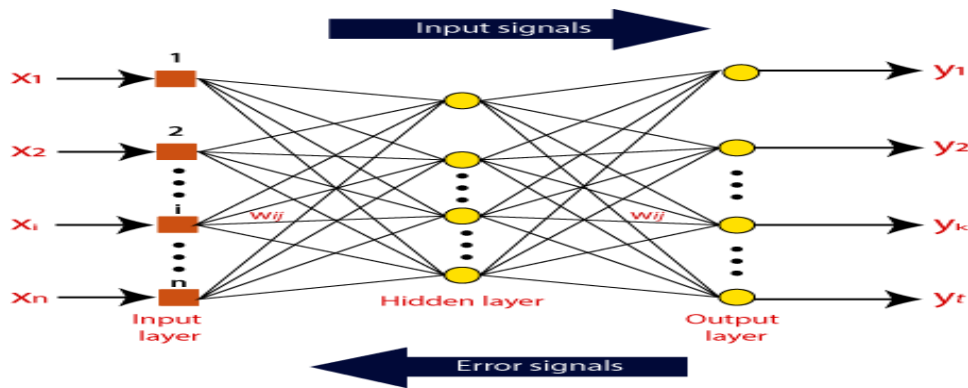
Afterward, each of the input is multiplied by its corresponding weights (these weights are the details utilized by the artificial neural networks to solve a specific problem). In general terms, these weights normally represent the strength of the interconnection between neurons inside the artificial neural network. All the weighted inputs are summarized inside the computing unit.

If the weighted sum is equal to zero, then bias is added to make the output non-zero or something else to scale up to the system's response. Bias has the same input, and weight equals to 1. Here the total of weighted inputs can be in the range of 0 to positive infinity. Here, to keep the response in the limits of the desired value, a certain maximum value is benchmarked, and the total of weighted inputs is passed through the activation function.

The activation function refers to the set of transfer functions used to achieve the desired output. There is a different kind of the activation function, but primarily either linear or non-linear sets of functions. Some of the commonly used sets of activation functions are the Binary, linear, and Tan hyperbolic sigmoidal activation functions. Let us take a look at each of them in details:

Binary:
In binary activation function, the output is either a one or a 0. Here, to accomplish this, there is a threshold value set up. If the net weighted input of neurons is more than 1, then the final output of the activation function is returned as one or else the output is returned as 0.

Sigmoidal Hyperbolic:
The Sigmoidal Hyperbola function is generally seen as an "**S**" shaped curve. Here the tan hyperbolic function is used to approximate output from the actual net input. The function is defined as:
$F(x) = (1/1 + \exp(-????x))$
Where ???? is considered the Steepness parameter.

Types of Artificial Neural Network:
There are various types of Artificial Neural Networks (ANN) depending upon the human brain neuron and network functions, an artificial neural network similarly performs tasks. The majority of the artificial neural networks will have some similarities with a more complex biological partner and are very effective at their expected tasks. For example, segmentation or classification.

Feedback ANN:
In this type of ANN, the output returns into the network to accomplish the best-evolved results internally. As per the **University of Massachusetts**, Lowell Centre for Atmospheric Research. The feedback networks feed information back into itself and are well suited to solve optimization issues. The Internal system error corrections utilize feedback ANNs.

Feed-Forward ANN:
A feed-forward network is a basic neural network comprising of an input layer, an output layer, and at least one layer of a neuron. Through assessment of its output by reviewing its input, the intensity of the network can be noticed based on group behavior of the associated neurons, and the output is decided. The primary advantage of this network is that it figures out how to evaluate and recognize input patterns.

44

Prerequisite
No specific expertise is needed as a prerequisite before starting this tutorial.

Audience
Our Artificial Neural Network Tutorial is developed for beginners as well as professionals, to help them understand the basic concept of ANNs.

PERCEPTRONS
One type of ANN system is based on a unit called a perceptron, illustrated in below Figure: A perceptron takes a vector of real-valued inputs, calculates a linear combination of these inputs, then outputs a 1 if the result is greater than some threshold and -1 otherwise. More precisely, given inputs x_l through x_n the output $o(x_l, \ldots, x_n)$ computed by the perceptron is

$$o(x_1, \ldots, x_n) = \begin{cases} 1 \text{ if } w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n > 0 \\ -1 \text{ otherwise} \end{cases}$$
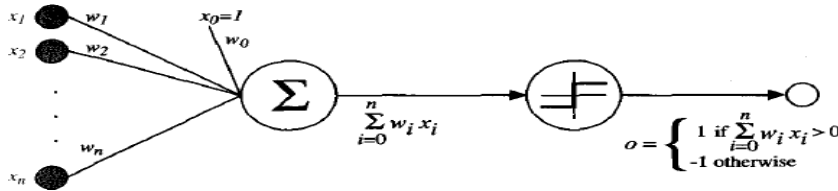
where each w_i is a real-valued constant, or weight, that determines the contribution of input x_i to the perceptron output. Notice the quantity $(-w_0)$ is a threshold that the weighted combination of inputs $w_1 x_1 + \ldots + w_n x_n$ must surpass in order for the perceptron to output a 1.

To simplify notation, we imagine an additional constant input $x_0 = 1$, allowing us to write the above inequality as $\sum_{i=0}^{n} w_i x_i > 0$, or in vector form as $\vec{w} \cdot \vec{x} > 0$. For brevity, we will sometimes write the perceptron function as

$$o(\vec{x}) = sgn(\vec{w} \cdot \vec{x})$$

where

$$sgn(y) = \begin{cases} 1 \text{ if } y > 0 \\ -1 \text{ otherwise} \end{cases}$$

Learning a perceptron involves choosing values for the weights $w_0, \ldots, w_n$ Therefore, the space H of candidate hypotheses considered in perceptron learning is the set of all possible real-valued weight vectors.

$$H = \{\vec{w} \mid \vec{w} \in \Re^{(n+1)}\}$$

Representational Power of Perceptrons:

We can view the perceptron as representing a hyperplane decision surface in the n--dimensional space of instances (i.e., points). The perceptron outputs a 1 for instances lying on one side of the hyperplane

45

and outputs a -1 for instances lying on the other side, as illustrated in Figure below The equation for this decision hyperplane is $\vec{w}.\vec{x} = 0$. Of course, some sets of positive and negative examples cannot be separated by any hyperplane. Those that can be separated are called linearly separable sets of examples.

(a)   (b)
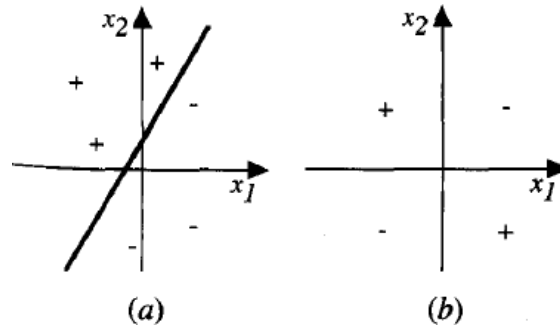
The decision surface represented by a two-input perceptron. *(a)* A set of training examples and the decision surface of a perceptron that classifies them correctly. *(b)* A set of training examples that is not linearly separable (i.e., that cannot be correctly classified by any straight line). x_l and x_2 are the Perceptron inputs. Positive examples are indicated by "+", negative by "-". The inputs are fed to multiple units, and the outputs of these units are then input to a second, final stage. One way is to represent the Boolean function in disjunctive normal form (i.e., as the disjunction (OR) of a set of conjunctions (ANDs) of the inputs and their negations). Note that the input to an AND perceptron can be negated simply by changing the sign of the corresponding input weight. Because networks of threshold units can represent a rich variety of functions and because single units alone cannot, we will generally be interested in learning multilayer networks of threshold units.

The Perceptron Training Rule

Although we are interested in learning networks of many interconnected units, let us begin by understanding how to learn the weights for a single perceptron. Here the precise learning problem is to determine a weight vector that causes the perceptron to produce the correct ± 1 output for each of the given training examples.

Several algorithms are known to solve this learning problem. Here we consider two: the perceptron rule and the delta rule. These two algorithms are guaranteed to converge to somewhat different acceptable hypotheses, under somewhat different conditions. They are important to ANNs because they provide the basis for learning networks of many units.

One way to learn an acceptable weight vector is to begin with random weights, then iteratively apply the perceptron to each training example, modifying the perceptron weights whenever it misclassifies an example. This process is repeated, iterating through the training examples as many times as needed until
the perceptron classifies all training examples correctly. Weights are modified at each step according to the perceptron training rule, which revises the weight w_i associated with input x_i according to the rule

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = \eta(t - o)x_i$$

Here t is the target output for the current training example, o is the output generated by the

perceptron, and η is a positive constant called the **learning rate.** The role of the learning rate is to moderate the degree to which weights are changed at each step. It is usually set to some small value (e.g., 0.1) and is sometimes made to decay as the number of weight-tuning iterations increases.

Why should this update rule converge toward successful weight values? To get an intuitive feel, consider some specific cases. Suppose the training example is correctly classified already by the perceptron. In this case, *(t - o)* is zero, making Δw_i zero, so that no weights are updated. Suppose the perceptron outputs a -1, when the target output is +1. To make the perceptron output a+1 instead of -1 in this case, the weights must be altered to increase the value of $\vec{w}.\vec{x}$ For example, if *x_i>0,* then increasing *w_i* will bring the perceptron closer to correctly classifying this example. Notice the training rule will increase *w,* in this case, because *(t - o),* η *,* and *x_i* are all positive. For example, if *x_i = .8,* η = *0.1, t = 1,* and *o = - 1,* then the weight update will be Δw_i = η *(t - o)x_i = O.1(1 - (-1))0.8 = 0.16.* On the other hand, if *t = -1* and *o* = 1, then weights associated with positive *x_i* will be decreased rather than increased.

In fact, the above learning procedure can be proven to converge within a finite number of applications of the perceptron training rule to a weight vector that correctly classifies all training examples, *provided the training examples are linearly separable* and provided a sufficiently small η is used*.* If the data are not linearly separable, convergence is not assured.

Gradient Descent and the Delta Rule

Although the perceptron rule finds a successful weight vector when the training examples are linearly separable, it can fail to converge if the examples are not linearly separable. **A** second training rule, called the ***delta rule,*** is designed to overcome this difficulty. If the training examples are not linearly separable, the delta rule converges toward a best-fit approximation to the target concept. The key idea behind the delta rule is to use ***gradient descent*** to search the hypothesis space of possible weight vectors to find the weights that best fit the training examples. This rule is important because gradient descent provides the basis for the BACKPROPAGATION algorithm, which can learn networks with many interconnected units. It is also important because gradient descent can serve as the basis for learning algorithms that must search through hypothesis spaces containing many different types of continuously parameterized hypotheses.

The delta training rule is best understood by considering the task of training an ***unthresholded*** perceptron; that is, a ***linear unit*** for which the output *o* is given by

$$o(\vec{x}) = \vec{w} \cdot \vec{x}$$

Thus, a linear unit corresponds to the first stage of a perceptron, without the threshold.

In order to derive a weight learning rule for linear units, let us begin by specifying a measure for the ***training error*** of a hypothesis (weight vector), relative to the training examples. Although there are many ways to define this error, one common measure that will turn out to be especially convenient is

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D}(t_d - o_d)^2$$

where D is the set of training examples, *td* is the target output for training example *d,* and *o_d* is the output of the linear unit for training example d. By this definition, $E(\vec{w})$ is simply half the squared

difference between the target output t_d and the hear unit output o_d, summed over all training examples. Here we characterize E as a function of $(\vec{w})$, because the linear unit output o depends on this weight vector. Of course E also depends on the particular **set** of training examples, but we assume these are fixed during training, so we do not bother to write E as an explicit function of these. In particular, there we show that under certain conditions the hypothesis that minimizes E is also the most probable hypothesis in H given the training data.

Multi-layer Perceptron

      Multi-layer Perceptron (MLP) is a supervised learning algorithm that learns a function $f(\cdot):R^m \rightarrow R^o$ by training on a dataset, where m is the number of dimensions for input and o is the number of dimensions for output. Given a set of features $X=x^1,x^2,...,x^m$ and a target y, it can learn a non-linear function approximator for either classification or regression. It is different from logistic regression, in that between the input and the output layer, there can be one or more non-linear layers, called hidden layers. Figure shows a one hidden layer MLP with scalar output.
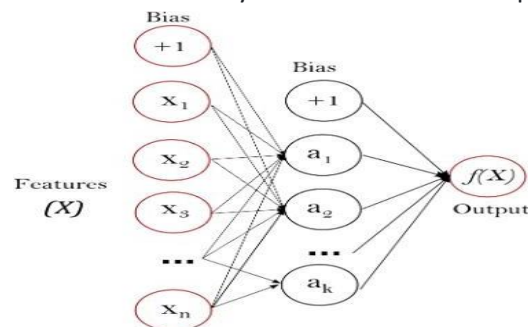
The leftmost layer, known as the input layer, consists of a set of neurons $\{x_i | x_1,x_2,...,x_m\}$ representing the input features. Each neuron in the hidden layer transforms the values from the previous layer with a weighted linear summation $w_1x_1+w_2x_2+...+w_mx_m$, followed by a non-linear activation function $g(\cdot):R \rightarrow R$ - like the hyperbolic tan function. The output layer receives the values from the last hidden layer and transforms them into output values.

The module contains the public attributes coefs_ and intercepts_. coefs_ is a list of weight matrices, where weight matrix at index i represents the weights between layer i and layer i+1. intercepts_ is a list of bias vectors, where the vector at index i represents the bias values added to layer i+1.

The advantages of Multi-layer Perceptron are:

- Capability to learn non-linear models.
- Capability to learn models in real-time (on-line learning) using partial_fit.

The disadvantages of Multi-layer Perceptron (MLP) include:

- MLP with hidden layers have a non-convex loss function where there exists more than one local minimum. Therefore different random weight initializations can lead to different validation accuracy.
- MLP requires tuning a number of hyperparameters such as the number of hidden neurons, layers, and iterations.
- MLP is sensitive to feature scaling.

Support Vector Machines

Support Vector Machine or SVM is one of the most popular Supervised Learning algorithms, which is used for Classification as well as Regression problems. However, primarily, it is used for Classification problems in Machine Learning. The goal of the SVM algorithm is to create the best line or decision boundary that can segregate n-dimensional space into classes so that we can easily put the new data point in the correct category in the future. This best decision boundary is called a hyperplane. SVM chooses the extreme points/vectors that help in creating the hyperplane. These extreme cases are called as support vectors, and hence algorithm is termed as Support Vector Machine. Consider the below diagram in which there are two different categories that are classified using a decision boundary or hyperplane:
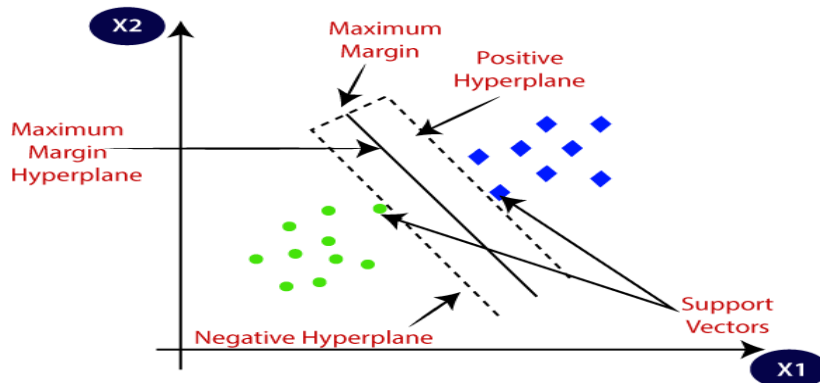
Example: SVM can be understood with the example that we have used in the KNN classifier. Suppose we see a strange cat that also has some features of dogs, so if we want a model that can accurately identify whether it is a cat or dog, so such a model can be created by using the SVM algorithm. We will first train our model with lots of images of cats and dogs so that it can learn about different features of cats and dogs, and then we test it with this strange creature. So as support vector creates a decision boundary between these two data (cat and dog) and choose extreme cases (support vectors), it will see the extreme case of cat and dog. On the basis of the support vectors, it will classify it as a cat. Consider the below diagram:
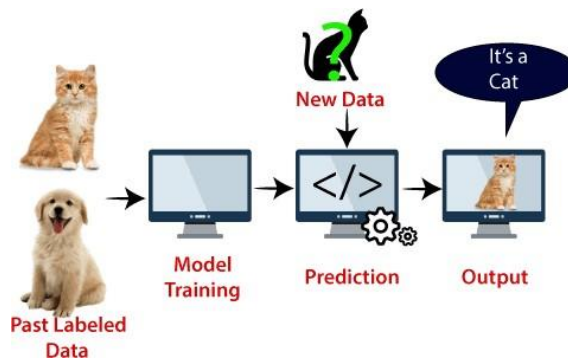
SVM algorithm can be used for **Face detection, image classification, text categorization,** etc.
Types of SVM
SVM can be of two types:

- o **Linear SVM:** Linear SVM is used for linearly separable data, which means if a dataset can be classified into two classes by using a single straight line, then such data is termed as linearly separable data, and classifier is used called as Linear SVM classifier.

- o **Non-linear SVM:** Non-Linear SVM is used for non-linearly separated data, which means if a dataset cannot be classified by using a straight line, then such data is termed as non-linear data and classifier used is called as Non-linear SVM classifier.

*Hyperplane and Support Vectors in the SVM algorithm***:**
    Hyperplane: There can be multiple lines/decision boundaries to segregate the classes in n-dimensional space, but we need to find out the best decision boundary that helps to classify the data points. This best boundary is known as the hyperplane of SVM.
The dimensions of the hyperplane depend on the features present in the dataset, which means if there are 2 features (as shown in image), then hyperplane will be a straight line. And if there are 3 features, then hyperplane will be a 2-dimension plane.
We always create a hyperplane that has a maximum margin, which means the maximum distance between the data points.
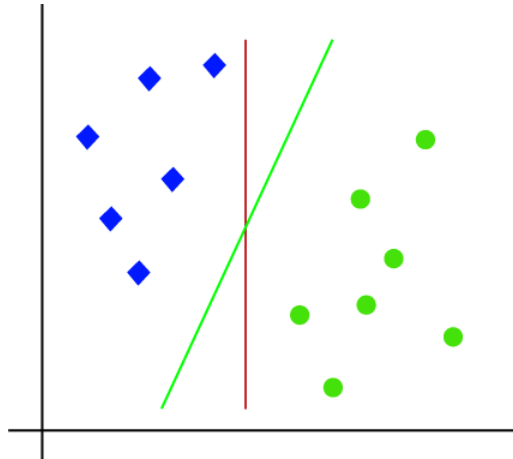
Support Vectors:
The data points or vectors that are the closest to the hyperplane and which affect the position of the hyperplane are termed as Support Vector. Since these vectors support the hyperplane, hence called a Support vector. How does SVM works?

Linear SVM:
The working of the SVM algorithm can be understood by using an example. Suppose we have a dataset that has two tags (green and blue), and the dataset has two features $x1$ and $x2$. We want a classifier that can classify the pair($x1$, $x2$) of coordinates in either green or blue. Consider the below image:
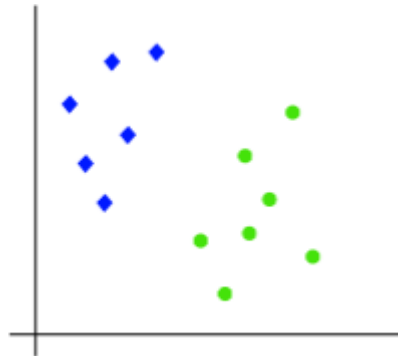

So as it is 2-d space so by just using a straight line, we can easily separate these two classes. But there can be multiple lines that can separate these classes. Consider the below image:

Hence, the SVM algorithm helps to find the best line or decision boundary; this best boundary or region is called as a **hyperplane**. SVM algorithm finds the closest point of the lines from both the classes. These points are called support vectors. The distance between the vectors and the hyperplane is called as **margin**. And the goal of SVM is to maximize this margin. The **hyperplane** with maximum margin is called the **optimal hyperplane**.

Non-Linear SVM:

If data is linearly arranged, then we can separate it by using a straight line, but for non-linear data, we cannot draw a single straight line. Consider the below image:
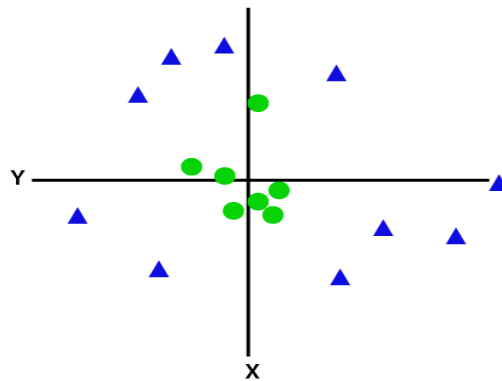
So to separate these data points, we need to add one more dimension. For linear data, we have used two dimensions x and y, so for non-linear data, we will add a third dimension z. It can be calculated as:

$z = x^2 + y^2$

By adding the third dimension, the sample space will become as below image:
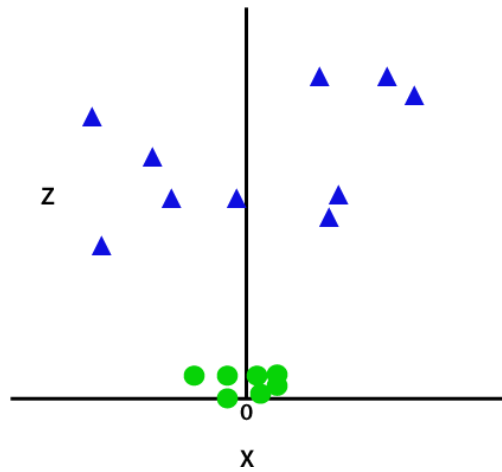
So now, SVM will divide the datasets into classes in the following way. Consider the below image:
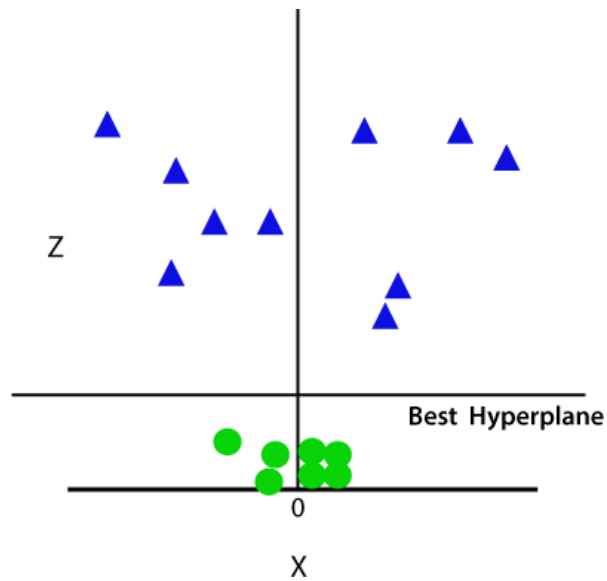
Since we are in 3-d Space, hence it is looking like a plane parallel to the x-axis. If we convert it in 2d space with z=1, then it will become as:
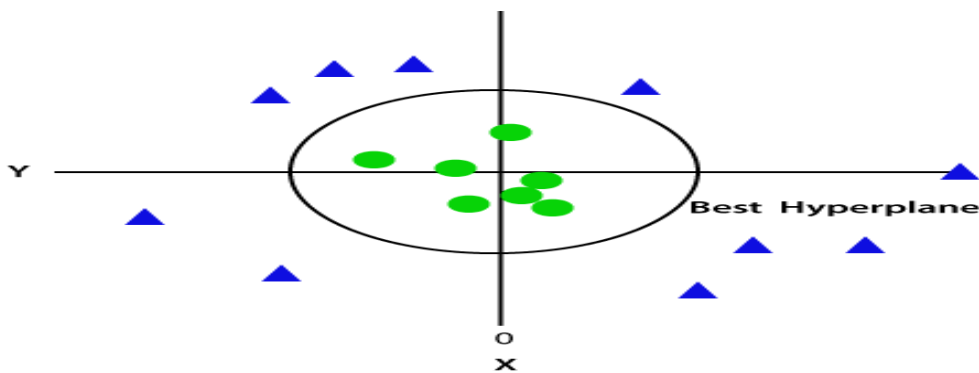
Hence we get a circumference of radius 1 in case of non-linear data.

SVM Kernels

In practice, SVM algorithm is implemented with kernel that transforms an input data space into

52

the required form. SVM uses a technique called the kernel trick in which kernel takes a low dimensional input space and transforms it into a higher dimensional space. In simple words, kernel converts non-separable problems into separable problems by adding more dimensions to it. It makes SVM more powerful, flexible and accurate. The following are some of the types of kernels used by SVM.

Linear Kernel

It can be used as a dot product between any two observations. The formula of linear kernel is as below

$$K(x,x_i)=sum(x*x_i)$$

From the above formula, we can see that the product between two vectors say x & xi is the sum of the multiplication of each pair of input values.

Unsupervised Machine Learning:
    Introduction to clustering

As the name suggests, unsupervised learning is a machine learning technique in which models are not supervised using training dataset. Instead, models itself find the hidden patterns and insights from the given data. It can be compared to learning which takes place in the human brain while learning new things. It can be defined as:

"*Unsupervised learning is a type of machine learning in which models are trained using unlabeled dataset and are allowed to act on that data without any supervision.*"

Unsupervised learning cannot be directly applied to a regression or classification problem because unlike supervised learning, we have the input data but no corresponding output data. The goal of unsupervised learning is to **find the underlying structure of dataset, group that data according to similarities, and represent that dataset in a compressed format**

Example: Suppose the unsupervised learning algorithm is given an input dataset containing images of different types of cats and dogs. The algorithm is never trained upon the given dataset, which means it does not have any idea about the features of the dataset. The task of the unsupervised learning algorithm is to identify the image features on their own. Unsupervised learning algorithm will perform this task by clustering the image dataset into the groups according to similarities between images.

Why use Unsupervised Learning?

Below are some main reasons which describe the importance of Unsupervised Learning:

- o Unsupervised learning is helpful for finding useful insights from the data.

- o Unsupervised learning is much similar as a human learns to think by their own experiences, which makes it closer to the real AI.

- o Unsupervised learning works on unlabeled and uncategorized data which make unsupervised learning more important.

- o In real-world, we do not always have input data with the corresponding output so to solve such cases, we need unsupervised learning.

Working of Unsupervised Learning

Working of unsupervised learning can be understood by the below diagram:
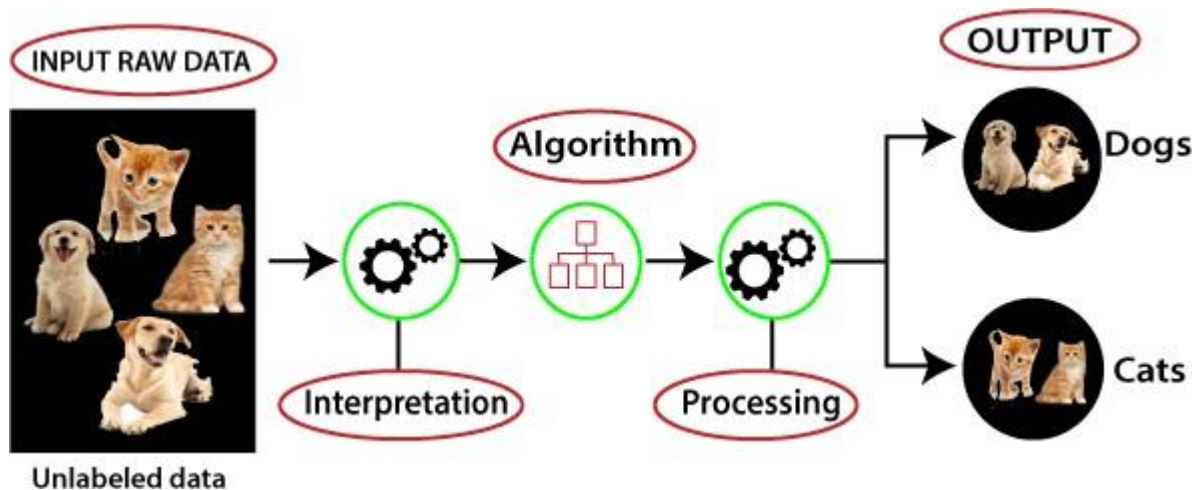
Here, we have taken an unlabeled input data, which means it is not categorized and corresponding outputs are also not given. Now, this unlabeled input data is fed to the machine learning model in order to train it. Firstly, it will interpret the raw data to find the hidden patterns from the data and then will apply suitable algorithms such as k-means clustering, Decision tree, etc.

Once it applies the suitable algorithm, the algorithm divides the data objects into groups according to the similarities and difference between the objects.

Types of Unsupervised Learning Algorithm:

The unsupervised learning algorithm can be further categorized into two types of problems:
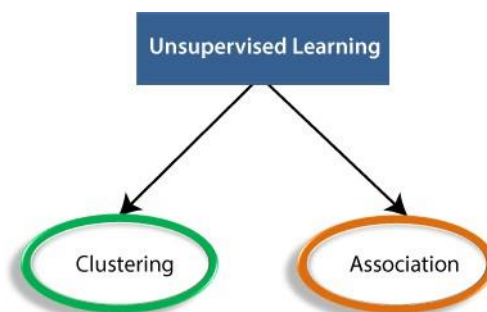
- o **Clustering**: Clustering is a method of grouping the objects into clusters such that objects with most similarities remains into a group and has less or no similarities with the objects of another group. Cluster analysis finds the commonalities between the data objects and categorizes them as per the presence and absence of those commonalities.

- o **Association**: An association rule is an unsupervised learning method which is used for finding the relationships between variables in the large database. It determines the set of items that occurs together in the dataset. Association rule makes marketing strategy more effective. Such as people who buy X item (suppose a bread) are also tend to purchase Y (Butter/Jam) item. A typical example of Association rule is Market Basket Analysis.

Unsupervised Learning algorithms:

Below is the list of some popular unsupervised learning algorithms:
- o K-means clustering

- o KNN (k-nearest neighbors)

- o Hierarchal clustering

- o Anomaly detection

- o Neural Networks

- o Principle Component Analysis

- o Independent Component Analysis

- o Apriori algorithm

- o Singular value decomposition

Advantages of Unsupervised Learning
- o Unsupervised learning is used for more complex tasks as compared to supervised learning because, in unsupervised learning, we don't have labeled input data.

- o Unsupervised learning is preferable as it is easy to get unlabeled data in comparison to labeled data.

Disadvantages of Unsupervised Learning
- o Unsupervised learning is intrinsically more difficult than supervised learning as it does not have corresponding output.

- o The result of the unsupervised learning algorithm might be less accurate as input data is not labeled, and algorithms do not know the exact output in advance.

Supervised Learning	Unsupervised Learning
Supervised learning algorithms are trained using labeled data.	Unsupervised learning algorithms are trained using unlabeled data.
Supervised learning model takes direct feedback to check if it is predicting correct output or not.	Unsupervised learning model does not take any feedback.
Supervised learning model predicts the output.	Unsupervised learning model finds the hidden patterns in data.
In supervised learning, input data is provided to the model along with the output.	In unsupervised learning, only input data is provided to the model.
The goal of supervised learning is to train the model so that it can predict the output when it is given new data.	The goal of unsupervised learning is to find the hidden patterns and useful insights from the unknown dataset.
Supervised learning needs supervision to train the model.	Unsupervised learning does not need any supervision to train the model.
Supervised learning can be categorized in **Classification** and **Regression** problems.	Unsupervised Learning can be classified in **Clustering** and **Associations** problems.
Supervised learning can be used for those cases where we know the input as well as corresponding outputs.	Unsupervised learning can be used for those cases where we have only input data and no corresponding output data.
Supervised learning model produces an accurate result.	Unsupervised learning model may give less accurate result as compared to supervised learning.
Supervised learning is not close to true Artificial intelligence as in this, we first train the model for each data, and then only it can predict the correct output.	Unsupervised learning is more close to the true Artificial Intelligence as it learns similarly as a child learns daily routine things by his experiences.
It includes various algorithms such as Linear Regression, Logistic Regression, Support Vector Machine, Multi-class Classification, Decision tree, Bayesian Logic, etc.	It includes various algorithms such as Clustering, KNN, and Apriori algorithm.

K-Mean Clustering

k-means clustering algorithm
One of the most used clustering algorithm is *k-means*. It allows to group the data according to the existing similarities among them in *k* clusters, given as input to the algorithm. I'll start with a simple example.
Let's imagine we have 5 objects (say 5 people) and for each of them we know two features (height and weight). We want to group them into *k=2* clusters.

Our dataset will look like this:

	Height (H)	Weight (W)
Person 1	167	55
Person 2	120	32
Person 3	113	33
Person 4	175	76
Person 5	108	25

First of all, we have to initialize the value of the centroids for our clusters. For instance, let's choose Person 2 and Person 3 as the two centroids $c1$ and $c2$, so that $c1=(120,32)$ and $c2=(113,33)$.

Now we compute the euclidian distance between each of the two centroids and each point in the data. If you did all the calculations, you should have come up with the following numbers:

	Distance of object from $c1$	Distance of object from $c2$
Person 1	52.3	58.3
Person 2	0	7.1
Person 3	7.1	0
Person 4	70.4	75.4
Person 5	13.9	9.4

At this point, we will assign each object to the cluster it is closer to (that is taking the minimum between the two computed distances for each object).

We can then arrange the points as follows:

Person 1 → cluster 1
Person 2 → cluster 1
Person 3 → cluster 2
Person 4 → cluster 1
Person 5→ cluster 2

Let's iterate, which means to redefine the centroids by calculating the mean of the members of each of the two clusters.

So $c'1$ = ((167+120+175)/3, (55+32+76)/3) = (154, 54.3) and $c'2$ = ((113+108)/2, (33+25)/2) = (110.5, 29)

Then, we calculate the distances again and re-assign the points to the new centroids.

We repeat this process until the centroids don't move anymore (or the difference between them is under a certain small threshold).

In our case, the result we get is given in the figure below. You can see the two different clusters labelled with two different colours and the position of the centroids, given by the crosses.
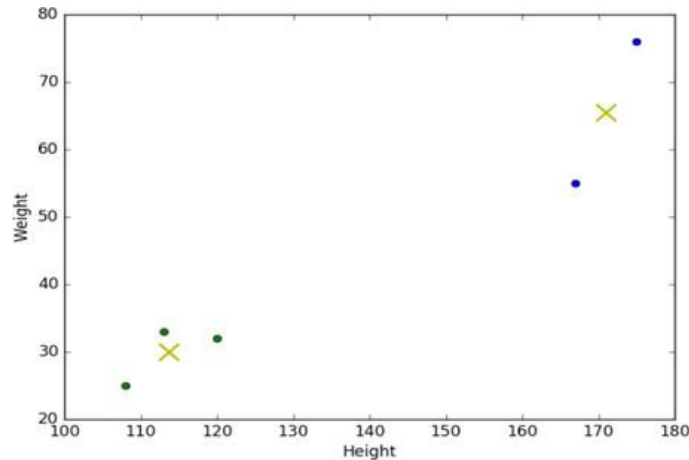
How to apply k-means?
As you probably already know, I'm using Python libraries to analyze my data. The *k-means* algorithm is implemented in the *scikit-learn* package. To use it, you will just need the following line in your script:

What if our data is… non-numerical?

At this point, you will maybe have noticed something. The basic concept of *k-means* stands on mathematical calculations (means, euclidian distances). But what if our data is non-numerical or, in other words, *categorical*? Imagine, for instance, to have the ID code and date of birth of the five people of the previous example, instead of their heights and weights.
We could think of transforming our categorical values in numerical values and eventually apply *k-means*. But beware: *k-means* uses numerical distances, so it could consider close two really distant objects that merely have been assigned two close numbers.

k-modes is an extension of *k-means*. Instead of distances it uses *dissimilarities* (that is, quantification of the total mismatches between two objects: the smaller this number, the more similar the two objects). And instead of means, it uses *modes.* A mode is a vector of elements that minimizes the dissimilarities between the vector itself and each object of the data. We will have as many modes as the number of clusters we required, since they act as centroids.

<div align="center">

Unit III
Ensemble and Probabilistic Learning

</div>

Ensemble Learning: Model Combination Schemes, Voting, Error-Correcting Output Codes, Bagging: Random Forest Trees, Boosting: Adaboost, Stacking.
Probabilistic Learning: Gaussian mixture models - The Expectation-Maximization (EM) Algorithm, Information Criteria, Nearest neighbour methods - Nearest Neighbour Smoothing, Efficient Distance Computations: the KD-Tree, Distance Measures.

3. Introduction:

Ensemble Learning
      Ensemble learning usually produces more accurate solutions than a single model would.

<div align="center">59</div>

Ensemble Learning is a technique that create multiple models and then combine them them to produce improved results. Ensemble learning usually produces more accurate solutions than a single model would.

- Ensemble learning methods is applied to regression as well as classification.
  - o Ensemble learning for regression creates multiple repressors i.e. multiple regression models such as linear, polynomial, etc.
  - o Ensemble learning for classification creates multiple classifiers i.e. multiple classification models such as logistic, decision tress, KNN, SVM, etc.
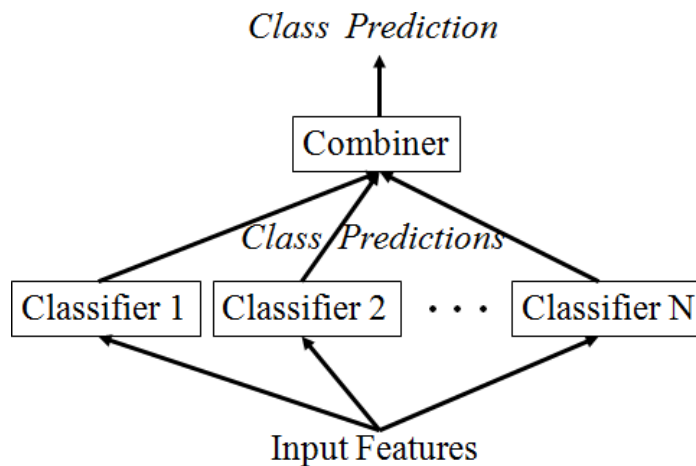
Figure 1: Ensemble learning view

Which components to combine?
- different learning algorithms
- same learning algorithm trained in different ways
- same learning algorithm trained the same way

There are two steps in ensemble learning:

Multiples machine learning models were generated using same or different machine learning algorithm. These are called "base models". The prediction perform on the basis of base models.

Techniques/Methods in ensemble learning

Voting, Error-Correcting Output Codes, Bagging: Random Forest Trees, Boosting: Adaboost, Stacking.

Model Combination Schemes - Combining Multiple Learners

We discussed many different learning algorithms in the previous chapters. Though these are generally successful, no one single algorithm is always the most accurate. Now, we are going to discuss models composed of multiple learners that complement each other so that by combining them, we attain higher accuracy.

There are also different ways the multiple base-learners are combined to generate the final output:

Figure2: General Idea - Combining Multiple Learners
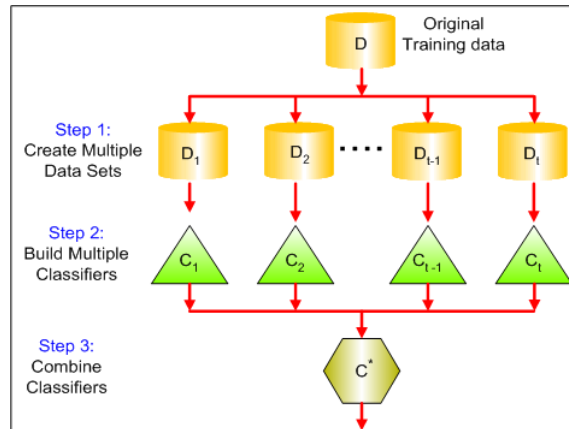
Multiexpert combination

Multiexpert combination methods have base-learners that work in parallel. These methods can in turn be divided into two:

- In the global approach, also called learner fusion, given an input, all base-learners generate an output and all these outputs are used.
  Examples are voting and stacking.
- In the local approach, or learner selection, for example, in mixture of experts, there is a gating model, which looks at the input and chooses one (or very few) of the learners as responsible for generating the output.

Multistage combination

Multistage combination methods use a *serial* approach where the next base-learner is trained with or tested on only the instances where the previous base-learners are not accurate enough. The idea is that the base-learners (or the different representations they use) are sorted in increasing complexity so that a complex base-learner is not used (or its complex representation is not extracted) unless the preceding simpler base-learners are not confident.
An example is *cascading*.

Let us say that we have *L* base-learners. We denote by *dj(x)* the prediction of base-learner M_j given the arbitrary dimensional input *x*. In the case of multiple representations, each M_j uses a different input representation x_j . The final prediction is calculated from the predictions of
the base-learners:

$$y = f (d_1, d_2, \ldots , d_L \,|\, \Phi)$$

where $f(\cdot)$ is the combining function with Φ denoting its parameters.
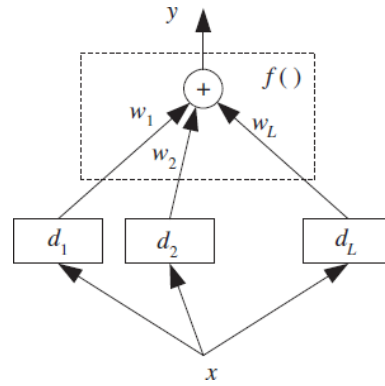
Figure 1: Base-learners are *dj* and their outputs are combined using *f (·)*. This is for a single output; in the case of classification, each base-learner has *K* outputs that are separately used to calculate *yi*, and then we choose the maximum. Note that here all learners observe the same input; it may be the case that different learners observe different representations of the same input object or event.

When there are *K* outputs, for each learner there are $d_{ji}(x)$, *i = 1, . . . , K,*
j = 1, . . . , L, and, combining them, we also generate *K* values, y_i, *i = 1, . . . , K* and then for example in classification, we choose the class with
the maximum *yi* value:

$$\text{Choose } C_i \text{ if } y_i = \max_{k-1}^{K} y_k$$

Voting

The simplest way to combine multiple classifiers is by *voting*, which corresponds to taking a linear combination of the learn

ers, Refer figure 1.

$$y_i = \sum_j w_j d_{ji} \text{ where } w_j \geq 0, \sum_j w_j = 1$$

This is also known as *ensembles* and *linear opinion pools*. In the sim plest case, all learners are given equal weight and we have *simple voting* that corresponds to taking an average. Still, taking a (weighted) sum is only one of the possibilities and there are also other combination rules, as shown in table 1. If the outputs are not posterior probabilities, these rules require that outputs be normalized to the same scale

Table 1 - Classifier combination rules

Rule	Fusion function $f(\cdot)$
Sum	$y_i = \frac{1}{L} \sum_{j=1}^{L} d_{ji}$
Weighted sum	$y_i = \sum_j w_j d_{ji}, w_j \geq 0, \sum_j w_j = 1$
Median	$y_i = \text{median}_j d_{ji}$
Minimum	$y_i = \min_j d_{ji}$
Maximum	$y_i = \max_j d_{ji}$
Product	$y_i = \prod_j d_{ji}$

An example of the use of these rules is shown in table 2, which demonstrates the effects of different rules. Sum rule is the most intuitive and is the most widely used in practice. Median rule is more robust to outliers; minimum and maximum rules are pessimistic and optimistic, respectively. With the product rule, each learner has veto power; regardless of the other ones, if one learner has an output of 0, the overall output goes to 0. Note that after the combination rules, y_i do not necessarily sum up to 1.

Table 2: Example of combination rules on three learners and three classes

	C_1	C_2	C_3
d_1	0.2	0.5	0.3
d_2	0.0	0.6	0.4
d_3	0.4	0.4	0.2
Sum	0.2	**0.5**	0.3
Median	0.2	**0.5**	0.4
Minimum	0.0	**0.4**	0.2
Maximum	0.4	**0.6**	0.4
Product	0.0	**0.12**	0.032

In weighted sum, d_{ji} is the vote of learner j for class C_i and w_j is the weight of its vote. Simple voting is a special case where all voters have equal weight, namely, $wj = 1/L$. In classification, this is called *plurality voting* where the class having the maximum number of votes is the winner.

When there are two classes, this is *majority voting* where the winning class gets more than half of the votes. If the voters can also supply the additional information of how much they vote for each class (e.g., by the posterior probability), then after normalization, these can be used as weights in a *weighted voting* scheme. Equivalently, if d_{ji} are the class posterior probabilities, $P(C_i \mid x, M_j)$, then we can just sum them up ($w_j = 1/L$) and choose the class with maximum y_i.

In the case of regression, simple or weighted averaging or median can be used to fuse the outputs of base-regressors. Median is more robust to noise than the average.

Another possible way to find w_j is to assess the accuracies of the learners (regressor or classifier) on a separate validation set and use that information to compute the weights, so that we give more weights to more accurate learners.

Voting schemes can be seen as approximations under a Bayesian framework with weights approximating prior model probabilities, and model decisions approximating model-conditional likelihoods.

$$P(C_i|x) = \sum_{\text{all models } \mathcal{M}_j} P(C_i|x, \mathcal{M}_j)P(\mathcal{M}_j)$$

Simple voting corresponds to a uniform prior. If we have a prior distribution preferring simpler models, this would give larger weights to them. We cannot integrate over all models; we only choose a subset for which we believe $P(M_j)$ is high, or we can have another Bayesian step and calculate $P(C_i \mid x, M_j)$, the probability of a model given the sample, and sample high probable models from this density.

Let us assume that d_j are iid with expected value $E[d_j]$ and variance $Var(d_j)$, then when we take a simple

63

average with $w_j = 1/L$, the expected value and variance of the output are

$$E[y] \;=\; E\left[\sum_j \frac{1}{L}d_j\right] = \frac{1}{L}LE[d_j] = E[d_j]$$

$$\mathrm{Var}(y) \;=\; \mathrm{Var}\left(\sum_j \frac{1}{L}d_j\right) = \frac{1}{L^2}\mathrm{Var}\left(\sum_j d_j\right) = \frac{1}{L^2}L\mathrm{Var}(d_j) = \frac{1}{L}\mathrm{Var}(d_j)$$

We see that the expected value does not change, so the bias does not change. But variance, and therefore mean square error, decreases as the number of independent voters, L, increases. In the general case,

$$\mathrm{Var}(y) = \frac{1}{L^2}\mathrm{Var}\left(\sum_j d_j\right) = \frac{1}{L^2}\left[\sum_j \mathrm{Var}(d_j) + 2\sum_j \sum_{i<j} \mathrm{Cov}(d_j, d_i)\right]$$

which implies that if learners are positively correlated, variance (and error) increase. We can thus view using different algorithms and input features as efforts to decrease, if not completely eliminate, the positive correlation.

Error-Correcting Output Codes

The **Error-Correcting Output Codes** method is a technique that allows a multi-class classification problem to be reframed as multiple binary classification problems, allowing the use of native binary classification models to be used directly.

Unlike [one-vs-rest and one-vs-one methods](#) that offer a similar solution by dividing a multi-class classification problem into a fixed number of binary classification problems, the error-correcting output codes technique allows each class to be encoded as an arbitrary number of binary classification problems. When an overdetermined representation is used, it allows the extra models to act as "error-correction" predictions that can result in better predictive performance.

In *error-correcting output codes* (ECOC), the main classification task is defined in terms of a number of subtasks that are implemented by the base-learners. The idea is that the original task of separating one class from all other classes may be a difficult problem. Instead, we want to define a set of simpler classification problems, each specializing in one aspect of the task, and combining these simpler classifiers, we get the final classifier.

Base-learners are binary classifiers having output $-1/+1$, and there is a *code matrix* **W** of $K \times L$ whose K rows are the binary codes of classes in terms of the L base-learners d_j. For example, if the second row of **W** is $[-1,+1,+1,-1]$, this means that for us to say an instance belongs to C_2, the instance should be on the negative side of d_1 and d_4, and on the positive side of d_2 and d_3. Similarly, the columns of the code matrix defines the task of the base-learners. For example, if the third column is $[-1,+1,+1]T$, we understand that the task of the third base-learner, d_3, is to separate the instances of C_1 from the instances of C_2 and C_3 combined. This is how we form the training set of the base-learners. For example in this case, all instances labeled with C2 and C3 form $X^\pm_3$ and instances labeled with C_1 form $X^=_3$, and d_3 is trained so that $x^t \in X^\pm_3$ give output $+1$ and $x^t \in X^=_3$ give output -1.

The code matrix thus allows us to define a polychotomy ($K > 2$ classification problem) in terms of dichotomies ($K = 2$ classification problem), and it is a method that is applicable using any learning

algorithm to implement the dichotomizer base-learners—for example, linear or multilayer perceptrons (with a single output), decision trees, or SVMs whose original definition is for two-class problems.

The typical one discriminant per class setting corresponds to the diagonal code matrix where $L = K$. For example, for $K = 4$,

we have

$$W = \begin{bmatrix} +1 & -1 & -1 & -1 \\ -1 & +1 & -1 & -1 \\ -1 & -1 & +1 & -1 \\ -1 & -1 & -1 & +1 \end{bmatrix}$$

The problem here is that if there is an error with one of the baselearners, there may be a misclassification because the class code words are so similar. So the approach in error-correcting codes is to have $L > K$ and increase the Hamming distance between the code words. One possibility is *pairwise separation* of classes where there is a separate baselearner to separate C_i from C_j, for $i < j$. In this case, $L = K(K − 1)/2$ and with $K = 4$, the code matrix is

$$W = \begin{bmatrix} +1 & +1 & +1 & 0 & 0 & 0 \\ -1 & 0 & 0 & +1 & +1 & 0 \\ 0 & -1 & 0 & -1 & 0 & +1 \\ 0 & 0 & -1 & 0 & -1 & -1 \end{bmatrix}$$

where a 0 entry denotes "don't care." That is, d_1 is trained to separate C_1 from C_2 and does not use the training instances belonging to the other classes. Similarly, we say that an instance belongs to C_2 if $d_1 = −1$ and $d_4 = d_5 = +1$, and we do not consider the values of d_2, d_3, and d_6. The problem here is that L is $O(K^2)$, and for large K pairwise separation may not be feasible.

If we can have L high, we can just randomly generate the code matrix with $−1/ + 1$ and this will work fine, but if we want to keep L low, we need to optimize **W**. The approach is to set L beforehand and then find **W** such that the distances between rows, and at the same time the distances between columns, are as large as possible, in terms of Hamming distance. With K classes, there are $2^{(K-1)} − 1$ possible columns, namely, two-class problems. This is because K bits can be written in $2K$ different ways and complements (e.g., "0101" and "1010," from our point of view, define the same discriminant) dividing the possible combinations by 2 and then subtracting 1 because a column of all 0s (or 1s) is useless. For example, when $K = 4$, we have

$$W = \begin{bmatrix} -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & +1 & +1 & +1 & +1 \\ -1 & +1 & +1 & -1 & -1 & +1 & +1 \\ +1 & -1 & +1 & -1 & +1 & -1 & +1 \end{bmatrix}$$

When K is large, for a given value of L, we look for L columns out of the $2^{(K-1)} − 1$. We would like these columns of **W** to be as different as possible so that the tasks to be learned by the base-learners are as different from each other as possible. At the same time, we would like the rows of **W** to be as different as possible so that we can have maximum error correction in case one or more base-learners fail.

ECOC can be written as a voting scheme where the entries of **W**, w_{ij}, are considered as vote weights:

$$y_i = \sum_{j=1}^{L} w_{ij} d_j$$

and then we choose the class with the highest y_i. Taking a weighted sum and then choosing the maximum instead of checking for an exact match allows d_j to no longer need to be binary but to take a value between −1 and +1, carrying soft certainties instead of hard decisions. Note that a value p_j between 0 and 1, for example, a posterior probability, can be converted to a value d_j between −1 and +1 simply as

$$d_j = 2p_j - 1$$

One problem with ECOC is that because the code matrix **W** is set a priori, there is no guarantee that the subtasks as defined by the columns of **W** will be simple.

Bagging

Bootstrap aggregating, often abbreviated as bagging, involves having each model in the ensemble vote with equal weight. In order to promote model variance, bagging trains each model in the ensemble using a randomly drawn subset of the training set. As an example, the random forest algorithm combines random decision trees with bagging to achieve very high classification accuracy.

The simplest method of combining classifiers is known as bagging, which stands for bootstrap aggregating, the statistical description of the method. This is fine if you know what a bootstrap is, but fairly useless if you don't. A bootstrap sample is a sample taken from the original dataset with replacement, so that we may get some data several times and others not at all. The bootstrap sample is the same size as the original, and lots and lots of these samples are taken: *B* of them, where *B* is at least 50, and could even be in the thousands. The name bootstrap is more popular in computer science than anywhere else, since there is also a bootstrap loader, which is the first program to run when a computer is turned on. It comes from the nonsensical idea of 'picking yourself up by your bootstraps,' which means lifting yourself up by your shoelaces, and is meant to imply starting from nothing.

Bootstrap sampling seems like a very strange thing to do. We've taken a perfectly good dataset, mucked it up by sampling from it, which might be good if we had made a smaller dataset (since it would be faster), but we still ended up with a dataset the same size. Worse, we've done it lots of times. Surely this is just a way to burn up computer time without gaining anything. The benefit of it is that we will get lots of learners that perform slightly differently, which is exactly what we want for an ensemble method. Another benefit is that estimates of the accuracy of the classification function can be made without complicated analytic work, by throwing computer resources at the problem (technically, bagging is a variance reducing algorithm; the meaning of this will become clearer when we talk about bias and variance). Having taken a set of bootstrap samples, the bagging method simply requires that we fit a model to each dataset, and then combine them by taking the output to be the majority vote of all the classifiers. A NumPy implementation is shown next, and then we will look at a simple example.

```
# Compute bootstrap samples
samplePoints = np.random.randint(0,nPoints,(nPoints,nSamples))
classifiers = []

for i in range(nSamples):
        sample = []
        sampleTarget = []
for j in range(nPoints):
        sample.append(data[samplePoints[j,i]])
```

```
        sampleTarget.append(targets[samplePoints[j,i]])
# Train classifiers
classifiers.append(self.tree.make_tree(sample,sampleTarget,features))
```

The example consists of taking the party data that was used to demonstrate the decision tree, and restricting the trees to stumps, so that they can make a classification based on just one variable

When we want to construct the decision tree to decide what to do in the evening, we start by listing everything that we've done for the past few days to get a suitable dataset (here, the last ten days):

Deadline?	Is there a party?	Lazy?	Activity
Urgent	Yes	Yes	Party
Urgent	No	Yes	Study
Near	Yes	Yes	Party
None	Yes	No	Party
None	No	Yes	Pub
None	Yes	No	Party
Near	No	No	Study
Near	No	Yes	TV
Near	Yes	Yes	Party
Urgent	No	No	Study

The output of a decision tree that uses the whole dataset for this is not surprising: it takes the two largest classes, and separates them. However, using just stumps of trees and 20 samples, bagging can separate the data perfectly, as this output shows:

```
Tree Stump Prediction
['Party', 'Party', 'Party', 'Party', 'Pub', 'Party', 'Study', 'Study',
'Party', 'Study']
Correct Classes
['Party', 'Study', 'Party', 'Party', 'Pub', 'Party', 'Study', 'TV', 'Party',
'Study']
Bagged Results
['Party', 'Study', 'Party', 'Party', 'Pub', 'Party', 'Study', 'TV', 'Party',
'Study']
```

RANDOM FORESTS

A random forest is an ensemble learning method where multiple decision trees are constructed and then they are merged to get a more accurate prediction.

If there is one method in machine learning that has grown in popularity over the last few years, then it is the idea of random forests. The concept has been around for longer than that, with several different people inventing variations, but the name that is most strongly attached to it is that of Breiman, who also described the CART algorithm in unit 2.

Figure 3: Example of random forest with majority voting

Test Instance
Tree-1
Tree-2
Tree-n
Class-A
Class-B
Class-B
Majority-Voting
Final-Class
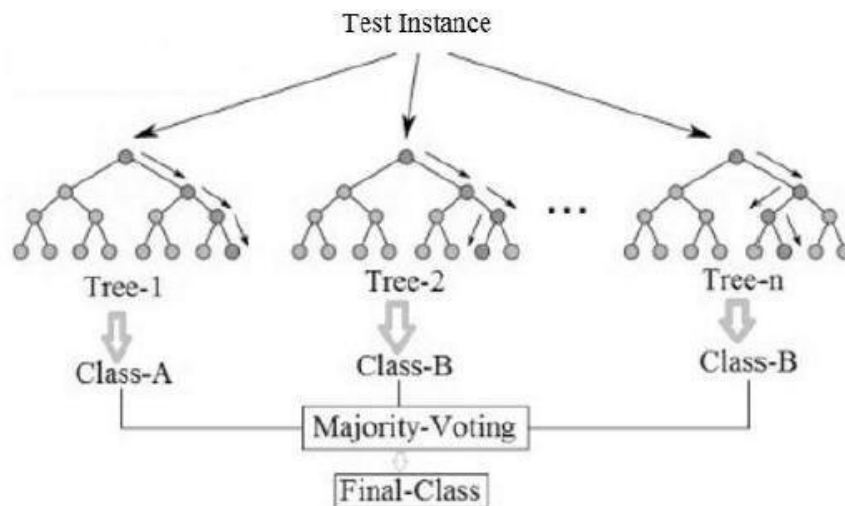
The idea is largely that if one tree is good, then many trees (a forest) should be better, provided that there is enough variety between them. The most interesting thing about a random forest is the ways that it creates randomness from a standard dataset. The first of the methods that it uses is the one that we have just seen: bagging. If we wish to create a forest then we can make the trees different by training them on slightly different data, so we take bootstrap samples from the dataset for each tree. However, this isn't enough randomness yet. The other obvious place where it is possible to add randomness is to limit the choices that the decision tree can make. At each node, a random subset of the features is given to the tree, and it can only pick from that subset rather than from the whole set.

As well as increasing the randomness in the training of each tree, it also speeds up the training, since there are fewer features to search over at each stage. Of course, it does introduce a new parameter (how many features to consider), but the random forest does not seem to be very sensitive to this parameter; in practice, a subset size that is the square root of the number of features seems to be common. The effect of these two forms of randomness is to reduce the variance without effecting the bias. Another benefit of this is that there is no need to prune the trees. There is another parameter that we don't know how to choose yet, which is the number of trees to put into the forest. However, this is fairly easy to pick if we want optimal results: we can keep on building trees until the error stops decreasing.

Once the set of trees are trained, the output of the forest is the majority vote for classification, as with the other committee methods that we have seen, or the mean response for regression. And those are pretty much the main features needed for creating a random forest. The algorithm is given next before we see some results of using the random forest.

Algorithm
Here is an outline of the random forest algorithm.
1. The random forests algorithm generates many classification trees. Each tree is generated as follows:
    a) If the number of examples in the training set is N, take a sample of N examples at random - but with replacement, from the original data. This sample will be the training set for generating the tree.
    b) If there are M input variables, a number m is specified such that at each node, m variables are selected at random out of the M and the best split on these m is used to

split the node. The value of m is held constant during the generation of the various trees in the forest.

   c)   Each tree is grown to the largest extent possible.

2. To classify a new object from an input vector, put the input vector down each of the trees in the forest. Each tree gives a classification, and we say the tree "votes" for that class. The forest chooses the classification

The implementation of this is very easy: we modify the decision to take an extra parameter, which is *m*, the number of features that should be used in the selection set at each stage. We will look at an example of using it shortly as a comparison to boosting.

Looking at the algorithm you might be able to see that it is a very unusual machine learning method because it is embarrassingly parallel: since the trees do not depend upon each other, you can both create and get decisions from different trees on different individual processors if you have them. This means that the random forest can run on as many processors as you have available with nearly linear speedup.

There is one more nice thing to mention about random forests, which is that with a little bit of programming effort they come with built-in test data: the bootstrap sample will miss out about 35% of the data on average, the so-called out-of-bootstrap examples. If we keep track of these datapoints then they can be used as novel samples for that particular tree, giving an estimated test error that we get without having to use any extra datapoints.

This avoids the need for cross-validation.

As a brief example of using the random forest, we start by demonstrating that the random forest gets the correct results on the Party example that has been used in both this and the previous chapters, based on 10 trees, each trained on 7 samples, and with just two levels allowed in each tree:

```
RF prediction
['Party', 'Study', 'Party', 'Party', 'Pub', 'Party', 'Study', 'TV', 'Party', 
'Study']
```

As a rather more involved example, the car evaluation dataset in the UCI Repository contains 1,728 examples aiming to classify whether or not a car is a good purchase based on six attributes. The following results compare a single decision tree, bagging, and a random forest with 50 trees, each based on 100 samples, and with a maximum depth of five for each tree. It can be seen that the random forest is the most accurate of the three methods.

```
Tree
Number correctly predicted 777.0
Number of testpoints  864
Percentage Accuracy  89.9305555556

Number of cars rated as good or very good 39.0
Number correctly identified as good or very good 18.0
Percentage Accuracy 46.1538461538
-----
Bagger
Number correctly predicted 678.0
Number of testpoints  864
Percentage Accuracy  78.4722222222

Number of cars rated as good or very good 39.0
Number correctly identified as good or very good 0.0
Percentage Accuracy 0.0
-----
Forest
Number correctly predicted 793.0
Number of testpoints  864
Percentage Accuracy  91.7824074074

Number of cars rated as good or very good 39.0
Number correctly identified as good or very good 20.0
Percentage Accuracy 51.28205128
```

Strengths and weaknesses

Strengths
The following are some of the important strengths of random forests.
- It runs efficiently on large data bases.
- It can handle thousands of input variables without variable deletion.
- It gives estimates of what variables are important in the classification.
- It has an effective method for estimating missing data and maintains accuracy when a large proportion of the data are missing.
- Generated forests can be saved for future use on other data.
- Prototypes are computed that give information about the relation between the variables and the classification.
- The capabilities of the above can be extended to unlabeled data, leading to unsupervised clustering, data views and outlier detection.
- It offers an experimental method for detecting variable interactions.
- Random forest run times are quite fast, and they are able to deal with unbalanced and missing data.
- They can handle binary features, categorical features, numerical features without any need for scaling.

Weaknesses
- A weakness of random forest algorithms is that when used for regression they cannot predict beyond the range in the training data, and that they may over-fit data sets that are particularly noisy.

- The sizes of the models created by random forests may be very large. It may take hundreds of megabytes of memory and may be slow to evaluate.
- Random forest models are black boxes that are very hard to interpret.

Boosting
- **Boosting: train next learner on mistakes made by previous learner(s)**

In bagging, generating complementary base-learners is left to chance and to the unstability of the learning method. In boosting, we actively try to generate complementary base-learners by training the next learner on the mistakes of the previous learners. The original *boosting* algorithm combines three weak learners to generate a strong learner. A *weak learner* has error probability less than 1/2, which makes it better than random guessing on a two-class problem, and a *strong learner* has arbitrarily small error probability.

Original Boosting Concept
Given a large training set, we randomly divide it into three. We use X1 and train $d1$. We then take X2 and feed it to $d1$. We take all instances misclassified by $d1$ and also as many instances on which $d1$ is correct
from X2, and these together form the training set of $d2$. We then take X3 and feed it to $d1$ and $d2$. The instances on which $d1$ and $d2$ disagree form the training set of $d3$. During testing, given an instance, we give it to $d1$ and $d2$; if they agree, that is the response, otherwise the response of $d3$ is taken as the output.

1. Split data X into {X_1, X_2, X_3}
2. Train d1 on X_1
    - Test d_1 on X_2
3. Train d_2 on d_1's mistakes on X_2 (plus some right)
    - Test d_1 and d_2 on X_3
4. Train d_3 on disagreements between d_1 and d_2
- Testing: apply d_1 and d_2; if disagree, use d_3
- Drawback: need large X

overall system has reduced error rate, and the error rate can arbitrarily be reduced by using such systems recursively, that is, a boosting system of three models used as dj in a higher system.

Though it is quite successful, the disadvantage of the original boosting method is that it requires a very large training sample. The sample should be divided into three and furthermore, the second and third classifiers are only trained on a subset on which the previous ones err. So unless one has a quite large training set, d_2 and d_3 will not have training
sets of reasonable size.

AdaBoost
Freund and Schapire (1996) proposed a variant, named *AdaBoost*, short for adaptive boosting, that uses the same training set over and over and thus need not be large, but the classifiers should be simple so that they do not overfit. AdaBoost can also combine an arbitrary number of baselearners, not three.

AdaBoost algorithm

```
Training:
    For all {x^t, r^t}_{t-1}^N ∈ X, initialize p_1^t = 1/N
    For all base-learners j = 1,...,L
        Randomly draw X_j from X with probabilities p_j^t
        Train d_j using X_j
        For each (x^t, r^t), calculate y_j^t ← d_j(x^t)
        Calculate error rate: ε_j ← Σ_t p_j^t · 1(y_j^t ≠ r^t)
        If ε_j > 1/2, then L ← j - 1; stop
        β_j ← ε_j/(1 - ε_j)
        For each (x^t, r^t), decrease probabilities if correct:
            If y_j^t = r^t, then p_{j+1}^t ← β_j p_j^t  Else p_{j+1}^t ← p_j^t
        Normalize probabilities:
            Z_j ← Σ_t p_{j+1}^t;  p_{j+1}^t ← p_{j+1}^t/Z_j
Testing:
    Given x, calculate d_j(x), j = 1,...,L
    Calculate class outputs, i = 1,...,K:
        y_i = Σ_{j-1}^L (log 1/β_j) d_{ji}(x)
```

The idea is to modify the probabilities of drawing the instances as a function of the error. Let us say p_j^t denotes the probability that the instance pair (x^t, r^t) is drawn to train the jth base-learner.

Initially, all $p_1^t = 1/N$. Then we add new base-learners as follows, starting from $j = 1$: ϵ denotes the error rate of d_j .

AdaBoost requires that learners are weak, that is, $\epsilon_j < 1/2, \forall j$; if not, we stop adding new base-learners. Note that this error rate is not on the original problem but on the dataset used at step j. We define $\beta_j = \epsilon_j /(1 - \epsilon_j) < 1$, and we set $p_{j+1}^t = \beta_j p_j^t$ if d_j correctly classifies xt ; otherwise, $p_{j+1}^t = p_j^t$.

Because p_{j+1}^t should be probabilities, there is a normalization where we divide p_{j+1}^t by $\sum_t p_{j+1}^t$, so that they sum up to 1. This has the effect that the probability of a correctly classified instance is decreased, and the probability of a misclassified instance increases. Then a new sample of the same size is drawn from the original sample according to these modified probabilities, p_{j+1}^t with replacement, and is used to train d$_{j+1}$.

This has the effect that d_{j+1} focuses more on instances misclassified by d_j ; that is why the base-learners are chosen to be simple and not accurate, since otherwise the next training sample would contain only a few outlier and noisy instances repeated many times over. For example, with decision trees, *decision stumps*, which are trees grown only one or two levels, are used. So it is clear that these would have bias but the decrease in variance is larger and the overall error decreases. An algorithm like the linear discriminant has low variance, and we cannot gain by AdaBoosting linear discriminants.
.

Stacking - Stacked Generalization
Stacked generalization is a technique proposed by Wolpert (1992) that extends voting in that the way the output of the base-learners is combined need not be linear but is learned through a combiner system, $f (\cdot|\Phi)$, which is another learner, whose parameters Φ are also trained. (see the below given figure)
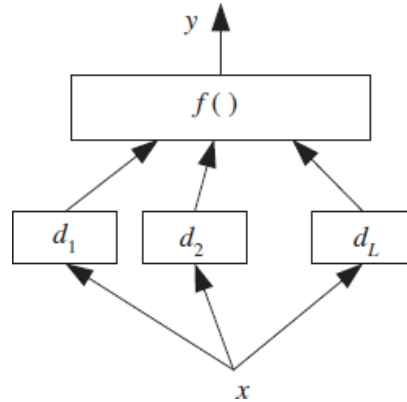
Figure: In stacked generalization, the combiner is another learner and is not restricted to being a linear combination as in voting.

$$y = f(d_1, d_2, \ldots, d_L | \Phi)$$

The combiner learns what the correct output is when the base-learners give a certain output combination. We cannot train the combiner function on the training data because the base-learners may be memorizing the training set; the combiner system should actually learn how the baselearners make errors. Stacking is a means of estimating and correcting for the biases of the base-learners. Therefore, the combiner should be trained on data unused in training the base-learners.

If $f(\cdot | w1, \ldots, wL)$ is a linear model with constraints, $wi \geq 0$, $\sum_j W_j = 1$, the optimal weights can be found by constrained regression, but of course we do not need to enforce this; in stacking, there is no restriction on the combiner function and unlike voting, $f(\cdot)$ can be nonlinear. For example, it may be implemented as a multilayer perceptron with Φ its connection weights.

The outputs of the base-learners d_j define a new L-dimensional space in which the output discriminant/regression function is learned by the combiner function.

In stacked generalization, we would like the base-learners to be as different as possible so that they will complement each other, and, for this, it is best if they are based on different learning algorithms. If we are combining classifiers that can generate continuous outputs, for example, posterior probabilities, it is better that they be the combined rather than hard decisions.
    When we compare a trained combiner as we have in stacking, with a fixed rule such as in voting, we see that both have their advantages: A trained rule is more flexible and may have less bias, but adds extra parameters, risks introducing variance, and needs extra time and data for training. Note also that there is no need to normalize classifier outputs before stacking.

Probabilistic Learning
    In machine learning, a **probabilistic classifier** is a classifier that is able to predict, given an observation of an input, a probability distribution over a set of classes, rather than only outputting the most likely class that the observation should belong to. Probabilistic classifiers provide classification that can be useful in its own right or when combining classifiers into ensembles.

One criticism that is often made of neural networks—especially the MLP—is that it is not clear exactly what it is doing: while we can go and have a look at the activations of the neurons and the weights, they don't tell us much.

73

In this topic (**probabilistic classifier**) we are going to look at methods that are based on statistics, and that are therefore more transparent, in that we can always extract and look at the probabilities and see what they are, rather than having to worry about weights that have no obvious meaning.

GAUSSIAN MIXTURE MODELS

However, suppose that we have the same data, but without target labels. This requires unsupervised learning, Suppose that the different classes each come from their own Gaussian distribution. This is known as multi-modal data, since there is one distribution (mode) for each different class. We can't fit one Gaussian to the data, because it doesn't look Gaussian overall.

There is, however, something we can do. If we know how many classes there are in the data, then we can try to estimate the parameters for that many Gaussians, all at once. If we don't know, then we can try different numbers and see which one works best. We will talk about this issue more for a different method (the *k*-means algorithm) in Unit 2. It is perfectly possible to use any other probability distribution instead of a Gaussian, but Gaussians are by far the most common choice. Then the output for any particular datapoint that is input to the algorithm will be the sum of the values expected by all of the *M* Gaussians:

$$f(\mathbf{x}) = \sum_{m=1}^{M} \alpha_m \phi(\mathbf{x}; \boldsymbol{\mu}_m, \boldsymbol{\Sigma}_m),$$

where $\phi(\mathbf{x}; \boldsymbol{\mu}_m, \sum m)$ is a Gaussian function with mean $\boldsymbol{\mu}_m$ and covariance matrix $\sum m$, and the α_m are weights with the constraint that $\sum_{m=1}^{M} \alpha m = 1$.

The given figures 4 shows two examples, where the data (shown by the histograms) comes from two different Gaussians, and the model is computed as a sum or mixture of the two Gaussians together.
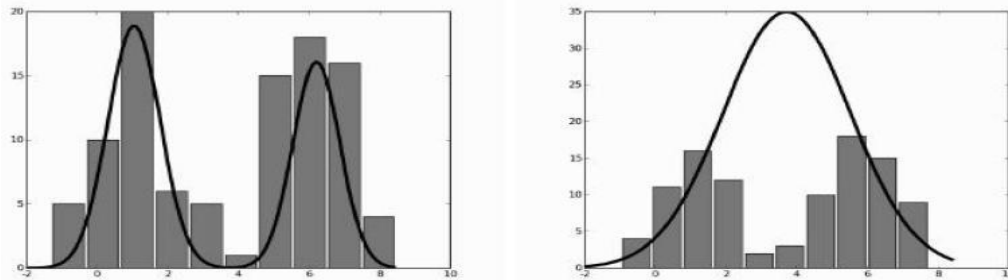

FIGURE 4: Histograms of training data from a mixture of two Gaussians and two fitted models, shown as the line plot. The model shown on the left fits well, but the one on the right produces two Gaussians right on top of each other that do not fit the data well.

The figure also gives you some idea of how to use the mixture model once it has been created. The probability that input $\mathbf{x}_i$ belongs to class *m* can be written as (where a hat on a variable (^·) means that we are estimating the value of that variable):

$$p(\mathbf{x}_i \in c_m) = \frac{\hat{\alpha}_m \phi(\mathbf{x}_i; \hat{\boldsymbol{\mu}}_m; \hat{\boldsymbol{\Sigma}}_m)}{\sum_{k=1}^{M} \hat{\alpha}_m \phi(\mathbf{x}_i; \hat{\boldsymbol{\mu}}_k; \hat{\boldsymbol{\Sigma}}_k)}.$$

The problem is how to choose the weights α_m. The common approach is to aim for the maximum

likelihood solution (the likelihood is the conditional probability of the data given the model, and the maximum likelihood solution varies the model to maximise this conditional probability). In fact, it is common to compute the log likelihood and then to maximise that; it is guaranteed to be negative, since probabilities are all less than 1, and the logarithm spreads out the values, making the optimisation more effective. The algorithm that is used is an example of a very general one known as the expectation-maximisation (or more compactly, EM) algorithm.

The Expectation-Maximisation (EM) Algorithm

The basic idea of the EM algorithm is that sometimes it is easier to add extra variables that are not actually known (called hidden or latent variables) and then to maximise the function over those variables. This might seem to be making a problem much more complicated than it needs to be, but it turns out for many problems that it makes finding the solution significantly easier.

In order to see how it works, we will consider the simplest interesting case of the Gaussian mixture model: a combination of just two Gaussian mixtures. The assumption now is that sample from that Gaussian. If the probability of picking Gaussian one is p, then the entire model looks like this (where N(μ, σ^2) specifies a Gaussian distribution with mean μ and standard deviation σ):

$$G_1 = \mathcal{N}(\mu_1, \sigma_1^2)$$
$$G_2 = \mathcal{N}(\mu_2, \sigma_2^2)$$
$$y = pG_1 + (1-p)G_2.$$

If the probability distribution of p is written as π, then the probability density is:

$$P(\mathbf{y}) = \pi\phi(\mathbf{y}; \mu_1, \sigma_1) + (1 - \pi)\phi(\mathbf{y}; \mu_2, \sigma_2).$$

Finding the maximum likelihood solution (actually the maximum log likelihood) to this problem is then a case of computing the sum of the logarithm of Equation over all of the training data, and differentiating it, which would be rather difficult. Fortunately, there is a way around it. The key insight that we need is that if we knew which of the two Gaussian components the datapoint came from, then the computation would be easy. The mean and standard deviation for each component could be computed from the datapoints that belong to that component, and there would not be a problem. Although we don't know which component each datapoint came from, we can pretend we do, by introducing a new variable f. If $f = 0$ then the data came from Gaussian one, if $f = 1$ then it came from Gaussian two.

This is the typical initial step of an EM algorithm: adding latent variables. Now we just need to work out how to optimise over them. This is the time when the reason for the algorithm being called expectation-maximisation becomes clear. We don't know much about variable f (hardly surprising, since we invented it), but we can compute its expectation (that is, the value that we 'expect' to see, which is the mean average) from the data:

$$\gamma_i(\hat{\mu}_1, \hat{\mu}_2, \hat{\sigma}_1, \hat{\sigma}_2, \hat{\pi}) = E(f|\hat{\mu}_1, \hat{\mu}_2, \hat{\sigma}_1, \hat{\sigma}_2, \hat{\pi}, D)$$
$$= P(f = 1|\hat{\mu}_1, \hat{\mu}_2, \hat{\sigma}_1, \hat{\sigma}_2, \hat{\pi}, D),$$

where D denotes the data. Note that since we have set $f = 1$ this means that we are choosing Gaussian two.

75

Computing the value of this expectation is known as the E-step. Then this estimate of the expectation is maximised over the model parameters (the parameters of the two Gaussians and the mixing parameter π), the M-step. This requires differentiating the expectation with respect to each of the model parameters. These two steps are simply iterated until the algorithm converges. Note that the estimate never gets any smaller, and it turns out that EM algorithms are guaranteed to reach a local maxima. To see how this looks for the two-component Gaussian mixture, we'll take a closer look at the algorithm:

The general algorithm has pretty much exactly the same steps (the parameters of the model are written as θ, θ' is a dummy variable, D is the original dataset, and D' is the dataset with the latent variables included):

The General Expectation-Maximisation (EM) Algorithm

- Initialisation
  - guess parameters $\hat{\theta}^{(0)}$
- Repeat until convergence:
  - (E-step) compute the expectation $Q(\theta', \hat{\theta}^{(j)}) = E(f(\theta'; D')|D, \hat{\theta}^{(j)})$
  - (M-step) estimate the new parameters $\hat{\theta}^{(j+1)}$ as $\max_{\theta'} Q(\theta', \hat{\theta}^{(j)})$

The trick with applying EM algorithms to problems is in identifying the correct latent variables to include, and then simply working through the steps. They are very powerful methods for a wide variety of statistical learning problems. We are now going to turn our attention to something much simpler, which is how we can use information about nearby datapoints to decide on classification output. For this we don't use a model of the data at all, but directly use the data that is available.

Information Criteria

we introduced the idea of a validation set, or using cross-validation if there was not enough data. However, this replaces data with computation time, as many models are trained on different datasets.

An alternative idea is to identify some measure that tells us about how well we can expect this trained model to perform. Probabilistic model selection (or "**information criteria**") provides an analytical technique for scoring and choosing among candidate models. Models are scored both on their performance on the training dataset and based on the complexity of the model.There are two such information criteria that are commonly used:

Aikake Information Criterium

$$AIC = \ln(\mathcal{L}) - k$$

Bayesian Information Criterium

$$BIC = 2\ln(\mathcal{L}) - k\ln N$$

In these equations, k is the number of parameters in the model, N is the number of training examples, and L is the best (largest) likelihood of the model. In both cases, based on the way that they are written here, the model with the largest value is taken.

Nearest neighbour methods

Suppose that you are in a nightclub and decide to dance. It is unlikely that you will know the dance moves for the particular song that is playing, so you will probably try to work out what to do by looking at what the people close to you are doing. The first thing you could do would be just to pick the person closest to you and copy them. However, since most of the people who are in the nightclub are also unlikely to know all the moves, you might decide to look at a few more people and do what most of them are doing. This is pretty much exactly the idea behind nearest neighbour methods: if we don't have a model that describes the data, then the best thing to do is to look at similar data and choose to be in the same class as them.

We have the datapoints positioned within input space, so we just need to work out which of the training data are close to it. This requires computing the distance to each datapoint in the training set, which is relatively expensive: if we are in normal Euclidean space, then we have to compute d subtractions and d squarings (we can ignore the square root since we only want to know which points are the closest, not the actual distance) and this has to be done $O(N^2)$ times. We can then identify the k nearest neighbours to the test point, and then set the class of the test point to be the most common one out of those for the nearest neighbours. The choice of k is not trivial. Make it too small and nearest neighbour methods are sensitive to noise, too large and the accuracy reduces as points that are too far away are considered. Some possible effects of changing the size of k on the decision boundary are shown in below Figure 5.
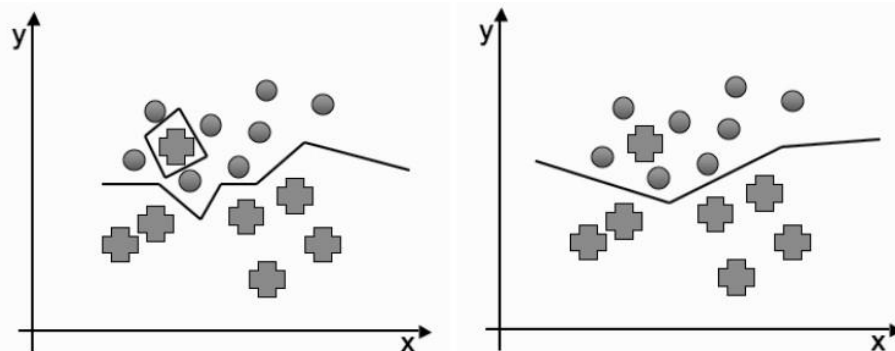

FIGURE 5: The nearest neighbours decision boundary with *left:* one neighbour and *right:* two neighbours.

This method suffers from the curse of dimensionality. First, as shown above, the computational costs get higher as the number of dimensions grows. This is not as bad as it might appear at first: there are sets of methods such as KD-Trees (will discuss in upcoming topics) that compute this in $O(N \log N)$ time. However, more importantly, as the number of dimensions increases, so the distance to other datapoints tends to increase. In addition, they can be far away in a variety of different directions—there might be points that are relatively close in some dimensions, but a long way in others. There are methods for dealing with these problems, known as adaptive nearest neighbour methods, and there is a reference to them in the Further Reading section at the end of the chapter.

The only part of this that requires any care during the implementation is what to do when there is more than one class found in the closest points, but even with that the implementation is nice and simple:

```
def knn(k,data,dataClass,inputs):

        nInputs = np.shape(inputs)[0]
        closest = np.zeros(nInputs)

        for n in range(nInputs):
                # Compute distances
                distances = np.sum((data-inputs[n,:])**2,axis=1)

                # Identify the nearest neighbours
                indices = np.argsort(distances,axis=0)

                classes = np.unique(dataClass[indices[:k]])
                if len(classes)==1:
                        closest[n] = np.unique(classes)
                else:
                        counts = np.zeros(max(classes)+1)
                        for i in range(k):
                                counts[dataClass[indices[i]]] += 1
                        closest[n] = np.max(counts)

        return closest
```

We are going to look next at how we can use these methods for regression, before we turn to the question of how to perform the distance calculations as efficiently as possible, something that is done simply but inefficiently in the code above. We will then consider briefly whether or not the Euclidean distance is always the most useful way to calculate distances, and what alternatives there are.

For the *k*-nearest neighbours algorithm the bias-variance decomposition can be computed as:

$$E((y - \hat{f}(x))^2) = \sigma^2 + \left[f(x) - \frac{1}{k}\sum_{i=0}^{k} f(x_i)\right]^2 + \frac{\sigma^2}{k}.$$

The way to interpret this is that when *k* is small, so that there are few neighbours considered, the model has flexibility and can represent the underlying model well, but that it makes mistakes (has high variance) because there is relatively little data. As *k* increases, the variance decreases, but at the cost of less flexibility and so more bias.

Nearest Neighbour Smoothing

Nearest neighbour methods can also be used for regression by returning the average value of the neighbours to a point, or a spline or similar fit as the new value. The most common methods are known as kernel smoothers, and they use a kernel (a weighting function between pairs of points) that decides how much emphasis (weight) to put onto the contribution from each datapoint according to its distance from the input.

Here we shall simply use two kernels that are used for smoothing. Both of these kernels are designed to give more weight to points that are closer to the current input, with the weights decreasing smoothly to zero as they pass out of the range of the current input, with the range specified by a parameter λ.

They are the Epanechnikov quadratic kernel:

$$K_{E,\lambda}(x_0, x) = \begin{cases} 0.75\left(1 - (x_0 - x)^2/\lambda^2\right) & \text{if } |x - x_0| < \lambda \\ 0 & \text{otherwise} \end{cases},$$

78

and the tricube kernel:

$$K_{T,\lambda}(x_0, x) = \begin{cases} \left(1 - \left|\frac{x_0 - x}{\lambda}\right|^3\right)^3 & \text{if } |x - x_0| < \lambda \\ 0 & \text{otherwise} \end{cases}.$$

The results of using these kernels are shown in below Figure 6 on a dataset that consists of the time between eruptions (technically known as the repose) and the duration of the eruptions of Mount Ruapehu, the large volcano in the centre of New Zealand's north island. Values of λ of 2 and 4 were used here. Picking λ requires experimentation. Large values average over more datapoints, and therefore produce lower variance, but at the cost of higher bias.
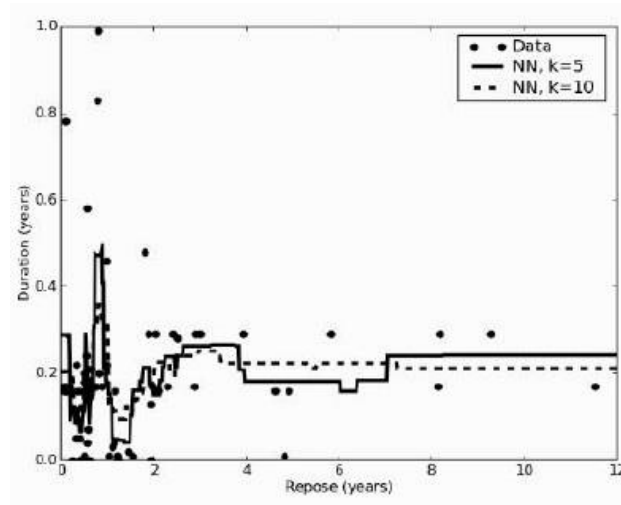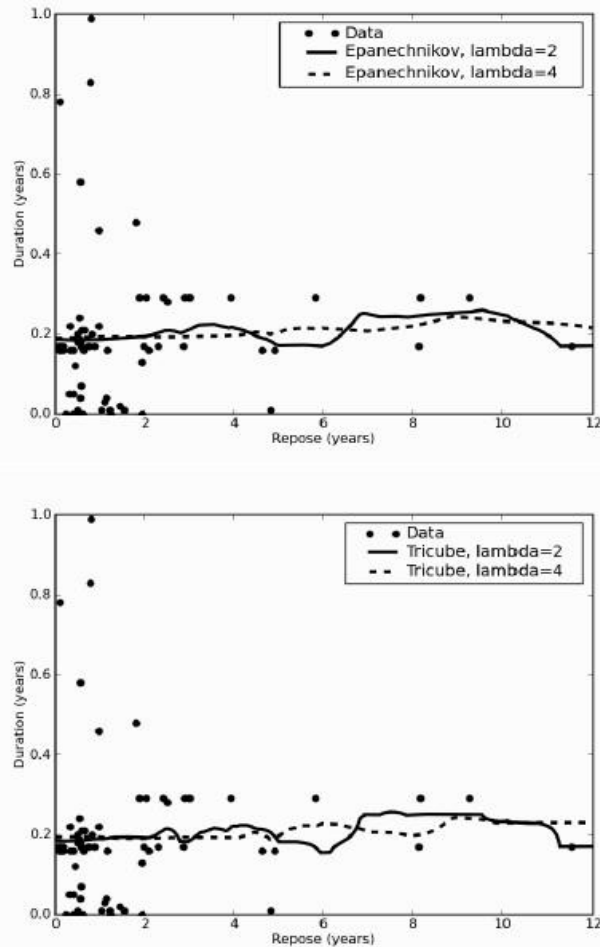
FIGURE 6: Output of the nearest neighbour method and two kernel smoothers on the data of duration and repose of eruptions of Mount Ruapehu 1860–2006.

Efficient Distance Computations: the KD-Tree

As was mentioned above, computing the distances between all pairs of points is very computationally expensive. Fortunately, as with many problems in computer science, designing an efficient data structure can reduce the computational overhead a lot. For the problem of finding nearest neighbours the data structure of choice is the KD-Tree. It has been around since the late 1970s, when it was devised by Friedman and Bentley, and it reduces the cost of finding a nearest neighbour to $O(\log N)$ for $O(N)$ storage. The construction of the tree is $O(N \log^2 N)$, with much of the computational cost being in the computation of the median, which with a naïve algorithm requires a sort and is therefore $O(N \log N)$, or can be computed with a randomised algorithm in $O(N)$ time.

The idea behind the KD-tree is very simple. You create a binary tree by choosing one dimension at a time to split into two, and placing the line through the median of the point coordinates of that dimension. The points themselves end up as leaves of the tree. Making the tree follows pretty much the same steps as usual for constructing a binary tree: we identify a place to split into two choices, left and right, and then carry on down the tree. This makes it natural to write the algorithm recursively. The choice of what to split and where is what makes the KD-tree special. Just one dimension is split in each step, and the position of the split is found by computing the median of the points that are to be split in that one dimension, and putting the line there. In general, the choice of which dimension to split

80

alternates through the different choices, or it can be made randomly. The algorithm below cycles through the possible dimensions based on the depth of the tree so far, so that in two dimensions it alternates horizontal and vertical splits.

The centre of the construction method is simply a recursive function that picks the axis to split on, finds the median value on that axis, and separates the points according to that value, which in Python can be written as:

```
# Pick next axis to split on
whichAxis = np.mod(depth,np.shape(points)[1])

# Find the median point
indices = np.argsort(points[:,whichAxis])
points = points[indices,:]
median = np.ceil(float(np.shape(points)[0]-1)/2)

# Separate the remaining points
goLeft = points[:median,:]
goRight = points[median+1:,:]

# Make a new branching node and recurse
newNode = node()
newNode.point = points[median,:]
newNode.left = makeKDtree(goLeft,depth+1)
newNode.right = makeKDtree(goRight,depth+1)
return newNode
```

Suppose that we had seven two-dimensional points to make a tree from: (5, 4), (1, 6), (6, 1), (7, 5), (2, 7), (2, 2), (5, 8) (as plotted in Figure 7).

FIGURE 7: The initial set of 2D data.
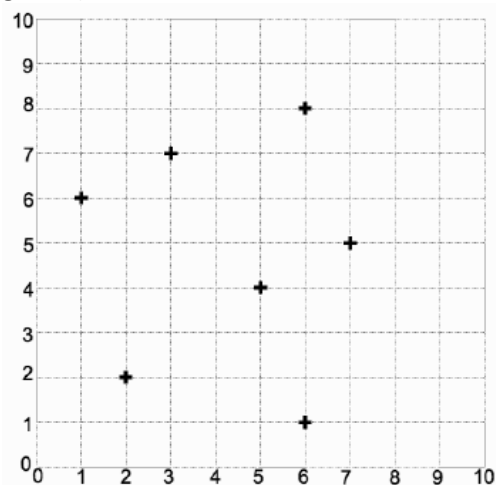
The algorithm will pick the first coordinate to split on initially, and the median point here is 5, so the split is through $x = 5$. Of those on the left of the line, the median y coordinate is 6, and for those on the right it is 5. At this point we have separated all the points, and so the algorithm terminates with the split shown in Figure 8 and the tree shown in Figure 9.

81
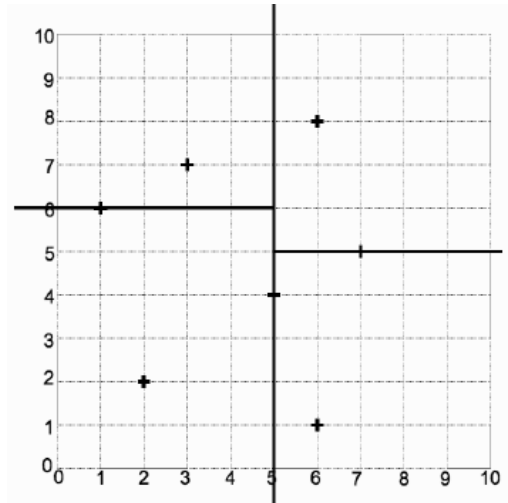
FIGURE 8: The splits and leaf points found by the KD-tree.
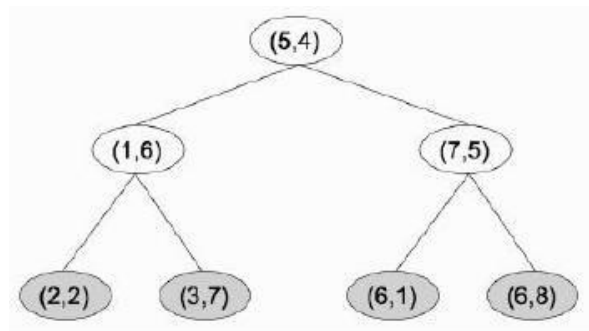
FIGURE 9: The KD-tree that made the splits.

Searching the tree is the same as any other binary tree; we are more interested in finding the nearest neighbours of a test point. This is fairly easy: starting at the root of the tree you recurse down through the tree comparing just one dimension at a time until you find a leaf node that is in the region containing the test point. Using the tree shown in Figure 9 we introduce the test point (3, 5), which finds (2, 2) as the leaf for the box that (3, 5) is in. However, looking at Figure 10 we see that this is not the closest point at all, so we need to do some more work.

FIGURE 10 Two test points for the example KD-tree.

Distance Measures

We have computed the distance between points as the Euclidean distance, which is something that you learnt about in high school. However, it is not the only option, nor is it necessarily the most useful. In this section we will look at the underlying idea behind distance calculations and possible alternatives.
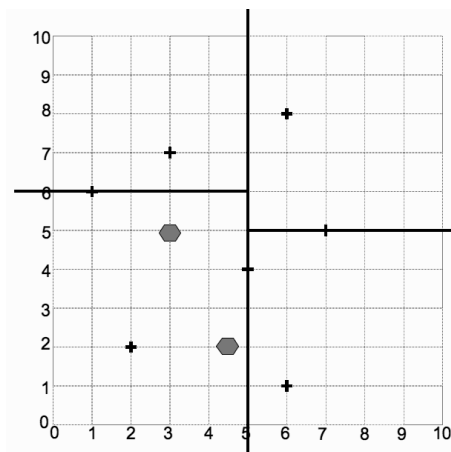
If I were to ask you to find the distance between my house and the nearest shop, then your first guess might involve taking a map of my town, locating my house and the shop, and using a ruler to measure the distance between them. By careful application of the map scale you can now tell me how far it is. However, when I set out to buy some milk I'm liable to find that I have to walk rather further than you've told me, since the direct line that you measured would involve walking through (or over) several houses, and some serious fence-scaling. Your 'as the crow flies' distance is the shortest possible path, and it is the straight-line, or Euclidean, distance. You can measure it on the map by just using a ruler, but it essentially consists of measuring the distance in one direction (we'll call it north-south) and then the distance in another direction that is perpendicular to the first (let's call it east-west) and then squaring them, adding them together, and then taking the square root of that. Writing that out, the Euclidean distance that we are all used to is:

$$d_E = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2},$$

where ($x1$, $y1$) is the location of my house in some coordinate system (say by using a GPS tracker) and ($x2$, $y2$) is the location of the shop.

If I told you that my town was laid out on a grid block system, as is common in towns that were built in the interval between the invention of the motor car and the invention of innovative town planners, then you would probably use a different measure. You would measure the distance between my house and the shop in the 'north-south' direction and the distance in the 'east-west' direction, and then add the two distances together. This would correspond to the distance I actually had to walk. It is often known as the city-block or Manhattan distance and looks like:

$$d_C = |x_1 - x_2| + |y_1 - y_2|.$$

The point of this discussion is to show that there is more than one way to measure a distance, and that they can provide radically different answers. These two different distances can be seen in Figure 11. Mathematically, these distance measures are known as metrics. A metric function or norm takes two inputs and gives a scalar (the distance) back, which is positive, and 0 if and only if the two points are the same, symmetric (so that the distance to the shop is the same as the distance back), and obeys the triangle inequality, which says that the distance from *a* to *b* plus the distance from *b* to *c* should not be less than the direct distance from *a* to *c*.
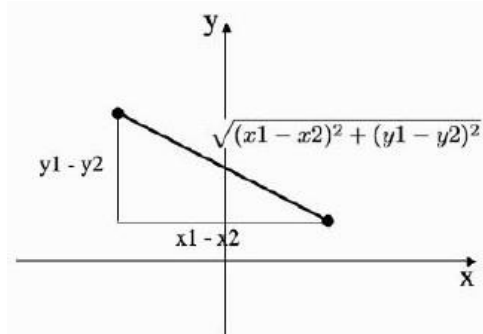
FIGURE 10: The Euclidean and city-block distances between two points.

Most of the data that we are going to have to analyse lives in rather more than two dimensions. Fortunately, the Euclidean distance that we know about generalises very well to higher dimensions (and so does the city-block metric). In fact, these two measures are both instances of a class of metrics that work in any number of dimensions. The general measure is the Minkowski metric and it is written as:

$$L_k(\mathbf{x}, \mathbf{y}) = \left(\sum_{i=1}^{d} |x_i - y_i|^k \right)^{\frac{1}{k}}.$$

If we put $k = 1$ then we get the city-block distance (Equation (7.12)), and $k = 2$ gives the Euclidean distance (Equation (7.11)). Thus, you might possibly see the Euclidean metric written as the $L2$ norm and the city-block distance as the $L1$ norm. These norms have another interesting feature. Remember that we can define different averages of a set of numbers. If we define the average as the point that minimises the sum of the distance to every datapoint, then it turns out that the mean minimises the Euclidean distance (the sum-of-squares distance), and the median minimises the $L1$ metric.

There are plenty of other possible metrics to choose, depending upon the dataspace. We generally assume that the space is flat (if it isn't, then none of these techniques work, and we don't want to worry about that). However, it can still be beneficial to look at other metrics. Suppose that we want our classifier to be able to recognise images, for example of faces. We take a set of digital photos of faces and use the pixel values as features. Then we use the nearest neighbour algorithm that we've just seen to identify each face. Even if we ensure that all of the photos are taken fully face-on, there are still a few things that will get in the way of this method. One is that slight variations in the angle of the head (or the camera) could make a difference; another is that different distances between the face and the camera (scaling) will change the results; and another is that different lighting conditions will make a difference. We can try to fix all of these things in preprocessing, but there is also another alternative: use a different metric that is invariant to these changes, i.e., it does not vary as they do. The idea of invariant metrics is to find measures that ignore changes that you don't want. So if you want to be able to rotate shapes around and still recognize them, you need a metric that is invariant to rotation.

A common invariant metric in use for images is the tangent distance, which is an approximation to the Taylor expansion in first derivatives, and works very well for small rotations and scalings; for example, it was used to halve the final error rate on nearest neighbor classification of a set of handwritten letters. Invariant metrics are an interesting topic for further study, and there is a reference for them in the Further Reading section if you are interested.

84

Introduction, Learning Task, Q Learning, Non deterministic Rewards and actions, temporal-difference learning, Relationship to Dynamic Programming, Active reinforcement learning, Generalization in reinforcement learning.
Motivation, Basics of Sampling Theory: Error Estimation and Estimating Binomial Proportions, The Binomial Distribution, Estimators, Bias, and Variance

Reinforcement learning addresses the question of how an autonomous agent that senses and acts in its environment can learn to choose optimal actions to achieve its goals.

Introduction

- Consider building a **learning robot**. The robot, or *agent*, has a set of sensors to observe the state of its environment, and a set of actions it can perform to alter this state.

- Its task is to learn a control strategy, or *policy*, for choosing actions that achieve its goals.

- The goals of the agent can be defined by a *reward function* that assigns a numerical value to each distinct action the agent may take from each distinct state.

- This reward function may be built into the robot, or known only to an external teacher who provides the reward value for each action performed by the robot.

- The **task** of the robot is to perform sequences of actions, observe their consequences, and learn a control policy.

- The control policy is one that, from any initial state, chooses actions that maximize the reward accumulated over time by the agent.

Example:

- A mobile robot may have sensors such as a camera and sonars, and actions such as "move forward" and "turn."

- The robot may have a goal of docking onto its battery charger whenever its battery level is low.

- The goal of docking to the battery charger can be captured by assigning a positive reward (Eg., +100) to state-action transitions that immediately result in a connection to the charger and a reward of zero to every other state-action transition.

Reinforcement Learning Problem

- An agent interacting with its environment. The agent exists in an environment described by some set of possible states S.

- Agent perform any of a set of possible actions A. Each time it performs an action a, in some state st the agent receives a real-valued reward r, that indicates the immediate value of this state-action transition. This produces a sequence of states si, actions ai, and immediate rewards ri as shown in the figure.

- The agent's task is to learn a control policy, $\pi: S \rightarrow A$, that maximizes the expected sum of these rewards, with future rewards discounted exponentially by their delay.
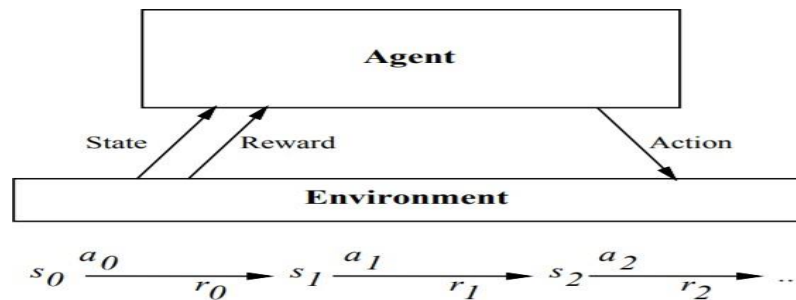
Goal: Learn to choose actions that maximize

$$r_0 + \gamma r_1 + \gamma^2 r_2 + \dots \;, \quad \text{where } 0 \le \gamma < 1$$

Reinforcement learning problem characteristics

1. Delayed reward: The task of the agent is to learn a target function π that maps from the current state s to the optimal action a = π (s). In reinforcement learning, training information is not available in (s, π (s)). Instead, the trainer provides only a sequence of immediate reward values as the agent executes its sequence of actions. The agent, therefore, faces the problem of *temporal credit assignment*: determining which of the actions in its sequence are to be credited with producing the eventual rewards.

2. Exploration: In reinforcement learning, the agent influences the distribution of training examples by the action sequence it chooses. This raises the question of which experimentation strategy produces most effective learning. The learner faces a trade-off in choosing whether to favor exploration of unknown states and actions, or exploitation of states and actions that it has already learned will yield high reward.

3. Partially observable states: The agent's sensors can perceive the entire state of the environment at each time step, in many practical situations sensors provide only partial information. In such cases, the agent needs to consider its previous observations together with its current sensor data when choosing actions, and the best policy may be one that chooses actions specifically to improve the observability of the environment.

4. Life-long learning: Robot requires to learn several related tasks within the same environment, using the same sensors. For example, a mobile robot may need to learn how to dock on its battery charger, how to navigate through narrow corridors, and how to pick up output from laser printers. This setting raises the possibility of using previously obtained experience or knowledge to reduce sample complexity when learning new tasks.

Learning Task
- Consider Markov decision process (MDP) where the agent can perceive a set S of distinct states of its environment and has a set A of actions that it can perform.

- At each discrete time step t, the agent senses the current state st, chooses a current action at, and performs it.

- The environment responds by giving the agent a reward rt = r(st, at) and by producing the succeeding state st+l = δ(st, at). Here the functions δ(st, at) and r(st, at) depend only on the current state and action, and not on earlier states or actions.

The task of the agent is to learn a policy, **π: S → A**, for selecting its next action a, based on the current observed state st; that is, **(st) = at**.

How shall we specify precisely which policy π we would like the agent to learn?

1. One approach is to require the policy that produces the greatest possible *cumulative reward* for the robot over time.

- To state this requirement more precisely, define the cumulative value Vπ (st) achieved by following an arbitrary policy π from an arbitrary initial state st as follows:

$$V^{\pi}(s_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots$$

$$\equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i} \qquad \text{equ (1)}$$

- Where, the sequence of rewards rt+i is generated by beginning at state st and by repeatedly using the policy π to select actions.

- Here $0 \leq \gamma \leq 1$ is a constant that determines the relative value of delayed versus immediate rewards. if we set γ = 0, only the immediate reward is considered. As we set γ closer to 1, future rewards are given greater emphasis relative to the immediate reward.

- The quantity Vπ (st) is called the *discounted cumulative reward* achieved by policy π from initial state *s*. It is reasonable to discount future rewards relative to immediate rewards because, in many cases, we prefer to obtain the reward sooner rather than later.

2. Other definitions of total reward is *finite horizon reward,*

$$\sum_{i=0}^{h} r_{t+i}$$

Considers the undiscounted sum of rewards over a finite number *h* of steps

3. Another approach is *average reward*

$$\lim_{h \to \infty} \frac{1}{h} \sum_{i=0}^{h} r_{t+i}$$

Considers the average reward per time step over the entire lifetime of the agent.

We require that the agent learn a policy π that maximizes Vπ (st) for all states s. such a policy is called an *optimal policy* and denote it by π*

$$\pi^* \equiv \underset{\pi}{\text{argmax}} \, V^\pi(s), \, (\forall s) \qquad \text{equ (2)}$$

Refer the value function Vπ*(s) an optimal policy as V*(s). V*(s) gives the maximum discounted cumulative reward that the agent can obtain starting from state **s**.

Example:
A simple grid-world environment is depicted in the diagram
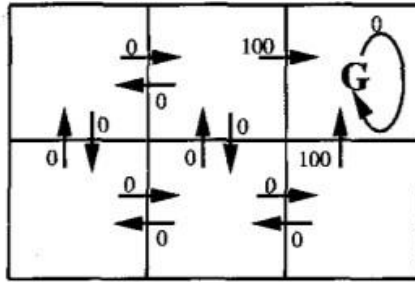
$r(s, a)$ (immediate reward) values

- The six grid squares in this diagram represent six possible states, or locations, for the agent.

- Each arrow in the diagram represents a possible action the agent can take to move from one state to another.

- The number associated with each arrow represents the immediate reward *r(s, a)* the agent receives if it executes the corresponding state-action transition

- The immediate reward in this environment is defined to be zero for all state-action transitions except for those leading into the state labelled G. The state G as the goal state, and the agent can receive reward by entering this state.

Once the states, actions, and immediate rewards are defined, choose a value for the discount factor γ, determine the optimal policy π * and its value function V*(s).

Let's choose γ = 0.9. The diagram at the bottom of the figure shows one optimal policy for this setting.
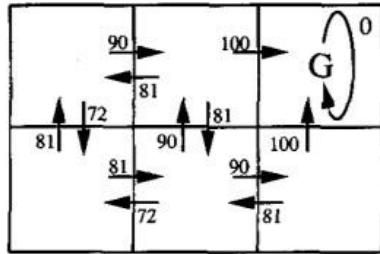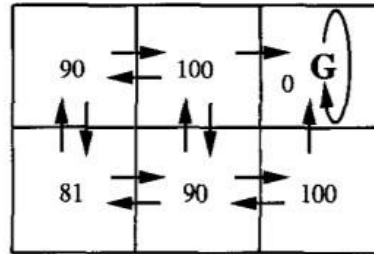
Q(s, a) values   V*(s) values
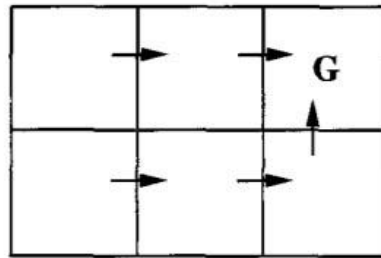
One optimal policy

Values of V*(s) and Q(s, a) follow from r(s, a), and the discount factor γ = 0.9. An optimal policy, corresponding to actions with maximal Q values, is also shown.

The discounted future reward from the bottom centre state is

$$0+ \gamma\ 100+ \gamma 2\ 0+ \gamma 3\ 0+... = 90$$

Q LEARNING

*How can an agent learn an optimal policy π * for an arbitrary environment?*

      The training information available to the learner is the sequence of immediate rewards $r(s_i, a_i)$ for i = 0, 1,2,.......... Given this kind of training information it is easier to learn a numerical evaluation function defined over states and actions, then implement the optimal policy in terms of this evaluation function.

What evaluation function should the agent attempt to learn?

One obvious choice is V*. The agent should prefer state sl over state s2 whenever V*(sl) > V*(s2), because the cumulative future reward will be greater from sl

The optimal action in state *s* is the action *a* that maximizes the sum of the immediate reward r(s, a) plus the value V* of the immediate successor state, discounted by γ.

$$\pi^*(s) = \underset{a}{\arg\max}[r(s, a) + \gamma V^*(\delta(s, a))] \qquad \text{equ (3)}$$

The Q Function

      The value of Evaluation function *Q(s, a)* is the reward received immediately upon executing action a from state *s*, plus the value (discounted by *γ*) of following the optimal policy thereafter

$$Q(s, a) \equiv r(s, a) + \gamma V^*(\delta(s, a)) \qquad \text{equ (4)}$$

Rewrite Equation (3) in terms of **Q(s, a)** as

$$\pi^*(s) = \operatorname*{argmax}_{a} Q(s, a) \qquad \text{equ (5)}$$

Equation (5) makes clear, it need only consider each available action **a** in its current state **s** and choose the action that maximizes **Q(s, a)**.

An Algorithm for Learning **Q**

- Learning the Q function corresponds to learning the **optimal policy**.

- The key problem is finding a reliable way to estimate training values for **Q**, given only a sequence of immediate rewards **r** spread out over time. This can be accomplished through *iterative approximation*

$$V^*(s) = \max_{a'} Q(s, a')$$

Rewriting Equation

$$Q(s, a) = r(s, a) + \gamma \max_{a'} Q(\delta(s, a), a')$$

Q learning algorithm:

Q learning algorithm

For each s, a initialize the table entry $\hat{Q}(s, a)$ to zero.
Observe the current state s
Do forever:

- Select an action a and execute it
- Receive immediate reward r
- Observe the new state s'
- Update the table entry for $\hat{Q}(s, a)$ as follows:

$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$$

- $s \leftarrow s'$

- Q learning algorithm assuming deterministic rewards and actions. The discount factor γ may be any constant such that $0 \leq \gamma < 1$

- Q to refer to the learner's estimate, or hypothesis, of the actual Q function

An Illustrative Example

- To illustrate the operation of the Q learning algorithm, consider a single action taken by an agent, and the corresponding refinement to Q shown in below figure

Initial state: s_1 Next state: s_2
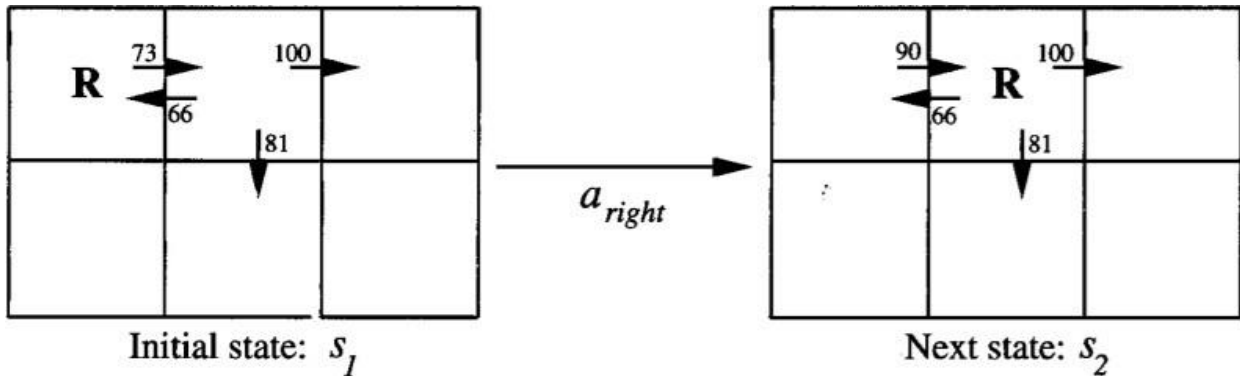
- The agent moves one cell to the right in its grid world and receives an immediate reward of zero for this transition.

- Apply the training rule of Equation

$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$$

to refine its estimate Q for the state-action transition it just executed.

- According to the training rule, the new Q estimate for this transition is the sum of the received reward (zero) and the highest Q value associated with the resulting state (100), discounted by γ (.9).

$$\hat{Q}(s_1, a_{right}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a')$$

$$\leftarrow 0 + 0.9 \ \max\{66, 81, 100\}$$

$$\leftarrow 90$$

Convergence
Will the Q Learning Algorithm converge toward a Q equal to the true Q function?
Yes, under certain conditions.
1. Assume the system is a deterministic MDP.

2. Assume the immediate reward values are bounded; that is, there exists some positive constant c such that for all states s and actions a, **| r(s, a)| < c**

3. Assume the agent selects actions in such a fashion that it visits every possible state-action pair infinitely often

Theorem Convergence of Q learning for deterministic Markov decision processes.

Consider a Q learning agent in a deterministic MDP with bounded rewards $(\forall s, a)|r(s, a)| \leq c$.

The Q learning agent uses the training rule of Equation $\hat{Q}(s, a) \leftarrow r + \gamma \max \hat{Q}(s', a')$ initializes its table $\hat{Q}(s, a)$ to arbitrary finite values, and uses a discount factor γ such that $0 \leq \gamma < 1$. Let $\hat{Q}_n(s, a)$ denote the agent's hypothesis $\hat{Q}(s, a)$ following the nth update. If each state-action pair is visited infinitely often, then $\hat{Q}_n(s, a)$ converges to $Q(s, a)$ as $n \to \infty$, for all s, a.

Proof. Since each state-action transition occurs infinitely often, consider consecutive intervals during which each state-action transition occurs at least once. The proof consists of showing that the maximum error over all entries in the $\hat{Q}$ table is reduced by at least a factor of γ during each such interval. $\hat{Q}_n$ is the agent's table of estimated Q values after n updates. Let Δ_n be the maximum error in $\hat{Q}_n$; that is

$$\Delta_n \equiv \max_{s,a} |\hat{Q}_n(s, a) - Q(s, a)|$$

Below we use s' to denote $\delta(s, a)$. Now for any table entry $\hat{Q}_n(s, a)$ that is updated on iteration $n + 1$, the magnitude of the error in the revised estimate $\hat{Q}_{n+1}(s, a)$ is

$$|\hat{Q}_{n+1}(s, a) - Q(s, a)| = |(r + \gamma \max_{a'} \hat{Q}_n(s', a')) - (r + \gamma \max_{a'} Q(s', a'))|$$

$$= \gamma |\max_{a'} \hat{Q}_n(s', a') - \max_{a'} Q(s', a')|$$

$$\leq \gamma \max_{a'} |\hat{Q}_n(s', a') - Q(s', a')|$$

$$\leq \gamma \max_{s'',a'} |\hat{Q}_n(s'', a') - Q(s'', a')|$$

$$|\hat{Q}_{n+1}(s, a) - Q(s, a)| \leq \gamma \Delta_n$$

The third line above follows from the second line because for any two functions f_1 and f_2 the following inequality holds

$$|\max_a f_1(a) - \max_a f_2(a)| \leq \max_a |f_1(a) - f_2(a)|$$

In going from the third line to the fourth line above, note we introduce a new variable s'' over which the maximization is performed. This is legitimate because the maximum value will be at least as great when we allow this additional variable to vary. Note that by introducing this variable we obtain an expression that matches the definition of Δ_n.

Thus, the updated $Q_{n+1}(s, a)$ for any s, a is at most γ times the maximum error in the $\hat{Q}_n$ table, Δ_n. The largest error in the initial table, Δ_0, is bounded because values of $\hat{Q}_0(s, a)$ and $Q(s, a)$ are bounded for all s, a. Now after the first interval during which each s, a is visited, the largest error in the table will be at most $\gamma \Delta_0$. After k such intervals, the error will be at most $\gamma^k \Delta_0$. Since each state is visited infinitely often, the number of such intervals is infinite, and $\Delta_n \to 0$ as $n \to \infty$. This proves the theorem.

Experimentation Strategies
The Q learning algorithm does not specify how actions are chosen by the agent.
- One obvious strategy would be for the agent in state ***s*** to select the action ***a*** that maximizes Q(s, a), thereby exploiting its current approximation Q.

- However, with this strategy the agent runs the risk that it will overcommit to actions that are found during early training to have high Q values, while failing to explore other actions that have even higher values.

- For this reason, Q learning uses a probabilistic approach to selecting actions. Actions with higher Q values are assigned higher probabilities, but every action is assigned a nonzero probability.

- One way to assign such probabilities is

$$P(a_i|s) = \frac{k^{\hat{Q}(s,a_i)}}{\sum_j k^{\hat{Q}(s,a_j)}}$$

- Where, **P(ai |s)** is the probability of selecting action **ai**, given that the agent is in state **s**, and **k > 0** is a constant that determines how strongly the selection favors actions with high Q values

Evaluating Hypotheses
   Motivation
It is important to evaluate the performance of learned hypotheses as precisely as possible.
- One reason is simply to understand whether to use the hypothesis.

- A second reason is that evaluating hypotheses is an integral component of many learning methods.

Two key difficulties arise while learning a hypothesis and estimating its future accuracy given only a limited set of data:

1. **Bias in the estimate**. The observed accuracy of the learned hypothesis over the training examples is often a poor estimator of its accuracy over future examples. Because the learned hypothesis was derived from these examples, they will typically provide an optimistically biased estimate of hypothesis accuracy over future examples. This is especially likely when the learner considers a very rich hypothesis space, enabling it to overfit the training examples. To obtain an unbiased estimate of future accuracy, test the hypothesis on some set of test examples chosen independently of the training examples and

the hypothesis.

2. **Variance in the estimate.** Even if the hypothesis accuracy is measured over an unbiased set of test examples independent of the training examples, the measured accuracy can still vary from the true accuracy, depending on the makeup of the particular set of test examples. The smaller the set of test examples, the greater the expected variance.

Estimating Hypothesis Accuracy

Sample Error –
The sample error of a hypothesis with respect to some sample S of instances drawn from X is the fraction of S that it misclassifies.

Definition: The sample error (***error$_S$(h)***) of hypothesis h with respect to target function f and data sample S is

$$error_S(h) \equiv \frac{1}{n} \sum_{x \in S} \delta(f(x), h(x))$$

Where n is the number of examples in S, and the quantity $\delta(f(x), h(x))$ is 1 if $f(x) \neq h(x)$, and 0 otherwise.

True Error –
The true error of a hypothesis is the probability that it will misclassify a single randomly drawn instance from the distribution **D**.

Definition: The true error (error**D** (h)) of hypothesis h with respect to target function f and distribution **D**, is the probability that h will misclassify an instance drawn at random according to **D**.

$$error_{\mathcal{D}}(h) \equiv \Pr_{x \in \mathcal{D}}[f(x) \neq h(x)]$$

Confidence Intervals for Discrete-Valued Hypotheses
Suppose we wish to estimate the true error for some discrete valued hypothesis h, based on its observed sample error over a sample S, where

- The sample S contains n examples drawn independent of one another, and independent of h, according to the probability distribution **D**

- n ≥ 30

- Hypothesis h commits r errors over these n examples (i.e., errors (h) = r/n).

Under these conditions, statistical theory allows to make the following assertions:
1. Given no other information, the most probable value of error**D** (h) is error$_S$(h)

2. With approximately **95% probability**, the true error error**D** (h) lies in the interval

$$errors(h) \pm 1.96\sqrt{\frac{errors(h)(1 - errors(h))}{n}}$$

Example:
Suppose the data sample S contains n = 40 examples and that hypothesis h commits r = 12 errors over this data.

- The *sample error* is errors(h) = r/n = 12/40 = 0.30

- Given no other information, *true error* is errorD (h) = errors(h), i.e., errorD (h) = 0.30

- With the 95% confidence interval estimate for errorD (h).

$$errors(h) \pm 1.96\sqrt{\frac{errors(h)(1 - errors(h))}{n}}$$

= 0.30 ± (1.96 * 0.07)
= 0.30 ± 0.14

3. A different constant, *ZN*, is used to calculate the **N% confidence interval**. The general expression for approximate N% confidence intervals for errorD (h) is

$$errors(h) \pm z_N\sqrt{\frac{errors(h)(1 - errors(h))}{n}}$$

Where,

N%:	50%	68%	80%	90%	95%	98%	99%
z_N:	0.67	1.00	1.28	1.64	1.96	2.33	2.58

The above equation describes how to calculate the confidence intervals, or error bars, for estimates of errorD (h) that are based on errors(h)

Example:
Suppose the data sample S contains n = 40 examples and that hypothesis h commits r = 12 errors over this data.

- The *sample error* is errors(h) = r/n = 12/40 = 0.30

- With the 68% confidence interval estimate for errorD (h).

$$errors(h) \pm 1.00\sqrt{\frac{errors(h)(1 - errors(h))}{n}}$$

= 0.30 ± (1.00 * 0.07)
= 0.30 ± 0.07

Basics of Sampling Theory
      Error Estimation and Estimating Binomial Proportions

- Collect a random sample S of n independently drawn instances from the distribution D, and then measure the sample error errors(h). Repeat this experiment many times, each time drawing a different random sample Si of size n, we would expect to observe different values for the various errorsi(h), depending on random differences in the makeup of the various Si. We say that errorsi(h), the outcome of the ith such experiment, is a ***random variable***.

- Imagine that we were to run k random experiments, measuring the random variables errors1(h), errors2(h) . . . errorssk(h) and plotted a histogram displaying the frequency with which each possible error value is observed.

- As ***k*** grows, the histogram would approach a particular probability distribution called the ***Binomial distribution*** which is shown in below figure.

Binomial distribution for $n = 40$, $p = 0.3$
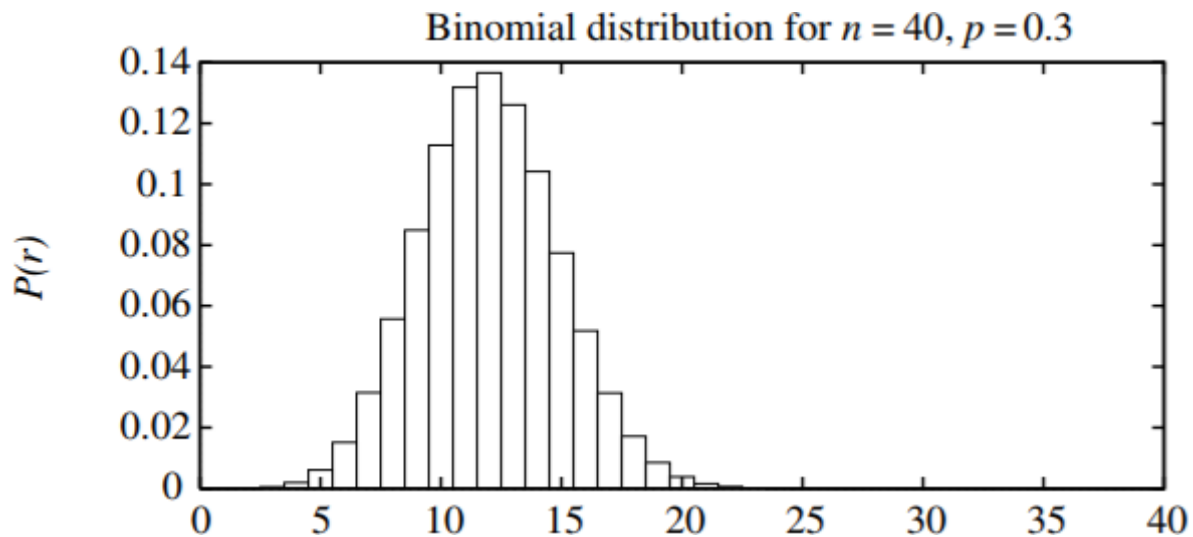
A Binomial distribution is defined by the probability function

$$P(r) = \frac{n!}{r!(n-r)!} \, p^r (1-p)^{n-r}$$

If the random variable ***X*** follows a Binomial distribution, then:

- The probability ***Pr(X = r)*** that ***X*** will take on the value ***r*** is given by ***P(r)***

- Expected, or mean value of X, $E[X]$, is

$$E[X] \equiv \sum_{i=0}^{n} i P(i) = np$$

- Variance of X is

$$Var(X) \equiv E[(X - E[X])^2] = np(1 - p)$$

- Standard deviation of X, σ_X, is

$$\sigma_X \equiv \sqrt{E[(X - E[X])^2]} = \sqrt{np(1 - p)}$$

The Binomial Distribution

Consider the following problem for better understanding of Binomial Distribution

- Given a worn and bent coin and estimate the probability that the coin will turn up heads when tossed.

- Unknown probability of heads *p*. Toss the coin *n* times and record the number of times *r* that it turns up heads.

Estimate of *p = r / n*

- If the experiment were *rerun*, generating a new set of *n* coin tosses, we might expect the number of heads *r* to vary somewhat from the value measured in the first experiment, yielding a somewhat different estimate for *p*.

- The Binomial distribution describes for each possible value of *r* (i.e., from 0 to n), the probability of observing exactly *r* heads given a sample of *n* independent tosses of a coin whose true probability of heads is *p*.

The general setting to which the Binomial distribution applies is:

1. There is a base experiment (e.g., toss of the coin) whose outcome can be described by a random variable 'Y'. The random variable Y can take on two possible values (e.g., Y = 1 if heads, Y = 0 if tails).

2. The probability that Y = 1 on any single trial of the base experiment is given by some constant p, independent of the outcome of any other experiment. The probability that Y = 0 is therefore (1 - p). Typically, p is not known in advance, and the problem is to estimate it.

3. A series of n independent trials of the underlying experiment is performed (e.g., n independent coin tosses), producing the sequence of independent, identically distributed random variables Y1, Y2, . . . , Yn. Let R denote the number of trials for which Yi = 1 in this series of n experiments

$$R \equiv \sum_{i=1}^{n} Y_i$$

4. The probability that the random variable R will take on a specific value r (e.g., the probability of observing exactly r heads) is given by the Binomial distribution

$$\Pr(R = r) = \frac{n!}{r!(n-r)!} \, p^r (1-p)^{n-r} \qquad \text{equ (1)}$$

Mean, Variance and Standard Deviation
The Mean (expected value) is the average of the values taken on by repeatedly sampling the random variable

Definition: Consider a random variable Y that takes on the possible values y1, . . . yn. The expected value (Mean) of Y, E[Y], is

$$E[Y] \equiv \sum_{i=1}^{n} y_i \, \Pr(Y = y_i)$$

The Variance captures how far the random variable is expected to vary from its mean value.
Definition: The variance of a random variable Y, Var[Y], is

$$Var[Y] \equiv E[(Y - E[Y])^2]$$

The variance describes the expected squared error in using a single observation of Y to estimate its mean E[Y].

The square root of the variance is called the standard deviation of Y, denoted σy

Definition: The standard deviation of a random variable Y, σy, is

$$\sigma_Y \equiv \sqrt{E[(Y - E[Y])^2]}$$

In case the **random variable Y is governed by a Binomial distribution**, then the Mean, Variance and standard deviation are given by

$$E[Y] = np$$
$$Var[Y] = np(1 - p)$$
$$\sigma_Y = \sqrt{np(1 - p)}$$

Estimators, Bias, and Variance
Let us describe errors(h) and errorD(h) using the terms in Equation (1) defining the Binomial distribution. We then have

$$error_S(h) = \frac{r}{n}$$
$$error_D(h) = p$$

Where,
- n is the number of instances in the sample S,

- r is the number of instances from S misclassified by h

- p is the probability of misclassifying a single instance drawn from *D*

- Estimator:

errors(h) an **estimator** for the true error errorD(h): An estimator is any random variable used to estimate some parameter of the underlying population from which the sample is drawn

- Estimation bias: is the difference between the expected value of the estimator and the true value of the parameter.

Definition: The estimation bias of an estimator Y for an arbitrary parameter p is

$$E[Y] - p$$

UNIT V
Genetic Algorithms

Motivation, Genetic Algorithms: Representing Hypotheses, Genetic Operator, Fitness Function and Selection, An Illustrative Example, Hypothesis Space Search, Genetic Programming, Models

of Evolution and Learning: Lamarkian Evolution, Baldwin Effect, Parallelizing Genetic Algorithms.

Motivation

   Genetic algorithms (GAS) provide a learning method motivated by an analogy to biological evolution. Rather than search from general-to-specific hypotheses, or from simple-to-complex, GAS generate successor hypotheses by repeatedly mutating and recombining parts of the best currently known hypotheses. At each step, a collection of hypotheses called the current population is updated by replacing some fraction of the population by offspring of the most fit current hypotheses. The process forms a generate-and-test beam-search of hypotheses, in which variants of the best current hypotheses are most likely to be considered next. The popularity of GAS is motivated by a number of factors including:

- Evolution is known to be a successful, robust method for adaptation within biological systems.
- GAS can search spaces of hypotheses containing complex interacting parts, where the impact of each part on overall hypothesis fitness may be difficult to model.
- Genetic algorithms are easily parallelized and can take advantage of the decreasing costs of powerful computer hardware.

9.2 Genetic Algorithms

   The problem addressed by GAS is to search a space of candidate hypotheses to identify the best hypothesis. In GAS the "best hypothesis" is defined as the one that optimizes a predefined numerical measure for the problem at hand, called b the hypothesis fitness. For example, if the learning task is the problem of approximating an unknown function given training examples of its input and output, then fitness could be defined as the accuracy of the hypothesis over this training data. If the task is to learn a strategy for playing chess, fitness could be defined as the number of games won by the individual when playing against other individuals in the current population.

Although different implementations of genetic algorithms vary in their details, they typically share the following structure: The algorithm operates by iteratively updating a pool of hypotheses, called the population. On each iteration, all members of the population are evaluated according to the fitness function. A new population is then generated by probabilistically selecting the fit individuals from the current population. Some of these selected individuals are carried forward into the next generation population intact. Others are used as the basis for creating new offspring individuals by applying genetic operations such as crossover and mutation.

GA($Fitness$, $Fitness\_threshold$, p, r, m)

    $Fitness$: A function that assigns an evaluation score, given a hypothesis.

    $Fitness\_threshold$: A threshold specifying the termination criterion.

    p: The number of hypotheses to be included in the population.

    r: The fraction of the population to be replaced by Crossover at each step.

    m: The mutation rate.

- *Initialize population:* $P \leftarrow$ Generate p hypotheses at random
- *Evaluate:* For each h in P, compute $Fitness(h)$
- *While* $[\max_h Fitness(h)] < Fitness\_threshold$ do

Create a new generation, P_S:

1. *Select:* Probabilistically select $(1-r)p$ members of P to add to P_S. The probability $\Pr(h_i)$ of selecting hypothesis h_i from P is given by

$$\Pr(h_i) = \frac{Fitness(h_i)}{\sum_{j=1}^{p} Fitness(h_j)}$$

2. *Crossover:* Probabilistically select $\frac{r \cdot p}{2}$ pairs of hypotheses from P, according to $\Pr(h_i)$ given above. For each pair, $\langle h_1, h_2 \rangle$, produce two offspring by applying the Crossover operator. Add all offspring to P_s.

3. *Mutate:* Choose m percent of the members of P_s with uniform probability. For each, invert one randomly selected bit in its representation.

4. *Update:* $P \leftarrow P_s$.

5. *Evaluate:* for each h in P, compute $Fitness(h)$

- Return the hypothesis from P that has the highest fitness.

The inputs to this algorithm include the fitness function for ranking candidate hypotheses, a threshold defining an acceptable level of fitness for terminating the algorithm, the size of the population to be maintained, and parameters that determine how successor populations are to be generated: the fraction of the population to be replaced at each generation and the mutation rate. Notice in this algorithm each iteration through the main loop produces a new generation of hypotheses based on the current population. First, a certain number of hypotheses from the current population are selected for inclusion in the next generation. These are selected **probabilistically,** where the probability of selecting hypothesis h_i is given by

$$\Pr(h_i) = \frac{Fitness(h_i)}{\sum_{j=1}^{p} Fitness(h_j)}$$

Thus, the probability that a hypothesis will be selected is proportional to its own fitness and is inversely proportional to the fitness of the other competing hypotheses in the current population.

Once these members of the current generation have been selected for inclusion in the next generation population, additional members are generated using a crossover operation. Crossover, defined in detail in the next section, takes two parent hypotheses from the current generation and creates two offspring hypotheses

by recombining portions of both parents. The parent hypotheses are chosen probabilistically from the current population, again using the probability function given by Equation (9.1). After new members have been created by this crossover operation, the new generation population now contains the

desired number of members. At this point, a certain fraction m of these members are chosen at random, and random mutations all performed to alter these members.

This GA algorithm thus performs a randomized, parallel beam search for hypotheses that perform well according to the fitness function. In the following subsections, we describe in more detail the representation of hypotheses and genetic operators used in this algorithm.

Representing Hypotheses

Hypotheses in GAS are often represented by bit strings, so that they can be easily manipulated by genetic operators such as mutation and crossover. The hypotheses represented by these bit strings can be quite complex. For example, sets of if-then rules can easily be represented in this way, by choosing an encoding of rules that allocates specific substrings for each rule precondition and postcondition.

To see how if-then rules can be encoded by bit strings, .first consider how we might use a bit string to describe a constraint on the value of a single attribute. To pick an example, consider the attribute *Outlook,* which can take on any of the three values *Sunny, Overcast,* or *Rain.* One obvious way to represent a constraint on *Outlook* is to use a bit string of length three, in which each bit position corresponds to one of its three possible values. Placing a 1 in some position indicates that the attribute is allowed to take on the corresponding value. For example, the string 010 represents the constraint that *Outlook* must take on the second of these values, or *Outlook = Overcast.* Similarly, the string 011 represents the more general constraint that allows two possible values, or *(Outlook = Overcast v Rain).* Note 11 1 represents the most general possible constraint, indicating that we don't care which of its possible values the attribute takes on.

Given this method for representing constraints on a single attribute, conjunctions of constraints on multiple attributes can easily be represented by concatenating the corresponding bit strings. For example, consider a second attribute, *Wind,* that can take on the value *Strong* or *Weak. A* rule precondition such as

(Outlook = Overcast ^Rain) A (Wind = Strong)

can then be represented by the following bit string of length five:

Outlook   Wind
01 1   10

Rule postconditions (such as *PlayTennis = yes)* can be represented in a similar fashion. Thus, an entire rule can be described by concatenating the bit strings describing the rule preconditions, together with the bit string describing the rule postcondition. For example, the rule

IF Wind = Strong THEN *PlayTennis = yes*

would be represented by the string

Outlook   Wind   PlayTennis
111   10   10

where the first three bits describe the "don't care" constraint on *Outlook,* the next two bits describe the constraint on *Wind,* and the final two bits describe the rule postcondition (here we assume *PlayTennis*

102

can take on the values *Yes* or *No).* Note the bit string representing the rule contains a substring for each attribute in the hypothesis space, even if that attribute is not constrained by the rule preconditions. This yields a fixed length bit-string representation for rules, in which substrings at specific locations describe constraints on specific attributes. Given this representation for single rules, we can represent sets of rules by similarly concatenating the bit string representations of the individual rules.
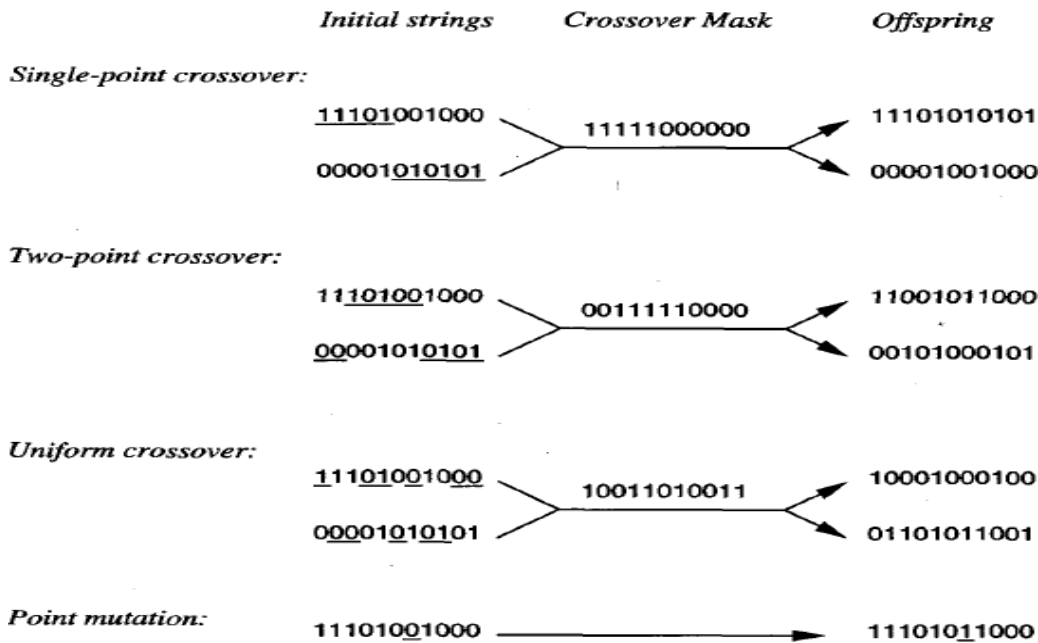
In designing a bit string encoding for some hypothesis space, it is useful to arrange for every syntactically legal bit string to represent a well-defined hypothesis. To illustrate, note in the rule encoding in the above paragraph the bit string 11 1 10 11 represents a rule whose postcondition does not constrain the target attribute *PlayTennis.* If we wish to avoid considering this hypothesis, we may employ a different encoding (e.g., allocate just one bit to the *PlayTennis* postcondition to indicate whether the value is *Yes* or *No),* alter the genetic operators so that they explicitly avoid constructing such bit strings, or simply assign a very low fitness to such bit strings.

In some GAS, hypotheses are represented by symbolic descriptions rather than bit strings.

Genetic Operators

The generation of successors in a GA is determined by a set of operators that recombine and mutate selected members of the current population. These operators correspond to idealized versions of the genetic operations found in biological evolution. The two most common operators are *crossover* and *mutation.*

The *crossover operator* produces two new offspring from two parent strings, by copying selected bits from each parent. The bit at position i in each offspring is copied from the bit at position i in one of the two parents. The choice of which parent contributes the bit for position i is determined by an additional string called the *crossover mask.* To illustrate, consider the *single-point crossover* operator at the top of Table Consider the topmost of the two offspring in this case. This offspring takes its first five bits from the first parent and its remaining six bits from the second parent, because the crossover mask 11 11 1000000 specifies these choices for each of the bit positions. The second offspring uses the same crossover mask, but switches the roles of the two parents. Therefore, it contains the bits that were not used by the first offspring. In single-point crossover, the crossover mask is always constructed so that it begins with a string containing *n* contiguous Is, followed by the necessary number of 0s to complete the string. This results in offspring in which the first n bits are contributed by one parent and the remaining bits by the second parent. Each time the single-point crossover operator is applied the crossover point *n* is chosen at random, and the crossover mask is then created and applied.

Initial strings | Crossover Mask | Offspring

Single-point crossover:

11101001000
00001010101
11111000000
11101010101
00001001000

Two-point crossover:

11101001000
00001010101
00111110000
11001011000
00101000101

Uniform crossover:

11101001000
00001010101
10011010011
10001000100
01101011001

Point mutation:

11101001000 ⟶ 11101011000

In **two-point crossover,** offspring are created by substituting intermediate segments of one parent into the middle of the second parent string. Put another way, the crossover mask is a string beginning with **n_o** zeros, followed by a contiguous string of **n_l** ones, followed by the necessary number of zeros to complete the string. Each time the two-point crossover operator is applied, a mask is generated by randomly choosing the integers **n_o** and **n_l.**

Fitness Function and Selection

The fitness function defines the criterion for ranking potential hypotheses and for probabilistically selecting them for inclusion in the next generation population. If the task is to learn classification rules, then the fitness function typically has a component that scores the classification accuracy of the rule over a set of provided training examples. Often other criteria may be included as well, such as the complexity or generality of the rule. More generally, when the bit-string hypothesis is interpreted as a complex procedure (e.g., when the bit string represents a collection of if-then rules that will be chained together to control a robotic device), the fitness function may measure the overall performance of the resulting procedure rather than performance of individual rules.

In our prototypical GA shown in above Table , the probability that a hypothesis will be selected is given by the ratio of its fitness to the fitness of other members of the current population as seen in Equation above . This method is sometimes called **fitness proportionate selection,** or roulette wheel selection. Other methods for using fitness to select hypotheses have also been proposed. For example, in **tournament selection,** two hypotheses are first chosen at random from the current population. With some predefined probability p the more fit of these two is then selected, and with probability (1 - p) the less fit hypothesis is selected. Tournament selection often yields a more diverse population than fitness proportionate selection. In another method called **rank selection,** the hypotheses in the current population are first sorted by fitness. The probability that a hypothesis will be selected is then proportional to its rank in this sorted list, rather than its fitness.

5.3. An Illustrative Example

A genetic algorithm can be viewed as a general optimization method that searches a large space of candidate objects seeking one that performs best according to the fitness function. Although not guaranteed to find an optimal object, GAS often succeed in finding an object with high fitness. GAS have been applied to a number of optimization problems outside machine learning, including problems such as circuit layout and job-shop scheduling. Within machine learning, they have been applied both to function-approximation problems and to tasks such as choosing the network topology for artificial neural network learning systems.

To illustrate the use of GAS for concept learning, we briefly summarize the GABIL system described by DeJong et al. (1993). GABIL uses a GA to learn boolean concepts represented by a disjunctive set of propositional rules. In experiments over several concept learning problems, GABIL was found to be roughly comparable in generalization accuracy to other learning algorithms such as the decision tree learning algorithm C4.5 and the rule learning system AQ14. The learning tasks in this study included both artificial learning tasks designed to explore the systems' generalization accuracy and the real world problem of breast cancer diagnosis.

The specific instantiation of the GA algorithm in GABIL can be summarized as follows:

Representation. Each hypothesis in GABIL corresponds to a disjunctive set of propositional rules, encoded as described in Section 9.2.1. In particular, the hypothesis space of rule preconditions consists of a conjunction of constraints on a fixed set of attributes, as described in that earlier section. To represent a set of rules, the bit-string representations of individual rules are concatenated. To illustrate, consider a hypothesis space in which rule preconditions are conjunctions of constraints over two Boolean attributes, a_1 and a_2. The rule postcondition is described by a single bit that indicates the predicted value of the target attribute c. Thus, the hypothesis consisting of the two rules

$$\text{IF } a_l = T \wedge a_2 = F \text{ THEN } c = T; \text{ IF } a_2 = T \text{ THEN } c = F$$

would be represented by the string

```
a₁  a₂  c        a₁  a₂  c
10  01  1        11  01 0
```

Note the length of the bit string grows with the number of rules in the hypothesis. This variable bit-string length requires a slight modification to the crossover operator, as described below.

Genetic operators. GABIL uses the standard mutation operator of above Table in which a single bit is chosen at random and replaced by its complement. The crossover operator that it uses is a fairly standard extension to the two-point crossover operator described in Table 9.2. In particular, to accommodate the variable-length bit strings that encode rule sets, and to constrain the system so that crossover occurs only between like sections of the bit strings that encode rules, the following approach is taken. To perform a crossover operation on two parents, two crossover points are first chosen at random in the first parent string. Let dl (dz) denote the distance from the leftmost (rightmost) of these two crossover points to the rule boundary immediately to its left. The crossover points in the second parent are now randomly chosen, subject to the constraint that they must have the same d_l and d_2 value. For example, if the two parent strings are

$$
\begin{array}{cccccc}
 & a_1 & a_2 & c & a_1 & a_2 & c \\
h_1: & 10 & 01 & 1 & 11 & 10 & 0
\end{array}
$$

and

$$
\begin{array}{cccccc}
 & a_1 & a_2 & c & a_1 & a_2 & c \\
h_2: & 01 & 11 & 0 & 10 & 01 & 0
\end{array}
$$

and the crossover points chosen for the first parent are the points following bit positions 1 and 8,

$$
\begin{array}{cccccc}
 & a_1 & a_2 & c & a_1 & a_2 & c \\
h_1: & 1[0 & 01 & 1 & 11 & 1]0 & 0
\end{array}
$$

where "[" and "1" indicate crossover points, then dl = 1 and dz = 3. Hence the allowed pairs of crossover points for the second parent include the pairs of bit positions (1,3), (1,8), and **(6,8).** If the pair (1,3) happens to be chosen,

$$
\begin{array}{cccccc}
 & a_1 & a_2 & c & a_1 & a_2 & c \\
h_2: & 0[1 & 1]1 & 0 & 10 & 01 & 0
\end{array}
$$

then the two resulting offspring will be

$$
\begin{array}{cccc}
 & a_1 & a_2 & c \\
h_3: & 11 & 10 & 0
\end{array}
$$

and

$$
\begin{array}{ccccccccc}
 & a_1 & a_2 & c & a_1 & a_2 & c & a_1 & a_2 & c \\
h_4: & 00 & 01 & 1 & 11 & 11 & 0 & 10 & 01 & 0
\end{array}
$$

As this example illustrates, this crossover operation enables offspring to contain a different number of rules than their parents, while assuring that all bit strings generated in this fashion represent well-defined rule sets.

Fitness function. The fitness of each hypothesized rule set is based on its classification accuracy over the training data. In particular, the function used to measure fitness is

$$ Fitness(h) = (correct(h))^2 $$

where **correct (h)** is the percent of all training examples correctly classified by hypothesis **h.**

In experiments comparing the behavior of GABIL to decision tree learning algorithms such as C4.5 and ID5R, and to the rule learning algorithm AQ14report roughly comparable performance among these systems, tested on a variety of learning problems. For example, over a set of 12 synthetic problems, GABIL achieved an average generalization accuracy of 92.1 %, whereas the performance of the other systems ranged from 91.2 % to 96.6 %.

Extensions

106

In one set of experiments they explored the addition of two new genetic operators that were motivated by the generalization operators common in many symbolic learning methods. The first of these operators, **AddAlternative,** generalizes the constraint on a specific attribute by changing a 0 to a 1 in the substring corresponding to the attribute. For example, if the constraint on an attribute is represented by the string 10010, this operator might change it to 101 10. This operator was applied with probability .O1 to selected members of the population on each generation. The second operator, **Dropcondition** performs a more drastic generalization step, by replacing all bits for a particular attribute by a 1. This operator corresponds to generalizing the rule by completely dropping the constraint on the attribute, and was applied on each generation with probability .60. The authors report this revised system achieved an average performance of 95.2% over the above set of synthetic learning tasks, compared to 92.1% for the basic GA algorithm.

In the above experiment, the two new operators were applied with the same probability to each hypothesis in the population on each generation. In a second experiment, the bit-string representation for hypotheses was extended to include two bits that determine which of these operators may be applied to the hypothesis. In this extended representation, the bit string for a typical rule set hypothesis would be

$$
\begin{array}{cccccccc}
a_1 & a_2 & c & a_1 & a_2 & c & AA & DC \\
01 & 11 & 0 & 10 & 01 & 0 & 1 & 0
\end{array}
$$

where the final two bits indicate in this case that the **AddAlternative** operator may be applied to this bit string, but that the **Dropcondition** operator may not. These two new bits define part of the search strategy used by the GA and are themselves altered and evolved using the same crossover and mutation operators that operate on other bits in the string. While the authors report mixed results with this approach (i.e., improved performance on some problems, decreased performance on others), it provides an interesting illustration of how GAS might in principle be used to evolve their own hypothesis search methods.

Hypothesis Space Search
As illustrated above, GAS employ a randomized beam search method to seek a maximally fit hypothesis. This search is quite different from that of other learning methods we have considered in this book. To contrast the hypothesis space search of GAS with that of neural network BACKPROPAGATION, for example, the radiant descent search in BACKPROPAGATION moves smoothly from one hypothesis to a new hypothesis that is very similar. In contrast, the GA search can move much more abruptly, replacing a parent hypothesis by an offspring that may be radically different from the parent. Note the GA search is therefore less likely to fall into the same kind of local minima that can plague gradient descent methods.
One practical difficulty in some GA applications is the problem of **crowding.** Crowding is a phenomenon in which some individual that is more highly fit than others in the population quickly reproduces, so that copies of this individual and very similar individuals take over a large fraction of the population. The negative impact of crowding is that it reduces the diversity of the population, thereby slowing further progress by the GA. Several strategies have been explored for reducing crowding. One approach is to alter the selection function, using criteria such as tournament selection or rank selection in place of fitness proportionate roulette wheel selection. A related strategy is "fitness sharing," in which the measured fitness of an individual is reduced by the presence of other, similar individuals in the population. A third approach is to restrict the kinds of individuals allowed to recombine to form offspring. For example, by allowing only the most similar individuals to recombine, we can encourage

the formation of clusters of similar individuals, or multiple "subspecies" within the population. A related approach is to spatially distribute individuals and allow only nearby individuals to recombine. Many of these techniques are inspired by the analogy to biological evolution.

Population Evolution and the Schema Theorem

It is interesting to ask whether one can mathematically characterize the evolution over time of the population within a GA. The schema theorem provides one such characterization. It is based on the concept of **schemas,** or patterns that describe sets of bit strings. To be precise, a schema is any string composed of 0s, 1s, and *'s. Each schema represents the set of bit strings containing the indicated 0s and 1s, with each "*" interpreted as a "don't care." For example, the schema 0*10 represents the set of bit strings that includes exactly 0010 and 01 10.

An individual bit string can be viewed as a representative of each of the different schemas that it matches. For example, the bit string 0010 can be thought of as a representative of 2^4 distinct schemas including 00**, 0* 10, ****, etc. Similarly, a population of bit strings can be viewed in terms of the set of schemas that it represents and the number of individuals associated with each of these schema.

The schema theorem characterizes the evolution of the population within a GA in terms of the number of instances representing each schema. Let $m(s, t)$ denote the number of instances of schema **s** in the population at time t (i.e., during the tth generation). The schema theorem describes the expected value of $m(s, t+1)$ in terms of $m(s, t)$ and other properties of the schema, population, and GA algorithm parameters.

The evolution of the population in the GA depends on the selection step, the recombination step, and the mutation step. Let us start by considering just the effect of the selection step. Let f (h) denote the fitness of the individual bit string h and $\bar{f}(t)$ denote the average fitness of all individuals in the population at time t. Let *n* be the total number of individuals in the population. Let $h \varepsilon s \cap p_t$, indicate that the individual h is both a representative of schema **s** and a member of the population at time t.

Finally, let $\hat{u}(s, t)$ denote the average fitness of instances of schema **s** in the population at time t.

We are interested in calculating the expected value of $m(s, t+1)$, which we denote $E[m(s, t+1)]$. We can calculate $E[m(s, t+1)]$ using the probability distribution for selection given in Equation, which can be restated using our current terminology as follows

$$\Pr(h) = \frac{f(h)}{\sum_{i=1}^{n} f(h_i)}$$

$$= \frac{f(h)}{n\bar{f}(t)}$$

Now if we select one member for the new population according to this probability distribution, then the probability that we will select a representative of schema **s** is

$$\Pr(h \in s) = \sum_{h \in s \cap p_t} \frac{f(h)}{n\bar{f}(t)}$$

$$= \frac{\hat{u}(s, t)}{n\bar{f}(t)} m(s, t)$$

The second step above follows from the fact that by definition,

$$\hat{u}(s, t) = \frac{\sum_{h \in s \cap p_t} f(h)}{m(s, t)}$$

Equation gives the probability that a single hypothesis selected by the **GA** will be an instance of schema **s.** Therefore, the expected number of instances of **s** resulting from the *n* independent selection steps that create the entire new generation is just *n* times this probability.

$$E[m(s, t+1)] = \frac{\hat{u}(s, t)}{\bar{f}(t)} m(s, t)$$

Equation states that the expected number of instances of schema **s** at generation t+1 is proportional to the average fitness $\hat{u}(s, t)$ of instances of this schema at time **t ,** and inversely proportional to the average fitness $\bar{f}(t)$ of all members of the population at time t. Thus, we can expect schemas with above average fitness to be represented with increasing frequency on successive generations. If we view the **GA** as performing a virtual parallel search through the space of possible schemas at the same time it performs its explicit parallel search through the space of individuals, then Equation indicates that more fit schemas will grow in influence over time.

While the above analysis considered only the selection step of the **GA,** the crossover and mutation steps must be considered as well. The schema theorem considers only the possible negative influence of these genetic operators (e.g., random mutation may decrease the number of representatives of **s,** independent of $\hat{u}(s, t)$ and considers only the case of single-point crossover. The full schema theorem thus provides a lower bound on the expected frequency of schema **s,** as follows:

$$E[m(s, t+1)] \geq \frac{\hat{u}(s, t)}{\bar{f}(t)} m(s, t) \left(1 - p_c \frac{d(s)}{l-1}\right) (1 - p_m)^{o(s)}$$

Here, p_c is the probability that the single-point crossover operator will be applied
to an arbitrary individual, and p_m, is the probability that an arbitrary bit of an arbitrary individual will be mutated by the mutation operator. **o(s)** is the number of *defined bits* in schema **s,** where 0 and 1 are defined bits, but * is not. **d(s)** is the distance between the leftmost and rightmost defined bits in **s.** Finally, *l* is the length of the individual bit strings in the population. Notice the leftmost term in Equation is identical to the term from Equation and describes the effect of the selection step. The middle term describes the effect of the single-point crossover operator-in particular, it describes the probability that an arbitrary individual representing **s** will still represent **s** following application of this crossover operator. The rightmost term describes the probability that an arbitrary individual representing schema **s** will still represent schema **s** following application of the mutation operator. Note that the effects of single-point crossover and mutation increase with the number of defined bits **o(s)** in the schema and with the distance **d(s)** between the defined bits. Thus, the schema theorem can be roughly interpreted as stating that more fit schemas will tend to grow in influence, especially schemas containing a small number of defined bits (i.e., containing a large number of *'s), and especially when these defined bits

are near one another within the bit string. The schema theorem is perhaps the most widely cited characterization of population evolution within a GA. One way in which it is incomplete is that it fails to consider the (presumably) positive effects of crossover and mutation. Numerous more recent theoretical analyses have been proposed, including analyses based on Markov chain models and on statistical mechanics models.

GENETIC PROGRAMMING

Genetic programming (GP) is a form of evolutionary computation in which the individuals in the evolving population are computer programs rather than bit strings. The basic genetic programming approach and presents a broad range of simple programs that can be successfully learned by GP.

Representing Programs

Programs manipulated by a GP are typically represented by trees corresponding to the parse tree of the program. Each function call is represented by a node in the tree, and the arguments to the function are given by its descendant nodes. For example, below Figure illustrates this tree representation for the function $\sin(x) + \sqrt{\left(x^2 + y\right)}$. To apply genetic programming to a particular domain, the user must define the primitive functions to be considered (e.g., sin, cos, $\sqrt{}$, +, -, exponential~), as well as the terminals (e.g., x, *y*, constants such as 2). The genetic programming algorithm then uses an evolutionary search to explore the vast space of programs that can be described using these primitives. As in a genetic algorithm, the prototypical genetic programming algorithm maintains a population of individuals (in this case, program trees). On each iteration, it produces a new generation of individuals using selection, crossover, and mutation. The fitness of a given individual program in the population is typically determined by executing the program on a set of training data. Crossover operations are performed by replacing a randomly chosen subtree of one parent program by a subtree from the other parent program.

110
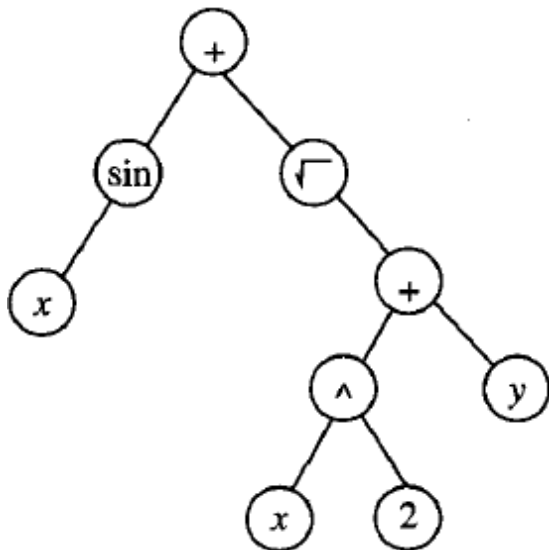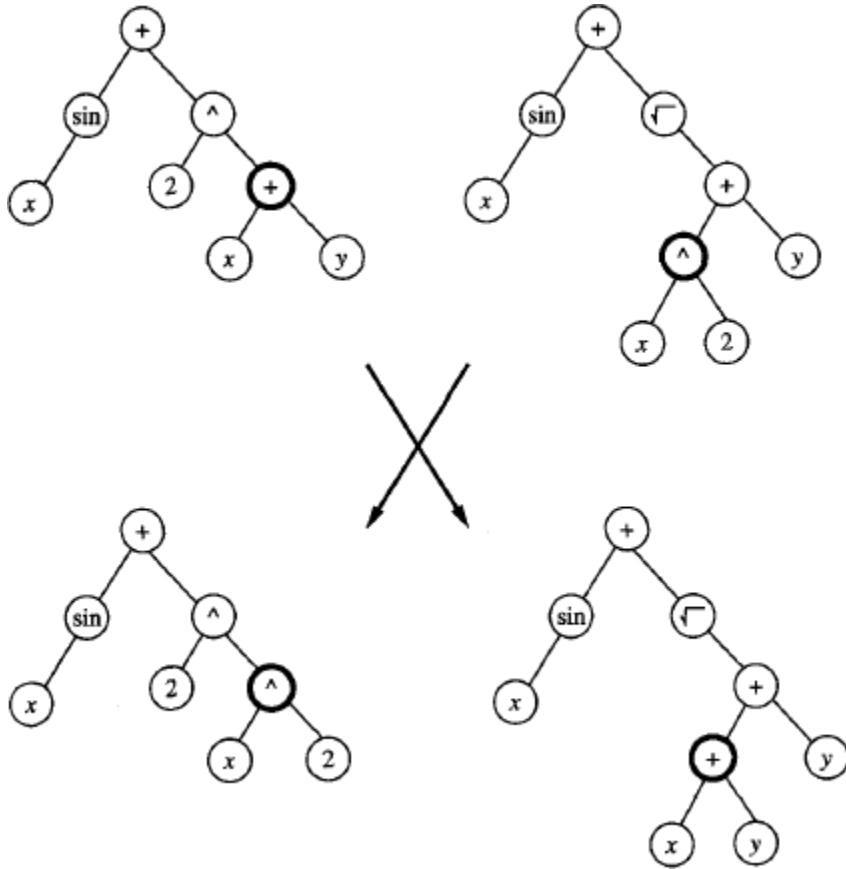
Above Figure illustrates a typical crossover operation. It describes a set of experiments applying a GP to a number of applications. In his experiments, 10% of the current population, selected probabilistically according to fitness, is retained unchanged in the next generation. The remainder of the new generation is created by applying crossover to pairs of programs from the current generation, again selected probabilistically according to their fitness. The mutation operator was not used in this particular set of experiments.

Illustrative Example

One illustrative example presented by Koza (1992) involves learning an algorithm for stacking the blocks shown in below Figure The task is to develop a general algorithm for stacking the blocks into a single stack that spells the word "universal," independent of the initial configuration of blocks in the world. The actions available for manipulating blocks allow moving only a single block at a time. In particular, the top block on the stack can be moved to the table surface, or a block on the table surface can be moved to the top of the stack.

FIGURE 9.3
A block-stacking problem. The task for GP is to discover a program that can transform an arbitrary initial configuration of blocks into a stack that spells the word "universal." A set of 166 such initial configurations was provided to evaluate fitness of candidate programs (after Koza 1992).

As in most GP applications, the choice of problem representation has a significant impact on the ease of solving the problem. In Koza's formulation, the primitive functions used to compose programs for this task include the following three terminal arguments:

- CS (current stack), which refers to the name of the top block on the stack, or F if there is no current stack.
- TB (top correct block), which refers to the name of the topmost block on the stack, such that it and those blocks beneath it are in the correct order.
- NN (next necessary), which refers to the name of the next block needed above TB in the stack, in order to spell the word "universal," or F if no more blocks are needed.

As can be seen, this particular choice of terminal arguments provides a natural representation for describing programs for manipulating blocks for this task. Imagine, in contrast, the relative difficulty of the task if we were to instead define the terminal arguments to be the x and y coordinates of each block.

In addition to these terminal arguments, the program language in this application included the following primitive functions:

- (MS x) (move to stack), if block x is on the table, this operator moves x to the top of the stack and returns the value T. Otherwise, it does nothing and returns the value F.
- (MT x) (move to table), if block *x* is somewhere in the stack, this moves the block at the top of the stack to the table and returns the value T. Otherwise, it returns the value F.
- (EQ x y) (equal), which returns T if x equals *y,* and returns F otherwise.
- (NOT x), which returns T if x = F, and returns F if x = T.
- (DU x y) (do until), which executes the expression *x* repeatedly until expressiony returns the value T.

To allow the system to evaluate the fitness of any given program, Koza provided a set of 166 training example problems representing a broad variety of initial block configurations, including problems of differing degrees of difficulty. The fitness of any given program was taken to be the number of these examples solved by the algorithm. The population was initialized to a set of 300 random programs. After 10 generations, the system discovered the following program, which solves all 166 problems.

(EQ (DU (MT CS)(NOT CS)) (DU (MS NN)(NOT NN)))

Notice this program contains a sequence of two DU, or "Do Until" statements. The first repeatedly

moves the current top of the stack onto the table, until the stack becomes empty. The second "Do Until" statement then repeatedly moves the next necessary block from the table onto the stack. The role played by the top level EQ expression here is to provide a syntactically legal way to sequence these two "Do Until" loops.

Somewhat surprisingly, after only a few generations, this GP was able to discover a program that solves all 166 training problems. Of course the ability of the system to accomplish this depends strongly on the primitive arguments and functions provided, and on the set of training example cases used to evaluate fitness.

Remarks on Genetic Programming

As illustrated in the above example, genetic programming extends genetic algorithms to the evolution of complete computer programs. Despite the huge size of the hypothesis space it must search, genetic programming has been demonstrated to produce intriguing results in a number of applications. A comparison of GP to other methods for searching through the space of computer programs, such as hillclimbing and simulated annealing, is given by O'Reilly and Oppacher (1994).

While the above example of GP search is fairly simple, Koza et al. (1996) summarize the use of a GP in several more complex tasks such as designing electronic filter circuits and classifying segments of protein molecules. The filter circuit design problem provides an example of a considerably more complex problem. Here, programs are evolved that transform a simple fixed seed circuit into a final circuit design. The primitive functions used by the GP to construct its programs are functions that edit the seed circuit by inserting or deleting circuit components and wiring connections. The fitness of each program is calculated by simulating the circuit it outputs (using the SPICE circuit simulator) to determine how closely this circuit meets the design specifications for the desired filter. More precisely, the fitness score is the sum of the magnitudes of errors between the desired and actual circuit output at 101 different input frequencies. In this case, a population of size 640,000 was maintained, with selection producing 10% of the successor population, crossover producing 89%, and mutation producing 1%. The system was executed on a 64-node parallel processor. Within the first randomly generated population, the circuits produced were so unreasonable that the SPICE simulator could not even simulate the behavior of 98% of the circuits. The percentage of unsimulatable circuits dropped to 84.9% following the first generation, to 75.0% following the second generation, and to an average of 9.6% over succeeding generations. The fitness score of the best circuit in the initial population was 159, compared to a score of 39 after 20 generations and a score of 0.8 after 137 generations. The best circuit, produced after 137 generations, exhibited performance very similar to the desired behavior.

In most cases, the performance of genetic programming depends crucially on the choice of representation and on the choice of fitness function. For this reason, **an** active area of current research is aimed at the automatic discovery and incorporation of subroutines that improve on the original set of primitive functions, thereby allowing the system to dynamically alter the primitives from which it constructs individuals. See, for example, Koza (1994).

Models of Evolution and Learning

In many natural systems, individual organisms learn to adapt significantly during their lifetime. At the same time, biological and social processes allow their species to adapt over a time frame of many generations. One interesting question regarding evolutionary systems is "What is the relationship between learning during the lifetime of a single individual, and the longer time frame species-level learning afforded by evolution?'

Lamarckian Evolution

Larnarck was a scientist who, in the late nineteenth century, proposed that evolution over many generations was directly influenced by the experiences of individual organisms during their lifetime. In

particular, he proposed that experiences of a single organism directly affected the genetic makeup of their offspring: If **an** individual learned during its lifetime to avoid some toxic food, it could pass this trait on genetically to its offspring, which therefore would not need to learn the trait. This is an attractive conjecture, because it would presumably allow for more efficient evolutionary progress than a generate-and-test process (like that of GAS and GPs) that ignores the experience gained during an individual's lifetime. Despite the attractiveness of this theory, current scientific evidence overwhelmingly contradicts Lamarck's model. The currently accepted view is that the genetic makeup of an individual is, in fact, unaffected by the lifetime experience of one's biological parents. Despite this apparent biological fact, recent computer studies have shown that Lamarckian processes can sometimes improve the effectiveness of computerized genetic algorithms (see Grefenstette 1991; Ackley and Littman 1994; and Hart and Belew 1995).

Baldwin Effect

Although Lamarckian evolution is not an accepted model of biological evolution, other mechanisms have been suggested by which individual learning can alter the course of evolution. One such mechanism is called the Baldwin effect, after J. M. Baldwin (1896), who first suggested the idea. The Baldwin effect is based on the following observations:

- If a species is evolving in a changing environment, there will be evolutionary pressure to favor individuals with the capability to learn during their lifetime. For example, if a new predator appears in the environment, then individuals capable of learning to avoid the predator will be more successful than individuals who cannot learn. In effect, the ability to learn allows an individual to perform a small local search during its lifetime to maximize its fitness. In contrast, nonlearning individuals whose fitness is fully determined by their genetic makeup will operate at a relative disadvantage.

- Those individuals who are able to learn many traits will rely less strongly on their genetic code to "hard-wire" traits. As a result, these individuals can support a more diverse gene pool, relying on individual learning to overcome the "missing" or "not quite optimized" traits in the genetic code. This more diverse gene pool can, in turn, support more rapid evolutionary adaptation. Thus, the ability of individuals to learn can have an indirect accelerating effect on the rate of evolutionary adaptation for the entire population.

To illustrate, imagine some new change in the environment of some species, such as a new predator. Such a change will selectively favor individuals capable of learning to avoid the predator. As the proportion of such self-improving individuals in the population grows, the population will be able to support a more diverse gene pool, allowing evolutionary processes (even non-Lamarckian generate-and-test processes) to adapt more rapidly. This accelerated adaptation may in turn enable standard evolutionary processes to more quickly evolve a genetic (nonlearned) trait to avoid the predator (e.g., an instinctive fear of this animal). Thus, the Baldwin effect provides an indirect mechanism for individual learning to positively impact the rate of evolutionary progress. By increasing survivability and genetic diversity of the species, individual learning supports more rapid evolutionary progress, thereby increasing the chance that the species will evolve genetic, nonlearned traits that better fit the new environment.

There have been several attempts to develop computational models to study the Baldwin effect. For example, Hinton and Nowlan (1987) experimented with evolving a population of simple neural networks, in which some network weights were fixed during the individual network "lifetime," while others were trainable. The genetic makeup of the individual determined which weights were

trainable and which were fixed. In their experiments, when no individual learning was allowed, the population failed to improve its fitness over time. However, when individual learning was allowed, the population quickly improved its fitness. During early generations of evolution the population contained a greater proportion of individuals with many trainable weights. However, as evolution proceeded, the number of fixed, correct network weights tended to increase, as the population evolved toward genetically given weight values and toward less dependence on individual learning of weights. Additional computational studies of the Baldwin effect have been reported by Belew (1990), Harvey (1993), and French and Messinger (1994). An excellent overview of this topic can be found in Mitchell (1996). A special issue of the journal *Evolutionary Computation* on this topic (Turney et al. 1997) contains several articles on the Baldwin effect.

Parallelizing Genetic Algorithms

GAS are naturally suited to parallel implementation, and a number of approaches to parallelization have been explored. *Coarse grain* approaches to parallelization subdivide the population into somewhat distinct groups of individuals, called *demes.* Each deme is assigned to a different computational node, and a standard GA search is performed at each node. Communication and cross-fertilization between demes occurs on a less frequent basis than within demes. Transfer between demes occurs by a *migration* process, in which individuals from one deme are copied or transferred to other demes. This process is modeled after the kind of cross-fertilization that might occur between physically separated subpopulations of biological species. One benefit of such approaches is that it reduces the crowding problem often encountered in nonparallel GAS, in which the system falls into a local optimum due to the early appearance of a genotype that comes to dominate the entire population. Examples of coarse-grained parallel GAS are described by Tanese (1989) and by Cohoon et al. (1987).

In contrast to coarse-grained parallel implementations of GAS, fine-grained implementations typically assign one processor per individual in the population. Recombination then takes place among neighboring individuals. Several different types of neighborhoods have been proposed, ranging from planar grid to torus. Examples of such systems are described by Spiessens and Manderick (1991). An edited collection of papers on parallel GAS is available in Stender (1993).