ECS-801: Artificial Intelligence (AI)

Unit-IV

Machine Learning : Supervised and unsupervised learning, Decision trees, Statistical learningmodels, Learning with complete data - Naive Bayes models, Learning with hidden data - EMalgorithm,Reinforcement learning

Machine Learning :

The goal of machine learning is to design and develop algorithms that allow systems to use empirical data, experience, and training to evolve and adapt to changes that occur in their environment. A major focus of machine learning research is to automatically induce models, such as rules and patterns, from the training data it analyzes. As shown in Figure 1, machine learning combines techniques and approaches from various areas, including probability and statistics, psychology, information theory, and artificial intelligence.

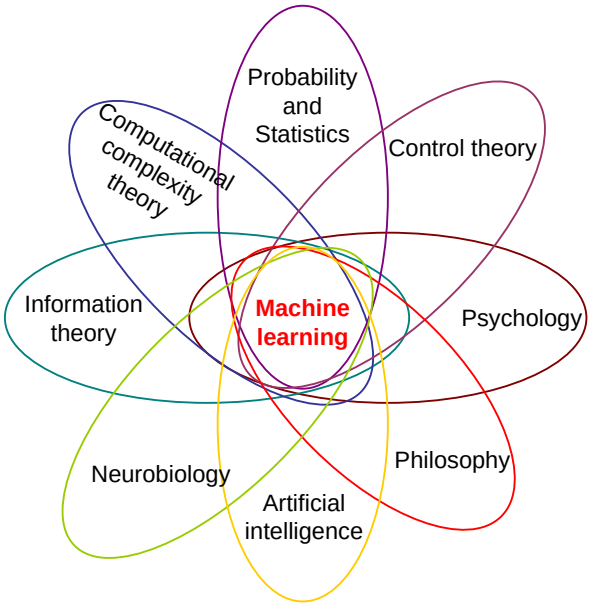


Figure 1: Machine learning is a broad discipline, combining approaches from many different areas.

Wireless sensor network (WSN) applications operate in very challenging conditions, where they constantly have to accommodate environmental changes, hardware degradation, and inaccurate sensor readings. Therefore, in order to maintain sufficient operational correctness, a WSN

application often needs to learn and adapt to the changes in its running environment. Machine learning has been used to help address these issues. A number of machine learning algorithms have been employed in a wide range of sensor network applications, including activity recognition, healthcare, education, and improving the efficiency of heating, ventilating, and air conditioning (HVAC) system.

The abundance of machine learning algorithms can be divided into two main classes, *supervised* and *unsupervised* learning, based on whether the training data instances are labeled. In supervised learning the learner is supplied with labeled training instances, where both the input and the correct output are given. In unsupervised learning the correct output is not provided with the input. Instead, the learning program must rely on other sources of feedback to determine whether or not it is learning correctly. A third class of machine learning techniques, called *semi-supervised learning*, uses a combination of both labeled and unlabeled data for training. Figure 2 shows the relationship between these three machine learning classes.

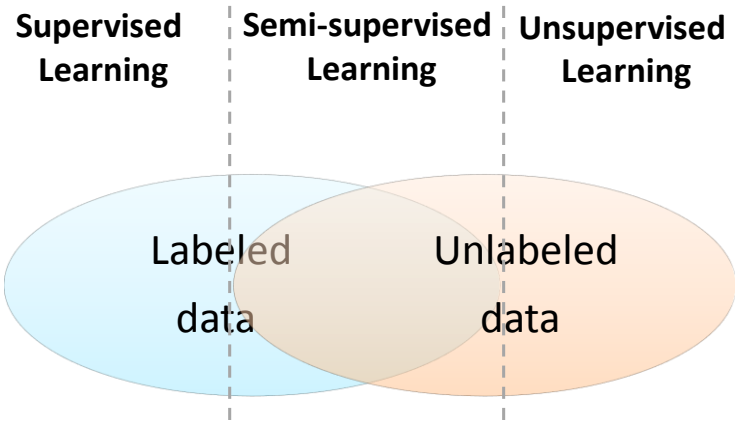


Figure 2: Machine learning algorithms are divided into supervised learning, which used labeled training data,

and unsupervised learning, where labeled training data is not available. A third class of machine learning technique, semi-supervised learning, makes use of both labeled and unlabeled training data.

In this chapter we have surveyed machine learning algorithms in sensor networks from the perspective of what types of applications they have been used for. We give examples from all three machine learning classes and discuss how they have been applied in a number of sensor network applications. We present the most frequently used machine learning algorithms, including clustering, Bayes probabilistic models, Markov models, and decision trees. We also analyze the challenges, advantages, and drawbacks of using different machine learning algorithms. Figure 3 shows the machine learning algorithms introduced in this chapter.

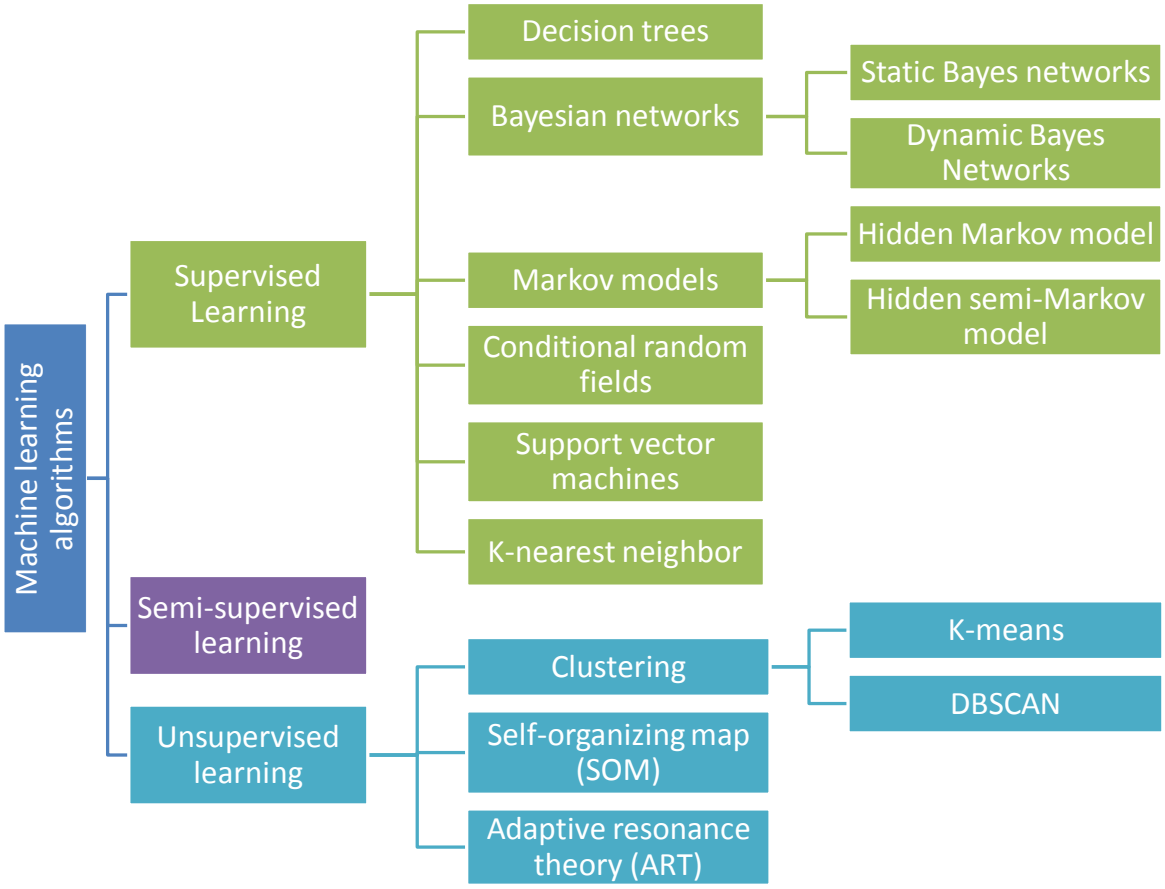


Figure 3: Classification of the machine learning algorithms most widely used in WSN applications.

Supervised Learning:

In supervised learning the learner is provided with labeled input data. This data contains a sequence of input/output pairs of the form ‹ x_i, y_i ›, where x_i is a possible input, and y_i is the correctly labeled output associated with it. The aim of the learner in supervised learning is to learn the mapping from inputs to outputs. The learning program is expected to learn a function *f* that accounts for the input/output pairs seen so far, $f(x_i) = y_i$ for all i. This function *f* is called a *classifier* if the output is discrete and a *regression function* if the output is continuous. The job of

the classifier/regression function is to correctly predict the outputs of inputs it has not seen before. For example, the inputs can be a set of sensor firings and the outputs can be the activities that have caused those sensor nodes to fire.

The execution of a supervised learning algorithm can be divided into 5 main steps (Figure 4).

Step 1 is to determine what training data is needed and collect that data. Here we need to answer two questions "What data is necessary?" and "How much of it?". The designers have to decide what training data can best represent real world scenarios for the specific application. They also need to determine how much training data should be collected. Although the more training data we have, the better we can train the learning algorithm, collecting training data and providing correct labels can often be expensive and laborious. Therefore, an application designer always strives to maintain the size of the training data large enough to provide sufficient training but also small enough to avoid any unnecessary costs associated with data collection and labeling.

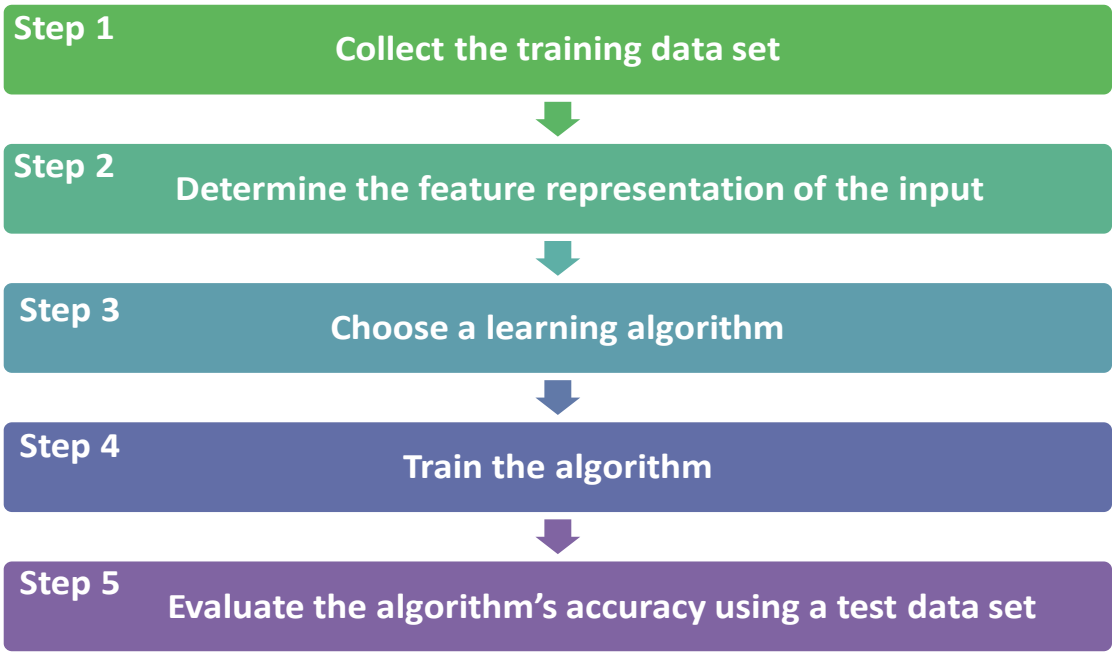


Figure 4: The stages of supervised machine learning.

Step 2 is to identify the feature set, also called *feature vector*, to be used to represent the input. Each feature in the feature set represents a characteristic of the objects/events that are being classified. There is a trade-off between the size of the feature vector and the classification accuracy of the machine learning algorithm. A large feature vector significantly increases the complexity of the classification. However, using a small feature vector, which does not contain sufficient description of the objects/events, could lead to poor classification accuracy. Therefore, the feature vector should be sufficiently large to represent the important features of the object/event and small enough to avoid excessive complexity.

Step 3 is to select a suitable learning algorithm. A number of factors have to be considered when choosing a learning algorithm for a particular task, including the content and size of the training dataset, noise in the system, accuracy of the labeling, and the heterogeneity and redundancy of the input data. We also have to evaluate the requirements and characteristics of the sensor network application itself. For example, for an activity recognition application the duration of sensor use plays a significant role in determining the activity being executed. Therefore, to achieve high activity recognition accuracy, we would prefer to use machine learning algorithms that can explicitly model state duration.

The most frequently used supervised machine learning algorithms include support vector machines, naïve Bayes classifiers, decision trees, hidden Markov models, conditional random field, and k-nearest neighbor algorithms. There are also a number of approaches that have been applied to improve the performance of the chosen classifiers, such as bagging, boosting, and using classifier ensembles. Each of the algorithms has its advantages and disadvantages, which make it suitable for some types of applications but inappropriate for others.

Step 4 is to train the chosen learning algorithm using the collected training data. In this step the algorithm learns the function that best matches the input / output training instances.

Step 5 is evaluation of the algorithm's accuracy. We assess the accuracy of the learned function with the help of testing dataset, where the testing dataset is different from the training dataset. In

this step we evaluate how accurately the machine learning algorithm classifies entries from the testing set based on the function is has learned though the training dataset.

Different supervised learning algorithms have been used and evaluated experimentally in a variety of sensor network applications. In the rest of this section we describe some of the algorithms that are most frequently used in WSN applications.

Decision trees:

Decision trees are characterized by fast execution time, ease in the interpretation of the rules, and scalability for large multi-dimensional datasets (Cabena, et al. 1998), (Han 2005). The goal of decision tree learning is to create a model that predicts the value of the output variable based on the input variables in the feature vector. Each node corresponds to one of the feature vector variables. From every node there are edges to children, where there is an edge per each of the possible values (or range of values) of the input variable associated with the node. Each leaf represents a possible value for the output variable. The output variable is determined by following a path that starts at the root and is guided by the values of the input variables.

Figure 5 shows an example decision tree for a sensor network activity detection application. In this scenario we assume that there are only two events of interest in the kitchen: *cooking* and *getting a drink*. The decision tree uses sensor node firings to distinguish between those two activities. For example, if there is movement in the kitchen and the stove is being used, the

algorithm determines that the residents must be cooking. However, if there is movement in the kitchen, the stove is not being used, and somebody opens the cups cupboard, the algorithm decides that the activity being performed at the moment is *getting a drink*. This is a simple example illustrating how decision trees can be applied to sensor network applications. In reality, the decision trees that are learned by real applications are much more complex.

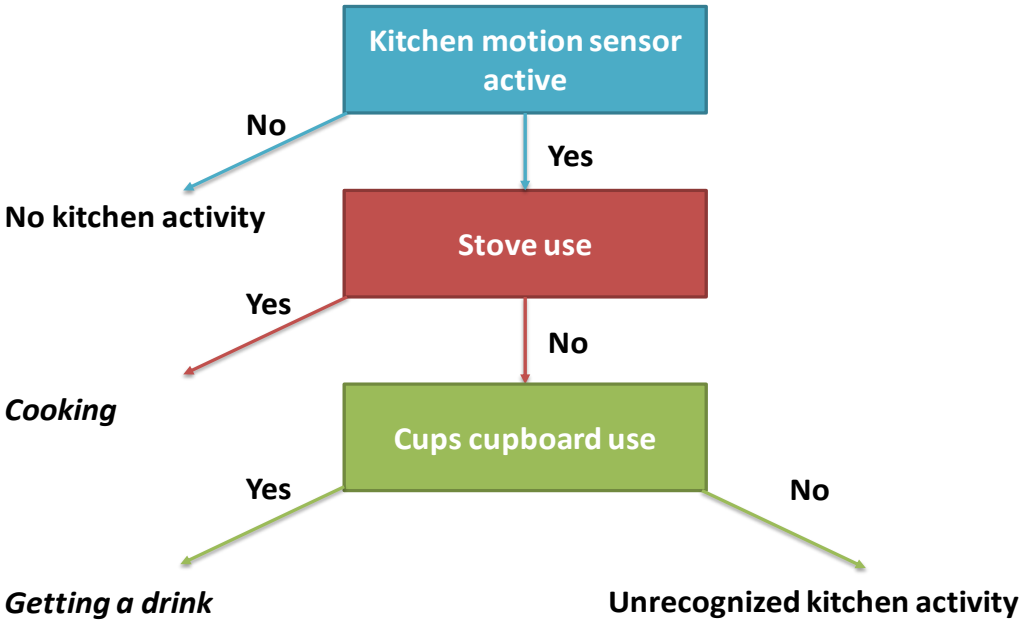


Figure 5: Example decision tree for an activity detection application. In this scenario we are only interested in two of the kitchen activities: cooking and getting a drink. The decision tree is used to determine which one of these activities is currently occurring based on the sensor nodes that are firing in the kitchen.

The C4.5 algorithms is one of the well-known, top-down, greedy search algorithms for building decision trees (Quinlan, C4.5: programs for machine learning 1993). The algorithm uses entropy and information gain metrics to induce a decision tree. The C4.5 algorithm has been used for

activity recognition in the PlaceLab project at MIT (Logan, et al. 2007). The authors of the project monitored a home deployed with over 900 sensors, including wired reed switched, current and water flow inputs, object and person motion detectors, and RFID tags. They collected data for 43 typical house activities and C4.5 was one of the classifiers used by their their activity recognition approach.

C4.5 was used for **target recognition** in an underwater wireless sensor surveillance system (Cayirci, et al. 2006). Each node in the network was equipped with multiple microsensors of various types, including acoustic, magnetic, radiation, and mechanical sensors. The readings from these sensors were used by the decision tree recognition algorithms to classify submarines, small delivery vehicles, mines, and divers.

C4.5 was also used as part of an algorithm to **automatically recognize physical activities and their intensities** (Tapia, et al. 2007). The algorithm monitors the readings of triaxal wireless accelerometers and wireless heart rate monitos. The approach was evaluated using datasets consisting of 10 physical gymnasium activities collected from a total of 21 people.

Bayesian network classifiers:

Bayesian probability interprets the concept of probability as *degree of belief*. A Bayesian classifier analyzes the feature vector describing a particular input instance and assigns the

instance to the most likely class. A Bayesian classifier is based on applying Bayes' theorem to evaluate the likelihood of particular events. Bayes' theorem gives the relationship between the *prior* and *posterior* beliefs for two events. In Bayes' theorem, P(A) is the prior initial belief in A. P(A|B) is the posterior belief in A, after B has been encountered, i.e. the conditional probability of A given B. Similarly for B, P(B) is the prior initial belief in A, and P(B|A) is the posterior belief in B given A. Assuming that $P(B) \neq 0$, Bayes' theorem states that

$$P(A \mid B) = \frac{P(B \mid A) \times P(A)}{P(B)}$$

The Bayesian network is a probabilistic model that represents a set of random variables and their conditional dependencies via a direct acyclic graph (DAG). For example, a Bayesian network could represent the probabilistic relationships between activities and sensor readings. Given a set of sensor readings, the Bayesian network can be used to evaluate the probabilities that various activities are being performed.

Bayesian networks have a number of advantages. Since a Bayes network only relates nodes that are probabilistically related by a causal dependency, an enormous saving of computation can result. Therefore, there is no need to store all possible configurations of states. Instead, all that needs to be stored is the combinations of states between sets of related parent-child nodes. Also, Bayes networks are extremely adaptable. They can be started off small, with limited knowledge about the domain, and grow as they acquire new knowledge.

Bayes networks have been applied to a variety of **sensor fusion problems**, where data from various sources must be integrated in order to build a complete picture of the current situation. They have also been used in **monitoring and alerting applications** where the application should recognize whether specific events have occurred and decide if an alert or a notification should be sent. Further, they have been applied to a number of **activity recognition** applications and evaluated using numerous single and multiple-resident home deployments.

Bayesian networks can be divided into two groups, *static* and *dynamic*, based on whether they are able to model temporal aspects of the events / activities of interest. We introduce an example classifier for each of these two classes: static naïve Bayes classifier and dynamic naïve Bayes classifier.

Static Bayesian network classifiers:

A very commonly used representative of the static Bayesian networks is the static *naïve Bayes classifier*. Learning Bayesian classifiers can be significantly simplified by making the naïve assumption that the features describing a class are independent. The classifier makes the assumption that the presence or absence of a feature of a class is unrelated to the presence or absence of any of the other features in the feature vector. The naïve Bayes classifier is one of the most practical learning methods and it has been widely used in many sensor network applications, including **activity recognition** in residence for elders (van Kasteren and Kröse, Bayesian activity recognition in residence for elders 2007), activity recognition in the PlaceLab

project at MIT (Logan, et al. 2007), **outlier detection** (Janakiram, AdiMallikarjuna Reddy and Phani Kumar 2006), and **body sensor networks** (Maurer, et al. 2006).

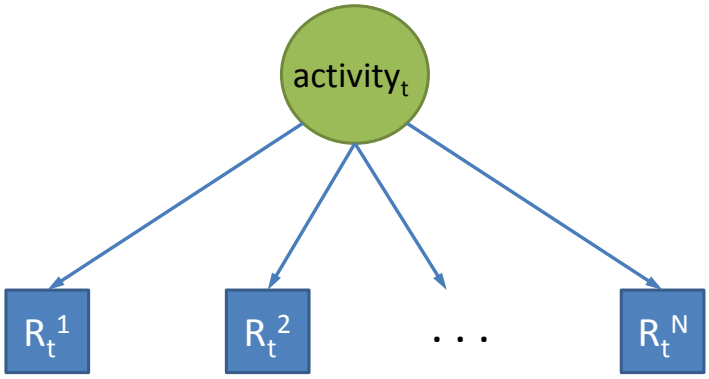


Figure 6: Static Bayesian network: activity$_t$ denotes the activity being detected at time t, and R_t^i represents the data from sensor *i* at time t.

Figure 6 shows a naïve Bayesian model for the recognition of an activity. In this scenario the activity at time *t*, activity$_t$, is independent of any previous activities. It is also assumed that the sensor data R_t is only dependent on the activity^t.

Naïve Bayes classifiers have a number of advantages:

1. They can be trained very efficiently.
2. They are very well suited for categorical features.
3. In spite of their naïve design and the independence assumptions, naïve Bayes classifiers have performed very well in many complex real-world situations. They can work with more than 1000 features.

4. They are good for combining multiple models and can be used in an iterative way.

A disadvantage of naïve Bayes classifiers is that, if conditional independence is not true, i.e. there is dependence between the features of the analyzed classes, they may not be a good model. Also naïve Bayes classifiers assume that all attributes that influence a classification decision are observable and represented. Despite these drawbacks, experiments have demonstrated that naïve Bayes classifiers are very accurate classifiers in a number of problem domains. Simple naïve Bayes networks have even been proved comparable to more complex algorithms, such as decision trees (E. Tapia 2004).

Dynamic Bayesian network classifiers

Another disadvantage of static Bayesian networks is that they cannot model the temporal aspect of sensor network events. Dynamic Bayesian networks, however, are capable of representing a sequence of variables, where the sequence can be consecutive readings from a sensor node. Therefore, dynamic Bayesian networks, although more complex, might be better suited for modeling events and activities in sensor network applications.

Figure 7 shows a naïve dynamic Bayesian model, where the $activity_{t+1}$ variable is directly influenced only by the previous variable, $activity_t$. The assumption with these models is that an event can cause another event in the future, but not vise-versa. Therefore, directed arcs between events/activities should flow forward in time and cycles are not allowed.

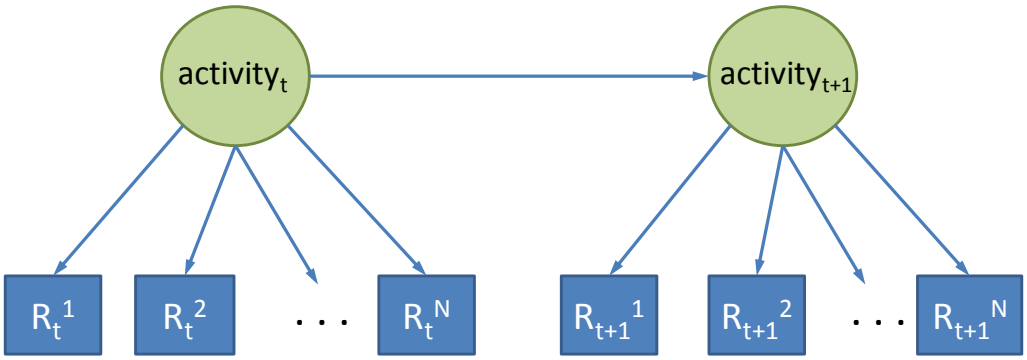


Figure 7: An example of a naïve dynamic Bayesian network.

Dynamic models have been used in **activity recognition** applications. A naïve dynamic Bayes classifier is compared to a naïve static Bayes classifier using two publicly available datasets (van Kasteren and Kröse, Bayesian activity recognition in residence for elders 2007). The dynamic Bayes classifier is shown to achieve higher activity recognition accuracy than the static model. A dynamic Bayesian filter was successfully applied to the **simultaneous tracking and activity recognition** (STAR) problem, which exploits the synergy between location and activity to provide the information necessary for automatic health monitoring (Wilson and Atkenson 2005).

Markov models:

A process is considered to be Markov if it exhibits the Markov property, which is the lack of memory, i.e. the conditional probability distribution of future states of the process depends only on the present state, and not on the events that preceded it. We discuss two types of Markov models: hidden Markov model and hidden semi-Markov model.

1. *Hidden Markov model*

A hidden Markov model (HMM) can be viewed as a simple dynamic Bayesian network. When using an HMM, the system is assumed to be a Markov process with unobserved (hidden) states. Even though the sequence of states is hidden, the output which is dependent on the state is visible. Therefore, at each time step there is a hidden variable and an observable output variable. In sensor network applications the hidden variable could be the event or activity performed, and the observable output variable is the vector of sensor readings.

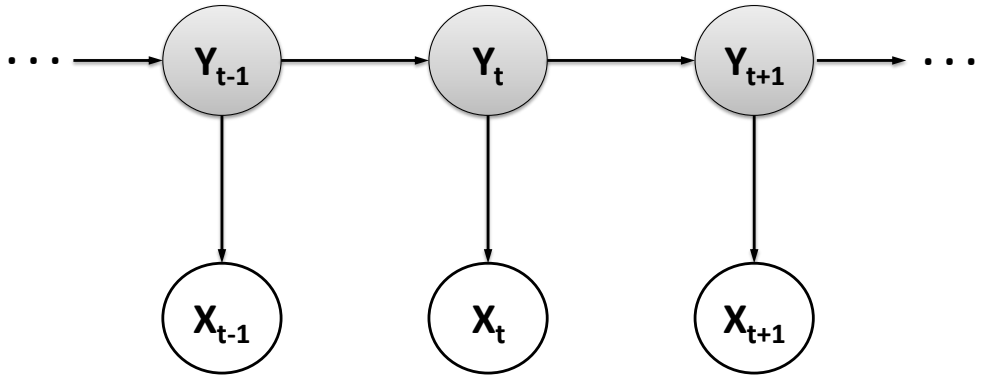


Figure 8: Hidden Markov model example. The states of the system Y_i are hidden, but their corresponding outputs X_i are visible.

Figure 8 shows an example HMM where the states of the system *Y* are hidden, but the output variables *X* are visible. There are two dependency assumptions that define this model, represented by the directed arrows in the figure:

1. Markov assumption: The hidden variable at time t, namely Y_t, depends only on the

previous hidden variable Y_{t-1}, (Rabiner 1989);

2. The observable output variable at time t, namely X_t, depends only on the hidden variable Y_t.

With these assumptions we can specify an HMM using three probability distributions:

1. *Initial state distribution:* the distribution over initial states $p(Y_1)$;
2. *Transition distribution*: the distribution $p(Y_t \mid Y_{t+1})$, which represents the probability of going from one state to the next;
3. *Observation distribution*: the distribution $p(X_t \mid Y_t)$, which indicates the probability that the hidden state Y_t would generate observation X_t.

Learning the parameters of these distributions corresponds to maximizing the joint probability distribution p(X, Y) of the paired observation and label sequences in the training data. Modeling the joint probability distribution p(X, Y) makes HMMs a ***generative model***.

HMMs have been extensively used in many sensor network applications. Most of the earlier work on **activity recognition** used HMMs to recognize the activities from sensor data (Wilson and Atkenson 2005) (Patterson, et al. 2005) (van Kasteren, Noulas, et al. 2008). An HMM is also used in the smart thermostat project (Lu, et al. 2010). The smart thermostat technology automatically **senses the occupancy and sleep patterns in a home**, and uses these patterns to automatically operate the heating, ventilation, and cooling (HVAC) system in the home. The authors employ an HMM to estimate the probability of the home being in each of three states:

unoccupied, occupied and the residents are active, and occupied with the residents sleeping. HMMs were also applied in a **biometric identification** application for multi-resident homes (Srinivasan, Stankovic and Whitehouse, Using Height Sensors for Biometric Identification in Multi-resident Homes 2010). In this project height sensors were mounted above the doorways in a home and an HMM was used to identify the location of each of the residents.

A weakness of conventional HMMs is their lack of flexibility in modeling state durations. With HMMs, there is a constant probability of changing state, given that the system is in its current state of the model. This, however, limits the modeling capability. For example, the activity preparing dinner typically spans at least several minutes. To prepare dinner in less than a couple of minutes is not very usual. The geometric distribution used by HMMs to represent time duration cannot be used to represent event distributions where shorter durations are less possible.

Hidden Semi-Markov Models:

A hidden semi-Markov model (HSMM) differs from a hidden Markov model in that HSMMs explicitly model the duration of hidden states. This means that the probability of there being a change in the hidden state depends on the amount of time that has elapsed since entry into the current state.

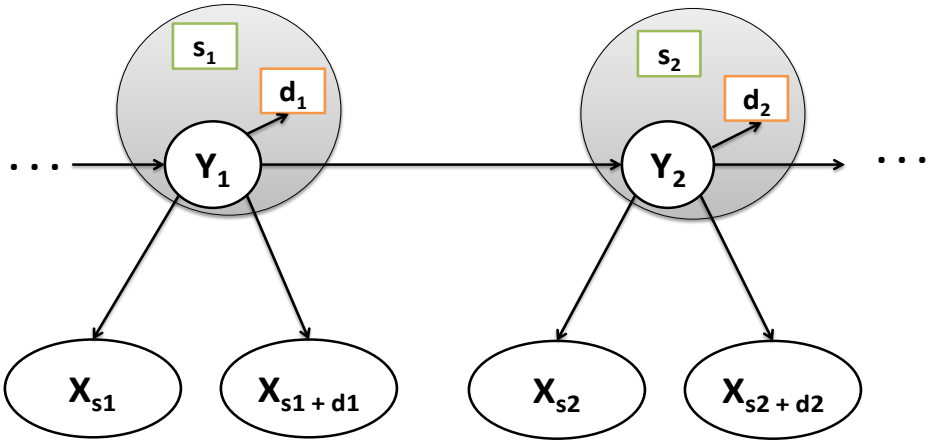


Figure 9: Hidden semi-Markov model. Each hidden state y_i is characterized by start position s_i and a duration d_i. This means that the system is in state y_i from time s_i to time $s_i + d_i$.

A number of projects have used HSMMs to **learn and recognize human activities** of daily living (Duong, et al. 2009) (Zhang, et al. 2008) (van Kasteren, Englebienne and Kröse, Activity recognition using semi-Markov models on real world smart home datasets. 2010). HSMMs were also applied to **behavior understanding** from video streams in a nursing center (Chung and Liu 2008). The proposed approach infers elderly behaviors through three contexts: spatial, activities, and temporal. HSMM were also used in a **mobility tracking** application for cellular networks (Mark and Zaidi 2002).

The activity recognition accuracy achieved by HSMM is compared to that of HMM (van Kasteren, Englebienne and Kröse, Activity recognition using semi-Markov models on real world smart home datasets. 2010). The authors evaluate the recognition performance of these models using two fully annotated real world datasets consisting of several weeks of data. The first

dataset was collected in a 3-room single-resident apartment and the second dataset was from a 6-room single-resident house. The results show that HSMM consistently outperforms HMM. This indicates that accurate duration modelling is important in real world activity recognition applications as it can lead to significantly better performance. The use of duration in the classification process helps especially in scenarios where the sensor data does not provide sufficient information to distinguish between activities.

Conditional random fields

Conditional random fields (CRF) are often considered an alternative to hidden Markov models. The CRF is a statistical modeling method, which is a type of an undirected probabilistic graphical model that defines a single log-linear distribution over label sequences given a particular observation sequence. It is used to encode known relationships between observations and construct consistent interpretations.

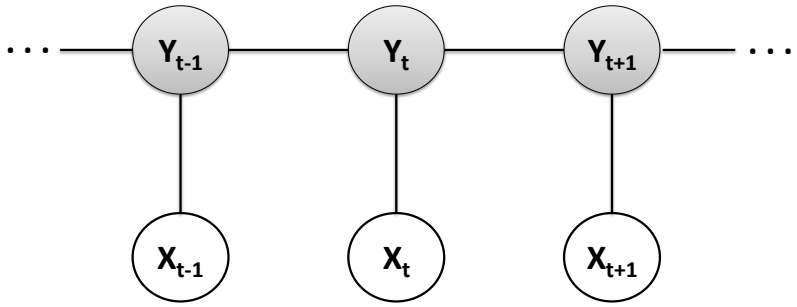


Figure 10: A linear-chain conditional random field (CRF) model. Similarly to an HMM, the states of the system Y_i are hidden, but their corresponding outputs X_i are visible. Unlike the HMM model, however, the graph represented by the CRF model is undirected.

The CRF model that most closely resembles an HMM is the linear-chain CRF. As figure 10 shows, the model of a linear-chain CRF is very similar to that of an HMM (Figure 8). The model still contains hidden variables and corresponding observable variables at each time step. However, unlike the HMM, the CRF model is undirected. This means that two connected nodes no longer represent a conditional distribution. Instead we can talk about potential between two connected nodes. In comparison to HMM, the two conditional probabilities, *observation* probability $p(X_t \mid Y_t)$ and *transition* probability $p(Y_t \mid Y_{t+1})$, have been replaces by the corresponding potentials. The essential difference lies in the way we learn the model parameters. In the case of HMMs the parameters are learned by maximizing the *joint* probability distribution p(X, Y). CRFs are ***discriminative models***. The parameters of a CRF are learned by maximizing the *conditional* probability distribution p(Y | X), which belongs to the family of exponential distributions (Sutton and McCailum 2006).

CRF models have been applied to **activity recognition** in home from video streams, in which primitive actions, such as '*go-from-A- to-B'* are recognized in a lab-like dining room and kitchen setup (Truyen, Bui and Venkatesh, Human Activity Learning and Segmentation using Partially Hidden Discriminative Models 2005). The results from these experiments show that CRFs perform significantly better than the equivalent generative HMMs even when a large portion of the data labels are missing. CRFs were also used for **modeling concurrent and interleaving activities** (Hu, et al. 2008). The authors perform experiments using one of the MIT PlaceLab

datasets (Logan, et al. 2007), PLA1, which consists of four hours of sensor data.

T. van Kasteren et al. use four different datasets, two bathroom datasets and two kitchen datasets, to compare the performance of HMM to that of CRF (van Kasteren, Englebienne and Kröse, Activity recognition using semi-Markov models on real world smart home datasets. 2010). The experiments show that, when applied to activity recognition tasks, CRF models achieve higher accuracy than HMM models. The authors contribute the results to the flexibility of discriminative models, such as CRF, in dealing with violations of the modeling assumptions. However, the higher accuracy achieved by CRF models comes at a price:

1. Discriminative models take much longer to train than their generative counterpart.
2. Discriminative models are more prone to *overfitting*. Overfitting occurs when a model describes random noise instead of the underlying relationship. This happens when the model is trained to maximize its performance on the training data. However, a model's efficiency is determined not by how well it performs on the training data but by its generality and how it performs on unseen data.

Whether the improved recognition performance of CRFs is worth the extra computational cost depends on the application. The data can be modeled mode accurately using an HSMM, which allows both speedy learning and good performance, and is less prone to overfitting. However, it does result in slower inference and depends on correct modeling assumptions for the durations.

Semi-Markov conditional random fields

Similarly to HMMs, which have their semi-Markov variant, conditional random fields also have a semi-Markov variant: semi-Markov conditional random fields (SMCRF). An example SMCRF model is shown in Figure 11. The SMCRF inherits features from both semi-Markov models and CRFs:

1. It models the duration of states explicitly (like HSMM).
2. Each of the hidden states is characterized by a start position and duration (like HSMM).
3. The graph of the model is undirected (like CRF).

Hierarchical SMCRF were used in an **activity recognition** application on a small laboratory dataset from the domain of video surveillance (Truyen, Phung, et al. 2008). The task was to recognize indoor trajectories and activities of a person from his noisy positions extracted from the video. The data had 90 sequences, each of which corresponded to one of three possible activities: preparing a short meal, preparing a normal meal, and having a snack. The hierarchical SMCRF outperformed both a conventional CRF and a dynamic CRF.

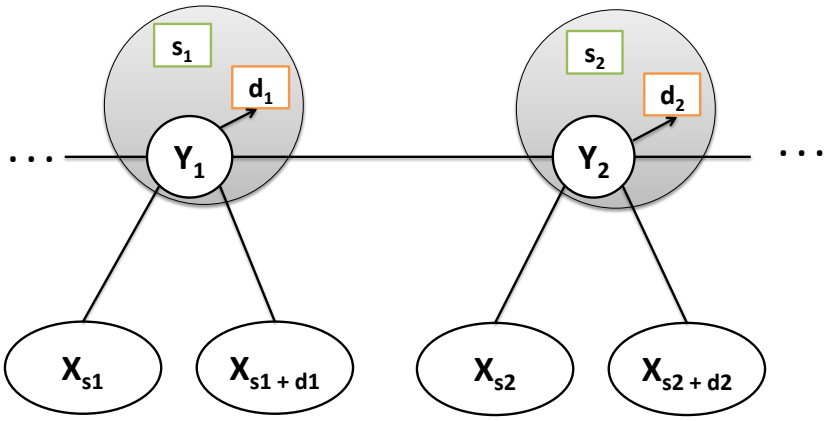


Figure 11: An example semi-Markov conditional random field (CRF). Similarly to an HSMM model, each of the hidden states y_i is characterized by start position s_i and a duration d_i. However, unlike an HSMM, the HMCRF graph is undirected.

SMCRFs were also used for activity recognition by van Kasteren et al. (van Kasteren, Englebienne and Kröse, Activity recognition using semi-Markov models on real world smart home datasets. 2010). The results show that unlike the big improvement achieved by using HSMMs over HMMs, SMCRFs only slightly outperform CRFs. The authors attribute this result to the fact that CRFs are more robust in dealing with violations to the modeling assumptions. Therefore, allowing to explicitly model duration distributions might not have the same significant benefits as seen with HSMM

Support vector machines

A support vector machine (SVM) is a non-probabilistic binary linear classifier. The output prediction of an SVM is one of two possible classes. Given a set of training instances, each marked as belonging to one of two classes, an SVM algorithm builds an N-dimensional

hyperplane model that assigns future instances into one of the two possible output classes.

As shown in Figure 12, an SVM model is a representation of the input instances as points in space, mapped so that the instances of the separate classes are divided by a clear gap. New examples are then mapped into that same space and predicted to belong to a class based on which side of the gap they fall on. In other words, the goal of the SVM analysis is to find a line that separates the instances based on their class. There are an infinite number of possible lines and one of the challenges with SVM models is finding the optimal line.

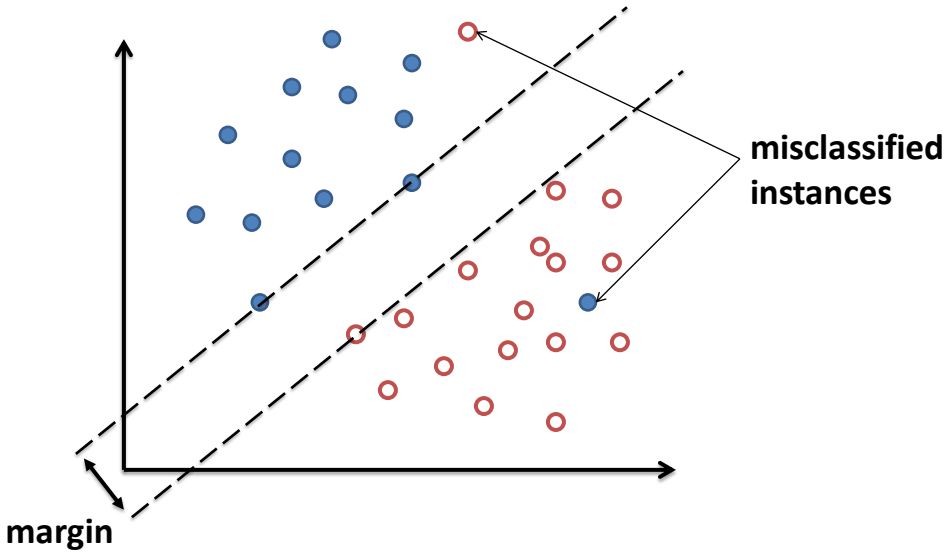


Figure 12: A two-dimensional support vector machine (SVM) model. The instances of the two possible classes are divided by a clear gap.

SVMs have been applied to a large number of sensor network applications. Sathik et al. use SVMs in an **early forest fire detection** applications (Mohamed Sathik, Syed Mohamed and Balasubramanian 2010). SVMs were also applied to **target classification** applications for

distributed sensor networks (Li, et al. 2001). The experiments were performed on real seismic and acoustic data. SVMs are compared to a k-nearest neighbor algorithm and a maximum likelihood algorithm and are shown to achieve the highest target classification accuracy. Tran et al. use SVMs to achieve accurate **geographic location estimations** for nodes in a WSN, where the majority of nodes do not have effective self-positioning functionality (Tran and Nguyen 2008). SVMs were also applied to investigating the possibility of **recognizing visual memory recall** (Bulling and Roggen 2011). The project aims to find if people react differently to images they have already seen as opposed to images they are seeing for the first time.

B. K-nearest neighbor algorithms

The k-nearest neighbor (k-NN) algorithm is among the simplest of machine learning algorithms, yet it has proven to be very accurate in a number of scenarios. The training examples are vectors in a multidimensional feature space, each with a class label. The training phase of the algorithm consists only of storing the feature vectors and class labels of the training samples. A new instance is classified by a majority vote of its neighbors, with the instance being assigned the class that is most common among its *k* nearest neighbors.

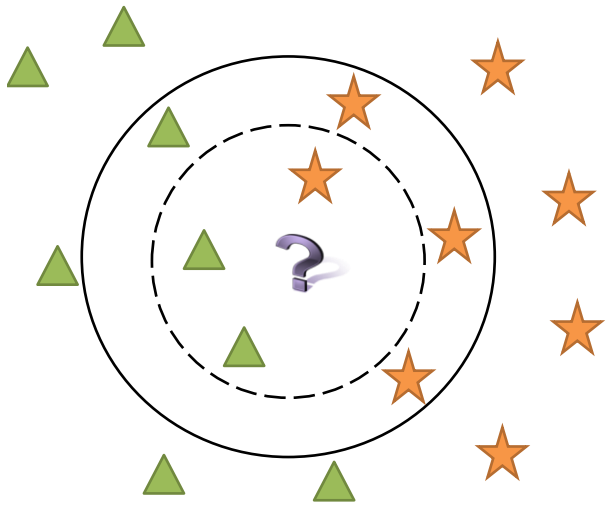


Figure 13: Example of k-nearest algorithm classification. The question mark is the test sample and it should be classified as either a star or a triangle. If k = 3, the test sample is assigned to the class of triangles because there are 2 triangles and 1 star inside the inner circle. If k = 7, the test sample is assigned to the class of stars since there are 4 stars and 3 triangles in the outer circle.

The best choice of *k* depends upon the data. *k* must be a positive integer and it is typically small. If *k* = 1, the new instance is simply assigned to the class of its nearest neighbor. Larger values of *k* reduce the effect of noise on the classification but make boundaries between classes less distinct. A good *k* can be selected by various heuristic techniques, for example cross-validation.

Although the k-NN algorithm is quite accurate, the time required to classify an instance could be high since the algorithm has to compute the distances (or similarity) of that instance to all the instances in the training set. Therefore, the classification time of k-NN is proportional to the number of features and the number of training instances.

k-NN algorithms have been applied to a wide variety of sensor network applications. Ganesan et

al. propose the use of k-NN for **spatial data interpolation** in sensor networks (Ganesan, et al. 2004). Due to its simplicity, k-NN allows the sampling to be done in a distributed and inexpensive manner. A disadvantage with this approach, however, is that k-NN interpolation techniques might perform poorly in highly irregular settings. Winter et al. also analyze the application of k-NN queries for **spatial data queries** in sensor networks (Winter, Xu and Lee 2005). They design two algorithms based on k-NN, which are used to intelligently prune off irrelevant nodes during query propagation, thus reducing the energy consumption while maintaining high query accuracy. Duarte et al. evaluate the accuracy of k-NN in the context of **vehicle classification** (Duarte and Hu 2004). The authors collect a real-world dataset and analyze both the acoustic and the seismic modality. The results show that in this application scenario k-NN algorithms achieve comparable accuracy to that of SVMs.

Unsupervised Learning:

Collecting labeled data is resource and time consuming and accurate labeling is often hard to achieve. For example, obtaining sufficient training data for activity recognition in a home might require three or four weeks of collecting and labeling data. Further, labeling is difficult not only for remote areas which are not easily accessible, but also for home and commercial building deployments. For any of those deployments someone has to perform the data labeling. In a home deployment, the labeling can be done by the residents themselves, in which case they have to keep a log of what they are doing and at what time. Previous experience has shown that these logs are often incomplete and inaccurate. An alternative solution is to install cameras throughout

the house and monitor the activities of the residents. However, this approach is considered to be privacy-invasive and therefore not suitable.

In unsupervised learning the learner is provided with input data, which has not been labeled. The aim of the learner is to find the inherent patterns in the data that can be used to determine the correct output value for new data instances. The assumption here is that there is a structure to the input space, such that certain patterns occur more often than others, and we want to see what generally happens and what does not. In statistics, this is called *density estimation.*

Unsupervised learning algorithms are very useful for sensor network applications for a number of reasons:

- Collecting labeled data is resource and time consuming;
- Accurate labeling is hard to achieve;
- Sensor networks applications are often deployed in unpredictable and constantly changing environments. Therefore, the applications need to evolve and learn without any guidance, by using unlabeled patterns.

A variety of unsupervised learning algorithms have been used in sensor network applications, including different clustering algorithms, such as k-means and mixture models; self-organizing maps (SOM); and adaptive resonance theory (ART). In the rest of this section we describe some of the most commonly used unsupervised learning algorithms.

Clustering

Clustering, also called *cluster analysis*, is one form of unsupervised learning. It is often employed in pattern recognition tasks and activity detection applications. A clustering algorithm partitions the input instances into a fixed number of subsets, called *clusters*, so that the instances in the same cluster are similar to one another with respect to some set of metrics.

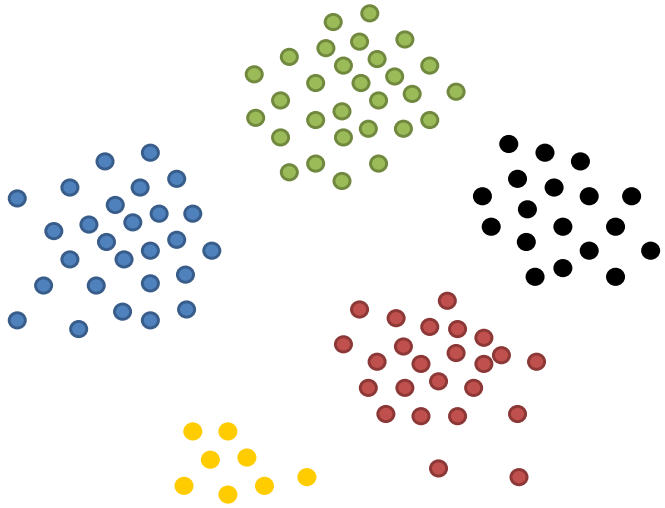


Figure 14: A clustering algorithm divides the set of input data instances into groups, called clusters. The instances in the same group are more similar to each other than to those in other clusters.

Cluster analysis itself is not one specific algorithm, but the general task to be solved. The clustering can be achieved by a number of algorithms, which differ significantly in their notion of what constitutes a cluster and how to efficiently find them. The choice of an appropriate clustering algorithms and parameter settings, including values, such as the distance function to use, a density threshold, or the number of expected clusters, depends on the individual dataset

and intended use of the results.

The notion of a *cluster* varies between algorithms and the clusters found by different algorithms vary significantly in their properties. Typical cluster models include:

- *Connectivity models*: An example of a connectivity model algorithm is hierarchical clustering which builds models based on distance connectivity.
- *Centroid models*: A representative of this set of algorithms is the k-means algorithm. With this algorithm each cluster is represented by a single mean vector.
- *Distribution models*: clusters are modeled using statistics distributions.
- *Density models*: an example of density model clustering is DBSCAN. In this type of clustering, clusters are identified as areas with higher density than non-clusters.
- *Group models*: These clustering algorithms are not able to provide a refined model for the results. Instead, they can only generate the group information.

We discuss in more detail two of the most common clustering algorithms used in sensor network applications: k-means clustering, and DBSCAN clustering.

K-means clustering

The goal of k-means clustering is to partition the input instances into *k* clusters, where each instance belongs to the cluster with the nearest mean. Since the problem is NP-hard, the common approach is to only search for approximate solutions. There are a number of efficient heuristic algorithms that can quickly converge to a local optimum, such as the Lloyd's algorithm (Lioyd 1982). Since the algorithms only find local optimums, they are usually run multiple times with

different random initializations.

An advantage of the k-means algorithm is that it is simple and converges quickly when the number of dimensions of the data is small. However, k-means clustering also has a number of drawbacks. First, *k* must be specified in advance. Also, the algorithms prefer clusters of approximately similar sizes. This often leads to incorrectly cut borders in between clusters, which is not surprising since, being centroid a model algorithm, k-means optimizes for cluster center rather than cluster borders.

Figure 15 shows a clustering example where k=2 and k-means is not able to accurately define the borders between the two clusters. There are two density clusters in that figure. One of them is much larger and contains circles. The other one is smaller and consists of triangles. Since k-means optimizes for cluster center and tends to produce clusters with similar sizes, it incorrectly splits the data instances into a green and a red cluster. These two clusters, however, do not overlap with the original density clusters of the input data.

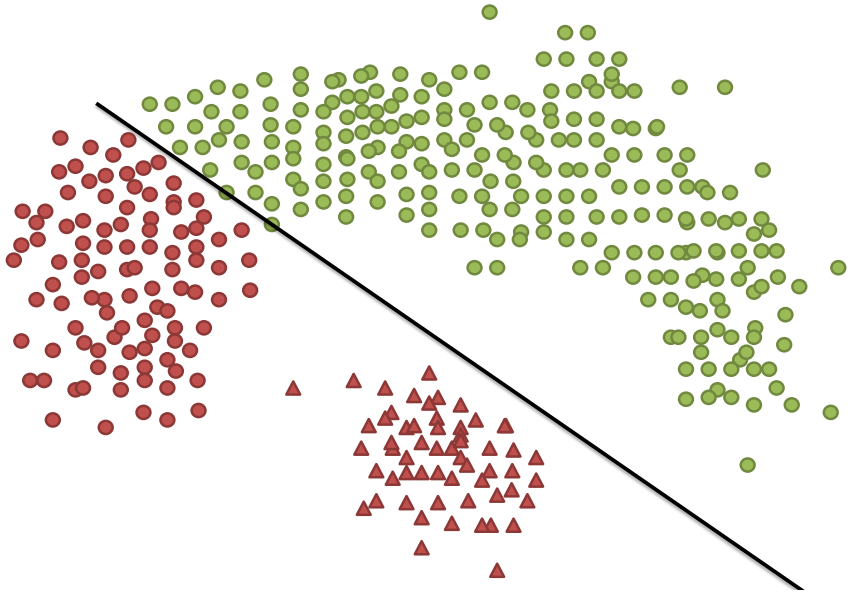


Figure 15: K-means clustering might incorrectly cut the bothers between density-based clusters.

K-means clustering has been used in a number of WSN applications. A k-means algorithm is used in the **fingerprint and timing-based snooping (FATS) security attack** to cluster together sensors that are temporally correlated (Srinivasan, Stankovic and Whitehouse, Protecting your Daily In-Home Activity Information from a Wireless Snooping Attack 2008). This allows the attack to identify sensors that fire together, and hence identify sensors that are located in the same room. K-means clustering has also been used to address the ***multiple sink location problem*** in large-scale WSNs (Oyman and Ersoy 2004). In large scale-networks with a large number of sensor nodes, multiple sink nodes should be deployed not only to increase the manageability of the network but also to prolong the lifetime of the network by reducing the energy dissipation of each node. Al-Karaki et al. apply k-means clustering to **data aggregation**,

and more specifically to finding the minimum number of aggregation points in order to maximize the network lifetime (Al-Karaki, Ul-Mustafa and Kamal 2004). The results from their experiments show that, compared to a number of other algorithms, such as a genetic algorithm and a simple greedy algorithm, k-means clustering achieves the highest network lifetime extension.

DBSCAN clustering

The density-based spatial clustering for applications with noise (DBSCAN) is the most popular density-based clustering algorithm. In density-based clustering, clusters are defined as areas of higher density than the remainder of the dataset. DBSCAN requires two parameters: distance threshold (Eps-neighborhood of a point) and minimum number of points required to form a cluster (MinPts) (Ester, et al. 1996). DBSCAN is based on connecting points within a certain distance of each other, i.e. points which are in the same Eps-neighborhood. However, in order to make a cluster, DBSCAN requires that for each point in the cluster there are at least MinPts number of points in the Eps-neighborhood. Figure 16 shows an example of DBSCAN clustering. The dataset is the same as that in Figure 15 but since a density-based clustering algorithm has been used, the data is clustered correctly.

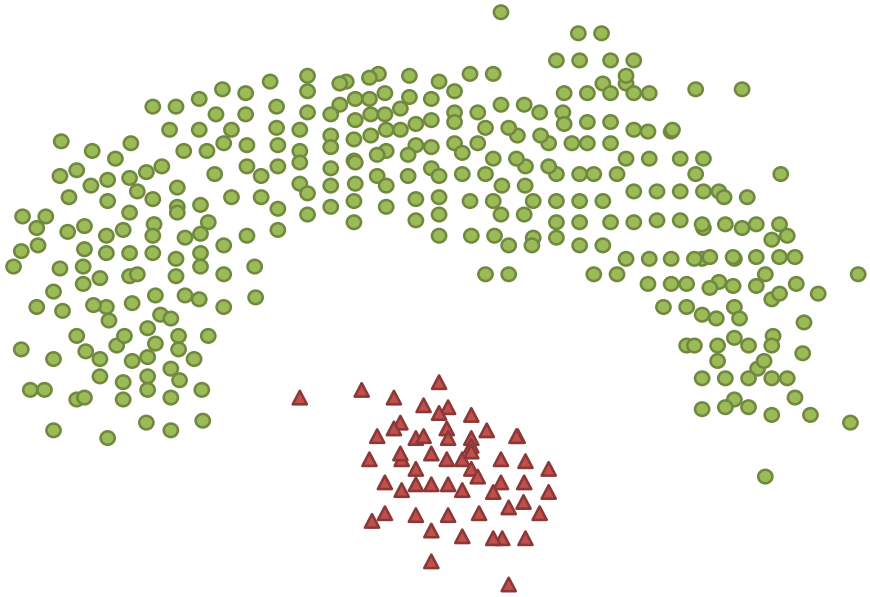


Figure 16: An example density-based clustering with DBSCAN.

An advantage of DBSCAN is that, unlike many other clustering algorithms, it can form clusters of any arbitrary shape. Another useful property of the algorithm is that its complexity is fairly low and it will discover essentially the same clusters in each run. Therefore, in contrast to k-means clustering, DBSCAN can be run only once rather than multiple times. The main drawback of DBSCAN is that it expects sufficiently significant density drop in order to detect cluster borders. If the cluster densities decrease continuously, DBSCAN might often produce clusters whose borders look arbitrary.

In sensor network applications, DBSCAN has been used as part of the **FATS security attack** to identify the function of each room, such as bathroom, kitchen, or bedroom (Srinivasan, Stankovic and Whitehouse, Protecting your Daily In-Home Activity Information from a Wireless

Snooping Attack 2008). DBSCAN generates temporal activity clusters, each of which forms a continuous temporal block with a relatively high density of sensor firings. Experiments show that DBSCAN performs very well because it automatically leaves out outliers and computes high-density clusters. However, when DBSCAN is applied to the step of identifying which sensors are in the same room, k-means clustering performs much better. This is especially true for scenarios where all devices are highly correlated temporally and there is no significant density drop on the boundary of clusters.

Apiletti et al. also apply DBSCAN to **detecting sensor correlation** (Apiletti, Baralis and Carquitelli 2011). The authors perform experiments using data collected from a sensor network deployed in university labs. The results show that DBSCAN is able to identify different numbers of clusters based on which day of the week it is analyzing. This allows it to construct more accurate models for the sensor use parrerns in the labs. DBSCAN also successfully detects noisy sensors.

Self-organizing map (SOM)

Self-organizing maps (SOM) provide a way of representing multidimensional data in much lower dimensional spaces – typically one or two dimensions. The process of reducing the dimensionality of the feature vectors is a data compression technique known as *vector quantisation*. SOMs, as indicated by their name, produce a representation of the compressed feature space, called a map. An extremely valuable property of these maps is that the information is stored in such a way that any topological relationships within the training set are maintained.

An SOM contains components called nodes. Each node is associated with 1) a position in the map space and 2) a vector of weights, where the dimension of this vector is the same as that of the input data instances. The nodes are regularly spaced in the map, which is typically a rectangular or a hexagonal grid. A typical example of SOMs is a color map (Figure 17). Each color is represented by a 3-dimensional vector containing values for red, green, and blue. However, the color SOM represents the colors in a 2-dimensional space.

Figure 17: An example SOM representation for colors.

The procedure of placing an input data instance onto the map is the following:

1. Initialize the weights of the nodes on the map.
2. Choose an input training instance.
3. Find the node with the closest vector to that of the input instance. This node is called the best matching unit (BMU).
4. Calculate the radius of the BMU's neighborhood. This value is often set to the radius of the whole map, but it decreases at each time step. Any node found within this radius is

considered to be inside the BMU's neighborhood.

5. Once the BMU is located, it is assigned the values from the vector of the input instance. In addition, the weights of the nodes close to the BMU are also adjusted towards the input vector. The closer a neighbor node is to the BMU, the more its weight is altered.

In sensor networks, SOMs have been applied to **anomaly detection** caused by faulty sensors and unusual phenomenon, such as harsh environmental conditions (Siripanadorn, Hattagam and Teaumroong 2010). Paladina et al. have also used SOMs for **node localization** (Paladina, et al. 2007). Their localization technique is based on a simple SOM implemented on each of the sensor nodes. The main advantages of this approach are the limited storage and computing cost. However, the processing time required by the SOMs increases with the size of the input data. Giorgetti et al. have also applied SOMs to addressing node localization (Giorgetti, Gupta and Manes 2007). Their SOM-based algorithm computes *virtual* coordinates that are used in location-aided routing. If the location information for a few anchor nodes is available, the algorithm is also able to compute the *absolute* positions of the nodes. The results from the experiments further show that the SOM-based algorithm performs especially well for networks with low connectivity, which tend to be harder to localize, and in the presence of irregular radio patterns or anisotropic deployment. A variation of an SOM, called a growing self-organized map (GSOM) is employed to achieve accurate **detection of human activities** of daily living within smart home environments (Zheng, Wang and Black 2008).

Adaptive resonance theory (ART)

Most existing learning algorithms are either *stable* (they preserve previously learned information) or *plastic* (they retain the potential to adapt to new input instances indefinitely). Typically, algorithms that are stable cannot easily learn new information, and algorithms that are plastic tend to forget the old information they have learned. This conflict between stability and plasticity is called the *stability-plasticity dilemma* (Carpenter and Grossberg 1987).

The adaptive resonance theory (ART) architectures attempt to provide a solution to the stability-plasticity dilemma. ART is a family of different neural architectures that address the issue of how a learning system can preserve its previously learned knowledge while keeping its ability to learn new patterns. An ART model is capable of distinguishing between familiar and unfamiliar events, as well as between expected and unexpected events.

An ART system contains two functionally complementary subsystems that allow it to process familiar and unfamiliar events: *attentional subsystem* and *orienting subsystem*. Familiar events are processed within the attentional subsystem. This goal of this subsystem is to constantly establish even more precise internal representations of and responses to familiar events. By itself, however, the attentional subsystem is unable to simultaneously maintain stable representations of familiar categories and to create new categories for unfamiliar events. This is where the orienting subsystem helps. It is used to reset the attentional subsystem when an unfamiliar event occurs. The orienting subsystem is essential for expressing whether a novel pattern is familiar and well

represented by an existing recognition code, or unfamiliar and in need of a new recognition code.

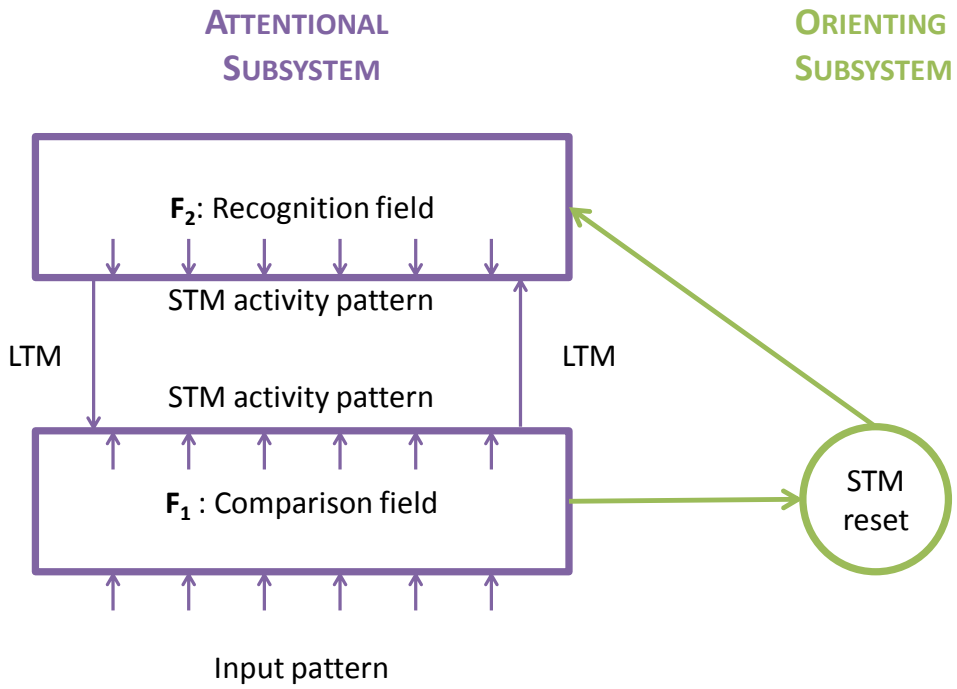


Figure 18: The architecture of an ART system has two subsystems: attentional, responsible for processing familiar events, and orienting, which helps reset the attentional subsystem when an unfamiliar event occurs. The attentional subsystem contains a comparison field, where the input is received, and a recognition field, which assigns the input to a category. Both short term memory (STM) and long term memory (LTM) are employed.

Figure 18 shows the architecture of an ART system. The attentional system has two successive stages, F_1 and F_2, which encode patterns of activation in short term memory (STM). The input pattern is received at F_1, and the classification is performed at F_2. Bottom-up and top-down pathways between the two stages contain adaptive long term memory (LTM) traces. The orienting subsystem measures the similarity between the input instance vector and the pattern

produced by the fields in the attentional subsystem. If the two are similar, i.e. if the attentional subsystem has been able to recognize the input instance, the orienting subsystem does not interfere. However, if the two patterns are significantly different, the orienting subsystem resets the output of the recognition layer. The effect of the reset is to force the output of the attentional system back to zero, which allows the system to search for a better match.

A drawback of some of the ART architectures is that the results of the models depend significantly on the order in which the training instances are processed. The effect can be reduced to some extent by using a slower learning rate, where differential equations are used and the degree of training on an input depends on the time the input is available. However, even with slow training, the order of training still affects the system regardless of the size of the input dataset.

ART classifiers have been applied to WSN applications to address **anomaly detection problems** in unknown environments (Li, Thomason and Parker, Detecting Time-Related Changes in Wireless Sensor Networks Using Symbol Compression and Probabilistic Suffix Trees 2010). A fuzzy ART classifier is used to label multi-dimensional sensor data into discrete classes and detect sensor-level anomalies. An ART classification is also employed by an **intruder detection** system that uses a WSN and mobile robots (Li and Parker, Intruder detection using a wireless sensor network with an intelligent mobile robot response 2008). The sensor network uses an unsupervised fuzzy ART classifier to learn and detect intruders in a previously unknown

environment. Upon the detection of an intruder, a mobile robot travels to investigate the position where the intruder is supposed to be. Kulakov et al. incorporate ART into a technique used for detection of unusual sensor events and sensor failures (Kulakov and Davcev 2005). Through simulation, where one of the input sensor nodes is failed on purpose, the authors show the improvement in data robustness achieved by their approach.

Other unsupervised machine learning algorithms

There is a wide variety of unsupervised learning algorithms, in addition to k-means clustering, DBSCAN, SOM, and ART, which have been often applied to WSN application. The SmartHouse project uses a system of sensors to monitor a person's activities at home (Barger, Brown and Alwan 2005). The goal of the project is to recognize and detect different behavioral patterns. The authors use **mixture models** to develop a probabilistic model of the behavioral patterns. The mixture model approach serves to cluster the observations with each cluster considered to be a different event type.

A number of activity recognition projects have developed unsupervised learning algorithms that **extract models from text corpora or the web**. The Guide project uses unsupervised learning methods to detect activities using RFID tags placed on objects (Philipose, et al. 2003). This method relies on data mining techniques to extract activity models from the web in an unsupervised fashion. For this project the authors have mined the temporal structure of about fifteen thousand home activities.

Gu et al. develop another unsupervised approach based on RFID-tagged object-use fingerprints

to recognize activities without human labeling (Gu, et al. 2010). The activity models they use are built based on object-use fingerprints, which are sets of contrast patterns describing significant differences in object-use between any two activity classes. This is done by first mining a set of object terms for each activity class from the web, and then mining contrast patterns among object terms based on emerging patterns to distinguish between any two activity patterns.

Wyat et al. also employ generic mined models from the web (Wyatt, Philipose and Choudhury 2005). Given an unlabeled trace of object names from a user performing their activities of daily living, they use the generic mined models to segment the trace into labeled instances of activities. After that they use the labeled instances to learn custom models of the activity from the data. For example, they learn details such as order of object use, duration of use, and whether additional object are used.

Tapia et al. develop a similar approach where they extract relevant information on the functional similarity of objects automatically from WordNet, which is an online lexical reference system for the English language (Tapia, Choudhury and Philipose 2006). The information about the functional similarity between objects is represented in a hierarchical form known as *ontology*. This ontology is used to help mitigate the problem of model incompleteness, which often affects the techniques used to construct activity recognition models.

An unsupervised approach based on detecting and analyzing the sequence of objects that are being used by the residents is described in (Wu, et al. 2007). The activity recognition method is

based on RFID object-use correlated with video streams, and information collected from how-to websites such as about.com. Since video streams are used, the approach provides high-grained activity recognition. For example, it can differentiate between making tea and making coffee. However, as previously mentioned, collecting video data of home activities is difficult due to privacy concerns.

Dimitrov et al. develop a system that relies on unsupervised recognition to identify activities of daily living in a smart home environment (Dimitrov, Pauli and Naroska 2010). The system utilizes **background domain knowledge** about the user activities, which is stored in a self-updating probabilistic knowledge base. The system aims to build the best possible explanation for the observed stream of sensor events.

Semi-Supervised Learning

Semi-supervised learning algorithms use both labeled and unlabeled data for training. The labeled data is typically a small percentage of the training dataset. The goal of semi-supervised learning is to 1) understand how combining labeled and unlabeled data may change the learning behavior, and 2) design algorithms that take advantage of such a combination. Semi-supervised learning is a very promising approach since it can use readily available unlabeled data to improve supervised learning tasks when the labeled data is scarce or expensive.

There are many different semi-supervised learning algorithms. Some of the most commonly used

ones include:

- **Expectation-Maximization with generative mixture models**

Expectation-maximization (EM) is an iterative method for finding maximum likelihood estimates of parameters in statistical models, where the models depend on unobserved latent variables (Dempster, Laird and Rubin 1977). Each iteration of the algorithm consists of an expectation step (e-step) followed by a maximization step (m-step). EM with generative mixture models are suitable for applications where the classes specified by the application produce well clustered data.

- **Self-training**

Self-training can refer to a variety of schemes for using unlabeled data. Ng and Cardie implement self-training by *bagging* and majority voting (Ng and Cardie 2003). An ensemble of classifiers is trained on the labeled data instances and then the classifiers are used to classify the unlabeled examples independently. Only those examples, for which all classifiers assign the same label, are added to the labeled training set, and the classifier ensemble is retrained. The process continues until a stop condition is met.

A single classifier can also be self-trained. Similarly to the ensemble of classifiers, the single classifier is first trained on all labeled data. Then the classifier is applied to the unlabeled instances. Only those instances that meet a selection criterion are added to the labeled set and

used for retraining.

- **Co-training**

Co-training requires two or more *views* of the data, i.e. disjoint feature sets that provide different complementary information about the instances (Blum and Mitchell, Combining Labeled and Unlabeld Data with Co-Training 1998). Ideally, the two feature sets for each instance are conditionally independent. Also each feature set should be sufficient to accurately assign each instance to its respective class. The first step in co-training is to use all labeled data and train a separate classifier for each view. Then, the most confident predictions of each classifier are used on the unlabeled data to construct additional labeled training instances. Co-training is a suitable algorithm to use if the features of the dataset naturally split into two sets.

- **Transductive support vector machines**

Transductive SVMs extend general SVMs in that they could also use partially labeled data for semi-supervised learning by following the principles of *transduction* (Gammerman, Vovk and Vapnik 1998). In inductive learning, the algorithm is trained on specific training instances but the goal is to learn general rules, which are then applied to the test cases. In contrast, transductive learning is reasoning from specific training cases to specific testing cases.

- **Graph-based methods**

These are algorithms that utilize the graph structure obtained by capturing pairwise similarities

between the labeled and unlabeled instances (Zhu 2007). These algorithms define a graph structure where the nodes are labeled and unlabeled instances and the edges, which may be weighted, represent the similarity of the nodes they connect.

In sensor networks, semi-supervised learning has been applied to **localization** of mobile objects. Pan et al. develop a probabilistic semi-supervised learning approach to reduce the calibration effort and increase the tracking accuracy of their system (Pan, et al. 2007). Their method is based on semi-supervised CRFs, which effectively enhance the learned model from a small set of training data with abundant unlabeled data. To make the method more efficient, the authors employ a Generalized EM algorithm coupled with domain constraints. Yang et al. use a semi-supervised manifold learning algorithm to estimate the locations of mobile nodes in a WSN (Yang, et al. 2010). The algorithm is used to compute a subspace mapping function between the signal space and the physical space by using a small amount of labeled data and a large amount of unlabeled data.

Wang et al. develop a semi-supervised learning algorithm based on SVM (Wang, et al. 2007). The algorithm has been applied to target classification and the experimental results show that it can accurately classify targets in sensor networks.

STATISTICAL LEARNING METHOD

STATISTICAL LEARNING

Let us consider a *very* simple example. Our favorite Surprise candy comes in two flavors: cherry

(yum) and lime (ugh). The candy manufacturer has a peculiar sense of humor and wraps each piece of candy in the same opaque wrapper, regardless of flavor. The candy is sold in very large bags, of which there are known to be five kinds—again, indistinguishable from the outsid

h_1: 100% cherry

h_2: 75% cherry + 25% lime h_3: 50% cherry + 50% lime h_4: 25% cherry + 75% lime h_5: 100% lime

Given a new bag of candy, the random variable H (for *hypothesis*) denotes the type of the bag, with possible values h_1 through h_5. H is not directly observable, of course. As the pieces of candy are opened and inspected, data are revealed—D_1, D_2, : : :, D_N , where each D_i is a random variable with possible values **cherry** and **lime**. The basic task faced by the agent is to predict the flavor of the next piece of candy.[1] Despite its apparent triviality, this scenario serves to introduce many of the major issues. The agent really does need to infer a theory of its world, albeit a very simple one.

Bayesian learning simply calculates the probability of each hypothesis, given the data, and makes predictions on that basis. That is, the predictions are made by using *all* the hypotheses, weighted by their probabilities, rather than by using just a single "best" hypothesis. In this way, learning is reduced to probabilistic inference. Let **D** represent all the data, with observed value **d**; then the probability of each hypothesis is obtained by Bayes' rule:

$P(h_i|\mathbf{d}) = P(\mathbf{d}|h_i)P(h_i)$:

Now, suppose we want to make a prediction about an unknown quantity X . Then we have

$$\mathbf{P}(X|\mathbf{d}) = \sum_i \mathbf{P}(X|\mathbf{d}, h_i)\mathbf{P}(h_i|\mathbf{d}) = \sum_i \mathbf{P}(X|h_i)P(h_i|\mathbf{d}) ;$$

where we have assumed that each hypothesis determines a probability distribution over X .

This equation shows that predictions are weighted averages over the predictions of the indi-

vidual hypotheses. The hypotheses themselves are essentially “intermediaries” between the

raw data and the predictions. The key quantities in the Bayesian approach are the **hypothesis**

prior, $P(h_i)$, and the **likelihood** of the data under each hypothesis, $P(\mathbf{d}|h_i)$.

For our candy example, we will assume for the time being that the prior distribution over $h_1; : : : ; h_5$ is given by h0:1; 0:2; 0:4; 0:2; 0:1i, as advertised by the manufacturer. The

likelihood of the data is calculated under the assumption that the observations are

i.i.d.—that

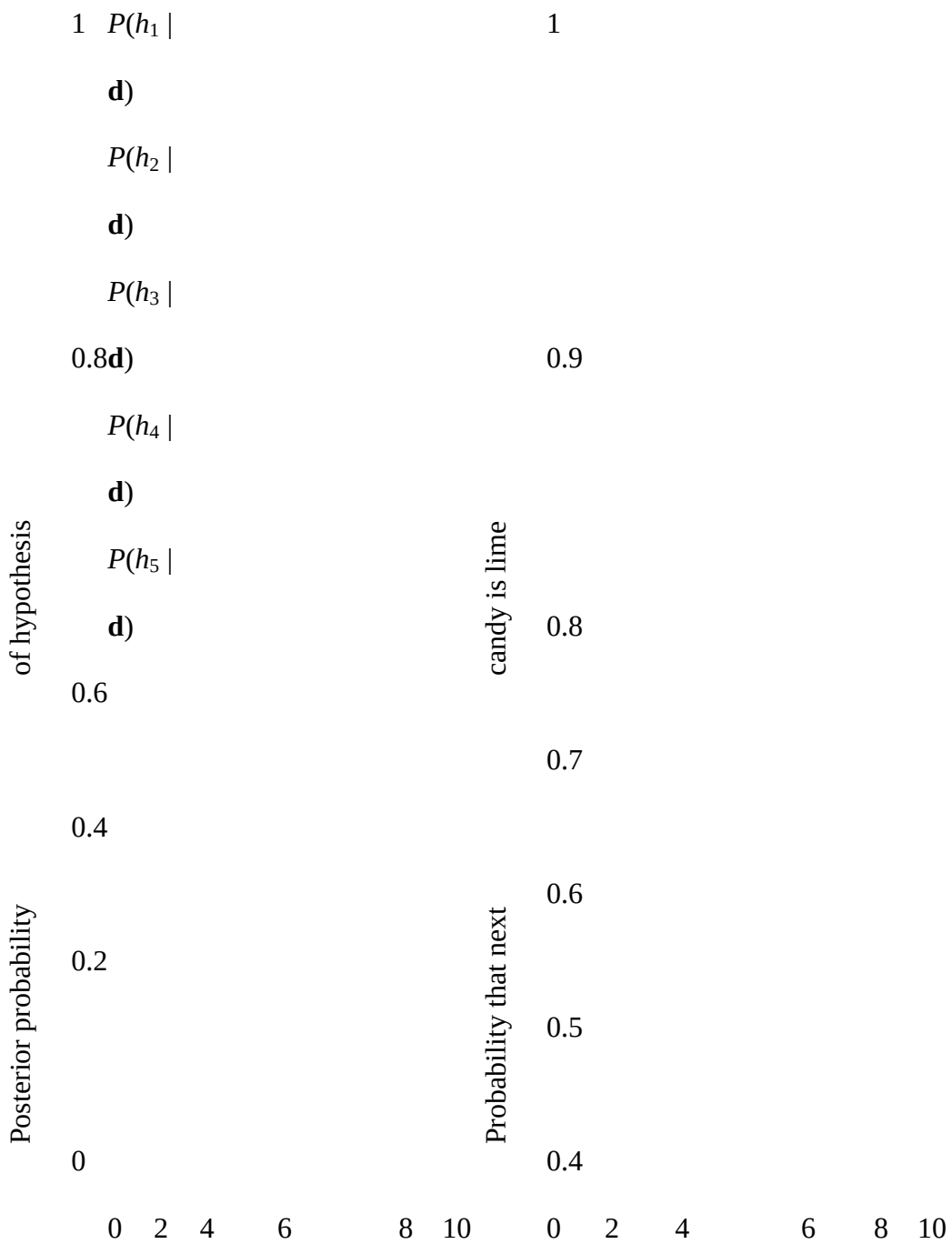
is, independently and identically distributed—so that

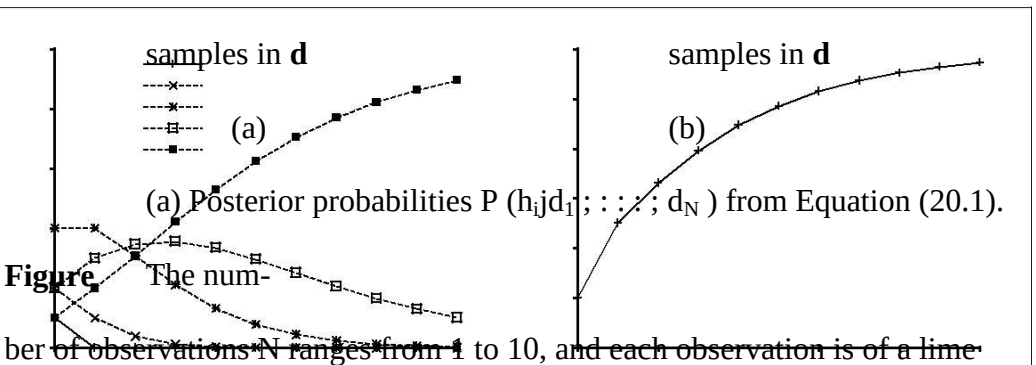
P (**d**jh_i) = P (d_j jh_i) :

j

Y

For example, suppose the bag is really an all-lime bag (h_5) and the first 10 candies are all lime; then P (**d**jh_3) is $0:5^{10}$, because half the candies in an h_3 bag are lime.[2] Figure 20.1(a) shows how the posterior probabilities of the five hypotheses change as the sequence of 10 lime candies is observed. Notice that the probabilities start out at their prior values, so h_3 is initially the most likely choice and remains so after 1 lime candy is unwrapped. After 2


Posterior probability of hypothesis
1
0.8
0.6
0.4
0.2
0
0 2 4 6 8 10
P(h1 | d)
P(h2 | d)
P(h3 | d)
P(h4 | d)
P(h5 | d)
Probability that next candy is lime
1
0.9
0.8
0.7
0.6
0.5
0.4
0 2 4 6 8 10



(a) Posterior probabilities P ($h_i|d_1$; : : : ; d_N) from Equation (20.1).

Figure The number of observations N ranges from 1 to 10, and each observation is of a lime candy. (b) Bayesian prediction P (d_{N+1} = **lime** $|d_1$; : : : ; d_N) from Equation (20.2).

lime candies are unwrapped, h_4 is most likely; after 3 or more, h_5 (the dreaded all-lime bag) is the most likely. After 10 in a row, we are fairly certain of our fate. Figure 20.1(b) shows the predicted probability that the next candy is lime, based on Equation (20.2). As we would expect, it increases monotonically toward 1.

The example shows that *the true hypothesis eventually dominates the Bayesian predic-tion.* This is characteristic of Bayesian learning. For any fixed prior that does not rule out the true hypothesis, the posterior probability of any false hypothesis will eventually vanish, sim-ply because the probability of generating "uncharacteristic" data indefinitely is vanishingly small. (This point is analogous to one made in the discussion of PAC learning in Chapter 18.) More importantly, the Bayesian prediction is *optimal*, whether the data set be small or large. Given the hypothesis prior, any other prediction will be correct less often.

The optimality of Bayesian learning comes at a price, of course. For real learning problems, the hypothesis space is usually very large or infinite, as we saw in Chapter 18. In some cases, the summation in Equation (20.2) (or integration, in the continuous case) can be carried out tractably, but in most cases we must resort to approximate or simplified methods.

A very common approximation—one that is usually adopted in science—is to make pre-dictions based on a single *most probable* hypothesis—that is, an h_i that maximizes $P(h_i|\mathbf{d})$. This is often

called a **maximum a posteriori** or MAP (pronounced "em-ay-pee") hypothe-sis. Predictions made according to an MAP hypothesis h_{MAP} are approximately Bayesian to the extent that **P**(X j**d**) **P**(X jh_{MAP}). In our candy example, $h_{MAP} = h_5$ after three lime candies in a row, so the MAP learner then predicts that the fourth candy is lime with prob-ability 1.0—a much more dangerous prediction than the Bayesian prediction of 0.8 shown in Figure 20.1. As more data arrive, the MAP and Bayesian predictions become closer, be-cause the competitors to the MAP hypothesis become less and less probable. Although our example doesn't show it, finding MAP hypotheses is often much easier than Bayesian learning, because it requires solving an optimization problem instead of a large summation (or integration) problem. We will see examples of this later in the chapter.

In both Bayesian learning and MAP learning, the hypothesis prior P (h_i) plays an important role. We saw in Chapter 18 that **over fitting** can occur when the hypothesis space is too expressive, so that it contains many hypotheses that fit the data set well. Rather than placing an arbitrary limit on the hypotheses to be considered, Bayesian and MAP learning methods use the prior to *penalize complexity*. Typically, more complex hypotheses have a lower prior probability—in part because there are usually many more complex hypotheses than simple hypotheses. On the other hand, more complex hypotheses have a greater capacity to fit the data. (In the extreme case, a lookup table can reproduce the data exactly with probability 1.) Hence, the hypothesis prior embodies a trade-off between the complexity of a hypothesis and its degree of fit to the data.

We can see the effect of this trade-off most clearly in the logical case, where H contains only *deterministic* hypotheses. In that case, P (**d**jh_i) is 1 if h_i is consistent and 0 otherwise. Looking at

Equation (20.1), we see that h_{MAP} will then be the *simplest logical theory that is consistent with the data.* Therefore, maximum *a posteriori* learning provides a natural embodiment of Ockham's razor.Another insight into the trade-off between complexity and degree of fit is obtained by taking the logarithm of Equation (20.1). Choosing h_{MAP} to maximize P (**d**jh_i)P (h_i) is equivalent to minimizing

$\log_2$ P (**d**jh_i) $\log_2$ P (h_i) :

Using the connection between information encoding and probability that we introduced in Chapter 18, we see that the $\log_2$ P (h_i) term equals the number of bits required to specify the hypothesis h_i. Furthermore, $\log_2$ P (**d**jh_i) is the additional number of bits required to specify the data, given the hypothesis. (To see this, consider that no bits are required if the hypothesis predicts the data exactly—as with h_5 and the string of lime candies—and $\log_2$ 1 = 0.) Hence, MAP learning is choosing the hypothesis that provides maximum *com-pression* of the data. The same task is addressed more directly by the **minimum description length**, or MDL, learning method, which attempts to minimize the size of hypothesis and data encodings rather than work with probabilities.

A final simplification is provided by assuming a **uniform** prior over the space of hy-potheses. In that case, MAP learning reduces to choosing an h_i that maximizes P (**d**jH_i). This is called a **maximum-likelihood** (ML) hypothesis, h_{ML}. Maximum-likelihood learning is very common in statistics, a discipline in which many researchers distrust the subjective nature of hypothesis priors. It is a reasonable approach when there is no reason to prefer one hypothesis over another *a priori*—for example, when all hypotheses are equally complex. It provides a good

approximation to Bayesian and MAP learning when the data set is large, because the data swamps the prior distribution over hypotheses, but it has problems (as we shall see) with small data sets.

LEARNING WITH COMPLETE DATA

Our development of statistical learning methods begins with the simplest task: **parameter learning** with **complete data**. A parameter learning task involves finding the numerical parameters for a probability model whose structure is fixed. For example, we might be interested in learning the conditional probabilities in a Bayesian network with a given structure. Data are complete when each data point contains values for every variable in the probability model being learned. Complete data greatly simplify the problem of learning the parameters of a complex model. We will also look briefly at the problem of learning structure.

Maximum-likelihood parameter learning: Discrete models

Suppose we buy a bag of lime and cherry candy from a new manufacturer whose lime–cherry proportions are completely unknown—that is, the fraction could be anywhere between 0 and 1. In that case, we have a continuum of hypotheses. The **parameter** in this case, which we call , is the proportion of cherry candies, and the hypothesis is h . (The proportion of limes is just 1 .) If we assume that all proportions are equally likely *a priori*, then a maximum-likelihood approach

is reasonable. If we model the situation with a Bayesian network, we need just one random variable, **Flavor** (the flavor of a randomly chosen candy from the bag). It has values **cherry** and **lime**, where the probability of **cherry** is (see Figure 20.2(a)). Now suppose we unwrap N candies, of which c are cherries and ` = N c are limes. According to Equation (20.3), the likelihood of this particular data set is

N

P (**d**jh) = $\prod_{\mathbf{j}=1}$ P (d$_j$ jh) = ^{c} (1)$^{`}$:

The maximum-likelihood hypothesis is given by the value of that maximizes this expres- sion. The same value is obtained by maximizing the **log likelihood**,

N

X

L(**d**jh) = log P (**d**jh) = log P (d$_j$ jh) = c log + ` log(1) :

j = 1

(By taking logarithms, we reduce the product to a sum over the data, which is usually easier to maximize.) To find the maximum-likelihood value of , we differentiate L with respect to and set the resulting expression to zero:

$$\frac{dL(\mathbf{d}jh)}{d} = c\left[\frac{`}{1} = 0\right] = c\left[\frac{}{c + `N}\right] = c\left[\frac{}{}\right] :$$

In English, then, the maximum-likelihood hypothesis h_{ML} asserts that the actual proportion of cherries in the bag is equal to the observed proportion in the candies unwrapped so far!

It appears that we have done a lot of work to discover the obvious. In fact, though, we have laid out one standard method for maximum-likelihood parameter learning:

1. Write down an expression for the likelihood of the data as a function of the parameter(s).

2. Write down the derivative of the log likelihood with respect to each parameter.

3. Find the parameter values such that the derivatives are zero.

P(*F=che*

rry)

□

P(*F=che*

rry) *Flavor*

□

F *P*(*W=red* | *F*)

Flavor *cherry* □1

lime □2

Wrapper

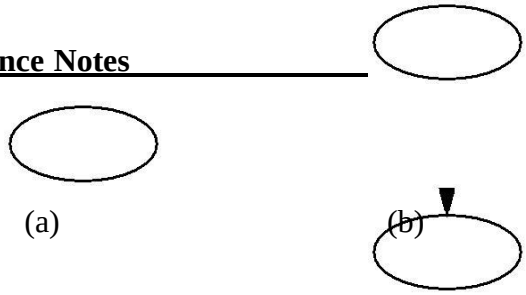


Figure 20.2 (a) Bayesian network model for the case of candies with an unknown proportion of cherries and limes.

(b) Model for the case where the wrapper color depends (probabilistically) on the candy flavor.

The trickiest step is usually the last. In our example, it was trivial, but we will see that in many cases we need to resort to iterative solution algorithms or other numerical optimization techniques, as described in Chapter 4. The example also illustrates a significant problem with maximum-likelihood learning in general: *when the data set is small enough that some events have not yet been observed—for instance, no cherry candies—the maximum likelihood hypothesis assigns zero probability to those events.* Various tricks are used to avoid this problem, such as initializing the counts for each event to 1 instead of zero.

Let us look at another example. Suppose this new candy manufacturer wants to give a little hint to the consumer and uses candy wrappers colored red and green. The **Wrapper** for each candy is selected *probabilistically*, according to some unknown conditional distribution, depending on the flavor. The corresponding probability model is shown in Figure 20.2(b). Notice that it has three parameters: , $_1$, and $_2$. With these parameters, the likelihood of seeing, say, a cherry candy in a green wrapper can be obtained from the standard semantics for Bayesian networks (page 495):

P (**Flavor** = **cherry** ; **Wrapper** = **green**jh ; $_1$; $_2$)

= P (**Flavor** = **cherry** jh ; $_1$; $_2$)P (**Wrapper** = **green**j**Flavor** = **cherry** ; h ; $_1$; $_2$)

= (1 $_1$) :

Now, we unwrap N candies, of which c are cherries and ` are limes. The wrapper counts are as follows: r_c of the cherries have red wrappers and g_c have green, while r‘ of the limes have red and g‘ have green. The likelihood of the data is given by

P (**d**jh ; $_1$; $_2$) = ^{c}(1) ` $_1^{rc}$ (1 $_1$)$^{gc}$ $_2^{r`}$ (1 $_2$)$^{g`}$:

This looks pretty horrible, but taking logarithms helps:

L = [c log + ` log(1)] + [r_c log $_1$ + g_c log(1 $_1$)] + [r‘ log $_2$ + g‘ log(1 $_2$)] :

The benefit of taking logs is clear: the log likelihood is the sum of three terms, each of which contains a single parameter. When we take derivatives with respect to each parameter and set

them to zero, we get three independent equations, each containing just one parameter:

@ L ` c = c 1 = 0 c+

@ rc gc) = `

@ = 1 11 = 0 rc

L r` g` r`c

@ = 2 12 = 0) 1 = +gc

1 r`

@ r`

L) 2 = +g` :

@

2

The solution for is the same as before. The solution for $_1$, the probability that a cherry candy has a red wrapper, is the observed fraction of cherry candies with red wrappers, and similarly for $_2$.

These results are very comforting, and it is easy to see that they can be extended to any Bayesian network whose conditional probabilities are represented as tables.

The most impor-tant point is that, *with complete data, the maximum-likelihood parameter learning problem for a Bayesian network decomposes into separate learning problems, one for each parame-ter.*[3] The second point is that the parameter values for a variable, given its parents, are just the observed frequencies of the variable values for each setting of the parent values. As before, we must be careful to avoid zeroes when the data set is small.

Naive Bayes models

Probably the most common Bayesian network model used in machine learning is the **naive Bayes** model. In this model, the "class" variable C (which is to be predicted) is the root and the "attribute" variables X_i are the leaves. The model is "naive" because it assumes that the attributes are conditionally independent of each other, given the class. (The model in Figure 20.2(b) is a naive Bayes model with just one attribute.) Assuming Boolean variables, the parameters are

= P (C = **true**); $_{i1}$ = P (X_i = **true**jC = **true**); $_{i2}$ = P (X_i = **true**jC = **false**):

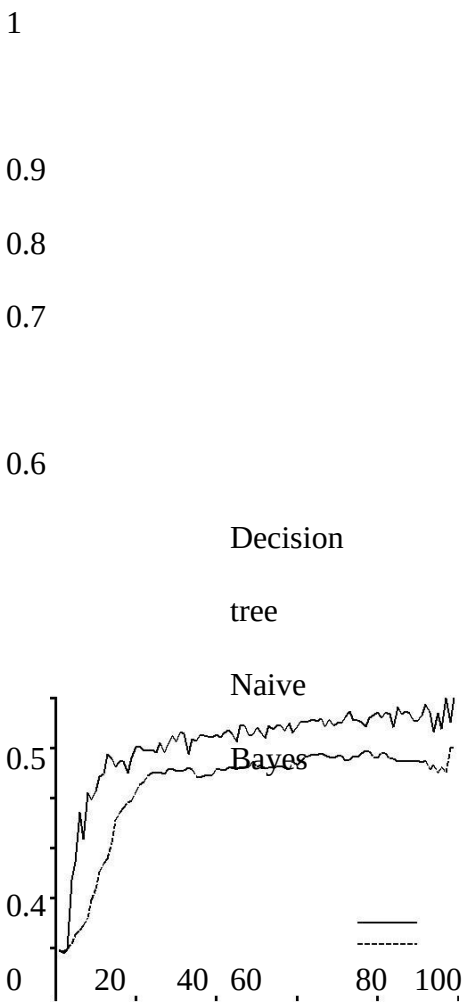
The maximum-likelihood parameter values are found in exactly the same way as for Fig-ure 20.2(b). Once the model has been trained in this way, it can be used to

classify new exam-ples for which the class variable C is unobserved. With observed attribute values $x_1; : : : ; x_n$, the probability of each class is given by

$$\mathbf{P}(Cjx_1; : : : ; x_n) = \mathbf{P}(C) \quad \mathbf{P}(x_ijC) : \mathbf{i}$$

A deterministic prediction can be obtained by choosing the most likely class. Figure 20.3 shows the learning curve for this method when it is applied to the restaurant problem from Chapter 18. The method learns fairly well but not as well as decision-tree learning; this is presumably because the true hypothesis—which is a decision tree—is not representable ex-actly using a naive Bayes model. Naive Bayes learning turns out to do surprisingly well in a wide range of applications; the boosted version (Exercise 20.5) is one of the most effective general-purpose learning algorithms. Naive Bayes learning scales well to very large problems: with n Boolean attributes, there are just 2n + 1 parameters, and *no search is required to find* h_{ML}*, the maximum-likelihood naive Bayes hypothesis.* Finally, naive Bayes learning has no difficulty with noisy data and can give probabilistic predictions when appropriate.

Learning with Complete Data



Training set size

Figure 20.3 The learning curve for naive Bayes learning applied to the restaurant problem from Chapter 18; the learning curve for decision-tree learning is shown for comparison.

Maximum-likelihood parameter learning: Continuous models

Continuous probability models such as the **linear-Gaussian** model were introduced in Sec-tion 14.3. Because continuous variables are ubiquitous in real-world applications, it is im-portant to know how to learn continuous models from data. The principles for maximum-likelihood learning are identical to those of the discrete case.

Let us begin with a very simple case: learning the parameters of a Gaussian density function on a single variable. That is, the data are generated as follows:

$$P(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{(x)^2}{2}} :$$

The parameters of this model are the mean and the standard deviation . (Notice that the normalizing "constant" depends on, so we cannot ignore it.) Let the observed values be x_1; : : : ; x_N . Then the log likelihood is

$$L = \sum_{j=1}^{N} \log \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x_j - \mu)^2}{2\sigma^2}} = N\left(\log\frac{1}{\sqrt{2\pi}\sigma}\right) - \frac{1}{2\sigma^2}\sum_{j=1}^{N}(x_j - \mu)^2$$

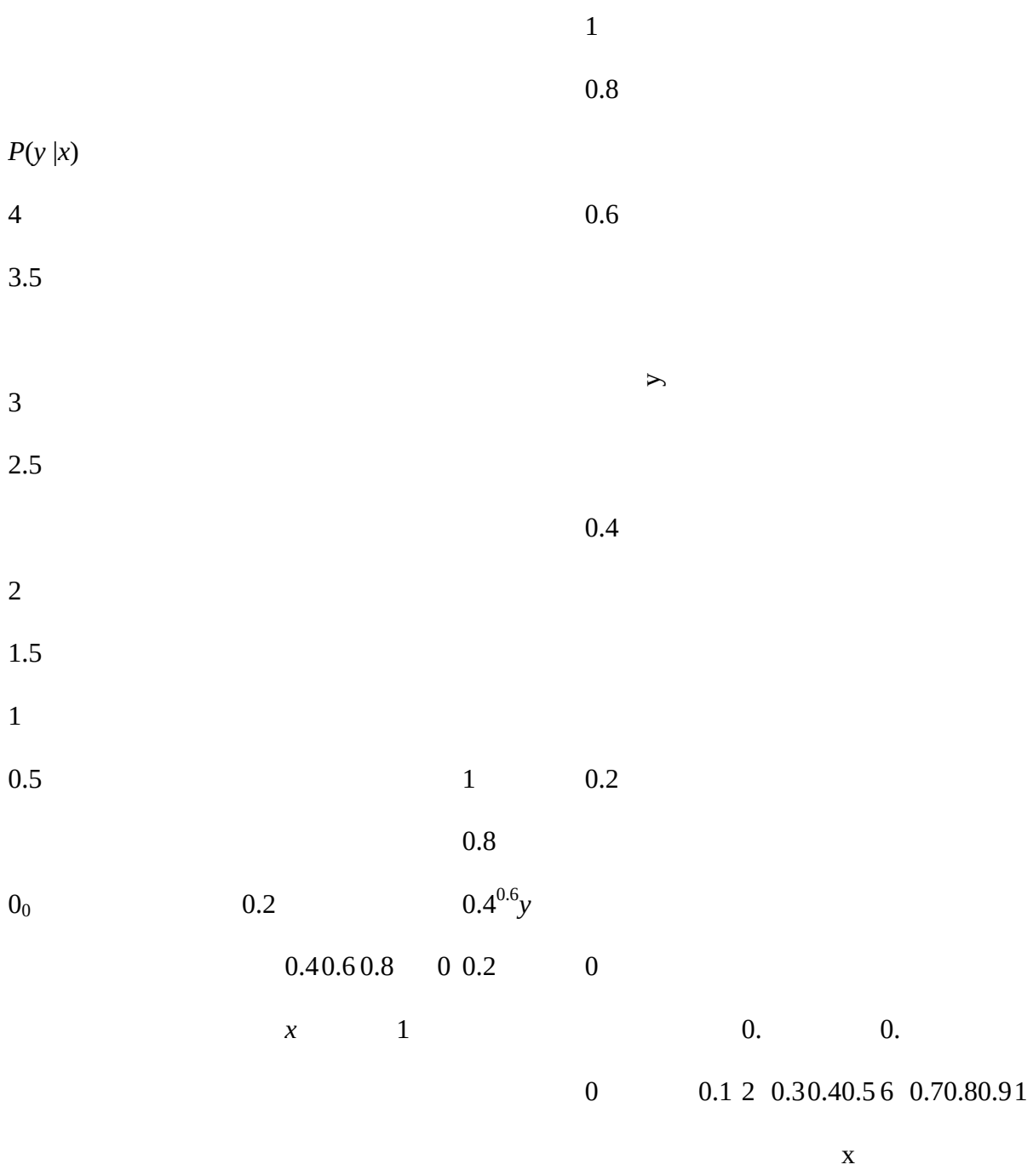
Setting the derivatives to zero as usual, we obtain

$$\frac{\partial L}{\partial \mu} = \frac{1}{\sigma^2}\sum_{j=1}^{N}(x_j - \mu) = 0 \quad \Rightarrow \quad \mu = \frac{\sum_j x_j}{N} \tag{20.4}$$

$$\frac{\partial L}{\partial \sigma} = -\frac{N}{\sigma} + \frac{1}{\sigma^3}\sum_{j=1}^{N}(x_j - \mu)^2 = 0 \quad \Rightarrow \quad \sigma = \sqrt{\frac{\sum_j (x_j - \mu)^2}{N}}$$

That is, the maximum-likelihood value of the mean is the sample average and the maximum-likelihood value of the standard deviation is the square root of the sample variance. Again, these are comforting results that confirm "commonsense" practice.

Now consider a linear Gaussian model with one continuous parent X and a continuous child Y . As explained on page 502, Y has a Gaussian distribution whose mean depends linearly on the value of X and whose standard deviation is fixed. To learn the conditional


1
0.8
P(y |x)
4
0.6
3.5
y
3
2.5
0.4
2
1.5
1
0.5
1
0.2
0.8
0 0.2
0.4 0.6 y
0.4 0.6 0.8
0 0.2
0
x
1
0.
0.
0
0.1 2 0.3 0.4 0.5 6 0.7 0.8 0.9 1
x

(a) (b)

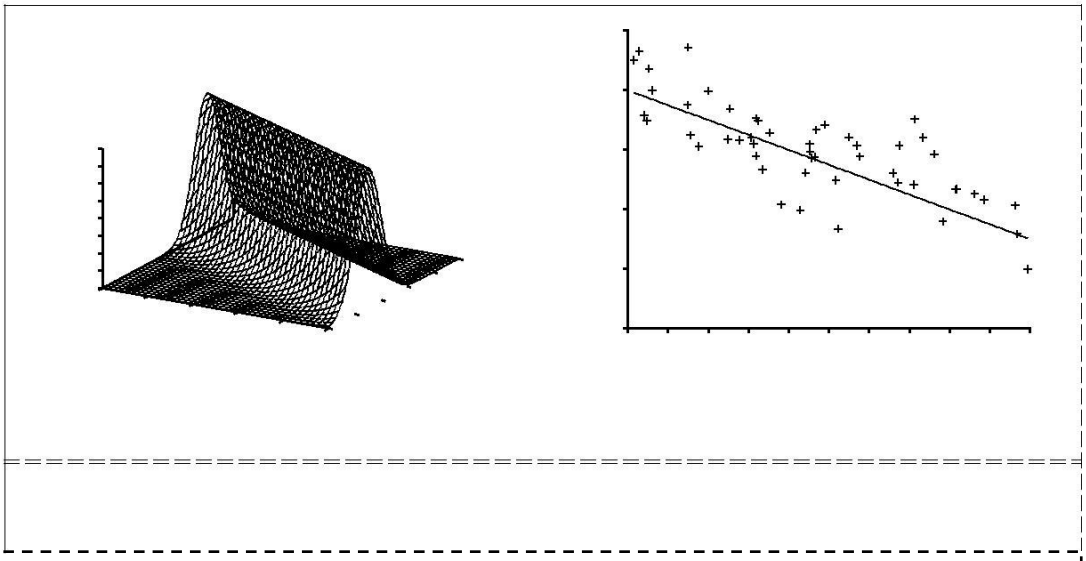


Figure 20.4 (a) A linear Gaussian model described as $y = {}_1 x + {}_2$ plus Gaussian noise with fixed variance. (b) A set of 50 data points generated from this model.

distribution P (Y jX), we can maximize the conditional likelihood

$$P(y|x) = p \frac{1}{2\ ^2} e^{\frac{(y\ (\ {}_1x+\ {}_2))^2}{2\ ^2}} : \quad (20.5)$$

2

Here, the parameters are $_1$, $_2$, and . The data are a collection of $(x_j ; y_j)$ pairs, as illustrated in Figure 20.4. Using the usual methods (Exercise 20.6), we can find the maximum-likelihood values of the parameters. Here, we want to make a different point. If we consider just the

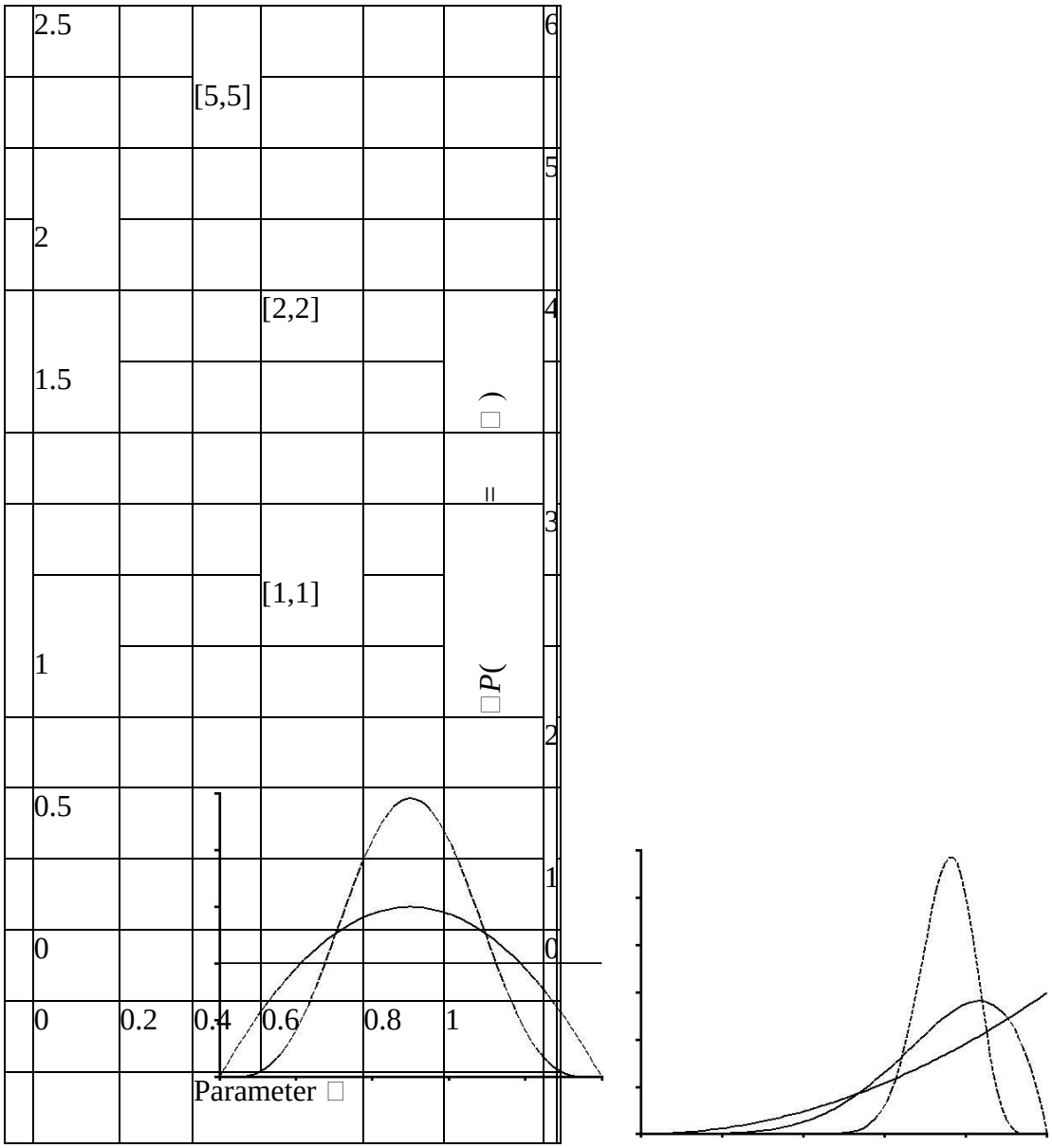
parameters $_1$ and $_2$ that define the linear relationship between x and y, it becomes clear that maximizing the log likelihood with respect to these parameters is the same as *minimizing* the numerator in the exponent of Equation (20.5):

$$E = \sum_{j=1}^{N} (y_j - (_1 x_j + _2))^2 :$$

The quantity $(y_j - (_1 x_j + _2))$ is the **error** for $(x_j ; y_j)$—that is, the difference between the actual value y_j and the predicted value $(_1 x_j + _2)$—so E is the well-known **sum of squared errors**. This is the quantity that is minimized by the standard **linear regression** procedure. Now we can understand why: minimizing the sum of squared errors gives the maximum-likelihood straight-line model, *provided that the data are generated with Gaussian noise of fixed variance*.

Bayesian parameter learning

Maximum-likelihood learning gives rise to some very simple procedures, but it has some serious deficiencies with small data sets. For example, after seeing one cherry candy, the maximum-likelihood hypothesis is that the bag is 100% cherry (i.e., = 1:0). Unless one's hypothesis prior is that bags must be either all cherry or all lime, this is not a reasonable conclusion. The Bayesian approach to parameter learning places a hypothesis prior over the possible values of the parameters and updates this distribution as data arrive.


2.5
[5,5]
2
1.5
[2,2]
1
[1,1]
0.5
0
0
0.2
0.4
0.6
0.8
1
Parameter
P(

(a) (b)

Figure 20.5 Examples of the beta[a; b] distribution for different values of [a; b].

The candy example in Figure 20.2(a) has one parameter, : the probability that a randomly selected piece of candy is cherry flavored. In the Bayesian view, is the Value of a random variable ; the hypothesis prior is just the prior distribution **P**( ). Thus,

P (=) is the prior probability that the bag has a fraction of cherry candies.

If the parameter can be any value between 0 and 1, then **P**( ) must be a continuous distribution that is nonzero only between 0 and 1 and that integrates to 1. The uniform density

P ( ) = U [0; 1]( ) is one candidate. (See Chapter 13.) It turns out that the uniform density is a member of the family of **beta distributions**. Each beta distribution is defined by two **hyperparameters**[4] a and b such that

beta[a; b]( ) = $^{a\ 1}(1\quad)^{b\ 1}$; (20.6)

for in the range [0; 1]. The normalization constant

depends on a and b. (See Exercise 20.8.)

Figure 20.5 shows what the distribution looks like for various values of a and b.

The mean value of the distribution is a=(a + b), so larger values of a suggest a belief that

is closer to 1 than to 0. Larger values of a + b make the distribution more peaked, suggest-

ing greater certainty about the value of . Thus, the beta family provides a useful range of

possibilities for the hypothesis prior.

Besides its flexibility, the beta family has another wonderful property: if has a prior beta[a; b], then, after a data point is observed, the posterior distribution foris also a beta

distribution. The beta family is called the **conjugate prior** for the family of distributions for

a Boolean variable.[5] Let's see how this works. Suppose we observe a cherry candy; then

$$P(\ jD_1 = \textbf{cherry}\ ) = P(D_1 = \textbf{cherry}\ j\ )P(\ )$$

$$= {}^{0}\ \text{beta}[a; b](\ ) = {}^{0}\ {}^{\mathbf{a}}\ {}^{1}(1\quad)^{\mathbf{b}}\ {}^{1}$$

$$= {}^{0}\ {}^{\mathbf{a}}(1\ )^{\mathbf{b}}\ {}^{1} = \text{beta}[a + 1; b](\ ) :$$

They are called hyper parameters because they parameterize a distribution over , which is itself a parameter. Other conjugate priors include the **Dirichlet** family for the parameters of a discrete multi-valued distribution

and the **Normal–Wish art** family for the parameters of a Gaussian distribution. See Bernardo and Smith (1994).

Thus, after seeing a cherry candy, we simply increment the a parameter to get the posterior; similarly, after seeing a lime candy, we increment the b parameter. Thus, we can view the a and b hyperparameters as **virtual counts**, in the sense that a prior beta[a; b] behaves exactly as if we had started out with a uniform prior beta[1; 1] and seen a 1 actual cherry candies and b 1 actual lime candies.By examining a sequence of beta distributions for increasing values of a and b, keeping the proportions fixed, we can see vividly how the posterior distribution over the parameter changes as data arrive. For example, suppose the actual bag of candy is 75% cherry. Fig-ure 20.5(b) shows the sequence beta[3; 1], beta[6; 2], beta[30; 10]. Clearly, the distribution is converging to a narrow peak around the true value of . For large data sets, then, Bayesian learning (at least in this case) converges to give the same results as maximum-likelihood learning.The network in Figure 20.2(b) has three parameters, , $_1$, and $_2$, where $_1$ is the probability of a red wrapper on a cherry candy and $_2$ is the probability of a red wrapper on a lime candy. The Bayesian hypothesis prior must cover all three parameters—that is, we need to specify **P**(; $_1$; $_2$). Usually, we assume **parameter independence**:

P(; $_1$; $_2$) = **P**( )**P**($_1$)**P**($_2$) :

With this assumption, each parameter can have its own beta distribution that is updated separately as data arrive.

Once we have the idea that unknown parameters can be represented by random variables such as , it is natural to incorporate them into the Bayesian network itself. To do this, we also need to make copies of the variables describing each instance. For example, if we have observed three candies then we need **Flavor** $_1$, **Flavor** $_2$, **Flavor** $_3$ and **Wrapper** $_1$, **Wrapper** $_2$, **Wrapper** $_3$. The parameter variable determines the probability of each **Flavor** $_i$ variable:

P (**Flavor** $_i$ = **cherry** j =) = :

Similarly, the wrapper probabilities depend on $_1$ and $_2$, For example,

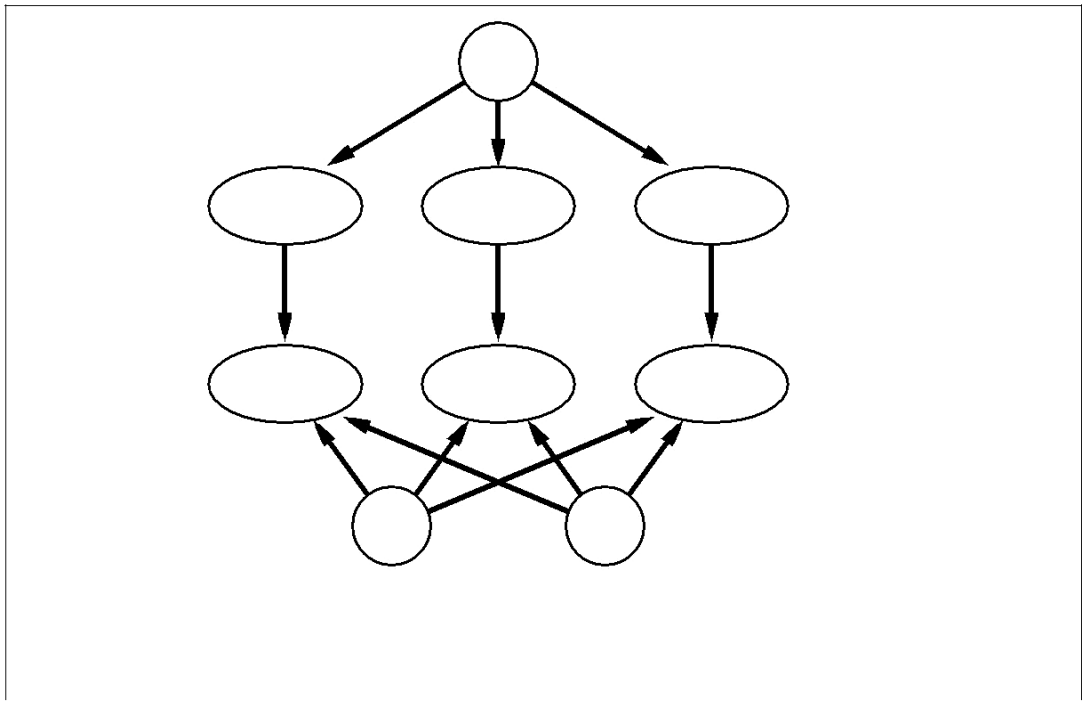
P (**Wrapper** $_i$ = **red** j**Flavor** $_i$ = **cherry**; $_1$ = $_1$) = $_1$:

Now, the entire Bayesian learning process can be formulated as an *inference* problem in a suitably constructed Bayes net, as shown in Figure 20.6. Prediction for a new instance is done simply by adding new instance variables to the network, some of which are queried. This formulation of learning and prediction makes it clear that Bayesian learning requires no extra

"principles of learning." Furthermore, *there is, in essence, just one learning algorithm,* i.e., the inference algorithm for Bayesian networks.

Learning Bayes net structures

So far, we have assumed that the structure of the Bayes net is given and we are just trying to learn the parameters. The structure of the network represents basic causal knowledge about the domain that is often easy for an expert, or even a naive user, to supply. In some cases, however, the causal model may be unavailable or subject to dispute—for example, certain corporations have long claimed that smoking does not cause cancer—so it is important to



$Flavor_1$ $Flavor_2$ $Flavor_3$ . . .

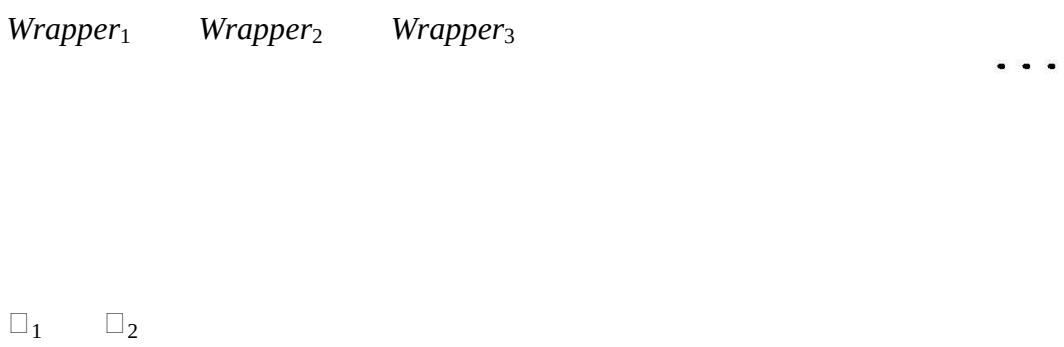


Figure A Bayesian network that corresponds to a Bayesian learning process. Poste-rior distributions for the parameter variables, $_1$, and $_2$ can be inferred from their prior distributions and the evidence in the **Flavor** $_i$ and **Wrapper** $_i$ variables.

understand how the structure of a Bayes net can be learned from data. At present, structural learning algorithms are in their infancy, so we will give only a brief sketch of the main ideas.

The most obvious approach is to *search* for a good model. We can start with a model containing no links and begin adding parents for each node, fitting the parameters with the methods we have just covered and measuring the accuracy of the resulting model. Alternatively, we can start with an initial guess at the structure and use hill-climbing or simulated annealing search to make modifications, retuning the parameters after each change in the structure. Modifications can include reversing, adding, or deleting arcs. We must not introduce cycles in the process, so many

algorithms assume that an ordering is given for the variables, and that a node can have parents only among those nodes that come earlier in the ordering (just as in the construction process Chapter 14). For full generality, we also need to search over possible orderings.

There are two alternative methods for deciding when a good structure has been found. The first is to test whether the conditional independence assertions implicit in the structure are actually satisfied in the data. For example, the use of a naive Bayes model for the restaurant problem assumes that

P(**Fri** =**Sat** ; **Bar** j**WillWait**) = **P**(**Fri** =**Sat** j**WillWait**)**P**(**Bar** j**WillWait**)

and we can check in the data that the same equation holds between the corresponding conditional frequencies. Now, even if the structure describes the true causal nature of the domain, statistical fluctuations in the data set mean that the equation will never be satisfied *exactly*, so we need to perform a suitable statistical test to see if there is sufficient evidence that the independence hypothesis is violated. The complexity of the resulting network will depend

on the threshold used for this test—the stricter the independence test, the more links will be added and the greater the danger of over fitting.

An approach more consistent with the ideas in this chapter is to the degree to which the proposed model explains the data (in a probabilistic sense). We must be careful how we measure this, however. If we just try to find the maximum-likelihood hypothesis, we will end up with a fully connected network, because adding more parents to a node cannot decrease the likelihood (Exercise 20.9). We are forced to penalize model complexity in some way. The MAP (or MDL) approach simply subtracts a penalty from the likelihood of each structure (after parameter tuning) before comparing different structures. The Bayesian approach places a joint prior over structures and parameters. There are usually far too many structures to sum over (super exponential in the number of variables), so most practitioners use MCMC to sample over structures.

Penalizing complexity (whether by MAP or Bayesian methods) introduces an important connection between the optimal structure and the nature of the representation for the conditionals distributions in the network. With tabular distributions, the complexity penalty for a node's distribution grows exponentially with the number of parents, but with, say, noisy-OR distributions, it grows only linearly. This means that learning with noisy-OR (or other com-pactly parameterized) models tends to produce learned structures with more parents than does

learning with tabular distributions.

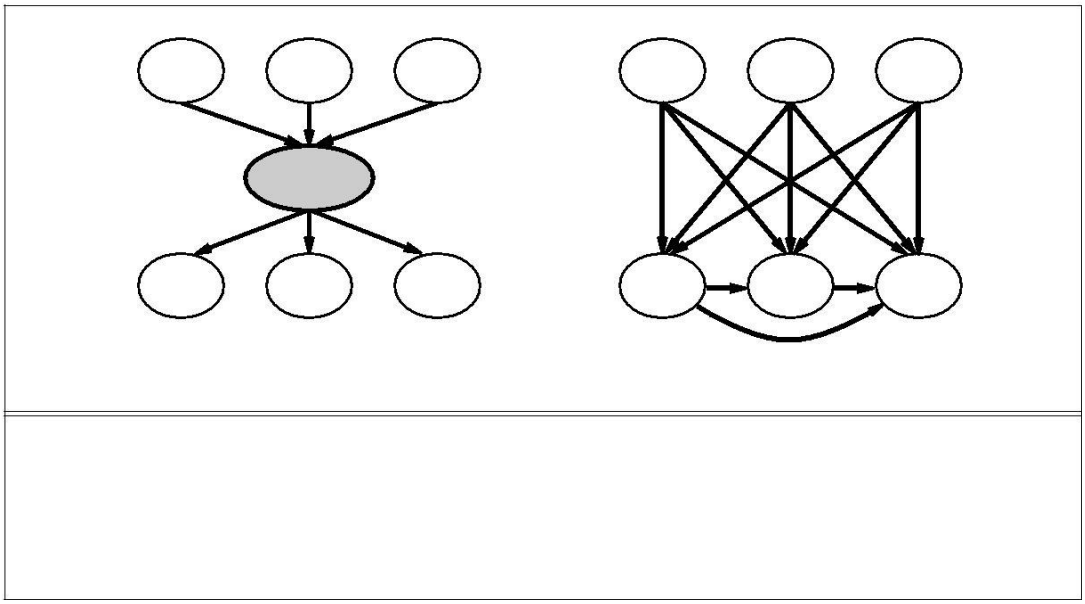
LEARNING WITH HIDDEN VARIABLES: THE EM ALGORITHM

The preceding section dealt with the fully observable case. Many real-world problems have

LATENT VARIABLES **hidden variables** (sometimes called **latent variables**) which are not observable in the data that are available for learning. For example, medical records often include the observed symptoms, the treatment applied, and perhaps the outcome of the treatment, but they seldom contain a direct observation of the disease itself![6] One might ask, "If the disease is not observed, why not construct a model without it?" The answer appears in Figure 20.7, which shows a small, fictitious diagnostic model for heart disease. There are three observable predisposing factors and three observable symptoms (which are too depressing to name). Assume that each variable has three possible values (e.g., **none**, **moderate**, and **severe**). Removing the hidden variable from the network in (a) yields the network in (b); the total number of parameters increases from 78 to 708. Thus, *latent variables can dramatically reduce the number of parameters required to specify a Bayesian network.* This, in turn, can dramatically reduce the amount of data needed to learn the parameters.

Hidden variables are important, but they do complicate the learning problem. In Figure 20.7(a), for example, it is not obvious how to learn the conditional distribution for **Heart Disease**, given its parents, because we do not know the value of **Heart Disease** in each case; the same problem

arises in learning the distributions for the symptoms. This section Learning with Hidden Variables: The EM Algorithm



N

2	2	2	2	2	2
Smoking	*Diet*	*Exercise*	*Smoking*	*Diet*	*Exercise*
54	*HeartDisease*				

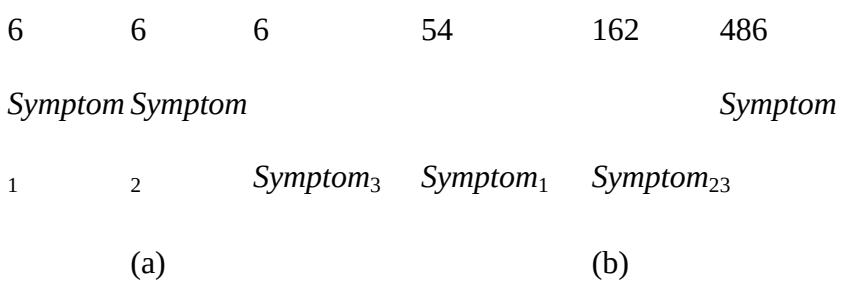


Figure (a) A simple diagnostic network for heart disease, which is assumed to be a hidden variable. Each variable has three possible values and is labeled with the number of independent parameters in its conditional distribution; the total number is 78. (b) The equivalent network with **Heart Disease** removed. Note that the symptom variables are no longer conditionally independent given their parents. This network requires 708 parameters describes an algorithm called **expectation–maximization**, or EM, that solves this problem in a very general way. We will show three examples and then provide a general description. The algorithm seems like magic at first, but once the intuition has been developed, one can find applications for EM in a huge range of learning problems.

Unsupervised clustering: Learning mixtures of Gaussians

Unsupervised clustering is the problem of discerning multiple categories in a collection of objects. The problem is unsupervised because the category labels are not given. For example, suppose we record the spectra of a hundred thousand stars; are there different *types* of stars revealed by the spectra, and, if so, how many and what are their characteristics? We are all familiar with terms such as "red giant" and "white dwarf," but the stars do not carry these labels on their hats—astronomers had to perform unsupervised clustering to identify these categories. Other examples include the identification of species, genera, orders, and so on in the Linnæan taxonomy of organisms and the creation of natural kinds to categorize ordinary objects

Unsupervised clustering begins with data. Figure (a) shows 500 data points, each of which specifies the values of two continuous attributes. The data points might correspond to stars, and the attributes might correspond to spectral intensities at two particular frequencies. Next, we need to understand what kind of probability distribution might have generated the data. Clustering presumes that the data are generated from a **mixture distribution**, P . Such a distribution has k **components**, each of which is a distribution in its own right. A data point is generated by first choosing a component and then generating a sample from that component. Let the random variable C denote the component, with values 1; : : : ; k; then the mixture

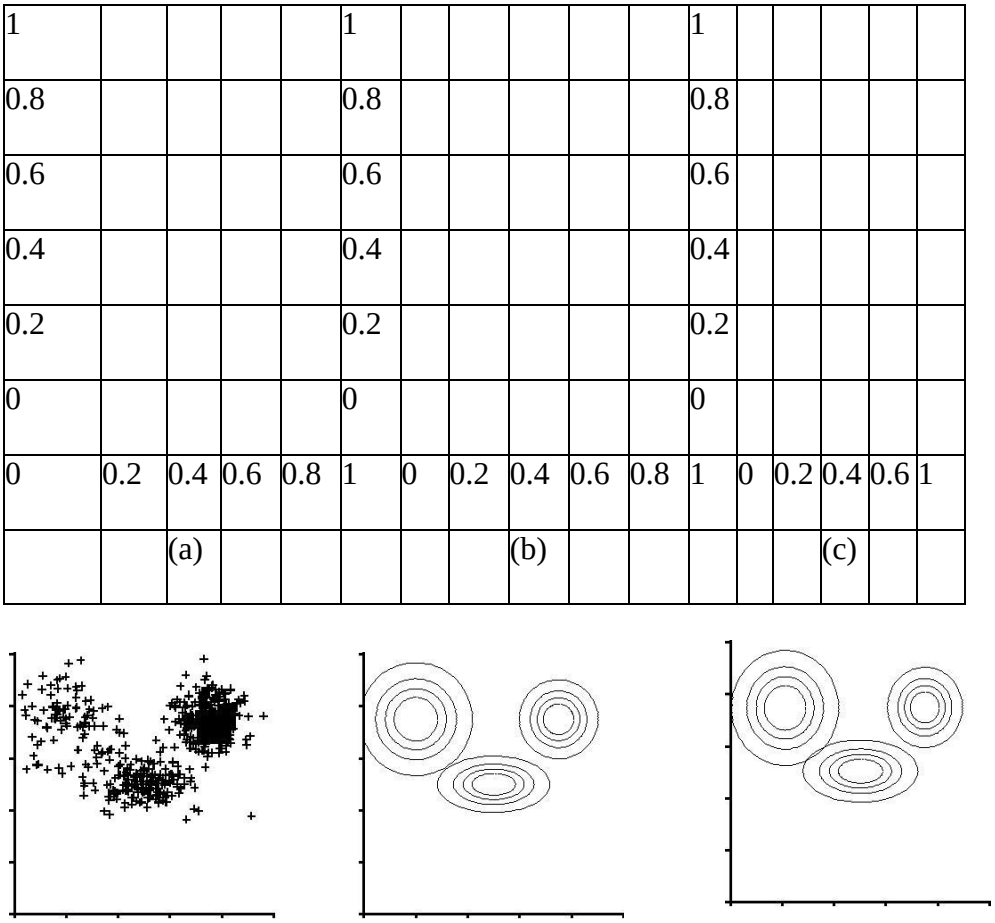


Figure 20.8 (a) 500 data points in two dimensions, suggesting the presence of three clus-ters. (b) A Gaussian mixture model with three components; the weights (left-to-right) are 0.2, 0.3, and 0.5. The data in (a) were generated from this model. (c) The model reconstructed by EM from the data in (b).

distribution is given by **k**

X

P (**x**) = P (C = i) P (**x**jC = i) ;

i = 1

where **x** refers to the values of the attributes for a data point. For continuous data, a natural choice for the component distributions is the multivariate Gaussian, which gives the so-called **mixture of Gaussians** family of distributions. The parameters of a mixture of Gaussians are w_i = P (C = i) (the weight of each component), $_i$ (the mean of each component), and $_i$ (the covariance of each component). Figure 20.8(b) shows a mixture of three Gaussians; this mixture is in fact the source of the data in (a).

The unsupervised clustering problem, then, is to recover a mixture model like the one in Figure 20.8(b) from raw data would be easy to recover the component Gaussians: we could just select all the data points a given component and then apply (a multivariate version of) Equation (20.4) for fitting the parameters of a Gaussian to a set of data. On the other hand, if we *knew* the parameters of each component, then we could, at least in a probabilistic sense, assign each data point to a component. The problem is that we know neither the assignments nor the parameters.

The basic idea of EM in this context is to *pretend* that we know the the parameters of the model

and then to infer the probability that each data point belongs to each component. After that, we refit the components to the data, where each component is fitted to the entire data set with each point weighted by the probability that it belongs to that component. The process iterates until convergence. Essentially, we are “completing” the data by inferring probability distributions over the hidden variables—which component each data point belongs to—based on the current model. For the mixture of Gaussians, we initialize the mixture model parame-ters arbitrarily and then iterate the following two steps:

1. E-step: Compute the probabilities $p_{ij} = P(C = ij\mathbf{x_j})$, the probability that datum $\mathbf{x_j}$ was generated by component i. By Bayes' rule, we have $p_{ij} = P(\mathbf{x_j}\ jC = i)P(C = i)$. The term $P(\mathbf{x_j}\ jC = i)$ is just the probability at $\mathbf{x_j}$ of the ith Gaussian, and the term $P(C = i)$ is just the weight parameter for the ith Gaussian. Define $p_i = {}^{\mathbf{P}}{}_{\mathbf{j}}\ p_{ij}$.

2. M-step: Compute the new mean, covariance, and component weights as follows:

$$\mathbf{i} \quad p_{ij}\ \mathbf{x_j} = p_i$$

$$\mathbf{j}$$

$$\mathbf{X}$$

$$\mathbf{i} \quad {}^{x}\ p_{ij}\ \mathbf{x_j}\ \mathbf{x_j}^{>} = p_i$$

$$\mathbf{j}$$

$$w\mathbf{i} \quad p_i :$$

The E-step, or *expectation* step, can be viewed as computing the expected values p_{ij} of the INDICATOR VARIABLE hidden **indicator variables** Z_{ij} , where Z_{ij} is 1 if datum $\mathbf{x_j}$ was generated by the i^{th} component and 0 otherwise. The M-step, or *maximization* step, finds the new values of the parameters that maximize the log likelihood of the data, given the expected values of the hidden indicator variables.

The final model that EM learns when it is applied to the data in Figure 20.8(a) is shown in Figure 20.8(c); it is virtually indistinguishable from the original model from which the data were generated. Figure 20.9(a) plots the log likelihood of the data according to the current model as

EM progresses. There are two points to notice. First, the log likelihood for the final learned model slightly *exceeds* that of the original model, from which the data were generated. This might seem surprising, but it simply reflects the fact that the data were generated randomly and might not provide an exact reflection of the underlying model. The second point is that *EM increases the log likelihood of the data at every iteration.* This fact can be proved in general. Furthermore, under certain conditions, EM can be proven to reach a local maximum in likelihood. (In rare cases, it could reach a saddle point or even a local minimum.) In this sense, EM resembles a gradient-based hill-climbing algorithm, but notice that it has no "step size" parameter!.

Things do not always go as well as Figure 20.9(a) might suggest. It can happen, for example, that one Gaussian component shrinks so that it covers just a single data point. Then its variance will go to zero and its likelihood will go to infinity! Another problem is that two components can "merge," acquiring identical means and variances and sharing their data points. These kinds of degenerate local maxima are serious problems, especially in high dimensions. One solution is to place priors on the model parameters and to apply the MAP version of EM. Another is to restart a component with new random parameters if it gets too small or too close to another component. It also helps to initialize the parameters with reasonable values.

Learning Bayesian networks with hidden variables

To learn a Bayesian network with hidden variables, we apply the same insights that worked for mixtures of Gaussians. Figure 20.10 represents a situation in which there are two bags of candies that have been mixed together. Candies are described by three features: in addition to the **Flavor** and the **Wrapper**, some candies have a **Hole** in the middle and some do not.

	700					-1975							
	600					-1980							
	500					-1985							
L					L								
	400					-1990							
	300					-1995							
						-2000							
-likelihood	200				-likelihood	-							

						2005							
	100					-2010							
Log					Log								
	0					-2015							
	-100					-2020							
	-200					-2025							
	0	5	10	15	20		0	20	40	60	80	100	120

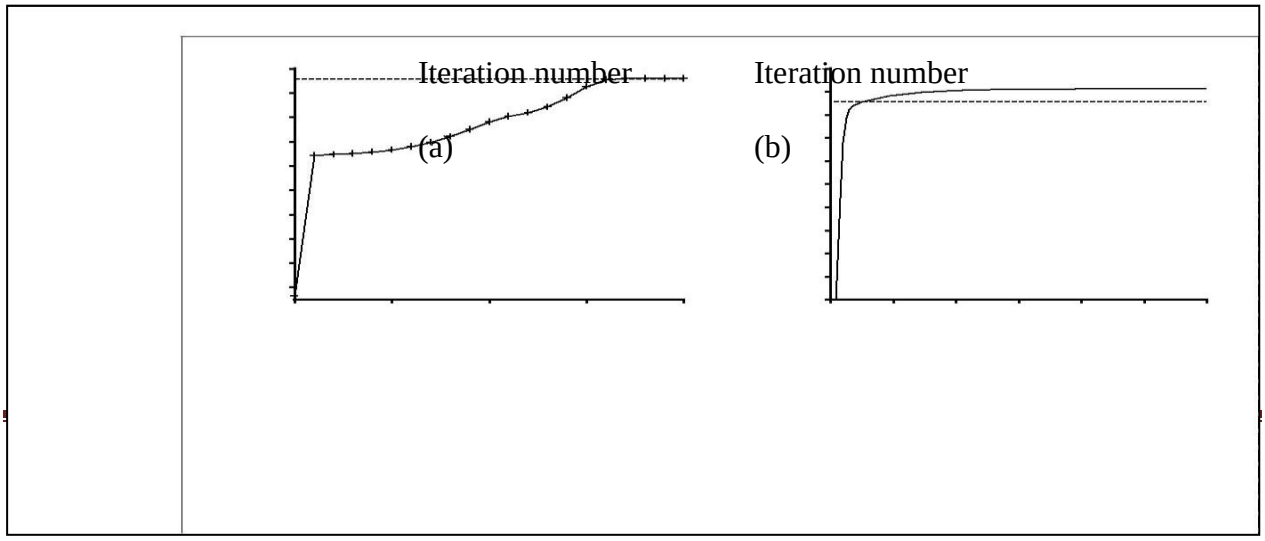
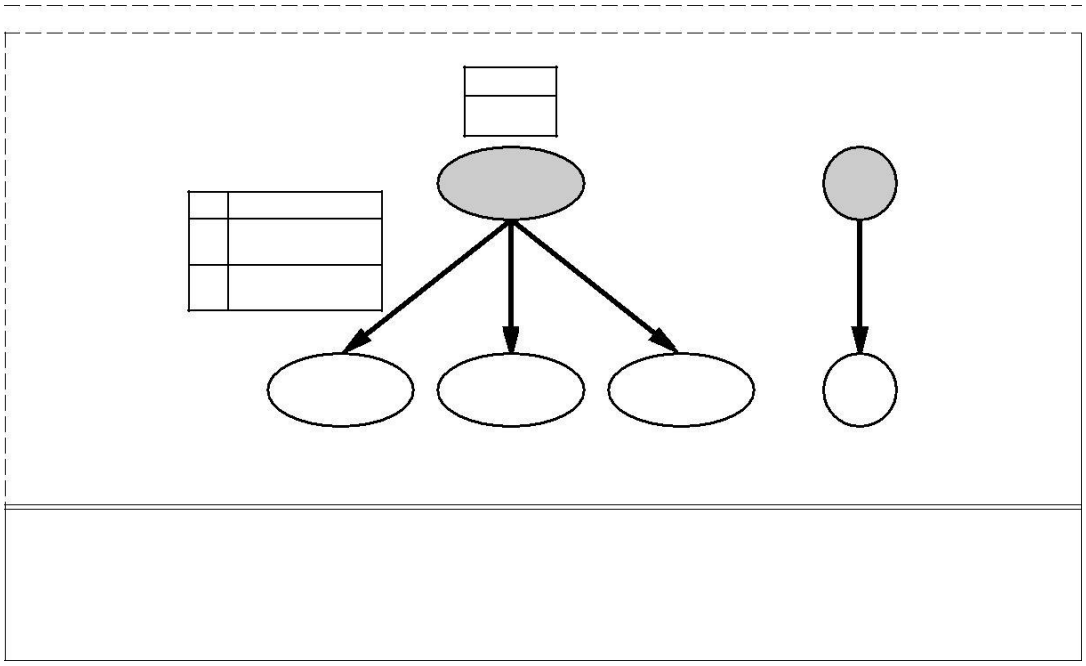


Figure 20.9 Graphs showing the log-likelihood of the data, L, as a function of the EM iteration. The horizontal line shows the log-likelihood according to the true model. (a) Graph for the Gaussian mixture model in Figure 20.8. (b) Graph for the Bayesian network in Figure 20.10(a).



$P(Bag=1)$

□

Bag *C*

Bag P(*F=cherry* | *B*)

1 θ_{F1}

2 θ_{F2}

Flavor *Wrapper* *Holes* **X**

(a) (b)

Figure 20.10 (a) A mixture model for candy. The proportions of different flavors, wrap-pers, and numbers of holes depend on the bag, which is not observed. (b) Bayesian network for a Gaussian mixture. The mean and covariance of the observable variables **X** depend on the component C.

The distribution of candies in each bag is described by a **naive Bayes** model: the features are independent, given the bag, but the conditional probability distribution for each feature depends on the bag. The parameters are as follows: is the prior probability that a candy comes from Bag 1; θ_{F1} and θ_{F2} are the probabilities that the flavor is cherry, given that the candy comes from Bag 1 and Bag 2 respectively; θ_{W1} and θ_{W2} give the probabilities that the wrapper is red; and θ_{H1} and θ_{H2} give the probabilities that the candy has a hole. Notice that

the overall model is a mixture model. (In fact, we can also model the mixture of Gaussians as a Bayesian network, as shown in Figure 20.10(b).) In the figure, the bag is is a hidden variable because, once the candies have been mixed together, we no longer know which bag each candy came from. In such a case, can we recover the descriptions of the two bags by observing candies from the mixture?

Let us work through an iteration of EM for this problem. First, let's look at the data. We generated 1000 samples from a model whose true parameters are

$$= 0{:}5; \quad _{\mathbf{F}\,1} = \ _{\mathbf{W}\,1} = \ _{\mathbf{H}\,1} = 0{:}8; \ _{\mathbf{F}\,2} = \ _{\mathbf{W}\,2} = \ _{\mathbf{H}\,2} = 0{:}3 : \qquad (20.7)$$

That is, the candies are equally likely to come from either bag; the first is mostly cherries with red wrappers and holes; the second is mostly limes with green wrappers and no holes. The counts for the eight possible kinds of candy are as follows:

	W = **red**		W = **green**	
	H = 1	H = 0	H = 1	H = 0

F = **cherry**	273	93	104	90
F = **lime**	79	100	94	167

We start by initializing the parameters. For numerical simplicity, we will choose[7]

$$(0) = 0:6; \quad (0) = (0) = 0:6; \quad (0) = (0) = (0) = 0:4 : \qquad (20.8)$$

F 1 **W** 1 **H** 1 **F** 2 **W** 2 **H** 2

First, let us work on the parameter. In the fully observable case, we would estimate this directly from the *observed* counts of candies from bags 1 and 2. Because the bag is a hidden variable, we calculate the *expected* counts instead. The expected count $\hat{}$ is the

N (**Bag** = 1)

sum, over all candies, of the probability that the candy came from bag 1:

$$\mathbf{N} \qquad (1) \qquad \hat{} \qquad = N\,(\mathbf{Bag} = 1) = N = P\,(\mathbf{Bag} = 1 j\, \mathbf{avor}_j ; \mathbf{wrapper}_j ; \mathbf{holes}_j) = N : \sum_{\mathbf{j}=1}^{\mathbf{x}}$$

These probabilities can be computed by any inference algorithm for Bayesian networks. For a naive Bayes model such as the one in our example, we can do the inference "by hand," using Bayes' rule and applying conditional independence:

$$P(\text{Bag} = 1 \mid \text{flavor}_1, \text{wrapper}_1, \text{holes}_1, \ldots, \text{flavor}_N, \text{wrapper}_N, \text{holes}_N) = \frac{P(\text{flavor}_1 \mid \text{Bag} = 1) P(\text{wrapper}_1 \mid \text{Bag} = 1) P(\text{holes}_1 \mid \text{Bag} = 1) P(\text{Bag} = 1)}{\sum_i P(\text{flavor}_1 \mid \text{Bag} = i) P(\text{wrapper}_1 \mid \text{Bag} = i) P(\text{holes}_1 \mid \text{Bag} = i) P(\text{Bag} = i)} \cdot \prod_{j=1}^{N} \frac{P(\text{flavor}_j \mid \text{Bag} = 1) P(\text{wrapper}_j \mid \text{Bag} = 1) P(\text{holes}_j \mid \text{Bag} = 1)}{\sum_i P(\text{flavor}_j \mid \text{Bag} = i) P(\text{wrapper}_j \mid \text{Bag} = i) P(\text{holes}_j \mid \text{Bag} = i)} \quad (1)$$

(Notice that the normalizing constant also depends on the parameters.) Applying this formula to, say, the 273 red-wrapped cherry candies with holes, we get a contribution of

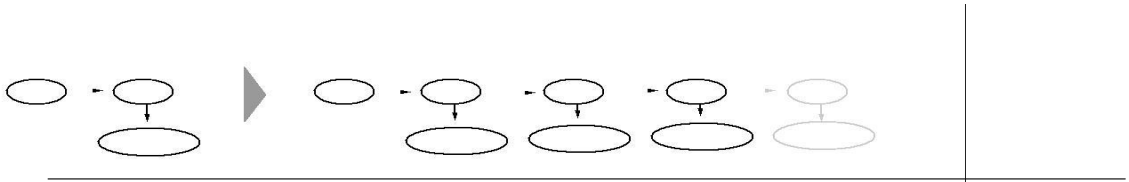
$$\frac{273}{100} \cdot \frac{\theta(1)\theta(1)\theta(1)}{\theta(1)\theta(1)\theta(1) + \theta(2)\theta(2)\theta(2)} \cdot \frac{\theta(1)}{\theta(1) + \theta(2)} \approx 0.22797 :$$

Continuing with the other seven kinds of candy in the table of counts, we obtain $\theta^{(1)} = 0.6124$.

[7] It is better in practice to choose them randomly, to avoid local maxima due to symmetry.

	R_0	$P(R_1)$			R_0	$P(R_1)$		R_1	$P(R_2)$		R_2	$P(R_3)$		R_3	$P(R_4)$	
$P(R_0)$	t	0.7		$P(R_0)$	t	0.7		t	0.7		t	0.7		t	0.7	
	f	0.3			f	0.3		f	0.3		f	0.3		f	0.3	
0.7				0.7												
$Rain_0$		$Rain_1$		$Rain_0$		$Rain_1$			$Rain_2$			$Rain_3$			$Rain_4$	
		$Umbrella_1$				$Umbrella_1$			$Umbrella_2$			$Umbrella_3$			$Umbrella_4$	
		R_1	$P(U_1)$			R_1	$P(U_1)$		R_2	$P(U_2)$		R_3	$P(U_3)$		R_4	$P(U_4)$
		t	0.9			t	0.9		t	0.9		t	0.9		t	0.9
		f	0.2			f	0.2		f	0.2		f	0.2		f	0.2

Figure 20.11 An unrolled dynamic Bayesian network that represents a hidden Markov model (repeat of Figure 15.14).



Now let us consider the other parameters, such as $_{F\ 1}$. In the fully observable case, we would estimate this directly from the *observed* counts of cherry and lime candies from bag 1. The *expected* count of cherry candies from bag 1 is given by

X

P (**Bag** = 1j**Flavor** $_{j}$ = **cherry** ; **wrapper** $_{j}$; **holes** $_{j}$) :

j:Flavor $_{j}$ = cherry

Again, these probabilities can be calculated by any Bayes net algorithm.

process, we obtain the new values of all the parameters:

$^{(1)}$ = 0:6124; $_{F}{}^{(1)}{}_{1}$ = 0:6684; $_{W}{}^{(1)}{}_{1}$ = 0:6483; $_{H}{}^{(1)}{}_{1}$ = 0:6558; $_{F}{}^{(1)}{}_{2}$ = 0:3887; $_{W}{}^{(1)}{}_{2}$ = 0:3817; $_{H}{}^{(1)}{}_{2}$ = 0:3827 :

(20.9)

Completing this

The log likelihood of the data increases from about 2044 initially to about 2021 after the first iteration, as shown in Figure 20.9(b). That is, the update improves the likelihood itself by a factor of about e^{23} 10^{10}. By the tenth iteration, the learned model is a better fit than the original model (L = 1982:214). Thereafter, progress becomes very slow. This is not uncommon with EM, and many practical systems combine EM with a gradient-based algorithm such as Newton–Raphson (see Chapter 4) for the last phase of learning.

The general lesson from this example is that the parameter updates for Bayesian net-work learning with hidden variables are directly available from the results of inference on each example. Moreover, only local posterior probabilities are needed for each parameter.

For the general case in which we are learning the conditional probability parameters for each variable X_i, given its parents —that is, $_{ijk} = P(X_i = x_{ij} j\mathbf{Pa}_i = \mathbf{pa}_{ik})$—the update is given by the normalized expected counts as follows:

$\wedge \quad \wedge$

$_{ijk} \quad N(X_i = x_{ij}; \mathbf{Pa}_i = \mathbf{pa}_{ik}) = N(\mathbf{Pa}_i = \mathbf{pa}_{ik}) :$

The expected counts are obtained by summing over the examples, computing the probabilities P ($X_i = x_{ij}$; $\mathbf{Pa}_i = \mathbf{pa}_{ik}$) for each by using any Bayes net inference algorithm. For the exact algorithms—including variable elimination—all these probabilities are obtainable directly as a by-product of standard inference, with no need for extra computations specific to learning. Moreover, the information needed for learning is available *locally* for each parameter.

Learning hidden Markov models

Our final application of EM involves learning the transition probabilities in hidden Markov models (HMMs). Recall from Chapter 15 that a hidden Markov model can be represented by a dynamic Bayes net with a single discrete state variable, as illustrated in Figure 20.11. Each data point consists of an observation *sequence* of finite length, so the problem is to learn the transition probabilities from a set of observation sequences (or possibly from just one long sequence).

We have already worked out how to learn Bayes nets, but there is one complication: in Bayes nets, each parameter is distinct; in a hidden Markov model, on the other hand, the individual transition probabilities from state i to state j at time t, $_{ijt} = P(X_{t+1} = jjX_t = i)$, are *repeated* across time—that is, $_{ijt} = _{ij}$ for all t. To estimate the transition probability from state i to state j, we simply calculate the expected proportion of times that the system undergoes a transition to state j when in state i:

$$\mathbf{X}\ \wedge$$

$$\mathbf{ij}\ N\ (X\mathbf{t}+1 \qquad \wedge$$

$$\mathbf{t} \qquad = j;\ X_{\mathbf{t}\ =\ \mathbf{i}})=N\ (X_{\mathbf{t}\ =\ \mathbf{i}})\ :$$

$$\mathbf{t}$$

$$\mathbf{X}$$

Again, the expected counts are computed by any HMM inference algorithm. The **forward– backward** algorithm shown in Figure 15.4 can be modified very easily to compute the neces-sary probabilities. One important point is that the probabilities

required are those obtained by **smoothing** rather than **filtering**; that is, we need to pay attention to subsequent evidence in estimating the probability that a particular transition occurred. As we said in Chapter 15, the evidence in a murder case is usually obtained *after* the crime (i.e., the transition from state i to state j) occurs.

The general form of the EM algorithm

We have seen several instances of the EM algorithm. Each involves computing expected values of hidden variables for each example and then recomputing the parameters, using the expected values as if they were observed values. Let **x** be all the observed values in all the examples, let **Z** denote all the hidden variables for all the examples, and let be all the parameters for the probability model. Then the EM algorithm is

$^{(i+1)}$ = argmax ^{X} P (**Z** = **z**j**x**; $^{(i)}$)L(**x**; **Z** = **z**j) :

z

This equation is the EM algorithm in a nutshell. The E-step is the computation of the sum-mation, which is the expectation of the log likelihood of the "completed" data with respect to the distribution P (**Z** = **z**j**x**; $^{(i)}$), which is the posterior over the hidden variables, given the data. The M-step is the maximization of this expected log likelihood with respect to the parameters. For mixtures of Gaussians, the hidden

variables are the Z_{ij} s, where Z_{ij} is 1 if example j was generated by component i. For Bayes nets, the hidden variables are the values of the unobserved variables for each example. For HMMs, the hidden variables are the i ! j transitions. Starting from the general form, it is possible to derive an EM algorithm for a specific application once the appropriate hidden variables have been identified.

As soon as we understand the general idea of EM, it becomes easy to derive all sorts of variants and improvements. For example, in many cases the E-step—the computation of

posteriors over the hidden variables—is intractable, as in large Bayes nets. It turns out that one can use an *approximate* E-step and still obtain an effective learning algorithm. With a sampling algorithm such as MCMC (see Section 14.5), the learning process is very intuitive: each state (configuration of hidden and observed variables) visited by MCMC is treated ex-actly as if it were a complete observation. Thus, the parameters can be updated directly after each MCMC transition. Other forms of approximate inference, such as variational and loopy methods, have also proven effective for learning very large networks.

Learning Bayes net structures with hidden variables

In Section 20.2, we discussed the problem of learning Bayes net structures with complete data. When hidden variables are taken into consideration, things get more difficult. In the simplest case, the hidden variables are listed along with the observed variables; although their values are not observed, the learning algorithm is told that they exist and must find a place for them in the network structure. For example, an algorithm might try to learn the structure shown in Figure 20.7(a), given the information that **HeartDisease** (a three-valued variable) should be included in the model. If the learning algorithm is not told this information, then there are two choices: either pretend that the data is really complete—which forces the algorithm to learn the parameter-intensive model in Figure 20.7(b)—or *invent* new hidden variables in order to simplify the model. The latter approach can be implemented by including new modification choices in the

structure search: in addition to modifying links, the algorithm can add or delete a hidden variable or change its arity. Of course, the algorithm will not know that the new variable it has invented is called **HeartDisease**; nor will it have meaningful names for the values. Fortunately, newly invented hidden variables will usually be connected to pre-existing variables, so a human expert can often inspect the local conditional distributions involving the new variable and ascertain its meaning.

As in the complete-data case, pure maximum-likelihood structure learning will result in a completely connected network (moreover, one with no hidden variables), so some form of complexity penalty is required. We can also apply MCMC to approximate Bayesian learning. For example, we can learn mixtures of Gaussians with an unknown number of components by sampling over the number; the approximate posterior distribution for the number of Gaussians is given by the sampling frequencies of the MCMC process.

So far, the process we have discussed has an outer loop that is a structural search pro-cess and an inner loop that is a parametric optimization process. For the complete-data case, the inner loop is very fast—just a matter of extracting conditional frequencies from the data set. When there are hidden variables, the inner loop may involve many iterations of EM or a gradient-based algorithm, and each iteration involves the calculation of posteriors in a Bayes net, which is itself an NP-hard problem. To date, this approach has proved impractical for learning complex models. One possible improvement is the so-called **structural EM** algo-rithm, which operates in much

the same way as ordinary (parametric) EM except that the algorithm can update the structure as well as the parameters. Just as ordinary EM uses the current parameters to compute the expected counts in the E-step and then applies those counts in the M-step to choose new parameters, structural EM uses the current structure to compute expected counts and then applies those counts in the M-step to evaluate the likelihood for potential new structures. (This contrasts with the outer-loop/inner-loop method, which com-putes new expected counts for each potential structure.) In this way, structural EM may make several structural alterations to the network without once recomputing the expected counts, and is capable of learning nontrivial Bayes net structures. Nonetheless, much work remains to be done before we can say that the structure learning problem is solved.

INSTANCE-BASED LEARNING

So far, our discussion of statistical learning has focused primarily on fitting the parameters of a *restricted* family of probability models to an *unrestricted* data set. For example, unsupervised clustering using mixtures of Gaussians assumes that the data are explained by the *sum* a *fixed* number of *Gaussian* distributions. We call such methods **parametric learning**. Para-metric learning methods are often simple and effective, but assuming a particular restricted family of models often oversimplifies what's happening in the real world, from where the data come. Now, it is true when we have very little data, we cannot hope to learn a complex and detailed model, but it seems silly to keep the hypothesis complexity fixed even when the data set grows very large!

In contrast to parametric learning, **nonparametric learning** methods allow the hypoth-esis complexity to grow with the data. The more data we have, the wigglier the hypothesis can be. We will look at two very simple families of nonparametric **instance-based learning** (or **memory-based learning**) methods, so called because they construct hypotheses directly from the training instances themselves.

Nearest-neighbor models

The key idea of **nearest-neighbor** models is that the properties of any particular input point **x** are likely to be similar to those of points in the neighborhood of **x**. For example, if we want to do **density estimation**—that is, estimate the value of an unknown probability density at **x**— then we can simply measure the density with which points are scattered in the neighborhood of **x**. This sounds very simple, until we realize that we need to specify exactly what we mean by "neighborhood." If the neighborhood is too small, it won't contain any data points; too large, and it may include *all* the data points, resulting in a density estimate that is the same everywhere. One solution is to define the neighborhood to be just big enough to include k points, where k is large enough to ensure a meaningful estimate. For fixed k, the size of the neighborhood varies—where data are sparse, the neighborhood is large, but where data are dense, the neighborhood is small. Figure 20.12(a) shows an example for data scattered in two dimensions. Figure 20.13 shows the results of k-nearest-neighbor density estimation from these data with k = 3, 10, and 40 respectively. For k = 3, the density estimate at any point is based on only 3 neighboring points and is highly variable. For k = 40, the estimate provides a good reconstruction of the true density shown in Figure 20.12(b). For k = 40, the neighborhood becomes too large and structure of the data is altogether lost. In practice, using

1											
					Density						
0.8											
					18						
					16						
					14						
0.6											
					12						
					10						
					8						
0.4					6						
					4						
											1
					2						
											0.8
					0						0.6
0.2					0	0.2					0.4

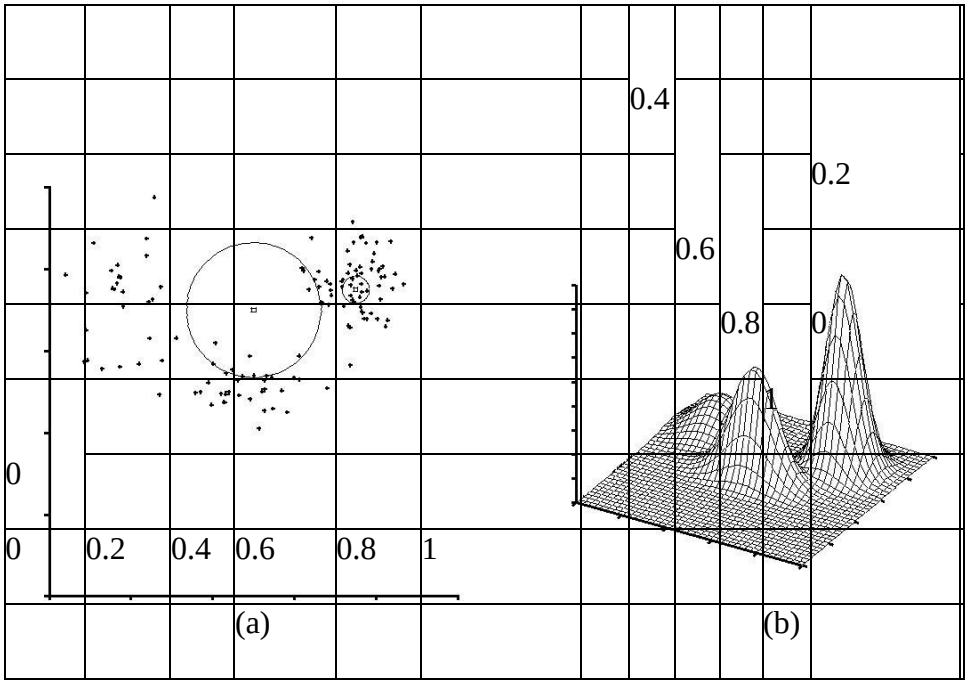
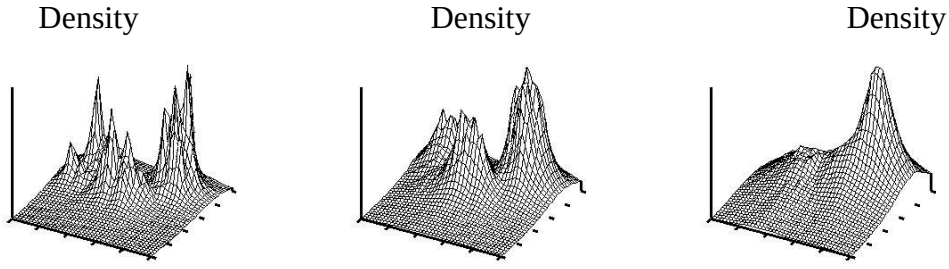


Figure 20.12 (a) A 128-point subsample of the data shown in Figure 20.8(a), together with two query points and their 10-nearest-neighborhoods. (b) A 3-D plot of the mixture of Gaussians from which the data were generated.



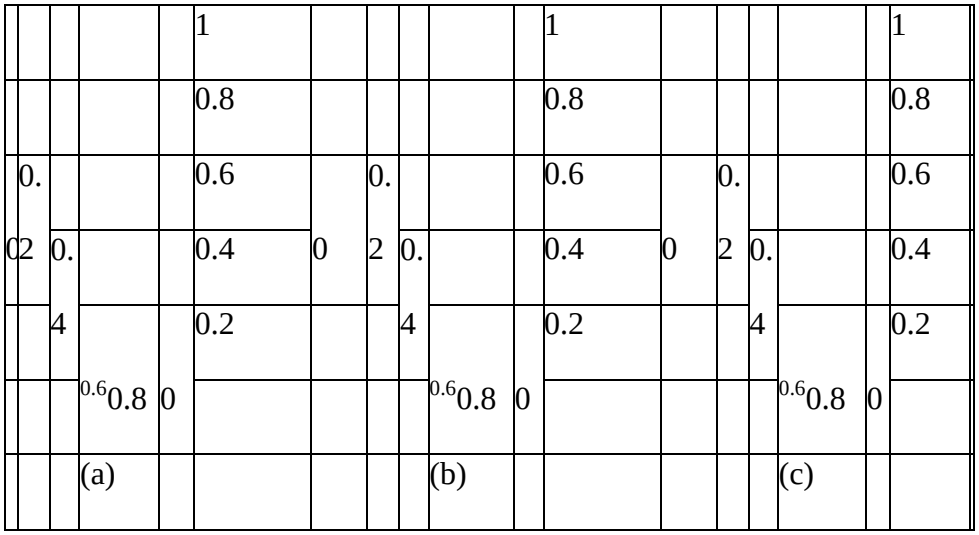


Figure 20.13 Density estimation using k-nearest-neighbors, applied to the data in Fig-ure 20.12(a), for k = 3, 10, and 40 respectively.

a value of k somewhere between 5 and 10 gives good results for most low-dimensional data sets. A good value of k can also be chosen by using cross-validation.

To identify the nearest neighbors of a query point, we need a distance metric, $D(\mathbf{x}_1; \mathbf{x}_2)$. The two-dimensional example in Figure 20.12 uses Euclidean distance. This is inappropriate when each dimension of the space is measuring something different—for example, height and weight—because changing the scale of one dimension would change the set of nearest neighbors. One solution is to standardize the scale for each dimension. To do this, we measure the standard deviation of each feature over the whole data set and express feature values as multiples of the

standard deviation for that feature. (This is a special case of the **Mahalanobis distance**, which takes into account the covariance of the features as well.) Finally, for discrete features we can use the **Hamming distance**, which defines $D(\mathbf{x}_1; \mathbf{x}_2)$ to be the number of features on which $\mathbf{x}_1$ and $\mathbf{x}_2$ differ.

Density estimates like those shown in Figure 20.13 define joint distributions over the input space. Unlike a Bayesian network, however, an instance-based representation cannot contain hidden variables, which means that we cannot perform unsupervised clustering as we did with the mixture-of-Gaussians model. We can still use the density estimate to predict a target value y given input feature values **x** by calculating P (yj**x**) = P (y; **x**)=P (**x**), provided that the training data include values for the target feature.

It is also possible to use the nearest-neighbor idea for direct supervised learning. Given a test example with input **x**, the output y = h(**x**) is obtained from the y-values of the k nearest neighbors of **x**. In the discrete case, we can obtain a single prediction by majority vote. In the continuous case, we can average the k values or do local linear regression, fitting a hyper plane to the k points and predicting the value at **x** according to the hyper plane.

The k-nearest-neighbor learning algorithm is very simple to implement, requires little in the way of tuning, and often performs quite well. It is a good thing to try first on a new learning problem. For large data sets, however, we require an efficient mechanism for finding the nearest neighbors of a query point **x**—simply calculating the distance to every point would take far too long. A

variety of ingenious methods have been proposed to make this step efficient by preprocessing the training data. Unfortunately, most of these methods do not scale well with the dimension of the space (i.e., the number of features).

High-dimensional spaces pose an additional problem, namely that nearest neighbors in such spaces are usually a long way away! Consider a data set of size N in the d-dimensional unit hypercube, and assume hypercubic neighborhoods of side b and volume b^d. (The same argument works with hyper spheres, but the formula for the volume of a hyper sphere is more complicated.) To contain k points, the average neighborhood must occupy a fraction k=N of the entire volume, which is 1. Hence, b^d = k=N , or b = $(k=N)^{1=d}$. So far, so good. Now let the number of features d be 100 and let k be 10 and N be 1,000,000. Then we have b 0:89—that is, the neighborhood has to span almost the entire input space! This suggests that nearest-neighbor methods cannot be trusted for high-dimensional data. In low dimensions there is no problem; with d = 2 we have b = 0:003.

Kernel models

In a **kernel model**, we view each training instance as generating a little density function—a **kernel function**—of its own. The density estimate as a whole is just the normalized sum of all the little kernel functions. A training instance at $\mathbf{x_i}$ will generate a kernel function K($\mathbf{x}$; $\mathbf{x_i}$) that assigns a probability to each point $\mathbf{x}$ in the space. Thus, the density estimate is

1 N

X

P (^{x}) = $_{N\ i=1}$ K(^{x}; ^{x}**i**) :

Density						Density						Density					
					1					1							1
					0.8					0.8							0.8
0	0.2	0.4			0.6	0	0.2	0.4		0.6	0	0.2	0.4				0.6
					0.4					0.4							0.4
			$0.6_{0.8}$	0	0.2				$0.6_{0.8}$	0.2				$0.6_{0.8}$	0		0.2
			(a)						(b)					(c)			

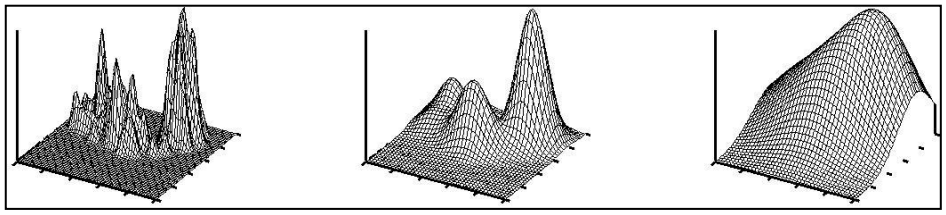


Figure 20.14 Kernel density estimation for the data in Figure 20.12(a), using Gaussian kernels with w = 0:02, 0:07, and 0:20 respectively.

The kernel function normally depends only on the *distance* D(**x**; $\mathbf{x_i}$) from **x** to the instance $\mathbf{x_i}$. The most popular kernel function is (of course) the Gaussian. For simplicity, we will assume spherical Gaussians with standard deviation w along each axis, i.e.,

$$K(^{x}; ^{x}\mathbf{i}) = {}_{(w2}{}^{p}2_{)}\mathbf{d}\ ^{e};\quad D(\mathbf{x};\mathbf{x})^2 \quad 1 \quad i2w^2$$

where d is the number of dimensions in **x**. We still have the problem of choosing a suitable value for w; as before, making the neighborhood too small gives a very spiky estimate—see Figure 20.14(a). In (b), a medium value of w gives a very good reconstruction. In (c), too large a

neighborhood results in losing the structure altogether. A good value of w can be chosen by using cross-validation. Supervised learning with kernels is done by taking a *weighted* combination of all the predictions from the training instances. (Compare this with k-nearest-neighbor prediction, which takes an unweighted combination of the nearest k instances.) The weight of the ith instance for a query point **x** is given by the value of the kernel $K(\mathbf{x}; \mathbf{x}_i)$. For a discrete prediction, we can take a weighted vote; for a continuous prediction, we can take weighted average or a weighted linear regression. Notice that making predictions with kernels requires looking at *every* training instance. It is possible to combine kernels with nearest-neighbor indexing schemes to make weighted predictions from just the nearby instances.

NEURAL NETWORKS

A **neuron** is a cell in the brain whose principal function is the collection, processing, and dissemination of electrical signals. Figure showed a schematic diagram of a typical neuron. The brain's information-processing capacity is thought to emerge primarily from networks of such neurons.