Chapter 16
Machine Learning (Supervised)

Shailesh Kumar

Every time we search the Web, buy a product online, swipe a credit card, or even check our e-mail, we are using a sophisticated *machine learning* system, built on a massive cloud platform, driving billions of decisions every day. Machine learning has many paradigms. In this chapter, we explore the philosophical, theoretical, and practical aspects of one of the most common machine learning paradigms—*supervised learning*—that essentially learns a mapping from an *observation* (e.g., symptoms and test results of a patient) to a *prediction* (e.g., disease or medical condition), which in turn is used to make *decisions* (e.g., prescription). This chapter explores the process, science, and art of building supervised learning models. The examples, corresponding code, and exercises for the chapter are given in the online appendices to the chapter.

1 Introduction

We are drowning in data yet starving for knowledge

The last few decades have seen an unprecedented growth in our ability to collect and process large volumes of data in a variety of domains—from science to social media, e-commerce to enterprises, Internet to Internet-of-things, and healthcare

Electronic supplementary material The online version of this chapter (https://doi.org/10.1007/978-3-319-68837-4_16) contains supplementary material, which is available to authorized users.

S. Kumar (✉)
Reliance Jio, Navi Mumbai, Maharashtra, India
e-mail: skumar.0127@gmail.com

© Springer Nature Switzerland AG 2019

B. Pochiraju, S. Seshadri (eds.), *Essentials of Business Analytics*, International Series in Operations Research & Management Science 264,
https://doi.org/10.1007/978-3-319-68837-4_16

to human resource management. Today's data-driven decision systems enable us to make intelligent, accurate, and real-time decisions using this data. They have the potential of making research, manufacturing, businesses, processes, enterprises, education, transportation, agriculture, and governance increasingly automated, efficient, and effective.

Today's data-driven decision systems are a result of a serendipitous convergence of three key technologies that matured over the last few decades: First, the *Internet* that made it possible for everyone to contribute to, and connect with the collective human knowledge and services globally. Second, *cloud computing* that made it possible for individuals and enterprises to store and process enormous amounts of data, and third, *machine learning*—the process, science, and art of converting data into insights, insights into predictions, and predictions into decisions. At a high level, there are three broad paradigms in machine learning:

- ***Unsupervised learning*** is typically used to *describe* the structure in the data (e.g., projection, density estimation, clustering) or *discover* latent patterns in it (e.g., communities in networks, topics in a corpus, or frequent item-sets in market basket data). The goal of unsupervised learning is to improve our understanding of the data and derive actionable insights from it.
- ***Supervised learning*** is typically used to learn a mapping from an *observation* (e.g., activities of a user in a bank) to a *prediction* (e.g., is the customer about to churn), leading to a *decision* (e.g., take action to prevent customer churn). Most decision systems today in a variety of domains are based on Supervised Learning models.
- ***Reinforcement learning*** is typically used in sequential decision tasks to predict the best *action* (e.g., next Chess move) from the current *state* (e.g., board position) to maximize immediate (e.g., strengthening the board position) and eventual (e.g., winning the game) *reward.*

In this chapter, we will focus on the philosophical, theoretical, and practical aspects of machine learning in general and supervised learning in particular.

- The ***philosophical*** goal of machine learning is to understand the nature of intelligence and learning itself. Here we will explore the fundamentals of *understanding* and *generalization* in the context of supervised learning.
- The ***theoretical*** goal of machine learning is to build and improve formal learning frameworks and algorithms. Here, we will explore various supervised learning algorithms, relationships among them, and their pros and cons.
- The ***practical*** goal of supervised learning is to blend data, domain knowledge, and learning algorithms to build accurate and, if needed, interpretable prediction models. Here we will explore some of the real-world challenges and practical aspects of building models.

2 The Philosophy: Nature of Intelligence

Our technology, our machines, is part of our humanity. We created them to extend ourselves, and that is what is unique about human beings!—Ray Kurzweil

Ever since the dawn of mankind, we have been trying to extend ourselves in all our faculties: If we could not lift more, we created levers and cranes; if we could not move fast and far, we created horse carts and cars; if we could not see far, we created telescopes; if we could not speak loud enough, we created microphones; if we could not compute fast enough, we created calculators and computers; if we could not talk far enough, we created telephones and mobiles; etc. In this journey, we are also extending one of our most important faculties that make us unique—our intelligence. Using machine learning and Artificial Intelligence, we are now at the early stages of building intelligent machines that can see, listen, speak, read, learn, understand, think, create, plan, and converse like humans.

Before we can build intelligent machines, however, it is essential to understand the nature of intelligence itself. Intelligence has many facets; for example, it is the ability to:

- *Learn* causality or correlation from past data (e.g., Should I approve this loan?)
- *Recognize* structures in the data (e.g., words in speech, objects in images)
- *Understand* semantics using context (e.g., *apple* is healthy, I like *apple* products)
- *Adapt* to novel situation (e.g., network routers react to change in traffic patterns)
- *Reason* about alternate ways of solving a problem (e.g., playing chess)
- *Synthesize* data (e.g., next word in a ??, next utterance in a conversation)

In the context of supervised learning, let us explore two of these notions of intelligence in a little more depth: understanding and generalization.

2.1 Understanding: From Syntax to Semantics

Does the Google Search Engine actually *understand* the Web? Do YouTube or NetFlix understand the videos they store? Do Amazon and Zomato understand the reviews written by their customers? Do our "smart" phones actually understand what we are speaking into them? It is one thing to collect, store, transfer, or index a large amount of data, but it is completely a different thing to actually *understand* it. One of the first fundamental qualities of an intelligent system is its ability to interpret the raw data it is receiving at the right level of abstraction (e.g., pixels, lines, blobs, eyes, face, body). But what does *understanding* mean?

Our language and sensory systems evolved not only to capture and transmit the raw data to the brain, but to actually *understand* it in real time, that is, to *identify* structures, objects, and attributes in them. Our visual system—perhaps the most sophisticated intelligent system so far—*looks* at pixels in the retina but *sees* a fresh red rose or a flying eagle in the brain. Similarly, when we process a sequence of words (e.g., "*Apple filed a suit against Orange*") we *interpret* or assign meaning to each part (word)—for example, "Apple" the company, not fruit—so the whole (sentence) makes sense.

***Understanding** is a hierarchical process of using **context** to interpret each part so the whole—as a juxtaposition of its parts—makes sense.*

A large class of Unsupervised and Supervised Learning algorithms today are understanding algorithms as they try to interpret the raw data, for example, this word is a noun (part of speech tagger), this document is about hockey (document classifier), this article mentions Mahatma Gandhi (information extraction), this video segment shows bungee jumping (activity recognition), this image shows a cat under a table (object recognition in images), this person is Mr. X (speaker recognition from voice, face recognition from image, or fingerprint recognition).

2.2 Generalization: From This to Such

A database that stores even billions of records is neither considered *knowledgeable* (since it does not understand the data it contains) nor *intelligent*. Now consider the letter in Fig. 16.1. We can tell what this letter is immediately in spite of the fact that we have never seen such renditions of this letter before. If we were a database system, we should have seen all possible renditions of this letter to recognize it. But that is not how our brain works: we do not ***memorize***; we ***generalize***. The second important property of intelligence—the basis of Supervised Learning—is this ability to *generalize* what we have learnt from the past mappings of input to output to new inputs we have never seen before.

In Supervised Learning, we try to mimic this aspect of intelligence. So to build a "classifier" that can recognize say a handwritten letter or an action in a video (e.g., bowling, batting, catching) we first provide it with enough "training examples" of what is the input and what should this input be called (the class label). We learn a

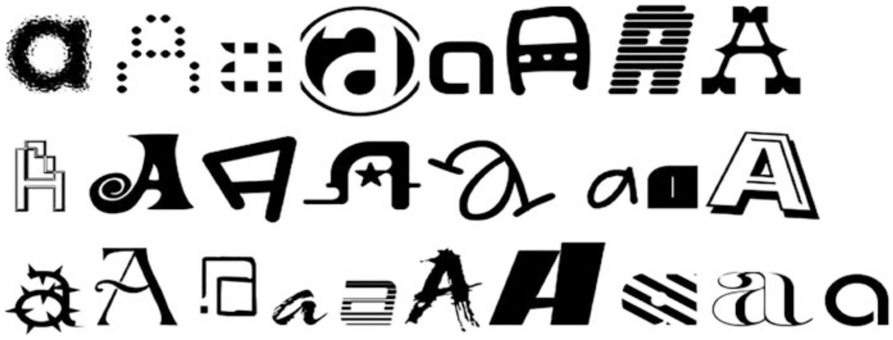


Fig. 16.1 We can recognize this letter immediately without having seen any of these renditions before

"classifier" with this training data that can now recognize new examples that it has never seen before. The basic principle of generalization is that:

Inputs that belong to the ***same*** *class must be* ***similar*** *to each other in some way.*

This brings up another important notion in machine learning called *similarity*. What makes two documents or two gene sequences or two pieces of music or two customers similar? One might say that learning to generalize an input is implicitly the art of defining what makes two inputs similar. For example, if we show a lot of images of cats to a computer vision system where they all differ by the background, the size of the cats, the color of the cat and we keep telling the learning system they are all cats then the model will figure out what is really similar among all these images (furry, big eyes, whiskers, etc.). Now if we distinguish all cats from all dogs the system will further figure out what makes cats more similar to each other, dogs more similar to each other, and cats different from dogs. So, in a way *supervised learning* is the art of *similarity learning* and the ability to determine what aspects to focus on (signal) and what aspects to ignore (noise) given a set of training examples.

Supervised learning learns to make "similar" objects "nearby" and "different" objects "distant" either by *compactly describing similar objects* (**descriptive**) or by *robustly discriminating between different* (**discriminative**) objects.

3 The Supervised Learning Paradigms

At a high-level supervised learning is a mapping between an input (e.g., cause) and an output (e.g., effect). Different paradigms in supervised learning—classification, regression, retrieval, recommendation—differ by the nature of their input and output:

3.1 The Classification Paradigm

A classification model maps a set of ***input features*** to a ***discrete class label*** (e.g., Digit (0–9) or character (A–Z) (class labels) from images; emotions (sad, happy, confused, frustrated, . . .) from face images; land-cover (water, marsh, sand, etc.) from remote sensing data; e-mail type (spam, promotion, finance, update) from e-mails (e-mail classifier); words (e.g., words in any language) from speech data (speech to text); objects (cat, dog, car, tree, etc.) in images; computer vision activity (stealing, holding, throwing, etc.) from videos (activity recognition), part-of-speech of a word in a sentence (POS taggers); sentiment in a tweet; or review about an entity (movie, product, etc.)

In all these cases the *input* could take any form—multivariate, text, image, speech, video, sequence of transactions, etc. This raw data is further converted to some meaningful *features*. The output is a discrete class label from a predefined set of labels. In two-class classification problems (e.g., spam vs. not-spam, churn vs. not-churn, conversion probability estimation), the output is typically interpreted as probability of target class (e.g., spam, churn, click). Appropriate thresholds on this probability can be used to make a binary decision. In general, the classes themselves might have hierarchies (e.g., news articles might be labelled as sports, entertainment, politics, science, etc. at the first level of hierarchy while within *sports* class, there might be subclasses for various sports or within science there might be subclasses such as space, medicine, and technology).

3.2 The Regression Paradigm

A regression model maps ***input features*** to a ***real or ordinal value*** (e.g., click-through-rate prediction in search and advertising, lifetime-value prediction of a customer, efficiency prediction from device sensors, capacity of a customer to take loan, and value of a house/property in a local neighborhood): Regression is used in many ways: either to predict a value, to predict *score* in a certain range (e.g., in ordinal regression we might want to predict a score from say 1 to 5 for ratings corresponding to poor, fair, bad, good, and excellent), or for forecasting a value into the future (e.g., demand prediction for products in retail).

3.3 The Recommendation Paradigm

A recommendation model maps a ***past behavior*** into ***future potential activities*** (e.g., which product a customer might buy given what he/she purchased, browsed, etc. in the past, which movie a user might like on NetFlix or YouTube given his/her past content consumption, who will a user like to connect to on Facebook or LinkedIn given his/her current connections and interactions, which news or tweet a user might like given his/her past consumptions, which topic the student should study next given how he/she has fared in past topics). A typical recommendation engine uses a two-stage process:

Creating user profile: In this first stage, user's past behavior is used to build his/her profile (a set of features and their weights). For example, in retail the profile might be built based on the products the user searched, added to wish list, purchased, read a review about, wrote a review about, etc. In education, the profile of a student may be created using the time spent on learning, number of problems solved, and test scores on problems associated with each topic. In YouTube, a user profile may be built based on videos previously watched, liked, and commented by the user. Note that a user may have different types of interactions with the same entity. Each interaction type could be given a different weightage (e.g., purchase is more important than browse, writing a review might be more important than reading one) while creating the user profile. Once the user profile is built it is used to make the actual recommendation.

Making Recommendations: In this second stage, the user's profile is now "matched" with the (properties of the) entities to determine whether a user would like to engage with that entity (product, topic in education, or YouTube video). The matching and profiling can be done at the *id level* (e.g., which movie, which video) or at the *property level* (which genera of movie, which director, etc.). The score may be further refined using a *utility* function based on the business goals; for example, for certain set of customers we might use recommendation for immediate cross-sell, while for others we might use it to maximize their lifetime value.

3.4 The Retrieval Paradigm

A retrieval model maps a ***query*** into a ***sorted list of entities*** (e.g., relevant Web pages on search engines for a given text query, relevant images, videos, news stories on search engine given a text query, relevant images/videos for an image query (content-based image retrieval), relevant song for a given humming or audio snippet, relevant property/car/products on various entity search portals, relevant flights/hotels on various travel portals (structured queries), and relevant gene sequences for a gene snippet query.

In both retrieval and recommendation paradigms, the output is a ***list of entities*** sorted by a score. The key difference is that in recommendation, the (recommendation) score is based on the ***behavior summarized into a profile*** of the user, while in retrieval the (relevance) score is based on the match to a ***query***. For example, in Web search, one might use URL match, title match, anchor text match (text associated with all incoming links to this page), header match, body match, etc. Click feedback data is used to learn the relative importance of various types of matches between query and entity fields to synthesize the final relevance score.

One of the key skills of a data scientist is to ***formulate*** a business problem as one (or more) of these paradigms, pick the right kind of modelling approach within the paradigm, and using data and domain knowledge to learn these models. In the rest of this chapter, we will focus primarily on the classification paradigm. You can refer Chaps. 7 and 8 (Linear and Advanced Regression) for the regression paradigm and Chap. 21 (Social Media and Web Analytics) for examples on the retrieval paradigm.

4 The Process: From Data to Decisions

Data Science is a continuous dialogue between data and business.

One of the primary goals of Data Science is to drive business and operational decisions from data to maximize profitability or efficiency metrics, respectively. Figure 16.2 shows the overall ***process*** typically used to drive decisions from data. We will explore each of the stages of this process in this section.

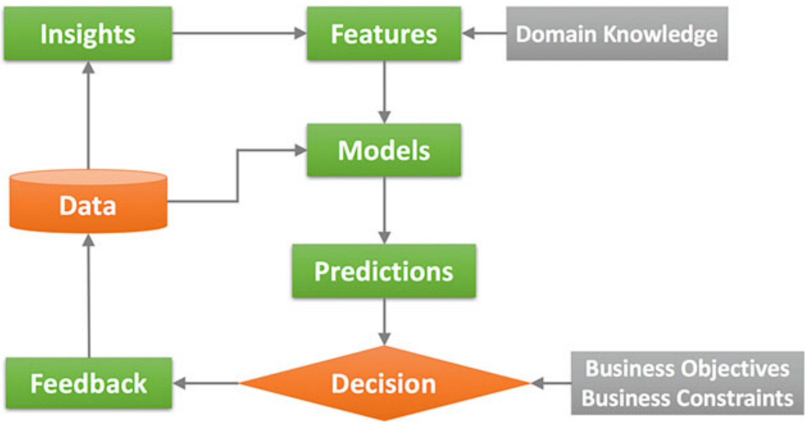


Fig. 16.2 The overall data science process for building and improving models

4.1 The Insights Stage: Dating the Data

Often the most effective way to ***describe***, ***explore***, *and* ***summarize*** *a set of* ***numbers***—*even a very large set*—*is to* ***look at pictures*** *of those numbers.*—*Edward R. Tufte*

The first stage in the data science process is to understand the nuances in the data itself before we start building models. The insights hidden in the data either confirm some of our own hypotheses about the underlying process that generated the data or reveal new aspects of the process that we did not know before. Some of the basic practices for revealing insights in the data include:

Feature Distributions: One of the most basic set of insights comes from individual feature distributions. Most modelling techniques assume normal or well-behaved distributions, while most real-world features are either exponentially or log-normally distributed. Looking at feature histograms reveals such nuances and helps correct for them by, for example, taking the log of those features that are exponentially distributed. Further, looking at feature distributions of different classes reveals whether or not a certain feature would be useful for discriminating various classes. Feature distributions reveal structure in each feature independent of other features.

Scatter Plots: A powerful yet simple technique in understanding feature interactions is scatter plots between all pairs of features. This visually shows correlation among features, if any. Further, color coding each data point with class label reveals combination of features that might help discriminate classes. Figure 16.3 shows scatter plots of IRIS dataset[1] with respect to a few pairs of features. Scatter plots limit us to only visualize the data two or three features at a time.

[1] https://archive.ics.uci.edu/ml/datasets/iris (Retrieved August 2, 2018).

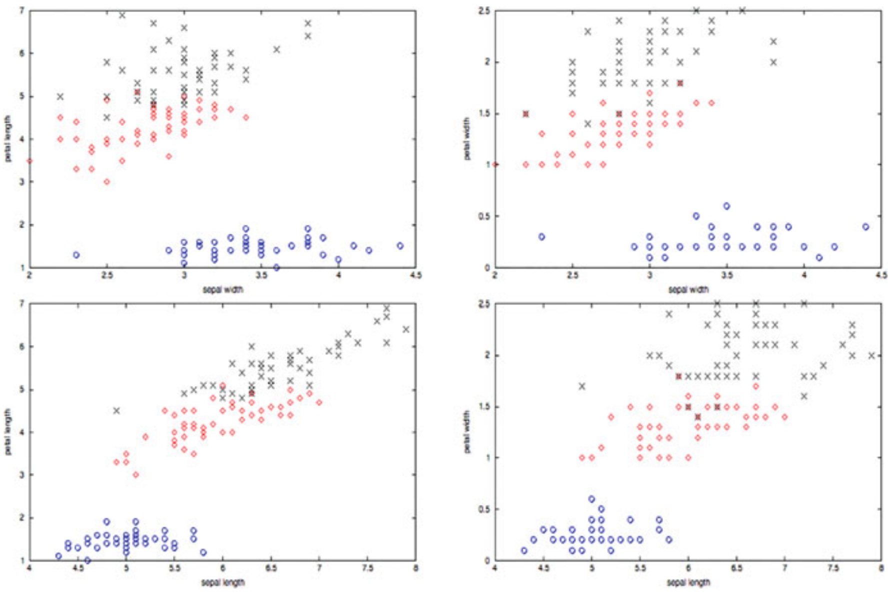


Fig. 16.3 Scatter plots of IRIS data—reveals how two classes are more similar to each other

Principal components analysis (PCA): Principal components analysis projects the data into lower-dimensional spaces by preserving maximal spread or variance in the projected space. Scatter plot of the top two or three principal components projection of the data reveals the “joint” structure in the data across all features.

Fisher discriminant analysis: PCA is an unsupervised projection that only preserves spread of all data points irrespective of their class labels. For classification problems it is far more useful to do a Fisher Discriminant Analysis where projection is done to exaggerate the differences between classes. Figure 16.4 shows the PCA vs. FDA projections of three classes in MNIST data.[2] It shows how Fisher projections try to separate the three classes while PCA projections do not care about the class labels.

Other visualization techniques: A large number of visualization techniques including self-organizing maps, multidimensional scaling, and t-SNE can be used to gain deeper understanding in the data.

[2]http://yann.lecun.com/exdb/mnist/ (Retrieved August 2, 2018).

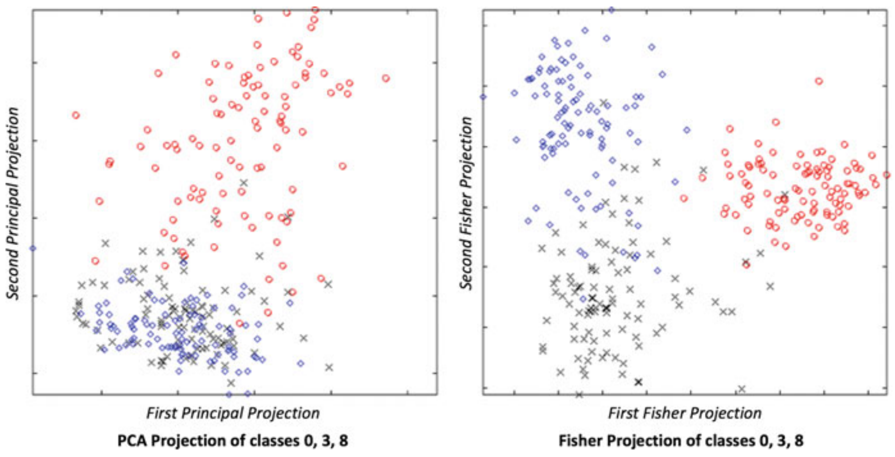


Fig. 16.4 PCA vs. Fisher projection of same data (classes 0, 3, and 8 from MNIST)

4.2 The Feature Engineering Stage: Exaggerate Signals that Matter

One of the most creative parts of the data science process—literally the art of data science—is the feature engineering stage. There are two types of data scientists, viz., feature engineering:

The **feature-centric** data scientists believe in systematically and painstakingly creating meaningful features to make the modelling stage simple. For them "real" data science happens here. They marry their deep understanding of the data (acquired from insights stage) with substantial appreciation of the domain knowledge (acquired from domain experts) to build features. These features are highly interpretable, semantically deeper than the original data, and cover all potential aspects of input-output mapping. This traditional approach to data science is more useful when labelled data is less compared to domain knowledge and interpretability of model output is as important as its accuracy.

The **model-centric** data scientists, on the other hand, believe that throwing a large amount of labelled data and computational resources (e.g., GPUs) will automatically learn the right features (in the lower layers) as well as the mapping between those features and the output (in the higher layers) of a *deep learning* model. Here, the creative process takes the form of designing the right architecture—nature and type of layers in the deep learning models as opposed to designing individual features. This model-centric deep learning approach works well in domains such as text, vision, speech, and time series data where (1) the space of possible features is very large, which makes it impractical to explore it through traditional feature engineering; (2) the amount of data is substantial enough to learn the large number of parameters in deep learning models; and (3) the semantic gap between the raw input (e.g., pixels in images or words in text) to the final output (e.g., activity in video or meaning of a document) is so large that we need a hierarchy of features and not just a single layer of features.

In the rest of this section, we will explore a number of transformations on raw data that constitute traditional feature engineering:

Feature transformation: In a typical model, the different input features might have very different distributions and ranges. Combining them into a model such as logistic regression without first making their distributions "compatible" makes the life of the model miserable. Taking log of certain features (that are exponentially or log-normally distributed), binning the values, or applying any domain-specific transformation (e.g., Fourier transformations or wavelet transformations on time-series data) might help build better models than just shoving the raw inputs into the model. For example, in many models using income as a feature, it might be better to either bin the income or take the log of the income since income is typically exponentially distributed (lots of people have low income, very few have very high income). Using percentile scores or cumulative density binning is also an example of taming the distribution variability in the data.

Feature normalization: Even after proper transformations, the raw inputs might be in different ranges and their values in different units. For example, to predict the value of a house, one might need features such as number of rooms and bathrooms (count), area of the house (square foot), distance from nearest school or places of interest (kilometers), prices of nearby houses sold recently (money), and age of the house (years). While the distribution can be tamed as described above, the values might still need to be brought into comparable ranges. For this, the features might need to be transformed to some min-max range (so min is always 0 and max is always 1) or z-scored values could be used (so the mean of each feature is zero and standard deviation is 1). Such transformations then let the model do the actual job of learning the relative importance of these features instead of forcing them to ***also*** compensate for these feature differences. Care must be taken to first remove outliers in each feature before learning parameters for min-max or z-score normalization.

Creating invariant features: Often the raw data contains variances in it that are not related to the problem at hand. For example, speech recognition problems have accent variances; images might have illumination, pose, rotation, and scale variances; and transaction and time series data might have seasonal variances. In essence, the final "data" that we see (e.g., sound of a word spoken by a person) is a "joint" of the actual signal in it (e.g., the actual word spoken) with additional factors (accent, tonal quality, loudness, etc.). Keeping what is essential for the task (signal) and ignoring what is not (noise) is the key to good feature engineering. Understanding and removing these variances is perhaps the most intricate part of feature engineering and requires deep understanding of the domain, possible sources of such variances, and the tools to remove these variances. If not removed, the model will become complex and will try to learn these variances instead of doing actual classification.

Ratio Features: A lot of features contain variances that can be removed simply by dividing them with other features. For example, in information retrieval models, query length bias is removed by dividing the total match between query and document field (e.g., title) with query length. In credit models, instead of using total debt it is better to use debt-to-income ratio, instead of using total-payment a better feature would be the percent of EMI paid, and instead of total-credit-taken, percent of credit limit reached might be better features. Such ratio features cannot be "discovered" by the modelling techniques that are only doing linear combination of features (e.g., logistic regression or linear Support Vector Machines). Infusing domain knowledge through ratio features helps model explore the right "space" in which to discriminate classes.

Output feature ratios: Not only the input features, even the output features might also have biases that must be corrected for before trying to predict them. For example, instead of predicting click-through-rate of a document for a query, we might want to first take into account the expected click-through-rate bias at each position (e.g., people are anyway more likely to click on the first result than second and so on irrespective of the query and document). In forecasting sales, instead of predicting the raw sales count, we might want to predict deviation from the expected sales given the context (city, season, etc.). The ratings data (movie or product rating) has inherent "consumer bias." A critical consumer will typically rate most products say 1–3 out of 5 and hardly give a rating of 5, while a generous customer might rate most products between 3 and 5. Now a rating of 4 on a certain product does not mean the same thing for these two customers. It should be "calibrated" correctly to remove individual customer's rating biases to make them "comparable" across customers.

Creating new features: Additional features beyond basic transformations, normalizations, ratios, and bias corrections are also needed in many domains. Consider, for example, four features in a credit card fraud prevention problem: location and time of the last and the current transaction. These four features by themselves put into a logistic regression model might not be able to predict whether the current transaction is fraud or not. But a common sense domain knowledge that "there should be sufficient time between two distant transactions" can be used to translate these four features into say a *velocity* feature, that is, ratio of distance between current and previous transaction to time between current and previous transaction is a single "semantic" feature that can help predict fraud. In speech and vision domain, biologically motivated semantic features are extracted from raw signals.

Defining output variable: In some of the problems the prediction variable might be very obvious (click through rate in search, spam vs. not-spam in Web page or e-mail classification, land-cover type in remote sensing, etc.). However, in many other domains, we might have to first define the output variable itself. For example, in *churn prediction,* we might have to define churn in terms of *future* user behavior (e.g., did not make any purchase in the last 3 months). In credit modelling, we might define a high-risk customer as someone who missed his last three EMIs in a row. In such problems where *future* is to be predicted based on *current and past* observation, defining the future output to be predicted becomes very critical.

Setting the right defaults: A default value is typically associated with a feature if no meaningful value can be assigned. For numeric features, often such default values are zero. Assuming such defaults or not setting them thoughtfully is one of the most common "bugs" in modelling. Consider a feature called *first-occurrence* of a query word in a document field. The earlier the word occurs in the field, the better—so lower the value of first-occurrence, the better. Now if in a field no query word is present, what should be the default value of this feature? If we pick a default value of 0, then it will confuse the model where both for the *best case (*when the query word is the first word (at position 0) of the field) and the *worst case* (where the query word is not at all present in the field) take the same feature value. A better default might be the length of the field plus a constant or a high number. It is essential to deliberate over the default values of all features to make sure that the default value in conjunction with the regular values are "consistent" with the goals of the modelling.

Imputing missing features: One of the realities of real-world data science is the absence of features in the collected data. This happens either because the data was never collected for a period of time and plugins to collect a feature were added later, or the sensor was down for a while, or there are data corruption issues. In these cases, either we use one of the many feature imputation techniques or use modelling techniques (such as decision trees and their variants) that gracefully handle missing features. Again substituting the wrong defaults or simple average value of a feature may not always work.

Feature selection: Once a large number of features have been engineered, we might decide not to use all of them together in the same model because some of them might be highly correlated with each other. Feature selection methods can be model agnostic (aka filter methods) or model centric (aka wrapper methods). In a model-agnostic approach, features are sorted by some measure of "goodness," which is computed based on their discriminative power (e.g., Fisher discriminant) and nonredundancy with other features. The best features are then chosen to build the models. Filter methods are used when we have a large number of features (say tens of thousands) and it is not clear which modelling technique we want to use. In model-centric feature, selection features are added one at a time (forward feature selection) or removed one at a time (backward feature selection) in a greedy manner to maximally increase the model performance (e.g., accuracy). Being model centric, every time a set of features is evaluated, the model has to be trained and evaluated. This makes model-centric feature selection potentially very time-consuming. Feature selection is a classic NP—hard "subset selection problem" where we know how to compute the "goodness" of a "set" of features but there is no simple (polynomial) algorithm to find an optimal set for a given dataset and modelling technique. Many other techniques such as genetic algorithms and simulated annealing have also been explored for feature selection.

4.3 The Modelling Stage: Matching Data and Model Complexity

Over the last several decades, the field of machine learning has given birth to a very large number of modelling techniques—some of which are described in the next section. Each technique has its own pros and cons and was developed to specifically address a set of weaknesses in other modelling techniques or "reformulate" the classification problem differently. In this section, we will explore the common guiding principles typically used for choosing the right modelling technique and using the output of these models correctly to solve the business problems.

Interpretability vs. accuracy: In a number of business problems, it is more important to interpret the output of the prediction model (i.e., give a reason for why the score is high or low) and not just to be accurate at it. For example, credit models are legally required to give top three reasons why a user has been denied a loan. Similarly, in churn prediction models, it might be useful not just to know that a certain customer is about to churn but also the reason why the customer is about to churn. This "reason code" can help address those reasons specifically for each customer. In such cases, it is better to use modelling techniques that are more interpretable and can generate a *reason code* along with a prediction score for each input. In cases where accuracy is more important than interpretability, another class of modelling techniques is preferred.

Scoring time vs. training time: Most models are deployed in high-throughput environments. For example, a search engine must be able to generate the top ten matches within half a second, a credit card fraud model must approve or disapprove each transaction within a second, in autonomous vehicles, the car must respond to the environment in real-time. In taxi hailing services, a cab must be allocated within a few seconds of a request. In all such cases the *scoring throughput* of the model must be high. While part of this is an engineering problem, part of it is also a data science problem where the right modelling technique makes all the difference. Similarly, the training time of a model might also matter when the model has to be updated frequently to compensate for real-time inputs from the data. ETA prediction models in Google Maps, for example, must update their predictions about expected arrival times in real time as new data is fed into the model continuously. Traffic routing models must respond quickly to the changes in the traffic patterns or network issues in real time. Modelling techniques that have a high training time might not be useful here.

Matching data complexity with model complexity: Once the modelling technique is chosen, one of the fine arts in data science is to pick the right complexity of the model. In other words, we must *match* the complexity of the model with the complexity of the data itself. If a more complex model is chosen, it might *memorize* the training data and may not *generalize* well to the unseen data. If a less complex model is chosen, it might not be able to capture the essential causal structure in the data. This principle of picking the right complexity of model is known by many names: bias–variance trade-off, signal-to-noise ratio, or Occam's razor. In essence,

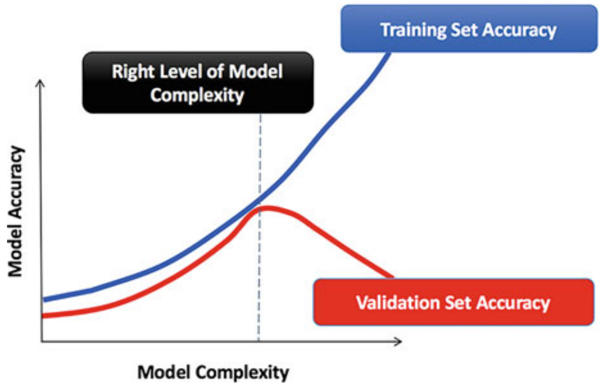


Fig. 16.5 Model complexity is chosen based on the gap between training and validation accuracy

the model needs to be *just complex enough and not any more*. In practice, the right model complexity for a given labelled data and modelling technique is arrived at as follows: We start with a simple model and increase its complexity slowly while measuring the training and test set accuracy—that is, how well it does on the data that was used to build the model and how well it does on the unseen data. As model complexity increases the training and test accuracies will go up. But beyond the point of peak generalization, the test accuracy will start falling as the model will start to learn the noise in the training data. This is a good indication of the right model complexity as shown in Fig. 16.5. Each modelling technique comes with a set of "knobs" to increase their model complexity.

From predictions to decisions: The output of a model is typically a score—for example, the credit score, the fraud score, or the predicted demand in a forecasting model. Machine learning stops where this score is generated. Data science starts where this score is now used to make decisions. Often a number of business constraints and metrics determine how the score should be used. For example, a bank with a higher risk appetite might give loans at a lower score than another. In recommendation engines, for example, we might not just recommend the product with the highest recommendation score but might decide to recommend products that are also highly connected to other products for increasing cross-sell beyond just the current recommendation. Decisions are made, often, with conflicting business metrics in mind and the model prediction outputs serve as key inputs to the overall business logic that tries to solve a complex multiobjective optimization actually make the final decision.

Feedback and continuous learning: Once the model is deployed, feedback is collected on how well it is doing. This feedback is critical for monitoring model performance and continuously updating the models. For example, search engines continuously update their models based on real-time click feedback data by moving the search results up or down based on whether they are getting higher or lower than expected clicks for the result at that position. This feedback data is the real goldmine in any modelling exercise. It is the cheapest and most consistent source of "ground truth" that is very critical for building supervised learning models. This feedback also comes in implicit form. For example, if the model predicted that a customer is about to churn but he/she did not or vice versa then such implicit feedback can be used to continuously improve the models. Modelling is therefore never a one-time exercise. Using this feedback data to automatically and periodically update the model really completes the "continuous learning" loop in real-time, large-throughput systems that evolve as the business processes, customer behavior, and environment evolves.

4.4 The Algorithms: Classification Models

If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.

In the rest of the chapter, we will focus primarily on the classification paradigm. We will assume that the raw data has already been transformed into a meaningful ***feature space*** as discussed above.

Definition of a classifier: Essentially, a classifier partitions the feature space into ***pure*** regions. A region is considered pure if most of the points in that region belong to the same class. There are two ways of characterizing pure regions: Either we learn to "describe" each class (the descriptive classifiers) or we learn to "discriminate" between the classes (the Discriminative Classifiers). Figure 16.6 shows how a descriptive vs. a discriminative classifier approaches the same two-class problem.

We seem to be using both classifiers: as we discover new objects in the world and see one or more examples of it, we build a descriptive classifier that learns the essence of the class. But when we are confused between two classes (e.g., "dog" vs "goat," letter "o" vs. "c"), that is, their descriptions "overlap" quite a bit, then

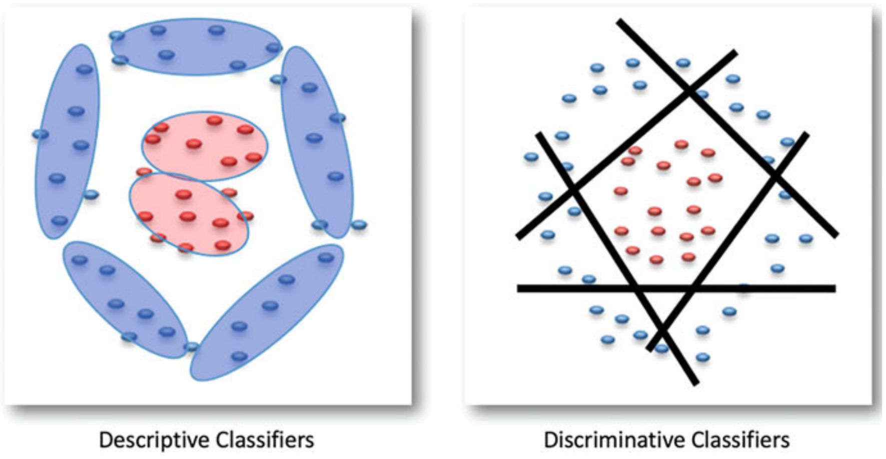


Fig. 16.6 A descriptive classifier learns the shape of each class. A discriminative classifier tries to find the decision boundaries between the classes

we fine-tune these descriptive models to discriminate between them. A number of both descriptive classifiers and discriminative classifiers are discussed below. *The website contains corresponding R code, data, examples, and exercises.*

Rule-Based Classifiers

Rule-based classifiers are the simplest, handcrafted, interpretable classifiers that codify existing knowledge into a set of rules of the form: *If (Condition) then Class.* Such rule-based *descriptive* classifiers occur in many domains including science and medicine (e.g., blood group classifiers (A, B, O, AB), obesity classifiers (underweight, normal, overweight, obese based on simple BMI thresholds), diabetes classifiers (type I vs. type II), symptoms-based disease classifiers (e.g., if fever >103 and throat infection and shivering, then viral infection), periodic table (valence-based element classifier into inert gases, heavy metals, etc.), organic vs. inorganic, hydrophilic vs. hydrophobic, acidic vs. alkaline, etc. Classification of species in a hierarchical fashion is an enormous rule-based classifier. Even businesses and financial institutions have been running for a long time on rule-based systems.

Rule-based classifiers are great at encoding human knowledge (Fig. 16.7). These rules can be simple or complex, depending on one or many features, and can be nested hierarchically such that the class prediction of one rule can become a feature to another rule at the next level in the hierarchy. Such a rule-based system, also known as an ***Expert System*** is the best way to bootstrap a data-starved, knowledge-rich process until it becomes data rich itself and rules can actually be learnt from data. One of the biggest advantages of rule-based systems is that they are highly interpretable and every decision they make for each input can be explained. Rule-based classifiers, however, have a few limitations: The knowledge that these rule-based systems contain may not be complete, adding new knowledge, updating obsolete knowledge, keeping all knowledge consistent in a large knowledge-base is an error-prone cumbersome process, human generated rule-bases might contain "subjective-bias" of the experts, and finally, not all knowledge is deterministic or binary—adding uncertainty or degrees to which a rule is true requires data/evidence.

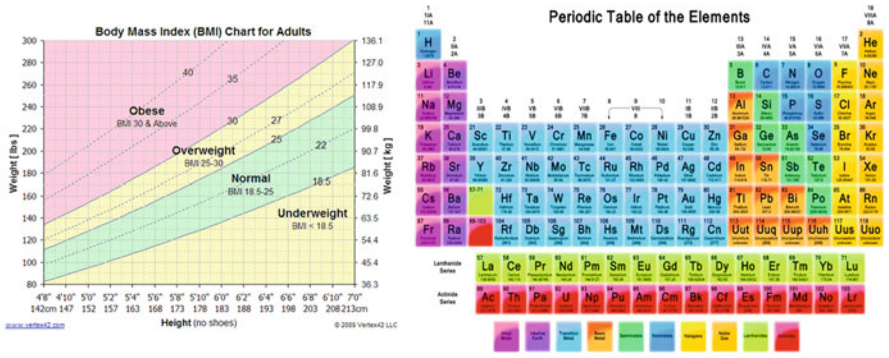


Fig. 16.7 Examples of simple rule-based classifiers in healthcare and nature

As data started to become more and more abundant and the rule-based systems started to become harder and harder to manage and use, a new opportunity of *learning rules from data* emerged. This led to the first algorithm—decision trees—that marked the beginning of machine *learning*. Decision trees combined the interpretability of rule-based classifiers with learnability of data-driven systems that do not need humans to handcraft the rules enabling the discovery of interactions among features that are far more complex for a human to encode.

Decision Trees Classifier

One of the earliest use cases of machine learning was to learn rules directly from data, adapt the rules as data changes, and enable us to even quantify the goodness of the rules given a dataset. Decision trees are an early attempt to learn rules from data. Decision trees follow a simple recursive process of greedily partitioning the feature space, one level at a time, discovering *pure* regions. A region is a part of the feature space represented by a node in the decision tree. The root-node represents the entire feature space.

Purity of a region: In a classification problem, a region is considered "pure" if it contains points only from one class and "impure" if it contains almost equal number of examples from each class. There are several measures of purity that have been used in various decision tree algorithms. Consider a region in the feature space that contains n_c points from class $c \in \{1, 2, \ldots, C\}$ for a C class classification problem. The class distribution $\mathbf{p} = \{p_c\}_{c=1}^{C}$is given by:

$$p_c = \frac{n_c + \lambda}{\sum_{c'=1}^{C} n_{c'} + C\lambda}$$

where λ is the "Laplacian smoothing" parameter that makes the distribution estimate more robust to small counts. We describe different measures of purity as a function of $\mathbf{p}$.

- **Accuracy**: The first measure of purity is accuracy itself. If the class label assigned to a region is its majority class then the accuracy with which data points in the region are correctly labelled is:

$$Purity_{ACC}\,(p_1, p_2, \ldots, p_C) = \max_c \{p_c\}$$

- **Ginni index**: In the accuracy measure of purity we *hard*-assign a region to its majority class. This can be brittle to noise in data. If a region is *soft*-assigned to class c with probability p_c then the *expected accuracy* of the region is called the Ginni index of purity:

$$Purity_{GINNI}\,(p_1, p_2, \cdots, p_C) = \sum_c p_c^2$$

- **Entropy**: An information theoretic measure of *impurity* of a distribution is its Shannon Entropy, which is highest (say 1) when the distribution is uniform and 0 when the entire probability mass is centered on one class. *Purity is an inverse of this entropy.*

$$\begin{aligned} Purity_{INFO}\left(p_1, p_2, \cdots, p_C\right) &= 1 - Entropy\left(p_1, p_2, \cdots, p_C\right) \\ &= 1 + \sum_c p_c^2 \log_C p_c \end{aligned}$$

Gain in purity: A decision tree recursively partitions the entire feature space into pure subregion using a greedy approach. At each node (starting from the root node), it finds the best feature with which to partition the region into subregions. The "best" feature is the one that maximizes the gain in purity of the subregions resulting from that feature.

Let us say node m is partitioned using feature f (e.g., COLOR) into $K_{m,f}$ children nodes (e.g., RED, GREEN, BLUE): $\left\{R_{f,1}^m, R_{f,2}^m, \ldots, R_{f,K_{m,f}}^m\right\}$. Let $Purity\left(R_{f,k}^m\right)$ be the purity of subregion $R_{f,k}^m$ and $p\left(R_{f,k}^m\right)$ be the fraction of data at m that goes to the subregion $R_{f,k}^m$. Then purity gain due to feature f at node m is:

$$PurityGain_m(f) = \sum_{k=1}^{K_{m,f}} p\left(R_{f,k}^m\right) \times Purity\left(R_{f,k}^m\right) - Purity(m)$$

Decision tree algorithm: Decision tree recursively partitions each region into subregions by picking that feature at each node that yields the maximum purity gain. Figure 16.8 shows a decision tree over a dataset over five variables {A, B, C, D, E}. Let us say variable A takes two possible values {A1, A2}, variable B takes two values {B1, B2 }, C takes two values {C1, C2}, D takes two values {D1, D2}, and E takes two values {E1, E2}. At the root node, the decision tree algorithm tries all the five variables and picks the one (in this case variable B) that gives the highest purity gain. The entire region is now partitioned into two parts: B = B1, and B = B2. Now that variable B has already been used, the remaining four variables are considered at each of these nodes. In this example, it turns out that under B = B2, variable A is the best choice; under node A = A2, variable C is the best choice; and under C = C1, variable E is the best choice among all the other choices within those regions. Variable D does not increase purity at any node.

The sample data "Decision_Tree_Ex.csv" and R code "Decision_Tree_Ex.R" are available on website.

A leaf node at any time in the growing process is considered for growing further: (1) Its depth (distance from the root node) is less than a depth-threshold, (2) its purity is less than a purity-threshold, and (3) its size (number of data points) is more than a size-threshold. These thresholds (Depth, Purity, and Size) control the *complexity* or

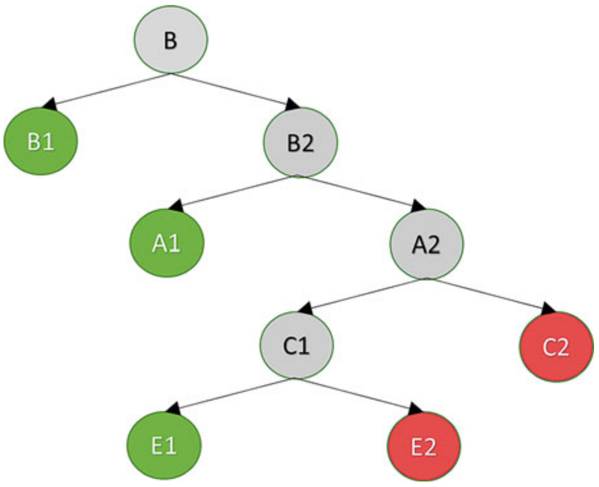


Fig. 16.8 A decision tree over a dataset with five features \{A, B, C, D, E\}

size of the decision tree. Different values of these thresholds might yield a different tree for the same dataset, but it will look the same from the root node onward. Sometimes, a tree is overgrown and pruned to a smaller tree as needed.

Decision trees were created to learn rules from data. A Decision Tree model can be easily written as a collection of highly interpretable rules. For example, the tree in Fig. 16.8 learns the five rules, one for each leaf node. Each rule is essentially an AND of the path from the root node to the leaf node.

- B = B1 ➔ Class = Green
- B = B2 and A = A1 ➔ Class = Green
- B = B2 and A = A2 and C = C2 ➔ Class = Red
- B = B2 and A = A2 and C = C1 and E = E1 ➔ Class = Green
- B = B2 and A = A2 and C = C1 and E = E2 ➔ Class = Red

Apart from interpretability, decision trees are also very deterministic—they generate the same tree given the same data—thanks to their greedy nature. This is essential for robustness, stability, and repeatability. The scoring throughput of decision trees is high. They just have to apply at most *D* conjunctions, where *D* is the depth of the tree. Apart from this, decision trees are also known to handle a combination of numeric and categorical features together. Numeric features at any node are partitioned into two by rules like Age <25. Finally decision trees handle missing data gracefully. They either ignore the missing features (so when a feature is missing the training data is ignored for that feature's purity computation) or assume the most likely value of that feature at that node (fine-grained imputation).

One of the key criticisms of decision trees is that they are not guaranteed to yield an optimal partition of the feature space due to their greedy nature. It is possible that a bad feature chosen early in the tree can lead to a pretty suboptimal sub-tree below that as there is no mechanism of "backtracking" and correcting for a bad-greedy choice made earlier. This is a classic example of the fundamental trade-off in AI between optimality and speed. Decision trees were originally designed for categorical features only. They handle numeric features using thresholding type rules—for example, if (temperature <100 degrees). This often limits them to partitioning the numeric subspace only along the numeric axes. If the required decision boundary is oblique, then decision trees land up learning staircase functions that could lead to very large trees. This can be addressed by learning logistic regression models at each internal node using all the numeric variables available at that node and using a threshold on that logistic model to partition it into two parts. This is a classic example of overcoming model limitations by combining them with other techniques.

Decision tree classifiers have evolved over the last few decades. Ensemble version of decision trees including Random Forest and XGBoost are commonly used for complex supervised learning problems. You can read more about decision trees in Chap. 3 of "Machine Learning" by Carbonell et al. (1983) or Chap. 8 of "Data Mining: Concepts and Techniques" by Han et al. (2011).

4.4.1 k-Nearest Neighbor Classifier (k-NN)

Often when we have to make important decisions, we take the advice of our *near* ones. We are more influenced by the opinions (about products, movies, restaurant, politics, etc.) of our friends, family, social circle, etc. than those of strangers. This principle that *nearby things have a higher influence than far-off things* is the essence of a whole family of algorithms starting with *k*-nearest neighbor (*k*-NN) classifier. In k-NN, the training data is stored as is and there is no modelling. Hence, this is an example of a ***nonparametric*** classifier. During the scoring phase, first the k-nearest neighbors in the training set (previously seen and labelled examples) are sought. Then the new example is assigned the majority class among these k-nearest neighbors as shown in Fig. 16.9. Let the two classes—blue triangle and red star—training data be stored as is. The new example—the green square—is classified as a blue triangle if k = 5 is chosen because in the top-5 neighbors of the new example, blue triangle is the majority class. While, for the case of k = 10, it is classified as red star class.

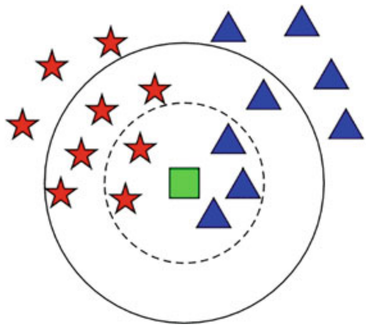


Fig. 16.9 k-NN classifier: the new example (green square) is classified based on the majority class in its top neighborhood (i.e., for k = 5, it is classified as blue triangle)

In k-NN, the hyperparameter k determines the “complexity” of the k-NN classifier. For a small *k* (e.g., 1), a new data point is assigned a class label with only very little “evidence” and will hence be very brittle to noise. For a large k (e.g., 17), we might get a smooth boundary but dependence on too many data points might “average out” the right structure in decision boundaries. Since there is no training, the training time for k-NN classifiers is zero; however, the space complexity of k-NN is high as it just stores the entire training data as “model.”

Scoring a new data point is also expensive as it involves computing its distance from all the training data points before finding the k-nearest neighbors. While advancements have been made to store the training data in ways that reduce the time complexity of scoring, still it renders k-NN not practical for high-throughput, real-time scoring. k-NN classifier is brittle to noise as its decision changes abruptly with majority classes. Choosing a high value of k makes it more noise-robust.

One of the key criticisms of k-NN classifier is that it loses information about the actual distance between the training and the test point once the training point is deemed to be within its k-Nearest. A true k-NN classifier should take distances into account wherever it matters. The distance function is the key to the k-NN classifier. For simple multivariate data, Euclidian distances (after proper transformations and normalizations) suffices but for nontrivial data types (e.g., LinkedIn profiles, Gene Sequences, Images, Videos), defining distances is yet another creative process. Read more about k-NN in Chap. 8 in “Machine Learning” by Carbonell et al. (1983) or Chap. 9 in “Data Mining: Concepts and Techniques” by Han et al. (2011).

Parzen Window Classifier (PWC)

In nature, each point mass has a gravitational field of influence. Similarly, each magnet has a magnetic field of influence both of which decrease as one moves away from the source of that influence. Similarly, each data point in a space has a “density field of influence” that decreases with distance. Parzen Window Classifier (PWC) uses this basic intuition to overcome some of the shortcomings of the k-NN classifier and makes a much more robust *softer variant* of the k-NN classifier. Unlike in k-NN where the distance between a training data point and a test data point is used only to *pick* the top k-NN points, PWC uses the actual distances for estimating the influence of *all* training data points on the test example. This makes PWC far more robust to noise and a “Soft variant” of the k-NN classifiers.

Let $\boldsymbol{X} = \{(\boldsymbol{x}_n, c_n)\}_{n=1}^{N}$ denote the training set containing N labelled examples. Let us assume that there is a Kernel field (K) around each of these training points. Now for a test point $\boldsymbol{x}$ we first compute the total influence it is receiving from all the points in each class, c. Influence of a data point $\boldsymbol{x}_n$ on $\boldsymbol{x}$ is given by a kernel function $K_\sigma(\|\boldsymbol{x} - \boldsymbol{x}_n\|)$ that decreases as $\boldsymbol{x}$ goes away from $\boldsymbol{x}_n$ and integrates to one.

$$P(\boldsymbol{x}\,|c)\,\alpha\frac{1}{N_c}\sum_{n-1}^{N}\delta\,(c = c_n)\,K_\sigma\,(\|\boldsymbol{x} - \boldsymbol{x}_n\|)$$

The point is then assigned to the class whose cumulative influence on it is maximum.

The complexity of the PWC is controlled by the hyperparameter σ that essentially captures the "spread" of the kernel field. If the field associated with each data point is too wide, then we learn "coarse" decision boundaries that might be very robust to noise but not capture the actual shape of the decision boundaries. On the other hand, if σ is too low, we might land up getting very jagged boundaries that are easily influenced by only a handful of nearby training data points. A low k in k-NN is equivalent to a low σ in PW classifier. As in k-NN, since here also we are not really training a model but just storing the training data as is, there is no training complexity in PW classifier. The scoring complexity, similar to k-NN classifiers is high since to score a new data point, its distance from all the training examples must be computed.

While PWCs address some of the robustness to noise issues of k-NN classifiers, they still do not *learn* anything (nonparametric) and have to depend on the entire training data all the time. Refer to Chap. 14 in "Machine Learning—A Probabilistic Perspective" by Murphy (2012) or Chap. 6 in "The Elements of Statistical Learning" by Friedman et al. (2001) to learn more about PWC.

4.4.2 Bayesian Classifier

The key to PW classifier is the *density function* $P(\boldsymbol{x}|\,c)$ that essentially quantifies whether the point $\boldsymbol{x}$ "looks like" previously seen points that belong to class c. PW *aggregates the influence* of all training points in class c on $\boldsymbol{x}$ to estimate $P(\boldsymbol{x}|\,c)$. But as humans, we do not classify by first remembering all previous examples of each class and comparing a new example with them. We, on the other hand, build a "representation" of each class by *summarizing* or *describing the essence* of all the data per class into a class "model."

In Bayesian Classifiers, each class c is modelled by (a) its *class prior* $P(c)$ that quantifies the probability that an unseen data point would belong to class c and (b) the *class conditional density function* $P(\boldsymbol{x}|\,c)$ that quantifies the probability of having seen "such" a data point from class c in the training data. Unlike in PW, in Bayesian Classifiers, $P(\boldsymbol{x}|\,c)$ is *modelled* (and not just *computed*) using a parametric density function that takes into account the nature of the data (e.g., multivariate, text, speech) as well as the parametric form used to model it (e.g., Normal distribution). The class prior and class conditional density functions are learnt from the training data. They are then used to compute the *class posteriori probability* $P(c|\,\boldsymbol{x})$ over all the classes c for a new data point $\boldsymbol{x}$. This is done by using one of the most celebrated relationships in statistics and probability theory—a relationship between cause and effect, between learning and scoring, between past observations and future predictions, and between data and knowledge—the **Bayes Theorem**:

$$P\left(c|\boldsymbol{x}\right) = \frac{P(c)P\left(\boldsymbol{x}|c\right)}{P\left(\boldsymbol{x}\right)} = \frac{P(c)P\left(\boldsymbol{x}|c\right)}{\sum_{c'} P\left(c'\right) P\left(\boldsymbol{x}|c'\right)}$$

While computing class priors $\boldsymbol{P}(\boldsymbol{c})$ is straightforward—all we have to do is normalize the counts of each class in the data, it is in modelling the class conditional density function $\boldsymbol{P}(\boldsymbol{x}|\,\boldsymbol{c})$ where a Bayesian Data Scientist spends most of his time. We give a flavor of a few common density functions below.

Unimodal Bayesian classifier (UBC): The simplest Bayesian classifier on multivariate numeric data models each class c as a unimodal (assuming normally distributed) cloud, centered around a mean $\boldsymbol{\mu_c}$ and with a certain covariance $\boldsymbol{\Sigma_c}$ that captures the shape of the cloud, that is, $\boldsymbol{P}(\boldsymbol{x}|\,\boldsymbol{c}) = \mathbf{N}(\boldsymbol{x}|\,\boldsymbol{\mu_c}, \boldsymbol{\Sigma_c})$. The mean and covariance are computed by maximizing the log-likelihood of the class data:

$$\boldsymbol{\mu_c} = \frac{1}{N}\sum_{n=1}^{N} \delta\left(c_n = c\right) x_n \text{ and } \boldsymbol{\Sigma}_c = \frac{1}{N}\sum_{n=1}^{N} \delta\left(c_n = c\right) \left(x_n - \mu_c\right)\left(x_n - \mu_c\right)^T$$

Unimodal Bayesian classifier and PW are two extreme ways of estimating the same statistic: $\boldsymbol{P}(\boldsymbol{x}|\,\boldsymbol{c})$. In nonparametric PW each training data point is associated with a Gaussian kernel of a certain width around it. In the parametric unimodal Bayesian classifier, all the data points associated with a class are "described" using a single Gaussian—in terms of its mean and covariance parameters.

Multimodal Bayesian classifier (MBC): Clearly there is a continuum of complexity from PW—that uses one Gaussian per data point—a potential overkill to UBC—that uses one Gaussian per class—which might not be sufficient to describe the class. In many domains, a class might be multimodal, that is, it might have subclasses. For example, the same word might have two very different pronunciations in different accents. An object in image might look very different in different pose, illumination, and scale. A letter in OCR might have different fonts and emphases (bold, italics, etc.). In handwritten digits, for example, people write a digit "7" or "1" or "9" in different ways. In all such cases, the entire class cannot be modelled as a single Gaussian but as a mixture-of-Gaussians (MoG), that is, two or more Gaussians—one representing each subclass. Figure 16.10 shows a two-class problem data where using one Gaussian per class (left) might yield a low accuracy classifier (under trained) but using three Gaussians per class might be the right level of complexity for this dataset. In general, MoG is a generic and powerful way of modelling arbitrarily complex density functions to match complexity of data (number of subclasses per class). An MoG is written as:

$$\boldsymbol{P}\left(\boldsymbol{x}|\boldsymbol{c}\right) = \sum_{k=1}^{M_c} \pi_k^{(c)} N\left(\boldsymbol{x}|\mu_k^{(c)}, \Sigma_k^{(c)}\right)$$

where:

- $\pi_k^{(c)}$ is the prior proportion of subclass k in class c.
- $\mu_k^{(c)}$ is the mean of subclass k of class c.
- $\Sigma_k^{(c)}$ is the covariance of subclass k of class c.

These parameters are learnt using the EM algorithm using the data from each class independently. The number of mixture components for each class can vary depending on the number of subclasses it might have. The EM algorithm for learning MoG is described in the unsupervised learning chapter. The unimodal

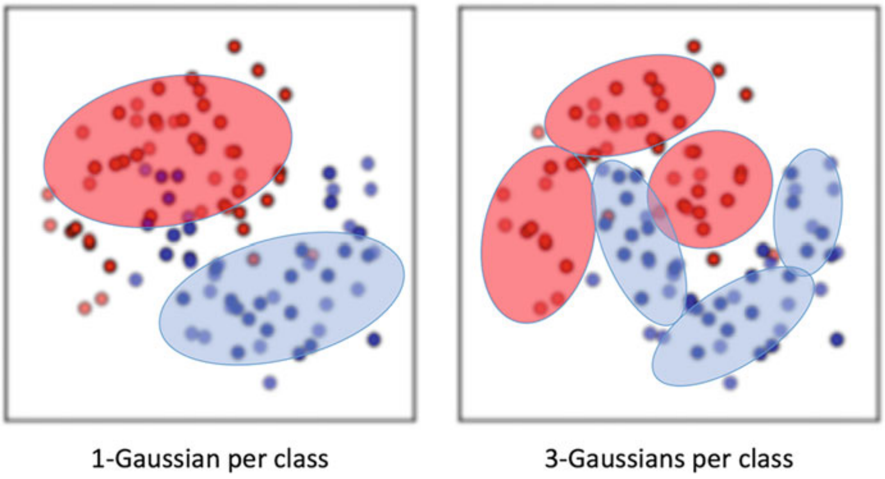


Fig. 16.10 Using 3 instead of 1 Gaussian per class: matching data complexity with model complexity

Gaussians, mixture-of-Gaussians, and Parzen window form a continuum of models for modelling each class with a single, multiple, and maximum possible Gaussians.

For more details, the interested reader can refer to Chap. 8 in "The Elements of Statistical Learning" by Friedman et al. (2001) or Chap. 8 in "Data Mining: Concepts and Techniques" by Han et al. (2011).

Naïve Bayes classifier: One of the fundamental problems in estimating density functions in particular and learning robust ML models in general is the *curse of dimensionality*—as the dimensionality of the feature space increases, the volume of data needed to get the same level of robustness increases exponentially. In cases where the number of features is large and the amount of data we have is not sufficient to populate each region in the "joint" space of all features, a *naïve* yet practical assumption is made that *all features are conditionally independent, given the* class. This results in the following simplification of the class conditional density function:

$$\boldsymbol{P}\left(\boldsymbol{x}|\boldsymbol{c}\right) = \prod_{\boldsymbol{d}=\mathbf{1}}^{\boldsymbol{D}} \boldsymbol{P}\left(\boldsymbol{x}_{\boldsymbol{d}}|\boldsymbol{c}\right)$$

where D is the number of dimensions. This significantly reduces the number of parameters. In case of discrete data with C classes, D features each of which taking M values, the total number of parameters for computing the joint probability density function is $\boldsymbol{O}(\boldsymbol{CM^D})$. The amount of data needed to have sufficient statistics in each cell would also be enormous. A naïve Bayes classifier on this data only needs to estimate $\boldsymbol{O}(\boldsymbol{CMD})$ parameters.

A common application for Naïve Bayes classifiers is in text classification where the number of dimensions, that is, words, phrases, bigrams, and trigrams can be very large compared to the size of the labelled corpus. A document is represented as a *bag-of-words* where each document $\boldsymbol{x}$ is represented by the number of times each word $\boldsymbol{w_d}$ occurs in the document—the term frequency: $\boldsymbol{tf}(\boldsymbol{w_d}|\,\boldsymbol{x})$. In the training phase, class conditional probabilities of each word are computed from labelled data as follows:

$$\boldsymbol{P}\left(\boldsymbol{w_d}|\boldsymbol{c}\right) = \frac{\boldsymbol{n}\left(\boldsymbol{w_d}, \boldsymbol{c}\right) + \boldsymbol{\lambda}}{\sum_{\boldsymbol{w}} \boldsymbol{n}\left(\boldsymbol{w}, \boldsymbol{c}\right) + \boldsymbol{D\lambda}}$$

Here $\boldsymbol{n}(\boldsymbol{w}, \boldsymbol{c}) = \sum_{\boldsymbol{x} \in c} \boldsymbol{tf}(\boldsymbol{w}|\,\boldsymbol{x})$ is the number of times word $\boldsymbol{w}$ occurs in class c, D is the total number of words in the dictionary, and $\boldsymbol{\lambda}$ is the Laplacian smoothing constant used to make sure that none of the $\boldsymbol{P}(\boldsymbol{w_d}|\,\boldsymbol{c})$ becomes 0.

These estimates can be used to compute $\boldsymbol{P}(\boldsymbol{x}|\,\boldsymbol{c})$. A new document $\boldsymbol{x}$ is classified by first computing its class conditional probability density. But since the document lies in a high-dimensional (number of unique words after preprocessing) sparse space (each document only contains a very small fraction of the total words in the dictionary), it is not possible to model the density in the joint space. We therefore make a naïve assumption that all words are independent given the document belongs to a certain class:

$$\boldsymbol{P}\left(\boldsymbol{x}|\boldsymbol{c}\right) = \prod_{\boldsymbol{d}=\boldsymbol{1}}^{\boldsymbol{D}} \boldsymbol{P}(\boldsymbol{w_d}|\boldsymbol{c})^{\boldsymbol{tf}(\boldsymbol{w_d}|\boldsymbol{x})}.$$

Each occurrence of each word $\boldsymbol{w_d}$ in the document is multiplied to itself $\boldsymbol{tf}(\boldsymbol{w_d}|\,\boldsymbol{x})$ times. The Bayes rule is then used to compute the posterior probability of a class given a new document. To prevent numerical underflow issues, instead of computing $\boldsymbol{P}(\boldsymbol{c}|\,\boldsymbol{x})$ we compute:

$$\ln \boldsymbol{P}\left(\boldsymbol{x}\,|\boldsymbol{c}\right) \alpha \ln \boldsymbol{P}\left(\boldsymbol{c}\right) + \sum_{\boldsymbol{d}} \boldsymbol{tf}\left(\boldsymbol{w_d}\,|\boldsymbol{x}\right) \ln \boldsymbol{P}\left(\boldsymbol{w_d}\,|\boldsymbol{c}\right)$$

where the denominator is left out as it not involved in the classification.

As k-NN classifiers are a good baseline for numeric multivariate classification problems, Naïve Bayes classifiers are a good baseline for text classification problems. Feature engineering in bag-of-words representation of text data involves (1) removing stop words (e.g., articles), (2) doing stemming (so all variants of a word, e.g., “run,” “running,” “runs,” “ran,” are mapped to the root word “run”) and before stemming, adding higher-order words (e.g., bigrams, trigrams, or better yet adding phrases discovered through other means). One might also move from bag-of-*words* to bag-of-*topics* representation as discussed in Chap. 15 on unsupervised machine learning).

Bayesian classifiers are robust to data and feature noise, they can adapt to feature covariance within each subclass, incorporate class priors systematically where needed, and are well grounded in theory. The creativity in Bayesian classifiers is in learning class conditional probability density function $P(\boldsymbol{x}|\,c)$.

Discriminant analysis: Earlier we defined classification as the art of partitioning the feature space into pure regions. We can achieve this in two ways. Either by describing each pure region as is done by Bayesian classifiers via class conditional probability density functions $P(\boldsymbol{x}|\, c)$ or equivalently by characterizing the boundaries between two pure regions that *discriminate* the two classes. Here we will explore the descriptive to discriminative transition through Discriminant Analysis.

The decision whether $\boldsymbol{x}$ belongs to one class or the other depends on which of the two posterior probabilities is higher. In this sense $g_c(\boldsymbol{x}) = \boldsymbol{P}(c|\,\boldsymbol{x})$ is called a *discriminant function* and any monotonic variant of this is also a discriminant function. Simplifying this we get:

$$g_c\,(\boldsymbol{x}) = \boldsymbol{P}\,(c|\boldsymbol{x}) \propto \ln \boldsymbol{P}\,(c|\boldsymbol{x}) \propto \ln P\,(\boldsymbol{x}|c) + \ln P(c)$$

The *decision boundary* between two classes (assuming a two-class problem for simplicity) is the locus of all points $\boldsymbol{x}$ where the two posterior probabilities are same, that is, where the two regions intersect or where the points cannot be classified in one class or the other: $P(c_1|\,\boldsymbol{x}) = P(c_2|\,\boldsymbol{x})$. The discriminant classifiers label a data point into the maximum discriminant value class: $c^*\,(\boldsymbol{x}) = \arg\max_c \{g_c\,(\boldsymbol{x})\}$. The decision boundary can be derived by solving: $g_1(x) = g_2(x)$ or $g_1(x) - g_2(x) = 0$.

Linear discriminant analysis (LDA): In a two-class problem, if we make the assumption that the covariance (shapes of the Gaussians) of the two classes are the same, that is, $\Sigma_1 = \Sigma_2 = \Sigma$ then the decision boundary is given by: $\ln P(\boldsymbol{x}|\, c_1) + \ln P(c_1) - \ln P(\boldsymbol{x}|\, c_2) - \ln P(c_2) = 0$, which when simplified leads to a *linear decision boundary*: $\boldsymbol{w}^T\boldsymbol{x} + w_0 = 0$, where:

$$\boldsymbol{w} = \boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2) \text{ and } w_0 = \frac{1}{2}\,(\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)\,\Sigma^{-1}\,(\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1) + \ln \frac{P(c_1)}{P(c_2)}$$

Quadratic discriminant analysis (QDA): If we allow the two covariance matrices to be arbitrary, that is, the classes take any shape possible within the unmodal constraints, then the decision boundaries become more quadratic: $\boldsymbol{x}^T\boldsymbol{W}_2\boldsymbol{x} + \boldsymbol{w}_1^T\boldsymbol{x} + w_0 = 0$. Figure 16.11 shows the linear and quadratic discriminant decision boundaries:

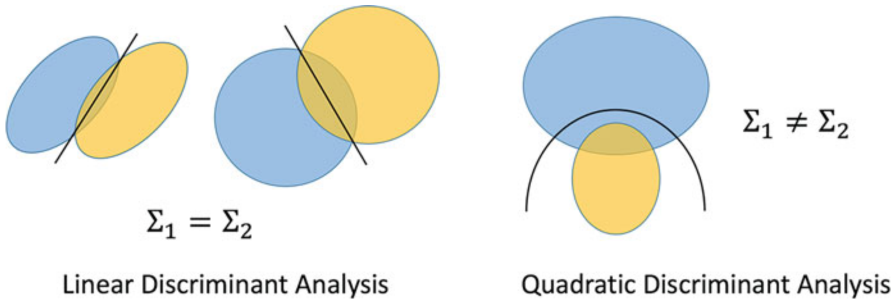


Fig. 16.11 Linear vs. quadratic discriminant analysis (LDA vs. QDA)

LDA and QDA are a bridge between descriptive and discriminative classifiers. The decision boundary in LDA and QDA is simply *computed*—in terms of class prior, mean, and covariance properties—but not *learnt*. Hence, they are still fundamentally descriptive classifiers yet a bridge between the shape and the boundary of the class.

You may read more about LDA and QDA in Chap. 4 in "Machine Learning—A Probabilistic Perspective" by Murphy (2012) or Chap. 4 in "The Elements of Statistical Learning" by Friedman et al. (2001).

Perceptron: One of the earliest discriminative classifiers is a **perceptron**—a simple, biologically inspired, functional model of what we believe the neuron in the brain does. Our brain contains billions of neurons, each connected to thousands of other neurons both laterally (within the same layer) and hierarchically (across layers). Each neuron does, more or less, functionally the same thing—it aggregates the inputs received from incoming neurons (connected to its dendrites), attenuates the aggregated activation, and makes it available at its axons to pass on to its "children" neurons. While a neuron sitting at the lower layer (e.g., on the retina of the eye) might take raw pixel level input and combine them to detect lines, the *face detecting neuron* at much higher up in the visual cortex hierarchy might be taking inputs from *eye detecting* neurons, *nose detecting* neurons, *mouth detecting* neurons, etc. as inputs and predict whether it is "seeing" a face. The simplicity of each neuron combined with the complexity with which they are arranged and work together makes the brain one of the most mysterious and powerful masterpieces of evolution. This also forms the basis of the modern deep learning paradigms that use a variety of neurons and deep layered architecture to replicate some of the most complex human brain capabilities of vision, speech, and text understanding. All of this complexity starts with the "transistors of the brain"—the neurons (Fig. 16.12).

The basic perceptron algorithm that captures the early essence of a neuron for a two-class problem is very simple. Here, let $\boldsymbol{P}$ and $\boldsymbol{N}$ be the set of positive and negative examples, respectively. Let $\boldsymbol{w}_t$ be the weights of the perceptron in iteration t, initialized randomly (imagine the neurons of a newborn baby that has never seen any data yet but has this powerful infrastructure to learn a hierarchical representation

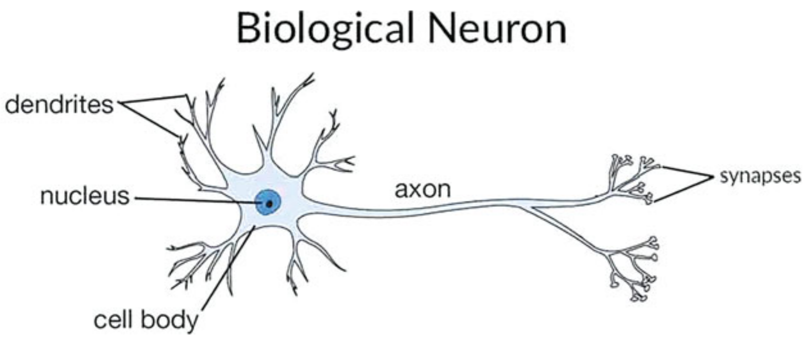


Fig. 16.12 A neuron—the building block of the brain

of the world he/she is about to interact with). Then the perceptron algorithm updates these weights iteratively by (1) sampling a data point, (2) classifying it into one of the two classes based on its current weights, (3) determining whether it has classified it correctly or not given the class label associated with the data point, and (4) update its weights if it made a mistake in the classification:

- Sample a data point $\boldsymbol{x} \in \boldsymbol{P} \cup \boldsymbol{N}$
- If $\boldsymbol{x} \in \boldsymbol{P}$ and $\boldsymbol{w}_t.\, \boldsymbol{x} \leq 0$ then: $\boldsymbol{w}_{t+1} \leftarrow \boldsymbol{w}_t + \boldsymbol{x}$; $t \leftarrow t+1$
- If $\boldsymbol{x} \in \boldsymbol{N}$ and $\boldsymbol{w}_t.\, \boldsymbol{x} \geq 0$ then: $\boldsymbol{w}_{t+1} \leftarrow \boldsymbol{w}_t - \boldsymbol{x}$; $t \leftarrow t+1$

The perceptron "converges" if we either reach a maximum number of iterations or better yet when no more examples are wrongly classified by the perceptron. Perceptron-based classifiers are mostly useful for two-class problems, they are not very robust to noise, they learn in an online fashion and therefore very sensitive to the order in which the data is presented, and finally they make a *hard* decision—if a point is on the correct side—no matter how far, they will try to self-correct—rendering them brittle. Perceptron is equivalent to k-NN classifier, which is also a hard classifier sharing some of the similar problems that perceptron-based classifiers have.

Logistic Regression: One of the oldest, time-tested, discriminative classifiers in machine learning is Logistic Regression. On the one hand, it is the softer version of the perceptron (pretty much like the Parzen Windows is a softer version of the k-NN and mixture-of-Gaussians is the softer version of k-Means clustering); on the other hand, it is the nonlinear version of linear regression. It models the log-odds ratio of the target class vs. the background class as a linear combination of the inputs.

$$\ln\left(\frac{P\left(Y=1|\boldsymbol{x}\right)}{P\left(Y=0|\boldsymbol{x}\right)}\right) = \boldsymbol{w}^T\boldsymbol{x} \Longrightarrow P\left(Y=1|\boldsymbol{x}\right) = \frac{1}{1+\exp\left(-\boldsymbol{w}^T\boldsymbol{x}\right)}$$

where $\boldsymbol{w} \in R^{D+1}$ is the set of $D + 1$ parameters including the constant bias term.

Most machine learning is optimization. Every parametric modelling technique optimizes an objective function written in terms of the data (or some statistics on the data) and some parameters. Clustering, for example, minimizes the distance between a data point and *its* cluster center, Fisher discriminant maximizes separation between classes, decision trees try to split a leaf node into the purest possible subregions, and perceptron tries to minimize misclassification error, etc. Modelling is essentially the art of formulating and solving an objective function. Sometimes, the solution is closed form (PCA, Fisher, LDA, QDA, etc.), sometimes it is greedy (e.g., decision trees), and sometimes it is iterative (e.g., perceptron). The objective function too can take multiple forms. Sometimes it is a variant of the sum-squared-error (e.g., K-means clustering), sometimes it is maximizing (log) likelihood of seeing the data.

Logistic regression is the solution of a maximum log-likelihood objective function:

$$J\left(\boldsymbol{w}\right) = \ln \prod_{n=1}^{N} P(Y=1|\boldsymbol{x}_n)^{y_n} P(Y=0|\boldsymbol{x}_n)^{1-y_n}$$

Substituting logistic function for $P(Y = 1|\boldsymbol{x}_n)$ and optimizing for $\boldsymbol{\theta}$ yields the following update rule:

$$\boldsymbol{w}_t \leftarrow \boldsymbol{w}_{t-1} + \eta \sum_{n=1}^{N} \left(y_n - P\left(Y = 1\left|\boldsymbol{x}_n\right.\right)\right) \boldsymbol{x}_n$$

where $\boldsymbol{w}_t$ is the weight parameters at iteration t initialized at $t = 0$ to low random values and η is the learning rate that needs to be carefully set for fast convergence with stability. A logistic regression converges when these weights stop changing substantially. Note that when learning rate η is 1 and $P(Y = 1|\boldsymbol{x}_n)$ is either 1 or 0 (hard), the above update rule reduces to that of perceptron algorithm.

Logistic regression is a highly interpretable classifier. The weights (both sign and magnitude) learnt by logistic regression reveal a lot about the nature of relationship between the corresponding feature and the binary output label. The complexity of a logistic regression is controlled by penalizing the magnitude of each weight, that is, no single weight is allowed to dominate the overall decision. This is done by adding a penalty term: $\lambda\|\boldsymbol{w}\|^2$ to the objective function.

One of the limitations of Logistic Regression is that it can only learn linear relationship between input and output. To learn nonlinear decision boundaries using logistic regression, we can add higher-order terms $(x_i x_j)$, ratios (x_i/x_j), log transforms $(\ln x_i)$, and other complex functions (e.g., $(x_i^2/(x_j + x_k))$) of the input features. In fact, adding all variants of a variety of such transform (all second-order or third-order terms, log transforms of all features, etc.) and its ability to pick the right combination from this large pool is what makes them simple, powerful, flexible, and highly versatile modelling paradigm. This ability to add new variables is also known as generalized linear models where instead of just considering the raw features, any transformations of raw features can be used as inputs.

$$\ln\left(\frac{P\left(Y=1|\boldsymbol{x}\right)}{P\left(Y=0|\boldsymbol{x}\right)}\right) = \boldsymbol{w}^T \boldsymbol{\Phi}\left(\boldsymbol{x}\right) \Longrightarrow P\left(Y=1|\boldsymbol{x}\right) = \frac{1}{1+\exp\left(-\boldsymbol{w}^T \boldsymbol{\Phi}\left(\boldsymbol{x}\right)\right)}$$

where $\boldsymbol{\Phi}(\boldsymbol{x})$ is a potentially very large number of features obtained from $\boldsymbol{x}$.

$$\boldsymbol{\Phi}\left(\boldsymbol{x}\right) = \left(\ldots x_i \ldots, \ldots x_i x_j \ldots, \ldots x_i x_j x_k \ldots, \ldots \ln x_i \ldots\right)$$

The other limitation of Logistic Regression is that it can only be used with numeric features. When the dataset contains both numeric and non-numeric features, we can use a 1-hot-encoding (e.g., if R, G, B are three colors then R can be represented by a vector (1 0 0), B with (0 1 0) and G with (0 0 1)) to create a multivariate representation out of these symbolic data. In spite of all the flexibility and simplicity, the onus of "engineering" features to capture nonlinearity in the input-output mapping is still on the modeller. In other words, logistic regression just learns the mapping given the features. It does not learn the features themselves. In that sense, it is only *partially* intelligent. A truly intelligent system should be able to figure out arbitrary relationships between the input and output without us having to even partially engineer that knowledge via feature engineering or guess work it through hit and trial. In other words, we need a *universal function approximator* that can *learn* any arbitrary mapping—both features and decision boundaries, no just latter given former.

Neural Networks: One of the most important breakthroughs in machine learning was precisely such a universal function approximator—an artificial neural network (ANN)—that can learn arbitrary mappings given enough training data, computational resources, and model complexity. Consider the two datasets in Fig. 16.13, each being a two class classification problem: red squares vs blue circles. Clearly, a single logistic regression classifier cannot solve this problem in this feature space. As discussed earlier, there are two mindsets to address this mismatch between data complexity and model complexity:

In the **feature-centric mindset**, the raw input features are first transformed into new features (e.g., $x_3 = \max\{x_1(1-x_2), (1-x_1)x_2\}$ (left dataset) or $x_3 = (x_1 - a_1)^2 + (x_2 - a_2)^2$ (right dataset), where (a_1, a_2) is the center of the entire data). While transforming features obviates the need for a complex model, in most real-world problems, it is not clear which and how many such transformed features are needed to make the class separation well behaved or linearly separable.

In the **model-centric mindset**, the problem of learning both the features and the decision boundaries is solved *simultaneously.* Neural networks are a canonical example of this mindset where the overall model is composed of multiple layers such that the layers closer to the input features try to learn better features (e.g., the lines shown in Fig. 16.13 are logistic regression lines representing binary features—indicating which side of the line a data point is on) while the layers closer to the output try to learn the decision boundaries (e.g., in the left case, the final output is red class for a data point that is above the lower line and below the upper line). The beauty of neural networks is that all neurons seem to be doing the same thing—trying to partition the feature space using hyperplanes—but the role they are playing depends on the layer in which they occur and the input they see from the previous layer. ANNs are biologically inspired ML models. Our sensory-brain-mortar system is organized in layers of neurons with both feed forward connections from lower layers (e.g., neurons in our retina) to higher layers (e.g., neurons responsible for recognizing faces) and feedback connections going from higher layers (whole) to lower layers (part).

Neural Network Architecture: In the simplest neural network architecture—a fully connected, feed forward artificial neural network—neurons are organized in $L+1$ layers. Layer 0 is the *input* layer and layer L is the *output* layer with $L-1$ *hidden* layers in between. Let N_ℓ be the number of neurons in layer ℓ. Figure 16.14 shows a neural network with one input layer (grey) with $N_0 = 2$ input units, $N_1 = 5$ hidden units, and $N_2 = 2$ output units.

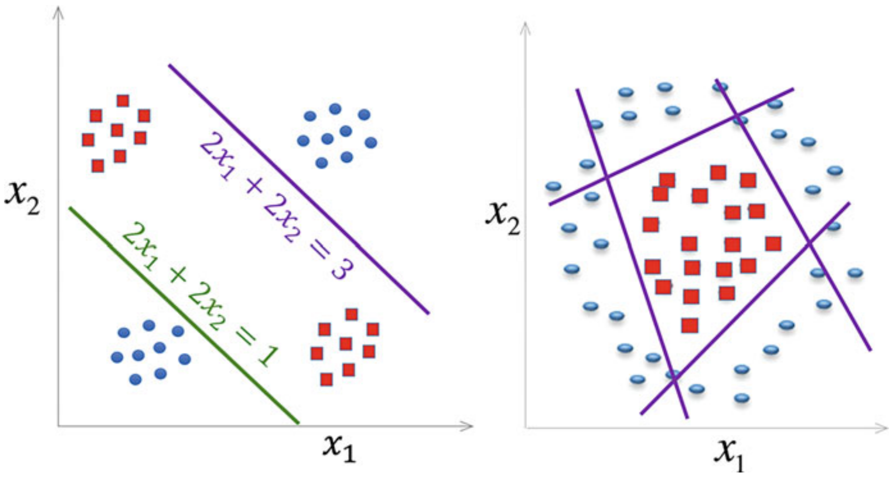


Fig. 16.13 A neural network classifier on two datasets

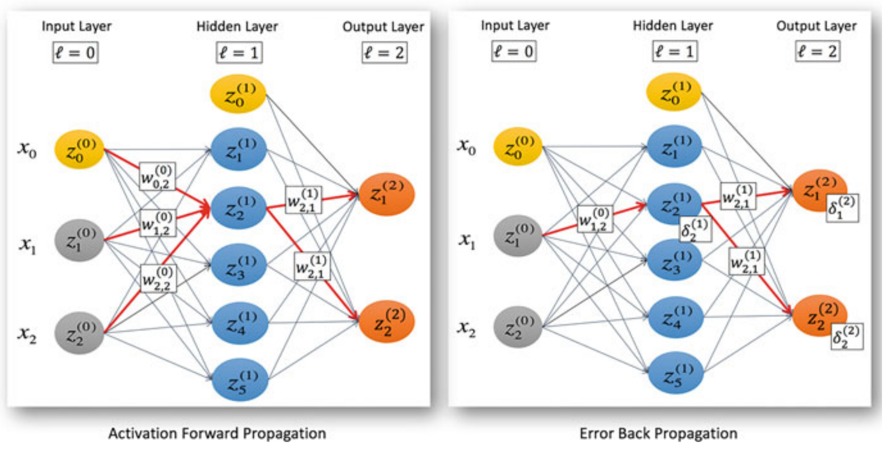


Fig. 16.14 A neural network with one hidden layer

In the **Activation Forward Propagation** (Fig. 16.14, left), the input layers are activated by the input data $\boldsymbol{z}^{(0)} = \boldsymbol{x}$; these activations travel to the subsequent layers to eventually activate the output layer.

Each hidden unit first aggregates the activations of all previous layer neurons and applies the bias term:
$a_i^{(\ell+1)} = w_0^{(\ell)} + \sum_{i=1}^{N_\ell} w_i^{(\ell)} z_i^{(\ell)} = \sum_{i=0}^{N_\ell} w_i^{(\ell)} z_i^{(\ell)}$ (where $z_0^{(\ell)}$, the bias term is always set to 1).

It then transforms these aggregates nonlinearly to generate activations of these hidden units:

$$Z_k^{(\ell+1)} = g\left(a_i^{(\ell+1)}\right) = g\left(w_0^{(\ell)} + \sum_{i=1}^{N_\ell} w_i^{(\ell)} z_i^{(\ell)}\right) = g\left(\sum_{i=0}^{N_\ell} w_i^{(\ell)} z_i^{(\ell)}\right)$$

These activation functions provide nonlinearity and limit the aggregate in a certain range. Examples of some of the activation functions are:

- Sigmoid/Logistic: $g(a) = \frac{1}{1+\exp(-\lambda a)}$ (between [0,1]).
- Hyperbolic: $g(a) = \tanh(\lambda a) = \frac{\exp(\lambda a)-\exp(-\lambda a)}{\exp(\lambda a)+\exp(-\lambda a)}$ (between [−1,1])
- Soft Max: $g(a_i) = \frac{\exp(\lambda a_i)}{\sum_{j=1}^{N_L}\exp(\lambda a_j)}$ (used in output layers for learning posterior probabilities)

The final activations in layer L, $\boldsymbol{z}^{(L)} = \widehat{\boldsymbol{y}}$ is the output that the neural network predicts for the input $\boldsymbol{x}$ for the current set of weights. During the training phase, enough of these input-output pairs are given.

It is in the **Error Back Propagation** (Fig. 16.14, right) that the real magic of neural network learning happens, where the weights are updated using an **Error Backpropagation Algorithm**, perhaps one of the most important algorithms in ML. For a given labelled data ($\boldsymbol{x},\boldsymbol{y}$), it minimizes the squared error $E(\boldsymbol{W}) = ||\boldsymbol{y} - \widehat{\boldsymbol{y}}(\boldsymbol{W})||^2$ between the expected $\boldsymbol{y}$ and the actual $\widehat{\boldsymbol{y}}$ output. This error is *back propagated* from layer ℓ to layer $\ell - 1$ by updating the weights $w_{i,j}^{(\ell)}$ as follows:

$$\Delta w_{i,j}^{(\ell)} \propto -\frac{\partial E(\boldsymbol{W})}{\partial w_{i,j}^{(\ell)}} = -\left(\frac{\partial E(\boldsymbol{W})}{\partial a_j^{(\ell+1)}}\right)\left(\frac{\partial a_j^{(\ell+1)}}{\partial w_{i,j}^{(\ell)}}\right) = -\delta_j^{(\ell+1)} z_i^{(\ell)}$$

In other words, the update in the weight $w_{i,j}^{(\ell)}$ is proportional to the activation on its input neuron $z_i^{(\ell)}$ at the error at its output neuron $\delta_j^{(\ell+1)}$. The negative sign indicates that we are trying to minimize the error. The error $\delta_j^{(L)}$ for the output layer is simply:

$$\delta_j^{(L)} = \frac{\partial E(\boldsymbol{W})}{\partial a_j^{(L)}} = \frac{\partial E(\boldsymbol{W})}{\partial z_j^{(L)}}\frac{\partial z_j^{(L)}}{\partial a_j^{(L)}} = \left(z_j^{(L)} - y_j\right) g'\left(a_j^{(L)}\right)$$

The error $\delta_j^{(\ell)}$ for all the other layers is given by:

$$\delta_i^{(\ell)} = \frac{\partial E(\boldsymbol{W})}{\partial a_i^{(\ell)}} = g'\left(a_i^{(\ell)}\right)\sum_{j=1}^{N_{\ell+1}} w_{ij}^{(\ell)} \delta_j^{(\ell+1)}$$

In other words, the total error in each of the nodes in the next layer $\delta_j^{(\ell+1)}$ propagates backward in proportion to the weight $w_{ij}^{(\ell)}$ to form the total the error $\delta_i^{(\ell)}$at this node.

A wide variety of applications, architectures, and heuristics have been proposed in the last couple of decades that have made neural networks one of the most common and powerful machine learning algorithms especially in two cases: (1) where there is sufficient data to train large networks and (2) where accuracy is more important than interpretability. Recent advances in deep learning have carved a special place for neural networks and their variants—recurrent neural networks (CNN), auto-encoders, convolution neural networks (CNN), and generative adversarial networks—in machine learning. In Chap. 17 on deep learning, a few examples, sample code, and further details are presented. Other interesting books to learn more about ANN are "Machine Learning—A Probabilistic Perspective" by Murphy (2012), "The Elements of Statistical Learning" by Friedman et al. (2001), and "Machine Learning" by Carbonell et al. (1983).

Support Vector Machines: Machine learning is really an art of *formulating an intuition into an objective function*. In classification, the fundamental problem is to find pure regions. So far we have explored a number of classification paradigms that greedily, iteratively, or hierarchically try to find such pure regions in the feature space. A good classifier should be both deterministic and robust to data and label noise. Perceptron, logistic regression, and neural networks are nondeterministic as they depend on model initialization and choice of hyperparameters governing model training. Decision tree, k-NN, Parzen windows, on the other hand, are more deterministic as they yield the same model for a given dataset and hyperparameters but k-NN and Parzen windows could be sensitive to data noise.

Consider the two-class classification problem shown in Fig. 16.15 (left). A perceptron trained on this data, could give an infinite number of solutions depending on its initialization, each of which will have a 100% accuracy. The fundamental question that forms the basis of support vector machines classifier is *which hyperplane is "optimal" among the infinite possibilities.* From a robustness point of view, the "best" hyperplane maximizes the width or margin of the linear decision boundary. This gives a unique solution for this problem (Fig. 16.15, right).

It is easier to understand this using an analogy. Assume that we want to build a straight road between two villages. Let the labelled data points $\{(\boldsymbol{x}_n, y_n)\}_{n=1}^N$ where $y_n \in \{-1, +1\}$ denote the houses of the two villages (classes). The goal is to build the *widest possible straight road* (i.e., maximum margin) that can be built without destroying any house in either of the two villages. This can be done by choosing the center and direction of the road in such a way that as we increase its width equally on either side and stop as soon as it touches the first house on either side. More formally let us say $\boldsymbol{x}^T\boldsymbol{w} + b$ (the solid green line in the right Fig. 16.15) denote the center of the road. The dotted lines parallel to it, on either side denote the boundaries of the roads obtained by extending the road on either side and stopping as soon as it hits a house (data point) on either side.

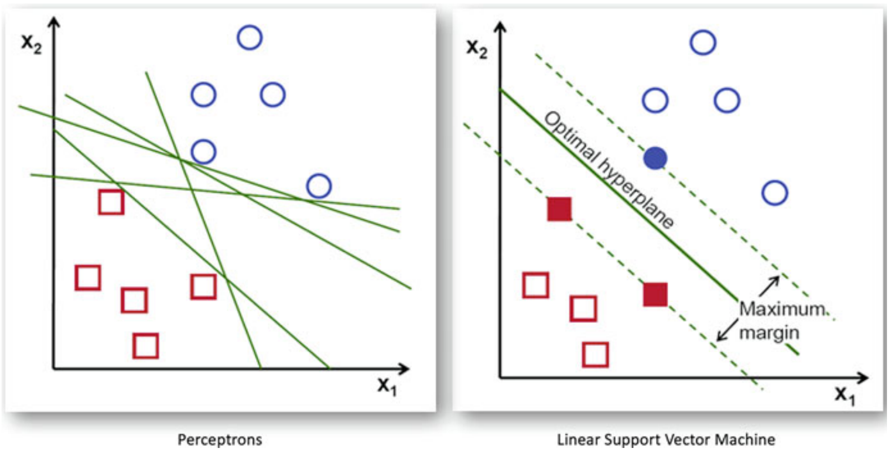


Fig. 16.15 The intuition behind maximum margin classifiers

This can be formulated as the following optimization problem. There are N constraints, one for each data point (house) so that it lies on the correct side of the road, that is,

$$\begin{Bmatrix} \boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x}_n + b \geq +1 \ \forall n \text{ where } y_n = +1 \\ \boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x}_n + b \leq -1 \ \forall n \text{ where } y_n = -1 \end{Bmatrix} \Longrightarrow y_n\left(\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x}_n + b\right) \geq 1, \forall n = 1 \ldots N$$

Note that we can use 1 as a threshold on both sides because any scaling factor can be subsumed in the linear coefficients $\boldsymbol{w}$ and constant term b. Using Geometry, (or examining the distance when there is equality), the width of the road is:

$$J(\boldsymbol{w}) = \left| \frac{(-1-b)}{\|\boldsymbol{w}\|} - \frac{(1-b)}{\|\boldsymbol{w}\|} \right| = \frac{2}{\|\boldsymbol{w}\|}$$

This needs to be maximized. To make the overall function well behaved we write it as:

$$\text{minimize}: \frac{1}{2}\|\boldsymbol{w}\|^2, s.t. y_n\left(\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x}_n + b\right) - 1 \geq 0, \forall n = 1 \ldots N$$

This can then be written using Lagrange Multiplier as the *primal* objective function. Every time the n^{th} constraint is violated (house is broken), a positive penalty α_n is paid. Since we want to ***minimize*** the objective function (which is obtained by inverting and squaring the margin), whenever $y_n(\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x}_n + b) - 1$ is negative the value of the overall objective should go up. This effect is obtained by the following Lagrange Multiplier terms combining the objective with constraints:

$$L_P(\boldsymbol{w}, b) = \frac{1}{2}\|\boldsymbol{w}\|^2 - \sum_{n=1}^{N} \alpha_n \left[y_n \left(\boldsymbol{w}^{\boldsymbol{T}} \boldsymbol{x}_n + b \right) - \right]$$

$$= \frac{1}{2}\|\boldsymbol{w}\|^2 - \sum_{n=1}^{N} \alpha_n y_n \left(\boldsymbol{w}^{\boldsymbol{T}} \boldsymbol{x}_n + b \right) + \sum_{n=1}^{N} \alpha_n$$

The points on either side of the road on which the margin "hinges" are called the Support Vectors—these are highlighted in Fig. 16.15 (right). Further SVM formulation is built on three "SVM Tricks."

SVM Trick 1—Primal to Dual: The primal objective function above contains two types of parameters—the original parameters of the hyperplane ($\boldsymbol{w}$ and b) as well as the Lagrange multipliers α_n. Note that hyperplane parameters can be used to determine the support vectors and similarly knowing the support vectors can determine the hyperplane parameters. Hence, the two sets of parameters are complementary to each other and both need not be present in the same objective function. To clean this up, let us optimize w.r.t. the hyperplane parameters first:

$$\frac{\partial L_P(\boldsymbol{w}, b)}{\partial \boldsymbol{w}} = \boldsymbol{w} - \sum_{n=1}^{N} \alpha_n y_n \boldsymbol{x}_n = 0 \Rightarrow \boldsymbol{w}^* = \sum_{n=1}^{N} \alpha_n y_n \boldsymbol{x}_n$$

$$\frac{\partial L_P(\boldsymbol{w}, b)}{\partial b} = \sum_{n=1}^{N} \alpha_n y_n = 0$$

The solution for $\boldsymbol{w}$ shows how knowing the support vectors and the data can be used to find the hyperplane. The second equation implies that the total penalty associated with the positive class is the same as the total penalty associated with the negative class. Substituting both these back into the primal, and simplifying, we get:

$$L_D(\boldsymbol{\alpha}) = \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{m<n} \alpha_m \alpha_n y_m y_n \boldsymbol{x}_m^{\boldsymbol{T}} \boldsymbol{x}_n, \; s.t. \sum_{n=1}^{N} \alpha_n y_n = 0 \; and \; \alpha_n \leq 0, \forall n$$

The dual objective function is *pure*—it is only in terms of the Lagrange multipliers. The solution to this convex optimization problem gives the right support vectors that form the boundaries of the widest possible road we can build without breaking any house. The first term minimizes the total penalty, while the second term uses pairwise dot-products or similarities between all pairs of points.

SVM Trick 2—Slack Variables: More often than not, either because of the nature of the decision boundary or noise in the data, the two classes may not be linearly separable. In such cases we will have to break some houses (violate some constraints) to build a road. Not only that, even if the data is linearly separable, it is possible that we might be able to build a wider road (find a better margin) if we were allowed to break some houses as shown in Fig. 16.16 below. Here for the

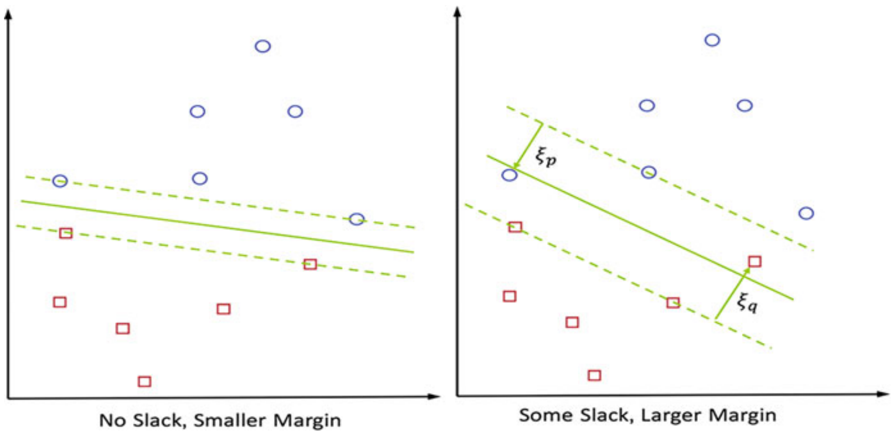


Fig. 16.16 The trade-off between bigger margin and violating the constraints on same dataset

same data if no constraints were allowed to be violated we can only build a smaller margin classifier (left), but if two constraints are allowed to be violated (two houses were allowed to be broken) then we can build a wider margin classifier. This ability to trade-off between maximizing the margin and violating some constraints is the second kernel trick. It is realized by introducing "slack variables" $\{\xi_n \geq 0\}_n^N$ that allow a certain slack on each constraint:

$$\left\{\begin{array}{l} \boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x}_n+b\geq+1-\xi_n \ \forall n \text{ where } y_n=+1 \\ \boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x}_n+b\leq-1+\xi_n \ \forall n \text{ where } y_n=-1 \end{array}\right\} \Longrightarrow y_n\left(\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x}_n+b\right)\geq 1-\xi_n, \forall n=1\ldots N$$

The Primal Objective function with slack variables has two additional terms. First a cost C associated with the total slack given and a set of terms to ensure that all the slack variables are positive.

$$L_P(\boldsymbol{w}, b) = \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_n \xi_n - \sum_{n=1}^{N}\alpha_n\left[y_n\left(\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{x}_n+b\right)-1+\xi_n\right]-\sum_n \mu_n\xi_n$$

Here, ξ_n are slack variables and μ_n are Lagrange multipliers on these slack variables. Converting this to the dual, however, gives a very elegant variation of the original dual.

$$L_D(\boldsymbol{\alpha}) = \sum_{n=1}^{Z}\alpha_n - \frac{1}{2}\sum_{m<n}\alpha_m\alpha_n y_m y_n \boldsymbol{x}_m^{\boldsymbol{T}}\boldsymbol{x}, s.t. \sum_{n=1}^{N}\alpha_n y_n = 1 \ and \ \alpha_n \leq C, \forall n$$

Note that the **only difference** now is that earlier the penalty of violating a constraint had no upper bound (i.e., $\alpha_n \geq 0$), which means that violating even a single constraint could result in an infinite cost. But with the introduction of the slack variables and a cost C on these slack variables changes the dual in only one way: It just upper-limits the amount of penalty that any single violation can cause, that is, $0 \leq \alpha_n \leq C$. This implies that even if a few constraints are violated, the maximum penalty could at most be C for each such violation and if that leads to a wider margin, so be it. The cost parameter C controls the complexity of the SVM classifiers. A low value of C will allow more constraints to be violated and larger margin, simpler classifier be learnt while a high value of C will allow smaller number of constraints to be violated and smaller margin, complex classifier to be learnt.

SVM Trick 3—Kernel Functions: Machine learning is the art of matching data complexity with model complexity. This is accomplished in two ways: Either we use linear (simple) models with nonlinear (complex) features or nonlinear (complex) models with linear (simple) features. For example, in logistic regression if the raw features are used as-is, we are not able to learn the complex decision boundaries and so we add nonlinear features (via generalized linear models). The third kernel trick is on the same lines. The original SVM formulation is only for two-class problems and learns a linear large margin classifier. To build more complex models than linear, we can introduce nonlinear features and "warp" the space and learn a linear classifier in the warped space. Note, however, that in SVM the only way data points are used in a space is to take their dot-products $\boldsymbol{x}_m^T\boldsymbol{x}_n$. Let us call this the kernel or similarity between these two data points: $K(\boldsymbol{x}_m, \boldsymbol{x}_n)$. SVM classifier really needs (only) this pairwise dot product (the Gram Matrix) as input. Now if there were a class of kernels where it was possible to actually compute this pairwise dot product in the transformed space directly without actually having to first transform the data into that space, then we could use this generalized kernels directly. In other words, let us say

$$K\left(\boldsymbol{x}_m, \boldsymbol{x}_n\right) = \boldsymbol{\phi}(\boldsymbol{x}_m)^{\boldsymbol{T}} \boldsymbol{\phi}\left(\boldsymbol{x}_n\right)$$

where $\boldsymbol{\phi}(\boldsymbol{x})$ is the nonlinear high-dimensional space to which the raw input $\boldsymbol{x}$ is mapped. Some of the common kernels used in SVM are:

- Polynomial Kernels: $K_{c,d}^{poly}\left(\boldsymbol{x}, \boldsymbol{x}'\right) = \left(\boldsymbol{x}^{\boldsymbol{T}}\boldsymbol{x}' + c\right)^d$ with hyperparameters c *and d.*
- Radial Basis Function Kernels: $K_{\sigma}^{rbf}\left(\boldsymbol{x}, \boldsymbol{x}'\right) = \exp\left(-\frac{\|\boldsymbol{x}-\boldsymbol{x}'\|^2}{2\sigma^2}\right)$

Using such nonlinear kernels to first warp the space into a hypothetical high-dimensional space, building a linear large margin classifier, in that space, and therefore realizing a nonlinear large margin classifier in the original space is the third kernel trick. Together, these three tricks make SVMs one of the most elegant formulations of an intuition into a powerful machine learning algorithm.

Scoring Using SVM

Given a dataset, the cost parameter C, the kernel, the kernel, and its hyperparameters, SVM learns a large margin classifier by finding the support vectors and generate as output the set of support vector weights $\{\alpha_n\}$. A new data point $\boldsymbol{x}$ is scored as:

$$S(\boldsymbol{x}) = \sum_n \alpha_n y_n K(\boldsymbol{x}, \boldsymbol{x}_n)$$

The class label is the sign of the score. Note that this scoring function is similar to Parzen window scoring except that in Parzen windows all training data points are used while in SVM, the weighted sum is taken only w.r.t. the support vectors, hence the time complexity is much lower.

One of the key drawbacks of SVM methods is that their training is quadratic in the number of training data (as they need pairwise cosine similarity) and hence with larger dataset learning an SVM can take much longer and can become quite infeasible. Sampling the data can address this.

In many domains, it is easier and more natural to quantify similarity between two data points than to represent a data point in a multidimensional space. For example, similarity between two LinkedIn or Facebook profiles, two gene sequences, two images or words, or two documents is much more natural than to represent them in a multidimensional feature space. In such cases, kernel-based approaches including SVM, k-NN, and Parzen windows might be more natural to use than traditional models such as decision trees or logistic regressions.

SVM in particular and Kernel methods in general have been applied in a variety of applications. Text classification using TFIDF representation was one of the areas in which they have shown remarkable success. A lot of research has gone into discovering new kernel functions for specific datasets and extending the SVM thinking (large margin) to other domains such as regression and outlier detection as well.

To learn more about SVM, you can refer to Chap. 14 in "Machine Learning—A Probabilistic Perspective" by Murphy (2012), Chap. 12 in "The Elements of Statistical Learning" by Friedman et al. (2001), or Chap. 9 of "Data Mining: Concepts and Techniques" by Han et al. (2011).

Ensemble learning: We have explored two broad approaches of building models: First, extracting better features (i.e., semantic, hierarchical, domain knowledge driven, statistics driven) and building complex models (deeper decision trees, deeper neural networks, nonlinear SVM vs. Linear SVM, neural networks vs. logistic regression, mixture-of-Gaussians vs. single Gaussian per class, etc.). Instead of building a single increasingly complex model (both feature and model complexity), the third approach is to *divide-and-conquer,* that is, break the problem into simpler subproblems, solve each subproblem independently, and combine their solutions. This is called *ensemble* learning, where the models must be different from each other in some ways while being similar to each other with respect to the nature of the modelling technique and complexity. In other words, we neither want to create an ensemble of, say, a neural network and a decision tree—they should all be

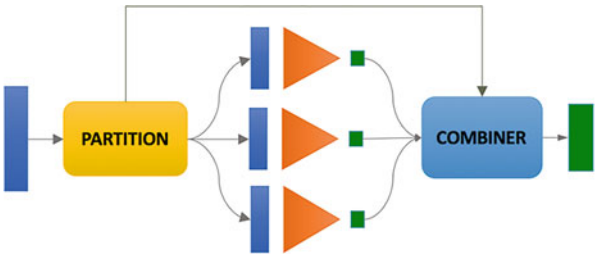


Fig. 16.17 The general architecture for ensemble learning

the same modelling technique—nor do we want to create an ensemble where each model by itself is too powerful. A good ensemble is a collection of diverse, shallow, similar models. The goal of ensemble learning could be accuracy, robustness, and even interpretability in some cases. Figure 16.17 shows the generic architecture of ensemble learning where the Partition layer divides the problem into multiple subproblems, the learners learn the model for each partition and produce an output for any new input. These outputs are then combined by the combiner—which is aware of the nature of the partition.

A number of different ensemble learning frameworks, each using a different way of partitioning and combining the models have been proposed in the past. Here we summarize a few.

Sample-Based Ensemble (Bagging)

Bagging or Bootstrap Aggregation is one of the oldest forms of ensemble learning methods especially used to build robust "average" models when the amount of labelled data is small. In Bagging, multiple samples (say 80%) of the original training data are used to build different models. These models are then "averaged" by taking the average of the output of these models for a given input. This averaging reduces the variance. The Partition is based on random sampling of the data and combiner is just a simple average.

Feature-Based Ensemble (Random Forest)

Instead of sampling a subset of rows (i.e., data points) as in bagging, we can also sample a subset of columns (i.e., features). Feature sampling is especially useful with modelling techniques such as decision trees where models are very sensitive to the set features they are allowed to use. Since decision trees are greedy, given all the features, they will generate the same tree. Lately, Random Forest has become a popular and powerful modelling tool. For a dataset with D features, we sample $\sqrt{D}$ features at a time, build a tree with these features, and then take an average of all these trees to generate the final output. Both sample-based and feature-based bagging are highly parallelizable as each of the models can be built independent of the other.

Accuracy-Based Ensemble (Boosting)

In both sample-based and feature-based bagging, all models are created equal—there is no bias or preference or order among the models. They are as random and independent as possible. The goal in bagging is "model averaging" not "accuracy improvement." Boosting is another class of ensemble learning approach that tries to improve accuracy of the model by building a sequence of models such that the next model focuses on the cumulative weakness of the models built so far. In boosting, the first model gives equal importance to all data points. The second model tries to focus more on (increase weight) those data points for which the first model does not do as well. The third model increases the weights of those data points whose error according to the cumulative first and second model so far is high. Boosting is not amenable to parallelism as the next model depends on the previous *k—1* models. Nevertheless, it is one of the most powerful techniques for building ensemble models. The key to boosting is again a large number of shallow/weak models. One of the most famous boosting algorithms is XGBoost that applies boosting to decision trees.

Region-Based Ensemble (Mixture-of-Experts)

The bagging and boosting algorithms focused on sampling the data or features randomly. Another class of ensemble learning algorithms is where each model focuses on a different part of the input space instead of building a single model for the entire space. For example, if we were to build credit models for all businesses, one approach would be to build a single complex model for all types (size $\times$ vertical) of businesses. Another approach might be to build a separate model for small, medium, and large businesses and also businesses in different verticals. The business size and vertical now become the "partitioning variables" instead of "input features" to the model. And each of the models becomes an "expert" in that cohort of businesses. Such a framework is called a mixture-of-experts. Local linear embedding is an example of a mixture-of-experts where to model a complex regression function, instead of using a high-order polynomial we might use local linear planes where each plane is valid only over a small region in the input space and near the boundaries the outputs of two planes are interpolated to give the final output.

Output-Based Ensemble (Binary Hierarchical Classifiers)

Most machine learning algorithms such as logistic regression or support vector machines are natural at solving two-class problems. But more often than not we are faced with classification problems with more than two classes (digits recognition, remote sensing, etc.). In such cases, we can apply these two-class classification algorithms in creative ways:

1-vs-rest classifier: One approach is to take a C-class problem and break it into C 2-class problems, where each problem takes one of the C classes as a positive class and all the other C-1 classes as negative class. This approach has a few drawbacks as the negative class can become large and create an imbalanced two-class problem each time. If we were to sample the negative class (C-1), choosing the right negative samples becomes critical to build a good 1-vs-rest classifier. Finally, the decision boundary where one class has to be discriminated from all the others might be too complex to learn.

Pairwise classifier: Here, the C-class problem is divided into (C choose 2) two-class problems where a 2-class classifier is built for each pair of classes. The advantage here is that each of the pairwise classifiers can select or engineer its own set of features (e.g., features needed to distinguish digits 1 vs. 7 are very different from the features needed to distinguish digits 3 vs. 8). With such specific features that focus on discriminating just the two classes at a time, the accuracy of these pairwise classifiers can be very high even with simple models. The domain knowledge that is discovered in terms of which features are needed to discriminate which pair of classes is an additional outcome. At the time of scoring, a new data point is first sent through all the pairwise classifiers where each one gives a label from among its class pair. Note that here each of the C classes has equal votes. The majority voting is used to then combine the output. The only drawback here is that the number of classifiers needed to be built is quadratic in number of classes. This can, however, be parallelized. Similarly, at scoring time each new data point has to be sent through all the pairwise classifiers. This again can be parallelized. Pairwise classifiers do not suffer from some of the problems of 1-vs-rest classifiers.

Binary hierarchical classifiers: In hierarchical clustering, data is clustered either in top-down (divisive) or bottom-up (agglomerative) fashion. In the same way, if we have a large number of classes, then the classes themselves can be clustered hierarchically. The distance between two classes can be measured by the accuracy of the pairwise classifier itself. Figure 16.18 shows an example of such a binary tree discovered from classifying letters of the English alphabet using OCR features. Here classes G and Q are merged first, then classes (M, W) and (F, P), etc. are merged. This is based on the training accuracy between those pairwise classes. These two classes are merged together (bottom up) and a new meta-class {G,O} is created. Now we are left with C-1 classes. The process is repeated again and a whole binary tree of classes is created.

Each internal node in this tree is a two-class classifier with its own set of features that best discriminate the two child (meta)classes. When a new data point has to be classified, it first goes to the root node where the root node classifier decides whether it looks more like "left meta-class" or "right meta-class." The data point is

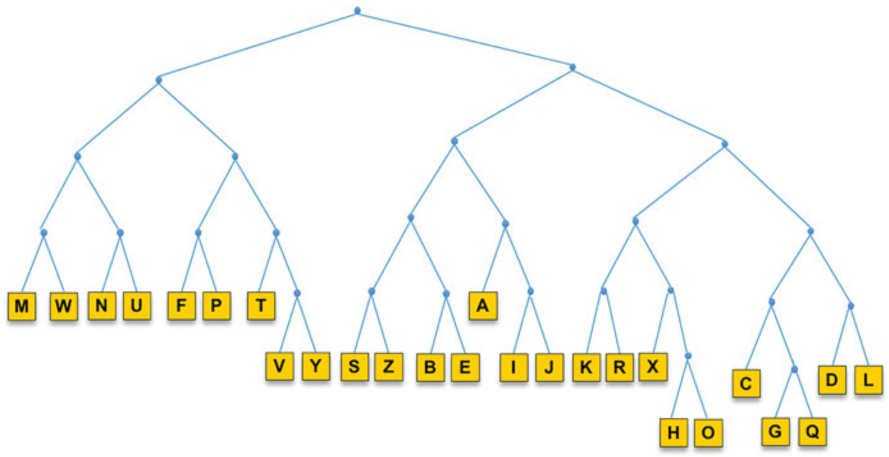


Fig. 16.18 A binary hierarchical classifier for a 26-class classification problem

then passed along that path recursively. This can be done both in a "hard" way, that is, send it either to left or right. Or this can be done in a "soft" way, that is, send it to both left and right with the posterior probability weight. These posterior weights are then multiplied across each path leading from the root node to the leaf node to get the overall posterior probability of each class. Such a classifier eventually needs only C-1 pairwise classifiers (as opposed to C choose 2 for pairwise classifier), it will still use features that discriminate only the two meta-classes at each node, and most importantly, this will automatically discover the class hierarchy as additional domain knowledge that we might not be aware of before.

Ensemble learning is used where individual complex models are not enough and we need to build robust and accurate models. The mixture-of-experts and binary hierarchical classifiers not only improve model accuracy but also improve model interpretation as they focus us on the right features and therefore a simpler yet more accurate decision boundary. In general, ensembles are more reliable than individual models as they explore the possible space of input/output mapping more thoroughly.

To learn more about various ensembling techniques, you can refer to Chap. 16 in "Machine Learning—A Probabilistic Perspective" by Murphy (2012), Chaps. 8, 10, and 15 in "The Elements of Statistical Learning" by Friedman et al. (2001), Chap. 11 in "Machine Learning" by Carbonell et al. (1983), or Chap. 8 of "Data Mining: Concepts and Techniques" by Han et al. (2011).

In this chapter, so far, we have explored the classification paradigm in depth. In this section, we will discuss various aspects of the recommendation engine paradigm.

4.5 The Algorithms: Recommendation Engine

Birds of a feather flock together.

Use-cases for recommendation engines: Today recommendation engines are used across many domains; for example, in the e-commerce domain, offers are recommended; in the media domain, movies, songs, news, videos, TV shows are recommended; in the food app domains, restaurants and dishes are recommended; in the travel domain, hotel and vacation packages are recommended; in the social media domain, people to connect are recommended, and in personalized education, the next concept, the right content, and the next problem the student should attempt are recommended. These recommendation engines serve multiple user needs including *discovery*, *personalization*, *serendipity*, and *optimization*.

Discovery—typically in domains such as e-commerce, media, social networking, or education there are a large number of choices (products, songs, connections, topics to study) for the user to choose from. There are different (and often a combination of) modes in which the applications make it convenient for the user to discover what they are looking for:

- In ***Search mode***, the user knows the complete or partial name of the item he/she is looking for (e.g., product, song, movie, or topic name).

- In ***browse mode***, the items are *organized* in categories (e.g., electronics vs. sports) and hierarchies (e.g., electronics ➔ cameras ➔ SLR cameras) so the user can navigate through this organization structure to reach the item he/she is looking for.
- In ***filter and sort*** **mode**, any list of items (obtained from search or browse) is further refined by either filtering (including or excluding) or sorting (in ascending or descending order) the items by various properties (brand, rating, price, etc.). In all three modes, the onus is on the user to discover what they are looking for using these modes.
- In ***recommendation mode***, the discovery process becomes proactive where the system itself suggests or pushes the items to the user that he/she is most likely to engage with next. This is one use-case of recommendation engines—build products that enable "intelligent proactive discovery" of items from a large collection of items for the user.

Personalization—most Apps or home pages of services today have an entry point that can be personalized for each user. For example, each user sees a different set of videos when they log into YouTube. They see a different set of suggestions for potential connections when they log into their LinkedIn, Facebook, or Twitter accounts. They also see different Netflix, Amazon Prime, Gaana, Saavn home pages depending on their previous activities on these Apps. The home pages differ not just from other users but also from their last visit. This degree of personalization of home pages of websites or apps is also powered by recommendation engines in the backend.

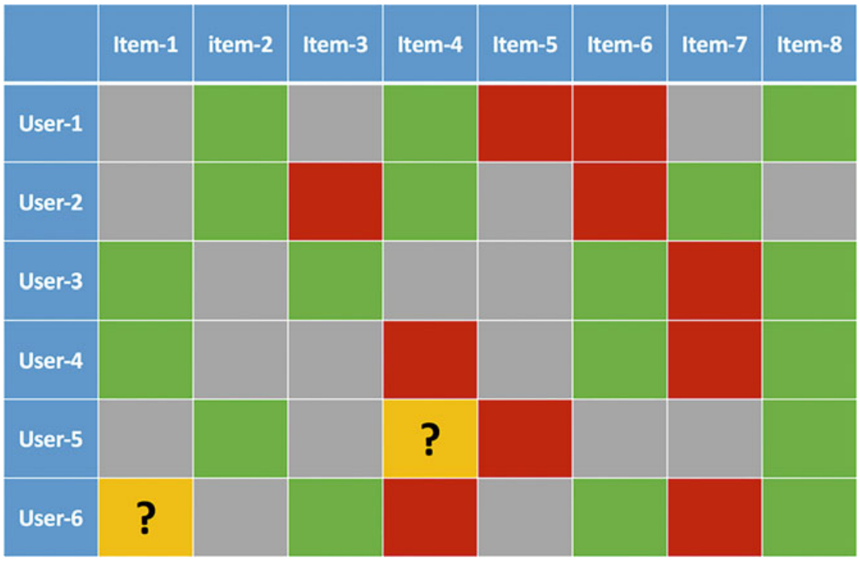
Serendipity—The epitome of intelligence is not just to do what makes sense but to do what might surprise us *while* it makes sense, to *exceed* our expectations. An essential element of advanced recommendation systems is this serendipity. Search is a zeroth order intelligence where the user already knows what he/she is looking for and the system is just trying to "match the query" with content meta-data; personalization is the first-order discovery where the user is suggested proactively what he/she might be looking for next, but serendipity is really a second-order intelligence where the user is suggested what he/she might not even be looking for but is pleasantly surprised by it. Serendipity opens up gates to a new dimension of exploration for the user. Serendipity in recommendations is like mutation in evolution. It allows for random yet connected exploration.

Optimization—Finally, recommendation engines can also be used to optimize different utility functions depending on the life stage of a customer. For example, when a new customer boards on a business (e.g., a bank, a retailer), the business tries to optimize the relationship of the customer over a period of time. In the beginning, the goal is just to transition a new customer to a loyal customer. Then from a loyal to a valuable customer, and then from a valuable to a retained customer. Each of these stages of a customer journey vis-à-vis the business can apply a different utility function when recommending the next set of products. For example, to make a new customer loyal, a retailer might offer daily use products at a lower price to the customer (milk, soap, groceries, etc.). Once the customer is loyal, the business might want to cross-sell the customer into other categories that are more profitable to the business and relevant to the customer (clothing, shoes, etc.). After that, the business might want to up-sell the customer even more profitable items such as high-end electronics or jewelry, etc. This delicate hand-holding of a customer leads to a long-term lifetime value of a customer. Some of the most advanced recommendation engines fine-tune their recommendations to each customer based not just on the customer's historical and demographic indicators but also the inferred stage of the customer's life stage in this journey.

4.5.1 Recommendation Engine Problem Formulation

Problem statement: Given the past engagement of a user with items in the domain, predict whether or not a particular user will exhibit high or low engagement with a particular item that he/she may or may not have been exposed to yet.

Figure 16.19 shows an example of an Engagement Matrix with six users engaging with eight items. If a user likes an item, the corresponding cell is marked green and if the user does not like an item, that cell is marked red. So, for example, user 2 likes items 2, 4, and 7 but does not like the items 3 and 6. These items could be movies or songs or products, etc. The gray boxes indicate that the corresponding user (row) has not yet interacted with the item (column). Let us say we have this data collected over a large number of users and items and we want to predict whether user-5 will like item-4 and whether user-6 will like item-1. In other words, should we recommend item-4 to user-5 and item-1 to user-6? How will we compute these "recommendation scores"?



	Item-1	item-2	Item-3	Item-4	Item-5	Item-6	Item-7	Item-8
User-1	⬜	🟩	⬜	🟩	🟥	🟥	⬜	🟩
User-2	⬜	🟩	🟥	🟩	⬜	🟥	🟩	⬜
User-3	🟩	⬜	🟩	⬜	⬜	🟩	🟥	🟩
User-4	🟩	⬜	⬜	🟥	⬜	🟩	🟥	🟩
User-5	⬜	🟩	⬜	?	🟥	⬜	⬜	🟩
User-6	?	⬜	🟩	🟥	⬜	🟩	🟥	🟩

Fig. 16.19 An engagement matrix with six users and eight items. Red indicates that the user did not like the item and green indicates that the user liked the item. Gray indicates the user has not yet interacted with the item

The key components of a recommendation engine are as follows:

- **Interaction data**—captures the transactions/interaction between user and the items. Given the application, there could be many types of interactions such as search, browse, write a comment, like, and share. Capturing all possible interactions makes our applications, not just a tool to deliver value, but also a sensor to capture data from user's likes and dislikes.
- **Domain knowledge**—is the set of properties associated with items (e.g., actors, director, genre of a movie; service, ambience, food quality of a restaurant; or category and attributes of a product; teacher, speed, teaching style, other attributes of an educational video) and users (e.g., demographics).
- **Engagement**—quantifies the degree of affiliation between a user and an item that the user has interacted with. Capturing and quantifying a holistic engagement metric by itself is perhaps the most important art in building a great recommendation engine. We will discuss this in more detail in the next section, "Engagement between a user and an item."
- **Item profile**—there are two broad classes of recommendation engine algorithms. In ***memory-based recommendation engines***, each item is treated as a unique ID. Here we only care about "which" items the user liked or not. However, the underlying idea is that every item has attributes and it is because of these attributes the user really liked or disliked the item (actor, music, direction, plot, genera, etc. of a movie). ***Model-based*** recommendation engines try to understand what "type" of items the user likes. In this case, the more detailed item features we can create, the more accurate model-based recommendation engines will become.
- **User profile**—in the model-based recommendation engines, we create a user profile in the same space as the item profile—so each user is a point in the same attribute space as the item is. These user profiles are a result of the cross between user engagement and the item profiles.
- **Recommendation score**—finally, the output of a recommendation engine depends on a score that predicts for every item that the user has not yet interacted with, the probability that the user would like to interact with the item. This score is then used to suggest the right items to each user. Various algorithms differ in whether they are memory or model based and how they compute the recommendation score.
- **Utility function**—finally, what is important is not necessarily the fact that the customer bought something that was suggested but the overall utility to the business. When a search Engine shows an ad, for example, it is not necessarily the most likely ad the user is going to click but if clicked, the advertiser will pay the most to the publisher. Hence, the utility function here is different. Similarly, in recommendation engines, we often put a layer of utility function that drives the final recommendations.

We will describe how these components come together to create a recommendation engine. We will define some notation that we will use going forward.

- There are N users: $\mathbf{U} = \{u_1, u_2, \ldots, u_N\}$
- There are M items: $\mathbf{I} = \{i_1, i_2 \ldots, i_M\}$
- Let $\mathbf{I}(u_n)$ be the set of items user u_n has interacted with.
  - So in the above example, $\mathbf{I}(u_1) = \{i_2, i_4, i_5, i_6, i_8\}$.
- Let $\mathbf{U}(i_m)$ be the set of users who have interacted with item i_m.
  - So, in the above example, $\mathbf{U}(i_3) = \{u_2, u_3, u_6\}$.

4.5.2 Engagement Between a User and an Item

One of the most creative parts of a recommendation engine is the definition of "engagement" between a user and an item that he/she has interacted with. Engagement can be measured in several ways: explicit vs. implicit feedback, one overall vs. multiaspect feedback.

Explicit Feedback—All intelligent systems improve with feedback—both explicit and implicit. In explicit feedback, user is explicitly asked for either a binary (thumbs up/thumbs down) or multilevel star rating at the end of the activity (e.g., after watching a movie, using a product, visiting a restaurant, or taking a cab ride). This could be even at a fine-grained level, that is, for each aspect of the product or service (e.g., acting, script, direction, music of a movie). Explicit feedback could be reliable if collected unambiguously in the right context (e.g., immediately after the activity). However, most of the time, explicit feedback is insufficient (not everyone gives it), fraught with problems of subjective user bias and contains both deliberate and natural noises. For example, a conservative user might always give ratings 1–3 out of 5 while a liberal user might always give ratings 3–5 for the same level of experience. This implies that a rating of 3 does not mean the same thing for the two types of users. Correction of such subjective bias is essential in explicit feedback-based recommendation systems.

Implicit Feedback—The best feedback is implicit—that is, gleaned from the natural activity of the user on the system (e.g., clicks in search engines, clicks on recommended YouTube videos, purchases on e-commerce sites, likes on Facebook and Twitter, shares on news stories, listening to a song or watching a movie, and acceptance of a connection suggested on LinkedIn or Facebook). The fact that user engaged with an item (a Web page, a news story, a YouTube video, a song, a movie, a product, a person) is itself an indication of his/her affiliation for that item. Implicit feedback is integrated into the user experience, it is abundant—available with every user activity, has no subjective bias, and is highly reliable. The real silent innovation in AI in the last decade was the art of collecting such implicit feedback and using it to continuously improve these services.

Refining the Implicit Feedback—Converting user actions captured in the logs to a dependable measure of engagement is an art that requires deep domain understanding and understanding of the nuances in the logs data. One example of this that we saw earlier was the subjective user bias in the way users rate an item.

We can remove that bias by a simple z-scoring of each user's past rating, that is, for each user, we can create a normalized rating based on his/her mean and standard deviation of all the ratings given in the past and use this normalized rating instead of the raw rating. Another example is in media consumption—songs, movies, videos. If a user clicks on a recommended item but does not finish it, only consumes a few seconds of it and then returns back—this also indicates lack of engagement. So, clicking is not engagement, finishing the experience up to a certain percentage is true engagement. Furthermore, repeated interaction of a user with an item indicates deeper engagement. Temporally, recent interactions should get a higher engagement score.

Combining Multiple Feedbacks—Finally, for the same item, the user might be giving more than one feedback. For example, in e-commerce, the user might be searching for product, spending time browsing the product, reading reviews on the product, adding it to wish list, removing from wish list, purchasing the product, returning the product, writing a review on the product, or responding to a review of the product. In media, the user might be again searching for a content, consuming the content partially or fully, downloading it, liking it, sharing it, etc. Combining all these various interactions both implicit and explicit—normalizing their scales, weighting them appropriately, etc., to come up with the final engagement score is again a fine art.

More formally, let there be K different types of engagements between a user and an item:

$$\boldsymbol{e}\left(u_n, i_m\right) = \left\{e_1\left(u_n, i_m\right), e_2\left(u_n, i_m\right), \ldots, e_K\left(u_n, i_m\right)\right\}$$

These different engagements are combined systematically using a set of weights to get the final engagement score between a user and an item:

$$e\left(u_n, i_m\right) = \sum_{k=1}^{K} w_k e_k\left(u_n, i_m\right)$$

Further, these weights combining various engagements could either be chosen or learnt, they could either be global or depend on a user segment, etc. The simplest example of an engagement is that a user rated an item on a scale of say 1–5. In the rest of the chapter, we will assume that each user–item combination has an engagement score obtained by combining the various engagement scores as discussed above.

One of the most basic principles of generalization in machine learning is that "*similar inputs lead to similar outputs*." This principle is used in different ways in classification algorithms discussed above. In the recommendation engine, the same principle will be used slightly differently in the different recommendation engine paradigms presented below.

Collaborative Filtering (CF) Paradigm

The earliest recommendation engines are nonparametric **memory-based recommendation engines** that are heuristics based purely on the user–item engagement matrix. Properties of users or properties of items are not used in these. These are collaborative filtering recommendation engines that are of two types, **user–user** and **item–item** collaborative filtering. Examples of commonly used collaborative filtering methods such as user-based and item-based recommendation systems are given in the online appendix.

User–User Collaborative Filtering: The basic intuition behind a user–user CF is as follows. In the above matrix if we could compute user–user similarity, we will find that user-1 and user-2 are both similar to each other in the way they like and not-like the items that both of them have interacted with. User-5 is also similar to user-1 and user-2. On the same lines, user-3 and user-4 are similar to each other and user-6 is also similar to users 3 and 4. Now since user-5 is more similar to users-1 and 2, and users-1 and 2 both liked item-4, there is a good chance that user-5 will also like item-4. Similarly, since user-6 is more similar to users 3 and 4, and they both liked item-1, there is a high chance that user-6 will also like item-1. This intuition led to the birth of user–user collaborative filtering.

There are two parts to building such a collaborative filtering-based recommendation engine: (a) quantifying user–user similarity and (b) estimating recommendation score using this similarity.

Quantifying User–User Similarity: The only data we have about two users is how they have engaged with (rated) all the items that they have interacted with. If there are items that both users have interacted with and their interactions were "correlated," then the two users will be considered similar to each other. The two common measures of similarity are:

- **Cosine similarity**: L2-normalized dot product between ***rows*** in the engagement matrix:

$$Sim_{COSINE}\left(u,v\right)=\frac{\sum_{i\in\mathbf{I}(u)\cap\mathbf{I}(v)}e\left(u,i\right)\times e\left(v,i\right)}{\sqrt{\sum_{i\in\mathbf{I}(u)}e(u,i)^2}\sqrt{\sum_{i\in\mathbf{I}(v)}e(v,i)^2}}$$

- **Pearson's correlation**: User-bias removed, L2-normalized over common elements:

$$Sim_{PEARSON}\left(u,v\right)=\frac{\sum_{i\in I(u)\cap I(v)}\left(e\left(u,i\right)-e(u)\right)\times\left(e\left(v,i\right)-e(v)\right)}{\sqrt{\sum_{i\in\mathbf{I}(u)\cap\mathbf{I}(v)}\left(e\left(u,i\right)-e(u)\right)^2}\sqrt{\sum_{i\in\mathbf{I}(u)\cap\mathbf{I}(v)}\left(e\left(v,i\right)-e(v)\right)^2}}$$

where $e(u)$ is the average engagement of user u, that is, $e(u)=\frac{1}{|\mathbf{I}(u)|}\sum_{i\in I(u)}e\left(u,i\right)$

Estimating Recommending Score: Once the user–user similarity is computed, we can use these to compute the rating of an item (e.g., item-4 above) for a user (e.g., user-5 above) given his/her similarity with all the other users (e.g., user-3 and user-4) and whether they liked or did not like the item. This can be written as a simple weighted sum as follows:

$$\widehat{e}(u,i) = e(u) + \frac{\sum_{v \in \mathbf{UNeb}(u,i)} \mid Sim\,(u,v) \mid \times \left(e\left(v,i\right) - e(v)\right)}{\sum_{v \in \mathbf{UNeb}(u,i)} |Sim\,(u,v)|}$$

where **UNeb**(u, i) is the set of ***users*** that are:

– Similar to the user u (i.e., have a nonzero similarity score)
– These users have an engagement score for item i

In the above example, Neb(user-5, item-4) = {item-1, item-2, item-4, item-6}. The neighbor list can be further pruned by choosing say just the top K most similar users. This measure also normalizes for user's bias by subtracting the average engagement score of the neighboring user $e(v)$ from the actual engagement $e(v, i)$. The estimated recommendation score, $\widehat{e}(u,i)$ is computed for all cells that are empty in the engagement matrix.

Item–Item Collaborative Filtering: Engagement matrix is a user–item matrix. We used it on a user-centric manner to learn user–user similarity, we can also use it in an item-centric manner to learn item–item similarity and then compute recommendation score. There is another reason why item–item similarity might be more practical in some cases. Typically, in a large retailer (e.g., Amazon, Flipkart) or media outlet (e.g., NetFlix, JioCinema), the number of users is O(100M) and computing user–user similarity becomes prohibitively expensive. The number of items, however, is typically O(100K) and it is easier to compute item–item similarity among them. Again, the same two-stage process is applied here:

Quantifying item–item similarity: The user–user similarity matrix was computed from rows of the engagement matrix. Similarly, the item–item similarity matrix can be computed from columns of the engagement matrix.

- **Cosine Similarity**: L2-normalized dot product between columns in the engagement matrix:

$$Sim_{COSINE}\,(i,j) = \frac{\sum_{u \in \mathbf{U}(i) \cap \mathbf{U}(j)} e\,(u,i) \times e\,(u,j)}{\sqrt{\sum_{u \in \mathbf{U}(i)} e(u,i)^2}\sqrt{\sum_{u \in \mathbf{U}(j)} e(u,j)^2}}$$

- **Pearson's Correlation**: Item-bias removed, L2-normalized over common elements:

$$Sim_{PEARSON}\,(i,j) = \frac{\sum_{u \in \mathbf{U}(i) \cap \mathbf{U}(j)} (e\,(u,i) - e(i)) \times (e\,(u,j) - e(j))}{\sqrt{\sum_{u \in \mathbf{U}(i) \cap \mathbf{U}(j)} (e\,(u,i) - e(i))^2}\sqrt{\sum_{u \in \mathbf{U}(i) \cap \mathbf{U}(j)} (e\,(v,j) - e(j))^2}}$$

where $e(i)$ is the average engagement of item i, that is, $e(i) = \frac{1}{|\mathbf{U}(i)|}\sum_{u \in \mathbf{U}(i)} e\left(u, i\right)$

Estimating Recommendation Score: We can use the item–item similarity to compute the rating of a user (e.g., user-5 above) for an item (e.g., item-4 above) given the item's similarity with all the other items and whether they were liked or not liked by this user. This can be written as a simple weighted sum as follows:

$$\widehat{e}(u, i) = e(i) + \frac{\sum_{j \in \mathbf{INeb}(u,i)} \mid Sim\ (i, j) \mid \times \left(e\left(u, j\right) - e(j)\right)}{\sum_{j \in \mathbf{INeb}(u,i)} |Sim\ (i, j)|}$$

where $\mathbf{INeb}(u, i)$ is the set of ***items*** that are:

– Similar to the item i (i.e., have a nonzero similarity score with i)
– These items have an engagement score with user u

The Cold Start Problem: Memory-based recommendation engines are ***brute force heuristics*** that gives high confidence recommendation scores for users who have enough interactions or items that have enough interactions. But whenever a new user (a more frequent scenario) or a new item (a less frequent scenario) is introduced in the system, the corresponding user's row or item's column in the engagement matrix is very sparse as he/she has had no interactions yet. This problem is typically addressed by recommending the most popular items to the new user in his/her context (e.g., city, country, demographic cohort). As the data on the user grows, we shift from this "default" model slowly to the collaborative filtering model. This transition from some default model for a data poor entity (e.g., new user or new item) to the target model (e.g., CF) as the data increases is another common theme across many practical implementations of ML algorithms.

Clustering Versions of CF: The CF approaches are computationally expensive and are not robust to "engagement noise." Another class of recommendation engines uses standard clustering algorithms to either cluster the users or items or both to first create a smoother, more robust representation of "similar customers" or "similar items" and use these as "representatives" to compute recommendation scores.

Matrix Factorization Approaches

In a way a CF recommendation engine score over the empty cells in the user–item matrix (where there is no interaction yet between a user–item pair) can be interpreted as a "smearing" or "smoothing" or "interpolation" of the cell given the corresponding row (user) and the column (item). The user–user CF was doing a row-centric interpolation and the item–item CF was doing the column-centric interpolation of the cell. What if we want to do both at the same time? A variety of matrix factorization approaches have been proposed in the literature. The most basic matrix factorization approach explored was to take the *singular value decomposition (SVD)* of the user–item engagement matrix. SVD is mathematically the best way to approximate the matrix, but the singular vectors (left and right) could have negative components that made them less interpretable.

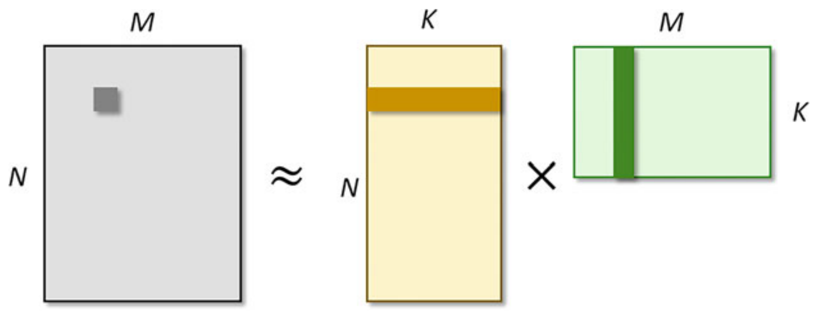


Fig. 16.20 Non-negative matrix factorization for approximating engagement matrix

This was further refined into a *non-negative matrix factorization* as shown in Fig. 16.20 where the original $N \times M$ engagement matrix (with N users and M items) is approximated as a product of two matrices: $\mathbf{E} \approx \mathbf{E}_K = \mathbf{A}_K \times \mathbf{B}_K$ where:

- $\mathbf{A}_K$ is an $N \times K$ matrix with one row corresponding to each user.
- $\mathbf{B}_K$ is a $K \times M$ matrix with one column corresponding to each item.
- Where $K \ll \min \{M, N\}$.

The goal is to find $\mathbf{A}_K$ and $\mathbf{B}_K$ iteratively starting with $\mathbf{A}_1$ and $\mathbf{B}_1$ (i.e., finding the first column of $\mathbf{A}$ and first row of $\mathbf{B}$ that, when multiplied, best approximate $\mathbf{E} = \mathbf{E}_0$. In general, by the time we reach iteration $k - 1$, $\mathbf{E}$ has already been approximated to $\mathbf{E}_{k-1} = \mathbf{A}_{k-1} \times \mathbf{B}_{k-1}$. The remaining error in the engagement matrix that is yet to be modelled, $\Delta\mathbf{E}_{k-1} = \mathbf{E} - \mathbf{E}_{k-1}$, is minimized by learning column $\mathbf{a}_k$ and row $\mathbf{b}_k$ as follows:

$$J\left(\mathbf{a}_k, \mathbf{b}_k | \Delta\mathbf{E}_{k-1}\right) = \sum\nolimits_{(u,i)\in\mathbf{E}} \left(\Delta e_{k-1}(u,i) - a_k(u) b_k(i)\right)^2 + \lambda_A \sum\nolimits_u a_k(u)^2 + \lambda_B \sum\nolimits_i b_k(i)^2$$

Here:

- $(u, i) \in \mathbf{E}$ implies that the summation is over cells where user u has engaged with item i.
- The first term minimizes error of approximation between residual error and parameters.
- The second and third are regularization terms that penalize for high values of parameters.
- Also note that in iteration 1, $\Delta\mathbf{E}_0 = \mathbf{E}$ itself.

Solving for the parameters we get:

$$\frac{\partial J\left(\boldsymbol{a}_k, \boldsymbol{b}_k | \Delta\mathbf{E}_{k-1}\right)}{\partial a_k(v)} = 2 \sum_{i\in\mathbf{I}(v)} \left(a_k(v) b_k(i)\right) - \Delta e_{k-1}(v,i)\, b_k(i) + 2\lambda_A a_k(v) = 0$$

$$a_k^{(t+1)}(v) \leftarrow (1-\eta)\, a_k^{(t)}(v) + \eta \frac{\sum_{i \in \mathbf{I}(u)} \Delta e_{k-1}(v,i)\, b_k^{(t)}(i)}{\lambda_A + \sum_{i \in \mathbf{I}(v)} b_k^{(t)}(i)^2}$$

$$\frac{\partial J(\boldsymbol{a}_k, \boldsymbol{b}_k | \Delta \mathbf{E}_{k-1})}{\partial b_k(j)} = 2 \sum_{u \in \mathbf{U}(j)} (a_k(u) b_k(j)) - \Delta e_{k-1}(u,j)\, a_k(u) + 2\lambda_B b_k(j) = 0e$$

$$b_k^{(t+1)}(j) \leftarrow (1-\eta)\, b_k^{(t)}(j) + \eta \frac{\sum_{u \in \mathbf{U}(j)} \Delta e_{k-1}(u,j)\, a_k^{(t)}(u)}{\lambda_B + \sum_{u \in \mathbf{U}(j)} a_k^{(t)}(u)^2}$$

Here, η is the learning rate. Vectors $\mathbf{a}_k$ and $\mathbf{b}_k$ are initialized to small values and learnt via these alternate updates until convergence is achieved. Once we have reached the maximum dimensionality K, we can compute the "interpolated" or "smeared" score for the cells with no engagement scores:

$$\widehat{e}(u,i) = \sum_{k=1}^{K} a_k(u) \times b_k(i)$$

Again, this can be used in two ways:

- Find the most likely users who will highly engage with item *i* (and have not done so yet)
- Find the most likely items that user *i* will highly engage with (and has not done so yet)

The cold start problem—when a user (row in the engagement matrix) or a new item (column in the engagement matrix)—is added, the recommendations have to be done using averages.

In both, the memory-based CF (user–user or item–item) and the model-based CF (matrix factorization) we have treated each user and item as an *ID* in a dictionary without considering their properties themselves. This approach models only the "what" and does not model the "why" behind a user engaging or not engaging with an item. Machine learning is really the art of finding the *why* behind the *what* and the CF type approaches do not provide such insights. We next discuss another class of recommendation engines that address this problem.

4.5.3 Profile-Based Recommendation Engines

CF-based recommendation engines answer the point question: "while ***this*** user engaged with ***this*** item." This is fine when we do not know anything about a user and an item and they are just IDs in the two dictionaries. But if we know or infer enough about a user (e.g., demographics, behavior patterns) and if we know enough about

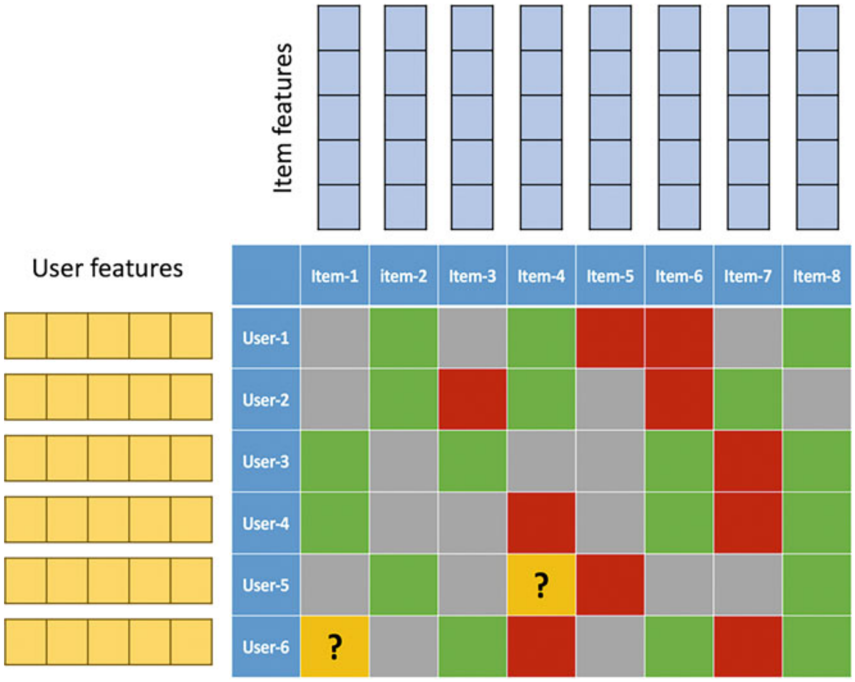


Fig. 16.21 In profile-based recommendation engines, we do not just know the past engagement between users and items, but we also know or infer additional user and item features that can be used to determine which type of users like which type of items

the items (e.g., meta-data), then we can answer the space question: "will ***such*** user engage with ***such*** an item." These are also known as profile-based recommendation engines and work with not only the engagement matrix but also the user and item properties beyond just the interaction among them as shown in Fig. 16.21.

There are four stages in building a profile-based recommendation engine.

1. **Characterize the features of users and items**: The meta-data (given or inferred) characterizes the "space" in which users and items live. For example:
   - **User features** include the basic demographics of the user—their age group, income group, gender, location, device, behavior patterns, preferences, etc.
   - **Item features** depend on the nature of the item, for example:
     - Movies—features are actor(s), actress(es), director(s), genre, plot, producer(s), etc.
     - Songs—features are singer(s), music director(s), album, genre, melody, length, etc.
     - News—features are entities, events, location, actions, sentiment, etc.
     - Videos—features are keywords, playlists, source, etc.
     - Tweets—features are hash-tags, keywords, source, sentiments, etc.
     - Clothes—features are brand, the fabric, the color, fashion type, fitting style, etc.

2. **Profile users (items) in item (user) space**: The intuition behind profile-based recommendation engines is as follows: We hypothesize that a user is engaging with an item because of ***certain properties of that item***. If we consider all items that a user is heavily engaging with and find out what is common among them—then we start to build a ***user profile*** in terms of item properties. For example:

   - A movie customer likes movies with certain plots and a certain set of directors.
   - A song customer likes classic songs sung by particular artists.
   - A retail customer likes clothes with bright colors, of a certain fabric, from a certain brand.

   In other words, the explicit engagements (e.g., like, buy, consume, add to list, write comment), of a user with an item (i.e., values in the engagement matrix) can be used to build the "***implicit profile***" of the user (item) characterizing ***what kind*** *of items* (users) a user (an item) likes (is liked by) instead of ***what*** items (users) a user (item) likes (is liked by). More formally,

   - Let $\boldsymbol{\pi}(u) = \{\pi_1(u), \pi_2(u), \ldots, \pi_L(u)\}$ be the L properties associated with user u
   - Let $\boldsymbol{\nu}(i) = \{\nu_1(i), \nu_2(i), \ldots, \nu_K(i)\}$ be the K properties associated with item i

   For now, let us assume these properties are binary ***indicator functions*** (e.g., "is actor X in movie *i*"). Like in CF, we could take a user-centric approach (e.g., *user–user* similarity), an item-centric approach (e.g., *item–item* similarly), or joint user–item centric approach (e.g., matrix factorization), here too we can either take a user-centric, item-centric, or a joint approach.

   - **User profiling**: Given the engagement matrix **E** between users and items and item properties, we can build a user profile by aggregating profiles of all items that the user engaged with. This answers the question: What kind of items this user is engaging with?

$$\phi_k(u) = \frac{\sum_{i \in \mathbf{I}(u)} e(u, i) \times \nu_k(i)}{\sum_{i \in \mathbf{I}(u)} e(u, i)}, \forall 1 \le k \le K$$

   - **Item profiling**: Again, given the engagement matrix **E** and user properties, we can build an item profile by aggregating profiles of all users that engaged with this item. This answers the question: What kind of users engage with this item?

$$\theta_\ell(i) = \frac{\sum_{u \in \mathbf{U}(i)} e(u, i) \times \pi_\ell(u)}{\sum_{u \in \mathbf{U}(i)} e(u, i)}, \forall 1 \le \ell \le L$$

   These user profiles are in item-spaces and the item-profiles are in user-spaces. In a way, we can think of a user profile as a point in the item-space and all items are also points in the same space. Similarly, an item-profile is a point in the user-space and all users are also points in the user-space.

3. **Profile matching**: The next step is to define a distance or similarity between two points in the item- or user-spaces. We can do this, for example, as a cosine similarity between the two.

   - ***Similarity between an item and a user profile***: Similarity between an item (a K-dimensional point, $\boldsymbol{\nu}(i)$, in item-space) and a user profile (a K-dimensional point, $\boldsymbol{\phi}(u)$, in item-space)

$$Sim\,(\boldsymbol{\nu}(i), \boldsymbol{\phi}(u)) = \frac{\sum_k \nu_k(i) \times \phi_k(u)}{\sqrt{\sum_k \nu_k(i)^2}\sqrt{\sum_k \phi_k(u)^2}}$$

- ***Similarity between a user and an item-profile***: Similarity between a user (an L-dimensional point, $\boldsymbol{\pi}(u)$, in user-space) and an item-profile (an L-dimensional point, $\boldsymbol{\theta}(i)$, in user-space)

$$Sim\,(\boldsymbol{\pi}(u), \boldsymbol{\theta}(i)) = \frac{\sum_\ell \pi_\ell(u) \times \theta_\ell(i)}{\sqrt{\sum_k \pi_\ell(u)^2}\sqrt{\sum_\ell \theta_\ell(i)^2}}$$

4. **The final recommendation score**: The final recommendation score between a user u and item i can be computed as a combination of the two similarity scores:

$$\widehat{e}(u, i) = Sim(\boldsymbol{\pi}(u), \boldsymbol{\theta}(i))^{\alpha}\; Sim(\boldsymbol{\nu}(i), \boldsymbol{\phi}(u))^{1-\alpha}$$

There are two key advantages of using profile-based recommendation engines:

- **Better generalization**—Being a model-based approach, these profile-based recommendation engines tend to generalize better as data increases. The key is to define the Profiling and Similarity functions better.
- **Cold start problem**—If we want to recommend a new item to a user who has a profile, all we have to do is compare the properties of the item with the profile of the user. So for example, if a movie is an action movie with a certain actor and we know that the user likes "such" movies (this genre and this actor), then we can recommend it to this user even if this movie has never been seen before.

Advanced Topics in Recommendation Engines

Recommending content, product, and services is a fine art done with different degrees of knowledge about a customer and with different sets of end-goals in mind. Below we highlight some of the other aspects that are typically considered beyond the recommendation score in making real-world recommendations.

1. *Deep content-based recommendations*: Profile-based recommendation engines put an item as a point in a feature space. Typically, this feature space is composed of the meta-data associated with the item as discussed above. In addition to this meta-data we can also extract deeper features from the items themselves that can be augmented with their profiles. This is possible only for a certain kind of

items that are rich in content. For example, if we are recommending a product on an e-commerce portal, we might have access to a number of additional product features but beyond that we do not have much to go on to accentuate the product's profile. But in content-rich items such as songs, videos, movies, news articles, teaching content, etc. we can extract deeper features from the content itself, add them to the meta-data features and then build a more holistic profile of the items to improve their representation. For example:

- *Song recommendation* can be enhanced by learning its melody and style by extracting say frequency characteristics of the songs, the instruments playing, etc. Why a user likes a song is not just because of its meta-data but because of its content.
- *Movie recommendation* can be enhanced by extracting activity features, for example, is there a car chase or an action sequence or a court scene or a cultural scene in the movie. What is the storytelling style or background music or nature of language used, etc.
- *News recommendation* can be enhanced by extracting entities, events, issues, topics, and sentiments about them in the news article and not just by representing it as a bag-of-words. The reader is interested in the real-world stories that the news represents, not just the words.
- *Teaching content recommendations* can be enhanced by identifying the different parts of a teaching content—real-world example, definition, detail, humor, motivation for the topic being taught—and how they are ordered. This will help extract the "teaching style" of the teaching content that must be matched with the "learning style" of the student.

Ultimately, algorithms in machine learning can take us only so far. It is the features that we can extract about our items that makes these algorithms come alive.

2. Strategic recommendations: Often recommendations are done on a trigger and for a purpose. The triggers could be entering the store (imagine a face recognition system recognizes customer) or logging into the online portal, reviewing a product (people who bought this also bought that), or making a final purchase of all items in the basket (printing coupons on the back of the receipt) These recommendations could serve very different purposes, which might be both tactical or strategic. The choice of the "utility function" to apply for a particular recommendation instance depends on the stage and context of the customer. For example:

   - *Recommendation for Loyalty*—Often if the goal is just to maximize the loyalty of a new customer, the business might just recommend commonly or repeatedly bought products (e.g., groceries or clothes) at a discounted price to a certain customer.

- *Recommendation for Cross-Sell*—Here the recommendations could increase the market basket by suggesting related products that the user might need along with the main product he/she is purchasing, for example, selling additional cartridges with a printer, additional storage with a camera, or a sun screen with beach wear. This is done through deeper understanding of "what products go together well" and not just item–item correlations as done in CF. This is applicable mostly in product domains where product relationships matter, more than in media domains where the only relationship between content is their similarity and not dependence (unless there are sequels).
- *Recommendation for Upsell*—In order to increase the value of a loyal customer we might recommend products in the same category (e.g., TV, fridge, cell phones) that are of a higher value to the seller than the one that the customer is currently looking for. Giving a higher resolution phone or a bigger TV or a product variant with more features is another kind of recommendation that is typically done at the time of purchase itself.
- *Recommendation for Lifetime Value*—Often, businesses strive to foster a long-term, value-oriented strategic relationships with customers (retailers, banks, cab-services, etc.). Here recommendations are chosen carefully not just for the value the next recommendation will add but how that will help open the gates for the next set of recommendations later. For example, a bank might give a lower rate on the car loan to a customer to recommend a home loan later. A bank might give very good interest rates to a customer on his/her salary account to maximize his/her lifetime association.
- *Recommendation for Preventing Churn*—Finally, some recommendations or offers might have a corrective intonation. For example, in a cab hailing service, if a customer is known to have had a bad experience (driver cancelled a ride or customer gave a poor feedback), then an immediate offer that brings the customer back on the positive side of the business might be useful—again this needs to be relevant to the customer as well.

3. Blended recommendations: The traditional notion of recommendation score is based on a user's past engagement but when a final recommendation page is rendered (e.g., personalized home screen of a user or recommendations of YouTube videos), other biases are also introduced in the final recommendation scores:

   - *Engagement bias*: This is the traditional recommendation scores based on past (positive or negative) engagements of the user with other items. This is done using some of the algorithms discussed above.
   - *Preference bias*: Often when a user is boarded on, his/her preference for coarse categories (e.g., sports, science, entertainment in news portals or artist, genre in the song) are recorded and if a new item falls into a customer's preference bucket, it is shown.

- *Location bias*: Typically, our apps know our location and if there are events (e.g., news or shows or sale) that are relevant to a user's location and are also relevant with respect to past engagements, then it might be shown to the user.
- *Popularity bias*: If something is becoming suddenly popular because it is either important or trending and even if it is only slightly relevant to the user it might still be shown to him/her. For example, a drastic news event (e.g., a terrorist attack or natural disaster or a major business announcement, best-selling book, a hit movie, or a viral video). This is typically seen in verticals where users give explicit feedback—likes, shares, buys, etc. indicating popularity.
- *Social bias*: Finally, in a social network setting, items that are explicitly popular in one's neighborhood in the social graph might also surface in a customer's recommendations as *birds of a feather flock together.* One might like what his/her friends on the network like.

What a user finally sees might be a combination of all these aspects together giving a ranking of what the user might see. The actual feedback by the user might be used to learn the weights of which biases among the above are more important to the user than others.

4. Cross-domain recommendations: Earlier, each service was focused only on one aspect of the user. For example, banks know only about a user's financial view, retailers know only about their purchase behavior, and cab hailing services know only about a user's travel behaviors within the city and its neighborhood, while airline services know only about the user's air travel. They all have a siloed view of a customer and can only suggest recommendations that are best suited accordingly. The next-generation businesses might provide different types of services to the same customer (e.g., Amazon has both a retail business and a media business) or might have different views on the same customer via different channels (e.g., firms such as Paytm or banks understand from the customer's payment behavior what kind of cross-vertical engagements the customer is having). The public profile of the customer—their Facebook, Twitter, and LinkedIn profiles—can also provide additional insights about a customer. Soon, recommendation engines will be able to combine all these views and suggest the right products and services with a more holistic view of the customer. For example,

   - Knowing that customer is booking a flight to a beach resort city (e.g., Florida or Goa) during the summer, one might recommend the right clothes and beach products for the customer.
   - Knowing that a customer just bought sports shoes might lead to recommendation of "sporty music" to a customer.
   - Knowing that a customer just took a home loan, a bank might recommend home furnishing products from a partner retailer to him/her.

5. Workflow-based recommendations: Finally, recommendation to a user could also be based on a well-defined workflow based on a "prerequisite graph" of items. For example:

   - *Next Concept Recommendation in Personalized Education*: When a student has engaged with the past concepts and mastered them to different degrees, a curriculum personalization engine can decide what next concept the student is ready to learn next. As shown in Fig. 16.22 below, the system might recommend some concepts that the student needs to master well before he/she can move forward or some concepts that he/she is ready to learn next because he/she has already mastered all of their prerequisites well.
   - *Next Action Recommendation in Agriculture*: Decades of agricultural experience and knowledge that mankind has accumulated can be used to recommend the right action to farmers at the right time in the right region depending on the climate, soil, and pest infestations prevalent in the region, etc. When to prepare the soil and how, when to plant the seed, when to put fertilizers and pesticides, when to water the plants, when to worry about the rains, and when to harvest the crop. A personalized workflow-based recommendation engine with flexibility to adapt to changing conditions on the ground can be made available to all farmers to alleviate them from guesswork and ignorance.

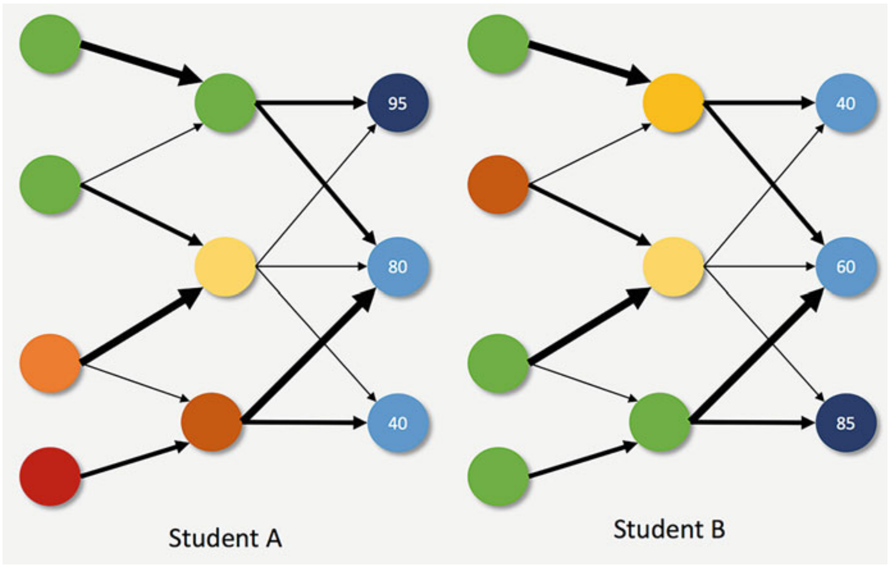


Fig. 16.22 Students A and B have mastered different set of concepts well (green), not so well (orange), and not at all (red). Depending on their mastery levels and the prerequisite or concept dependence graph, the workflow-based recommendation engine might suggest a different set of concepts to learn for student A vs. student B

- *Next Best Career Move*: Career building is a strategic art that requires right choices at the right time. Today, most people make some of the most important career decisions with limited understanding or ad hoc criteria. Each potential move in the career might have different prerequisites (e.g., an MBA school admission might require a minimum of say 3 years of job experience, a job might require one to have a certain set of hard or soft skills, or a profession, e.g., researcher, doctor, professor, might require one to have an advanced degree). A career moves recommendation engine, aware of prerequisite constraints, a user's personality, and aspirations might recommend the best next move—whether it is taking a certain MOOC course, pursuing a degree from a certain college, an internship experience in a certain company, or a volunteer work in a certain organization.

6. Contextual Recommendations: So far, we talked about which item the customer is most likely to engage with, but the success of that engagement might depend not just on the accuracy of the recommendation score but also on the context in which the recommendation is made. For example,

   - Recommending one's potential cuisine just before meal times
   - Recommending movies just before weekend or holiday starts
   - Recommending back-to-school items toward the close of summer vacations
   - Recommending a tourist spot when one has just landed in a new city for vacation
   - Recommending cartridge exactly 2 months after a customer bought a printer
   - Recommending different songs in the morning than evening than weekends

   The timing, the triggers, the location, the device, and the channel—are other aspects to be considered when making a relevant recommendation to the user.

Overall, recommendation engines play a very important role for many types of user interactions with the business, for discovering new items, for keeping the user engaged and informed, and helping the users make better choices. These recommendation engines become better with deeper understanding of the user—both where they have been and where they are heading. To read more on recommendation engines, you can refer to Singhal et al. (2017), Li et al. (2011), or Chap. 13 of "Data Mining: Concepts and Techniques" by Han et al. (2011).

5 Conclusion

In this chapter, we have explored the philosophy of generalization, the process of building a classifier, and the theoretical aspects of a wide variety of classifiers. We have explored trade-offs between accuracy and interpretability, hard vs. soft classifiers, descriptive vs. discriminative approaches, and feature-centric vs. model-centric thinking. The real art in building the right classifier comes from understanding the features, the nature of the decision boundary, and picking the right modelling algorithm to match the data complexity to the model complexity. Every decision today has a potential to be driven by data. The real challenge is to find the right insights, engineer the right features, build the right models, and apply the right business optimization to convert model predictions into decisions. Doing all this right will improve our ability to make more accurate, personalized, and real-time decisions, improving our businesses and processes multifold.

Electronic Supplementary Material

All the datasets, code, and other material referred in this section are available in www.allaboutanalytics.net.

- Data 16.1: Decision_Tree_Ex.csv
- Code 16.1: Decision_Tree_Ex.R

More examples, corresponding code, and exercises for the chapter are given in the online appendices to the chapter.

References

Carbonell, J. G., Michalski, R. S., & Mitchell, T. M. (1983). An overview of machine learning. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning volume 1. Symbolic computation* (pp. 3–23). Berlin: Springer Science & Business Media.

Friedman, J., Hastie, T., & Tibshirani, R. (2001). *The elements of statistical learning (Vol. 1, No. 10) Springer series in statistics*. New York, NY: Springer.

Han, J., Pei, J., & Kamber, M. (2011). *Data mining: Concepts and techniques*. Amsterdam: Elsevier.

Li, L., Chu, W., Langford, J., & Wang, X. (2011). Unbiased offline evaluation of contextual-bandit-based news article recommendation algorithms. In WSDM'11 (Ed.), *Proceedings of the Fourth ACM International Conference on Web Search and Data Mining* (pp. 297–306). New York City, NY: ACM.

Murphy, K. (2012). *Machine learning – A probabilistic perspective*. Cambridge, MA: The MIT Press.

Singhal, A., Sinha, P., & Pant, R. (2017). Use of deep learning in modern recommendation system: A summary of recent works. *International Journal of Computers and Applications, 180*(7), 17–22.