Chapter 16

Machine learning

We live in the era of machine learning. Machine learning, also called statistical learning, is the science and technology of building predictive models that generalize from data. The advancement of statistical learning and deep learning, supported by a deluge of data and computing power, has created enormous progress: now computers can identify objects in a photo more accurately than humans; computers can diagnose some diseases and predict health outcomes as accurately as doctors; computers can draw photo-realistic paintings from simple written prompts; computers can write documents and answer questions in natural language. Machine learning has not only disrupted many industries, it's revolutionizing how science is done. Problems that seemed to defy solution, such as protein structure prediction, have been effectively solved by a collaboration between scientists and machines.

Because of its ubiquity, network data are also heavily leveraged by machine learning. Thus, scientists working with network data (and all forms of data) can benefit from the tools and techniques of machine learning, as we explore in this chapter. There are several types of machine learning methods: supervised learning and unsupervised learning are the primary categories and then there are self-supervised learning, reinforcement learning, and more. Because the space of machine learning is vast and the book's focus is not machine learning, we will focus only on a subset of machine learning problems and techniques that are most relevant to network data.

We begin with an overview of machine learning problems common to network analysis before proceeding into the background and practice of machine learning.

16.1 Common network machine learning tasks

Although numerous machine learning tasks exist with network data, there are several fundamental tasks that cover a lot of our use cases. Traditionally, these tasks were tackled by designing network properties (Ch. 12) and using those properties as "features" for traditional, general-purpose machine learning models like logistic regression. However, with the advancement of neural networks, most state-of-the-art methods use graph embedding or graph neural networks, methods designed specifically for network data and which often determine features automatically.

https://doi.org/10.1017/9781009212601.019 Published online by Cambridge University Press

16.1.1 Machine learning with *and near* network data

Typical machine learning problems (such as classifying spam emails) and the general methods employed to address them (Sec. 16.2) are not particularly connected to network data. Nevertheless, general-purpose methods are helpful to the network researcher in a variety of ways.

For one, non-network data may need to be analyzed during the course of a study. A computational humanities researcher, for instance, may wish to apply classifiers to written documents describing the nodes of her network. Or a systems biologist may want to use dimensionality reduction on gene expression data as part of the upstream task of extracting the network. These cases work with data *near* the network, but in the traditional guise of supervised or unsupervised learning. Second, many network prediction tasks we describe below such as link prediction can actually be addressed using non-network-specific methods or a combination of network-specific and general-purpose methods. Sometimes, it may be better to use methods tailored to network applications, but not necessarily always. Many state-of-the-art network methods require large amounts of data (or large numbers of networks) and if your networks don't yield enough information, you may actually have better results with general-purpose methods.

If predictive models will be part of your research, it is best to have a good grounding in how they work and how they fail. And be sure to have good data hygiene (Sec. 16.6).

16.1.2 Node classification

One fundamental prediction task for network data is "node classification," where we need to predict a given node's class or label. For instance, we can imagine an online communication network of people where each person may belong to one of (or none of) multiple political parties. The task is to predict the party affiliation (label) of a person based on information about them (node features) as well as that of people around them (network features).

16.1.3 Link prediction

Another common task is "link prediction," as discussed in Sec. 10.6.2. Link prediction refers to the task of predicting missing links from a network data. Link prediction is ubiquitous when dealing with network data because the data is rarely complete and perfect (Ch. 10). One of the most direct examples would be the suggestions that social media platforms make. On Twitter, Facebook, and other social media services, you can see the feature where the service suggests some people to follow or send a friending request. These suggestions are made through link prediction.

At the simplest level, it makes use of the network structure around each pair of people. Let's say Alice and Bob are not friends on Facebook, but share 20 friends (they are all friends with both Alice and Bob). In such a case, it is reasonable to assume that Alice and Bob may know each other or at least know about each other through their shared friends. Unconnected users with many common friends are more likely to be connected (or *become* connected) than users with few or no common friends.

Link prediction is, however, not confined to social media services. Many problems related to network data can be formulated as link prediction problems. Biological networks are rarely "complete" and a large fraction of links between biological elements are missing.[1] For instance, protein–protein interaction networks are constructed with various experimental and data-mining methods and the overlap of links probed by different protocols is not that high, indicating that most links are never seen. Predicting likely missing links is therefore a very important problem that can guide biologists to discover new interactions.

Another example is the protein folding problem, where we need to predict the 3D structure of proteins from their sequences. To do this, we should be able to predict the long-range interactions between amino acid pairs that are located far apart in the sequence. One of the breakthroughs of DeepMind's AlphaFold is exactly this—being able to predict the links between distant amino acid pairs.

16.1.4 Community detection (clustering)

Another fundamental machine learning task is community detection (Sec. 12.7). It is a form of the clustering problem, a canonical unsupervised learning task. The goal of community detection is to find groups of nodes that are densely connected to each other while being sparsely connected to the rest of the network. A common class of community detection algorithms define a quality function (e.g., modularity) and then optimize it. Another approach, covered in Ch. 23, uses statistical inference: a statistical model is first defined that captures the relationship between the observed network structure and the communities and then the best parameters of the model (or their distribution) are estimated from the data.

The community detection problem is closely related to both the node classification and link prediction problems. A node classification task may be formulated as a community detection problem where node labels represent the communities. A link prediction task may be formulated as a community detection problem because it is usually assumed that the nodes in the same community should have a higher probability of being connected to each other than another pair that belong to different communities.

16.1.5 Graph representation learning (embedding)

Finally, there is a "meta" task, learning representations (Sec. 16.3) for a network. This is usually called "graph representation learning" or simply "graph embedding." In this task, rather than making specific predictions, we want to obtain a generally useful representation of nodes, edges, or even the whole network. Learning good representations is an essential part of machine learning and often good representations can automatically addresses downstream tasks. For instance, excellent performance can often be achieved by simply applying good graph embedding methods to obtain node embeddings and then combine those embeddings with simple machine learning methods (e.g., logistic regression). We discuss graph embedding in Sec. 16.7.

[1] We discuss techniques to estimate how many links are missing in Ch. 24.

https://doi.org/10.1017/9781009212601.019 Published online by Cambridge University Press

In addition to the network itself, we often have attributes, additional data describing the nodes, links, or both (Ch. 9). These data can themselves be used as features for machine learning but recently, new methods have been devised that are able to learn the representation of nodes by *combining* attributes with structural information from the network. The idea is that we can train a neural network to effectively combine and aggregate the node attributes through their local network neighborhood to make use of both structural (graph-level) and local (node-level) information. These methods are usually called "graph neural networks." We will examine this approach in more detail in Ch. 26.

16.2 Supervised learning

Tom Mitchell, one of the pioneers of machine learning, broadly defines machine learning as [312]:

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

The most straightforward type of machine learning that perfectly fits this definition is *supervised learning*. In supervised learning problems, such as labeling an email as spam or not, we have examples of what we wish to predict, called *training data*, that can guide our "learner" or model as it is built. The training data should have both the "hints" and the "answers"; the "answers" are called *labels* (or "ground truth") and they are what we wish to predict (e.g., whether a given email is a spam or not) and the "hints" are (usually) called *features*, the measurements that we can use to predict the label.

The pair of known hints and answers "supervise" the training of the model to improve its predictive abilities. Some examples of supervised learning problems and their training data include:

- predicting whether an email is spam or not after being given a large body of prelabeled spam and non-spam messages,
- classifying the type of animal present in a photograph after being given a large body of prelabeled animal images,
- forecasting the future value of a stock given the trading histories of all stocks in a market.

The general supervised model can be described as follows. Let $\mathbf{X}$ be an $n \times p$ data matrix where each row, sometimes denoted $\mathbf{x}$, represents a datapoint (an observation) and each column represents a feature (or variable). Let $\mathbf{Y}$ be an $n \times 1$ column vector called the target. The target $\mathbf{Y}$ contains our labels and is available in our training data, but not in general, and this is what we wish to predict using the information in $\mathbf{X}$, assuming that $\mathbf{X}$ can inform us enough to predict $\mathbf{Y}$ with some accuracy. In general, the link between the data matrix and the target is given by

$$\mathbf{Y} = f(\mathbf{X}) + \epsilon. \tag{16.1}$$

https://doi.org/10.1017/9781009212601.019 Published online by Cambridge University Press

Here $f(\mathbf{X})$ is an unknown, row-wise function of $\mathbf{X}$ and ϵ is a zero-mean, random noise term independent of $\mathbf{X}$. Because ϵ has a mean of zero,[2] we can use the mean of $f(\mathbf{X})$ as a replacement for $\mathbf{Y}$: when the target is not available we predict it using information from the features:

$$\hat{y} = f(\mathbf{x}), \tag{16.2}$$

where $\hat{y}$ is the prediction for a datapoint and $\mathbf{x}$ is a vector of features describing that datapoint. The fundamental challenge, however, is that f is unknown. Supervised learning proceeds by finding a functional "proxy" for f, denoted $\hat{f}$ (read "f-hat"):

$$f(\mathbf{X}) \approx \hat{f}\,(\mathbf{X}; \theta)\,. \tag{16.3}$$

Here $\hat{f}$ is a general purpose function that includes a vector of fit parameters or coefficients θ that we can alter to better approximate f. "Learning" is the process of modifying these parameters to improve the approximation using training data. The proxy function $\hat{f}$, or simply the *model*, can be any of a wide range of models, from linear regression equations to deep neural networks.

16.2.1 Regression and classification

There are two general types of supervised machine learning problems: *regression* and *classification*, determined by whether the target $\mathbf{Y}$ contains continuous/numeric (regression) or discrete/categorical entries (classification). The most basic method to perform a regression task would be *linear regression*, and the most basic method to perform a classification task would be *logistic regression*.

From Eq. (16.1), linear regression can be written as

$$\mathbf{Y} = \mathbf{X}\beta + \epsilon, \tag{16.4}$$

where $\mathbf{X}$ is a $n \times (p+1)$ matrix with each row as features with a constant term (1) added to each row (hence $p+1$ columns), β is $(p+1) \times 1$ column vector called *parameters* or *coefficients*, and ϵ is a zero-mean random *error* term or *noise* that should be independent of $\mathbf{X}$. The extra constant column is added to represent the intercept coefficient β_0 of the linear model. In other words,

$$\mathbf{X} = \begin{pmatrix} 1 & x_{11} & \cdots & x_{1p} \\ 1 & x_{21} & \cdots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \cdots & x_{np} \end{pmatrix}, \; \beta = \begin{pmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_p \end{pmatrix}, \text{ and } \epsilon = \begin{pmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \end{pmatrix}. \tag{16.5}$$

The β can be learned from the training data by minimizing errors. The error term is usually squared and summed and, if so, this is called the *ordinary least squares* (OLS) model:

$$\|\epsilon\|^2 = \|\mathbf{Y} - \mathbf{X}\beta\|^2 = \sum_{i=1}^{n} \left(y_i - \sum_{j=0}^{p} X_{ij}\beta_j \right)^2, \tag{16.6}$$

[2] This assumption is still general: if ϵ did not have a zero mean, we would subtract off its mean and fold it as a constant into f.

https://doi.org/10.1017/9781009212601.019 Published online by Cambridge University Press

where $\|\cdot\|$ is the Euclidean or ℓ_2 norm. Linear regression is useful for predictions but is often outperformed by more advanced, nonlinear methods such as neural networks. Yet linear methods have a very distinct advantage: interpretability. Once the fit parameters β are learned, we can easily reason about the relationship between an observation $\mathbf{x}$ and y. Specifically, if one feature x_i changes by a unit amount and all other features x_j ($j \neq i$) are held fixed, then our model predicts that y will change by an amount β_i. In comparison, for other methods, such as neural networks, we cannot as easily isolate the effects of particular features on the final predictions.

Logistic regression—although it is literally called "regression"—is probably the most widely used binary classification model.[3] Logistic regression is still a regression model in the sense that it tries to predict the *probability* for a data point to have a given $(0, 1)$-label, rather than the label itself, and that its fit parameters can be interpreted similar to a linear regression. Because probabilities must be between zero and one, we generally should not run a linear regression model as it won't be limited to $y \in [0, 1]$. Instead, we need a function that obeys those bounds. A choice of function that turns out to be quite convenient is the *sigmoid function*:

$$\frac{1}{1 + \mathrm{e}^{-x}}. \tag{16.7}$$

This function goes to zero as $x \to -\infty$ and goes to 1 as $x \to \infty$, obeying the bounds we need and smoothly transitioning between the two at some point. By applying linear (affine) transformations to x (i.e., $x \to \beta_1 x + \beta_0$), we can slide the transition point back and forth and we can make the transition steeper or more gradual. These "degrees-of-freedom" allow us to fit the model to training labels (0's and 1's at different values of x), although a fitting method other than OLS must be used [224].

Why the choice of sigmoid (Eq. (16.7)) instead of another function? A clever trick is to convert the probability into the *odds*, which is defined as the ratio between the probability (p) and the inverse probability ($1 - p$):

$$\frac{p}{1 - p}. \tag{16.8}$$

Applying a logarithm gives the *log-odds* (also called the "logit"):

$$\log \frac{p}{1 - p}. \tag{16.9}$$

The log-odds is useful because, unlike p, it can vary across the whole range of real numbers: as $p \to 0$, the log-odds approaches $-\infty$ and as $p \to 1$, the log-odds approaches ∞. And if we assume that p follows Eq. (16.7) we get:

$$\log \frac{\frac{1}{1+\mathrm{e}^{-x}}}{1 - \frac{1}{1+\mathrm{e}^{-x}}} = \log \mathrm{e}^{x} = x. \tag{16.10}$$

[3] Logistic regression is designed around predicting one of two possible labels for each datapoint, which we can just refer to as 0 or 1. However, it can naturally be extended to more than two labels, using what is called *multinomial regression*. For simplicity here, we focus on the binary case.

https://doi.org/10.1017/9781009212601.019 Published online by Cambridge University Press

In other words, the sigmoid is the *inverse* of the log-odds. This means that, even though our target variable is a probability, we can formulate a *linear* regression problem[4] for its log-odds:

$$\log\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_m x_m + \epsilon. \tag{16.11}$$

This can be also written as

$$p = \frac{1}{1+\mathrm{e}^{-z}}, \tag{16.12}$$

where $z = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_m x_m + \epsilon$ and z now represents the linear transformations we can use to move the sigmoid around to fit the training data. This sigmoid function *linking* the right-hand side of Eq. (16.11) to the probability is also called the *logistic function* and that is why this method is called *logistic regression*. In fact, the logistic regression model is but one example of *generalized linear models* that use various *link functions* to model different types of target variables.[5]

Once we can model the probability, we can perform a binary classification task: when the predicted probability $\hat{p}$ is larger than $1/2$, we can assign $\hat{y} = 1$; otherwise, assign $\hat{y} = 0$. And, while this idea is couched in binary classification, this idea can be generalized to multi-class classification problems as well, through multinomial (also known as softmax) regression.

16.2.2 Neural networks

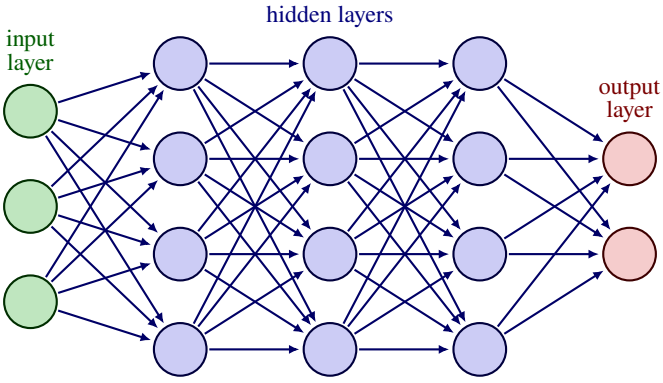
Neural networks have long been studied as part of artificial intelligence and machine learning. While promising in principle, it was not until recently that their practical significance was achieved, with the wild revolution of machine learning in the 2010s powered by neural networks. Neural networks are a general-purpose class of machine learning methods that are suitable to supervised learning, unsupervised learning, and anything in between.

Let's describe the basics of the prototypical neural network, the fully connected feedforward network. This network can be used for both regression and classification problems. Many variations exist, such as convolutional neural networks which are ideal for image data, but the principles are roughly the same. (We discuss neural networks for unsupervised problems below.) The neural network consists of a set of nodes called units or neurons that are arranged in some number of successive layers:

[4] Abusing notation slightly, here we use $x_1, x_2, \ldots$ to represent an observation's value for each variable, and the p we are estimating corresponds to that observation. Being more explicit, we would write p_i and $x_{i1}, x_{i2}, \ldots$ to note we are describing an observation i.

[5] Furthermore, we can imagine taking many different logistic regression models and nesting them together recursively. Models that do this are called neural networks.

https://doi.org/10.1017/9781009212601.019 Published online by Cambridge University Press



The particular arrangement of the units, the number of layers and the numbers of units per layer, is called the *architecture* of the network. In our fully connected case, each node in one layer is connected to every node in the next layer (this is what we mean by "fully connected"). Nodes in the first layer, called input nodes, receive the original data being analyzed. Nodes in the last layer, called output nodes, compute the final output of the network, used for predictions. As data move through the nodes between the input and output, it is transformed using an "integrate-and-fire" idea drawn in analogy to biological neurons. Specifically, the output z of a given unit is a weighted sum of its inputs $\mathbf{x}$, that was passed through a nonlinear *activation function* σ: $z = \sigma(\mathbf{w}^\top \mathbf{x})$, where $\mathbf{w}$ is a vector representing the weights on each input to the unit and $\mathbf{w}^\top \mathbf{x} = w_1 x_1 + w_2 x_2 + \cdots$.[6] Often, σ is the sigmoid function, but a variety of other options are possible.[7] The outputs at one layer then form the inputs to the next layer and the process repeats until the final layer whose outputs form the final prediction of the neural network. Our data dictate the shape of the output layer. For example, if we wish the network to be used for a binary classification, we would have two output nodes, one assigned to each possible label, and when data are fed through the network, we take as our predicted label whichever of the two outputs is larger. Just as our data shapes the output layer it also shapes the input layer. If we are performing a regression with 10 features, for instance, we would have 10 units in the input layer, one for each feature. If our data was text, we may have an input unit for each unique word in the language. And if we were working with images we might have an input unit for each pixel in the image.

As mentioned, a weight w is assigned to each link between units. These serve as the fit parameters and are learned via training. They are (typically) initialized at random, then training data are passed through the network, the final output is compared to the training label and the weights are adjusted to "nudge" the network's output towards the desired label. The process repeats many times until the weights are tuned and the network (hopefully) is well trained. The weights are usually updated with an algorithm called *backpropagation* [408]. For a fully connected network, we can combine all the weights between nodes in adjacent layers into a weight matrix, and compute all the weighted sums simultaneously using matrix multiplication. A neural network is essentially a large

[6] Usually, each unit also has a *bias* term, which acts similar to the intercept coefficient in a linear regression model.

[7] They must be nonlinear. Why?

https://doi.org/10.1017/9781009212601.019 Published online by Cambridge University Press

combination of matrix multiplications passed through nonlinear functions.

A major drawback of neural networks is their black box nature: unlike simpler methods such as linear and logistic regression, it is often challenging to interpret or explain how and why a neural network predicted what it did. This is the interpretability–flexibility tradeoff: neural networks work by being highly flexible and able to capture complex, nonlinear patterns in data, but in doing so they lose interpretability over simpler, more "rigid" methods.

16.3 Unsupervised, self-supervised, and representation learning

Unsupervised learning problems lack the training data used by supervised methods to learn fit parameters. There is an $\mathbf{X}$ but there is no $\mathbf{Y}$. Then how can any learning happen? One answer to this conundrum is that it is possible to detect, even without any labels, some interesting patterns in the data. One classic example is *clustering*: finding groups of data points that are very similar to one another while being different from other clusters. Many clustering methods exist, the classic being k-means clustering. Another example is *dimensionality reduction*: using an algorithm to recognize simpler, lower-dimensional representation of your high-dimensional data. Numerous dimensionality reduction methods have been developed, including principal component analysis (PCA), various matrix factorization techniques, t-SNE (t-distributed stochastic neighbor embedding) [471], and UMAP (uniform manifold approximation and projection) [302].

Unsupervised methods can also be combined with supervised learning. In *semi-supervised learning*, we have a small set of labeled training examples and a much larger set of unlabeled data. If the dataset contains useful features, then the data points with similar ground truth labels should be clustered together. Therefore, the clustering (unsupervised learning) can, in principle, be used to efficiently expand the small number of labels to the entire dataset.

Another answer to the conundrum is that it is often possible to create a supervised task from the unlabeled data by leveraging patterns in the data itself. For instance, if we have a large corpus of natural language sentences, we can think of a prediction task where we use all the previous words to predict the next word in each sentence. Likewise, if we wish to train a computer vision model and we have many photos but no supervising labels, we can create a training task by, for instance, taking each image and removing a small patch of that image, then ask the model to predict the correct missing patch from a set of small patches. In these cases, we are using the data itself—the patterns in the data—to produce labels for training. This is called *self-supervised learning*.

Self-supervised learning is deeply connected to *representation learning*. Representation learning focuses on obtaining useful *representations* (features) of the data elements. Many self-supervised tasks work by generating representations: a large language model for example becomes good at predicting words because it has generated computationally meaningful representations for those words, representations that capture the semantics and relationships between the words. The goal of representation learning is not just to make predictive models focused on the self-supervised task, but

https://doi.org/10.1017/9781009212601.019 Published online by Cambridge University Press

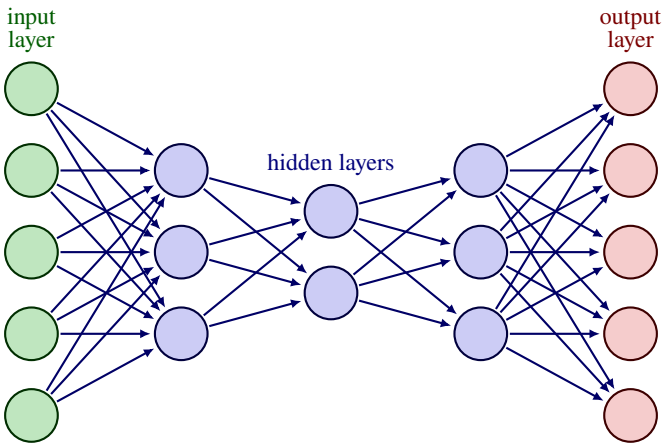
instead to obtain the most useful representations of an image, a word, a sentence, or whatever we are interested.

But how can this be useful? It turns out that learning good representations is one of the most important secrets to building powerful models in machine learning.

16.3.1 Neural networks for unsupervised problems

As discussed above, neural networks appear in both supervised and unsupervised problems. Neural networks need to be trained, generally using backpropagation and stochastic gradient descent. But these algorithms require training labels (you need errors if you want something to backpropagate), so how can they be used for unsupervised problems.

Many innovations come from transforming unsupervised problems into supervised problems. As an example, consider one type of neural network, the *autoencoder*:



This network takes a d-dimensional vector of data as input ($d = 5$ in the illustration) and uses *that same vector for the output*. No other labels are necessary. (Notice the input and output layers in the illustration are the same size.) Between the two is a bottleneck of hidden layers, and the network must learn to compress the input so that it can successfully reconstruct it after passing through the bottleneck. In other words, the neural network is learning to do (nonlinear) dimensionality reduction. Further, the learned weights become useful *representations* themselves.

An autoencoder is one example of using neural networks for unsupervised problems.

16.3.2 Importance of representation

Although we do not fully understand how and why many machine learning models work, there is already lots of practical knowledge that has been established. One of the most important lessons from machine learning research is that identifying good *features* for your data—or learning good *representations* from your data—is what matters most in building powerful machine learning models.

https://doi.org/10.1017/9781009212601.019 Published online by Cambridge University Press

If critical information is missing from the features, no fancy model can salvage it; on the other hand, it is easier to find a model that effectively extracts the information in the features as long as the features contains the signal. In other words, a simple model with useful features (information) tends to perform better than a state-of-the-art model with useless features. Therefore, *feature engineering*—the process of identifying and crafting useful features—has been historically recognized as a critical step in building a successful machine learning system in practice.

In fact, the huge progress of deep learning in recent years is largely thanks to the fact that deep neural networks excel at *learning* good features and useful representations. The lesson was that letting the model discover and learn features that are useful for the task at hand works much better than trying to hand-craft features, as long as the model that we are using is powerful enough and we have enough data—both conditions are satisfied with deep neural networks and massive datasets.

In the case of computer vision models, for instance, traditional feature extraction methods tended to be confined to low-level features such as sharp "edges" in the image and high-level features (human faces and their components such as eyes and noses) are not easily identified. By contrast, deep neural networks tie the process of learning both low-level and high-level features to the final prediction task, thereby extracting the highly useful and impossible-to-hand-craft features that help the task most. One of the meanings of the word "deep" in "deep learning" is this idea of connecting the feature discovery process itself to the prediction task.

And therein lies the power of representation learning. Representation learning allows powerful models (especially neural networks) to learn extremely rich representations of complex data such as images, videos, words, sentences, and so on. These representations can then be leveraged to solve a wide range of prediction tasks. A good example is *large language models* (LLMs). LLMs are "pre-trained" using text with a self-supervised task that asks the model to predict the next word in a sentence, fill the blank, fix the sentence, and the like. Through this pre-training and representation learning, LLMs have been shown to learn the high-level patterns of natural language extremely well—well enough to achieve human-level performance in some natural language understanding tasks (e.g., question-answering tasks). Once you have these LLMs that "understand" the patterns of natural language very well, then we can ask it to solve any other problems that involve natural language. For instance, the BERT (Bidirectional Encoder Representations from Transformers) model [129] has been used to solve a wide range of natural language understanding tasks, including question-answering, text classification, and so on, by *fine-tuning* the pre-trained LLM for a specific task using new training data. The incredible part is that the larger the pre-trained model becomes and the more pre-training it went through, the better it performs at specific tasks with small training data. In other words, scaling up the model's size and pre-training improves the quality of the representation and the quality of representation can then improve any tasks that use the representation. Furthermore, another type of model (decoder models such as GPT [383]) has been found that does not even need to be fine-tuned to a specific task! Instead, it is possible to simply *ask* the model as if it is a human and the model will *answer* the question [79]. This capacity of *few-shot* or *zero-shot* learning is opening up a new era of machine learning research and applications.[8]

[8] Along with a variety of thorny ethical questions.

https://doi.org/10.1017/9781009212601.019 Published online by Cambridge University Press

Representations and networks The same principle applies when building machine learning models with network data. If you are building a traditional model that relies on hand-crafted features, then it is absolutely critical to identify and engineer useful features that you can obtain from the network. But a lot of recent research on machine learning with network data is focused on ways to set up neural networks or other models so that we can learn good representations of the network—i.e., "graph embeddings"—then use such learned features to perform prediction tasks such as node label prediction and link prediction. We discuss ways of using machine learning with network data shortly.

16.4 Overfitting, bias–variance tradeoff, and regularization

As we have just discussed, the features included in the model limit the extent of information the model can use to learn the patterns in the data. Once we throw away a feature, the information uniquely contained in that feature cannot be recovered. Then a natural question is: is it always better to include more features?

The answer is: not necessarily. The more features we include, the more we are prone to *overfitting*. A great example is the case of a simple regression problem. Let's say we are predicting the price of houses using their square-footage x alone. The simplest linear regression model would be the following:

$$\hat{f}(x;\beta) = \beta_0 + \beta_1 x. \tag{16.13}$$

But we can also create a slightly more complex linear[9] regression model:

$$\hat{f}(x;\beta) = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + \beta_4 x^4 + \beta_5 x^5 + \cdots + \beta_d x^d. \tag{16.14}$$

This is a *polynomial regression* model as we're fitting a polynomial of degree d. The complexity of the model, and the number of parameters it contains, is given by d. What would happen with this model?

As we include more parameters by raising d the model will be able to fit the training data better and better.[10] But will this be a *better model*? Although this will reduce the error in our training data, the model overfits and begins to capture noise. It will not generalize well. If we throw new data points at it, the model would fail, especially with more and more parameters. We see this occur with some synthetic data in Fig. 16.1a–c. Here we created some ground truth data (Fig. 16.1a), and it is reasonably well-fitted when d is not too big (Fig. 16.1b). That model, with $d = 3$, should predict new data from the same underlying function fairly well. But larger d, especially in Fig. 16.1c where we chose $d = n$, the number of data points, while it fits the data we have perfectly,[11] it

[9] By the way, "linear" regression can absolutely use higher-order (nonlinear) features. What makes it linear is that the model is a linear combination of the features, a weighted sum where the fit parameters are the weights.

[10] Remember that, as John von Neumann put it, "with four parameters I can fit an elephant, with five I can make him wiggle his trunk."

[11] With a parameter for every data point, we have enough degrees-of-freedom to perfectly capture or *memorize* the data.

https://doi.org/10.1017/9781009212601.019 Published online by Cambridge University Press

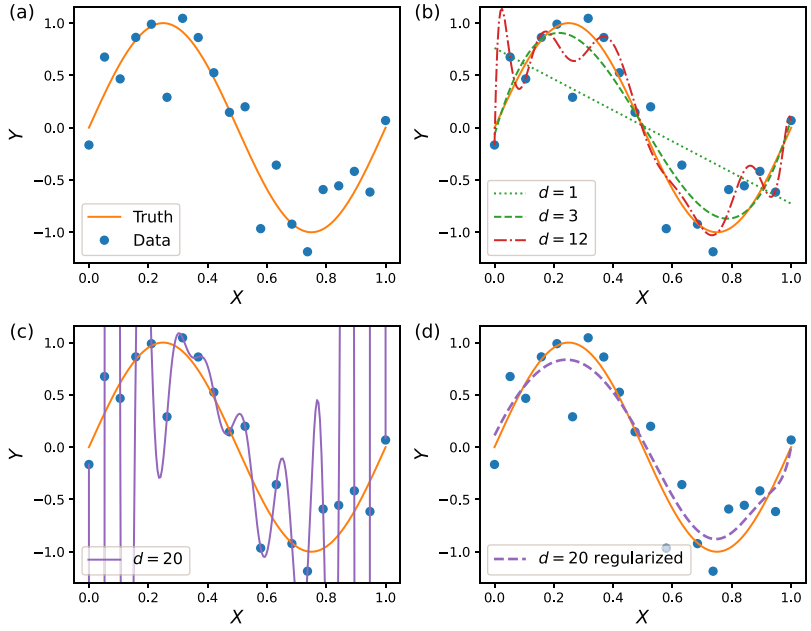


Figure 16.1 An example of overfitting and regularization using polynomial regression. (**a**) A small example of $n = 20$ data points generated by adding a small amount of noise to the ground truth function. (**b**) Different polynomial fits $\hat{y} = \beta_0 + \beta_1 X + \beta_2 X^2 + \cdots + \beta_d X^d$ of degree $d = 1$ (linear fit), $d = 3$, and $d = 12$. We see the curve start to deviate away from the ground truth at higher d, meaning that the fit is beginning to capture noise. (**c**) At $d = n = 20$ we have as many fit parameters as data points, and our curve perfectly fits the data points, interpolating them. But between the points it varies wildly, indicating that it will perform poorly at predicting any new observations. (**d**) A fit with the same number of parameters as (c), but now regularized. Despite having so many parameters, the quality of the fit is greatly improved. This model will likely predict new observations much better than the fit in (c).

varies wildly between those points—any new observations, drawn from the underlying example function, will be badly predicted by such a fit.

This phenomenon is called the *bias–variance tradeoff*. The *bias* refers to the error of the model. The worse the model predicts, the larger the bias is. The *variance* refers to the generalizability of the model. The worse the model generalizes, the larger the variance is. The bias–variance tradeoff is a fundamental property of statistical predictive models. The bias and the variance are connected: as you decrease the bias of a model, you inevitably increase the variance of the model. Conversely, this means if you increase the bias of the model, you decrease the variance, and it may make *better* predictions.

The traditional, statistical approach to this overfitting problem is to build the model carefully and then perform thorough diagnostics. This is strongly motivated by the need to accurately estimate the parameters of the model. For instance, a social scientist would start from certain theories and hypotheses, which dictates what kinds of features are

https://doi.org/10.1017/9781009212601.019 Published online by Cambridge University Press

measured and included in the model. Then the scientist would go through a series of tests to make sure that the model does not have problems of overfitting, or that the data violates any of the model assumptions such as multicollinearities.[12] By contrast, machine learning focuses on the prediction task and the model is built to perform the prediction task as well as possible. It is good to be able to estimate the relevant parameters, but it is not the primary goal. Therefore, there is a strong incentive to include as many features as possible.

Regularization An important solution to the challenge of overfitting with complex, many-parameter models is *regularization* (or *shrinkage*). Regularization is a technique that penalizes the model based on its complexity. For instance, the model that swings widely (Fig. 16.1c) would be considered complex because it has so many "important" parameters that should be accurately estimated to fit the data. We can make a simpler model by eliminating parameters, but we can also take a complex model and simplify it by enforcing constraints on the values of the parameters, preventing them from taking on arbitrary values. This "squeezes" the complex model down to something effectively simple, whichs turns out to be a very, very effective strategy for making accurate models.

Regularization techniques come in many forms, but for regression models like we've discussed so far, the general idea is to make parameter values that are large in magnitude more "expensive" than smaller values. For instance, two of the most common regularization methods for regression models are *ridge* and *LASSO* regression. The ridge regression penalizes the complexity of the model by considering a model's "cost" as:

$$\|\beta\|^2 = \sum_{i=1}^{p} \beta_i^2. \tag{16.15}$$

In other words, we use the Euclidean norm or ℓ_2 norm of the parameter vector β. We want this norm to not be too big; if it were that would indicate a model that is wildly varying the values of its fit coefficients, likely overfitting the model.

We incorporate this cost into our model by changing its optimization. Instead of just minimizing the sum of squared errors $\|\epsilon\|^2$ (Eq. (16.6)) by freely picking β, now we try to minimize $\|\epsilon\|^2$ but subject to a limit on how big the parameters can be:

$$\text{Find } \beta \text{ to min } \|\epsilon\|^2 \text{ subject to } \|\beta\|^2 < \alpha. \tag{16.16}$$

Here α is a new quantity we pick to represent our "budget" or how much we allow our parameters to vary. When $\alpha \to \infty$ we have no limit and we revert back to the original OLS fit. As we lower α, we place stricter limits on the fit parameters, *shrinking* them to smaller and smaller values. The new term α is an example of a *hyperparameter*; unlike β it is not estimated from fits to the data but must somehow be determined before fitting occurs.

Usually with optimization problems such as Eq. (16.16), we can write them in a second form, called the *dual problem*. Instead of minimizing one term subject to a limit

[12] For a linear regression, the fit becomes undetermined if one feature is a linear combination of other features. This is called *multicollinearity*. For data where one feature is close to a linear combination of other features, a fit is possible but it will be unstable: small changes to the data points will drastically change the fit parameter values. This makes the model particularly untrustworthy.

https://doi.org/10.1017/9781009212601.019 Published online by Cambridge University Press

on a second term, we can simply minimize a weighted combination of both terms. For Eq. (16.16), the dual formulation is:

$$\text{Find } \beta \text{ to min } \|\epsilon\|^2 + \lambda\|\beta\|, \tag{16.17}$$

where λ is related to α. (Notice we recover OLS fits when $\lambda = 0$.) By eliminating the constraint, we sometimes have an easier, although equivalent, optimization problem to solve. This is the usual form that ridge regression is presented. To demonstrate how powerful regularization is, Fig. 16.1d shows the same high-parameter polynomial model from Fig. 16.1c but with a ridge penalty included (we use $\lambda = 10^{-3}$ in the figure). The fit quality is greatly improved over the un-regularized model.

Another regularization technique similar to ridge is called LASSO. Its penalty is:

$$|\beta| = \sum_{i=1}^{p} |\beta_i|. \tag{16.18}$$

In other words, LASSO uses an ℓ_1 norm while ridge uses ℓ_2. This may seem like a minor difference, but they have fundamentally different implications for the types of the models that they encourage. A simple way to think about these two regularization methods is that you want to use LASSO if you want a *sparse* model—where you throw away more features from the model by setting some $\beta_i = 0$—and use ridge if that is not your priority (ridge may make β_i smaller but not exactly zero).

We've focused on regularizing OLS models, but regularization is not limited to linear models. Neural network models often use various regularization techniques during training. This makes sense because deep neural networks regularly employ a huge number of parameters (weights in the neural network), sometimes even much greater than the number of data points. Although neural networks seem much less prone to overfitting issues, it is often beneficial, even required, to use regularization techniques.

Double descent and neural networks Regularization produces a very important phenomenon called *double descent*, which is exhibited by highly expressive models such as deep neural networks. It is called double descent because, as we increase the number of parameters, the *test error* of the model first decreases and then increases (as expected by the bias–variance tradeoff), but then decreases *again* as the number of parameters increases further! This is highly counterintuitive and seems to violate the principle of the bias–variance tradeoff. But this indeed happens, not only in neural networks, but also in simple regression models. The key ingredient that makes this possible is regularization. When regularization is in place, the model can sometimes generalize better when over-parametrized! Specifically, when the flexibility of the model is greater than the number of datapoints, there exist numerous possible solutions that minimize the error function—they can "memorize" the training data. Yet, among these many possible solutions, only the simpler and more generalizable ones tend to be selected because of the strong constraints imposed by the regularization. Despite memorizing the training data, including its noise, they still generalize well. Although the power of deep neural networks is still in ways a mystery, the double descent phenomenon does not necessarily contradict the bias–variance tradeoff.

https://doi.org/10.1017/9781009212601.019 Published online by Cambridge University Press

So how to regularize a neural network model? It is possible to apply similar ideas as we did for regression models—penalizing large weights in the network. But there exist many creative solutions as well. One famous technique is called *dropout*. Dropout is a technique that randomly drops some of the neurons (like *half* of the neurons!) in the neural network during the training process. These dropped neurons are invisible for some portion of the training data, then returned and other neurons, also randomly chosen, are dropped. The idea is that we want to force the neural network to be able to perform the task even if an unknown but substantial fraction of neurons are missing. This forces the network to spread the "signals" across the network rather than "remembering" the answers.

Another interesting aspect of neural networks is that the training process of the neural network itself is known to be *self-regularizing*. The most common way to train neural networks is using *stochastic gradient descent* (SGD).[13] Neural networks are usually trained by feeding in the training data, seeing how well the output matches the training data labels, then slightly adjusting the network weights, sending the training data through again, adjusting again, over and over to improve the network. The SGD algorithm uses the gradient of the model—the direction and the amount that we need to adjust the parameters of the model, according to backpropagation—from a *random subset* of the training data, different for each iteration, rather than the full dataset. In other words, the SGD algorithm makes training more stochastic than gradient descent using the whole dataset, as the model never knows exactly what data it will see. Although this method was originally developed so that we can speed up training, there is a somewhat unexpected side-effect—the training process is self-regularizing, is less likely to become trapped in local optima, and overall is more robust because it is messier and noisier.

16.5 Model selection

An important concept closely tied to the bias–variance tradeoff, overfitting, and regularization is *model selection*. Model selection is the process of selecting a specific model from a set of competing models. For example, should we use a neural network or a linear regression? Should we use a linear regression with ridge regularization or should we use one with LASSO? If we decide to use polynomial regression, what order of polynomial is best? Whether choosing between different models, different fitting methods for the same model, or even different hyperparameter values, model selection methods help us make our choices.

The primary question in model selection is: *how can we know which model is better?* This question is intimately linked to the bias–variance tradeoff. A better model should not only have low bias, but also low variance. In other words, a good model should not only *fit the training data well* but also *generalize well*. But how can we know its generalizability without testing it on new data?

One way to think about model selection is to assume that *generalizability* or the variance of the model can be approximated by the *complexity* of the model, which can be quantified by the number of parameters in the model or something similar. Then

[13] With backpropagation being used to compute the gradient in error that is "descended."

https://doi.org/10.1017/9781009212601.019 Published online by Cambridge University Press

we can define measures that capture the tradeoff. For instance, the *Akaike Information Criterion* (AIC) is defined as follows:

$$\text{AIC} = 2k - 2\ln(\hat{L}), \tag{16.19}$$

where k is the number of estimated parameters and $\hat{L}$ is the maximized likelihood of the model.[14] The more parameters we have (high variance), the larger AIC becomes; the larger the likelihood of the model (low bias), the smaller AIC becomes. When faced with multiple models or multiple options (such as hyperparameters) within a model, we can compute each choice's AIC and select the model by picking the smallest.

Another, probably more common, approach to model selection is simulating a scenario of model generalization by utilizing the data we have. Specifically, we can split the training data available to us into a smaller training dataset and a dataset for model selection (often called a "validation set"). We then train the model using only the newly reduced training set and calculate the bias from it, then apply the model to the held-out validation set to estimate how well it generalizes. By ensuring the validation data is previously unseen to the model, we ensure it is not overfitting to it, or memorizing it. If the model is memorizing the training data, then we will see it perform badly on the newly revealed validation data. So we then select the model that performs and generalizes well. In practice, "cross-validation," a related technique, is often employed to efficiently use the dataset on hand. We will discuss these methods, known as *resampling techniques*, in more detail in the next section.

16.6 Data hygiene and evaluation

When we apply machine learning to data, one of the most critical things is to practice good *data hygiene*. Usually, in most other things, if you screw up, you would score worse. However, in machine learning, your performance may look *better* when you screw up![15] This is of course because of the bias–variance tradeoff—it is easy to overfit while it is difficult to obtain a *generalizable* model. To ensure the model performs well not only with the dataset that we currently have but also with any future data that we will have, we need to think carefully about how to train our models and how to measure their performance.

What screw-up are we talking about here? The key is to prevent *information leaking* from the validation dataset into the training process. If we have somehow accidentally included information from our validation set within the training data, then our attempts to validate the model will be polluted because the model already sees something about the validation examples. The more information leaks into our training, the more prone to overfitting our models will be.

Here's a simple and quite insidious way leakage can occur. Suppose we have a feature x_i and we want to rescale its values to have zero mean and unit variance. (This is often necessary for models and data where the typical magnitude of values in one

[14] The likelihood $L(X)$ of data X assuming a statistical model is the probability of X given the model's parameter(s) θ: $L(X) = \Pr(X \mid \theta)$. The maximized likelihood $\hat{L}$ is the likelihood that comes from the "best" parameters $\hat{\theta}$, the parameters that maximize L.

[15] At least until you measure the performance with another dataset.

https://doi.org/10.1017/9781009212601.019 Published online by Cambridge University Press

feature is very different from those of another feature.) This rescaling can be done with a z-score, here for the jth value of x_i:

$$z_{ij} = \frac{x_{ij} - \langle x_i \rangle}{\sigma_{x_i}}, \tag{16.20}$$

where $\langle x_i \rangle$ and σ_{x_i} are the sample mean and sample standard deviation, respectively, of the values in x_i. All well and good, we can now use z_i instead of x_i. This is standard practice, so what's the problem? Well, suppose we compute z_i and then split the data into training and testing. The mean and standard deviation in Eq. (16.20) were computed over *all* the data points, and thus the model we fit will have information it shouldn't, lurking in the sample statistics. The correct approach is to apply Eq. (16.20) *after* we split the data. Doing so captures the fact that we can't rescale data using observations we don't see. And if you think this will have only a small effect, think again: it can cause quite a false performance boost if you have lots of features and all are being transformed.

Training–test split First of all, for the sake of simplicity, let us focus on supervised learning. To be able to predict, machine learning models should generalize from the training data. But how can we measure their generalizability? As mentioned above, we can *estimate* it using proxies such as the number of parameters or by using the data splitting strategy to simulate the generalization scenario.

The latter approach is more commonly employed because it is a more direct test of generalizability. In essence, we need to find another dataset against which we can *test* the performance of our model. This new dataset can come from the future (if you can wait) or from a different system. But a much easier way is to simply split our existing dataset in two: *training data* and *test data*. (Usually we perform the split randomly. For each observation, with probability p it goes into the training data; otherwise with probability $1 - p$ it goes into the test data.) We can now train using the training data and then test using the test data to evaluate the generalizability of the model. Simple enough, right?

Although training–testing splits allows us to *test* the generalizability of the model, it is post-hoc. It only tells us how well the already-trained model generalizes, but cannot help us much in selecting the best model. To be clear, it is still possible to train multiple models on our training dataset and test them on the test set to choose the best model from the pack. However, this process—using the test data to select the best model—is not ideal because whenever we use the test data, the information only in the test dataset that allows us to estimate the generalizability of our models will *leak* to us. If we keep testing numerous models on the test set, eventually it will overfit, just as if we used the full data, not the split data, to train our models. Therefore, ideally, we want to avoid using *any information* from the test set in our training and model selection process, and this is the key to good data hygiene.

Training–validation/evaluation–test split When we need to perform model selection, a simple way to avoid information leakage from the test dataset into our training data is simply to split the training set *again* to create another test set (we usually call it

"validation set"). This validation set can then be used to evaluate and select models and hyperparameters. Although we can do even more "Russian doll"-like splits, the more splits we have, the less data will we be able to use to train the model.

Cross-validation Another very popular method for model selection and validation is *cross-validation*. With a single training–testing split we would only have a single measure of how well the model performs and it would be difficult to tell if that value is typical or not. Cross validation, on the other hand, uses multiple data splits to estimate the typical out-of-sample performance of a model. For instance, a 10-fold cross validation will split our data into 10 equal-sized sets. Once we have 10 sets of data, 9 of them can be used together as a training set and one will be used as a test set. After we estimate the performance on that test set, we put it back into the training set and take out another set to serve as the test set. We retrain the model from scratch on this new training set and re-evaluate on the new test set. We can do this for each of the 10 possible splits and this will produce 10 different results, which can then be averaged to assess the performance of our model.

Although k-fold cross validation[16] is most commonly employed, there are many other techniques. For instance, "leave-one-out" cross-validation uses a single data point as the testing set while the whole dataset except this single data point is used to train the model. This is then repeated n times, each time using one of the n observations as the test set.

Evaluation metrics Our procedure for splitting the training data to infer how well a model would generalize gives us unseen data for testing, but we still need to measure the trained model's performance on the testing data. How to evaluate performance? Many such metrics exist. For regression tasks where our model is predicting a numeric quantity, we want to compare a test value y_i to $\hat{y}_i = \hat{f}(x_i)$, the prediction our model $\hat{f}$, trained on the training data, makes on the corresponding input x_i. Common metrics are *mean squared error* (MSE),

$$\text{MSE}(y, \hat{y}) = \frac{1}{n_{\text{test}}} \sum_{i=1}^{n_{\text{test}}} (y_i - \hat{y}_i)^2 , \tag{16.21}$$

where n_{test} is the number of data points held in the test set, or the *coefficient of determination* (R^2)[17]

$$R^2(y, \hat{y}) = 1 - \frac{\sum_{i=1}^{n_{\text{test}}} (y_i - \hat{y}_i)^2}{\sum_{i=1}^{n_{\text{test}}} (y_i - \bar{y})^2}, \tag{16.22}$$

where $\bar{y} = \frac{1}{n} \sum_{i=1}^{n_{\text{test}}} y_i$, but many other metrics exist.

[16] $k = 10$ is the most common number of folds.

[17] A common "gotcha" in practice is to discover a problem with a model by finding bizarre values of R^2. We expect, because it is a squared quantity, for R^2 to always be positive, but this is not the case. If the model is making predictions that are so bad, it may be doing worse than a model that is just predicting the average value of y, in which case the second term in Eq. (16.22) will be greater than one. If you encounter negative R^2 scores, reconsider your modeling and evaluation methods.

https://doi.org/10.1017/9781009212601.019 Published online by Cambridge University Press

For classification tasks where our model is predicting a label such that test value y_i is now a categorical quantity, we use metrics that compare how often the predicted category $\hat{y}_i$ matches the test label. Here are some metrics for binary classification tasks, with only two possible categories, but many multi-class metrics are commonly used. The most natural binary metric is probably the *accuracy*

$$\text{Accuracy}(y, \hat{y}) = \frac{1}{n_{\text{test}}} \sum_{i=1}^{n_{\text{test}}} \mathbb{1}_{y_i = \hat{y}_i}, \tag{16.23}$$

where the *indicator function* $\mathbb{1}_x = 1$ if x holds, otherwise it is 0. That said, accuracy has some problems. For instance, on imbalanced data sets where most y_i are in one category, performance will appear inflated. It also doesn't distinguish where errors are happening; are you making false positives or false negatives[18]? Thus, it is common to employ more specific metrics, *precision* and *recall*:

$$\text{Precision} = \frac{TP}{TP + FP}, \tag{16.24}$$

$$\text{Recall} = \frac{TP}{TP + FN}, \tag{16.25}$$

where TP, FP, and FN are the numbers of true positive, false positive, and false negative evaluations, respectively. The intuition: Precision measures, whenever the model predicts a positive, how often it is correct. Recall measures, whenever the data are positive, how often the model is correct.

Working with two metrics is generally fine, but sometimes we need a single numeric quantity to evaluate the classifier. Usually, we then combine Precision and Recall using an F-score:

$$F_\beta = \left(1 + \beta^2\right) \frac{\text{Precision} \times \text{Recall}}{\beta^2 \text{Precision} + \text{Recall}}. \tag{16.26}$$

This is the harmonic mean of Precision and Recall, and β is a weight that tells us how much more important one is over the other; the F1-score, where $\beta = 1$ weighs them equally.

While we've described some of the common evaluation metrics, many others are used for both classification and regression. Please see our remarks below if you are interested in learning more.

16.7 Graph embedding

Graph embedding has emerged as a powerful tool for various machine learning and analysis tasks with network data. Graph embedding converts a network into a *vector representation*: a node embedding would learn a continuous and dense vector representation for each node, an edge embedding would do the same for edges and a whole graph embedding would do the same for each network. Once we find the vector representations, they can be readily plugged into machine learning models and pipelines to

[18] Assume that the two categories are $y = 0$ and $y = 1$. A false positive is when you predicted $\hat{y} = 1$ for a test point where $y = 0$. A false negative is when you predicted $\hat{y} = 0$ but $y = 1$.

https://doi.org/10.1017/9781009212601.019 Published online by Cambridge University Press

help with downstream predictions. The vector representations can be used to visualize the network data as well.

Let's start from the adjacency matrix (Eq. (8.1)). In the adjacency matrix, each row and column vector captures the neighborhood of a node and actually is an N-dimensional vector representation of the node. However, the adjacency matrix is not a good representation because it is neither *dense* nor *continuous*; it contains only zeros and ones and is sparse. Consider two random nodes from a large, sparse network. The probability that they have any overlap in their neighbors is very low. This means that the corresponding *vector representations* of these two random nodes will probably be orthogonal to each other, without any overlapping components. In other words, most node pairs will appear completely unrelated to each other in the vector space. If we took, say, the dot product[19] between the vectors to compute their similarity, it would almost always be zero. Even if a pair of nodes has neighbors in common, because of the non-continuous nature of the adjacency matrix, we will not have a fine-grained, useful measurement of similarity between them.

A simple way to overcome the issue of sparsity is to define a more appropriate form of similarity between nodes. For instance, instead of measuring the similarity between two nodes by the number of common neighbors they have, we could measure how close they are located to one another in the network. We can even use random walk trajectories to measure the similarity between nodes, for instance by calculating the transition probability between nodes, how likely or how often a random walker moving over the network moves from one node to the other. Now the representation we have—still N-dimensional—is continuous and dense. Is this the best representation we can have? One important drawback of this representation is that it is not compact—each node is represented by a vector of size N. It turns out we benefit from vector representations that are dense and sit in dimensions $r < N$.

16.7.1 Compact representations

One of the most popular approaches to find compact representations has been *matrix factorization* methods. A square[20] matrix $\mathbf{S}^{N\times N}$ can be factorized, into multiple matrices, for instance, with *singular value decomposition* (SVD): $\mathbf{S} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$, where $\mathbf{U}$ and $\mathbf{V}$ are $N \times N$ unitary matrices and $\mathbf{\Sigma}$ is a $N \times N$ diagonal matrix containing *singular values* on the diagonal. When we *truncate* $\mathbf{\Sigma}$ be retaining only the r largest singular values, we compute a *low-rank* approximation of $\mathbf{S}$. This approximation helps with prediction by eliminating noise in $\mathbf{S}$, improving generalization, especially when most singular values are small and we can choose r such that $r \ll N$. It also allows us to find the most meaningful, *compact* representations for our data, because we can treat the rows of $\mathbf{U}$ as vector representations but keeping only the r most relevant columns, giving us dense, compact r-dimensional vector representations.

When $\mathbf{S}$ is non-negative, for example, when it is an adjacency matrix, we may instead wish to use *non-negative matrix factorization* (NMF):

$$\mathbf{S} \simeq \mathbf{W}\mathbf{H}, \tag{16.27}$$

[19] Or, cosine similarity.

[20] SVD is not limited to square matrices although we only treat that case here.

https://doi.org/10.1017/9781009212601.019 Published online by Cambridge University Press

where $\mathbf{W}$ and $\mathbf{H}$ are $N \times r$ and $r \times N$ matrices usually with $r \ll N$. The benefits of this approach is that it captures the matrix being non-negative and gives representations that are dense, continuous, but also compact—each node is represented by a vector of size $r \ll N$.

Another approach to find such representations is graph embedding algorithms. For a network with N nodes, the objective of graph embedding is, just like matrix factorization, to learn an $N \times r$ matrix, where $r \ll N$ is the dimension of the embedding and each row will be the vector representation of the corresponding node. There are numerous approaches, but we can categorize them into several classes of models. The first class of methods—simply called graph embedding—does not use any additional information other than the network structure. Usually, structural proximity between nodes is captured through random walks or other graph traversal methods and then translated into a vector representation. The most straightforward approach in this class may be "DeepWalk" and "node2vec," where the random walk trajectories, the sequences of nodes visited at random as a random walker bounces around the network, are considered "sentences" and fed into the standard word2vec model.[21] Second, there are methods that make use of node and edge attributes, in addition to the structural information. These methods are usually referred to as "graph neural networks" and we will examine that approach briefly below and in more detail in Ch. 26.

Although it was not obvious at first, it was discovered that neural network-based representation learning, including graph embedding, is an *implicit* matrix factorization [269] (Ch. 26). Instead of going through the process of explicitly constructing the similarity matrix $\mathbf{S}$ and then factorizing it into low-rank matrices, neural graph embedding methods directly obtain the factorization (representation) by using neural networks to learn the representation from some self-supervised prediction tasks. This means that we do not have to create the large, dense matrix $\mathbf{S}$. Therefore, it is inherently space-efficient and scalable. Furthermore, thanks to the usage of stochastic gradient descent or similar methods in training, the training process is also computationally efficient and robust [270].

16.7.2 Applying neural networks for graph embedding

The most canonical graph embedding methods use random walks to generate sequences of *nodes* and treat them as *"sentences,"* from which the "word" (node) vectors are learned. Just like word2vec or similar models use natural language sentences to learn word embedding, this class of methods use the same neural architecture to learn node embeddings. DeepWalk and node2vec are two well-known, basic variations. Random walks are fundamental to understanding a network's structure and how it affects dynamics. We discuss random walks further in Ch. 25.

When applying these methods, as these methods use random walks, we need to pay close attention to structural properties of the networks that affect the random walks. For

[21] Word2vec [307] is a very influential natural language processing technique. It works by taking a large corpus of text as a sequence of words, and training a neural network to predict a word given the words that surround it, its context. This self-supervising, context–word prediction task leverages the "distributional semantics" hypothesis of linguistics [202, 159] and gives rise to "word vectors," representations that are semantically meaningful.

https://doi.org/10.1017/9781009212601.019 Published online by Cambridge University Press

instance, one must ask the following questions and decide whether to process the network further to make it more suitable for embedding methods. Is the network *connected*? If not, random walks will not connect the components and be completely disjoint across the components. As a result, the learned embedding will not be able to capture any useful information regarding the relative location of nodes in different components. Is the network *directed* or undirected? If the network is directed, then we may want to examine the prevalence of dead-ends and dangling nodes, areas where a random walk will become trapped if it enters because it has no way to exit, as such areas can create artifacts in the embedding. Dead-ends in the network may not be reached often by the random walks and therefore, the corresponding nodes may not be represented well in the embedding. Is the network *weighted*? Can the weights be interpreted in the random walk's perspective? If the network is weighted and the weights are meaningful, then we may want to make sure to use that information because random walks can be heavily affected by the weights. Is the network *bipartite*? If the network is bipartite, then the bipartite structure can produce artifacts in the embedding.

Other embedding methods also have their own requirements and assumptions. Even if they do not rely on random walk processes, it is always critical to understand the assumptions and requirements of the methods and make sure that the network is suitable for the methods. The types and structural properties of the network are important to consider.

16.7.3 Graph neural networks

Graph neural networks (GNNs) make use of both node-level attributes and the network structure by aggregating the node attributes from the neighboring nodes. Therefore, they tend to be used when rich node attributes are available. Because they make use of more information, GNNs also tend to be more powerful than simpler graph embedding methods. However, that is not always the case due to the complexity of the models and the amount of available data. When the network is small or the node attributes are not sufficiently informative, GNNs may not be able to learn how the network is organized. For instance, GAT (Graph Attention Network) needs to learn "how to pay attention" to certain nodes based on the attributes. This learning process tends to be more data-hungry and therefore it may not always work well for small network data. In such cases, simpler graph embedding methods, or even other traditional machine learning models, may be more suitable.

Some of the canonical graph neural network models are: Graph Convolutional Networks (GCN) [242], GraphSAGE [200], and GAT [475, 77]. Graph Convolutional Networks (GCN) is a generalization of convolutional neural networks developed for computer vision tasks. The idea is to use convolutional filters to aggregate node attributes from neighboring nodes. GraphSAGE allows a more flexible operation as well as ways to sample neighbors and perform inductive learning (learning on unseen nodes). Graph Attention Networks use the idea of the "attention mechanism" [474], which has been extremely successful in creating the *transformer architecture* upon which many large language models, as well as computer vision models, are now built.

We will discuss these in more detail in Ch. 26.

https://doi.org/10.1017/9781009212601.019 Published online by Cambridge University Press

16.8 Challenges and practical considerations

Above, we discussed why it is extremely important to maintain good data hygiene in supervised learning tasks. One critical challenge with network machine learning is that we often have only *one* instance of a network that is not amenable to resampling or splitting such as cross-validation. The data splitting process itself can destroy salient information that is useful to perform the machine learning task. For instance, let's assume that we are working on a link prediction problem on a social media network. Can we split this network into a training, validation, and test set? If we try to devise a method of assigning nodes or links to different sets, then we may encounter links that span between sets. For instance, suppose node i is in the training set but has several neighbors in the test set. If we train on i's degree, we will be under-counting its actual degree due to the missing test neighbors.

Likewise, if we try to construct predictive features from the network topology, it can be challenging to make them independent. If we wish to predict y as a function of two features x_1 and x_2, it will be easiest if x_1 and x_2 both give useful, *different*, information about the value of y. But many networks force relationships between features or those features are related by definition. For example, if one feature is node degree and another is clustering, these two quantities are themselves correlated. While not an insurmountable problem, predictive models typically perform better when useful features are unrelated.

Lastly, the network structure also affects the use of traditional machine learning methods applied to node attributes. Usually such methods assume observations are iid, but the network structure will introduce non-trivial relationships between the rows of an attribute matrix—the network guarantees that observations are *not* iid.

16.8.1 Feature engineering

While it's often under-emphasized in machine learning research, enough cannot be said about feature engineering. Feature engineering is the practice of devising what features to measure or include in your predictive model. In other words, what are the columns of our data matrix $\mathbf{X}$? A predictive model in general[22] is powered by a collection of variables, or features. We should take care to gather the features we need, even going back to the drawing board and gathering new data (Ch. 6).

What happens if you don't have the right feature? It is certainly possible that we missed something important, something so related to our prediction target that our performance would be far greater were the feature available. Unfortunately, it can be hard to conclude if this is the case. Fortunately, with domain knowledge of the data we can often reason, at least somewhat, about what we do have, and determine if there is enough signal among the available features.

We may also want to explore various manipulations of our input data matrix $\mathbf{X}$. It may be that applying various transformations such as taking the log will be more useful.

[22] In some problems such as computer vision, neural networks have become powerful enough to avoid the need for feature engineering. Instead, the neural network is able to develop features as needed out of the raw data. In this area, it is the neural network's form or *architecture* that needs careful "engineering."

https://doi.org/10.1017/9781009212601.019 Published online by Cambridge University Press

We could generate "indicator" features, for example finding values that are outliers and then creating a new binary feature to indicate outlier/inlier for an existing feature.

It is also helpful that some methods, particularly more flexible, black box methods like neural networks, depend less on feature engineering. In computer vision, for example, feature engineering, trying to devise useful numeric summaries of images, was the focus for many years. The advent of deep learning eliminated that, as it was shown that neural networks build both simple and complex features from an image's pixel values as they propagate through the layers of the neural network. Such feature composition is helpful for network data as well, although feature engineering still plays a role when it comes to non-network attributes that are associated with the nodes or edges of the network: even if they correlate closely with the network structure, it is still possible for useful, non-redundant information to be present in an unseen feature.

Closely related to feature engineering is *feature selection* and *feature importance*. Feature selection is the task of finding if there is a subset of our features that are most helpful for the predictive model. Selection is especially helpful in data that are very high dimensional, $\mathbf{X}$ has many columns, and we aren't sure which features are most important. Feature importance, then, determines which features contribute the most to the model's performance. Some methods, like linear or logistic regression, automatically tell us the importance of features by giving interpretable statistics for the fit parameters. Other methods, notably tree-based methods like decision trees and random forests [425, 315], have standard measures of importance because we know, inside the model, when a feature is used and when it is not. On the other hand, black box methods like neural networks are notorious when it comes to feature importance. Often it is not possible to tell which feature(s) contributed and which did not inside a neural network.

16.8.2 Model diagnostics

Suppose you've built a predictive model. Is it working well or poorly? Diagnostics can help answer this question. We've covered specific evaluation schemes like resampling methods and evaluation metrics like MSE, R^2, and F-scores in Sec. 16.6. One issue to be mindful of is when your training data are highly imbalanced. Suppose you are classifying nodes into two categories, A and B, but 90% of nodes in your example networks fall into category A. In this case, a model that always assigns a node to A will actually work very well. Indeed, it should be correct 90% of the time. But such a model is probably not going to be very useful or insightful to us. We should be mindful of data balance, take a healthy dose of skepticism, and always anticipate such explanations for good model performance.[23]

16.8.3 Dataset shift

A serious challenge facing predictive models is dataset shift, sometimes called distributional shift.[24] When the training data become outdated, a predictive model can be

[23] And another problem to be mindful of is leakage.

[24] We can further divide data shift into feature shift or covariate shift, where the input features change, and label shift, where the response or target change.

https://doi.org/10.1017/9781009212601.019 Published online by Cambridge University Press

expected to perform poorly. It won't generalize to new data because the new data have shifted away from what it was trained on.

One of the most insidious aspects of dataset shift is that resampling methods, the traditional means of inferring the performance of a predictive model, are poorly suited to capturing its effects. Suppose you used cross-validation to measure the performance of a fitted model. What you have done is taken a single dataset and divided its observations into disjoint training and testing sets.[25] But the testing set, although unseen by the model, always comes from the same pool of observations as the training set. There can be no dataset shift within resampling.

What remedies can we turn to when faced with dataset shift? Merely detecting the shift is already an important task. If model performance in practice begins to fall, we may suspect shift is to blame. With network data, we can investigate how the current network data compares to the training network data, using, for example, techniques and ideas from Ch. 14. We can also investigate if node or edge attributes (Ch. 9) have changed, a classic question of exploratory (and perhaps confirmatory) data analysis. It may also be that missingness patterns (Ch. 10) have changed.

Some research has considered remedies for dataset shift, but more work is needed. One approach has been to apply *transfer learning* methods. Transfer learning is the general idea of taking a predictive model trained for one intended task and modifying it to work on another predictive task (think of taking an AI system trained to play chess and transferring it to play Go). A shift in the training data is analogous to a shift in the task, or so the argument goes.

16.9 Summary

Machine learning has taken the world of data science by storm, and studies of network data often rely upon machine learning tools. Researchers and data scientists should be well versed in using such tools, including both general-purpose machine learning methods and methods tailored to the complexities of network data. Predictive models can play both upstream and downstream roles, helping researchers clean and process data to build their network and to study and work with the network once it is built. Methods that transform networks into meaningful representations are especially useful for specific network prediction tasks, such as classifying nodes and predicting links. Despite all the amazing advancements in machine learning, one must not lose focus on the fundamentals, including practicing good hygiene when it comes to evaluating a predictive model's performance, notable examples being recognizing leakage and detecting dataset shift. Often, a healthy dose of skepticism will go a long way to making sure your predictive models are helpful and reliable.

Bibliographic remarks

Machine learning has a rich scientific history going back to the roots of artificial intelligence. Nilsson [348] provides a fascinating tour of AI's early history. Readers

[25] Cross validation does this multiple times to average over different breakdowns of training and testing observations.

https://doi.org/10.1017/9781009212601.019 Published online by Cambridge University Press

wishing to learn more technical details are well served by Russell et al. [411], recently updated with a new addition, and the classic work of Bishop [56].

Many textbooks cover machine learning, some treating predictive models and learning in the context of traditional statistical methods and others taking the modern vantage point of neural networks and deep learning. James et al. [230] provide a fantastic introduction to statistical learning, which can be supplemented by the more technical work of Hastie et al. [205]. Hastie et al. [206] is a great, in-depth treatment of regularization, the practice of forcing overly parameterized models to generalize well, in the context of statistical learning. We continue to find surprises within over-parameterized models, including the recent exploration of double descent, which overthrows our classic intuition of overfitting [49, 506]. Another excellent book on statistical inference, Efron and Hastie [142], provides a wonderful, succinct treatment of neural networks and their training. Goodfellow et al. [186] is a useful overview of modern neural network and deep learning methods. And with so much of machine learning, including neural networks, predicated on linear algebra, in particular matrix decomposition and factorization, Strang [446] is an excellent dive into the intersection of learning and linear algebra.

It must also be said that there are profound ethical and safety concerns when it comes to using machine learning, particularly black box methods. From misuse in the criminal justice system, to automating disinformation in online social media, to accidents caused by experimental self-driving cars, AI systems have real-world, deadly consequences. O'Neil [352], give an exciting, and worrying, general audience overview of how big data and algorithms reinforce discrimination and cause societal harm. Readers wishing to avail themselves of even more existential dread may enjoy Bostrom [69], as we ponder the delightful question: Will AI research lead to a super-intelligence that causes humanity's extinction?

Exercises

16.1 Consider the polynomial regression example in Sec. 16.4. Reproduce Fig. 16.1d. Then, make a plot showing average fitting error as a function of polynomial order d. Include $d < n$, $d = n$, and $d > n$ (go out to at least $d = 2n$). Interpret the curve.

16.2 Neural networks always use nonlinear activation functions. Why? What does a neural network with linear functions end up doing?

16.3 Suppose you are analyzing a link prediction algorithm. Think of this predictive model as a binary classifier: given a pair of nodes, predict 0–no link, 1–link.

(a) You find that a baseline classifier that always predict "no link" works well. Why? What metrics would show this and what does it mean?

(b) Interpret performance: What does it tell us if the link predictor has very high *precision* but very low *recall*?

16.4 In an ideal world, training and testing data are unrelated to one another. This can be achieved in a tabular dataset, for example, when observations (rows) are independent from one another. We can then split the rows at random without

https://doi.org/10.1017/9781009212601.019 Published online by Cambridge University Press

fear of data leakage. But it's not so simple to split a dataset $\mathbf{X}$ where each row represents a node in some network G.

(a) What about G makes leakage more or less of a concern?

(b) If we were to split $\mathbf{X}$ into training and testing, what may be a good strategy for doing so?

16.5 (**Focal network**) Consider the Malawi Sociometer Network.[26] How predictable is this network? Divide the data into two parts based on time and make a weighted network for each part. Split the data so that equal numbers of days of data are covered in each part.

Build a predictive model (a classifier) to predict the presence or absence of an edge between a given pair of nodes. Use the first network as training data and the second network as test data.

(a) What model did you use? What features of the network are most useful for predicting edges?

(b) How well does your predictive model perform? Compare your predictions to a baseline model that predicts the test network is identical to the training network.

(c) (**Advanced**) How do your predictions depend on where in time the data were split?

[26] See also Exs. 15.5 and 15.6.

https://doi.org/10.1017/9781009212601.019 Published online by Cambridge University Press