Mathematical Engineering of Deep Learning

Book Draft

Benoit Liquet, Sarat Moka and Yoni Nazarathy

February 28, 2024

Contents

2 Principles of Machine Learning - DRAFT

At its core, deep learning is a class of machine learning models and methods. Hence, to understand deep learning, one must have at least a basic understanding of machine learning principles. There are dozens of general machine learning methods and models that one can cover and our purpose here is certainly not to present a detailed account of all of these methods. Instead, we take a path that presents a general overview of machine learning, and then focuses mostly on linear models which are the most elementary neural networks out there. In the process we explore gradient based learning for the first time, a topic that plays a key role in the chapters that follow.

In Section 2.1 we present an overview of the key activities of machine learning including supervised learning, unsupervised learning, and variants of these. In Section 2.2 we explore key elements of supervised learning. In Section 2.3 we introduce linear models. These form the basis for many other models as well as for the deep learning networks of this book. We then explore the basic gradient descent algorithm in Section 2.4. Linear models can often be trained without gradient descent, yet exploring gradient descent in the context of linear models is a useful warmup for the chapters that follow. In Section 2.5 we discuss generalization ability, overfitting, and introduce techniques of regularization. We further discuss the training process including splitting of the data and cross validation methods. Most of this book deals with supervised learning methods, however understanding basic techniques from unsupervised learning is also important. Hence, in Section 2.6, we take a brief look at unsupervised methods including K-means clustering, principal component analysis (PCA), and also touch the singular value decomposition (SVD).

2.1 Key Activities of Machine Learning

The world of machine learning intersects heavily with both the worlds of statistics and computer science. In statistics data and randomness are key. In contrast, in computer science, algorithms and computation are the focus. Machine learning borrows from both worlds and is about the combination of data and algorithms. It is all about training mathematical models on a computer in order to classify data, predict outcomes, estimate relationships, summarize data, control complex processes, and more.

We now present and characterize a few key activities of machine learning. These activities are carried out when training, calibrating, adjusting, or designing machine learning models and algorithms. Many of these activities are loosely called *learning* while other activities involve prediction or decision making.

Any activity of machine learning can be described as an interaction between the following entities: *data*, *models*, *algorithms*, and the *real world*. By *data* we mean both collected data

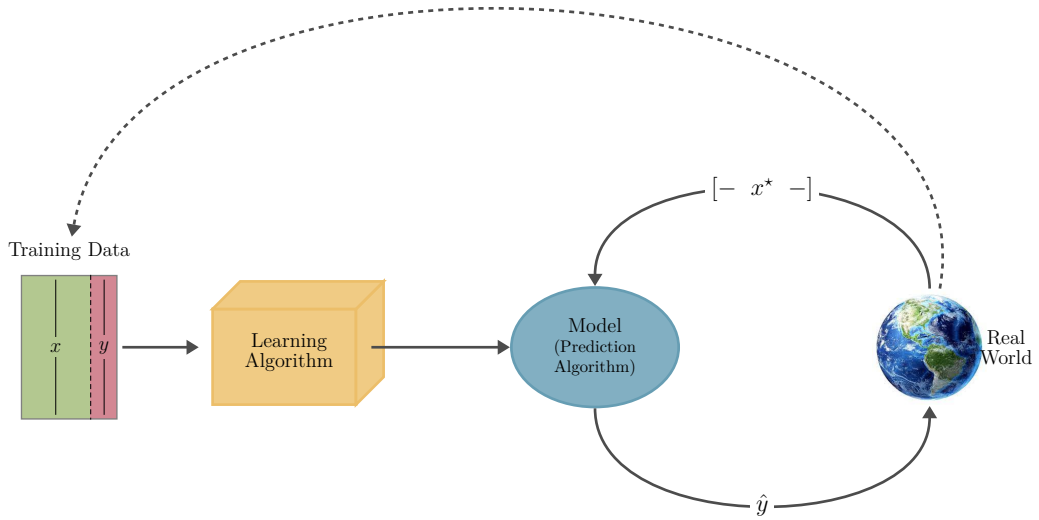


Figure 2.1: Supervised learning. Activities include training a model and prediction in production.

such as the (x, y), features-label, pairs described in the previous chapter, or data that is generated as output of models or algorithms. By *models* we mean mathematical objects stored and implemented on a computer, together with the parameters that specify these objects. By *algorithms* we mean the procedures for creating models, procedures for creating output datasets, as well as procedures for using the models themselves for prediction or related tasks. Finally, by the *real world* we refer to scenarios that support generation of data, annotation of data, as well as usage of output data for decision making, and control.

Machine learning activities are often dichotomized into two broad categories, *supervised learning*, and *unsupervised learning*. With supervised learning, data is assumed to be available as (x, y) pairs where each feature vector x is labeled via y. In the case of unsupervised learning, one only observes data points x and tries to find relationships between the various elements, variables, or coordinates of x. To understand this terminology consider the learning of babies or toddlers, which only involves the exploration of input sensory data without any indication of what is what. This is unsupervised learning since toddlers are typically not told explicitly "this does this" and "that does that". Then later on, for example during school, they engage in supervised learning since language and text are used to present the learners with examples x and their outcomes y.

A key activity in supervised learning is the usage of data to learn/train models for *prediction*. See Figure 2.1. This prediction is called *classification* in case the labels y are from a finite discrete set and it is called *regression* in case the labels y are continuous variables. There are also other cases of prediction where the labels y are vectors, images, or similar. A related activity is obviously to use the trained models for prediction when presented with unlabelled data from the real world; this is illustrated on the right of Figure 2.1. Both the training of models and usage of models for prediction involves the execution of algorithms. Sometimes the trained model is called an "algorithm" as well since it may be integrated in part of bigger systems that use it. Most of this book focuses on supervised learning and we begin in the next section, Section 2.2, by overviewing key concepts of supervised learning.

With unsupervised learning there are other activities beyond prediction, regression, and classification. See Figure 2.2. One important activity is *clustering*, which focuses on finding groups of similar data samples. The output of algorithms that perform clustering are typically not considered as models but are rather modified datasets that incorporate the clustering information. Another key unsupervised learning activity is to carry out *data reduction* where high dimensional vectors are transformed into lower dimensional vectors that still encode some of the key relationships between variables. While unsupervised learning is not the focus of this book, several unsupervised learning algorithms are overviewed in Section 2.6.

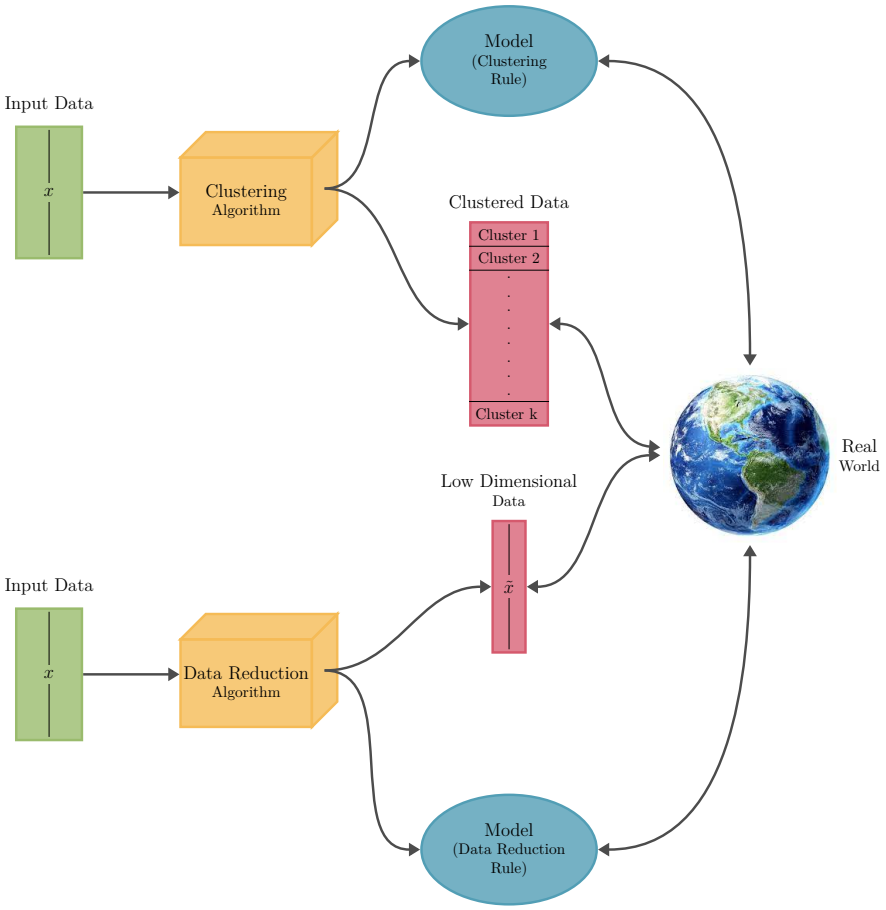


Figure 2.2: Some activities of unsupervised learning: Clustering (top) partitions the data. Dimension reduction (bottom) reduces the size of the features.

Beyond the dichotomy of machine learning into supervised and unsupervised, there are also additional popular activities that are not directly categorized as such. One popular class of activities is *reinforcement learning* introduced in Chapter 8. Here a temporal component is key and an *agent* is trained to carry out tasks in a dynamic environment. An additional class of activities is *generative modelling* also introduced in Chapter 8 in the context of variational autoencoders, diffusion models, and generative adversarial networks. Here models are trained to create artificial datasets with characteristics (or a distribution) similar to the input dataset. An additional suite of activities is *transfer learning*. Transfer learning is all about taking models that have been trained for one domain and adapting them to

other domains with new data. Related is *active learning* where the learning process is not static but is rather informed by the performance of the model on unseen data. This is very closely related to *semi-supervised learning* where like supervised learning, there are both feature vectors x and labels y, yet only a subset of the feature vectors have accompanying labels. The learning process tries to use all of the available data. Finally, *self-supervised learning*, briefly discussed in the context of deep learning natural language processing in Chapter 7, creates models where sequences of data are used to self-predict the future or missing elements of the sequences. This is useful for language models and related tasks.

Data: Seen, Unseen, Training, and Test

Data is a central part of machine learning. In considering data it is important to distinguish between *seen data* and *unseen data*. Seen data is the data available for learning, namely for training of models, model selection, parameter tuning, and testing of models. Unseen data is essentially unlimited since it is all data from the real world that is not available while learning takes place but is later available when the model is used. This can be data from the future, or data that was not collected or labelled with the seen data.

Needless to say, for machine learning to work well, the nature of the seen data should be similar to that of the unseen data. The underlying assumption of machine learning is that the seen data used to create models is generated by underlying processes of the real world that are similar to the processes generating unseen data. Practically, one needs to carry out data collection and labelling so that this resemblance between the seen and unseen data is maintained.

A common practice in the world of machine learning is to split the seen data into *training data* and *testing data*. These are sometimes called the *training set* and *test set*; an additional name for the test set is the *hold out set*. The key idea with such a split is to use the training data for learning and to use the testing data for mimicking a scenario of unseen data. As described in Section 1.4 some popular example datasets come with such a predefined split. In other cases, it is up to the machine learning engineer to split the data randomly according to some predetermined proportions. Examples follow in this chapter.

Since the purpose of the train-test data split is to mimic the unseen data with the test set, one should not recalibrate, adjust, or tune models on the training set while testing repeatedly on the test set. Carrying out such a repetitive use of the test set would invalidate its resemblance of unseen data. For this reason one sometimes performs an additional split of the training data by removing a chunk out of the training data and calling it the *validation set*. More on this practice and other alternatives such as *K-fold cross validation* is in Section 2.5.

The practice of splitting data into the train set and test set is very popular in machine learning so long as a large number of samples is available. However, in some cases there is not enough data to be able to separate a test set and thus one wishes to use all available data for model fitting. This is sometimes the case when working with experimental data and biomedical data. In such cases, statistical inference approaches for evaluating the quality of the model fit make heavier use of the model at hand. Some approaches for comparing models are *likelihood* based and include performance measures such as the *Akaike information criterion* (AIC) or *Bayesian information criterion* (BIC). The world of model fitting in a statistical content is vast and we do not focus on such methods further in this book.

Data Preprocessing

Raw data often requires *preprocessing* before it can be used for training, prediction, or as input to other machine learning models. Although a full description of data processing steps and practical aspects of data processing is beyond our scope, one important activity that we cover is *standardization of the data*, also sometimes called *normalization of the data*. This involves subtraction of the mean of each feature and division by the standard deviation of the feature.

Assume the values for some feature i are $x_i^{(1)}, \ldots, x_i^{(n)}$ where n is the number of data samples. The *sample mean* and *sample variance*[1] of the feature are respectively computed as,

$$\overline{x}_i = \frac{1}{n}\sum_{j=1}^{n} x_i^{(j)}, \qquad s_i^2 = \frac{1}{n}\sum_{j=1}^{n}(x_i^{(j)} - \overline{x}_i)^2. \tag{2.1}$$

Further, the *sample standard deviation* is the square root of the sample variance and is denoted via s_i. With these basic descriptive statistics of the feature available we may standardize the data samples of each feature $i = 1, \ldots, p$ to obtain *standardized samples*,

$$z_i^{(j)} = \frac{x_i^{(j)} - \overline{x}_i}{s_i} \qquad \text{for} \qquad j = 1, \ldots, n. \tag{2.2}$$

Now the standardized data for feature i, $z_i^{(1)}, \ldots, z_i^{(n)}$, has a sample mean of exactly 0 and a sample standard deviation of exactly 1. In the case the data samples of the feature are distributed according to a normal distribution,[2] then most standardized samples would lie in the range $[-3, 3]$. Even if the data is not normally distributed, the standardized samples will still lie in the vicinity of this range and are centered about 0.

Such standardization is useful as it places the dynamic range of the model inputs on a uniform scale and thus improves the numerical stability of algorithms. It also allows us to use similar models for different datasets that may, without standardization, have completely different dynamic ranges. In Section 2.4 we discuss how such standardization can also help optimization performance.

Learning ≈ Optimization

Almost any form of a learning or model training activity involves optimization either explicitly or implicitly. This is because learning is the process of seeking model parameters that are "best" for some given task. In fact, all of Chapter 4 is devoted to optimization techniques in the context of machine learning and deep learning, and a few of the sections of this current chapter contain aspects of optimization as well.

In some cases optimization is carried out directly on some performance measure that quantifies how good the model at hand performs. This is, for example, the case when one considers the *mean square error* criterion for regression problems, a concept which we study in detail in this chapter. However in other cases, a *loss function* is engineered for the problem at hand in a way that minimization of the loss function is a proxy for minimization of

[1] In a statistical context one often uses $n-1$ in the denominator of the sample variance instead of n. For non-small n this distinction is insignificant.

[2] A few attributes of the normal distribution, also known as the Gaussian distribution, are in Appendix B.

the actual performance measures that are of interest. This is, for example, the case when considering classification problems and aiming to get the most accurate classifier. In such a case, optimization is typically not carried out directly on the accuracy measure but rather on a loss function such as the mean square error, or cross entropy defined in Chapter 3. In any case, the design of loss functions as part of the learning procedure is central to machine learning and deep learning and appears throughout this book.

Note that in some cases, machine learning algorithms such as *decision trees* do not directly specify an optimization procedure but rather execute a predefined algorithm for fitting a model. However, even in such cases, there is typically an inherent hidden optimization problem associated with the procedure. Hence in general we can think of "learning" as the process of carrying out some sort of "optimization".

2.2 Supervised Learning

We now focus on supervised learning and outline key concepts, practices, and terminology. Supervised learning is about predicting an outcome $\hat{y}$ for y, where the prediction is based on a vector of input features x. When y is from a finite discrete set then the task is called *classification* and when y is a continuous variable then the task is called *regression*.

We begin with overviewing basic regression in the context of linear models and feature engineering. We then discuss aspects of binary classification which is the case when y only attains one of two possibilities. The more general *multi-class classification* case in which y takes on multiple possibilities is presented as part of specific examples in Section 2.3. We close this section with a high level overview of several methods and general approaches to supervised learning.

Regression and Feature Engineering

We begin by considering a very simple *univariate* example where the scalar ($p = 1$) feature x is the average number of rooms per dwelling and y is the median value of owner-occupied homes in thousands of dollars. These variables, respectively denoted `rm` and `medv`, represent data from the well-known Boston housing dataset.[3] A regression model $y = f_\theta(x)$ attempts to predict the median house price as a function of the average number of rooms.

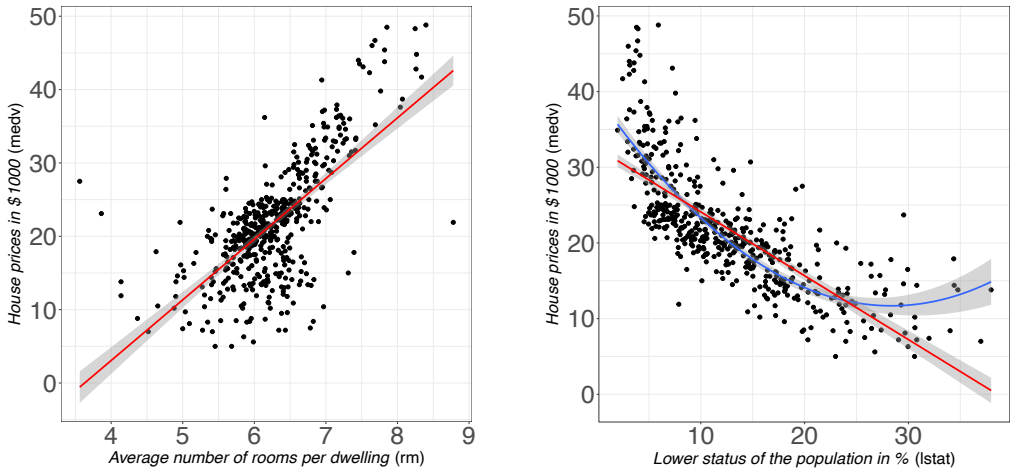
To illustrate this concept, consider the well-known *simple linear regression model* where $f_\theta(x) = \beta_0 + \beta_1 x$ and the parameter vector is $\theta = (\beta_0, \beta_1)$. Notice that in this case the dimension of the parameter vector is $d = 2$. This model can also be described statistically via,

$$y = \beta_0 + \beta_1 x + \epsilon, \tag{2.3}$$

where ϵ represents the *error* or noise term as it models the gap between y and $f_\theta(x)$. In statistical theory and practice, assumptions about the probability distribution of ϵ go a long way as they support inference outputs such as *confidence bands*, *hypothesis tests*, and more. However in practical machine learning culture, one often ignores ϵ and such statistical assumptions.

[3] This dataset originally published in [161], has 506 observation where each observation is associated with a suburb or town in the Boston Massachusetts area. Of these observations 16 are capped at `rm` $= 50$ and we remove these to stay with $n = 490$ observation.

Provided feature data $x^{(1)}, x^{(2)}, \ldots, x^{(n)}$ (`rm` – average number of rooms per house in a geographical area), and corresponding label data $y^{(1)}, y^{(2)}, \ldots, y^{(n)}$ (`medv` – median house prices in a geographical area), the training process involves finding a suitable or best $\hat{\theta} = (\hat{\beta}_0, \hat{\beta}_1)$. In Section 2.3 we study the process for finding $\hat{\theta}$ via minimization of a loss function, yet at this point let us just consider the *model parameters*, also known as *parameter estimates* $\hat{\theta}$, as an outcome of training.



(a) **(b)**

Figure 2.3: Examples of elementary linear models. (a) Median house prices per locality (`medv`) as a function of average number of rooms per dwelling (`rm`) is described via a simple linear (affine) relationship. (b) House prices as a function of lower status of the population in % (`lstat`) is not described well with a linear relationship (red), but by introducing an additional quadratic engineered feature it is described well via a three parameter linear model resulting in a quadratic fit (blue).

Figure 2.3 (a) presents a scatter of the $(x^{(i)}, y^{(i)})$ pairs. The parameters estimated for this model are $\hat{\beta}_0 = -30.01$ and $\hat{\beta}_1 = 8.27$. The figure also includes a plot of the fit or estimated model $f_{\hat{\theta}}(\cdot)$ as a red line. Clearly, with such a model, any unseen (new) observation $x^\star$ can be used to make a prediction $\hat{y} = f_{\hat{\theta}}(x^\star)$. If one is willing to make statistical assumptions about the error ϵ and probabilities of error then an extra benefit of the model is the confidence bands, presented as a the gray shaded area around the red line. Most of the modelling described in this book uses very complex models that do not take such a statistical approach and hence built-in inference outputs such as these confidence bands are often not available.

We also mention that the estimated parameters in such a model have an *interpretation*. For example $\hat{\beta}_1 = 8.27$ indicates that increasing the average number of rooms by one room implies an average rise in median price of $\$8.27K$. Many types of statistical models have the benefit of interpretable parameters, yet in the world of machine learning where models are often very complex, parameter interpretation is an exception rather than the rule.

As a follow up example arising from the same dataset, consider the relationship between the variable x taken as the percentage of the population that is of a low social economic status (`lstat`) and the variable y taken again as `medv`. A simple linear model fit to this will yield parameter estimates $\hat{\theta}$ since in almost any case one can fit any model to data. However the model may not always be suitable for the data or process at hand. For example, in

Figure 2.3 (b), a scatter plot of the data is presented and it is apparent that the downward sloping linear model fit in red does not do a good job in describing the relationship between x and y. Observe also that confidence bands for this simple linear model may look deceptively appealing (tight gray bands around the red line) especially if one was only to look at these and not the actual scatter plot of the data. The pitfall is that these confidence bands are computed under the assumption that the model fits the underlying process and data well, a case that does not hold here. Such a phenomena is often loosely called *model misspecification* and is one of the risks that one undertakes (and needs to mitigate) when using statistical inference techniques.

An alternative to the simple linear model is to seek a richer relationship such as,

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \epsilon. \tag{2.4}$$

One way of describing this relationship is via the function $f_\theta(\cdot)$ defined for the $d = 3$ dimensional θ via, $f_\theta(x) = \beta_0 + \beta_1 x + \beta_2 x^2$ where x is still a scalar ($p = 1$). An alternative description of the same model is to consider the squaring of the `lstat` variable as a new *engineered feature* and thus now consider x as a $p = 2$ dimensional vector with x_1 being the original feature and $x_2 = x_1^2$ the new engineered feature. In this case the model function is linear (affine) $f_\theta(x_1, x_2) = \beta_0 + \beta_1 x_1 + \beta_2 x_2$, and the fact that x_2 is the square of x_1 is considered a feature engineering aspect and not a model function aspect. In practice, in this case, both approaches are identical and yield the same θ. Figure 2.3 (b) presents the fit and corresponding error bands of this quadratic model in blue.

We mention that linear models for regression, which are the workhorse of classical statistics, can be extended in many ways. The notes at the end of this chapter point at some extensions such as *Generalized Linear Models* (GLM), *mixed models*, and more. Note also that non-linear relationships in a regression context could be explored using smoothing techniques. Popular techniques in this framework are the *Generalized Additive Model* (GAM), the *Locally Estimated Scatterplot Smoothing* (LOESS) method, as well as *Nadaraya-Watson kernel regression*. We also mention that generally when one considers feature engineering, one very important aspect is dealing with *interaction terms*. This means creating new engineered features that are based on the products of other features.

The world of machine learning has adopted these models often removing statistical assumptions (e.g. about the noise ϵ) while introducing additional non-linearities and mechanisms that yield very expressive models. The deep neural networks that we cover in this book include one such rich class of examples. We also mention that while the numerical house price context that we presented here appears to be simple and low dimensional, regression problems can often involve extremely high dimensional input feature vectors. For example, any regression problem where the input data is an image is of this nature. A concrete example of such a case is using images of a human face to predict the age of the person.

Binary Classification

Moving on from regression problems where y is continuous, we now consider binary classification where y attains one of two values, which are sometimes referred to as `positive` or `negative`. There are dozens of machine learning methods for binary classification and our purpose here is not to explore how these methods work. Instead we wish to illustrate how their performance is quantified.

Our exposition relies on a *logistic regression* based classifier where `positive` samples are encoded via $y = 1$ and `negative` samples are encoded via $y = 0$. Note that, in other scenarios positive and negative samples are sometimes encoded via $y = +1$ and $y = -1$, respectively. Logistic regression is explored in depth in sections 3.1 and 3.2 of Chapter 3. For now, we can treat such a classifier as being based on a function $f_\theta(x)$ where x is the feature vector. The output of $f_\theta(x)$ is a number in the continuous range $[0, 1]$ indicating the probability that x matches a `positive` label. Hence, the higher the value of $f_\theta(x)$, the more likely it is that the label associated with x is $y = 1$.

With the model $f_\theta(\cdot)$ at hand, a classifier can be constructed via a *decision rule* based on a threshold τ, with the predicted output being,

$$\widehat{\mathcal{Y}} = \begin{cases} 0 \text{ (}\texttt{negative}\text{)}, & \text{if} \quad \hat{y} \leq \tau, \\ 1 \text{ (}\texttt{positive}\text{)}, & \text{if} \quad \hat{y} > \tau, \end{cases} \qquad \text{where} \qquad \hat{y} = f_\theta(x). \tag{2.5}$$

In many cases one selects the threshold at $\tau = 0.5$. However as we see below, τ can often be adjusted. Also note the notational difference between $\widehat{\mathcal{Y}}$ and $\hat{y}$. Throughout the book, we use the former to signify the actual predicted label for classification problems whereas we use the latter to denote the output of the model which is usually (continuous) numerical in nature.

As an example we consider breast cancer prediction where the label, or outcome variable y has $y = 1$ in case of malignant lumps and $y = 0$ in case of benign lumps. A popular dataset in this context is called the *Wisconsin Breast Cancer Dataset* and is based on clinical data released in the early 1990s.[4] We make further use of this dataset in Section 3.1 where we dive into logistic regression. The feature vector x is of dimension $p = 30$ and is composed of continuous variables such as `radius_mean`, `texture_mean`, etc., each potentially affecting the probability of malignancy. The data has $n = 569$ observations and here we use the *80-20 splitting strategy* where we split it randomly between training data and testing data to have $n_{\text{train}} = 456 \approx 0.8 \times n$ training observations and $n_{\text{test}} = 113 \approx 0.2 \times n$ testing observations. Note that in some of the sections below, we simply use n to denote n_{train}, when not considering the test set explicitly .

We may train different forms of logistic regression for this data where we take x to be some subset or transformation of the original feature vector. At one extreme we can take x to be a single variable, and at another extreme x can be the full feature vector or even have additional engineered features similar to the home pricing example above.

Here for simplicity we consider two logistic regression models. For the first model we use only a single feature in x which is `smoothness_worst` where the actual physical meaning of this variable is not critical for our understanding at this point. This model is denoted $f_{\theta;\, p=1}(\cdot)$. In the second model we consider all 30 features in the dataset. This model is denoted $f_{\theta;\, p=30}(\cdot)$. For each of these models we use the n_{train} observations to obtain an estimated parameter vector where in the case of $p = 1$, the estimated parameters $\hat{\theta}$ is of dimension $d = 2$ and in the case of $p = 30$ the estimated parameters $\hat{\theta}$ are of dimension $d = 31$. Details on the actual meaning of the models and parameters of logistic regression are in Chapter 3. With these models at hand, at first let's fix τ and consider several performance measures of the classifiers defined via (2.5).

[4] See https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(Diagnostic) for more information and relevant references.

The standard way to compute binary a classification performance measure is to evaluate the classifier on the test set. This then allows us to consider the predictions $\widehat{\mathcal{Y}}^{(i)}$ and compare them to the test set labels $y^{(i)}$, for $i = 1, \ldots, n_{\text{test}}$. Observations where $\widehat{\mathcal{Y}}^{(i)} = y^{(i)}$ are counted as either *True Positive* (TP) or *true negative* (TN), depending on the value of $y^{(i)}$ being 1 or 0 respectively. Similarly, observations where $\widehat{\mathcal{Y}}^{(i)} \neq y^{(i)}$ are counted as either *False Positive* (FP) or *False Negative* (FN). These four counts, TP, TN, FP, and FN total up to n_{test} and are customarily summarized in the *2 by 2 confusion matrix*:

		Decision		
		Decide 0	Decide 1	
Reality	Label 0	True Negative (TN)	False Positive (FP)	Specificity $\frac{\text{TN}}{\text{TN+FP}}$
	Label 1	False Negative (FN)	True Positive (TP)	Sensitivity/Recall $\frac{\text{TP}}{\text{TP+FN}}$
		Negative Predictive Value (NPV) $\frac{\text{TN}}{\text{TN+FN}}$	Precision/Positive Predictive Value (PPV) $\frac{\text{TP}}{\text{TP+FP}}$	

Observe the ratios at the margins of the matrix that present various performance indicators, namely *sensitivity* (also known as *recall*), *specificity*, *precision* (also known as *positive predictive value*), and *negative predictive value*. In general, we would like all of these ratios to be as close to 1 as possible, however as we describe below, there are often tradeoffs.

An additional natural performance measure is the *accuracy*. It is simply defined as the proportion of correctly classified samples, namely,

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{n_{\text{test}}}. \tag{2.6}$$

This is often the first performance measure one considers[5], yet for unbalanced data it can be an extremely misleading measure. Indeed, a degenerate classifier that always predicts the most abundant class will have an accuracy equal to the proportion of that class. For example in binary classification if the `positive` class constitutes only 5% of the samples, then the degenerate classifier which always predicts $\widehat{\mathcal{Y}} = 0$ will have an accuracy of 95%. Then, even when considering a non-degenerate classifier, one observes an accuracy that is typically not worse than 95% and if for example the accuracy is 99% it is still not an indication that the classifier works well.

A more robust analysis of performance considers the competing objectives of *sensitivity* and *specificity*, where a high sensitivity (or recall) value indicates a good ability of the classifier to detect `positive` samples and a high specificity value indicates a good ability to detect `negative` samples. In the case of a threshold based classifier as in (2.5), varying τ alters the confusion matrix and thus the sensitivity and specificity values are modified as well.

A parametric curve based on τ known as the *Receiver Operating Characteristic (ROC) curve* is often used to visualize this tradeoff between sensitivity and specificity where the x-axis

[5]Note that the formula here is for binary classification, yet this performance measure is also used for multi-class classification where the numerator TP+TN needs to be replaced by the total number of correctly classified samples.

plots one minus the specificity also known as the *false positive rate*. Observe from (2.5) and the formulas at the bottom margin of the confusion matrix, that as $\tau \to 0$ the number of false negatives vanishes and hence the sensitivity approaches 1. Similarly as $\tau \to 1$ the number of false positives vanishes and hence the specificity approaches 1. More generally, as we vary τ between 0 and 1 a tradeoff emerges as captured in the ROC curve. This allows us to tune the threshold τ for balancing sensitivity and specificity, depending on the problem at hand. Figure 2.4 presents (smoothed) ROC curves for the breast cancer example where we compare the ROC curves for the $f_{\theta;\, p=1}(\cdot)$ and $f_{\theta;\, p=30}(\cdot)$ models, as well as a "coin flip" model (chance line) and the "ideal" model.

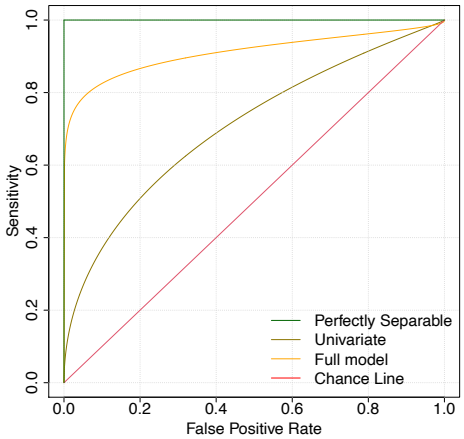


Figure 2.4: Receiver operating characteristic (ROC) curves for the breast cancer data. One model is a univariate model, and the other is a full model. A chance line (guessing model) and a perfectly separable line (ideal model) are also plotted. For each model, the ROC captures the tradeoff between the sensitivity and the false positive rate (one minus specificity).

Receiver operating characteristic curves allow us to asses the quality of models taking all possible threshold parameters into account. A related measure that tries to quantify the quality of a curve into a single number is the *area under the curve* (AUC) measure. For a classifier with an ROC curves that achieves perfect sensitivity under any level of specificity this measure is at 1 and corresponds to the perfectly separable green curve in Figure 2.4. However for classifiers that just choose a random class, this measure is at 0.5 corresponding to the chance line red line in Figure 2.4. In the case of the breast cancer data we see that on the test set the AUC for the $f_{\theta;\, p=1}(\cdot)$ model is 0.70 and for the $f_{\theta;\, p=30}(\cdot)$ model it is at 0.92. This may give an indication that the additional features in the richer model help obtain a better predictor.

Let us now fix the threshold at $\tau = 0.5$ and compare a few more performance measures. The test accuracy in this case is 0.73 for the $f_{\theta;\, p=1}(\cdot)$ model and 0.89 for the $f_{\theta;\, p=30}(\cdot)$ model. However, since the number of `positive` samples in the test set is 40 (out of $n_{\text{test}} = 113$), we see that this dataset is somewhat unbalanced and hence accuracy is not a good measure of performance. In such cases, machine learning practice typically focuses on both precision and recall (sensitivity). Note that this could have alternatively been a focus on specificity and sensitivity (recall), but in machine learning the precision–recall pair is more popular. Precision, similarly to specificity approaches 1 as the number of false positives FP approaches 0. However precision is based on the true positives number (TP), while specificity is based on the true negatives (TN) value.

For the $f_{\theta;\, p=1}(\cdot)$ model with $\tau = 0.5$ we have,

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} = 0.70, \qquad \text{and} \qquad \text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} = 0.4,$$

and for the $f_{\theta;\, p=30}(\cdot)$ model with $\tau = 0.5$ we have Precision $= 0.82$ and Recall $= 0.9$.

A popular way to consider both precision and recall is by averaging them using the harmonic mean of the two values. This is called the F_1 *score* and is computed as follows:

$$F_1 = \frac{2}{\frac{1}{\text{Precision}} + \frac{1}{\text{Recall}}} = 2\frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}. \tag{2.7}$$

In our example for the $f_{\theta;\, p=1}(\cdot)$ model with $\tau = 0.5$ we have, $F_1 = 50.8\%$ and for the $f_{\theta;\, p=30}(\cdot)$ model with $\tau = 0.5$ we have $F_1 = 85.7\%$. Note that one may also use F_1 scores to calibrate the threshold τ. Sometimes one uses a generalization of F_1 called the F_β where β determines how much more important recall is in comparison to precision. However, in general, if there is not a clear reason to price false positives and false negatives differently, then using the F_1 score as a single measure of performance is sensible.

In general, cases of unbalanced data should be treated with caution not just in terms of performance measures and threshold calibration, but also in terms of inference. There are multiple techniques for handling unbalanced data, some of which include over-sampling or under-sampling to balance the data. One of the more popular techniques is called *synthetic minority oversampling technique* (SMOTE). See the notes and references for further details.

Approaches and Algorithms for Supervised Learning

We cannot cover all aspects of supervised learning in a single section, a single chapter, or even in a single book. Yet now, after getting a taste of supervised learning in the context of regression and classification, let us discuss a few general approaches for supervised learning. Towards that end we first distinguish between *discriminative models* and *generative models*. Most of the models in this book are discriminative. This means that when viewed through a probabilistic lens (even though we mostly do not do that), they are based on learning aspects of the distribution $\mathbb{P}(y \mid x)$, i.e. the conditional distribution of the label y given the feature vector x. This is the case for linear models, logistic regression, general neural networks, and multiple additional models and algorithms that are not covered in this book.

In contrast, generative models involve learning the joint distribution $\mathbb{P}(x, y)$. As a byproduct knowledge of $\mathbb{P}(x, y)$ also means knowing the marginal distributions $\mathbb{P}(x)$ and $\mathbb{P}(y)$ as well as the conditional distributions $\mathbb{P}(y \mid x)$ and $\mathbb{P}(x \mid y)$. Hence generative models consider all of the data relationships as being learned, not just $\mathbb{P}(y \mid x)$. In this book, the most prominent appearance of generative models is in the context of variational autoencoders, diffusion models, and generative adversarial networks, appearing in Chapter 8. Another elementary generative type of model that we do not cover is the famous *naive Bayes classifier*, most notably known for early success of e-mail spam detection applications. One more common type of generative model is *linear discriminant analysis* (LDA) used in many experimental statistical contexts.

Naive Bayes classifiers are based on certain independence assumptions for $\mathbb{P}(x, y)$. We assume that given the label y, all features $x_1, \ldots, x_p$, are mutually independent. This (naive) independence assumption then allows us to represent the likelihood function[6] of the sample easily and in turn this enables efficient generative learning. LDA-based classifiers are also generative models (even though the name "discriminant" might be misleading). These classifiers fit a multivariate normal model to the data to carry out classification based on linear boundaries. Further details of both of these classifier models and algorithms are beyond our scope.

In terms of discriminative models, the linear models, logistic regression models, and more general deep learning models in this book are very common examples. Linear models and logistic regression models are simple deep neural networks. Linear models are studied in detail in this chapter. Logistic regression models and generalizations are the focus of Chapter 3, general fully connected deep learning models are the focus of Chapter 5, and other specialized deep learning models are in chapters 6 and 7. Beyond these deep learning models, other types of popular machine learning models that we do not cover in the book include *support vector machines (SVM)*, *decision trees*, and their generalizations, which include *random forests* as well as *gradient boosted trees*. There are also additional elementary models often used for instruction of machine learning such as the class of *K-nearest neighbours* classification models. Indeed, the world of machine learning is vast with ideas and algorithms for creating both discriminative and generative models. The notes and references at the end of this chapter point at key resources.

2.3 Linear Models at Our Core

In this section, we focus on linear models which are the basis for many other models including deep neural networks. This is the first model in the book where we explicitly use a loss function for learning. The basic principles of the linear model and the associated loss function alternatives extend to more advanced models covered in the sequel. Similarly, other concepts that we cover here in the context of the linear model, such as the treatment of categorical variables and aspects of multi-class classification, are also relevant for more advanced models.

For the linear model, let us consider a feature vector $x = (x_1, \ldots, x_p) \in \mathbb{R}^p$ and the output/response variable $y \in \mathbb{R}$. The linear model links the output y to the features x through

$$y = b + w^\top x + \epsilon \tag{2.8}$$

where the scalar parameter b is called the *intercept* or *bias*, the vector parameter w is called the *regression parameter* or *weight vector*, and the ϵ term represents the noise or error. This is a generalization of the simple linear regression model (2.3) allowing x to be a vector and we now denote β_0 via b. To facilitate the presentation of key concepts of linear models we often use a more compact representation of the model,

$$y = \begin{bmatrix} b & w_1 & \ldots & w_p \end{bmatrix} \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_p \end{bmatrix} + \epsilon = \theta^\top \tilde{x} + \epsilon, \tag{2.9}$$

[6] The likelihood function is a basic statistical concept that we survey in Chapter 3 in the context of logistic regression.

where $\theta = (b, w_1, \ldots, w_p)$ encapsulates both b and w and the feature vector x is extended to $\tilde{x}$ via a constant unit in its first position. The dimension of θ is the number of parameters and in this case it is $d = p + 1$.

In order to use the linear model for prediction of unseen data we have to *learn* the model. This means to find appropriate values for the parameters in θ based on training data such that the model performs well in prediction. Such a suitable learned parameter is further denoted via $\hat{\theta}$ and with such an estimate at hand, a prediction for a new data point $x^\star \in \mathbb{R}^p$ is given by,

$$\hat{y}(x^\star) = \hat{b} + \hat{w}_1 x_1^\star + \ldots + \hat{w}_p x_p^\star = \hat{\theta}^\top \tilde{x}^\star.$$

Learning the Linear Model

Consider a training dataset $\mathcal{D} = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$ composed of a collection of n samples. For such data it is convenient to define the $n \times d$ dimensional *design matrix* X for the features, and the corresponding output response vector y, via,

$$X = \begin{bmatrix} | & | & & | \\ 1 & x_{(1)} & \ldots & x_{(p)} \\ | & | & & | \end{bmatrix} \quad \text{with} \quad x_{(i)} = \begin{bmatrix} x_i^{(1)} \\ \vdots \\ x_i^{(n)} \end{bmatrix}, \quad \text{and} \quad y = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(n)} \end{bmatrix}. \tag{2.10}$$

Using this notation we can express the linear model for all the samples of the training set via

$$y = X\theta + \epsilon,$$

with $\epsilon = (\epsilon_1, \ldots, \epsilon_n)$ representing a vector of noise. From this representation given a learned parameter vector $\hat{\theta}$, we can further define the predicted output vector of the model for the input training data via,

$$\hat{y} = X\hat{\theta}, \qquad \text{where} \qquad \hat{y} = \begin{bmatrix} \hat{y}^{(1)} \\ \vdots \\ \hat{y}^{(n)} \end{bmatrix}.$$

A suitable value for $\hat{\theta}$ will yield $\hat{y} \approx y$. This closeness is captured via a *loss function*,

$$C(\theta\,;\mathcal{D}) = \frac{1}{n}\sum_{i=1}^{n} C_i(\theta), \tag{2.11}$$

where $C_i(\theta) := C_i(\theta\,; y^{(i)}, \hat{y}^{(i)})$ is the loss for the i-th data sample. Specifically $\hat{\theta}$ is typically chosen so that the loss function is minimal at the point $\theta = \hat{\theta}$.

For the linear model, the most popular loss function is the *square loss* function also called *quadratic loss* where the loss for each data sample is

$$C_i(\theta) = (y^{(i)} - \hat{y}^{(i)})^2. \tag{2.12}$$

This loss penalizes each element $e^{(i)} := y^{(i)} - \hat{y}^{(i)}$, also known as the *error* or *residual* for sample i, quadratically. In this case, the loss for the entire training data can be represented

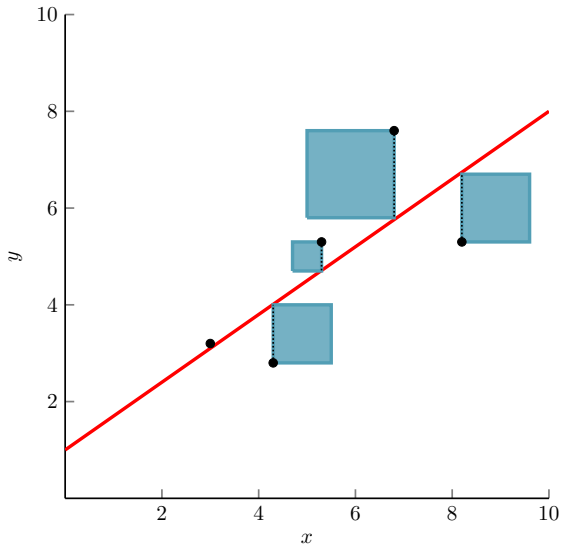


Figure 2.5: Squared loss visualisation for one input feature $p = 1$. The sum of the area of the squares is the loss.

in terms of the L_2 norm $\|\cdot\|$ of the corresponding error vector,

$$C(\theta\,;\mathcal{D}) = \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2 = \frac{1}{n}\|y - \hat{y}\|^2 = \frac{1}{n}\|e\|^2.$$

With this notation, by treating the learning of θ as an optimization problem and observing that the objective can be manipulated via monotonic transformations, we can now represent the learned parameter vector as,

$$\hat{\theta} = \underset{\theta\in\mathbb{R}^d}{\operatorname{argmin}}\ \frac{1}{n}\|y - X\theta\|^2 = \underset{\theta\in\mathbb{R}^d}{\operatorname{argmin}}\ \|y - X\theta\|^2. \tag{2.13}$$

The search for θ that optimizes (2.13) is known as the *least squares problem*. Figure 2.5 presents a visual representation of the squared loss in the case of a single input feature ($p = 1$ and $d = 2$). In this case we seek a line specified by b and w_1 such that the sum of the squares (total area of blue boxes in the figure) is minimized.

The least squares solution can be easily derived by first computing the gradient of $\|X\theta - y\|^2$ with respect to θ using vector and matrix differentiation rules (see Appendix A) as

$$\begin{aligned}\frac{\partial \|y - X\theta\|^2}{\partial \theta} &= \frac{\partial (y - X\theta)^\top (y - X\theta)}{\partial \theta} \\ &= \frac{\partial \left(y^\top y - 2y^\top X\theta + \theta^\top X^\top X\theta\right)}{\partial \theta} \\ &= -2X^\top y + 2X^\top X\theta.\end{aligned} \tag{2.14}$$

Then, by setting the gradient to 0, we get the *normal equations*,

$$X^\top X\theta = X^\top y, \tag{2.15}$$

which describe vectors θ that obtain a zero gradient, and, in this case, it can also be shown that they are global minima of the objective (see further discussion in Chapter 4 about global and local minima).

The normal equations have a unique solution when the $d \times d$ matrix $X^\top X$, also known as the *Gram matrix* of X, is invertible. In this case we can represent the estimator as $\hat{\theta} = (X^\top X)^{-1} X^\top y$, or, by setting $X^\dagger = (X^\top X)^{-1} X^\top$, we have,

$$\hat{\theta} = X^\dagger y, \tag{2.16}$$

where $X^\dagger$ is called the *Moore-Penrose pseudo inverse* of X.

In fact, the Moore-Penrose pseudo-inverse, $X^\dagger$, can be represented in different ways. An alternative form to $(X^\top X)^{-1} X^\top$ is based on the *singular value decomposition*[7] (SVD) of X. Here, $X = U\Delta V^\top$ where U is an $n \times n$ orthogonal matrix, Δ is an $n \times d$ matrix with non-zero elements only on the main diagonal, and V is a $d \times d$ orthogonal matrix. Using the SVD we can represent the Moore-Penrose pseudo-inverse as

$$X^\dagger = V\Delta^+ U^\top, \tag{2.17}$$

where Δ^+ contains the reciprocals of the non-zero (main diagonal) elements of Δ, and has 0 values elsewhere. This SVD-based representation holds both if $X^\top X$ is singular or not. Hence (2.17) can be viewed as the more general representation of the pseudo-inverse. Note that $X^\top X$ is non-singular if and only if the matrix X is a full column rank matrix (i.e., the columns of X are linearly independent).

Note that, if X is not full column rank (i.e. $X^\top X$ is singular), then there is not a unique solution to the normal equations (2.15). However, the solution given via (2.16), using the SVD form (2.17), has a minimal norm for θ out of all possible solutions.

In the context of high dimensional data when the number of features p is greater than the number of samples n, the design matrix X is never full column rank. This issue also appears when some of the features are linear combinations of the others. Even if X is mathematically full column rank, in some situations there is (strong) *multicollinearity* among the features, meaning that some of the features are approximately linear combinations of the others. This

[7] Note that the form of SVD presented here is sometimes called the *full SVD*. A different form called the *reduced SVD* is used in Section 2.6 in the context of PCA.

yields an $X^\top X$ matrix that is *ill-conditioned* and difficult to invert. In all these cases, the SVD-based representation (2.17) is useful for using in (2.16) to obtain a solution for (2.15).

Other Loss Functions

A first appealing result for the choice of the squared error loss function (2.12) is the closed form solution (2.16) for (2.15). Also, when it is assumed that y is measured with uncorrelated Gaussian noise ϵ, using the squared error loss function (2.12) is equivalent to using a solution derived by the *maximum likelihood estimation method*. This is a technique widely used in statistics for parameter estimation which is further discussed in Section 3.1 in the context of logistic regression.

A second popular loss function for the linear model is the *absolute error loss*,[8]

$$C_i(\theta) = \left|y^{(i)} - \hat{y}^{(i)}\right|. \tag{2.18}$$

It is known to be more robust to outliers, however, even in the case of the linear model, there is no closed-form solution for estimating the parameters. From a statistical point of view, minimizing the *absolute error loss* is equivalent to maximum likelihood estimation when assuming a Laplace distribution[9] for the error noise ϵ. The Laplace distribution has heavier tails than the Gaussian distribution, see Figure 2.6 (b). With these tails, large noise values, i.e. outliers, are more probable than with the Gaussian noise and hence the loss (2.18) is typically more robust to extreme values.

A third alternative, which is a hybrid between the absolute error loss and squared error loss, is the *Huber error loss*. It is parameterized by a hyper-parameter δ and represented via,

$$C_i(\theta) = \begin{cases} \frac{1}{2}(y^{(i)} - \hat{y}^{(i)})^2, & \text{if } \left|y^{(i)} - \hat{y}^{(i)}\right| < \delta, \\ \delta\left|y^{(i)} - \hat{y}^{(i)}\right| - \frac{1}{2}\delta^2, & \text{otherwise.} \end{cases} \tag{2.19}$$

This loss function penalizes small errors quadratically and deals with outliers by penalizing larger errors similarly to the absolute error loss. Figure 2.6 (a) provides a visual representation of these three loss functions. Even if it appears to be an appealing tradeoff between the absolute error loss and squared error loss, the Huber loss has the disadvantage of having to calibrate the arbitrary extra hyper-parameter δ and, like the absolute error loss, it does not have a closed form solution.

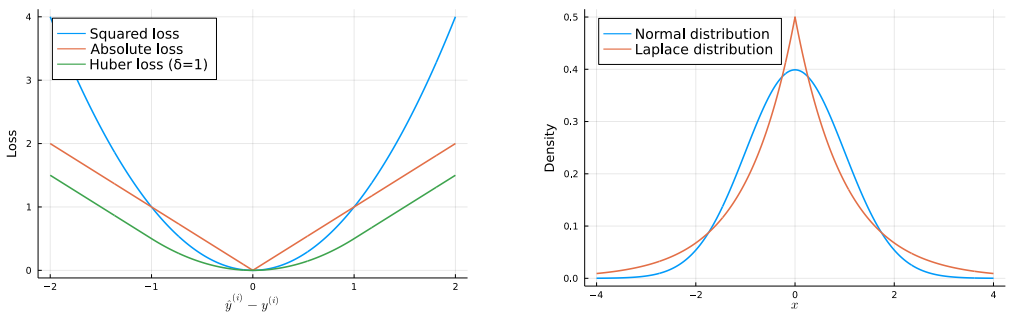
Categorical Input Features

We have so far considered numerical input features. We now describe methods for dealing with categorical input features. The methods we present are useful for linear models as well as almost any machine learning and deep learning model.

Before describing the general method of using *one-hot encoding*, let us highlight two cases that sometimes receive special treatment. One such case is when the categorical feature is

[8] Note that the use of the absolute error loss (2.18) in (2.11) is sometimes called the L_1 loss as it is related to the L_1 norm. Similarly the use of the square loss (2.12) is sometimes called the L_2 loss.

[9] This is a probability distribution over $\mathbb{R}$ with density function in the variable u, proportional to $e^{-\frac{|u-\mu|}{b}}$, where $\mu \in \mathbb{R}$ is the mean and $b > 0$ is a scaling parameter.



(a) **(b)**

Figure 2.6: Loss function and error distribution alternatives. (a) Squared, absolute, and Huber loss functions. (b) Gaussian (normal) and Laplace error distributions.

binary. For example, assume a feature that only takes on two values `red` or `blue`. In such a case, it can be encoded via 0 and 1 respectively and used as a numerical variable. A second special case is when the categorical feature is an ordinal variable, which may be interpreted as or converted to a numerical value. For example if the feature is a "user satisfaction rating" with values `low`, `medium`, and `high`, it may be transformed to numerical values 0, 1, and 2. Note however that this practice is sometimes problematic since different spacings between the assigned numerical values would yield different interpretations of the features and in general yield different models. For example assigning numerical values of 0, 1, and 4 would indicate a bigger gap between `medium` and `high` than between `low` and `medium`.

Moving on to the general case of non-binary, nominal, categorical features one can use one-hot encoding, a method that we present now. Denote the number of possible values that the feature attains via L and here for simplicity assume the feature values are `1`, ..., `L`. The idea is to create L binary features in place of the categorical feature where, if the categorical feature is z, we construct an L-dimensional vector $\tilde{z} = \big(\mathbf{1}\{z = \texttt{1}\}, \ldots, \mathbf{1}\{z = \texttt{L}\}\big)$ with $\mathbf{1}\{\cdot\}$ denoting the indicator function taking on 0 or 1. That is, $\tilde{z} = e_z$ where e_z is the unit vector with 1 in the position z and 0 elsewhere; see the unit vector notation defined in Section 1.6.

Thus, each one-hot encoded categorical feature is expanded into such a vector and, with this encoding, the new transformed total number of features is,

$$\tilde{p} = p_{\text{num}} + \sum_{i \text{ categorical}} L_i,$$

where p_{num} is the number of numerical features and L_i is the number of possible values for categorical feature i.

To see how this one-hot encoding affects the design matrix, assume for simplicity that in addition to the p_{num} numerical features there is a single categorical feature with L levels. In this case the $n \times (1 + p_{\text{num}} + L)$ dimensional design matrix is,

$$X = \begin{bmatrix} | & | & & | & | & & | \\ 1 & x_{(1)} & \cdots & x_{(p_{\text{num}})} & \tilde{z}_{(1)} & \cdots & \tilde{z}_{(L)} \\ | & | & & | & | & & | \end{bmatrix} \quad \text{with} \quad \tilde{z}_{(j)} = \begin{bmatrix} \tilde{z}_j^{(1)} \\ \vdots \\ \tilde{z}_j^{(n)} \end{bmatrix}.$$

Here $\tilde{z}_{(j)}$ for $j = 1, \ldots, L$ is the vector of indicator variables that marks which observations are at level j for the categorical feature. In statistics, the new L columns $\tilde{z}_{(j)}$ are called *indicator* or *dummy* variables. However, in statistics the practice is to include only $L - 1$ dummy variables instead of L. One reason for this is that when using L dummy variables the design matrix X will never be a full column rank matrix since the sum of the L dummy columns is equal to the first column of 1s. Thus, traditional statistical practice only includes $L - 1$ dummy variables in the model and the remainder is considered as the *reference level*. An alternative is to remove the bias term from the model (meaning drop the first column of X) and then keep all L dummy variables.

Multi-class Classfication

Now that we understand the linear model as well as ways of dealing with categorical variables, let us consider an application of the linear model for multi-class classification. We note that linear models are generally far from the state of the art when it comes to their application for classification problems, yet seeing the linear model applied to classification is instructive.

In classification problems each of the labels y takes on one of a finite number of values. The number of possible values is denoted via K. When $K = 2$ it is a binary classification problem but generally for $K > 2$ it is a *multi-class classification problem*. Notationally it is convenient to denote the set of label indices as $\{1, \ldots, K\}$ and consider some bijection between these indices and the actual label values e.g. `banana`, `dog`, etc. As a concrete example we consider the MNIST digits dataset where the label values are the digits `0`,`1`,...,`9` and notationally we use the label indices $\{1, \ldots, K = 10\}$. Here the obvious bijection shifts by 1.

A general scheme for multi-class classification is introduced in Section 3.3 in the context of multinomial (softmax) regression, and is further employed with other deep learning models in the chapters that follow. An alternative, which we introduce now in the context of linear models, is based on the fusion of multiple binary classification models into a multi-class classifier. For this we introduce two general methods, namely *one vs. rest* (also known as *one vs. all*) and *one vs. one*.

Both of these methods assume we have trained binary classifiers for sub-problems. With the one vs. rest strategy, we assume the availability of models for binary classification $f_{\theta_i}(\cdot)$ for $i = 1, \ldots, K$ where the ith model can discriminate between the label index i treated as `positive` and otherwise if the label index is not i then `negative`. With the one vs. one strategy we have $K(K-1)$ binary classifiers[10] denoted $f_{\theta_{i,j}}(x)$ for all $i, j = 1, \ldots, K$ such that $i \neq j$. Here the $(i, j)th$ classifier discriminates between the label being of index i (`positive`) or index j (`negative`).

The output range obtained by $f_{\theta_i}(\cdot)$ or $f_{\theta_{i,j}}(\cdot)$ is generally a value on the real number line $\mathbb{R}$. Positive outputs indicate `positive` while negative outputs indicate `negative`. The farther the model output is from 0 the stronger the confidence of the classification decision. A cutoff in similar nature to (2.5) is to apply the sign$(\cdot)$ function[11] to the model output and conclude either `positive` in case of $+1$ or `negative` in case of -1.

[10] In practice only half of this number of classifiers is needed because the classifier for (i, j) can be reverted to the classifier (j, i).

[11] In case the classifiers were trained with the 1, 0 encoding as in the case of logistic regression it is easy to transform them.

Now the one vs. rest or one vs. one strategies carry out prediction via,

$$\widehat{\mathcal{Y}} = \begin{cases} \text{argmax}_{i=1,\ldots,K} \, f_{\theta_i}(x) & \text{in case of one vs. rest,} \\ \text{argmax}_{i=1,\ldots,K} \sum_{j \neq i} \text{sign}\big(f_{\theta_{i,j}}(x)\big) & \text{in case of one vs. one.} \end{cases}$$

The idea of the one vs. rest strategy is to pick the label index which is most probable among the K classification models where each model focuses on a different label index. The idea of the one vs. one classifier is to pick the label i that when compared to the other $K-1$ labels, was chosen most often. This is achieved via comparison of a summation of $\text{sign}\big(f_{\theta_{i,j}}(x)\big)$ for all other labels j. In both cases, one needs to supply rules for handling ties in the argmax in the final decision, yet these details are generally insignificant.

We proceed with an example of using both strategies for the MNIST dataset using a linear model. The crux in creating the supporting binary classifiers is to set the label vector y used in (2.16) to have values of $+1$ for samples that are `positive` and values of -1 for samples that are `negative`.

For example, when learning the $f_{\theta_3}(\cdot)$ classifier we consider all digit images in the original dataset with the label value `2` as having[12] $y = +1$ and otherwise -1. Out of the $60,000$ MNIST training samples there are $5,958$ training samples that satisfy $y = +1$ and then for $y = -1$ there are $54,042$ samples. Obtaining the parameters for this classifier using (2.16) uses the design matrix X as in (2.10) which is of dimension $60,000 \times 785$ where $785 = 1 + 28 \times 28$. Each row of X corresponds to a different image and each of the columns 2 to 785 corresponds to a different pixel.

Similarly, when learning the $f_{\theta_{3,8}}(\cdot)$ classifier (this compares the digit `2` and the digit `7`) used in one vs. one, we set $+1$ for all $5,958$ training samples that have `2` and set -1 for all $6,265$ samples that have a label value of `7`. Here the design matrix X is of dimension $12,223 \times 785$ since $5,958 + 6,265 = 12,223$.

To obtain predictors $\hat{\theta}_i$ or $\hat{\theta}_{i,j}$ using (2.16) we compute the pseudo inverse of the respective design matrices using (for example) numerical procedures for (2.17). Note that the $\hat{\theta}_i$ classifiers require only a single pseudo inverse for all i while $\hat{\theta}_{i,j}$ has a different design matrix for each (i,j) pair and hence requires its own pseudo inverse.

[12] Here `2` is the label value that matches label index 3 when using label indexing $\{1,\ldots,K=10\}$.

Table 2.1: Confusion matrices for the MNIST digit test set using linear classifiers trained on the training set. (a) one vs. rest achieves an accuracy of 0.8603. (b) One. vs. one achieves an accuracy of 0.9297.

		Decision									
		0	1	2	3	4	5	6	7	8	9
Reality	0	944	0	18	4	0	23	18	5	14	15
	1	0	1107	54	17	22	18	10	40	46	11
	2	1	2	813	23	6	3	9	16	11	2
	3	2	2	26	880	1	72	0	6	30	17
	4	2	3	15	5	881	24	22	26	27	80
	5	7	1	0	17	5	659	17	0	40	1
	6	14	5	42	9	10	23	875	1	15	1
	7	2	1	22	21	2	14	0	884	12	77
	8	7	14	37	22	11	39	7	0	759	4
	9	1	0	5	12	44	17	0	50	20	801

(a)

		Decision									
		0	1	2	3	4	5	6	7	8	9
Reality	0	961	0	9	9	2	7	6	1	7	6
	1	0	1120	18	1	4	5	5	16	17	5
	2	1	3	936	18	6	3	12	17	8	1
	3	1	3	12	926	1	30	0	3	23	11
	4	0	1	10	2	931	8	5	11	10	30
	5	6	1	5	20	1	800	19	1	36	12
	6	8	4	10	1	7	17	908	0	10	0
	7	3	1	10	7	4	2	1	955	10	21
	8	0	2	22	21	3	15	2	1	840	3
	9	0	0	0	5	23	5	0	23	13	920

(b)

Now after training on the $60,000$ MNIST training samples and evaluating performance on the $10,000$ testing samples, we obtain an accuracy of 0.8603 using one vs. rest and an accuracy of 0.9297 using one vs. one. As MNIST is generally a balanced dataset, the use of accuracy to evaluate performance is sensible, and the level of accuracy obtained is impressive since the linear model is very simple and training these models using the pseudo-inverse computation only takes a few seconds at most. However, to get industrial grade performance one requires more advanced models such as the convolutional neural networks of Chapter 6. As mentioned in the previous chapter state of the art performance for MNIST is at an accuracy of over 99.8%.

When evaluating multi-class classifiers it is common to use a *confusion matrix* similar to the 2×2 confusion matrix presented in Section 2.2 for the case of binary classification. Table 2.1 presents the confusion matrices for both one vs. rest and one vs. one. It is insightful to pick out the entries where non-negligible misclassification occurs. For example with the one vs. rest classifier multiple real digits of 7 were classified as 9. This occurred 77 times. Similarly in the one vs. one case there were 36 misclassifications of the digit 5 as the digit 8.

2.4 Iterative Optimization Based Learning

Linear models coupled with quadratic loss (2.12) are gifted with a closed form solution for the parameter estimate as appearing in (2.16). This solution, which is based on the pseudo-inverse (2.17), is well studied in the field of numerical linear algebra and is typically suited for efficient numerical evaluation, a topic that we do not cover here any further. For example, in the section above, the pseudo inverse computation of the $60,000 \times 785$ design matrix X used for MNIST digit classification using the one vs. rest method can be evaluated in about a second on a modern laptop. Nevertheless, there are multiple scenarios where this closed form solution may not be used. This is the case when we use an alternative loss function to the quadratic loss, such as (2.18) or (2.19). Problems with using the explicit optimal quadratic solution may also arise when the number of features p is very large. In such cases and others, more generic methods for optimization are needed.

We now present a general iterative optimization method for obtaining this solution. It can be used in such scenarios where the pseudo-inverse based computation does not work, yet more importantly it serves to illustrate how gradient based optimization interplays with machine learning. Indeed the more complex deep learning models on which this book focuses use this type of method, and its generalizations, almost exclusively. We note that there are other methods used as well and Chapter 4 focuses entirely on optimization. Our purpose here is simply to introduce the essence of the most basic technique, namely *gradient descent*.

Algorithm 2.1: Gradient descent

Input: Dataset $\mathcal{D} = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$,
objective function $C(\cdot) = C(\cdot\,; \mathcal{D})$, and
initial parameter vector θ_{init}
Output: Approximately *optimal* θ

$\theta \leftarrow \theta_{\mathsf{init}}$
repeat
  Compute the gradient $\nabla C(\theta)$
  $\theta \leftarrow \theta - \alpha \nabla C(\theta)$
until θ satisfies a *termination condition*
return θ

This gradient descent procedure, presented in Algorithm 2.1, executes over iterations indexed by $t = 0, 1, 2, \ldots$ and works by taking small steps in the direction opposite to the gradient. That is, it traverses 'downhill' each time trying to descend in the steepest direction. In its simplest form, steps sizes are controled by a fixed $\alpha > 0$ called the *learning rate*. After some initialization with the vector $\theta^{(0)} = \theta_{\mathsf{init}}$, in each iteration t, the next vector $\theta^{(t+1)}$ is obtained via,

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \nabla C(\theta^{(t)}). \tag{2.20}$$

The algorithm repeats (2.20) where the key object that requires computation at each iteration t is the gradient of the loss function $\nabla C(\theta^{(t)})$. For complex deep learning models this computation is one of the core components of a deep learning framework and is carried out using the backpropagation algorithm studied in Chapter 5. However, for simpler models such as the linear model or logistic regression type models of Chapter 3, we have explicit expressions for the gradient.

In the case of the linear model with quadratic loss, we have already computed the gradient in (2.14) and we can represent it as,

$$\nabla C(\theta) = \frac{2}{n} X^\top (X\theta - y). \tag{2.21}$$

In general the algorithm iterates over t until $\theta^{(t+1)}$ and $\theta^{(t)}$ are close as measured by some stopping criteria such as,

$$\|\theta^{(t)} - \theta^{(t+1)}\| < \varepsilon, \tag{2.22}$$

with some fixed $\varepsilon > 0$. Other stopping criteria and variants of this method are studied in detail in Chapter 4. The final $\theta^{(t+1)}$ is used as $\hat{\theta}$ and if α and ε are well chosen the algorithm output may closely approximate the optimal θ.

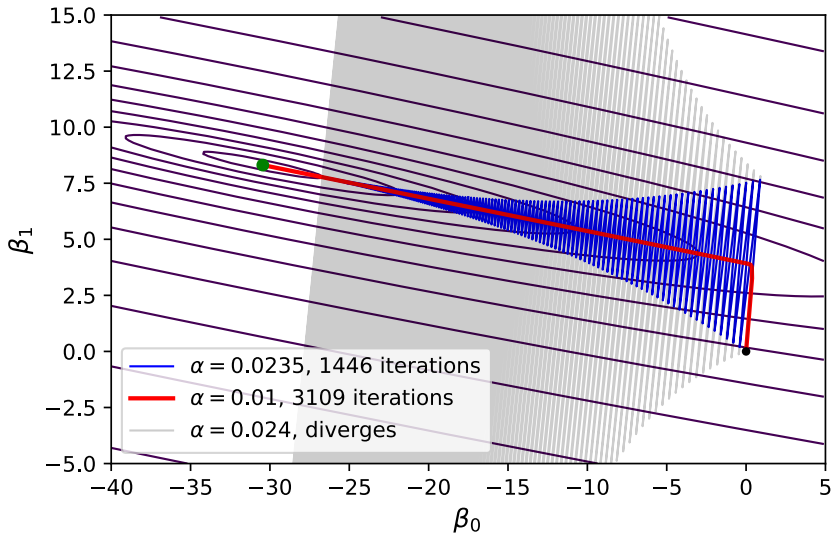


Figure 2.7: A contour plot of the loss function for a simple linear regression problem. The optimal point in green is reached when starting gradient descent at the origin (black point) with $\alpha = 0.01$ or $\alpha = 0.0235$. However with the learning rate slightly higher at $\alpha = 0.024$ gradient descent does not converge.

One of the main difficulties with the application of gradient descent is choosing the learning rate α. As an illustration, Figure 2.7 presents a contour plot of the squared error loss function associated with simple linear regression similar in nature to the Boston housing price data example of Figure 2.3 (a) where we optimize to find $\hat{\theta} = (\hat{\beta}_0, \hat{\beta}_1)$. When running with $\varepsilon = 10^{-5}$ and starting θ_{init} at the origin, we see that if $\alpha = 0.01$ the algorithm terminates near the optimal parameters in $3,109$ iterations. If $\alpha = 0.0235$ the algorithm terminates near the optimal parameters quicker with $1,446$ iterations yet follows a more jagged path. Finally, when running with the learning rate that is just slightly higher at $\alpha = 0.024$, the algorithm diverges and does not terminate at all. The plot in this divergent case is only for the first 300 iterations where the growing oscillations are still in the vicinity of the optimum. With further iterations the values quickly diverge.

This simple example illustrates that the value of the learning rate α is crucial. Different values imply drastically different behaviours. A more thorough investigation of gradient descent and its generalizations is in Chapter 4. Interestingly, for linear models more explicit results are available, as we present now.

Learning Rate Analysis for Linear Models

In general there is not a simple analytical way to determine a suitable learning rate α. Chapter 4 presents adaptive generalizations of gradient descent. Yet, universally there are no closed form recipes. Nevertheless, when it comes to the special case of the linear model, analysis of the dynamics of gradient descent is analytically attractive and we may explicitly describe the range of α for which convergence takes place. This description is not necessarily of direct practical use. However, it gives insight into the nature of gradient descent.

Consider the linear model with design matrix X as in (2.10) and denote $\lambda_{\max}$ as the maximal eigenvalue of the Gram matrix $X^\top X$. We can now show that gradient descent converges[13] to a solution of the normal equations (2.15) as long as

$$\alpha < \frac{n}{\lambda_{\max}}. \tag{2.23}$$

Note that for the data used in Figure 2.7, $\lambda_{\max} = 20,670.33$ and $n = 490$. Hence in this case the algorithm converges for α in the range $(0, 0.02371)$ and this bound is in agreement with the examples of Figure 2.7 where the first two paths have α in this range and the third path with $\alpha = 0.024$ diverges.

To see (2.23) use the gradient expression (2.21) and the gradient update rule of (2.20) to obtain the recursion,

$$\begin{aligned}\theta^{(t+1)} &= \theta^{(t)} - \alpha\frac{2}{n}X^\top(X\theta^{(t)} - y)\\ &= \underbrace{(I - \alpha\frac{2}{n}X^\top X)}_{A}\,\theta^{(t)} + \underbrace{\alpha\frac{2}{n}X^\top y}_{c},\end{aligned}$$

or in short $\theta^{(t+1)} = A\theta^{(t)} + c$. Such a recursion is known as an *affine discrete time linear dynamical system* and equilibrium points of such a system, denoted θ^* satisfy, $\theta^* = A\theta^* + c$ or $(I - A)\theta^* = c$. In our case, using A and c, it is evident that such equilibrium points are solutions of the normal equations (2.15). For simplicity let us assume here that X is full column rank and hence there is a unique θ^*.

It follows from linear systems theory that the spectral radius[14] of the matrix A determines the convergence or non-convergence of $\theta^{(t)}$ to θ^*. Specifically, if the spectral radius of the matrix A is less than unity, then $\theta^{(t)}$ converges to θ^* for any initial $\theta^{(0)}$ and, if the spectral radius is greater than unity, then for any initial $\theta^{(0)}$ the sequences diverges (the border case of the spectral radius being 1 is indeterminate). Hence, putting aside the border case of a spectral radius of 1, we see that convergence occurs if and only if the eigenvalues of A are in $(-1, 1)$.

[13] Here "convergence" formally means that for any $\varepsilon_2 > 0$ there is an $\varepsilon > 0$ of (2.22) where the algorithm terminates in a finite number of iterations with $\|\theta^* - \theta^{(t+1)}\| < \varepsilon_2$ and θ^* is a minimizer of the optimization problem.

[14] The spectral radius of a square matrix is the largest of all magnitudes of the eigenvalues.

Now, since since $X^\top X$ is a symmetric matrix, the eigenvalues of $X^\top X$ are real and since $X^\top X$ is positive semidefinite (this is a property of any Gram matrix), the eigenvalues lie in the range $(0, \lambda_{\max}]$ with $\lambda_{\max} > 0$. Further, the eigenvalues of $-2\alpha n^{-1} X^\top X$ lie in the range $[-2\alpha n^{-1}\lambda_{\max}, 0)$ and the eigenvalues of $A = I - 2\alpha n^{-1} X^\top X$ lie in the range $[1 - 2\alpha n^{-1}\lambda_{\max}, 1)$. Thus the critical inequality that ensures that the spectral radius of A is less than 1 is

$$-1 < 1 - 2\alpha \frac{1}{n}\lambda_{\max},$$

which is equivalent to (2.23).

The Loss Landscape and Standardization of Inputs

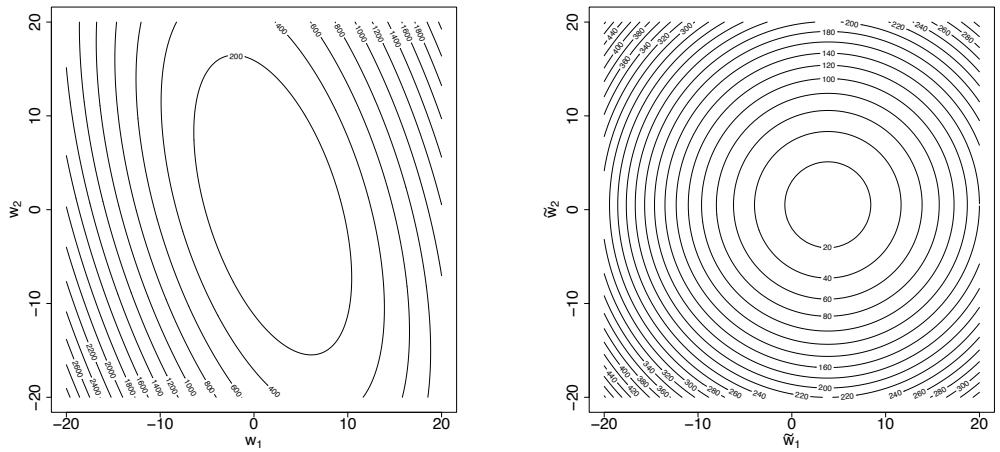
In Section 2.1 we presented standardization of inputs via (2.2) where the sample mean and sample variance are computed via (2.1). We now show that such standardization also affects the *loss landscape*, often yielding improvement in the execution of gradient descent. As above, our analysis is in the realm of linear models where explicit analysis is possible.

To illustrate this concept, we consider a simple case of a linear model with two input features x_1 and x_2 where,

$$y = w_1 x_1 + w_2 x_2 + \epsilon.$$

This is similar to the simple regression models of Section 2.2 yet is without an intercept term. Here, with n data samples, the $n \times 2$ design matrix X has columns composed of the vectors $(x_i^{(1)}, \ldots, x_i^{(n)})$ for $i = 1, 2$; the labels vector is $y = (y^{(1)}, \ldots, y^{(n)})$; and the model parameters are $\theta = (w_1, w_2)$. The square loss function where for simplicity we omit the $1/n$ term in (2.11) is

$$\begin{aligned} C(\theta\,;\, X, y) &:= \|y - X\theta\|^2 = (y - X\theta)^\top (y - X\theta) \\ &= \theta^\top X^\top X\theta - 2y^\top X\theta + y^\top y. \end{aligned}$$



(a) **(b)**

Figure 2.8: Contour levels of a loss function for an example with $p = 2$ parameters. (a) The loss as a function of (w_1, w_2) for the original data has accentuated elliptical contours. (b) The loss as a function of the parameters, $(\tilde{w}_1, \tilde{w}_2)$, associated with standardized data, yields much less accentuated contours.

A contour plot for some arbitrary (not standardized) data is illustrated in Figure 2.8 (a). It is evident that the contour levels of the loss function are ellipsoids. For some given loss level C, it can be shown that the lengths of the principal axes of the ellipse are $\sqrt{\frac{k}{\lambda_i}}$ for $i = 1, 2$, where λ_1 and λ_2 are the eigenvalues of the Gram matrix $X^\top X$ and $k = C - y^\top y$. With this, the elongation of the ellipse, which is the ratio of these lengths, is,

$$R_X = \frac{\sqrt{\frac{k}{\lambda_1}}}{\sqrt{\frac{k}{\lambda_2}}} = \sqrt{\frac{\lambda_2}{\lambda_1}}.$$

Now, when standardizing the inputs, each of the columns of X is transformed separately via (2.2). In such a case, the loss function associated with the standardized data is $C(\theta\,;\, Z, y)$ where we denote by Z the design matrix of the standardized data. It can now be shown that

$$Z^\top Z = n \begin{pmatrix} 1 & \rho \\ \rho & 1 \end{pmatrix}, \qquad \text{where} \qquad \rho = \frac{1}{n} \sum_{j=1}^{n} z_1^{(j)} z_2^{(j)},$$

is also known as the *sample correlation* between the two features. Note that $\rho \in [-1, 1]$.

With such a normalization and an explicit form for the Gram matrix $Z^\top Z$, we can compute the eigenvalues of $Z^\top Z$ to be $\lambda_1 = 1 + |\rho|$ and $\lambda_2 = 1 - |\rho|$. Thus, the elongation of the ellipsoid is

$$R_Z = \sqrt{\frac{\lambda_2}{\lambda_1}} = \sqrt{\frac{1 + |\rho|}{1 - |\rho|}}.$$

It is evident that in cases where the correlation between the features is low, then $R_Z \approx 1$ and this implies that the loss landscape of the standardized data is much more similar to Figure 2.8 (b).

Now, in terms of gradient descent, taking steps in a loss landscape such as Figure 2.8 (b) is generally more efficient than using the loss landscape in Figure 2.8 (a). Hence it is expected that as long as ρ is not close to 1 or -1, carrying out standardization will help gradient descent converge faster. Note that, while this analysis is carried out on a simplistic $p = 2$, $d = 2$ linear model with quadratic loss, the principle often applies to more complicated loss landscapes as well. Further note, that with standardization of the features or any other transformation one also has to encode the standardization transformation as part of the deployed model, since the model is now for the standardized features z instead of x.

2.5 Generalization, Regularization, and Validation

The data available while learning and calibrating a model is called the *seen data* and future data is called *unseen data*. These concepts were introduced in Section 2.1. Our purpose of fitting a model based on seen data is that it will ultimately work well for unseen data, a property known as *generalization ability*. With this view, when seeking models that generalize well, there are two competing negative attributes that one needs to balance, *underfitting* and *overfitting*. Underfitting is a case when the model is too simple and fails to capture the complexity of the underlying data. On the other hand, overfitting is a case where the model is so specialized to the training data such that unseen examples that slightly differ from the training data do not perform well. The theme of *model selection* in machine learning

and statistics deals with the calibration of underfitting and overfitting to yield models that generalize well.

Model selection, or the quest for optimal generalization ability, is one of the hardest problems in machine learning primarily because the unseen data is not available. For this, one needs to judiciously budget the seen data by splitting it into the training set, the test set, and also carry out validation in one of several ways that we outline in this section. In quantifying generalization ability there are several plots and measures that one can use. These include the quantification of *model bias*, *model variance*, and the *bias and variance tradeoff*. We present these in this section.

Some classes of models are by construction designed to enable calibration of underfitting, overfitting, or the bias–variance tradeoff. One general technique for this is called *regularization* which in one common form, includes the introduction of additional terms to the model's loss function. We present a taste of regularization techniques here and then in Section 5.7 we focus on regularization in the context of deep neural network models.

In terms of notation, throughout this book we use $\mathcal{D}$ to denote data with n samples. This sometimes means only the training set and in other cases means all of the seen data. When we focus on training specific types of models such as in Section 2.3, the symbol $\mathcal{D}$ is treated as the data allocated specifically for training and hence we assume there are n training samples for training and potentially other samples for testing and/or validation that we do not account for. In other cases $\mathcal{D}$ is treated as all of the available seen data, part of which may be used for testing via a testing or hold-out set which we denote via $\mathcal{D}_{\text{test}}$ with n_{test} samples.

Performance on Unseen Data

We have already introduced several examples of *performance metrics* in Section 2.2. These include accuracy (2.6), the F_1 score (2.7), mean square error in the case of regression, and others. In some cases one wishes to maximize the performance metric where as in other cases one wishes to minimize it. Note that the loss function used in model training is in some instances directly related or equal to the performance metric, and, in other instances, it is different.

It is notationally convenient to relate a *performance function* to the performance metric. We denote the performance function via $\mathcal{P}(\cdot, \cdot)$ and it penalizes differences between a single predicted label $\hat{y}$ and the actual label y. For example when the mean square error performance metric is used then the performance function is $\mathcal{P}(\hat{y}, y) = (\hat{y} - y)^2$. As another example, if the accuracy performance metric is used in classification then the performance function is $\mathcal{P}(\widehat{\mathcal{Y}}, y) = \mathbf{1}\{\widehat{\mathcal{Y}} \neq y\}$, where $\mathbf{1}\{\cdot\}$ is the indicator function and y is taken as the actual label. Note that we construct the performance function such that small values are desirable.[15]

When we train a model and create a predictor either for regression or classification, we use the data $\mathcal{D}$ and based on the model obtain a predictor denoted by $\widehat{y}(\cdot\,; \mathcal{D})$. Now, for some data pair (x, y), the value $\widehat{y}(x\,; \mathcal{D})$ is the prediction of y and the performance function evaluated for the prediction of this data pair is $\mathcal{P}\big(\widehat{y}(x\,; \mathcal{D}), y\big)$.

[15] In this section, to avoid notational confusion between $\hat{y}$ and $\widehat{\mathcal{Y}}$, we use the notation $\hat{y}$ for both cases.

As outlined in Section 2.1, we ensure that the nature of the seen data is similar to that of unseen data and with this, the underlying modelling assumption is that both seen and unseen data are generated by the same underlying processes. Hence, for both theoretical and empirical analysis, unless we know otherwise, we assume that the probability distribution of each data sample $(x^{(i)}, y^{(i)})$ is the same for all $i = 1, \ldots, n$ and is further the same as the distribution of each unseen data sample $(x^\star, y^\star)$. That is, we assume there is an underlying probability space for the observations and we vaguely denote the joint probabilities of the features and label via $\mathbb{P}(x, y)$. Our usage of probabilistic statements here is only via expected values where we denote the expectation operator via $\mathbb{E}[\cdot]$ and often use a subscript for the expectation to denote the objects that are treated as random.

With this notation, the expected value of the performance of the trained model for unseen data points $(x^\star, y^\star)$ is denoted via,

$$E_{\text{unseen}} = \mathbb{E}_{(x^\star, y^\star)}\big[\mathcal{P}\big(\widehat{y}(x^\star; \mathcal{D}_{\text{train}}), y^\star\big)\big], \tag{2.24}$$

where $\mathcal{D}_{\text{train}}$ is the training data. This quantity is called the *generalization performance* or *generalization error*. It may be viewed as an average over all possible unseen data points and hence E_{unseen} evaluates how well the predictor or model generalizes. With a given training dataset $\mathcal{D}_{\text{train}}$, our aim is to build a model that yields the smallest possible E_{unseen}.

Unfortunately, since it is based on unseen data, E_{unseen} is a theoretical construct and since we do not know the probability law $\mathbb{P}(x, y)$ exactly, we cannot compute E_{unseen}. However, as a first attempt, we can approximate the expectation by averaging over available training data. That is,

$$E_{\text{train}} = \frac{1}{n_{\text{train}}} \sum_{(x,y)\in\mathcal{D}_{\text{train}}} \mathcal{P}\big(\widehat{y}(x\,;\,\mathcal{D}_{\text{train}}),\, y\big), \tag{2.25}$$

where n_{train} is the number of observations in $\mathcal{D}_{\text{train}}$. It turns out that E_{train} is typically a poor estimator of E_{unseen} because the same training observations that were used to create the predictor are also used to evaluate the predictor performance. That is, the learned parameters of the model $\hat{\theta}$ that are used to construct $\hat{y}(\cdot)$ depend on $\mathcal{D}_{\text{train}}$. Hence while E_{train} does present us with some insight about the ability of our model to reproduce the data that has been learned, it lacks the ability to estimate performance on unseen data.

In order to get a better estimate of E_{unseen}, it is preferable to average over data that has not been used for training the model, namely over the test set. In an ideal situation where we use the test set only once and do not calibrate and adjust the model based on the test set, the test set observations are completely independent of the model. In such a case, the estimator,

$$E_{\text{test}} = \frac{1}{n_{\text{test}}} \sum_{(x,y)\in\mathcal{D}_{\text{test}}} \mathcal{P}\big(\widehat{y}(x; \mathcal{D}_{\text{train}}), y\big), \tag{2.26}$$

is a good estimator of E_{unseen}, especially for significantly large n_{test}. Specifically under the assumption that the unseen data and the test set have the same distribution, the expected value of E_{test} is exactly E_{unseen} making it a statistically *unbiased estimator* of performance. Further it is statistically *consistent* in the sense that if we are able to allocate more testing data and $n_{\text{test}} \to \infty$ then $E_{\text{test}} \to E_{\text{unseen}}$. This is simply a consequence of the law of large numbers.[16] Note that these desirable statistical properties are only for a fixed $\mathcal{D}_{\text{train}}$.

[16] Formally the convergence $E_{\text{test}} \to E_{\text{unseen}}$ may be seen as convergence in probability in one form or almost sure convergence in a different form. We do not focus on such subtleties here.

The straightforward statistical properties of unbiasedness and consistency enjoyed by E_{test} make the practice of holding out a test set for performance evaluation attractive. However, setting aside a test set is costly as we effectively 'throw away' n_{test} observations and do not use them for improving the model. For this reason, it is often tempting in practice to iteratively evaluate (2.26) while adjusting model settings or hyper-parameters. This frowned upon practice breaks the independence between $\mathcal{D}_{\text{test}}$ and the model at which point the desirable statistical properties of E_{test} are lost. Hence, as an alternative, we use a validation set or some other method as described below.

In addition to using an independent test set as in (2.26), other alternatives for estimation of performance also exist, which include using K-fold cross validation. This is a topic we describe below in the context of validation and hyper-parameter optimization, yet it may also be used for purposes of performance evaluation.

Model Choice, Underfitting, and Overfitting

The generalization performance in (2.24) is specific to a fixed single training dataset $\mathcal{D}_{\text{train}}$. However, for a given problem, when considering which type of model to use and what hyper-parameters to choose, it is often useful to think about the expectation over all possible training datasets. For this we define the *expected generalization performance*,

$$\widetilde{E}_{\text{unseen}} = \mathbb{E}_{\mathcal{D}_{\text{train}}}[E_{\text{unseen}}] = \mathbb{E}_{\mathcal{D}_{\text{train}}}\big[\mathbb{E}_{(x^\star,y^\star)}[\mathcal{P}(\widehat{y}(x^\star;\mathcal{D}_{\text{train}}),y^\star)]\big]. \tag{2.27}$$

It represents the average of E_{unseen} over all possible datasets $\mathcal{D}_{\text{train}}$ of a given size from the same probability law $\mathbb{P}(x,y)$ where we keep in mind that each dataset potentially yields a different representation of the model.

A similar quantity for the training set is

$$\widetilde{E}_{\text{train}} = \mathbb{E}_{\mathcal{D}_{\text{train}}}[E_{\text{train}}] = \frac{1}{n_{\text{train}}}\mathbb{E}_{\mathcal{D}_{\text{train}}}\Big[\sum_{(x,y)\in\mathcal{D}_{\text{train}}} \mathcal{P}(\widehat{y}(x\,;\mathcal{D}_{\text{train}}),y)\Big]. \tag{2.28}$$

Keep in mind that $\widetilde{E}_{\text{unseen}}$ and $\widetilde{E}_{\text{train}}$ are functions of the type of model used, the hyper-parameters, and the training dataset size. With such relationships present, the machine learning engineer can in principle ponder about the theoretical shape of $\widetilde{E}_{\text{unseen}}$ and $\widetilde{E}_{\text{train}}$ and seek a model that appears best. In this respect the *generalization gap* defined as $\widetilde{\Delta} = \widetilde{E}_{\text{unseen}} - \widetilde{E}_{\text{train}}$ is also important.

The combination of $\widetilde{E}_{\text{unseen}}$, $\widetilde{E}_{\text{train}}$, and the generalization gap $\widetilde{\Delta}$ based on estimates, allows one to seek a balance between underfitting and overfitting. There are multiple suggestions on "best practice" for using the available data to estimate $\widetilde{E}_{\text{unseen}}$, $\widetilde{E}_{\text{train}}$, $\widetilde{\Delta}$, and to select the best model. A thorough discussion of such best practices is beyond our scope. The notes and references at the end of this chapter link to further reading. Instead, let us consider the schematic Figure 2.9, which presents typical behavior of $\widetilde{E}_{\text{unseen}}$ and $\widetilde{E}_{\text{train}}$ as a function of model complexity.

Generally, as model complexity increases, expected training performance, $\widetilde{E}_{\text{train}}$, improves (decreases) since complex structured models can explain the training data better. At high extremes this is overfitting. Similarly an opposite phenomenon is that models with low

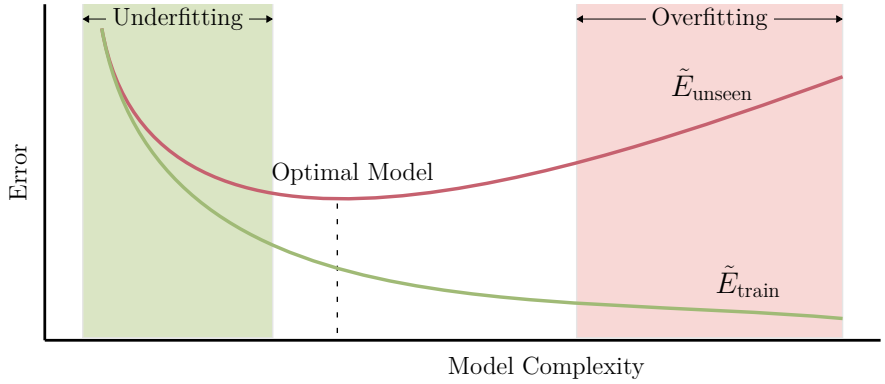


Figure 2.9: Typical behaviour of expected generalization performance and expected training performance as a function of model complexity

complexity are not able to describe the data well. The tradeoff between these two regimes is obtained at the minimum of $\widetilde{E}_{\text{unseen}}$ marked by the vertical dashed line.

In practice, unless presented with an infinite pool of data, one is not able to evaluate $\widetilde{E}_{\text{unseen}}$ and $\widetilde{E}_{\text{train}}$ directly and one is certainly not able to evaluate these quantities over all possibilities of models, hyper-parameters, and sample sizes. Nevertheless, much of the practice of model selection revolves around getting a feel for the dependence of $\widetilde{E}_{\text{unseen}}$ and $\widetilde{E}_{\text{train}}$ on model choice, hyper-parameters, and sample size. This is typically done using very limited measurements from one or several training and validation executions.

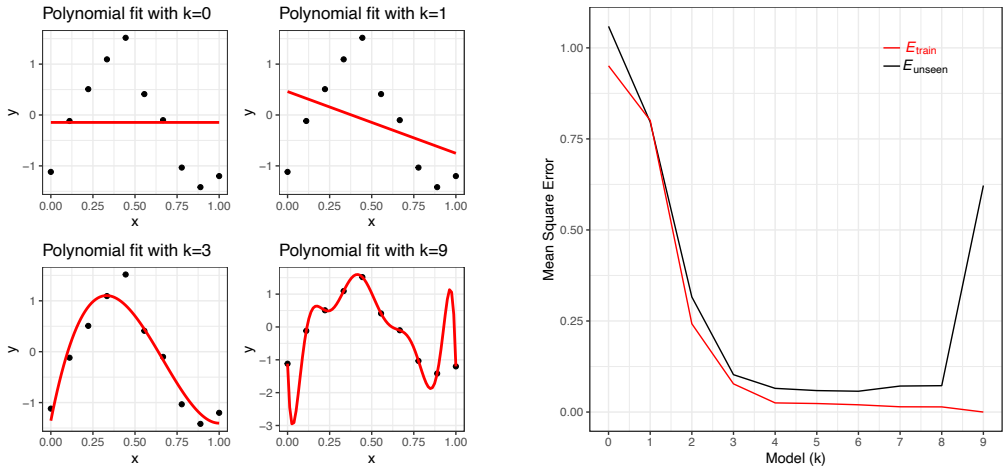
Typical practice is to monitor empirical estimates of these quantities as a function of model complexity, hyper-parameter choice, or sample size. The most basic practice is evaluation of E_{train} of (2.25) together with a validation performance measurement that is of similar nature to E_{test} of (2.26) such as the validation performance or K-fold cross validation performance which are defined in the sequel. See (2.37) and (2.38) below.

As one simple illustrative example capturing the tradeoffs of model complexity, let us consider linear models with polynomial features applied to synthetic univariate ($p = 1$) data. The model

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \ldots + \beta_k x^k + \epsilon \tag{2.29}$$

is denoted by $\mathcal{M}_k$ where k is the order of the polynomial. Hence $\mathcal{M}_0$ is the constant model, $\mathcal{M}_1$ is the simple linear model, $\mathcal{M}_2$ is the quadratic model, and so on. A quadratic model of this nature was used in (2.4) of Section 2.2. In this framework, model complexity corresponds to the degree of the polynomial model.

Now taking one possible realization of $\mathcal{D}_{\text{train}}$, in Figure 2.10 (a) we use this family of models to fit data of size $n_{\text{train}} = 10$. With this single realization we clearly see underfitting behaviour for models $\mathcal{M}_0$ and $\mathcal{M}_1$. In contrast, model $\mathcal{M}_9$ appears to overfit the observed data. Between these two extremes, model $\mathcal{M}_3$ looks like an appropriate representation of the observed data.



(a) **(b)**

Figure 2.10: Increasing model complexity illustrated via linear models with polynomial features where k, the order of the polynomial, captures the complexity. (a) Fitting several models to a single realization with $n = 10$ data-points. (b) The training performance in red and simulation estimates of the generalization performance in black.

In Figure 2.10 (b) the red curve presents E_{train} for this dataset. It is obvious that as k increases training fit improves. Further, in this hypothetical example since we know the underlying process with probability law $\mathbb{P}(x, y)$ used for purposes of simulation of synthetic data, we may sample as many $(x^\star, y^\star)$ pairs as we wish, to obtain a reliable estimate of E_{unseen}. This curve is plotted in black where in this case we use $10,000$ repetitions for each k, each time with the fixed model based on our single available dataset, $\mathcal{D}_{\text{train}}$. This Monte Carlo simulation, makes it clear that when $k = 9$ or $k = 8$ there is overfitting and when $k = 0, 1, 2$ there is underfitting. In practice plots exactly like Figure 2.10 (b) cannot be produced because we do not know $\mathbb{P}(x, y)$. Instead one can resort to estimates based on cross validation to obtain curves similar to the black curve in Figure 2.10 (b).

We also mention that, while we stated that key elements that affect expected performance are the model type, hyper-parameters, and sample size, in the world of deep learning there is also an additional major factor, *training time*. For deep learning models, since the number of parameters in the model is often huge, letting the model train for longer is similar to using a more complex model as presented in Figure 2.9.

Bias and Variance Decomposition

A related view to the analysis of expected generalization performance and the generalization gap is the so-called bias and variance decomposition. It focuses on the expected generalization performance in production, $\widetilde{E}_{\text{unseen}}$, and decomposes it into a sum of terms related to model bias, model variance, and the noise magnitude. With this decomposition, underfitting is said to be a situation with high model bias and overfitting is said to be a situation with high model variance. Using this terminology, balancing model bias and model variance is equivalent to balancing underfitting and overfitting respectively. This is known as the *bias and variance tradeoff*.

The bias and variance decomposition is mathematically elegant in the special case of the square error performance function $\mathcal{P}(\hat{y}, y) = (\hat{y} - y)^2$ and a specifically assumed underlying random reality

$$y = f(x) + \epsilon, \quad \text{with} \quad \mathbb{E}[\epsilon] = 0, \quad \text{and } \epsilon \text{ is independent of } x. \tag{2.30}$$

Here x a vector of features and y a scalar real valued label. Further $\mathbb{E}[\epsilon^2]$ is the variance of the noise term and is called the *inherent noise*. In this setting, for some unseen feature vector $x^\star$, the predictor trained on data $\mathcal{D}$ is $\widehat{y}(x^\star\,;\mathcal{D})$, which we also denote via $\hat{f}(x^\star\,;\mathcal{D})$ since it estimates $f(x^\star)$. Hence the expected generalization performance of (2.27) becomes

$$\widetilde{E}_{\text{unseen}} = \mathbb{E}_{\mathcal{D},x^\star,\epsilon}\Big[\big(\hat{f}(x^\star;\mathcal{D}) - (f(x^\star) + \epsilon)\big)^2\Big]. \tag{2.31}$$

Now, a standard algebraic manipulation common in statistics is to add and subtract $\mathbb{E}_{\mathcal{D},x^\star}[\hat{f}(x^\star;\mathcal{D})]$ inside (2.31), expand the expression, apply the external expectation operator, and then cancel out terms that have zero expectation (resulting from $\mathbb{E}[\epsilon] = 0$ and the fact that ϵ and $x^\star$ are independent). This manipulation transforms (2.31) to the *bias-variance-noise decomposition equation*,

$$\widetilde{E}_{\text{unseen}} = \underbrace{\Big(\mathbb{E}[\hat{f}(x^\star\,;\mathcal{D})] - \mathbb{E}[f(x^\star)]\Big)^2}_{\text{Bias squared of } \hat{f}(\cdot)} + \underbrace{\mathbb{V}\text{ar}\left(\hat{f}(x^\star\,;\mathcal{D})\right)}_{\text{Variance of } \hat{f}(\cdot)} + \underbrace{\mathbb{E}[\epsilon^2]}_{\text{Inherent Noise}}. \tag{2.32}$$

Here the first term is the square of the bias, the second term is the variance taking into consideration variability both from $\mathcal{D}$ used for training and $x^\star$, and the third term is the inherent noise. The expectations and variances in the bias and variance terms are with respect to the training dataset $\mathcal{D}$ and the arbitrary unseen feature vector $x^\star$. The main takeaway from (2.32) is that if we ignore the inherent noise, the loss of the model has two key components, *model bias* (technically it is the model bias squared), and *model variance*.

The model bias is a measure of how a typical (expectation over all possible data samples) model $\hat{f}(\cdot\,;\mathcal{D})$ misspecifies the correct relationship $f(\cdot)$. Model classes with high bias, have that $\hat{f}(\cdot\,;\mathcal{D})$ does not accurately predict $f(\cdot)$. That is, high bias generally implies *underfitting*. Similarly, model classes with low model bias are detailed descriptions of reality since the expected difference in the bias term is near zero.

The model variance is a measure of the variability of the model class $\hat{f}(\cdot\,;\mathcal{D})$ with respect to the random sample $\mathcal{D}$ and the distribution of $x^\star$ as implicitly implied by the probability law of the data $\mathbb{P}(x, y)$. Model classes with high model variance are often *overfit* (to the training data) and do not *generalize* (to unseen data) well. Similarly, model classes with low model variance are much more robust to the training data and generalize to the unseen data much better.

Similar analysis to the derivation that leads to (2.32) can also be attempted for other performance functions other than square error, and model structures other than (2.30). With such other settings, the mathematical elegance of (2.32) is often lost. Nevertheless, the concepts of model bias, model variance, and the bias and variance tradeoff still persist. For example in a classification setting we may compare the accuracy obtained on the training set to that obtained on a validation set. If there is a high discrepancy where the training accuracy is much higher than the validation accuracy, then there is probably a variance problem indicating that the model is overfitting.

Addition of Regularization Terms

One natural way to control model variance is to induce or force model parameters to remain within some confined subset of the parameter space. This is called *regularization*. At the extreme case where all model parameters are 0, the model variance vanishes as well. In less extreme cases where there is only some constraint on model parameters, model variance is still controlled. Such decreases in model variance may imply an increase of model bias. Nevertheless, the ultimate goal of optimizing the expected performance loss typically merits such adjustments.

A common way to keep model parameters at bay is to augment the optimization objective $\min_\theta C(\theta\,;\mathcal{D})$ with an additional *regularization term* $R_\lambda(\theta)$. The revised objective is then,

$$\min_\theta \; C(\theta\,;\mathcal{D}) + R_\lambda(\theta). \tag{2.33}$$

The regularization term $R_\lambda(\theta)$ depends on a *regularization parameter* λ, which is often a scalar in the range $[0,\infty)$ but also sometimes a vector. This *hyper-parameter* allows us to optimize the bias and variance tradeoff.

A common general regularization technique called *elastic net* has regularization parameter $\lambda = (\lambda_1, \lambda_2)$ and,

$$R_\lambda(\theta) = \lambda_1\|\theta\|_1 + \lambda_2\|\theta\|^2 \quad \text{with} \quad \|\theta\|_1 = \sum_{i=1}^{d}|\theta_i| \;\text{ and }\; \|\theta\|^2 = \sum_{i=1}^{d}\theta_i^2, \tag{2.34}$$

where d is the dimension of the parameter space.[17] Hence the values of λ_1 and λ_2 determine what kind of penalty the objective function will pay for high values of θ_i.

Clearly, with $\lambda_1, \lambda_2 = 0$ the original objective is unmodified. In contrast, as $\lambda_1 \to \infty$ or $\lambda_2 \to \infty$ the estimates $\theta_i \to 0$ and any information in the data $\mathcal{D}$ is fully ignored. Indeed, as λ_1 or λ_2 grow, the model bias grows while model variance is decreased and overfitting is mitigated. With regularization there is often a magical 'sweet spot' for λ where the objective (2.33) does a good job at fitting the model.

Particular cases of elastic net are the classic *ridge regression*, also called *Tikhonov regularization*, and *LASSO* standing for *least absolute shrinkage and selection operator*. In the former $\lambda_1 = 0$ and only λ_2 is used, and in the latter $\lambda_2 = 0$ and only λ_1 is used. One of the benefits of LASSO, also present in the more general elastic net case, is that the $\|\theta\|_1$ loss allows the algorithm to remove variables from the model by "zeroing out" their θ_i values completely. Hence LASSO is very useful as a model selection technique.

The case of ridge regression is slightly simpler to analyze than LASSO and it fits well within the framework of linear models presented in Section 2.3. We thus present the details now. For ridge regression the data fitting problem can be represented as,

$$\min_{\theta\in\mathbb{R}^d} \|y - X\theta\|^2 + \lambda\|\theta\|^2, \tag{2.35}$$

[17] Note that in cases such as linear regression or deep neural networks where there is a constant term (β_0 for example), the parameters for the constant term are typically not regularized and hence the norms are taken only on the other parameters.

where the design matrix X is as in (2.10) and we now consider λ as a scalar (previously in (2.34) it was denoted as λ_2) in the range $[0, \infty)$. Compare (2.35) with the original least squares objective (2.13). Now by manipulating the $\|\cdot\|^2$ expressions, the problem can be recast as[18]

$$\min_{\theta \in \mathbb{R}^d} \left\| \begin{bmatrix} y \\ 0 \end{bmatrix} - \tilde{X}_\lambda \, \theta \right\|^2 \quad \text{with} \quad \tilde{X}_\lambda = \begin{bmatrix} X \\ \sqrt{\lambda} I \end{bmatrix},$$

where I is the $d \times d$ identity matrix and 0 is the zero vector in $\mathbb{R}^d$. The pseudo-inverse associated with $\tilde{X}_\lambda$ is $\tilde{X}_\lambda^\dagger = (X^\top X + \lambda I)^{-1} \begin{bmatrix} X^\top & \lambda I \end{bmatrix}$. Hence, returning to (2.16), the parameter estimate for ridge regression is

$$\hat{\theta} = (X^\top X + \lambda I)^{-1} X^\top y. \tag{2.36}$$

As an aside, note that for any $\lambda > 0$ the matrix $X^\top X + \lambda I$ is not singular even if $X^\top X$ is singular. Also as $\lambda \to 0$ it can be shown that the pseudo-inverse $\tilde{X}_\lambda^\dagger$ converges to the SVD based pseudo-inverse (2.17) associated with X.

We note that while for linear models, the results are very elegant, in other cases closed form solutions such (2.36) do not exist. Still, many machine learning loss functions can be augmented with a regularization term. We revisit these concepts in Section 5.7 in the context of deep learning where other regularization methods are presented as well. Also note that the regularization parameter λ is a first class example of a hyper-parameter that one would like to calibrate during learning. This specific parameter serves as a good lever for optimizing the bias and variance tradeoff. We now discuss the general topic of hyper-parameter optimization.

Hyper-parameter Calibration and Cross Validation

As alluded to above, calibrating the model choice and the hyper-parameters while reusing the test set for performance evaluation is bad practice since it pollutes the test set performance estimator E_{test} of (2.26). For this reason it is common to further split the training data $\mathcal{D}_{\text{train}}$ using one of several ways while experimenting with model configurations and hyper-parameters. With such an approach $\mathcal{D}_{\text{test}}$ is reserved only for final performance evaluation before rolling out the model to production. Such use of the training data where some parts of the data are used for training parameters and the other parts are used for checking performance and tuning hyper-parameters is generally called *cross validation*.

There are multiple common cross validation techniques with many variants used in practice. Here we present only two main approaches, the *train-validate split* approach and *K-fold cross validation*. The train-validate split approach is common in situations where the total number of datapoints n is large. The K-fold cross validation approach is useful when data is limited.

The train-validate split approach simply implies that the original data with n samples is first split to training and testing as before and then the training data is further split into two subsets where the first is (confusingly) again called the *training set* and the latter is the *validation set*. Hence considering all of the available data $\mathcal{D}$, with this approach,

$$\mathcal{D} = \mathcal{D}_{\text{train}} \cup \mathcal{D}_{\text{validate}} \cup \mathcal{D}_{\text{test}}, \quad \text{where the unions are of disjoint sets.}$$

[18] In practice we often do not regularize the intercept term and this requires adjusting the identity matrix in $\tilde{X}_\lambda$.

When considering all of the data, this approach is also called the *train-validate-test split* approach. If, for example, we use a 80-20 rule for both splits and assuming divisibility holds, then $n_{\text{test}} = 0.2 \times n$, $n_{\text{train}} = 0.64 \times n$ and $n_{\text{validation}} = 0.16 \times n$.

As an example, assume the model is fixed yet regularized with elastic net as presented above. Hence the hyper-parameters in question are $\lambda = (\lambda_1, \lambda_2)$ and the choice of these needs to be tuned. The approach is then to evaluate the estimator on $\mathcal{D}_{\text{train}}$ over a grid of such hyper-parameters, retraining from scratch for every λ. We then choose $\lambda^* = \text{argmax}_\lambda \; E_{\text{validation}}(\lambda)$ with

$$E_{\text{validation}}(\lambda) = \frac{1}{n_{\text{validation}}} \sum_{(x,y)\in\mathcal{D}_{\text{validation}}} \mathcal{P}\big(\widehat{y}(x\,;\mathcal{D}_{\text{train}},\lambda), y\big), \tag{2.37}$$

where we can see that the predictor $\hat{y}$ depends on the hyper-parameter. With the optimal λ^* pair selected, the model with this λ^* is evaluated on the test set once via (2.26) before being rolled out to production. Note that this approach has many variants used in practice.

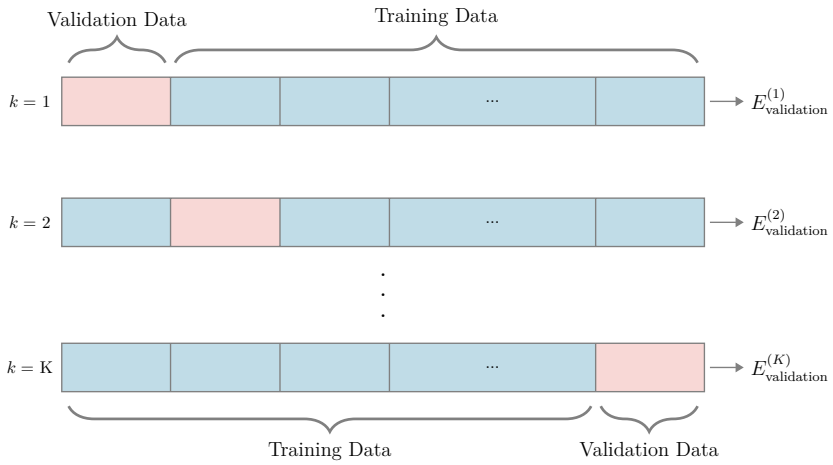


Figure 2.11: K-fold cross validation. For each $k = 1, \ldots, K$ the data is split into training data and validation data differently. This yields K estimates for performance and these estimates can be averaged.

In case of limited observations a train-validate-test split may be too wasteful of data and an alternative approach is K-fold cross validation as illustrated in Figure 2.11. This approach may be used on all of the data $\mathcal{D}$ or only on the training data after a train-test split is performed. Here for simplicity we apply it to some dataset $\mathcal{D}$. The approach is useful both for model selection, hyper-parameter optimization, and performance evaluation.

The value K of this approach which determines the number of data chunks or repetitions is a static configuration parameter with a typical value being $K = 5$ or $K = 10$. The approach is to split $\mathcal{D}$ into K equally sized data chunks each denoted $\mathcal{D}^k$ with,

$$\mathcal{D} = \mathcal{D}^1 \cup \mathcal{D}^2 \cup \ldots \cup \mathcal{D}^K, \quad \text{where again the unions are of disjoint sets.}$$

Then for each $k = 1, \ldots, K$ we fix a training set to be composed of all of the observations except for $\mathcal{D}^k$ and the validation set (may also be called a test set) to be $\mathcal{D}^k$. That is,

denoting set difference with '\' we set,

$$\mathcal{D}^{(k)}_{\text{train}} = \mathcal{D} \setminus \mathcal{D}^k, \quad \text{and} \quad \mathcal{D}^{(k)}_{\text{validation}} = \mathcal{D}^k. \quad \text{for} \quad k = 1, \ldots, K.$$

We may now retrain and evaluate the model separately for each data chunk k where each time we use $\mathcal{D}^{(k)}_{\text{train}}$ as the training data and $\mathcal{D}^{(k)}_{\text{validation}}$ as the validation (or testing) data. That is if for example $K = 10$ and originally $\mathcal{D}$ has n observations then for each k we have $n_{\text{train}} = 0.9 \times n$ and $n_{\text{validation}} = 0.1 \times n$ (again assuming n is properly divisible).

With the model trained separately for each repetition k, we can now estimate performance via,

$$E_{\text{cv}} = \frac{1}{K} \sum_{k=1}^{K} E^{(k)}_{\text{validation}}, \tag{2.38}$$

with,

$$E^{(k)}_{\text{validation}} = \frac{1}{n_{\text{validation}}} \sum_{(x,y) \in \mathcal{D}^{(k)}_{\text{validation}}} \mathcal{P}\big(\widehat{y}^{(k)}(x\,;\, \mathcal{D}^{(k)}_{\text{train}}), y\big),$$

where $\widehat{y}^{(k)}$ is the predictor trained for repetition k.

Once again, if needed, hyper-parameter optimization may take place by treating E_{cv} as a function of the hyper-parameter in question. Also, as mentioned above in situations where the total number of observations is low and if not tweaking parameters then K-fold cross validation may serve as an alternative approach to general performance evaluation using a train-test split. Again as with the train-validate-test split approach, there are multiple variations for K-fold cross validation with the exact method used in practice often depending on the specific situation encountered.

2.6 A Taste of Unsupervised Learning

Now that we have explored key aspects of supervised learning in the sections above, let us get a taste for the basics of unsupervised learning. In this context the data is unlabelled and is denoted via $\mathcal{D} = \{x^{(1)}, \ldots, x^{(n)}\}$. Here we assume that each sample or observation $x^{(j)}$ is a p dimensional vector of features in Euclidean space. Observe that there are no labels $y^{(j)}$.

We briefly introduce two popular unsupervised learning methods, one for clustering and one for data reduction. These are respectively the *K-means* algorithm and the framework of *principal component analysis* (PCA). In exploring PCA we also take a slightly deeper look at the linear algebraic concept of singular value decomposition (SVD) already used in Section 2.3 in the context of pseudo-inverse representation. Here we see how SVD has applications to data compression, a notion sometimes used in more complex deep learning models. Further reference to more advanced supervised learning methods are in the notes and references at the end of the chapter.

K-means Clustering

The machine learning activity of *clustering* allows us to identify meaningful groups, or clusters, among the data points and find representative centers of these clusters. The aim is

that the samples within each cluster are more closely related to one another than samples from different clusters.

Formally, for the dataset $\mathcal{D}$, clustering is the act of associating a cluster ℓ with each observation, where ℓ comes from a small finite set, $\{1, \ldots, K\}$. That is, a clustering algorithm works on the data $\mathcal{D}$ and outputs a function $c(\cdot)$ which maps individual data points to the label values $\{1, \ldots, K\}$. The clustered data (algorithm output) is then a collection of clusters denoted via,

$$C_\ell = \left\{ x^{(j)} \mid c(x^{(j)}) = \ell,\ j \in \{1, \ldots, n\} \right\}, \quad \text{for} \quad \ell = 1, \ldots, K.$$

A clustering algorithm attempts to choose the clusters such that the elements of each C_ℓ are as homogenous as possible.

The K-means algorithm is one very basic, yet powerful heuristic algorithm. With K-means, as with several other types of clustering algorithms, we pre-specify a number K, determining the number of clusters that we wish to find. Hence K may be treated as a hyper-parameter. As the algorithm seeks the function $c(\cdot)$, or alternatively the partition $C_1, \ldots, C_K$, it also seeks representative *centers* (also known as *centroids*), of the clusters, denoted by $J_1, \ldots, J_K$, each an element of $\mathbb{R}^p$.

One may view the ideal aim of K-means as minimization of,

$$\text{Clustering loss} = \sum_{\ell=1}^{K} \sum_{x \in C_\ell} \|x - J_\ell\|^2. \tag{2.39}$$

Such a minimization is generally computationally intractable since it requires considering all possible partitions of $\mathcal{D}$ into clusters. Yet it can be approximately minimized via the K-means algorithm using a classic iterative approach. The K-means algorithm does this by separating the problem into two sub-problems or sub-tasks called *mean computation*, and *labelling*. We define these now.

Mean computation: Given $c(\cdot)$, or a clustering $C_1, \ldots, C_K$ where $|C_\ell|$ denotes the number of elements in cluster ℓ, find $J_1, \ldots, J_K$ that minimizes (2.39) via,

$$J_\ell = \frac{1}{|C_\ell|} \sum_{x \in C_\ell} x, \qquad \text{for} \qquad \ell = 1, \ldots, K. \tag{2.40}$$

Here each J_ℓ is the vector obtained via the element-wise average over all the vectors in C_ℓ where each of the p coordinates is averaged separately.

Labelling: Given, $J_1, \ldots, J_K$ and assuming these values are fixed, find $c(\cdot)$ that minimizes (2.39) for every $x \in \mathcal{D}$. This is done by setting,

$$c(x) = \underset{\ell \in \{1, \ldots, K\}}{\operatorname{argmin}} \|x - J_\ell\|. \tag{2.41}$$

That is, the label of each element is determined by the closest center in Euclidean space.

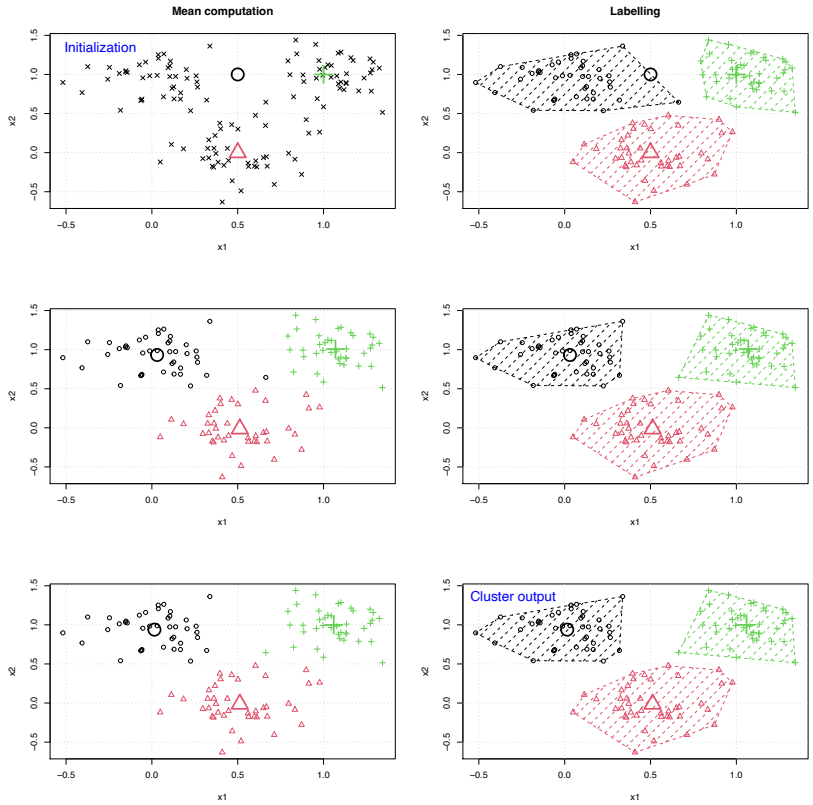


Figure 2.12: Workflow of the K-means algorithm on synthetic data with $K = 3$. The left column is the mean computation step and the right column is the labelling step. Each row is an iteration and the algorithm converges after three iterations. Initialization is presented in the top left corner and after three iterations the algorithm converges with the output presented in the bottom right corner.

The K-means algorithm starts with randomly initialized[19] centers $J_1, \ldots, J_K$. A labelling step is then executed and these initial random centers are then used to determine initial labels according to (2.41). The algorithm then iterates over the mean computation step (2.40), followed by the labelling step (2.41) and repeats the two steps one after the other. This is done until no more changes are made to the labels and the means. Such an iteration generally does not find the absolute minimum of the objective (2.39), however the approximation found is often satisfactory.

In Figure 2.12 we illustrate the workflow of the algorithm on synthetic data where we choose $K = 3$. The top left plot is the intilization with three random means represented by the black circle, the red triangle, and the green cross. Then each row of Figure 2.12 represents one iteration of the algorithm where the plots on the left column show the output of mean computation (except for the first row which is initilization), and the plots on the right column show the output of labelling.

[19] These may be randomly selected elements of $\mathcal{D}$ or some other random set of vectors.

Note that in practice when presented with data $\mathcal{D}$, one typically first standadizes the data as presented in Section 2.1. Then the process of selecting the hyper-parameter K which is external to the K-means algorithm is carried out. One way to do so is to run K-means for increasing values of K and seek a *knee point* or *elbow* when plotting (2.39) as a function of K. As K increases the objective (2.39) generally decreases, however beyond a certain K the value of adding further clusters quickly diminishes. In some cases, such as the visual pixel segmentation we present below, visual subjective measures can be used to find the most appropriate K.

Image Segmentation with K-means

We have already briefly discussed image segmentation in Section 1.1; see Figure 1.2 in that section. As discussed, the goal of image segmentation is to label each pixel of an image with a unique class from a finite number of classes. In Chapter 6 we briefly describe a supervised approach called semantic image segmentation which uses labeled data, namely class masks in addition to the image for training. Nevertheless, in the absence of such information one may still carry out unsupervised image segmentation. One way to carry out this task is to use the K-means clustering algorithm where each pixel of the image is considered a point in $\mathcal{D}$ and the dimension of each point is typically $p = 3$ (red, green, and blue) for color images. This can produce impressive image segmentation without any other information except for the image.

Figure 2.13 presents the segmentation of a color image where K-means is used for grouping the pixels into K different clusters. This color image in Figure 2.13 (a) is a $n = 640 \times 640 = 409,600$ pixel color image ($p = 3$). The segmentation consists of running the K-means algorithm which groups similar pixels based on their attributes and assigns the attributes of the corresponding cluster center to the pixel in the image. Figure 2.13 (b) presents the result of the segmentation using $K = 6$ and Figure 2.13 (c) does so with $K = 2$.

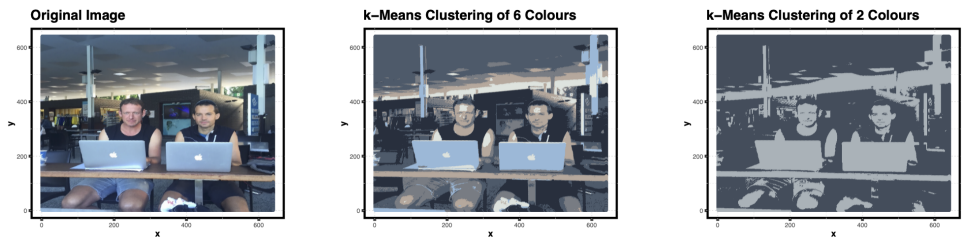


Figure 2.13: Unsupervised image segmentation using K-means. (a) Original image. (b) $K = 6$. (c) K=2.

Matrices in Unsupervised Learning

We often organize the data $\mathcal{D} = \{x^{(1)}, \ldots, x^{(n)}\}$ in the *data matrix* $X_{\mathcal{D}}$, similar to the design matrix (2.10) by stacking each observation vector $x^{(j)}$ in a separate row. The difference between the design matrix and $X_{\mathcal{D}}$ is that the latter does not have a first column of 1s. Thus $X_{\mathcal{D}}$ is an $n \times p$ matrix where the ith column has the data samples for feature i.

It is useful to *de-mean* the data by defining the *centered data matrix*,

$$X = X_{\mathcal{D}} - \mathbf{1}\overline{x}^{\top}, \tag{2.42}$$

where we (re)use the notation X for the matrix previously used for the design matrix and where $\mathbf{1}$ is a column vector of 1s of length p. In the centering process, the p dimensional vector $\overline{x}$ has coordinates $\overline{x}_i$ which are sample means of the features as defined in (2.1). That is, for each column (feature) in $X_{\mathcal{D}}$ we subtract the mean of the feature. Thus the new $n \times p$ matrix X has features that each have a sample mean of 0.

An important matrix for such data is the $p \times p$ *sample covariance matrix*,[20]

$$S = \frac{1}{n} X^{\top} X. \tag{2.43}$$

Written in scalar form, the (i, j)th element of the symmetric matrix S is,

$$S_{i,j} = \frac{1}{n} \sum_{k=1}^{n} (x_i^{(k)} - \overline{x}_i)(x_j^{(k)} - \overline{x}_j).$$

and it estimates the *covariance* between feature i and feature j. On the diagonal of S where $i = j$, $S_{i,i}$ equals the sample variance s_i^2 of (2.1). The off diagonal entries account for the measure and direction of linear dependence between features.

We note that the sample covariance matrix can be further normalized to a *sample correlation matrix* by dividing each (i, j)th entry by the product of the sample standard deviations, $s_i s_j$. Sample correlation matrices are important for multivariate descriptive statistics, yet we do not use them explicitly now. Our focus is rather on PCA which we introduce in terms of the de-meaned data matrix X and the sample covariance matrix S.

Principal Component Analysis

It is often the case that not all p dimensions of the data are equally useful. This is especially the case in the presence of high dimensional data (large p). Moreover, many features may be either completely redundant or uninformative. These cases are referred to as *correlated features* or *noise features* respectively. In such cases and others, *principal component analysis* (PCA) is often employed. It is a well-known and widely used dimensionality reduction technique applicable to a wide variety of applications such as data compression, feature extraction, and visualization.

The basic idea of PCA is to project each point of $\mathcal{D}$ which has many correlated coordinates onto fewer coordinates, called *principal components*, which are uncorrelated. This is done while still retaining most of the variability present in the data. In this setting, PCA offers a low-dimensional representation of the features that attempts to capture the most important information from the data. The principal components found via PCA are a new reduced set of features, indexed by $i = 1, \ldots, m$ where $m < p$ is some specified lower dimension. For visualization we often take $m = 2$ or $m = 3$. In other applications, such as for example the integration of PCA as part of other machine learning procedures, m is often calibrated as a hyper-parameter.

[20] In a statistical context one often uses $n - 1$ in the denominator instead of n. For non-small n this distinction is insignificant. See a similar comment in relation to (2.1)

As input, PCA uses the de-meaned data from the centered data matrix X of (2.42) where we denote by $x_{(i)}$ the ith column of X (corresponding to a vector of feature i for all n observations). PCA uses a linear combination of these columns to arrive at the vectors of the new features $\tilde{x}_{(1)}, \ldots, \tilde{x}_{(m)}$. This can simply represented as

$$\tilde{x}_{(i)} = v_{i,1} \begin{bmatrix} | \\ x_{(1)} \\ | \end{bmatrix} + v_{i,2} \begin{bmatrix} | \\ x_{(2)} \\ | \end{bmatrix} + \ \ldots \ + v_{i,p} \begin{bmatrix} | \\ x_{(p)} \\ | \end{bmatrix} \quad \text{for} \quad i = 1, \ldots, m,$$

where each new n dimensional vector, $\tilde{x}_{(i)}$, is a linear combination of the original features. The coefficients of this linear combination can be organized in the vector $v_i = (v_{i,1}, \ldots, v_{i,p})$ which is called the *loading vector* for i. Thus $\tilde{x}_{(i)} = X v_i$. This can also be represented for all the reduced features and loading vectors together via,

$$\underbrace{\begin{bmatrix} | & & | \\ \tilde{x}_{(1)} & \ldots & \tilde{x}_{(m)} \\ | & & | \end{bmatrix}}_{\substack{\widetilde{X}_{n \times m} \\ \text{Reduced data}}} = \underbrace{\begin{bmatrix} | & & & | \\ x_{(1)} & \ldots & \ldots & x_{(p)} \\ | & & & | \end{bmatrix}}_{\substack{X_{n \times p} \\ \text{Original de-meaned data}}} \times \underbrace{\begin{bmatrix} | & & | \\ | & & | \\ v_1 & \ldots & v_m \\ | & & | \\ | & & | \end{bmatrix}}_{\substack{\widetilde{V}_{p \times m} \\ \text{Matrix of loading vectors}}} . \tag{2.44}$$

It turns out the a very useful way to represent the loading vectors $v_1, \ldots, v_m$ is by normed eigenvectors associated with eigenvalues of the sample covariance matrix S as in (2.43). Specifically, since S is symmetric and positive semidefinite, the eigenvalues of S are real and non-negative, a fact which allows us to order them via $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_p \geq 0$. We then pick the loading vector v_i to be a normed eigenvector associated with λ_i, namely,

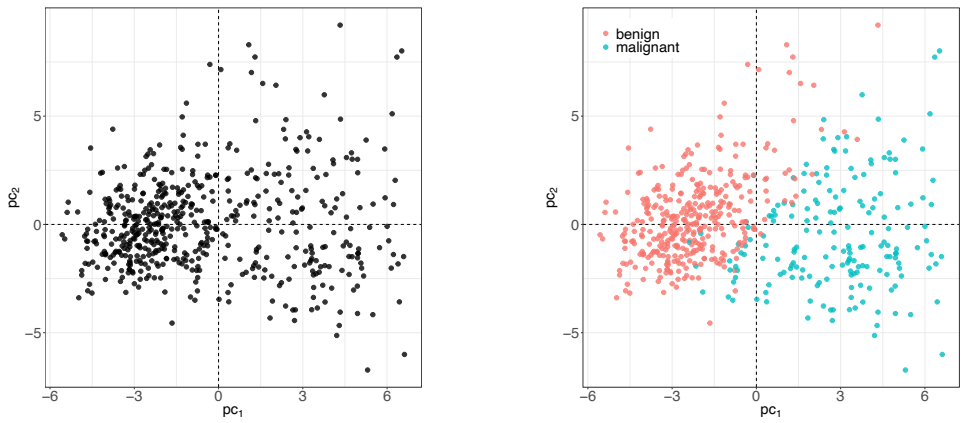
$$S v_i = \lambda_i v_i, \tag{2.45}$$

while keeping in mind that the first loading vector is associated with the highest eigenvalue; the second is associated with the second highest eigenvalue; and so forth. The symmetry of S also means that its eigenvectors are orthogonal and hence $\widetilde{V}$ is a matrix with orthonormal columns. In this setting we assume that at least the first m eigenvalues are strictly positive, namely $\lambda_m > 0$.

In the subsection below we derive the main result to show why this choice of loading vectors based on eigenvectors is attractive. At this point let us consider a numerical example.

We return to the Wisconsin breast cancer data used in Section 2.2 where $p = 30$ and $n = 569$. To visualize this data using PCA we set $m = 2$ and compute the first two loading vectors from the 30×2 matrix $\widetilde{V}$ using standard numerical procedures for eigenvalues and eigenvectors. Then by multiplying the 569×30 demeaned data matrix X by $\widetilde{V}$ as in (2.44) we get the 569×2 matrix $\widetilde{X}$ of principal components. We then plot each row which is 2 dimensional in Figure 2.14 (a).

On their own, the two dimensional points in Figure 2.14 (a) may not be insightful. After all the principal components coordinates `pc1` and `pc2` do not have any physical meaning in this context. Nevertheless, if we consider Figure 2.14 (b) where we frame this as a supervised learning problem and color the points based on the labels `benign` vs. `malignant`, a useful pattern emerges. There is quite a clear separation between the two classes and hence there is



(a) (b)

Figure 2.14: Breast tumor data samples projected on the two first principal component from the PCA. (a) Unlabeled data. (b) Once adding the label to each sample a pattern and separation between `benign` vs. `malignant` appears.

potential to classify points by separating the region in the principal components plane. We do not discuss concrete examples of constructing a classifier in this case. We rather point out that the data following a PCA transformation with reduced dimension $m < p$, can often be used as input to a supervised learning algorithm.

Derivation of PCA

The PCA framework tries to project the data in the directions with maximum variance. Returning to (2.44), since $\tilde{x}_{(i)} = Xv_i$ we can formulate this by maximizing the sample variance of the components of $\tilde{x}_{(i)}$. Keeping in mind that $\tilde{x}_{(i)}$ is a 0 mean vector, its sample variance using (2.1), is simply $\tilde{x}_{(i)}^\top \tilde{x}_{(i)}/n$. Hence substituting $\tilde{x}_{(i)} = Xv_i$, we have,

$$\text{Sample variance of component } i = \frac{1}{n} v_i^\top X^\top X v_i = v_i^\top S v_i,$$

where S is the sample covariance of the data as in (2.43). Thus in searching for the first loading vector v_1 we have the optimization problem,

$$\max_{v \in \mathbb{R}^p} v^\top S v, \quad \text{subject to} \quad \|v\| = 1. \tag{2.46}$$

Note the constraint which seeks a normalized direction v with $\|v\| = 1$ which is equivalent to $v^\top v = 1$. This representation allows us to use Lagrange multiplier techniques where we convert this constrained problem to an unconstrained quadratic problem. The objective is then,

$$\max_{v,\ \lambda} v^\top S v + \lambda \left(1 - v^\top v\right) \quad \text{or} \quad \max_{v,\ \lambda} v^\top S v - v^\top \lambda v + \lambda, \tag{2.47}$$

with the Lagrange multiplier λ and the constraint $1 - v^\top v = 0$. Now taking derivatives with respect to the vector v (see also Appendix A) we obtain $Sv - \lambda v$. Thus the first-order conditions in terms of v reduce to the eigenvalue problem $Sv = \lambda v$. This means that any eigenvector v of S adheres to the first-order conditions for the optimization problem (2.46).

By multiplying the eigenvalue equation by $v^\top$ we get $v^\top S v - v^\top \lambda v = 0$. As apparent from the representation on the right hand side of (2.47), this means that any eigenvector yields a maximization objective which is equal to the corresponding eigenvalue λ. Hence the optimization problem is solved by choosing the maximal eigenvalue λ_1 with v being an associated normalized eigenvector, v_1 as in (2.45).

The subsequent directions v_i for $i = 2, \ldots, m$ are chosen by maximizing the variance of new linear combinations which are orthogonal to previous ones. That is, the directions capture the part of variance which has not been previously captured. It can be shown that a normalized eigenvector which matches the second eigenvalue, v_2 maximizes the variance once the direction of v_1 is removed. This then continues for $i = 3, \ldots, m$ and in summary principal components are determined via (2.45).

PCA Through SVD

We have already used the *singular value decomposition* (SVD) in Section 2.3 in the context of the Moore-Penrose pseudo-inverse. We now further revisit the construction of SVD from linear algebra and see the relationship between SVD and PCA.

Any $n \times p$ dimensional matrix X of rank r can be represented as

$$X = U \Delta V^\top = \sum_{i=1}^{r} \delta_i \, u_i \, v_i^\top, \quad \text{with} \quad \Delta = \operatorname{diag}(\delta_1, \ldots, \delta_r), \quad \text{and} \quad \delta_i > 0. \tag{2.48}$$

Here the $n \times r$ matrix U and the $p \times r$ matrix V are both with orthonormal columns denoted u_i and v_i respectively for $i = 1, \ldots, r$. These columns are called the left and right *singular vectors* respectively. The values δ_i in the $r \times r$ diagonal matrix Δ are called *singular values* and are ordered as $\delta_1 \geq \delta_2 \geq \cdots \geq \delta_r > 0$. Note that this representation of the SVD differs from the one employed near (2.17) in Section 2.3 where U and V were taken as square matrices and Δ was not necessarily square. The form used in Section 2.3 is sometimes called the *full SVD* and the form we present here is called the *reduced SVD*.

Consider now X again as the de-meaned data matrix (2.42). Now using its SVD representation in the sample covariance (2.43) we obtain

$$S = \frac{1}{n} \underbrace{V \Delta^\top U^\top}_{X^\top} \underbrace{U \Delta V^\top}_{X} = \frac{1}{n} V \Delta^2 V^\top, \quad \text{with} \quad \Delta^2 = \operatorname{diag}(\delta_1^2, \ldots, \delta_r^2).$$

Here the fact that U has orthonormal columns implies $U^\top U$ is the $r \times r$ identity matrix and hence it cancels out. Hence,

$$S = \sum_{i=1}^{r} \frac{\delta_i^2}{n} \, v_i \, v_i^\top. \tag{2.49}$$

We can now compare to the eigenvector based representation of PCA where $\widetilde{V}$ is the matrix of PCA loading vectors as in (2.44). Take $m = r$ and denote by Λ the diagonal matrix with diagonal entries as the eigenvalues of S in decreasing order $\lambda_1 \geq \ldots \geq \lambda_r > 0$. Now using (2.45) we have the *spectral decomposition* of S,

$$S = \widetilde{V}^\top \Lambda \widetilde{V} = \sum_{i=1}^{r} \lambda_i \, v_i \, v_i^\top. \tag{2.50}$$

We now compare (2.49) and (2.50) and see that with $\lambda_i = \delta_i^2/n$ the loading vectors in (2.50) are the right singular vectors in (2.49). That is, $\widetilde{V} = V$.

Further, to obtain the data matrix of principal components, $\widetilde{X}$ of (2.44) we set $\widetilde{X} = XV$. Now using the SVD representation of X, (2.48) and assuming $m = r$, PCA can be represented as,

$$\widetilde{X} = \underbrace{U\Delta V^\top}_{X} V = U\Delta = \begin{bmatrix} | & | & & | \\ \delta_1 u_1 & \delta_2 u_2 & \dots & \delta_r u_r \\ | & | & & | \end{bmatrix}. \tag{2.51}$$

That is, each column of the reduced data matrix $\widetilde{X}$ is a left singular vector u_i stretched by the singular value δ_i. Further, for $m < r$ we only take the first m columns.

With these relationships between PCA and SVD, numerical methods for computing the SVD decomposition of X can be used for PCA. Indeed in practice, efficient and numerically robust computational methods for SVD are employed for PCA.

SVD for Compression

The singular value decomposition can also be viewed as a means for compressing any matrix X. Specifically, consider the SVD representation in (2.48) with $\delta_1 \geq \delta_2 \geq \ldots \geq \delta_r$. Then a rank $m < r$ approximation of X is,

$$\widehat{X} = \sum_{i=1}^{m} \delta_i \, u_i \, v_i^\top \approx X, \qquad \text{where} \qquad X - \widehat{X} = \sum_{i=m+1}^{r} \delta_i \, u_i \, v_i^\top. \tag{2.52}$$

The rank of $\widehat{X}$ is m and since one often uses m significantly smaller than r, this is called a *low rank approximation*. For small enough δ_{m+1} the approximation error is negligible since the summation of rank one matrices $\delta_i \, u_i \, v_i^\top$ for $i = m+1, \ldots, r$ is small. Observe that the number of values used in this representation of $\widehat{X}$ is $m \times (1 + n + p)$ and for small m this number is generally much smaller than $n \times p$ which is the number of values in X. Hence this may viewed as a compression method.

The usefulness of such low rank approximations is validated by a theoretical result called the *Eckart-Young-Mirsky theorem*. Here we consider the approximation-error matrix $X - \widehat{X}$ and we seek to have the best rank m approximation in terms of minimization of $\|X - \widehat{X}\|$. The theorem works for several types of matrix norms, yet here let us focus on the Frobenious norm[21] denoted $\|A\|_F$ for any matrix A. We now have for the Frobenious norm,

$$\min_{\widehat{X} \text{ of rank } m} \left\| X - \widehat{X} \right\|_F^2 = \left\| X - \sum_{i=1}^{m} \delta_i \, u_i v_i^\top \right\|_F^2 = \sum_{i=m+1}^{r} \delta_i^2. \tag{2.53}$$

Singular value decomposition based matrix approximations such as (2.52) are useful in multiple domains including improvement of neural network model size. We do not discuss these topics specifically in this book. Instead, consider a simple visual example with a 353×469 monochrome (grayscale) image appearing at the bottom right of Figure 2.15; this is X. Then the other images in Figure 2.15 are $\widehat{X}$ with $m = 10$, $m = 30$, and $m = 50$. As is evident, the $m = 50$ approximation appears close to the original image. The main plot

[21] This is the square root of the sum of the squared elements of the matrix.

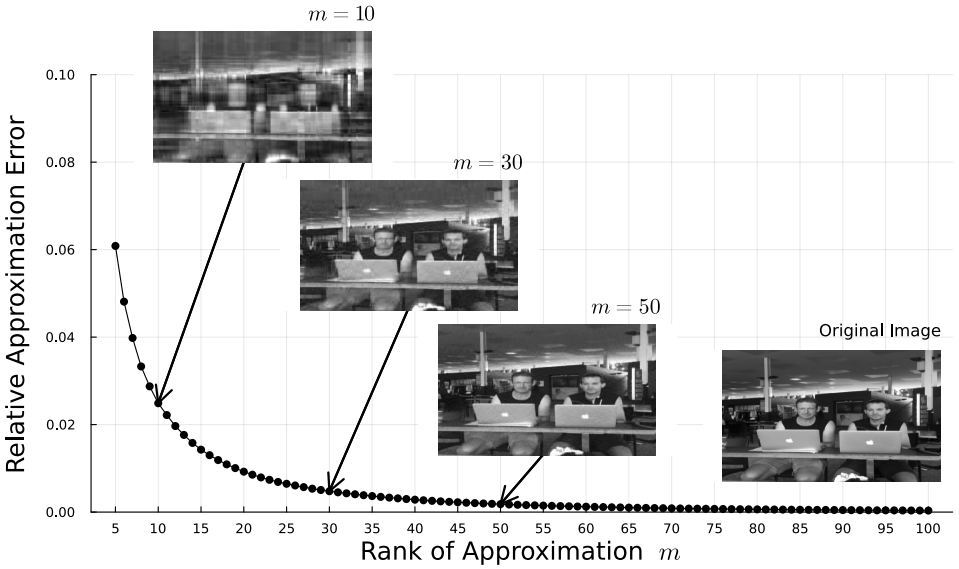


Figure 2.15: SVD for data compression: The original image is presented based on compression with $m = 10$ singular values, $m = 30$ singular values, and $m = 50$ singular values. The images are presented in terms of the relative approximation error based on the Frobenious norm.

in the figure is the relative approximation error as given by the right hand side of (2.53) divided by the sum of all singular values squared.

Note that the original image uses $353 \times 469 = 165,557$ values while the $m = 50$ approximation only uses $50 \times (1 + 353 + 469) = 41,150$ values. That is the approximation yields $\widehat{X}$ which is compressed to about 25% of the size of X and looks very similar.

Note that variants of the types of plots as in Figure 2.15 are also common when carrying out PCA. In that context the plot is called a *scree plot* and it presents the percentage of variance explained by the principal components.

Notes and References

One does not need to master all other branches of machine learning to understand deep learning, nevertheless getting a taste for key elements of the field is useful. Beyond the basics that we presented in this chapter, one may consult several general machine learning texts. We recommend [240] for a comprehensive mathematical account of practical machine learning and we recommend the more classic [39] as an additional resource. Further, the book [299] provides a probabilistic approach. Focusing on linear algebra, the introductory book [56] is a good introduction to foundations such as K-means, least squares, and ridge regression. Further, [391] provides a richer context covering PCA, SVD, and many aspects of matrix algebra appearing in machine learning. Finally for a short read which provides an overview of many practical aspects of machine learning, see [68]. An additional recommended reference is [263].

The worlds of machine learning and statistical inference are intertwined and methods developed in one field are often used in the other field and vice versa. For those with expertise in one or both of the fields it is quite easy to spot the differences between the approaches, however for those entering these worlds afresh it may be helpful to read the survey paper "Statistical modeling: The two cultures" by Leo Breiman, [61]. On that note, to get a feel for many statistical aspects of *linear regression*, see, e.g., [296] or one of many other statistical books. Note that [296] is also a good reference for understanding *interaction terms*, a concept that we mentioned in the chapter and did not cover. A general text that integrates methodology and algorithms with statistical inference and machine learning together with speculations of future directions is [115].

Throughout this chapter we have made reference to several aspects of statistics or machine learning that are not studied further in this book. Here are some references for each. In general, a good reference for likelihood based inference is [31]. Specifically *Akaike information criterion* (AIC), introduced in [7], and the *Bayesian information criterion* (BIC) are surveyed in [432]. A general class of models also appearing in the next chapter is *generalized linear models* (GLMs); these first appeared in [304] and a good contemporary applied reference is [121]. Other models are *general additive models* (GAMs) which extend generalized linear models in which some predictor variables are modelled by smooth functions; see [168]. In terms of non-linear regression the *LOESS* method is a generalization of moving average and polynomial regression, see [88]. Further, *Nadaraya-Watson kernel regression* is a non-parametric regression method in which a kernel function is exploited; see [378].

We have covered the basics of decision theory via binary classification however there are many more studies for these aspects. See the comprehensive survey [118] on metrics for binary classification as well as [158]. For a discussion of different uses of receiver operating curves and different approaches for them see [58] and [315]. The origins of the F_1 score can be attributed to Cornelis Joost van Rijsbergen who introduced the *effectiveness function* of which F_1 score is a special case; see [408]. The SMOTE method for dealing with unbalanced data is from [75]. See also the surveys [153], [211], and [348].

We briefly mentioned the differences between discriminative and generative learning. More on the topic is in chapter 9 of [299] together with a treatment of the *naive Bayes classifier* and *linear discriminant analysis* (LDA). The area of *support vector machines* became extremely popular in the world of machine learning with their height of popularity during the 1990's and the decade that followed. A complete treatment of these methods is in [240] together with associated ideas of *kernel methods*. Specific to this area is the concept of *VC dimension* (standing for Vapnik–Chervonenkis) which we did not cover here; see [409].

Decision trees are also very popular machine learning techniques; see chapter 8 of [240] for an overview. Within the study of machine learning, generic methods of *boosting* and *bagging* are prominent in the context of decision trees. Specifically see [366] and [59]. The *random forest* algorithm is one such method that has been hailed the most usable ad-hoc generic method when there is not further information about the problem; see [60]. *Gradient boosting* has become very popular due to a software package called *XGBoost*; see [77]. The *K-nearest neighbours* classification algorithm that we mention is often used as an introductory example. See for example Section 2.3 of [166].

The origins of *least squares* fitting are from the turn of the 19'th century, initially with applications to astronomy. The first least squares publication is typically attributed to an 1809 paper by Gauss

[132] although an earlier 1805 publication by Legendre publicized the concept first. An interesting historical investigation into "who invented least squares" is in [389]. Since then, least squares methods have become some of the most prominent tools in applied mathematics. The *Moore-Penrose pseudo inverse* was independently described in [297], [40], and [328]. *Singular value decomposition* (SVD) has origins in differential geometry with the first linear algebra publication typically associated with the 1936 Eckart Young paper [114]. To the best our knowledge the first association between SVD and least squares is in [139]. The survey [105] may also be of interest as it contrasts different numerical methods for least squares.

Aspects of *multi-collinearity* are treated in many statistical contexts, see for example [281]. Using regression with other methods such as *absolute error loss* (robustness) is covered in [302] and for a reference on the *Huber error loss* see [197]. See also [160] for a discussion on dealing with categorical input features by conversion to numbers.

The origins of gradient descent are attributed to Cauchy from 1847 with [72], way before the invention of any digital electronic computer; see [253] for an historical account. The analysis of the loss landscape in machine learning has been studied multiple times, see for example [279] for a survey, and [284] for theoretical result in a high dimensional context.[22]

We have only touched the tip of the iceberg in terms of *model selection*. See [390] for a survey as well as the book [87]. There are also recent developments such as [26] dealing with other approaches for balancing underfitting and overfitting or bias and variance. In general, model selection is still a very active open avenue of research. Our discussion surrounding the generalization gap follows similar lines to [263]. Our example in Figure 2.10 is inspired by a similar example in [39].

For an excellent reference dealing with the *LASSO* method see [167]. The original *Tikhonov regularization* (*ridge regression*) technique appeared in 1943 in [401] yet it is believed to have been invented in parallel in other contexts as well. *Elastic net* models are more of a speciality; see [453] and also relations to support vector machines in [452]. Generalizations, variants, and discussion of issues arising with *K-fold cross validation* are in [424] and [232]. See also [215] and [62] for recent developments as well as [443] for stratified cross-validation and variants.

The accepted first reference for *K-means* clustering is [277] from 1967 although the method was known prior. An applied book to understand the main concept and algorithms for cluster analysis is [226]. A recent comprehensive survey about clustering adding to what we presented here is in [120]. Further clustering approaches include *hierarchical clustering*, see [300]. See also an older general survey in [208]. Principal component analysis was first proposed by Pearson in [326] with initial ideas also attributed to Hotelling in [189]. A substantial book on the topic is [212]. Relationships to SVD are well explained in [391] and the first appearance of the *Eckart-Young-Mirsky theorem* for SVD was in [114] by Eckart and Young. It was further independently extended by Mirsky in [292]. One popular efficient method for numerical computation of SVD is the so called *Golub-Reinsch algorithm* first introduced in [140]. A further overview of additional SVD algorithms is in [89].

[22] See also https://losslandscape.com/ for a visual presentation of different loss landscapes.