Fundamentals of Machine Learning

V. V. Veeravalli and J. G. Ligo

February 4, 2021

Contents

Chapter 1

Introduction

This is a set of notes for the "Fundamentals of Machine Learning" section of ECE 365 at the University of Illinois at Urbana-Champaign. It is a short introduction to a few key topics in machine learning and statistics. The course will assume you are somewhat familiar with basic programming (Python will be used in the course, but no prior knowledge of Python is assumed), basic linear algebra (MATH 286 is sufficient), and probability (ECE 313 is sufficient). The notes are interspersed with numerous references, many of which are freely available online (some of these are only accessible on campus, or through the university library's proxy server).

1.1 What is Machine Learning?

Machine learning is a set of probabilistic tools to analyze data in an automated fashion. Machine learning can be broadly divided into two categories: supervised learning and unsupervised learning. Here we discuss the basic formulations of each category.

In *supervised learning*, we start with a set of example data known as the *training set* $\mathcal{T} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$ drawn independently and identically distributed (i.i.d.) from some unknown joint distribution $p(\boldsymbol{x}, y)$. The vectors $\{\boldsymbol{x}_i\}$ are known as *feature vectors* (*inputs*), whereas the values $\{y_i\}$ are known as *labels* (*outputs*, *responses*). The vector space containing the feature vectors is known as the *feature space*. One can think of there being a system that takes in randomly datapointd inputs (and possibly some other (random) data, which is not accessible) and produces the output labels; these together form the training set. When the labels are drawn from a finite set (e.g. take values in $\{-1, 1\}$), the supervised learning problem is known as *classification* or *pattern recognition*. The labels in this case are known as *classes*. When the labels are drawn from an infinite set (e.g. real-valued), the supervised learning problem is known as *regression*. In supervised learning, our goal is to use the training data to find a map (function or *model*), which, given a feature vector predicts the corresponding label. This process is called *learning* or *training* a model. Once we find a map relating the feature vectors and labels (there are typically many choices of maps), we try to determine how the map *generalizes*, i.e., how well does the model we found using the training data work on data we have not seen yet? A good supervised learning algorithm should yield a model that generalizes well (i.e. makes predictions that are accurate for data we have not seen). Note

that a model does not need to match the true phenomena at hand in order to generalize well—for many tasks we do every day, our mental models are extremely different from the true phenomena, yet we can complete them well.

Example: Consider describing a person using a feature vector consisting of their height, weight, age, sex, and cholesterol level. Based on these features, we may want to know if they are at risk for heart disease. A supervised learning approach for predicting if someone is at risk for heart disease would start by collecting data from a set of people, i.e., record their feature vectors and if they have heart disease or not. This provides a set of feature vector and label pairs, forming the training set. Based on the training set, we identify a model (function) that takes in a person's features and outputs a prediction of whether they will have heart disease or not. Then, we go find some people (not in our training set) and see how our model matches up with reality to *validate* our model. Note that some features are *categorical* (take on values in a finite set, e.g. sex) whereas others are *continuous* (are real-valued, e.g. height, weight). We may want to predict the probability of getting heart disease within the next 5 years (which is a regression problem), or if they will have heart disease in the next 5 years (which is a yes-no question, and therefore a classification problem).

In *unsupervised learning*, we start with a data set $\mathcal{D} = \{\boldsymbol{x}_i\}_{i=1}^N$ drawn from some unknown distribution $p(\boldsymbol{x})$ i.i.d., and the goal is to infer something interesting or useful about the data. Unlike supervised learning, we only have feature vectors in this case. The *clustering* problem is to take vectors in $\mathcal{D}$ and group them into sets that are "similar". We may also want to find relations among features based on the data set. Unsupervised learning is inherently a less well-defined problem than supervised learning; some describe it as an art rather than a science [43].

Example: Consider a supermarket that collects feature vectors corresponding to what shoppers buy (this can be acquired through rewards cards, for example). By applying clustering, the supermarket can lump together shoppers who have similar purchasing behavior. One thing the supermarket may discover, surprisingly, is whether a customer has a baby on the way [14]. From this knowledge, the supermarket can use advertising and coupons to steer customers towards certain brands in their stores. Another unsupervised procedure is market basket analysis—finding groups of items that are bought together—to suggest things for customers to add to their carts (e.g. the "Customers Who Bought This Item Also Bought" section on Amazon.com).

In both supervised and unsupervised learning, we develop *algorithms* to help us tease something useful out of data. In supervised learning, generalization performance as a function of amount of training data determines the quality of an algorithm. Note that in addition to just accuracy, it is often important to consider the resources needed by the algorithm, e.g. factors from computational complexity (time, memory, I/O operations, etc.).

To summarize, in supervised learning we discover things about data with the help of examples (like being a student in a class with a teacher). In unsupervised learning we discover things about data on our own (like taking an independent study). In both approaches, one has considerable

freedom as to how to use the data. It is important to note that there is no "best" way of doing machine learning, and often, domain-specific knowledge is crucial for designing the "best" algorithms for machine learning—this is often called the *no free lunch theorem* [36]. Often, there is considerable benefit in choosing the feature vectors appropriately (see, e.g., Chapters 5 and 16 of [17] for applications in computer vision where we see tremendous gains from good *feature selection*), which requires knowledge of the problem. For example, in the heart disease example, alcohol and tobacco consumption would be useful features to add to improve classification, while adding liking for REO Speedwagon[1] would be a bad feature (unless it has some latent connection to lifestyle factors such as alcohol and tobacco consumption), and just make classification harder.

1.2 References and Ancillaries

Our goal for this part of the class is to build a solid base for understanding machine learning, by presenting the workhorse tools of machine learning with enough background to have an appropriate understanding of how and why they work.

There are several good books on the topics we cover, which present more mathematical formulations, as well as many more topics than we have time to cover in a few weeks–we will refer to some of them for details on specific topics.

It is also useful to know several software packages for this course (and in general). Many talented people have worked on making algorithms for machine learning run quickly. While we will use Python and the scikit-learn package in this section of the course, it is also useful to be somewhat familiar with the R programming language and MATLAB to work in this field. In practice, if the software you are working with provides the right functionality, you should use it. However, for the purposes of the course, we will have you implement some machine learning algorithms in order to help understanding. The figures in these notes were done in a combination of Python + Scikit-Learn [38], GNU Octave (`https://www.gnu.org/software/octave/`, a free scientific programming environment mostly compatible with MATLAB) and GNU R (an implementation/extension of the S programming language, `https://www.r-project.org/`). The Scikit-Learn documentation[38] contains many excellent examples: `http://scikit-learn.org/stable/auto_examples/`.

To review probability, we suggest the ECE 313 notes written by Prof. Hajek [22]. A good source for linear algebra is Strang's textbook [41].

For experimentation and practice beyond the course, a great source of data sets is the UCI Machine Learning Repository: `https://archive.ics.uci.edu/ml/index.html` and Kaggle: `https://www.kaggle.com/`. Many machine learning packages also have subsets of datasets included as parts of demos.

As with any set of notes, there are always errors and/or unclear passages. Feedback is appreciated, and can be sent to Prof. Veeravalli by email.

[1] A famous rock band from Champaign, IL; Neal Doughty was in the Electrical Engineering program at Illinois, and Alan Gratzer also attended Illinois.

1.3 Notation

1. Capital Letters: Random variables, matrices, number of classes (clear by context)
2. **Bold face**: vectors (which are column vectors, unless transposed)
3. Hats: estimates ($\hat{\cdot}$)

Chapter 2

Classification

In this chapter, we will discuss how to learn classifiers from a set of training data $\mathcal{T} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$. Recall that a classifier $f(\boldsymbol{x})$ maps a feature vector $\boldsymbol{x}$ (of dimension d) to an estimate $\hat{y}$ of its corresponding label y. If the number of possible labels is two, we say the classifier is a *binary classifier*. In the binary case, we will generally use the set of labels $\{-1, 1\}$. The training data with label $+1$ are called *positive examples*, while those with label -1 are called *negative examples*. Similarly, M-ary classifiers work with M possible labels.

We will present the k-Nearest Neighbors, Linear Discriminant Analysis (LDA) and Naive Bayes classifiers for M-ary classification, and Logistic Regression and the Support Vector Machine for binary classification. The LDA and Naive Bayes classifiers are intimately connected with hypothesis testing, which is covered in ECE 313. The Nearest Neighbor and Linear Support Vector Machine algorithms are often the first classifiers used for a problem, since the Nearest Neighbor classifier is extremely simple to use and the Linear Support Vector Machine often gives very good performance with low training and classification computational requirements. Logistic regression is often competitive with the SVM, and Naive Bayes is extremely simple and works well on a wide range of problems.

2.1 A Note on Inputs to Learning Algorithms

We make a quick note on features, which is applicable to all machine learning algorithms. Always read the documentation when using an implementation of a learning algorithm – it often contains tips and tricks for getting good performance and examples.

The pre-processing steps on your feature vectors can make a vast difference in algorithm performance, with the correct pre-processing steps depending on the particular library used to implement the algorithm. For example, if one has a categorical feature which does not have the notion of ordering, such as one taking on K values, e.g., {UIUC,UMich,Purdue}, some algorithms work better if one uses a *one-hot encoding* (a.k.a. *1-of-K encoding*) [3], in which we replace the feature with K binary features, one for each different value it can take. In the example with features taking values in {UIUC,UMich,Purdue} we would replace this feature with 3 new binary features: "Is the feature UIUC or not?", "Is the feature UMich or not?" and "Is the feature Purdue or not?". If we used

the values $\{1, 2, 3\}$ for the feature instead, we may artificially give a notion of "small" and "large" values to the feature, which do not actually exist. The one-hot encoding tends to be useful for training SVMs for LibSVM [7] or applying linear regression. Depending on the application, other algorithms may work better with the original feature. Other algorithms may assume the features have been *centered* to have mean zero, *standardized* to have variance one or *scaled* to taken on values in a range, say, between 0 and 1.

In the notes, we will make note of what preprocessing is necessary for our presentation of the algorithm, but in practice, one must read the documentation of the library used in order to determine what preprocessing is necessary and experiment for what gives the best results. For example, LibSVM [7] and LibLinear [15] which are libraries commonly used for SVMs or logistic regression work best if the features are scaled between -1 and $+1$ or 0 and 1, for numerical stability and not exaggerating importance of features based on their range. The choice and preprocessing of features is highly dependent on the application at hand and the algorithms (and their implementations) used. Many libraries have tools to automatically do the necessary preprocessing. You will cover some pre-processing techniques in the context of particular applications in the labs, as well as in other parts of the course. As stated in the introduction, having the right features can help a lot. Some common preprocessing steps are implemented in Scikit-Learn in sklearn.preprocessing [38], including the transformations mentioned thus far among others such as outlier removal techniques and techniques for handling missing values (*imputation*).

2.2 Measuring Classifier Performance

In order to decide how good a classifier is, we need to have a loss function $L(\hat{y}, y) = L(f(\boldsymbol{x}), y)$, which tells us the cost we incur when our classifier estimates the label of a datapoint $\boldsymbol{x}$ by $\hat{y} = f(\boldsymbol{x})$, when the true label is y. In designing classifiers, a natural loss function is the *0,1-loss*, which is

$$L(\hat{y}, y) = \mathbb{1}_{\{\hat{y} \neq y\}} = \begin{cases} 1 & \text{if } \hat{y} \neq y \\ 0 & \text{if } \hat{y} = y \end{cases} \tag{2.1}$$

The $0, 1$-loss is 1 if we make an error in choosing our label, and 0 if we are correct. Unless otherwise noted, L will be taken to be the 0,1-loss.

The *training error* for a classifier f is the average loss over the training set $\mathcal{T} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$:

$$\mathrm{Err}_{\mathrm{train}}(\mathcal{T}) \triangleq \frac{1}{N} \sum_{i=1}^{N} L(f(\boldsymbol{x_i}), y_i). \tag{2.2}$$

For the 0,1-Loss, this is

$$\mathrm{Err}_{\mathrm{train}} = \frac{\#\text{ inputs in } \mathcal{T} \text{ labelled incorrectly by } f}{N}. \tag{2.3}$$

By making more complex classifiers, we can drive the training error to zero. For example, for 0,1-loss, take the binary classifier

$$f(\tilde{\boldsymbol{x}}) = \begin{cases} \tilde{y} & \text{if } (\tilde{\boldsymbol{x}}, \tilde{y}) \text{ is in the training set,} \\ -1 & \text{otherwise.} \end{cases}$$

This classifier clearly has $\text{Err}_{\text{train}} = 0$, but is useless since it just says label -1 for anything it has not been trained on. This problem is related to overfitting: having the classifier tuned to characteristics specific to the training set, rather than to the joint distribution of the feature vectors and labels $p(\boldsymbol{x}, y)$. Thus, training error is not an effective way to determine if a classifier is good. In the next chapter, we will see that by using a separate data set, known as the *validation set*, to help select the classifier, we can reduce overfitting.

The correct way to measure error is the *prediction error*. Remember that a classifier f is developed from a particular set of training data (we take our training set, feed it through an algorithm and out pops a classifier f). So the classifier's output $\hat{y}$ is a function of both the test datapoint $\boldsymbol{x}$ and the training data $\mathcal{T}$. How well a classifier performs in reality should be measured by the average loss of the classifier over all possible input output pairs $(\boldsymbol{x}, y)$:

$$\text{Err}_{\text{pred}} \triangleq \text{E}\left[L(f(\boldsymbol{X}), Y) | f \text{ is trained on } \mathcal{T}\right] \tag{2.4}$$

The expectation here is over the (unknown) joint distribution $p(\boldsymbol{x}, y)$ of $(\boldsymbol{X}, Y)$. The prediction error Err_{pred} tells us on average how the classifier will perform on unseen data. Since we do not have access to $p(\boldsymbol{x}, y)$, we have to estimate the prediction error by running the trained classifier on some data that was not part of $\mathcal{T}$. This is the problem of *model selection and assessment*, and is the focus of Chapter 3.

Given another data set $\mathcal{V} = \{(\tilde{\boldsymbol{x}}_i, \tilde{y}_i)\}_{i=1}^{V}$, which is independent from the training data set, which we call a *validation* data set, and a classifier f trained on training set $\mathcal{T}$, we can estimate the prediction error as

$$\widehat{\text{Err}}_{\text{pred}}(\mathcal{V}) = \frac{1}{V} \sum_{(\tilde{\boldsymbol{x}}_i, \tilde{y}_i) \in \mathcal{V}} L(f(\tilde{\boldsymbol{x}}_i, \tilde{y}_i)) = \text{Err}_{\text{val}}(\mathcal{V}). \tag{2.5}$$

The estimate of the prediction error $\widehat{\text{Err}}_{\text{pred}}$ is also called the *validation* error Err_{val}.

Note that this estimate is implicitly a function of $\mathcal{T}$ since $f(\boldsymbol{x})$ depends on $\mathcal{T}$. For the 0,1-Loss, this is

$$\widehat{\text{Err}}_{\text{pred}}(\mathcal{V}) = \text{Err}_{\text{val}}(\mathcal{V}) = \frac{\#\text{ of inputs in } \mathcal{V} \text{ classified incorrectly by } f}{V}. \tag{2.6}$$

Our goal for designing a good classifier is to take our given training set $\mathcal{T}$, and choose a classifier that minimizes the validation (or estimated prediction) error. The next subsection takes a more nuanced look at measuring classifier performance, by analyzing different types of errors that can occur separately. This can be important in many applications, for example detecting cancer (a missed detection can be far more dangerous than a false alarm).

2.2.1 Displaying Error Performance

Often times, the error performance of a classifier on a labelled data set (e.g., training, validation) for the 0-1 loss function is displayed with a *confusion matrix* or *contingency table*. We give a brief overview of their properties here, and defer to [16] and Sec. 5.7.2 of [36] for more details. We begin with the binary classifier case shown in Fig. 2.1.

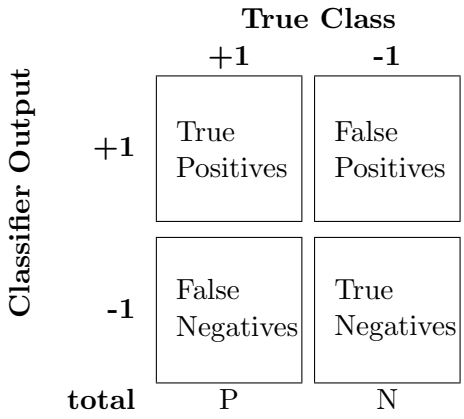


Figure 2.1: A Confusion Matrix For Binary Classification. Sometimes, the transpose of this matrix is used instead. Based on Fig. 1 in [16].

The *true positives*(TP) are the number of data from class $+1$ that the classifier outputted as $+1$. The *true negatives*(TN) are the number of data from class -1 that the classifier outputted as -1. The *false negatives*(FN) are the number of data from class $+1$ that the classifier outputted as -1. False negatives are known as Type II errors or miss(ed detections), depending on context. The *false positives*(FP) are the number of data from class -1 that the classifier outputted as $+1$. False positives are known as Type I errors or false alarms, depending on context.

The total number of datapoints with positive $(+1)$ labels in the dataset is P and the total number of datapoints with negative (-1) labels in the dataset is N. The total amount of data in the dataset is $N_{\text{tot}} = P + N$.

We define a few metrics that are commonly used. The *true positive rate* (also known as *hit rate* or *recall* or *sensitivity*) is given by

$$\text{TPR} = \frac{\text{TP}}{P}.$$

The true positive rate is the fraction of datapoints with positive labels correctly classified.

The *false positive rate* (also known as *false alarm rate*) is given by

$$\text{FPR} = \frac{\text{FP}}{N}.$$

The false positive rate is a the fraction of datapoints with negative labels incorrectly classified.

Note that the FPR and TPR are both numbers between 0 and 1. The FPR is an approximation of the false alarm probability, while the TPR is an approximation of the probability of detection (1− the probability of missed detection). A perfect classifier would achieve TPR = 1 and FPR = 0.

The *specificity* (or *true negative rate*) is

$$\text{Specificity} = 1 - \text{FPR}.$$

The *precision* (or *positive predictive value*) is given by

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}.$$

The precision is the fraction of true positives from all the positive outputs of the classifier.

The *accuracy* is the fraction of data that was correctly classified

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{N_{\text{tot}}}.$$

Note that the error measured by the 0,1-Loss is 1-Accuracy.

The harmonic mean[1] of the precision and recall is known as the *F-measure* (or *F-score* or F_1-*score*):

$$F = \frac{2(\text{Precision} \times \text{Recall})}{\text{Precision} + \text{Recall}}.$$

A perfect classifier would have $F = 1$.

The accuracy of a classifier can be a misleading predictor of its performance due to the *class imbalance problem*, where one may have far more data in some classes than others. Imbalances often occur in information retrieval problems, where one tries to classify documents as relevant to a user provided query ($+1$) or irrelevant to the query (-1). The user then receives the documents that are classified as relevant [33]. The precision is the fraction of documents that the user received that are relevant, while the recall is the fraction of relevant documents that are received by the user. In many information retrieval systems, such as search engines[2], there are many more irrelevant documents than relevant ones. In this case, a trivial classifier that simply always outputs -1 will give good accuracy. The F-measure (and its variants) do some balancing of the precision and recall to account for the imbalance, for summary of the performance of the classifier in one number. However, in many applications, reporting the confusion matrix is favorable versus a summary statistic such as the F-measure, due to the differing importance of precision and recall in different applications.

A way of summarizing classifier performance that is invariant to class imbalance is a tool from statistical signal processing known as the receiver operating characteristic (ROC) (see, e.g., [35]). The ROC is a plot of the true positive rate versus the false positive rate for a set of classifiers. Each classifier gives a particular pair (TPR, FPR). By varying parameters for the classifier (e.g. threshold for a linear classifier), one can draw curves of performance for the family of classifier determined by the parameters, known as an *ROC curve*. The line TPR=FPR is achievable by a classifier that flips a biased coin for each datapoint. Any classifier can be made to lie above the line TPR=FPR by simply inverting the output of the classifier. An important metric associated with the ROC is the *Area under an ROC curve (AUC)*, for which we defer discussion to [16]. The rough idea is that assume your classifier produces a *score*[3], which is a real value, rather than a label, with higher values for inputs that are thought to be in class $+1$ than those in -1. Comparing the output of the classifier to a threshold gives the classifier label. The AUC measures the probability that a random $+1$ datapoint will have a higher score than a random -1 datapoint. A larger AUC is better. Another common technique is plotting Precision versus Recall. The best classifiers will have Precision-Recall curves that approach $(1, 1)$.

[1] See Sec. 8.3 and Ex. 8.2 of [33] for a justification.

[2] It is more complex to measure the performance of search engines, since they output rankings rather than binary classes. Our discussion of information retrieval is for unranked information retrieval. See Ch. 8 of [33] for more details.

[3] Such a classifier is known as an *scoring classifier*. For some classifiers, there are natural ways of generating scores (e.g. if they maintain a probability internally), but there are other techniques for other classifiers. See the references in [16] for more details.

We will not discuss the non-binary case in much detail. The confusion matrix extends naturally to more than 2 classes. If there are M classes, the confusion matrix is a $M \times M$ table, where the i, jth component is the number of datapoints that were classified as class i when the true class is j. The diagonal of this matrix consists of the correctly classified datapoints. Reporting the confusion matrix provides a way to see which classes are commonly confused, and can be used to weigh different types of errors between classes differently. For example, many people write 4 and 9 similarly, so errors for digit classification (classes $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$) would occur more often between 4 and 9 than between 4 and 5. Further details and performance metrics can be found in the references cited at the beginning of this section. An example of confusion matrices and these error metrics will be given in Chapter 6, Fig. 6.4 and Fig. 6.5.

2.2.2 Computational Performance

Thus far, we have discussed the statistical performance of a classifier, i.e. how a classifier that is constructed from a training set performs on an unseen dataset. We now discuss computational performance, which is concerned with computational complexity (time and space requirements) for training and testing. Note that when computational performance is reported, it is dependent on the kind of computers and implementations of the algorithms that are used.

Generally, many classifiers are trained with a large amount of computation (both time and hardware). This is usually acceptable, since training is a one-time cost. At test time, generally less resources are available – the classification of a test datapoint often has to be done quickly without using too many computational resources (since many times, many datapoints are classified). When choosing a classifier, the statistical performance (the type of performance discussed above, e.g. with the 0,1-Loss above) must be balanced with computational requirements. An example case study for document recognition is provided in [31]. As an extreme example, the Nearest Neighbor classifier (discussed in the next section) requires basically no training resources. But, at test time, a pass over the entire training data is required. The neural network approach [31] certainly requires far more training resources than the Nearest Neighbor approach, but performs very well with far fewer computational resources (in both time and hardware)[4]. The balance between the tradeoff between computational burden (at training and/or testing) and error performance is application dependent, and is up to the practitioner to make a good decision. For example, a classifier for detecting cancer that runs instantly but has an error rate slightly better than a coin flip is obviously not as useful as a classifier that takes three days but has an error rate close to 0. On the other hand, if a classifier is designed to output how cute a cat is, on a scale of 1 to 10, an algorithm with a quick running time that misclassifies many cats may be more acceptable than one that takes a long time but classifies more cats correctly, keeping in mind that cuteness is in eyes of the beholder.

[4]One of the earlier neural networks for digit recognition had a higher error rate than the nearest neighbor classifier, but was favored due to its computational aspects [24, 31]. Later neural networks (e.g. LeNet-5) beat Nearest Neighbor on this application in both computational resources (memory, processor, etc.) and statistical performance (testing error). Further history and details are given in [31].

2.3 k-Nearest Neighbors

We are now ready to develop our first classifier algorithm – the k-Nearest Neighbors (k-NN) algorithm. The k-Nearest Neighbor (k-NN) classifier is quite simple and intuitive. Given a training set $\mathcal{T} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$, and an input feature vector $\boldsymbol{x}$ to be classified, find the k feature vectors in the training set that are closest to $\boldsymbol{x}$. Look at their labels, and choose the label that occurs the most (breaking ties arbitrarily). A common case is to choose $k = 1$, which is called the Nearest Neighbor (NN) algorithm. It is given in pseudocode in Alg. 1. A *dissimilarity measure* is a function that takes a pair of feature vectors and is larger the less similar they are (so, feature vectors that are more similar will have smaller dissimilarity).

Algorithm 1 Algorithm for k-Nearest Neighbors

function k-NearestNeighbors(test datapoint $\boldsymbol{x}$, Training Set $\mathcal{T} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$, dissimilarity measure between feature vectors $\Delta(\cdot, \cdot)$)
  for i=1,...,N **do**
    Calculate $\delta_i = \Delta(\boldsymbol{x}, \boldsymbol{x_i})$
  end for
  Find indices $\{i_1, \ldots, i_k\}$ corresponding to k smallest values of δ_i (break ties arbitrarily).
  return most frequently occurring element in $\{y_{i_1}, \ldots, y_{i_k}\}$ (break ties arbitrarily).
end function

Typically, the dissimilarity measure is of choice is the Euclidean distance (i.e. $\Delta(\boldsymbol{x}_1, \boldsymbol{x}_2) = \|\boldsymbol{x}_1 - \boldsymbol{x}_2\|$), and $k = 1$, which is the usual Nearest Neighbor algorithm given in Alg. 2. Other common choices are radial basis functions (introduced in Section 2.10)), or $1 - \cos\angle(\boldsymbol{x}_1, \boldsymbol{x}_2)$, where the *cosine similarity* $\cos\angle(\boldsymbol{x}_1, \boldsymbol{x}_2)$ is the cosine of the angle between $\boldsymbol{x}_1, \boldsymbol{x}_2$.

Algorithm 2 Algorithm for Nearest Neighbor with Euclidean distance

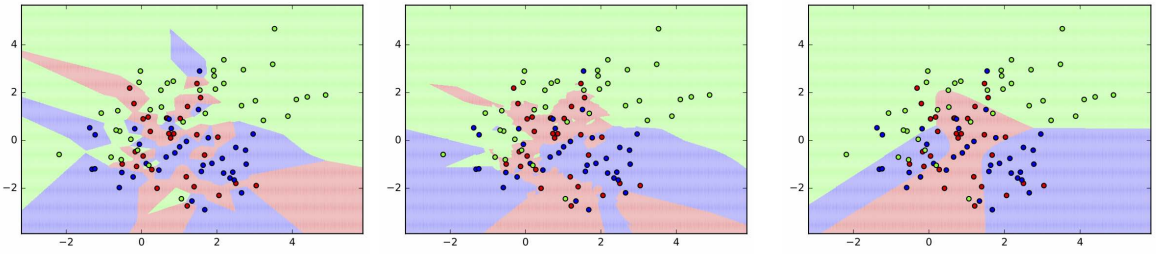
function NearestNeighbors(test datapoint $\boldsymbol{x}$, Training Set $\mathcal{T} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$)
  for i=1,...,N **do**
    Calculate $\delta_i = \|\boldsymbol{x} - \boldsymbol{x_i}\|$
  end for
  Find i^* such that δ_i is smallest (break ties arbitrarily).
  return y_{i^*}.
end function

An example where a different dissimilarity measure (tangent distance) is useful rather than the Euclidean distance is in recognizing zip codes on pieces of mail (See Section 13.3.3 in [24] or [31] for details – this can improve error rates by more than 3%, which translates to millions of pieces of mail).

The k-NN classifier is popular (particularly the NN variant in Alg. 2) since it is simple to implement and works surprisingly well for a wide variety of problems, such as identifying materials in hyperspectral images, digit recognition, EKG analysis [24] and is nearly as good as more sophisticated methods like the SVM for certain text classification problems (Table 15.2, [33]). It requires the same amount of computation regardless of the number of classes, and can divide the feature space into decision regions (i.e. places where it estimates a particular label) in a highly non-linear

manner. The NN classifier also has the nice theoretical property that if unlimited training data were available, its error would be at most twice the Bayes rate [8].

Example: In Fig. 2.2, we have training data in $\mathbb{R}^2$ from three classes (red,green,blue) drawn from different Gaussian mixture models[5], marked by circles. Fig. 2.2a shows the decision regions of a Nearest Neighbor classifier (i.e. with 1 neighbor), and Fig. 2.2b shows the decision regions of a 5-NN classifier. The NN classifier partitions the plane into regions into classes by which training datapoint is closest to a given point[6]. As the number of neighbors increases, the boundaries become a bit smoother. Note that the boundaries between classes estimated by the classifier are highly non-linear, and only imitate the training data. The optimal classifier, which is the Bayes classifier (discussed in the next section) is shown in Fig. 2.2c.



(a) Nearest Neighbor Classifier (b) 5-Nearest Neighbor Classifier (c) Bayes Classifier

Figure 2.2: Comparing the decision regions of the Bayes and Nearest Neighbor classifiers

---

The k-NN classifier does come with a few downsides, though. The naive implementation of k-NN requires calculating distances between the test datapoint and all feature vectors in the training set, for each test datapoint, which means the complexity of the algorithm grows effectively linearly in the size of the training set and requires keeping the whole training set in memory. Some trickery with data structures (such as kd-trees or ball trees) can help alleviate this, but this is still an open area of research particularly in higher dimensions due to the *curse of dimensionality*[7]. Another disadvantage of k-NN is that the classifier does not really tell us anything about the structure of the data. All it does is find what is closest in the training data and gives it to you (so the classifier is not trying to do any modeling of the training data) – this is known as *instance-based learning*.

We will see something similar in spirit in the clustering section, known as K-means clustering, which can be combined with k-NN to do classification more quickly.

[5] A Gaussian mixture model is a probability distribution that can be generated by rolling a weighted K sided die, and if i is the outcome, generate a datapoint from Gaussian distribution $\mathcal{N}(\boldsymbol{\mu}_i, \mathsf{C}_i)$, for fixed $\{(\boldsymbol{\mu}_i, \mathsf{C}_i)\}_{i=1}^K$ and fixed die weights. This model is commonly used in machine learning. Also see Sec. 2.4.3

[6] i.e. a Voronoi diagram in computational geometry parlance.

[7] Roughly speaking, many algorithms have problems in high dimensional feature spaces. See [12] for some examples.

2.4 Bayes Classifiers: ECE 313 Redux

2.4.1 Binary Classifiers

Binary hypothesis testing, studied in ECE 313, is closely related to binary classification. In binary hypothesis testing, we have observations $\boldsymbol{X}$ drawn from one of two hypotheses (labels, classes):

$$H_{-1} : \boldsymbol{X} \sim p(\boldsymbol{x}| - 1) \tag{2.7}$$

$$H_1 : \boldsymbol{X} \sim p(\boldsymbol{x}|1) \tag{2.8}$$

where $\boldsymbol{X} \sim p(\boldsymbol{x}| - 1)$ is used to denote that $\boldsymbol{X}$ has pdf/pmf $p(\boldsymbol{x}| - 1)$. The index of the hypothesis corresponds to the label y in the classification problem, and $\boldsymbol{x}$ corresponds to the feature vector. In hypothesis testing, it is assumed that the probability distributions $p(\boldsymbol{x}| - 1)$ and $p(\boldsymbol{x}|1)$ are known. Furthermore, in the Bayesian setting, we are given prior probabilities $\pi_{-1} and \pi_1$, for H_{-1} and H_1, respectively. This corresponds to specifying the joint distribution $p(\boldsymbol{x}, y)$ in the classification problem, since

$$p(\boldsymbol{x}, y) = p(\boldsymbol{x}|y)\pi_y$$

Given these probability distributions, we form the *Bayes classifier* (which is a *likelihood ratio test*, also known as the *Maximum A Posteriori (MAP) decision rule* or *Bayes Decision Rule*), which is given by:

$$\hat{y}_{\text{Bayes}} = f_{\text{Bayes}}(\boldsymbol{x}) = \arg \max_{y \in \{-1,1\}} \pi_y p(\boldsymbol{x}|y) = \arg \max_{y \in \{-1,1\}} p(\boldsymbol{x}, y) \tag{2.9}$$

The difference between the hypothesis testing problem and the classification problem (with supervised learning) is that in the latter, $p_1(\boldsymbol{x}), p_{-1}(\boldsymbol{x}), \pi_{-1}, \pi_1$ are not known, and so the function that is used to estimate labels must be inferred using only the training set $\mathcal{T}$.

The probability of error of a binary classifier under 0,1-loss is given by:

$$\mathrm{P_e} = \pi_{-1} P(\text{Say } H_1 | \boldsymbol{X} \text{ was drawn under } H_{-1}) + \pi_1 P(\text{Say } H_{-1} | \boldsymbol{X} \text{ was drawn under } H_1)$$

The Bayes classifier in (2.9) minimizes $\mathrm{P_e}$. Note that $\mathrm{P_e}$ is the same as the prediction error $\mathrm{Err}_{\text{pred}}$ defined in (2.4), except that the classifier is not constructed using training data but rather using (2.9). No algorithm can do better than the Bayes classifier in terms of prediction error (but remember that the Bayes classifier was derived under the knowledge of $p(\boldsymbol{x}, y)$ that a supervised learning algorithm does not have!) [36].

The quantity $P(\text{Say } H_1 | \boldsymbol{X} \text{ was drawn under } H_{-1})$ is known as the *probability of false alarm* (*Type I error*), while $P(\text{Say } H_{-1} | \boldsymbol{X} \text{ was drawn under } H_1)$ is known as the *probability of missed detection* (*Type II error* or *probability of miss*).

2.4.2 M-ary classifiers

The M-ary Bayes classifier (for any $M \geq 2$) can be constructed in essentially the same way as the binary Bayes classifier. In the M-ary classification problem, we have M hypotheses corresponding to the M labels.

$$H_y : \boldsymbol{X} \sim p(\boldsymbol{x}|y), \qquad y = 1, \ldots, M$$

We will assume we have priors $\{\pi_y\}_{y=1}^M$ on the labels so that the joint distribution of the feature vector and label is given by $p(\boldsymbol{x}, y) = p(\boldsymbol{x}|y)\pi_y$, as in the binary case.

The Bayes classifier in the M-ary case

$$\begin{aligned}\hat{y}_{\text{Bayes}} = f_{\text{Bayes}}(\boldsymbol{x}) &= \arg\max_{y\in\{1,\ldots,M\}} p(\boldsymbol{x}, y) = \arg\max_{y\in\{1,\ldots,M\}} \pi_y p(\boldsymbol{x}|y) \\ &= \arg\max_{y\in\{1,\ldots,M\}} [\ln \pi_y + \ln p(\boldsymbol{x}|y)]\end{aligned} \tag{2.10}$$

How is the Bayes classifier useful for machine learning? The answer is that classifiers can be derived by writing down a model of $p(\boldsymbol{x}, y)$ and using estimates of π_y and $p(\boldsymbol{x}|y)$ from the data in (2.10) to get a useful classifier. We will see two shortly: Gaussian Discriminant Analysis and Naive Bayes. This type of idea runs through a wide variety of inference tasks in machine learning, far beyond classification [36]. The general challenge in this approach is to infer the desired quantities (e.g. $p(y|\boldsymbol{x})$) efficiently, for a given model. Typically, the models considered are restricted, such as graphical models, which associate a particular form to the joint distribution to a graph [36]. The choice of model can give different power (e.g. an LDA classifier always has linear decision boundaries, but a Naive Bayes one can have non-linear ones).

2.4.3 The Bayes Classifier for Multivariate Gaussian Distributions

We say a vector $\boldsymbol{X} \in \mathbb{R}^d$ has a d-dimensional *(multivariate) Gaussian distribution* with mean $\boldsymbol{\mu}$ and covariance matrix C, where C is positive definite (symmetric, all positive eigenvalues), if it has the (joint) pdf

$$\phi(\boldsymbol{x}; \boldsymbol{\mu}, \mathsf{C}) = \frac{1}{(2\pi)^{d/2}\sqrt{|\mathsf{C}|}} e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^\top \mathsf{C}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})}. \tag{2.11}$$

To denote that $\boldsymbol{X}$ has the above pdf, we use the compact notation $\boldsymbol{X} \sim \mathcal{N}(\boldsymbol{\mu}, \mathsf{C})$. In ECE313, you saw the cases where $d = 1$ and $d = 2$ (check that the definition above matches what you know from ECE313!). Note that μ_ℓ, the ℓth element of $\boldsymbol{\mu}$, is equal to $\mathrm{E}[X_\ell]$, where X_ℓ is the ℓth element of $\boldsymbol{X}$. The covariance matrix C contains all the pairwise covariances of the elements of $\boldsymbol{X}$, i.e., the (ℓ, m)th element of C is given by:

$$\mathsf{C}_{\ell,m} = \mathrm{Cov}(X_\ell, X_m) = \mathrm{E}[(X_\ell - \mu_\ell)(X_m - \mu_m)].$$

The diagonal entries of the covariance matrix are the variances of the features. The off-diagonal entries can be thought of as measuring how X_ℓ and X_m, with $m \neq \ell$ are statistically related to each other. This idea is basically the same for d dimensions for $d \geq 2$. An example of what a 2-dimensional Gaussian distribution looks like is given in Fig. 2.3.

Now, consider the specific M-ary hypothesis testing (classification) problem where

$$H_y : p(\boldsymbol{x}|y) = \phi(\boldsymbol{x}; \boldsymbol{\mu}_y, \mathsf{C}_y)$$

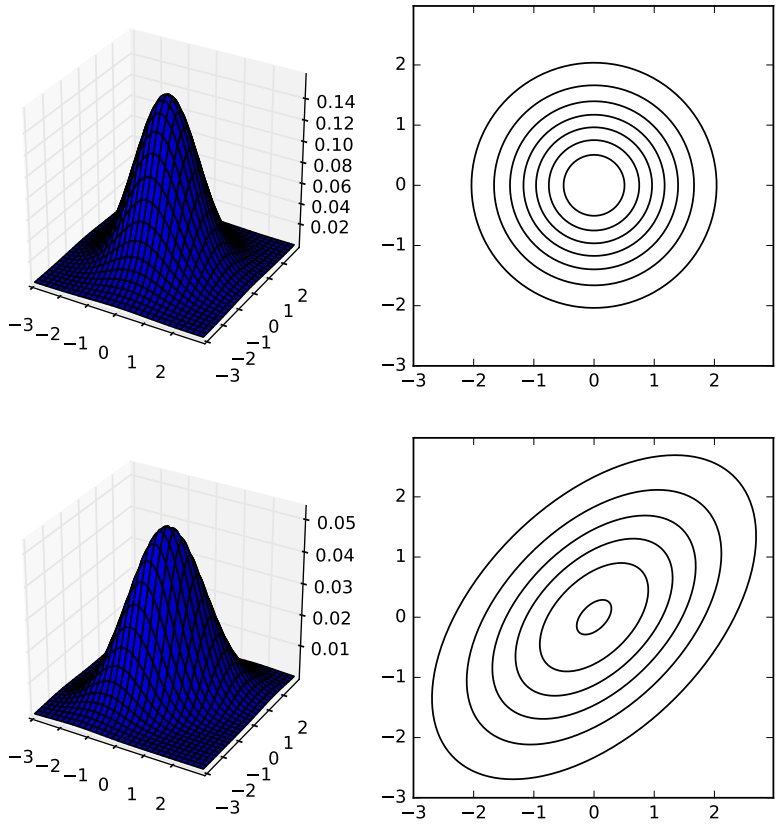


Figure 2.3: Top Row: A Gaussian with mean vector zero, and identity covariance matrix. Bottom row: A Gaussian with mean vector zero, and covariance matrix $\begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$. The pdf is given on the left, and its level curves on the right. The mean determines where the pdf is centered, while the covariance matrix determines the shape of the pdf via curves of constant $\boldsymbol{x}-\boldsymbol{\mu})^\top \mathsf{C}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})$ (which are ellipses).

By (2.10)

$$\begin{aligned} \hat{y}_{\text{Bayes}} = f_{\text{Bayes}}(\boldsymbol{x}) &= \arg \max_{y \in \{1,\ldots,M\}} [\ln \pi_y + \ln p(\boldsymbol{x}|y)] \\ &= \arg \max_{y \in \{1,\ldots,M\}} \left[\ln \pi_y - \frac{1}{2} \ln |\mathsf{C}_y| - \frac{1}{2} (\boldsymbol{x} - \boldsymbol{\mu}_y)^\top \mathsf{C}_y^{-1} (\boldsymbol{x} - \boldsymbol{\mu}_y) \right] \end{aligned} \tag{2.12}$$

where we eliminate some constants common to all terms in the optimization.

We can rewrite (2.12) as

$$\hat{y}_{\text{Bayes}} = f_{\text{Bayes}}(\boldsymbol{x}) = \arg \max_{y \in \{1,\ldots,M\}} \delta_y(\boldsymbol{x}) \tag{2.13}$$

with δ_y being the objective function (thing being maximized) in (2.12)

$$\delta_y(\boldsymbol{x}) = \ln \pi_y - \frac{1}{2} \ln |\mathsf{C}_y| - \frac{1}{2} (\boldsymbol{x} - \boldsymbol{\mu}_y)^\top \mathsf{C}_y^{-1} (\boldsymbol{x} - \boldsymbol{\mu}_y) \tag{2.14}$$

There are a couple of things to note about Bayes classifier for multivariate Gaussian inputs:

1. If the C_y's are different, the objective function $\delta_y(\boldsymbol{x})$ in (2.14) is *quadratic* in $\boldsymbol{x}$.
2. If $\mathsf{C}_y = \mathsf{C}$ for all y, (i.e. all the Gaussians have a common covariance matrix), the objective function in (2.13) can be written as a *linear function* in $\boldsymbol{x}$, by eliminating the quadratic term $\boldsymbol{x}^\top \mathsf{C}^{-1}\boldsymbol{x}$ and $\ln|\mathsf{C}|$ that is common to all terms. In particular, we can re-write (2.13) as

$$\hat{y}_{\text{Bayes}} = f_{\text{Bayes}}(\boldsymbol{x}) = \arg\max_{y\in\{1,\ldots,M\}} \delta_y(\boldsymbol{x}) \tag{2.15}$$

where $\delta_y(\boldsymbol{x})$ is the linear objective function:

$$\delta_y(\boldsymbol{x}) = \ln \pi_y + \boldsymbol{\mu}_y^\top \mathsf{C}^{-1}\boldsymbol{x} - \frac{1}{2}\boldsymbol{\mu}_y^\top \mathsf{C}^{-1}\boldsymbol{\mu}_y \tag{2.16}$$

Example: Consider the case where $M = 3$ and $d = 2$, with $\pi_1 = \pi_2 = \pi_3 = 1/3$, $\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$, $\boldsymbol{\mu}_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$, $\boldsymbol{\mu}_2 = \begin{bmatrix} 2 \\ -1 \end{bmatrix}$, $\boldsymbol{\mu}_3 = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$, and $\mathsf{C} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$. Then

$$\begin{aligned}
\hat{y}_{\text{Bayes}} &= \arg\max_{y\in\{1,2,3\}} \left[\ln(1/3) + \boldsymbol{\mu}_y^\top \boldsymbol{x} - \frac{1}{2}\boldsymbol{\mu}_y^\top \boldsymbol{\mu}_y\right] \\
&= \arg\max_{y\in\{1,2,3\}} \left[\ln(1/3) + \boldsymbol{\mu}_y^\top \boldsymbol{x} - \frac{5}{2}\right] \\
&= \arg\max_{y\in\{1,2,3\}} \boldsymbol{\mu}_y^\top \boldsymbol{x}
\end{aligned}$$

Now suppose $\boldsymbol{x} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$, then $\boldsymbol{\mu}_1^\top \boldsymbol{x} = 3$, and $\boldsymbol{\mu}_2^\top \boldsymbol{x} = \boldsymbol{\mu}_3^\top \boldsymbol{x} = 1$. Therefore the value of y that maximizes $\boldsymbol{\mu}_y^\top \boldsymbol{x}$ equals 1, which means that $\hat{y}_{\text{Bayes}} = 1$.

2.5 Linear Discriminant Analysis

In *Gaussian Discriminant Analysis* (GDA, also known as *Quadratic Discriminant Analysis* (QDA)), we assume that the feature vectors are modeled[8] as being distributed as a multivariate Gaussian distribution under each class, with different unknown means and/or covariance matrices[9]. Assume we have classes $\{1,\ldots,M\}$ and under class ℓ, the feature vectors are drawn from a $\mathcal{N}(\boldsymbol{\mu}_\ell, \mathsf{C}_\ell)$ distribution. We design a classifier by estimating the model parameters (prior probabilities, means and covariance matrices) from the training data and then using the Bayes classifier (see (2.13) and (2.14)) with these estimates plugged in place of the true values. The problem simplifies significantly

[8]The model used is not necessarily the actual model which the data came from, but a model in which we can perform inference tasks. This is often done in engineering tasks for analytical convenience.

[9]At least one of these parameters is different under each class, else the classes would be indistinguishable.

when it is assumed that the means are different, and the covariance matrices are the same, in which case the Bayes classifier reduces to (2.15) and (2.16). The result in this case is known as *linear discriminant analysis* (LDA).

LDA has been successfully applied on many problems – for an extensive overview, see the STATLOG project [30]. The STATLOG project took 12 data sets from diverse fields such as image processing, medicine and finance, and tried a variety of machine learning algorithms, including k-NN and LDA and found that LDA did quite well. The most likely reason why LDA tends to work well is because data often has approximately linear or quadratic decision boundaries, and the estimators used in LDA tend to be stable (even though the data is usually not Gaussian in each class). Two good references on LDA and GDA are [24, 36].

2.5.1 Estimating the Unknown Parameters

We estimate the parameters $\{(\boldsymbol{\mu}_\ell, \pi_\ell)\}_{\ell=1}^M$ and C from the training data $\mathcal{T} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$, by partitioning the training data by its label.

For each class ℓ, we can estimate the corresponding parameters:

$$\hat{\pi}_\ell = \frac{N_\ell}{N} \tag{2.17}$$

$$\hat{\boldsymbol{\mu}}_\ell = \frac{\sum\limits_{i:y_i=\ell} \boldsymbol{x}_i}{N_\ell} \tag{2.18}$$

where N_ℓ is the number of training datapoints with class ℓ.

Since we have a common covariance matrix for all the data, we can estimate the covariance matrix as:

$$\hat{\mathsf{C}} = \frac{\sum\limits_{\ell=1}^{M} \sum\limits_{i:y_i=\ell} (\boldsymbol{x}_i - \hat{\boldsymbol{\mu}}_\ell)(\boldsymbol{x}_i - \hat{\boldsymbol{\mu}}_\ell)^\top}{N - M} = \frac{\sum\limits_{i=1}^{N} (\boldsymbol{x}_i - \hat{\boldsymbol{\mu}}_{y_i})(\boldsymbol{x}_i - \hat{\boldsymbol{\mu}}_{y_i})^\top}{N - M}. \tag{2.19}$$

Basically, to estimate the prior probabilities and means, we partition the training data by their labels and use the empirical distributions to estimate them. The estimated prior is simply the fraction of datapoints from each class, while the estimated means are the means of the data from each class. Since the covariance matrix is assumed to be common to all hypotheses, we can use the mean estimate in each class and the datapoints in each class to estimate the covariance matrix. One can think of the covariance matrix estimate as first estimating the covariance in each class (which is assumed to be the same), and then averaging them to improve the estimate (proportionally, to account for the possibly different number of datapoints from each class)[10]. By substituting these estimates of unknown parameters in for the true parameters that we assumed we had, we have the Linear Discriminant Analysis algorithm, for which:

$$\hat{y}_{\text{LDA}} = \arg \max_{\ell \in \{1,\ldots,M\}} \delta_\ell(\boldsymbol{x}) \tag{2.20}$$

[10]This would give you the same result as (2.19), but with a N in the denominator rather than a $N - M$. The $N - M$ is so that on average, the estimate of the covariance matrix is the true covariance matrix. Such an estimator is said to be *unbiased*.

with

$$\delta_\ell(\boldsymbol{x}) = \left[\ln \hat{\pi}_\ell + \hat{\boldsymbol{\mu}}_\ell^\top \hat{\mathsf{C}}^{-1}\boldsymbol{x} - \frac{1}{2}\hat{\boldsymbol{\mu}}_\ell^\top \hat{\mathsf{C}}^{-1}\hat{\boldsymbol{\mu}}_\ell\right] \tag{2.21}$$

2.5.2 The LDA algorithm

The LDA algorithm is described in Alg. 3. The function trainLinearDiscriminantAnalysis is used to estimate the prior probabilities, means and covariance matrix from the data, under the assumption that feature vectors under each class are drawn from a Gaussian distribution with a different mean and the same covariance. Once this is done, we plug in these estimates into the Bayes classifier to classify datapoints (in classifyLinearDiscriminantAnalysis).

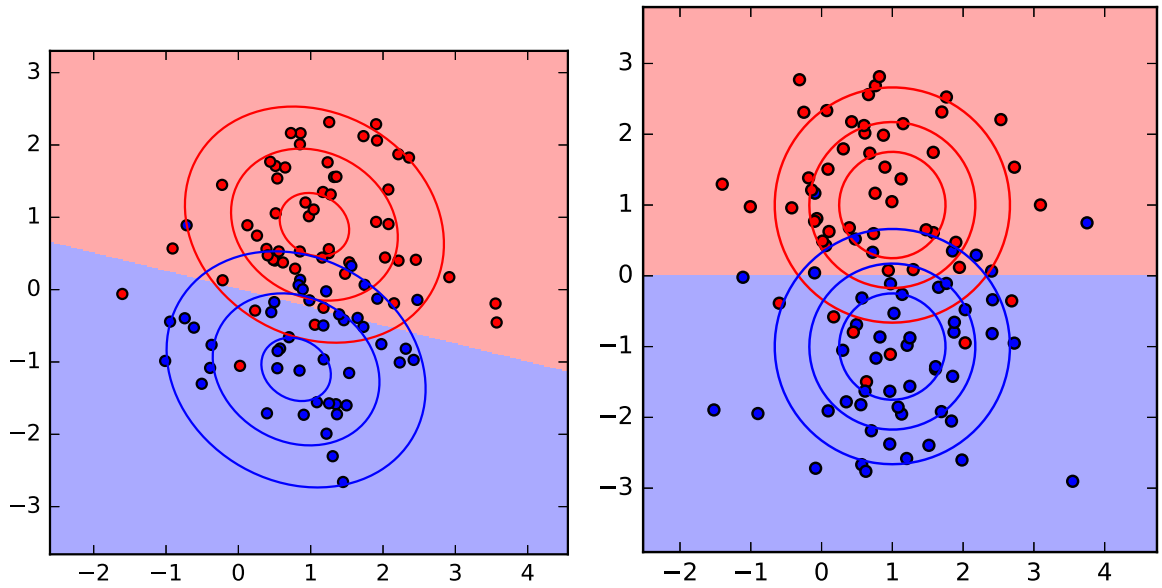
Algorithm 3 Linear Discriminant Analysis

function TRAINLINEARDISCRIMINANTANALYSIS(Training Set $\mathcal{T} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$) ▷ This is only run to make the model
  for each class $\ell = 1, \ldots, M$ **do**
    Calculate the estimate of the prior probability of class ℓ, $\hat{\pi}_\ell$ using (2.17)
    Calculate the estimate of the mean of class ℓ, $\hat{\boldsymbol{\mu}}_\ell$ using (2.18)
  end for
  Calculate the estimate of the covariance matrix (common to all classes), $\hat{\mathsf{C}}$, using (2.19).
end function
function CLASSIFYLINEARDISCRIMINANTANALYSIS(Test datapoint $\boldsymbol{x}$, Parameter estimates $\{\hat{\pi}_\ell\}_{\ell=1}^M, \{\hat{\boldsymbol{\mu}}_\ell\}_{\ell=1}^M, \hat{\mathsf{C}}$) ▷ This is run every time we want to classify a test datapoint
  for each class $\ell = 1, \ldots, M$ **do**
    Calculate the function $\delta_\ell(\boldsymbol{x}) = \ln \hat{\pi}_\ell + \hat{\boldsymbol{\mu}}_\ell^\top \hat{\mathsf{C}}^{-1}\boldsymbol{x} - \frac{1}{2}\hat{\boldsymbol{\mu}}_\ell^\top \hat{\mathsf{C}}^{-1}\hat{\boldsymbol{\mu}}_\ell$
  end for
  return the ℓ that has the largest $\delta_\ell(\boldsymbol{x})$ (ties broken arbitrarily)
end function

Example: An example application of LDA is given in Fig. 2.4. In Figure 2.4a, training data under two classes (red, blue) was used to train LDA, as in Alg. 3. The data was generated with identity covariance matrix under both classes and means $[1, 1]^\top, [1, -1]^\top$ respectively. We see that the decision boundary is linear, and there are training data datapoints on either side of the decision boundary (a binary labeled data set that is not able to be separated by a hyperplane is said to be *not linearly separable*). Thus, the classifier has a non-zero training error. The Bayes classifier is shown for comparison in Fig. 2.4b. Note that the Bayes classifier also does not have a zero-training error.

2.6 Linear Classifiers

Consider again the Bayes classifier for multivariate Gaussian inputs with $\mathsf{C}_y = \mathsf{C}$, which results in classifier given in (2.15) and (2.16). When deciding between the pair of labels ℓ and m, the classifier



(a) LDA (b) Bayes Classifier

Figure 2.4: Decision regions for the LDA and Bayes classifiers. The contours denote the level curves of the true distributions in Fig. 2.4b and of the estimated distributions in the LDA model in Fig. 2.4a. Visualization based on Fig. 4.5 in [36].

uses the rule

$$\ln \pi_\ell + \boldsymbol{\mu}_\ell^\top \mathsf{C}^{-1}\boldsymbol{x} - \frac{1}{2}\boldsymbol{\mu}_\ell^\top \mathsf{C}^{-1}\boldsymbol{\mu}_\ell \underset{m}{\overset{\ell}{\gtrless}} \ln \pi_m + \boldsymbol{\mu}_m^\top \mathsf{C}^{-1}\boldsymbol{x} - \frac{1}{2}\boldsymbol{\mu}_m^\top \mathsf{C}^{-1}\boldsymbol{\mu}_m$$

Rearranging terms on both sides we can write this as

$$\boldsymbol{w}_{\ell m}^\top \boldsymbol{x} \underset{m}{\overset{\ell}{\gtrless}} \beta_{\ell m} \tag{2.22}$$

where

$$\boldsymbol{w}_{\ell m} = \mathsf{C}^{-1}(\boldsymbol{\mu}_\ell - \boldsymbol{\mu}_m), \quad \text{and } \beta_{\ell m} = \ln \pi_m - \ln \pi_\ell + \frac{1}{2}\boldsymbol{\mu}_\ell^\top \mathsf{C}^{-1}\boldsymbol{\mu}_\ell - \frac{1}{2}\boldsymbol{\mu}_m^\top \mathsf{C}^{-1}\boldsymbol{\mu}_m \tag{2.23}$$

The classifier will prefer label ℓ or m, depending on whether $\boldsymbol{w}_{\ell m}^\top \boldsymbol{x}$ is greater than or less than $\beta_{\ell m}$. This boundary is *linear*, and we can call the function

$$g_{\ell,m}(\boldsymbol{x}) = \boldsymbol{w}_{\ell m}^\top \boldsymbol{x} - \beta_{\ell m} \tag{2.24}$$

a *linear discriminant function* between labels ℓ and m, with the understanding that the boundary between labels ℓ and m is given by $g_{\ell,m}(\boldsymbol{x}) = 0$. Note that by construction

$$g_{m,\ell}(\boldsymbol{x}) = -g_{\ell,m}(\boldsymbol{x}). \tag{2.25}$$

More generally, a linear classifier is defined to be *any* classifier for which the decision between each pair of label pair (ℓ, m) is made through a linear discriminant function of the form in (2.24),

i.e., $g_{\ell,m}(\boldsymbol{x}) = \boldsymbol{w}_{\ell m}^\top \boldsymbol{x} - \beta_{\ell m}$, for some $\boldsymbol{w}_{\ell m}$ and $\beta_{\ell m}$. Note that for a general linear classifier, the linear discriminant function $g_{\ell,m}$ is not necessarily derived from a linear objective function such as the one given in (2.16). The decision boundary for labels (ℓ, m) is given by

$$g_{\ell,m}(\boldsymbol{x}) = 0, \text{ equivalently } \boldsymbol{w}_{\ell m}^\top \boldsymbol{x} = \beta_{\ell m}. \tag{2.26}$$

Recall that the above equation is the equation for a *hyperplane* in $\mathbb{R}^d$, which is a line in $\mathbb{R}^2$, and a plane in $\mathbb{R}^3$. Here $\boldsymbol{w}_{\ell m}$ is a fixed vector normal to the hyperplane, and $\beta_{\ell m}$ is a constant that determines the offset of the hyperplane from the origin. A hyperplane partitions $\mathbb{R}^d$ into two parts: $\{\boldsymbol{x} : \boldsymbol{w}_{\ell m}^\top \boldsymbol{x} \leq \beta_{\ell m}\}$ and $\{\boldsymbol{x} : \boldsymbol{w}_{\ell m}^\top \boldsymbol{x} > \beta_{\ell m}\}$.

A *binary linear classifier* is one which divides the entire feature space into two parts (corresponding to the labels 1 and -1) by a hyperplane, i.e.,

$$\hat{y} = f(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \boldsymbol{w}^\top \boldsymbol{x} \geq \beta \\ -1 & \text{if } \boldsymbol{w}^\top \boldsymbol{x} < \beta \end{cases} \tag{2.27}$$

and is specified by the parameters specifying the hyperplane $(\boldsymbol{w}, \beta)$. A simple example of a binary linear classifier in $\mathbb{R}^2$ is given in Fig. 2.5.

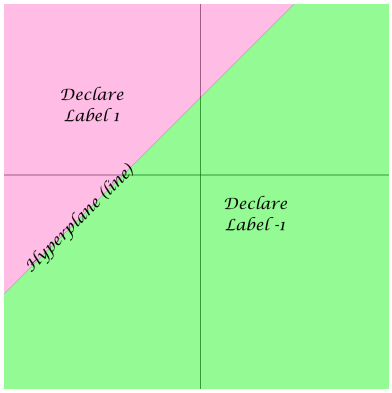


Figure 2.5: A linear classifier

Linear classifiers are quite useful in practice, especially with the addition of kernel tricks (discussed in Section 2.10) to build non-linear decision boundaries in the framework of linear classifiers. They tell us some modelling information about the data (a hyperplane that tries to separate the feature vectors based on their labels) unlike k-NN, are cheap to use on test data (take a dot product with the test datapoint and compare it to a threshold) and work well in practice (for appropriate chosen features). The ideas of linear classifiers are also useful for building *neural networks* by composing linear classifiers with appropriately chosen non-linearities, known as *activation functions*[13]. We already studied Linear Discriminant Analysis in Section 2.5. We will study two other linear classifiers in subsequent sections: Logistic Regression (which is a classification method, despite the name) in Section 2.7, and the support vector machine (SVM) in Section 2.8.

2.6.1 Building M-ary Linear Classifiers from Binary Linear Classifiers

Now suppose $M > 2$, and consider the special case where the linear discriminant functions $g_{\ell,m}$, $\ell \neq m$, are derived from a linear objective function such as the one given in (2.16) or the one in

(2.21). That is, assume that the labels are formed using

$$\hat{y} = \arg \max_{y \in \{1,\ldots,M\}} \delta_y(\boldsymbol{x}) \tag{2.28}$$

with

$$\delta_y(\boldsymbol{x}) = \boldsymbol{a}_y \boldsymbol{x} + b_y \tag{2.29}$$

for some $\{\boldsymbol{a}_y, b_y\}_{y=1,2,\ldots,M}$. Then if we set

$$g_{\ell,m}(\boldsymbol{x}) = \delta_\ell(\boldsymbol{x}) - \delta_m(\boldsymbol{x})$$

we can build the M-ary classifier from the binary classifiers between the $\binom{M}{2}$ pairs of labels as

$$\hat{y} = \ell^* \text{ if } g_{\ell^*,m}(\boldsymbol{x}) \geq 0 \text{ for all } m \neq \ell^* \tag{2.30}$$

and this $\hat{y}$ is the same as the solution to (2.28).

Example: Suppose $M = 3$, $d = 2$, with linear objective functions:

$$\delta_1(x) = x_1 + 2x_2, \quad \delta_2(x) = 2x_1 - x_2, \quad \delta_3(x) = -x_1 + 2x_2.$$

Then the linear discriminant functions derived from these objective functions are given by:

$$g_{1,2}(\boldsymbol{x}) = -x_1 + 3x_2, \quad g_{2,3}(\boldsymbol{x}) = 3x_1 - 3x_2, \quad g_{3,1}(\boldsymbol{x}) = -2x_1$$

Now suppose $\boldsymbol{x} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$. Then $g_{1,2}(\boldsymbol{x}) = 2 > 0$, $g_{2,3}(\boldsymbol{x}) = 0$, and $g_{3,1}(\boldsymbol{x}) = -2 < 0$. This means that $g_{1,3}(\boldsymbol{x}) = -g_{3,1}(\boldsymbol{x}) > 0$, which along with $g_{1,2}(\boldsymbol{x}) > 0$, implies that $\hat{y} = 1$.

Of course, if we determine the $\delta_y(\boldsymbol{x})$'s from the data as we do, for example, in LDA (see (2.21)), then we would use (2.28) to find $\hat{y}$, rather than using (2.30).

Now consider the case where we construct the individual linear discriminant functions $g_{\ell,m}$'s directly from the data for the classes ℓ and m. For example, this is how the Support Vector Machine (SVM) classifier is constructed (see Section 2.8). Then the approach described in (2.30) may not work to find $\hat{y}$ consistently, as the following example illustrates.

Example: Suppose $M = 3$, $d = 2$, and we happen to design the following linear discriminant functions using the training the data from the three classes:

$$g_{1,2}(\boldsymbol{x}) = -x_1 + 3x_2, \quad g_{2,3}(\boldsymbol{x}) = 2x_1 - x_2, \quad g_{3,1}(\boldsymbol{x}) = x_1.$$

Then the input $\boldsymbol{x} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ satisfies $g_{1,2}(\boldsymbol{x}) = 2 > 0$, $g_{2,3}(\boldsymbol{x}) = 1 > 0$, and $g_{3,1}(\boldsymbol{x}) = 1 > 0$. This means that there is no label ℓ^* that satisfies $g_{\ell^*,m}(\boldsymbol{x}) \geq 0$ for all $m \neq \ell^*$, i.e., we cannot find $\hat{y}$ using (2.30).

There are two popular heuristic approaches to constructing M-ary classifiers from binary classifiers (which work for both linear and non-linear classifiers).

One-verus-All (OVA): This also goes by the name One-versus-Other. Here we build M different binary classifiers. For the ℓ-th classifier we take the positive datapoints to be all the points in class ℓ, and the negative datapoints to be all the points not in ℓ. If we denote the discriminant function ℓ-th classifier by g_ℓ, then we classify an input $\boldsymbol{x}$ as:

$$\hat{y} = f(\boldsymbol{x}) = \arg\max_{\ell} g_\ell(\boldsymbol{x}) \tag{2.31}$$

One-versus-One (OVO): This also goes by the name All-versus-All. Here we build $\binom{M}{2}$ different binary classifiers, one for each pair of classes. Let the discriminant function for class pair (ℓ, m) be denoted by $g_{\ell,m}$, with $g_{m,\ell} = -g_{\ell,m}$. Then we classify an input $\boldsymbol{x}$ as:

$$\hat{y} = f(\boldsymbol{x}) = \arg\max_{\ell} \sum_{m \neq \ell} g_{\ell,m}(\boldsymbol{x}). \tag{2.32}$$

Example: Continuing with the previous example with $M = 3$, $d = 2$, and linear discriminant functions :

$$g_{1,2}(\boldsymbol{x}) = -x_1 + 3x_2, \quad g_{2,3}(\boldsymbol{x}) = 2x_1 - x_2, \quad g_{3,1}(\boldsymbol{x}) = x_1$$

for input $\boldsymbol{x} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ we compute:

$$\sum_{m \neq 1} g_{1,m}(\boldsymbol{x}) = g_{1,2}(\boldsymbol{x}) + g_{1,3}(\boldsymbol{x}) = g_{1,2}(\boldsymbol{x}) - g_{3,1}(\boldsymbol{x}) = 2 - 1 = 1$$

$$\sum_{m \neq 2} g_{2,m}(\boldsymbol{x}) = g_{2,1}(\boldsymbol{x}) + g_{2,3}(\boldsymbol{x}) = -g_{1,2}(\boldsymbol{x}) + g_{2,3}(\boldsymbol{x}) = -2 + 1 = -1$$

$$\sum_{m \neq 3} g_{3,m}(\boldsymbol{x}) = g_{3,1}(\boldsymbol{x}) + g_{3,2}(\boldsymbol{x}) = g_{3,1}(\boldsymbol{x}) - g_{2,3}(\boldsymbol{x}) = 1 - 1 = 0$$

Therefore using the OVO method:

$$\hat{y} = f(\boldsymbol{x}) = \arg\max_{\ell} \sum_{m \neq \ell} g_{\ell,m}(\boldsymbol{x}) = 1.$$

2.7 Logistic Regression

Logistic regression is one of the most common classification approaches, especially in application areas like biostatistics and social sciences [24]. We will consider the binary case. The ideas extend easily to the M-ary case, and can even be extended to cases like ordinals (i.e. classes which are ordered, such as how good something is) and beyond. It is also useful for determining which features are statistically significant[11] in a binary classification problem as well [24]. One can extend the results here to get non-linear classification boundaries with a kernel trick (see Section 2.10).

[11] Note that features which are statistically significant may be unimportant, and those which are not statistically significant may be important.

Logistic regression is a discriminative model (see Section 2.11 for a discussion), and directly attempts to model $p(y|\boldsymbol{x})$, the posterior probability of the class, rather than jointly modelling the distribution of the features and labels $p(\boldsymbol{x}, y)$ and then calculating $p(y|\boldsymbol{x})$ as in a generative model (such as all the classifiers we have discussed thus far). The model of posterior probabilities of the classes can be useful for a variety of tasks, such as measuring risk or loss, combining multiple models to form a larger model, and allow for rejection of classification (if an input cannot be classified with reasonable certainty to any class, declare that one is uncertain, rather than a class) [3]. As we will note in Sec. 2.11, the stability of the estimates of $p(y|\boldsymbol{x})$ from a discriminative classifier can be useful for these applications. All of the classifiers we will discuss in these notes, aside from the SVM (discussed in Section 2.8), are capable of producing estimates of $p(y|\boldsymbol{x})$.

In logistic regression, we begin with classes $+1, -1$ and model the posterior probabilities[12] $p(y|\boldsymbol{x})$ directly, without modelling $p(\boldsymbol{x}, y)$ first, as

$$p(1|\boldsymbol{x}) = \frac{e^{\beta_0 + \boldsymbol{\beta}^\top \boldsymbol{x}}}{1 + e^{\beta_0 + \boldsymbol{\beta}^\top \boldsymbol{x}}} \tag{2.33}$$

$$p(-1|\boldsymbol{x}) = \frac{1}{1 + e^{\beta_0 + \boldsymbol{\beta}^\top \boldsymbol{x}}} \tag{2.34}$$

The function $\frac{e^t}{e^t+1}$ is known as the *logistic function*. The logistic regression model gives us a model for the posterior probabilities $p(y|x)$ based on a linear model $\beta_0 + \boldsymbol{\beta}^\top \boldsymbol{x}$. We fit $\beta_0 + \boldsymbol{\beta}^\top \boldsymbol{x}$ by maximum likelihood. The likelihood function $h(\beta_0, \boldsymbol{\beta})$ is based on the training data $\mathcal{T} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$:

$$h(\beta_0, \boldsymbol{\beta}) = \prod_{i=1}^{N} p(y_i|\boldsymbol{x}_i). \tag{2.35}$$

Unlike the examples seen in ECE313 or thus far in these notes, maximizing this likelihood does not have a closed form and has to be done numerically through iterative algorithms. Many machine learning packages have the ability to handle fairly large training sets and number of features efficiently.

To get a classifier from the posterior probabilities, we simply pick the class which has the larger posterior probability. Note that logistic regression is a linear classifier, since

$$\ln \frac{p(1|\boldsymbol{x})}{p(-1|\boldsymbol{x})} = \beta_0 + \boldsymbol{\beta}^\top \boldsymbol{x} \tag{2.36}$$

and the left hand side is non-negative if and only if $p(1|\boldsymbol{x}) \geq p(-1|\boldsymbol{x})$. The transformation of the (2.36) is known as a *logit* transformation.

From (2.36), we see

$$\hat{y}_{\text{logistic}} = f_{\text{logistic}}(\boldsymbol{x}) = \begin{cases} 1 & \hat{\beta}_0 + \hat{\boldsymbol{\beta}}^\top \boldsymbol{x} \geq 0 \\ -1 & \hat{\beta}_0 + \hat{\boldsymbol{\beta}}^\top \boldsymbol{x} < 0 \end{cases} \tag{2.37}$$

where $\hat{\beta}_0, \hat{\boldsymbol{\beta}}$ are determined by maximizing (2.35).

[12] Hence the name regression; we are modelling a continuous value. We use $\boldsymbol{\beta}, \beta_0$ in the linear classifier derived from this model, due to the regression roots of this problem. In a more advanced course, we could unify the presentation of logistic regression and linear regression as *generalized linear models* (see, e.g., Chapter 9 in [36]).

2.8 Support Vector Machines

We begin by describing *linear support vector machines* (also known as *support vector classifiers* (SVC)), which are binary linear classifiers. By adding the kernel trick (see Section 2.10), we get what is normally called a *support vector machine* (SVM). We will gloss over the precise mathematical formulation of SVM's and defer it to the references [36, 24, 3, 6]. Our presentation mostly follows Chapter 12 of [24].

We first begin by looking at the case where the classes are linearly separable, i.e. there exists a hyperplane that separates the two classes perfectly. The *margin* of a binary linear classifier is the set of points within distance M of the boundary, where M is the minimum distance from the decision boundary[13] to any feature vector in the training set. (Note that the margin M is not to be confused with the number of hypotheses M.) This situation is depicted in Fig. 2.6 with a red class and blue class. When the classes are linearly separable, there exist infinitely many hyperplanes that separate the two classes perfectly (we can see this by starting with one hyperplane and shifting it a little bit or tilting it a little bit). The linear support vector machine is given by the hyperplane that maximizes the margin – intuitively, this makes sense, as the feature vectors that we expect to classify incorrectly the most are those closest to the decision boundary. Therefore, setting the decision boundary as far as possible from the training data might make the model generalize well. Furthermore, if we use the kernel trick, we hope that the data in the new feature space will be linearly separable.

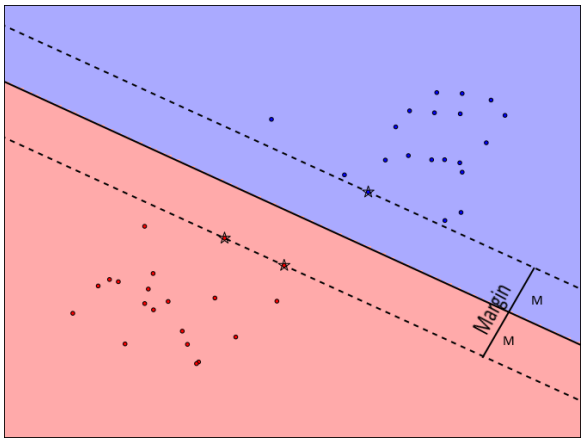


Figure 2.6: The linearly separable case for the support vector machine with training data overlayed. The support vector data points are marked with stars. The boundaries of the margin are shown in dashed lines, while the decision boundary is shown in a solid line.

The case where the data is not linearly separable is a bit trickier – no matter how we choose the hyperplane, some datapoints will be on the wrong side of the decision boundary. The basic idea is to relax what we mean by "margin". We will still consider the margin to be the set of points within distance M of the decision boundary, but we will allow training vectors to be within the margin or even in the wrong decision region. We define the correct side of the margin for a class to be the set of points which are not in the margin but are in the correct decision region for the

[13] i.e. measured in the direction perpendicular to the decision boundary

class (e.g. for the blue class, the blue region in Fig. 2.7 which is not in the margin). The wrong side of the margin for a class is the set of points not on the correct side of the margin (i.e. the set of points in the margin and the other class' decision region). Instead of requiring no training vectors to be within the margin and all training vectors to be in their correct decision region, we constrain how far training vectors on the wrong side of their margin can be from the correct side of their margin. In pictures, this means that the sum lengths of the arrows in Fig. 2.7 cannot be too large. The resulting algorithm with this relaxed definition of margin is called a *soft margin SVM*, while un-relaxed definition of margin yields a *hard margin SVM*. Unless specified otherwise, the term SVM refers to a soft margin formulation[14].

We now look at the soft margin SVM in a bit more detail. Given a decision boundary (hyperplane), we look at the set of training datapoints within distance m of the decision boundary or in the wrong decision region. For each of these training datapoints $(\boldsymbol{x}_i, y_i)$, we pay a penalty of ξ_i, where $\xi_i = \xi_i^*/m$, and ξ_i^* is the distance between $\boldsymbol{x_i}$ and the closest point in the decision region of y_i that is at least m away from the decision boundary[15]. The largest such m for a given decision boundary subject to the constraint that $\sum_i \xi_i \leq C$ is denoted M. The user-specified constant C controls how the much influence the training data on the wrong side of their margin can have, and can be used to control overfitting. The amount by which points can be on the wrong side of the margin is measured relative to the size of the margin via the scaling by m in the definition of ξ_i. Now, the linear support vector machine is given by the hyperplane that maximizes this new definition of the margin. This is shown pictorially in Fig. 2.7.

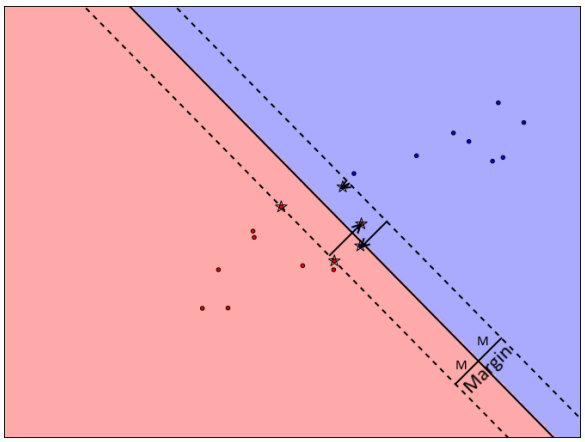


Figure 2.7: Support vector machine when the classes are not linearly separable with training data overlayed. The support vector data points are marked with stars. The boundaries of the margin are showed as dashed lines, while the decision boundary is shown as a solid line. The ξ^* quantities are marked as arrows. In general, we can even have points being outside the margin on the side of the boundary of the other class.

The name "support vector machine" comes from the fact that finding the support vector machine only depends strongly on a (hopefully small) subset of the training data near the decision boundary,

[14]Even for linearly separable data, the soft-margin SVM formulation is normally used.

[15]The scaling by m in the definition of ξ_i^* is important. Otherwise, calculating the SVM's hyperplane becomes a non-convex problem, which makes optimization difficult. See Section 12.2 of [24] for details.

called the *support vectors*[16]. In the linearly separable case, for a linear support vector machine, these are precisely the points that lie on the boundary of the decision region (see Section 14.5 in [36] for details). Both the large margin property and small number of support vectors help the SVM generalize well for many problems.

Training a SVM can be computationally expensive. The direct formulation of a SVM as a convex (quadratic) program takes cubic time in the size of the training set, while algorithms such as sequential minimal optimization take quadratic time in the size of the training set. Once a linear SVM has been trained, it is a linear classifier and therefore requires only taking dot products in order to test datapoints.

Normally, SVM's have a set of tuning parameters (e.g., for the kernel if we use the kernel trick (see Section 2.10), or the user specified constant C as above). These are often chosen via cross-validation in order to prevent overfitting and improve performance. Note that the form of the tuning parameters may be different than the ones given here depending on the software package.

Example: In Fig. 2.8, training data drawn from different Gaussian Mixture models is shown and the decision boundaries for logistic regression and a linear support vector machine are shown.

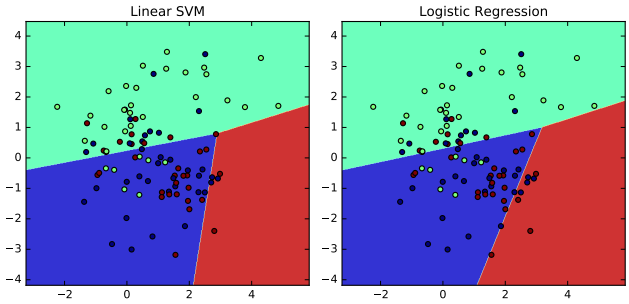


Figure 2.8: Comparison of decision boundaries on a data set using logistic regression and a linear support vector machine.

2.9 Naive Bayes Classifier

In the previous section, we saw how we could use the Bayes classifier for data that is Gaussian under each class along with empirical estimates (i.e., estimates from the data) of the mean and covariance to learn a classifier. In this section, we will show another simple way of using the Bayes classifier and empirical estimates in order to develop a classifier. Despite its simplicity, the naive Bayes classifier works very well on a variety of problems, both practically [24, 36, 33, 44, 32] and theoretically [45, 11]. For example, many of you use Naive Bayes Classifiers every day to prevent spam from getting into your inbox [34] or in information retrieval systems [33]. The Naive Bayes Classifier is given in Alg. 4. The Naive Bayes Classifier is not a linear classifier in general, but for pedagogical reasons, we are introducing it following Gaussian discriminant analysis.

[16] In mathematics, the support of a function is where a function is non-zero. In the case of a SVM, there is no dependence of the classifier on the non-support vectors.

Let us assume we have M classes, $1, \ldots, M$, and our features are d-dimensional. Recall the Bayes classifier

$$\hat{y}_{\text{Bayes}} = \arg \max_{y=1,\ldots,M} \pi_y p(\boldsymbol{x}|y) \tag{2.38}$$

The Naive Bayes Classifier is based on the assumption that the features are conditionally independent under each class. That is,

$$p(\boldsymbol{x}|y) = p(x_1|y)p(x_2|y)\ldots p(x_d|y) \tag{2.39}$$

This assumption is quite simplistic – features often do not have this conditional independence property. Nonetheless, there are some useful advantages to this formulation. The first is that fitting the model can be done for each class by looking at each feature individually (so, you only have to fit Md distributions, one for each $p(x_j|y)$, $j = 1, \ldots, d$ and $y = 1, \ldots, M$). This is much smaller than if you considered the features jointly (for example, with Gaussian Discriminant Analysis, this could be on the order of Md^2). The conditional independence assumption makes it useful for high-dimensional data (i.e. when the number of features is large), and helps reduce overfitting. The second is that you can easily mix different types of features, such as *categorical features* (ones which take on values in a finite set, such as true-false features) and *continuous features* (ones that take on values in a continuous set, such as real-valued features).

The Naive Bayes Classifier simply replaces the quantities in the Bayes Classifier (2.38) under the Naive Bayes assumption (2.39) with estimates (hatted quantities) from the training data $\mathcal{T} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$:

$$\hat{y}_{\text{NB}} = \arg \max_{y=1,\ldots,M} \hat{\pi}_y \hat{p}(x_1|y)\hat{p}(x_2|y)\ldots\hat{p}(x_d|y) \tag{2.40}$$

where $\hat{p}(x_i|y)$ is an estimate of the distribution of feature i under class y and

$$\hat{\pi}_y = \frac{N_y}{N}, \tag{2.41}$$

where N_y is the number of training datapoints in class y. The estimate of $\hat{p}(x_j|y)$ depends on your model – what type of distribution you choose for $p(x_j|y)$. This estimate is calculated on the training data, and may be calculated via maximum likelihood methods, maximum-a-posteriori methods, or non-parametric methods such as kernel density estimation (which is beyond the scope of the course), depending on the application.

Typically, (2.40) is implemented by summing the log-probabilities, rather than the multiplying probabilities in (2.40)

$$\hat{y}_{\text{NB}} = \arg \max_{y=1,\ldots,M} \ln \hat{\pi}_y + \sum_{j=1}^{d} \ln \hat{p}(x_j|y). \tag{2.42}$$

In the naive implementation, (2.40) involves multiplying possibly a large number of probabilities, which can cause underflow issues. Note that (2.40) and (2.42) are equivalent, because the logarithm is a monotone increasing function, and probabilities are non-negative.

Algorithm 4 Naive Bayes

function TRAINNAIVEBAYES(Training Set $\mathcal{T} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{N}$) ▷ This is only run to make the model
  for each class $y = 1, \ldots, M$ **do**
    Calculate the estimate of the prior probability of class y, $\hat{\pi}_y$ using (2.41)
    for each feature $j = 1, \ldots, d$ **do**
      Calculate the estimate of the distribution of the feature in class j, $\hat{p}(x_j|y)$ ▷ This can be done using (2.47) if the distribution is assumed to be Gaussian, or (2.51) if it is assumed to be Categorical
    end for
  end for
  return $\{\hat{\pi}_y\}_{y=1}^{M}$ and $\{\hat{p}(x_j|y)\}_{y=1,j=1}^{y=M,j=d}$
end function
function CLASSIFYNAIVEBAYES(Test datapoint $\boldsymbol{x}$, Estimates of class probabilities $\{\hat{\pi}_y\}_{y=1}^{M}$, Estimates of distribution of features in each class $\{\hat{p}(x_j|y)\}_{y=1,j=1}^{y=M,j=d}$) ▷ This is run every time we want to classify a test datapoint
  for each class $y = 1, \ldots, M$ **do**
    Calculate the function $\delta_y(\boldsymbol{x}) = \ln \hat{\pi}_y + \sum_{j=1}^{d} \ln \hat{p}(x_j|y)$
  end for
  return the y that has the largest $\delta_y(\boldsymbol{x})$ (ties broken arbitrarily)
end function

2.9.1 Choosing a Model for the Features

There are several common choices for the form of $p(x_j|y)$. Typically, the jth feature is assumed to come from the same family of distributions for all classes, but with different parameters. To make this explicit, we will write

$$p(x_j|y) = p(x_j|y; \boldsymbol{\theta}_{jy}) \tag{2.43}$$

where $p(x_j|y; \boldsymbol{\theta}_{jy})$ means that the distribution of feature j under class y is modeled a distribution which depends on the parameter vector $\boldsymbol{\theta}_{jy}$. Under this assumption, the Naive Bayes Classifier (2.42) becomes

$$\hat{y}_{\text{Naive Bayes}} = \arg \max_{y=1,\ldots,M} \ln \hat{\pi}_y + \sum_{i=1}^{d} \ln p(x_j|y; \hat{\boldsymbol{\theta}}_{jy}) \tag{2.44}$$

where $\hat{\boldsymbol{\theta}}_{jy}$ denotes an estimate of the parameter $\boldsymbol{\theta}_{jy}$ using the training data, typically by maximum likelihood (ML) methods (see Sections 2.8, 3.7 in the ECE313 notes [22]) or maximum-a-posteriori (MAP) methods (see, e.g., [36]) and $\hat{\pi}_y$ was defined above.

Recall that the ML estimate is the one that maximizes the likelihood of the data:

$$\hat{\boldsymbol{\theta}}_{jy,ML} = \arg \max_{\boldsymbol{\theta}} p(x_j|y; \boldsymbol{\theta}). \tag{2.45}$$

We will be using ML for the cases we consider below, so we will drop the subscript ML. The MAP case is covered in Section 3.5 in [36].

For continuous data, a common choice is the Gaussian distribution, where

$$p(x_j|y;\boldsymbol{\theta}_{jy}) = \frac{1}{\sqrt{2\pi\sigma_{jy}^2}} e^{-\frac{(x_j-\mu_{jy})^2}{2\sigma_{jy}^2}}. \tag{2.46}$$

So, for each feature and class, we will fit two parameters to the data: a mean and variance. Our parameter vector is $\boldsymbol{\theta}_{jy} = \begin{bmatrix}\mu_{jy}\\ \sigma_{jy}^2\end{bmatrix}$, which specifies the Gaussian distribution. This choice of distribution is related to Gaussian Discriminant Analysis (Section 2.5) with a diagonal covariance matrix assumption.

For the Gaussian distribution, it is easy to see (this is similar to LDA) that

$$\hat{\boldsymbol{\theta}}_{jy} = \begin{bmatrix}\hat{\mu}_{jy}\\ \widehat{\sigma_{jy}^2}\end{bmatrix} \tag{2.47}$$

where

$$\hat{\mu}_{jy} = \frac{\sum_{i:y_i=y} x_{ij}}{N_y} \tag{2.48}$$

and

$$\widehat{\sigma_{jy}^2} = \frac{\sum_{i:y_i=y} (x_{ij}-\hat{\mu}_{jy})^2}{N_y - 1}. \tag{2.49}$$

In other words, take the jth feature for all the data in class y and average it to find $\hat{\mu}_{jy}$, and use this estimate of the mean to calculate the variance of the jth feature for all the data in class y. These estimates tend to be simple and stable, so they tend to work well even if the data is non-Gaussian.

For categorical data, a common choice is the *Multinoulli* or *Categorical distribution*. If the feature takes on values $a_1, a_2, \ldots, a_\ell$, then we assume that it takes on the value i with probability ρ_{jy} (and the sum over all i is 1 for each y). Our parameter vector in this case is $\boldsymbol{\theta}_{jy} = \begin{bmatrix}\rho_{1,jy}\\ \rho_{2,jy}\\ \vdots\\ \rho_{\ell,jy}\end{bmatrix}$.

The maximum likelihood estimator $\hat{\boldsymbol{\theta}}_{jy} = \begin{bmatrix}\hat{\rho}_{1,jy}\\ \hat{\rho}_{2,jy}\\ \vdots\\ \hat{\rho}_{\ell,jy}\end{bmatrix}$ is quite simple; it is an *histogram of the data*:

$$\hat{\rho}_{k,jy} = \frac{\text{Number of datapoints in class } y \text{ whose } j\text{th feature is equal to } a_k}{N_y}. \tag{2.50}$$

However, the maximum likelihood estimator suffers from a problem in this case: Suppose we have a $\hat{\rho}_{k,jy}$ that takes the value 0 because there was no datapoint in the training dataset for which the jth feature is equal to a_k. This might happen because we don't have enough training data.

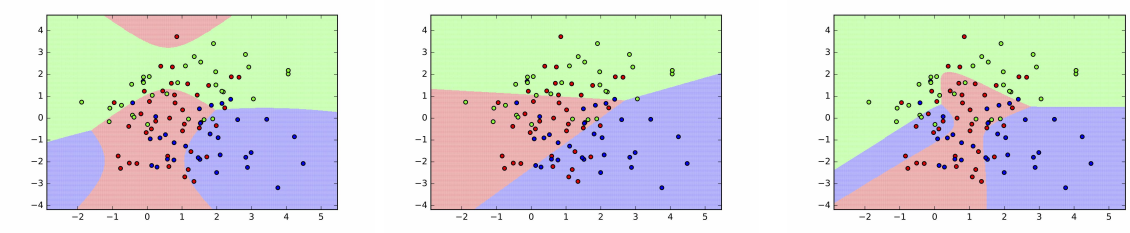
Now suppose that we when apply the classifier to a datapoint outside of the training dataset we see that the jth feature takes the value a_k, what do we do? Our Naive Bayes Classifier's objective function in (2.44) will have value $-\infty$ for class y in the optimization problem, regardless of the other features, and class y will not be chosen.

One way around this is *Laplace Smoothing* or *additive smoothing*: Pretend you have an additional α datapoints (typically, $\alpha = 1$) for each value the feature can take on, in addition to the ones you have. The smoothed estimator is

$$\hat{\rho}_{k,jy} = \frac{\alpha + \text{ Number of datapoints in class } y \text{ whose } j\text{th feature is equal to } a_k}{\alpha\ell + \text{Number of datapoints in class } y}. \tag{2.51}$$

Note that all the aforementioned estimators can be computed in one pass of the training data, and if training data arrives sequentially, the estimators can be easily updated.

Example: In Fig. 2.9, we have training data in $\mathbb{R}^2$ from three classes (red,green,blue) drawn from different Gaussian mixture models, marked by circles. A Naive Bayes Classifier was trained, with each feature modeled as a Gaussian distribution with unknown mean and variance (Fig. 2.9a). Note that the decision regions are non-linear. A linear classifier, LDA, was also trained (Fig. 2.9b). The Bayes classifier is the best we can do, and it is non-linear (Fig. 2.9c).



(a) Gaussian Naive Bayes (b) LDA (c) Bayes Classifier

Figure 2.9: Comparing the decision regions of a Gaussian Naive Bayes Classifier, Linear Discriminant Analysis and the Bayes Classifier

---

2.10 Kernel Tricks

Kernel tricks are a computational trick to adapt linear classification algorithms to give non-linear decision boundaries.

In order to develop kernel tricks, we will need the following key intuition: given a set of features, we can take a non-linear mapping $\boldsymbol{\phi}$ of them into a different (typically bigger) feature space such that in the new feature space, the classes can be linearly separated. When classes can be linearly separated, we can use a linear classifier to get good performance.

Example: Consider a data set formed by sampling points on the curves $x_2 = yx_1^2$ for $y = 1, 2$ (giving feature vectors $\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ in $\mathbb{R}^2$ and labels $y = 1, 2$). In the plane, these points are clearly not linearly separable. However, if we use the mapping of features $\boldsymbol{\phi}\left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}\right) = \begin{bmatrix} x_1^2 \\ x_2 \end{bmatrix}$ (which is non-linear), the data with label y will lie along a line in the (x_1^2, x_2) plane (which is our new feature space) through the origin with slope y. Placing the blue line between the two lines formed by the data shows that the classes can be linearly separated in the new feature space. In the original feature space, the (x_1, x_2) plane, this gives a non-linear boundary separating the classes as the blue parabola. This is shown pictorially in Fig. 2.10.

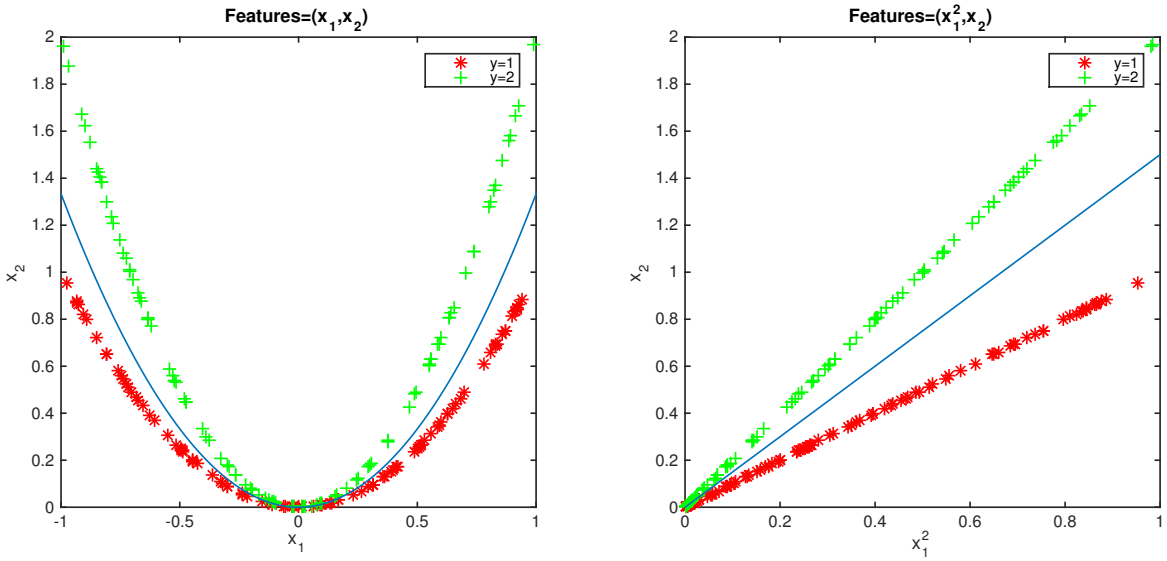


Figure 2.10: Example of a non-linear mapping that can make data linearly separable.

In general, we may not be able to identify a new feature space such that we can linearly separate the data in an explicit or useful manner. Also, if we explicitly were to compute the mapping, we would have to store $\boldsymbol{\phi}(\boldsymbol{x})$ for each $\boldsymbol{x}$ in the training set, which would increase storage and computations needed (since the vector $\boldsymbol{\phi}(\boldsymbol{x})$ typically has more components than $\boldsymbol{x}$).

In Euclidean space ($\mathbb{R}^d$), we measure the distance $\Delta(\boldsymbol{x}, \boldsymbol{r})$ between two vectors $\boldsymbol{x}, \boldsymbol{r}$ through the norm of the difference:

$$\Delta(\boldsymbol{x}, \boldsymbol{r}) = \|\boldsymbol{x} - \boldsymbol{r}\|. \tag{2.52}$$

The Euclidean norm is a special norm, because it comes from an inner (dot) product:

$$\|\boldsymbol{x}\| \triangleq \sqrt{\boldsymbol{x}^\top \boldsymbol{x}}. \tag{2.53}$$

Many linear classifiers can have their training algorithms written in terms of distances between training vectors (and thus, dot products between training vectors). Furthermore, as we saw in (2.27), the boundary between classes in a linear classifier can be written in terms of an inner product between the feature vector and the classifier coefficients $\boldsymbol{w}$.

We define the *kernel function* corresponding to the mapping $\boldsymbol{\phi}$ as

$$\kappa(\boldsymbol{x}, \boldsymbol{r}) \triangleq \boldsymbol{\phi}(\boldsymbol{x})^\top \boldsymbol{\phi}(\boldsymbol{r}). \tag{2.54}$$

The kernel function takes the dot product of two vectors after they have been mapped into the new space. A *kernel trick* uses the following idea: Assume we have a way of training a model that only depends on dot products. Then, replacing the dot products with the kernel function trains a linear classifier in the bigger feature space given by $\boldsymbol{\phi}$. This will give a non-linear classifier in the original feature space.

Thus, to get the kernel version of a model (i.e. apply a kernel trick) has two simple steps:

1. Write all the computations involved with the model so that they only involve dot products between pairs of feature vectors
2. Replace $\boldsymbol{x}^\top \boldsymbol{r}$ with $\kappa(\boldsymbol{x}, \boldsymbol{r})$

This avoids the computational complexity of implementing the mapping $\boldsymbol{\phi}$ explicitly (if it is even possible) – in fact, we do not need to know anything about $\boldsymbol{\phi}$, as everything only depends on the kernel function (which implicitly determines $\boldsymbol{\phi}$). A method that uses a kernel is known as a *kernel method*. There exist kernel-ized versions of many topics we will cover in this course – SVM, Logistic Regression, PCA, etc.

While not strictly required[17], in order for the motivation of the kernel trick to hold, the kernel function needs to act like its doing a dot product in a different space. One such family of kernels is the set of *Mercer kernels*. A Mercer kernel is a kernel $\kappa(\boldsymbol{x}, \boldsymbol{r})$ that satisfies the following three properties:

1. $\kappa(\cdot, \cdot)$ is continuous.
2. $\kappa(\cdot, \cdot)$ is symmetric: $\kappa(\boldsymbol{x}, \boldsymbol{r}) = \kappa(\boldsymbol{r}, \boldsymbol{x})$.
3. $\kappa(\cdot, \cdot)$ is positive semi-definite: Choose an arbitrary set of n vectors $\{\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}\}$. The $n \times n$ matrix formed by having $\kappa(\boldsymbol{x_i}, \boldsymbol{x_j})$ in the i, jth coordinate is positive semidefinite (i.e. has no negative eigenvalues).

By Mercer's theorem, one can actually find a characterization of $\boldsymbol{\phi}$ for Mercer kernels[10]. However, from a practical perspective, all you need is the kernel function. Proving something is a Mercer kernel is often non-trivial, but one can use several rules for building Mercer kernels as outlined in Section 6.2 of [3].

We list some of the most common kernels:

[17] If you plugged in any real-valued function with two arguments from the feature space as a kernel, you would get a model, but this model may not be useful for classification.

1. Linear kernel: $\kappa(\boldsymbol{x}, \boldsymbol{r}) = \boldsymbol{x}^\top \boldsymbol{r}$. In this case, $\boldsymbol{\phi}$ is the identity function, and the new feature space is the same as the old feature space. It is often a good choice as a starting point, and often works well when the feature space is high dimensional.

2. Radial basis function (RBF): $\kappa(\boldsymbol{x}, \boldsymbol{r}) = h(\|\boldsymbol{x} - \boldsymbol{r}\|)$. If h is completely monotonic on $[0, \infty)$, then the resultant kernel is Mercer. A common choice for a RBF is $e^{-\gamma\|\boldsymbol{x}-\boldsymbol{r}\|^2}$ for some $\gamma > 0$, which is known as a *Gaussian RBF*.

3. Polynomial Kernel: $\kappa(\boldsymbol{x}, \boldsymbol{r}) = (1 + \boldsymbol{x}^\top \boldsymbol{r})^\ell$ for some $\ell \in \mathbb{N}$. This corresponds to a case where the new feature space can be explicitly identified as having all monomials in the features up to degree ℓ.

4. Sigmoid Kernel (not Mercer): $\kappa(\boldsymbol{x}, \boldsymbol{r}) = \tanh(\gamma \boldsymbol{x}^\top \boldsymbol{r} + c)$

All of these kernels are implemented in scikit-learn [38].

When the model being trained is a linear classifier, one can use a kernel trick to get non-linear decision boundaries in the original feature space.

Example: In Fig. 2.11, training data drawn from different Gaussian Mixture models is shown, and several kernels (Sec. 2.10) are applied using a Support Vector Machine. We see that different non-linear boundaries can be had by different choices of kernel.

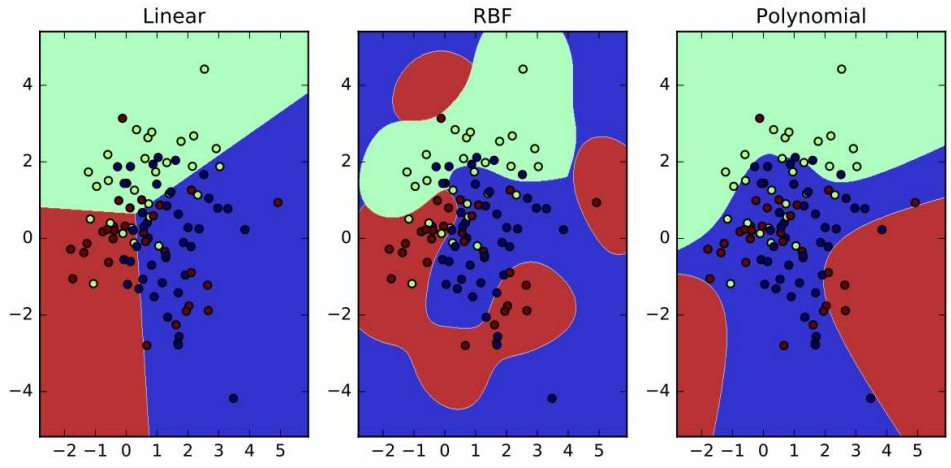


Figure 2.11: Comparison of decision boundaries on a data set using different kernels (linear, Gaussian RBF, Cubic Polynomial) with a Support Vector Machine (see Section 2.8).

While this section is a bit mathematical, there are two takeaways:

1. Non-linearly mapping the features to a bigger feature space can make them more linearly separable.

2. One can take advantage of the previous point to design non-linear classifiers from linear classifiers using kernel functions in place of dot products.

A word of caution: carelessly enlarging the feature space can lead to overfitting due to insufficient training data, and the techniques from Chapter 3 (e.g., cross-validation) should be used to make sure a given model generalizes well.

2.11 Generative versus Discriminative Models

A model which tries to directly model $p(y|\boldsymbol{x})$ (the distribution of classes given the features) is called a *discriminative model*, while a model which tries to model $p(\boldsymbol{x}, y)$ (the joint distribution of the features and labels) then uses this to get $p(y|\boldsymbol{x})$ is called a *generative model*. In this class, we have studied a few generative models: Gaussian Discriminant Analysis, Naive Bayes, k-Nearest Neighbors. We will also study/have studied three discriminative models: support vector machines[18], logistic regression and linear regression. Not all machine learning methods fall cleanly into one of these classifications; some are hybrids.

We give some high-level general statements about generative versus discriminative approaches. For more details see the paper [37], as well as Section 8.6 in [36], Section 1.5 in [3] and Section 4.4.5 in [24].

Generative models force stronger assumptions in a model, by modeling the inputs (which discriminative models do not need to do). For example, while logistic regression and LDA both result in linear classifiers, LDA is derived and trained under the assumption of Gaussian inputs, while logistic regression makes no such assumption. If the model is indeed Gaussian as assumed, LDA can use less training data to get the same performance, but can require more if the assumption is violated [37, 36]. The assumptions in generative models can also lead to poorly calibrated (i.e. close to 0 or 1) posterior probabilities for the classes, relative to discriminative models. Depending on the application, this can be problematic or okay. For example, Naive Bayes classifiers produce poor estimates of the posterior distribution, but tend to do well in many applications, such as document classification [33]. Discriminative models also lend themselves naturally to things like kernel tricks or transformations of the features, because transforming the features does not violate modeling assumptions on the features as in a generative model (e.g., applying a non-linear transform of features assumed to be Gaussian often results in non-Gaussianity in a non-trivial to quantify way).

Generative models do have advantages over discriminative models, though. For example, one can infer things about the feature vectors in a given class, $p(\boldsymbol{x}|y)$, which is common in tasks like speech and signal processing and Hidden Markov Models [39]. Generative models tend to also be easy to fit and handle test data with missing features. One can also perform some outlier detection via generative models by identifying datapoints with small $p(\boldsymbol{x})$, which may be poorly predicted [3].

A good model is a function of the task at hand, so one cannot say that discriminative models are always better than generative methods. But, discriminative models are often preferred (for example, logistic regression is often preferred to LDA, despite producing similar results on a variety of problems [24] for the aforementioned reasons).

[18] Arguably, since the SVM does not model $p(y|\boldsymbol{x})$, one may want to call it neither discriminative nor generative. Bishop uses the term decision machine instead [3]. If one wants to get an estimate of $p(y|\boldsymbol{x})$ from an SVM, a heuristic (without any real justification) often used is called Platt scaling, which roughly speaking, takes the SVM classifier's discriminant function and feeds it into a logistic regression model. See Sec. 14.5.2.3 in [36] for details.

Chapter 3

Safety First: How to Handle Data

Before applying learning algorithms to real-world data, we need to know how to infer statistics from the data without doing operations on the data that could lead to false conclusions. Our presentation of this material is based on Chapter 7 of [24] and Chapter 22 of [44]. Depending on the references you choose (including the ones in the bibliography), there will be variations in the procedures described[1], but the fundamental point remains the same: **do not train on your test data**. While this section is written from the viewpoint of supervised learning, some of the ideas can be used for unsupervised learning.

3.1 Model Selection and Assessment

When we get a data set, we need to be careful not to use data in a way that we draw circular conclusions. For example, in supervised learning, if we learn a model on training data, then use the *same* data to estimate how well it generalizes (i.e. how it behaves on data we have not seen before), we may run into problems because the data outside our training set may not look much like the training data. As we saw in Sec. 2.2, in the extreme case, one can design a classifier that behaves perfectly on the training data, but generalizes very poorly. A useful example of this is the nearest neighbor (NN) classifier, which always has zero training error, but has non-zero prediction error.

We have two fundamental steps to designing a machine learning algorithm:

1. *Model Selection*: Given several models, estimate their performance in order to choose which is best.
2. *Model Assessment*: Having decided on the best model, estimate how well it generalizes (does on data it has not seen).

The key problem in doing model assessment and model selection is that if we are not careful with the data, we may use the data to draw circular conclusions. To alleviate this problem, one

[1]Many authors do not present a separate test set, or use different names for the splitting of the data set.

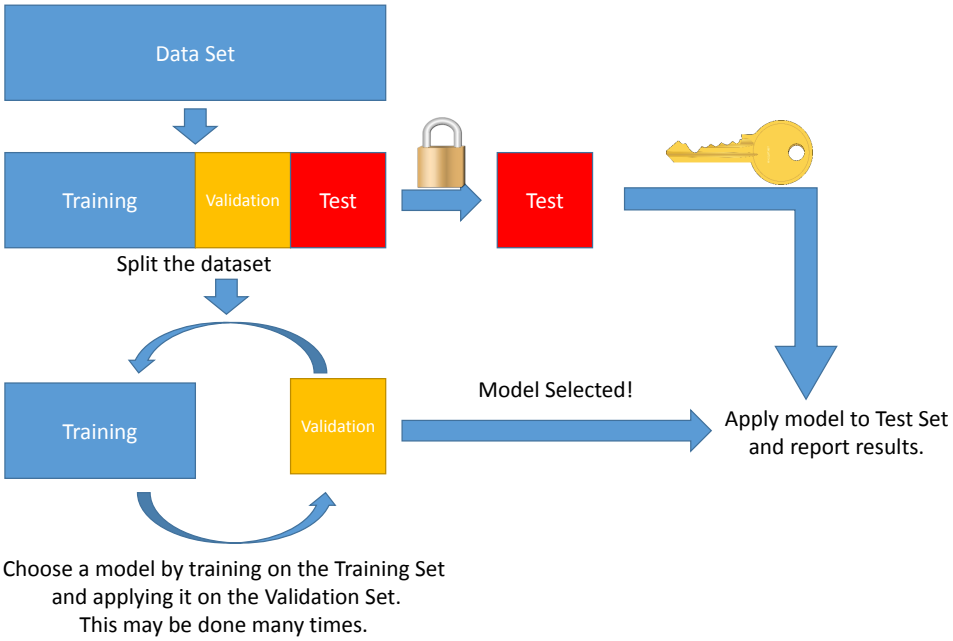


Figure 3.1: Work flow for model selection. Compare to Fig. 3.2 for using cross-validation.

should partition their data into three parts (without looking at the data): the *training set*, the *validation set*, and the *test set* as shown in Fig. 3.1. The training error is given by (2.2). The *validation error* is given by (2.5), where the data set used is the validation set. The *test error* is given by (2.5) where the data set used is the test set. The validation error and test error will both estimate the prediction error (2.4), but we will see shortly why the test error is a better estimate of the prediction error. Once we have split the data, we lock the test set in a dungeon and **do not allow anyone access to it until we are done – this is sacrosanct**. The test set is used exclusively for model assessment.

The first step is *model selection*. This uses the training and validation sets. The *training set* is the data set on which we train our models. Once we have an model, we run it on the *validation set* and measure its validation error. This estimates the prediction error of our model. We do this for a few different algorithms (say, by varying a *tuning parameter* for a family of algorithms (such as the parameters of a Gaussian RBF kernel), or by tuning some *measure of complexity*[2]. By trying different algorithms on the validation set, we can get an idea as to how good the prediction error of the model is, and when we are satisfied, we can decide on a final model that we want to use (such as the model with the lowest validation error). Since we picked a model with good performance on the validation set, it will typically underestimate the prediction error of our algorithm on new data (hence the need for the test set).

[2]For the purpose of this course, we will leave model complexity at a heuristic level. One could think of model complexity as the effective number of parameters in a model. For example, having more neighbors in a kNN model would decrease model complexity, whereas having more regression coefficients could increase model complexity. See Chapter 7 in [24] for more principled methods for measuring model complexity.

Once we have decided which algorithm to use based on the training and validation sets, we are ready for *model assessment*. **When we reach this point, we cannot go back and select another model or tune parameters** – we have chosen a model and have to live with it. We remove the test set from the dungeon and run our chosen algorithm on the test set and calculate the test error. We then report the test error as our estimate of the prediction error of our model. Once the test set is used, it is rendered useless for accurately estimating how a model generalizes, as anything we do would be based on the information we gleaned from the test set, which could lead to circular reasoning.

At this point, we are done – we have an algorithm (which we selected during model selection) and an estimate of how it generalizes (which we determined during model assessment). A graphical summary of this process is shown in Fig. 3.1.

A rough guideline of selecting the sizes of the training-validation-test split is 50%, 25%, 25% of the data, but this is dependent on how much data you have. In cases where data is scarce, one can use cross-validation (almost always used; discussed in the next section) or resampling techniques.

3.2 Selecting Models with Limited Data: Cross-Validation

Usually, we do not have enough data to split into training, validation and test sets. This can lead to problems in getting good estimates of errors, since most models perform better with more training data, and not using a large portion of it can cause significantly worse performance.

One way to work around this is to use *cross-validation* for model selection[3]. We first partition the data set into a training set and a test set (without looking at it) and reuse the training set in a clever way to avoid needing to have a separate validation set. As in the previous section, one must lock away the test data until the model has been selected. Typically, the data set is split into 90% training set and 10% test set. The work flow is similar to that of Fig. 3.1, and is summarized in Fig. 3.2. Most machine learning systems make extensive use of some form of cross-validation.

Cross-validation is a relatively simple procedure. It is given in Alg. 5. Typically, k is taken to be 5 or 10 and the resultant procedure is called 5-*fold* or 10-*fold cross-validation*, respectively. In some cases, k is taken to be the size of the training set, and the resultant procedure is called *leave-one-out cross-validation*.

A pictorial example of cross-validation is given in Fig. 3.3. In this example, 3-fold cross-validation is being used. We first train a model on folds $2, 3$ and calculate the error on fold 1 to find L_1. Then, we train a model on folds $1, 3$ and calculate the error on fold 2 to find L_2. Finally, we train a model on folds $1, 2$ and calculate the error on fold 3 to find L_3. Then, the cross-validation error is $\frac{L_1+L_2+L_3}{3}$.

[3] Another way is resampling, such as via the Bootstrap or Jackknife. See, e.g., Ch. 7 of [24].

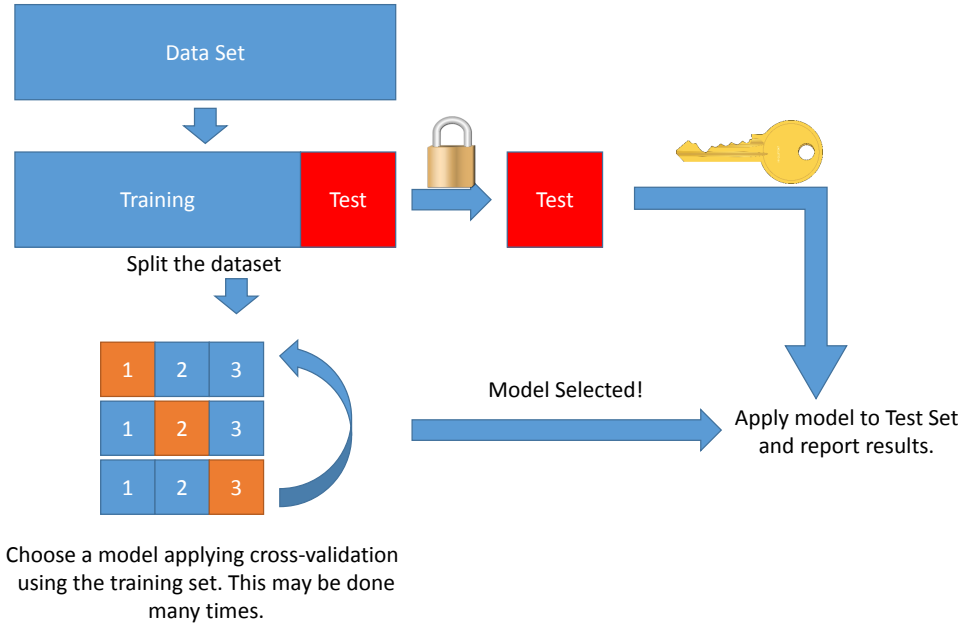


Figure 3.2: Work flow for model selection. Compare to Fig. 3.1.

Algorithm 5 Cross-validation with k folds. In many cases, it is also useful to report the standard error for the cross-validation error estimate.

function CROSSVALIDATION(Training set $\mathcal{T}$, number of folds k)

Split the data into k approximately equal parts randomly, $\mathcal{T}_1, \ldots, \mathcal{T}_k$, where $\mathcal{T}_\ell$ is called the ℓth *fold*.

for $\ell = 1, \ldots, k$ **do**

Use the data not in fold ℓ, $\tilde{\mathcal{T}}_\ell = \cup_{j \neq \ell} \mathcal{T}_j$, as the training set to train the model.

Use the data in fold ℓ, $\mathcal{T}_\ell$, as a validation set to calculate the validation error L_ℓ for the model trained on $\tilde{\mathcal{T}}_\ell$.

end for

Form the error estimate via cross-validation (the *cross-validation error*): $\mathrm{Err}_{\mathrm{cv}} = \frac{1}{k} \sum_{\ell=1}^{k} L_\ell$

return $\mathrm{Err}_{\mathrm{cv}}$

end function

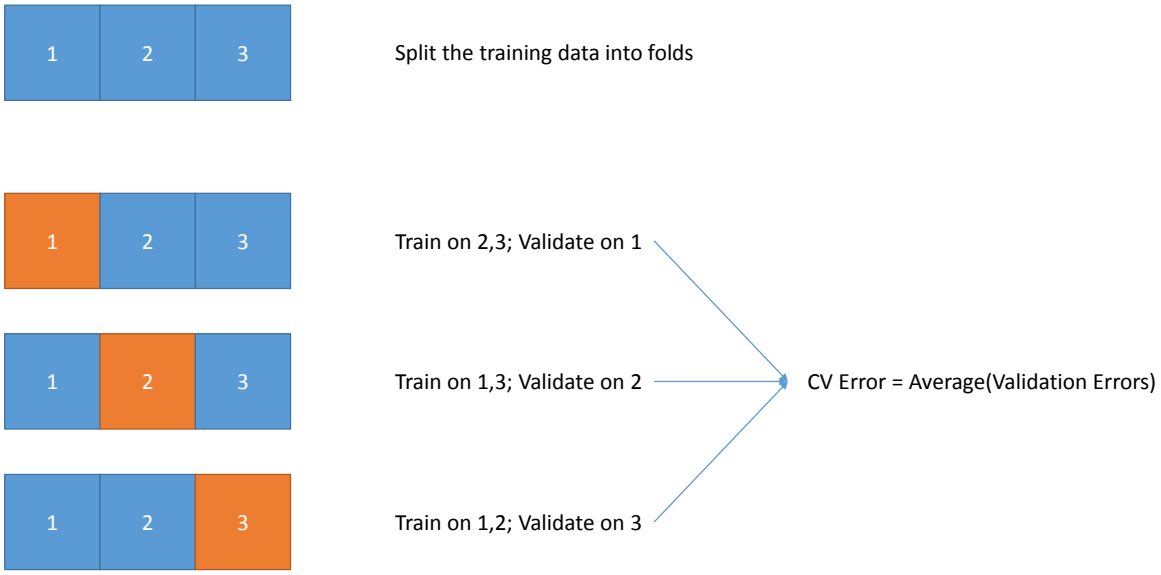


Figure 3.3: Applying 3-fold Cross-validation

To perform model selection with cross-validation, one simply evaluates the cross-validation error for all models under consideration with Alg. 5, and selects the one with the lowest cross-validation error.

Once the model has been selected, we can train our model on the whole training set and apply model assessment exactly as in the previous section in order to estimate how well the selected model generalizes.

One perhaps subtle point to note is that cross-validation error is not really quite an estimate of the prediction error. That is, it does not have quite the same interpretation as $\widehat{\text{Err}}_{\text{pred}}$, given in (2.5). However, we hope that the errors from cross-validation follow the same trend as the prediction error, and thus use the cross-validation errors to act as surrogates for the prediction error in order to choose a model.

Chapter 4

Clustering

Clustering is the problem of starting with a data set $\mathcal{D} = \{\boldsymbol{x}_i\}_{i=1}^N$ and trying to divide it into groups, known as *clusters*. Ideally, data points within the same cluster should be closer to each other than to members of other clusters. The notion of "close" is measured through a *similarity measure*, which is small when things are close (such as the cosine similarity introduced earlier, or Euclidean distance). Since clustering is done in an unsupervised setting, it is relatively difficult to measure the quality of clusters objectively [36]. Nonetheless, clustering has many applications, such as reducing the dimensionality of data in order to visualize it (e.g., if you have many data points, you can summarize them into a few data points via clustering), generating labels for unlabeled data, and applications such as the ones outlined at the end of this section. In *hierarchical clustering*, an ordering is imposed on the clusters by having a nested sequence of clusters. In *spectral clustering*, the data set is modeled as a graph, and tools from spectral graph theory are used to cluster the data.

Example: An example of hierarchical clustering would be to group the set {dog, cat, fern, human, cockroach, mushroom} based on some similarity metric. We can do this by a bottom-up (*agglomerative*) approach where we start with each data point in its own cluster. Then, we can combine them based on what seems similar. At the first iteration, we may have clusters {dog, cat}, {fern},{human}, {cockroach},{mushroom}. Then, we may cluster them again to get {dog, cat, human}, {fern, mushroom }, {cockroach}. Repeating again, we may get {dog, cat, human, cockroach},{fern, mushroom}. And finally, we may get the whole set back. In this case, the clusters form a hierarchy of similarity – dogs and cats are more similar to each other than to a human, but dogs, cats and humans are more similar to each other than to ferns and mushrooms. Hierarchical clusters are often visualized with *dendrograms* – an example from high school biology would be a phylogenetic tree as shown in Fig. 4.1. Note that we could also have done clustering from a top-down (*divisive*) approach, where we started with the whole set and iterative split it into sub-clusters.

In this class, we will focus on one of the most common forms of clustering known as K-means clustering, which has seen applications in areas such as signal and image processing (vector quanti-

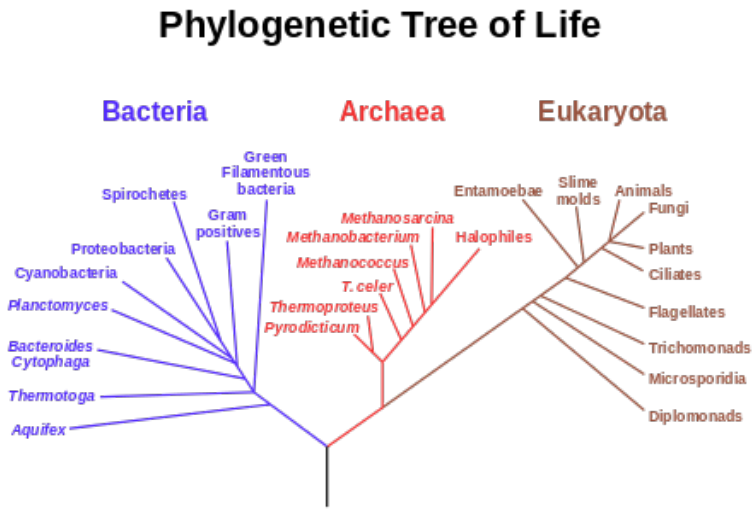


Figure 4.1: A phylogenetic tree (From `http://en.wikipedia.org/wiki/Phylogenetic_tree`; Public Domain)

zation, image segmentation), and learning (to reduce the complexity of NN, for example), to name a few. It also is a building block for more advanced forms of clustering, such as spectral clustering and serves as inspiration for other forms of clustering, such as K-medoids or K-modes[1] [29].

4.1 K-means

The K-means algorithm attempts to iteratively split a data set into exactly K clusters based the Euclidean distance. The algorithm tries to find clusters such that the sum of squared distances of each point to the center of its cluster is minimized[2] in an iterative fashion. That is, K-means clustering tries to minimize

$$J_K(\{\boldsymbol{\mu}_j\}_{j=1}^K, \{z_i\}_{i=1}^N) = \sum_{i=1}^N \|\boldsymbol{x}_i - \boldsymbol{\mu}_{z_i}\|^2 \tag{4.1}$$

over $\{\boldsymbol{\mu}_k\}, \{z_i\}$, where $\boldsymbol{x}_i$ is assigned to cluster z_i, and $\boldsymbol{\mu}_k$ is the center of cluster k. The objective function in (4.1) is also known as the K-means objective, and is related to the Expectation-Maximization algorithm for estimating the parameters of a Gaussian mixture model.

The K-means algorithm begins by initializing the *centers* (*prototypes*) of the K clusters, $\{\boldsymbol{\mu}_k\}_{k=1}^K$. Typically, this initialization is performed by randomly picking K points. The first step of the K-means algorithm is to assign each data point $\boldsymbol{x}_i$ to cluster z_i, where $\boldsymbol{\mu}_{z_i}$ is the closest cluster center to $\boldsymbol{x}_i$. The cluster centers are then updated to the average of all the points within the cluster. This

[1] Since K-means relies on Euclidean distance, it may not be appropriate for categorical data. K-modes clustering is a technique which may perform better on categorical data by using distances such as Hamming distance. See chapters 10 and 11 in [23] for more details and an extensive overview of other clustering algorithms.

[2] Exactly solving this problem is NP-hard. However, the K-means algorithm is simple and works well in practice. Under suitable modifications, one can show some approximation properties.

is repeated until convergence (e.g. a fixed number of iterations, or up to a point where the cluster centers do not change too much[3]). The algorithm is given in pseudocode in Alg. 6.

Algorithm 6 The K-means algorithm

function KMEANS(Data Set $\mathcal{D} = \{\boldsymbol{x}_i\}_{i=1}^N$, number of clusters K)
  Initialize the centers of the K clusters, $\{\boldsymbol{\mu}_k\}_{k=1}^K$. ▷ This is often done randomly.
  repeat
    for $i = 1, \ldots, N$ **do**
      Assign $\boldsymbol{x}_i$ to cluster $z_i = \arg\min_{k=1,\ldots,K} \|\boldsymbol{x}_i - \mu_k\|$
    end for
    for $k = 1, \ldots, K$ **do**
      Update the center of cluster k as $\boldsymbol{\mu}_k = \frac{1}{N_k} \sum_{i:z_i=k} \boldsymbol{x}_i$, where N_k is the number of points assigned to cluster k.
    end for
  until convergence.
  return the cluster assignment for each data point $\{z_i\}_{i=1}^N$ and the cluster centers $\{\boldsymbol{\mu_k}\}_{k=1}^K$
end function

It should be noted that the K-means algorithm is quite sensitive to the initial choice of cluster centers, and therefore it is often run a few times until a good result is achieved. Variations of the algorithm, such as the K-means++ algorithm are designed to alleviate this problem.

Example: In Fig. 4.2 the K-means algorithm is applied to form 2 clusters. The top left shows the unclustered data with the cluster centers marked as stars. Each row represents an iteration of K-means. The left column is the cluster assignment step, where each input vector is assigned to the closest cluster center. The right column is the cluster center update, which is the average of the vectors in each cluster. The output of the K-means algorithm after the appropriate number of iterations is shown in the right column.

A variation of the K-means algorithm is the K-medoids algorithm, in which the Euclidean distance is replaced with a more general dissimilarity measure, and the center of the cluster is replaced with the centroid of the cluster (i.e. the point that has minimum average dissimilarity with respect to other points in the cluster). One can view this as a kernelized version of K-means clustering. While K-means does a linear pass of the training data in each iteration, K-medoids typically requires quadratic time, due to the centroid computation[4]. Other useful variants do soft clustering, where ambiguity in which cluster a point is assigned to is allowed (see, e.g., Ch. 9 of [3]).

[3]The K-means algorithm will eventually give a fixed point, since $J_K((\{\boldsymbol{\mu}_j\}_{j=1}^K, \{z_i\})$ is non-increasing and there are only finitely many possible cluster assignments. However, this may take a long time in some cases.

[4]In some dissimilarity measures (e.g. Bregman Divergences), one can avoid computing the centroid and use the mean instead, and have a useful algorithm [2].

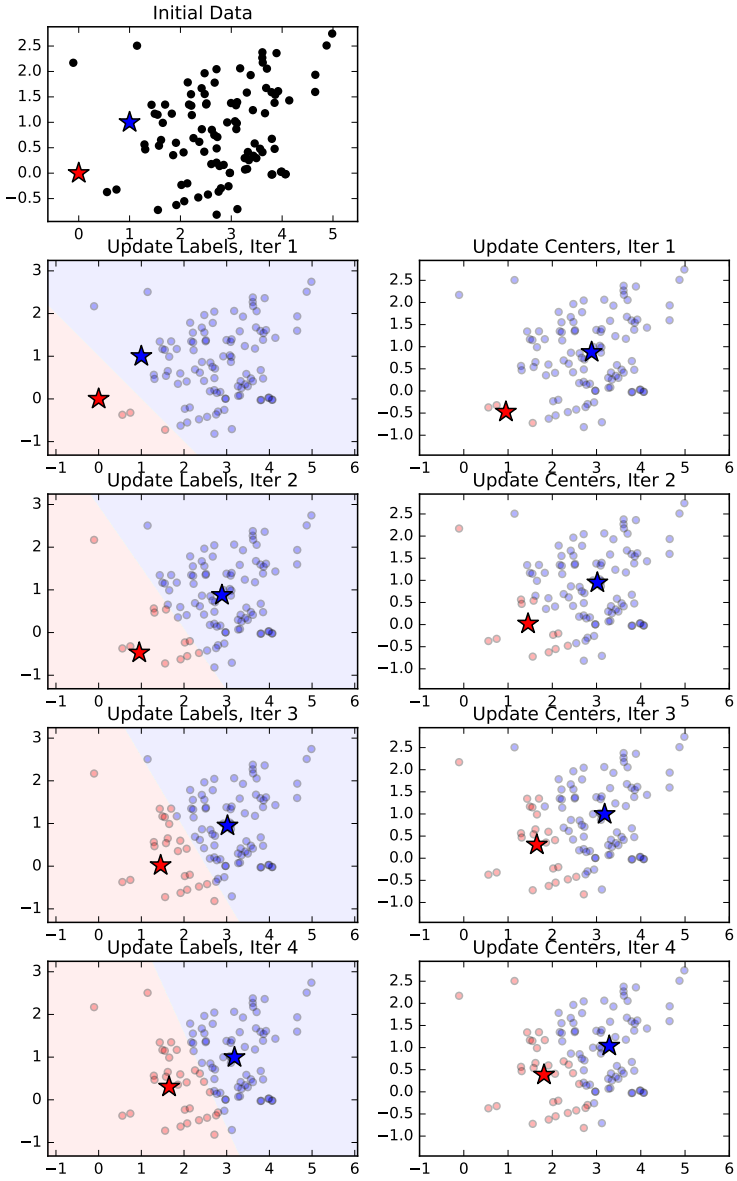


Figure 4.2: A demonstration of the K-means algorithm. The top left shows the unclustered data and selected initial cluster centers. The left column shows cluster assignment updates (with the background shading denoting which cluster center points would be assigned to), while the right column shows cluster center updates. Based on Fig. 9.1 in [3].

4.1.1 How to pick K?

One of the main problems with the K-means algorithm is the specification of the parameter K. In some cases, the choice of K is obvious – for example, if one knows the data set is handwritten digits, then $K = 10$ since the digits are $\{0, \ldots, 9\}$. There are several techniques for estimating K, but we present one of the simplest heuristics known as the "elbow" (or knee) method. One attempt to formalize this mathematically is the gap statistic discussed in Section 14.3.11 of [24].

Other options such as a likelihood-based method are given in [36].

First, we define the optimal value of the K-means objective, $J^*(K)$ as

$$J^*(K) = \min_{z_1,\ldots,z_N,\mu_1,\ldots,\mu_K} \sum_{i=1}^{N} \|\boldsymbol{x}_i - \boldsymbol{\mu}_{z_i}\|^2 \tag{4.2}$$

where $z_1, \ldots, z_N \in \{1, \ldots, K\}$. We can approximate $J^*(K)$ by calculating $\sum_{i=1}^{N} \|\boldsymbol{x}_i - \boldsymbol{\mu}_{z_i}\|^2$ for multiple applications of K-means clustering using different initializations, and using the smallest value. Note that $J^*(K)$ is non-increasing in K, with J^* approaching 0 as K approaches N.

The elbow method looks for a kink (elbow, knee) in the $J^*(K)$ versus K curve. The elbow method relies on $J^*(K)$ to be decreasing rapidly before the kink, and flattening out after the kink. The intuition behind this is that increasing the number of clusters should make $J^*(K)$ decrease rapidly if a cluster should really be split up into more than one cluster. If the number of clusters is right, breaking the clusters down further should not change $J^*(K)$ much, since the members of the cluster are already relatively similar. One then picks the number of clusters to be around where the kink occurs.

Note that in some cases, there is no kink or there may be multiple kinks, so this heuristic for selecting K may not produce useful results.

Example: We generate data with 3 clusters shown in Fig. 4.4. The $J^*(K)$ versus K curve, shown in Fig. 4.3, was estimated by running K-means 10 times for each K and keeping the smallest value of the K-means objective.

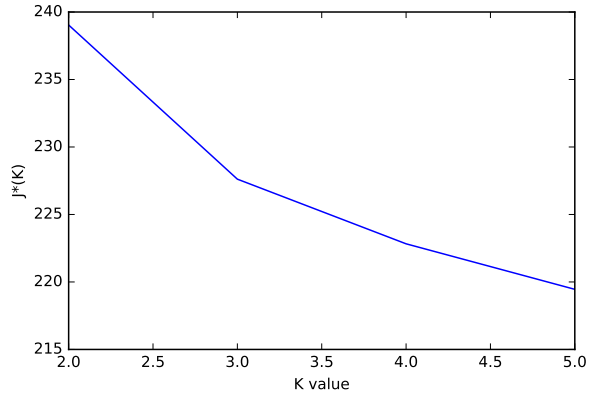
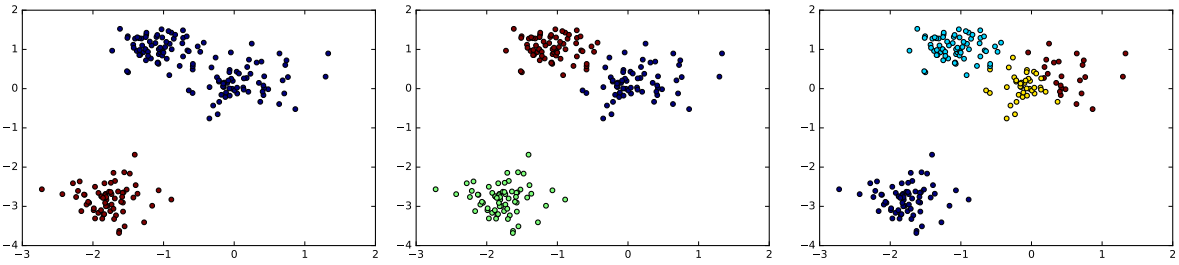


Figure 4.3: An example of an $J^*(K)$ versus K curve. There is a kink at $K = 3$.

From the $J^*(K)$ versus K curve, we see that a kink occurs at around 3, so we would select three clusters. Indeed, when we look at the clusters for $K = 2, 3, 4$ shown in Fig. 4.4, we see that moving from 2 clusters to 3 clusters splits datapoints which are truly in 2 clusters into a separate cluster (with a large decrease in the $J^*(K)$ versus K curve). Moving from 3 clusters to 4 clusters splits a true cluster into two clusters (with a smaller decrease in the $J^*(K)$ versus K curve).



(a) Two clusters (b) Three clusters (c) Four clusters

Figure 4.4: Examples of K-means clustering with 3 true clusters and $K = 2, 3, 4$

4.2 Applications

Since clustering is more open to interpretation than classification, we will highlight three simple applications of clustering, some of which you will implement in the lab.

4.2.1 Vector Quantization

Vector quantization (VQ) is a form of *lossy signal compression*. A *lossless compression* algorithm is one which the original input can be perfectly reconstructed from the compressed data. Examples of this are the zip file format on your computer. A *lossy compression* algorithm is one where the original input is approximately reconstructed from the compressed data – that is, the reconstructed input has some *distortion*. An example of this is the telephone system, where your voice is *quantized* (rounded to a finite set of values) and transmitted over the phone line. The receiver reconstructs your voice approximately [25]. Lossy compression can achieve tremendous savings over lossless compression, especially in fields like audio, images and video, where we do not perceive fine details. Compression algorithms like JPEG exploit ignoring fine details in order to get much smaller file sizes than lossless compression with little perceived distortion.

In vector quantization, we attempt to approximate vectors in our feature space, $\mathbb{R}^d$, by a finite set of K vectors called a *codebook*. The vectors in the codebook are called *code vectors* or *codewords*. If we want to send a feature vector to someone, we choose a codeword to approximate the vector and transmit its index. When we receive the index, we simply look up the corresponding codeword in the codebook and use this codeword as an estimate of what was to be transmitted. If the code book has K vectors, we need to send $\log_2(K)$ bits to send the index of a codeword[5], assuming the receiver has the codebook (to decode the indices). This can be significantly smaller than directly transmitting the input. An old example[6] of this is the telegraph system. Many times, people will send the same message, e.g. "Happy Birthday!", "How are you?", or "We are fine.". Instead of tapping out each message each time, a system was developed to assign indices to common messages, and transmit those indices instead. The receiver, with the knowledge of the mapping between indices and common messages, could then tell someone what the message was from the index (making the communication more efficient, and saving the telegraph operator a considerable

[5]For those with a bit of information theory background, if the codewords are not equally likely, we can pass the indices through an entropy coder in order to send even fewer bits.

[6]Not so old – the last telegram sent in India was on July 15, 2013.

amount of work) [9].

One way to find a good codebook is to use K-means clustering. Let us say we are trying to compress (grayscale) images. We first acquire some images on which to train our algorithm (which may just be the image we want to store). We extract features from these images by chopping the image up into small blocks, say 5 pixels by 5 pixels, and reshape them to get a set of vectors in $\mathbb{R}^{25}$. These vectors form our data set. The idea that makes vector quantization work is that in many images, there will be several similar blocks (such as constant color patches, sky textures, hair, etc.). So, if we group similar blocks together, we can use one block in a group to approximate the rest of the blocks in the group. If we want a codebook of size K, we use K-means clustering to split the blocks into K groups. For each cluster, we consider the center of the cluster as a codeword.

When we want to compress an image, we chop the image up into the same size blocks as the codewords. Then, for each block, we find the closest codeword and store its index instead of the block itself. We also store the codebook. When we want to decompress the image, we simply use the codeword corresponding to the index as the reconstruction of the block.

This form of vector quantization is the simplest, and produces images that do not look very nice (they tend to look quite blocky). However, the ideas can be extended to make much better compressors as shown in [18]. In the context of vector quantization, the K-means algorithm is often called *Lloyd's algorithm*. As the algorithm is easier to explain pictorially, we describe training a vector quantizer in Fig. 4.6 and compressing/decompressing an image via VQ in Fig. 4.7.

Example : A demo of vector quantization is given in Fig. 4.5. The initial image is a 630x420 image of Larry, the Chief Mouser to the Cabinet Office[7]. Vector quantization is applied with 5x5 blocks. One can see that 20 codewords gives an acceptable image.

4.2.2 Image Segmentation

Image segmentation is the problem of partitioning an image into areas that have common meaning (called segments, e.g. a face, the background of an image, the foreground of an image, areas of homogeneity, etc). There are many techniques to do this, including various signal processing approaches (edge detection), probability models (graphical models), and so on (see, .e.g., [17, 21]). One simple algorithm for image segmentation is based on K-means [3].

We start with a color image (with each pixel consisting of a red, green and blue (RGB) value[8]), and get our data set by considering the vectors in $\mathbb{R}^3$ corresponding to the RGB values for each pixel. If we want to segment the image into K types, we use K-means clustering to get K clusters. We then replace each pixel in the image with the center of the cluster to which it belongs. This approach to image segmentation is quite similar to the vector quantization example.

[7] Yes, this is a real title in the United Kingdom. And yes, that is Larry's official government photograph (under the OGL). See `https://en.wikipedia.org/wiki/Chief_Mouser_to_the_Cabinet_Office` for the source of this image, as well as details.

[8] This 3-dimensional vector space is known as a color space. There are many others color spaces that are useful for applications, such as a YUV or L*a*b space. We use RGB primarily for simplicity here.

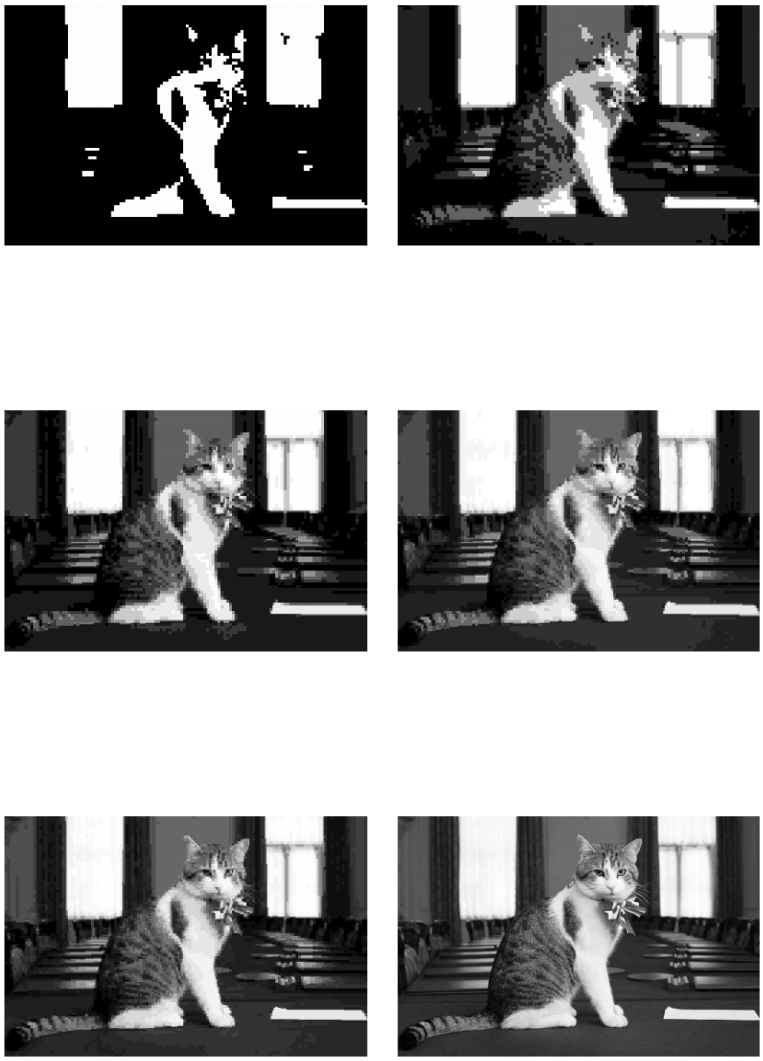


Figure 4.5: An example of vector quantization using 5 x 5 blocks. The number of codewords used is (from left to right and top to bottom): 2,5,10,20,50, Original image. The number of bits per pixel used is 0.04 , 0.09, 0.13, 0.17, 0.23, 8, respectively.

Note that this segmentation algorithm ignores the particular positions of the pixels, which can provide useful information for creating a good segmentation. However, K-means is a building block for more sophisticated segmenters, such as ones based on mixtures of Gaussians (which uses ideas that extend from what is presented here), to spectral methods (which use K-means in a way that is quite different than what is presented here) [40].

Example : A demo of K-means clustering to do segmentation is given in Fig. 4.8. We see that even with 2 clusters, we get some separation of the foreground and background, while increasing the number of clusters separates different sections of the image, such as the different types of hair on Larry.

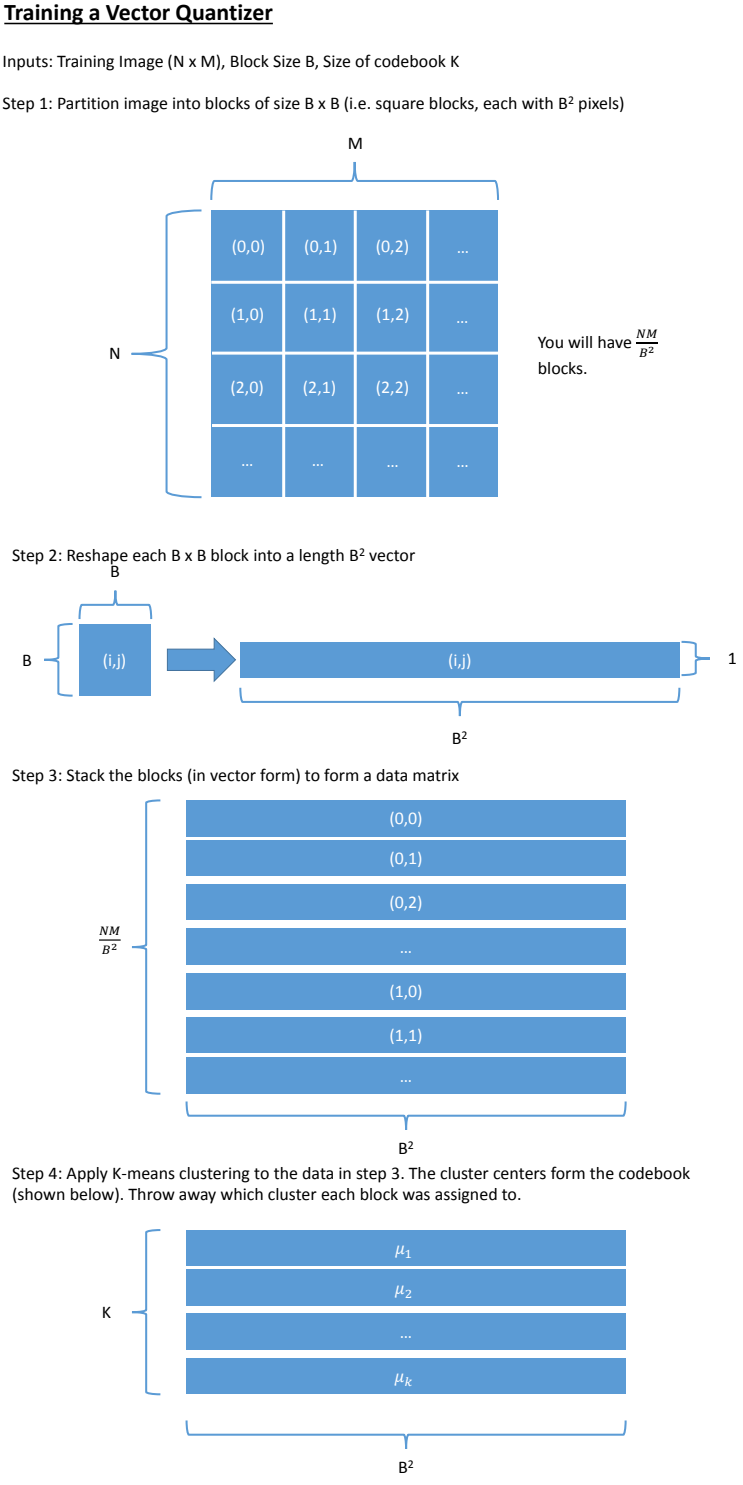


Figure 4.6: How to Train a Vector Quantizer

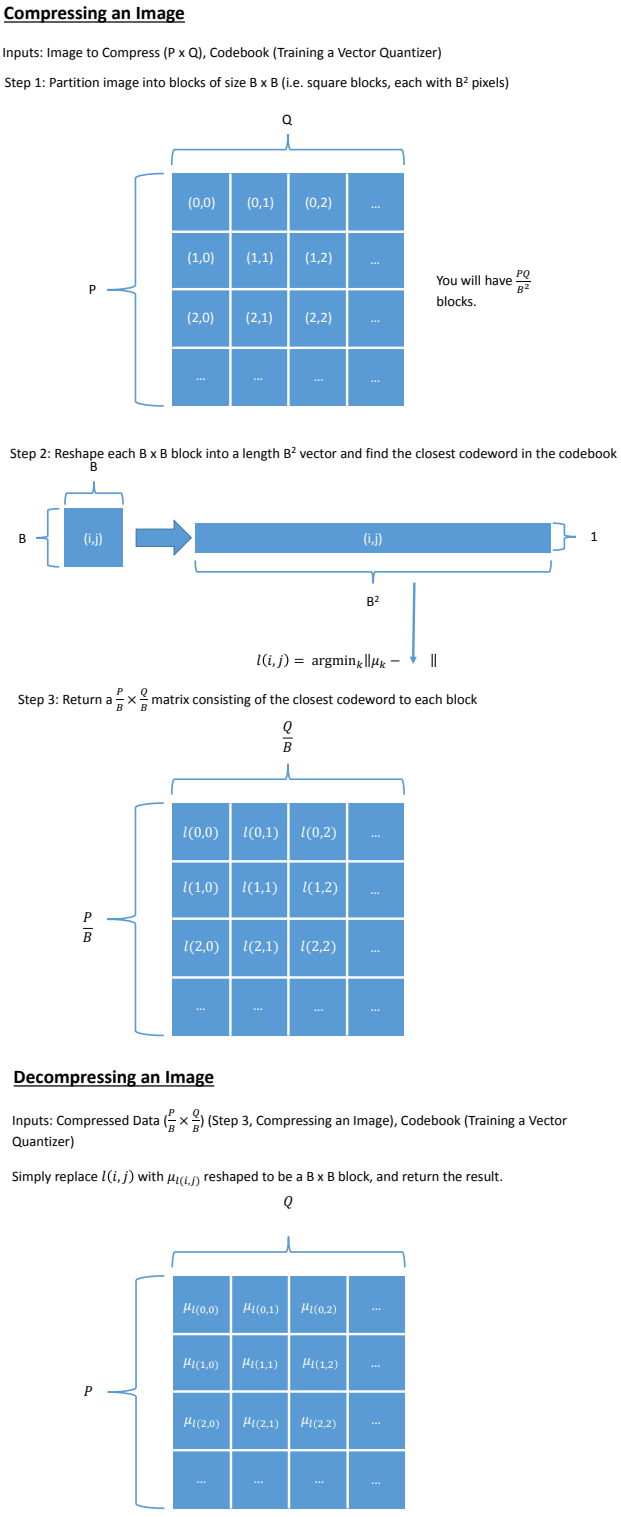


Figure 4.7: Compressing and Decompressing an Image via Vector Quantization

Figure 4.8: An example of image segmentation using K-means clustering with $K = 2, 3, 5, 10, 20$ and the original image. Image source is same as Fig. 4.5

4.2.3 Supervised Learning

As we discussed in Section 2.3, one of the problems with with Nearest Neighbor classification is that for each datapoint we want to test, we need to compute the distances between the test datapoint and all the feature vectors in the training data For large training data and/or many test datapoints, this can be quite inefficient. One way to get around this issue is to use clustering to reduce the number of distance calculations that need to be done for testing a datapoint.

We can use K-means clustering to do M-ary classification as shown in Alg. 7. We first split the training data according to its label. Then, for the feature vectors under label ℓ, we apply K-means to get K clusters. We call the centers $\{\boldsymbol{\mu}_{\ell,1}, \ldots, \boldsymbol{\mu}_{\ell,K}\}$ of these clusters the *prototypes for class* ℓ.

We then form a new data set $\mathcal{T}'$ consisting of the pairs of prototypes under each class and their corresponding class (generatePrototypes). We then apply a nearest neighbor classifier using $\mathcal{T}'$ as the training data (classifyQuickly).

Algorithm 7 Using K-means clustering to speed up Nearest Neighbors for M-ary classification

function GENERATEPROTOTYPES(Training data $\mathcal{T} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$, Number of prototypes per class K)
  Initialize $\mathcal{T}' = \emptyset$
  for $\ell = 1, \ldots, M$ **do**
    Apply K-means clustering on the set of training datapoints with $y_i = \ell$ to get cluster centers $\{\boldsymbol{\mu}_{\ell,1}, \ldots, \boldsymbol{\mu}_{\ell,K}\}$ ▷ Alg. 6
    Add the pairs $\{(\boldsymbol{\mu}_{\ell,1}, \ell), \ldots, (\boldsymbol{\mu}_{\ell,K}, \ell)\}$ to $\mathcal{T}'$.
  end for
  return $\mathcal{T}'$
end function
function CLASSIFYQUICKLY(Labeled Prototypes $\mathcal{T}'$, test datapoint $\boldsymbol{x}$)
  return Nearest Neighbor classification of $\boldsymbol{x}$ using $\mathcal{T}'$ as the training data ▷ Alg. 2
end function

Since the number of prototypes per class can be much fewer than the number of training datapoints, using Alg. 6 can result in a tremendous speedup. However, note that the classes are not considered jointly – that is, the prototypes that are used for class ℓ are independent of the training data of other classes. One approach to alleviate this is known as *learning vector quantization* and is outlined in section 13.2.2 in [24].

Example : In Fig. 4.9, 100 training datapoints from three classes (red, green and blue) is shown. The Bayes classifier, NN classifier and NN using 5 prototypes generated by applying K-means to generate 5 clusters under each class are shown. The prototypes are given by stars. Note that the NN with prototypes decision boundary is significantly less jagged than if nearest neighbors were applied directly, yet captures a lot of the separation between the classes. Each test datapoint only requires the calculation of 15 distances, versus an order of magnitude more distances with a direct application of nearest neighbors.

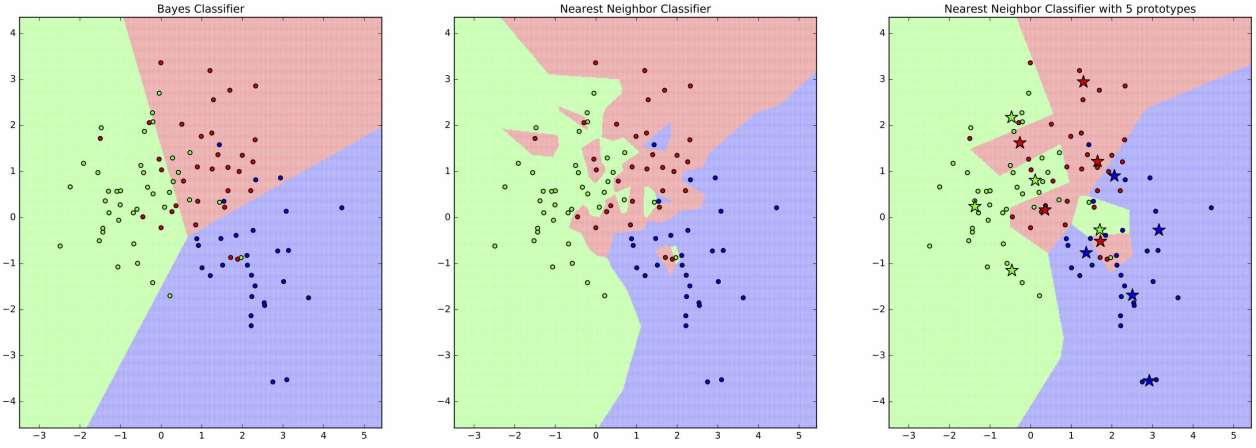


Figure 4.9: Using K-means clustering to speed up a nearest neighbor classifier.

Chapter 5

Regression

In previous sections, we studied the problem of classification, where we had data that came from one of several discrete labeled classes and given an input vector, we wanted to estimate its label. In *regression*, we have data that has a continuous label (which we will call a *response* or *output*), and given an input vector, we want to estimate its response.

Once again, we will start with a set of training data $\mathcal{T} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$. A *regression model* is a function $f(\boldsymbol{x})$ that maps a feature vector $\boldsymbol{x}$ (of dimension d) to an estimate (prediction) of its corresponding response y. In other words, we are performing *curve fitting* to the training data. In contrast to classification, the response will be continuous. We will focus on the case where the responses are real numbers, although the ideas can be easily extended to vector responses.

5.1 Measuring Performance for Regression and Model Selection

Measuring performance for regression is similar to classification, except we usually work with the square loss rather than 0-1 loss[1]. All ideas from the previous sections for estimating training error, test error, selecting parameters for cross-validation, etc. go through, except we use the squared loss, which is defined as:

$$L(f(\boldsymbol{x}), y) = (y - f(\boldsymbol{x}))^2. \tag{5.1}$$

While we will not discuss it much in class, an important (theoretical) aspect of using the square loss is the *bias-variance tradeoff*, which describes how far off predictions are over a training data set, averaged over training data sets (bias), and how the prediction error varies across all training data sets from its average (variance). The error under the square loss can be decomposed as a sum of irreducible error (due to randomness in the data), squared bias and the variance. A good discussion is given in Section 3.2 of [3], and Section 7.3 of [24].

The terms of bias and variance are also applied in the context of other losses; some examples of this are given in Section 7.3.1 in [24]. Note that the decomposition into bias and variance will not follow the form given by the square loss in general.

[1]For continuous y, with the 0-1 loss function, we would always incur a loss. The squared loss is commonly used for continuous y and is mathematically tractable.

The point to take away is that for the square loss, high complexity models have low bias, but high variance. Low complexity models have high bias, but low variance. Your goal should be to minimize the prediction error by finding a sweet spot in model complexity which balances the bias and variance appropriately – we do not want to underfit or overfit. A pictorial representation is given in Fig. 5.1. We will leave the idea of complexity at an intuitive level – some discussion for methods for quantifying complexity are given in Chapter 7 of [24].

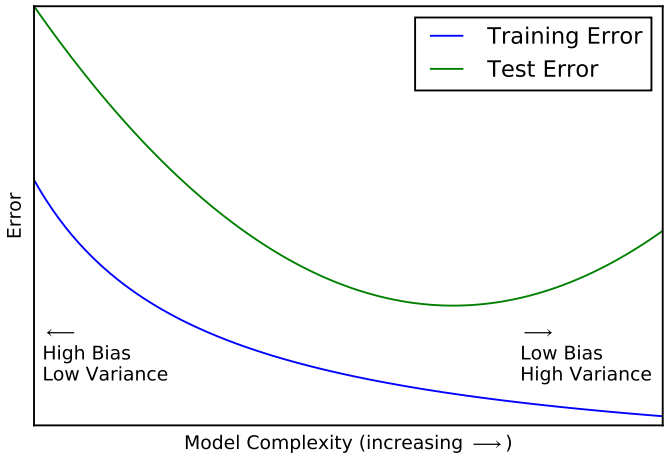


Figure 5.1: Prediction error versus model complexity. Based on Fig. 2.11 in [24].

In this chapter, which is primarily on linear regression, we will start with least squares for linear regression, which has no bias. But, in practice, gaining a bit of bias from using techniques like ridge regression can drop the variance to give overall better performance. We do note that in order to make fair comparisons of model complexity, the family of models need to have some sense of compatibility in information. For example, we can compare least squares, ridge regression, and the LASSO on a given feature space easily (since they are all linear regression models), but if we allowed a richer feature space (e.g. augmenting with non-linear functions of the features), we may get improvements in both bias and variance over the models in the original feature space (e.g. if we were trying to predict a quadratic function of the features with linear functions, and augmented with quadratic features). Again, we defer to Section 7.3 in [24] for more details.

5.1.1 k-Nearest Neighbor Regression

The simplest regression algorithm is a variation of the nearest neighbor algorithm for classification. It is reasonably intuitive: If you have an input feature vector $\boldsymbol{x}$ for which you are trying to predict its response, find the k closest points in the training set and average their responses to estimate the response for $\boldsymbol{x}$.

Unlike k-Nearest Neighbor classifiers, k-Nearest Neighbor regression is not very popular[2]. Like

[2]But, you can use it to motivate kernel smoothing (not related to kernel tricks in Section 2.10) and local regression methods [24], which are popular.

Algorithm 8 Algorithm for k-Nearest Neighbor Regression

function k-NEARESTNEIGHBORSREGRESSION(Test datapoint $\boldsymbol{x}$, Training Set $\mathcal{T} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^N$)
  for i=1,...,N **do**
    Calculate $\delta_i = \|\boldsymbol{x} - \boldsymbol{x_i}\|$
  end for
  Find indices $\{i_1, \ldots, i_k\}$ corresponding to k smallest values of δ_i (break ties arbitrarily).
  return $\frac{y_{i_1}+\ldots+y_{i_k}}{k}$.
end function

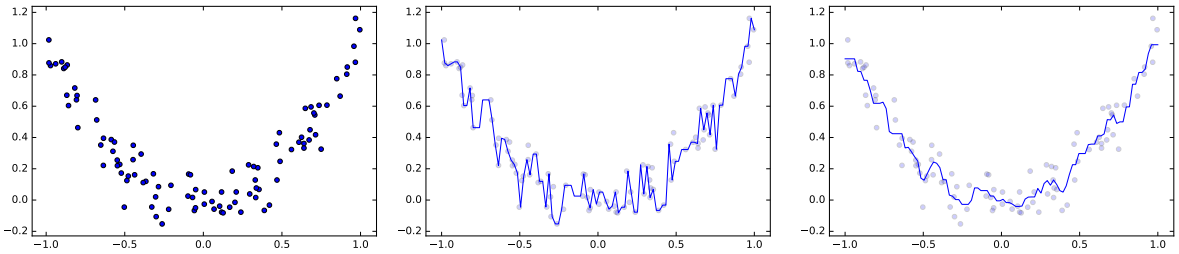
the corresponding classifiers, they are *local* methods. The model given by k-NN regression is piecewise constant, with the estimated output of an input depending solely the responses of the training points closest to the input. In contrast, the most popular formulation of regression, *linear regression* is a *global* method, since it uses information from all the training data to determine the model's estimate of an output.

However, the k-Nearest Neighbor Regression algorithm gives a simple way to highlight the Bias-Variance Tradeoff. We give an example based on Sec. 2.9 in [24]. Assume our data follows the model $Y = f(X) + \epsilon$ where ϵ is noise with mean zero and variance σ^2. Then, the test error for predicting $x = x_0$ with a particular instance of training data $\mathcal{T}$ can be written as

$$E[(Y - \hat{f}_k(x_0))^2 | X = x_0, \mathcal{T}] = \sigma^2 + \left(f(x_0) - \frac{y_{i_1} + y_{i_2} + \ldots + y_{i_k}}{k} \right)^2 + \frac{\sigma^2}{k}. \tag{5.2}$$

The first term, σ^2 is the irreducible error. The second term, $\left(f(x_0) - \frac{y_{i_1}+y_{i_2}+\ldots+y_{i_k}}{k} \right)^2$ is the squared bias, which is the squared difference between the truth and the average value of the estimate. The third term, $\frac{\sigma^2}{k}$ is the variance of the estimate, $\frac{y_{i_1}+y_{i_2}+\ldots+y_{i_k}}{k}$. As k increases, our model becomes less complex (the responses we predict will be smoother). We see that this can increase the bias (our predictions will take into account more training points farther away, so our predictions will be on average more off). But, the variance will drop.

Example: Let us consider the function $f(x) = x^2$ and apply k-Nearest Neighbor Regression in the model given above: $Y = f(X) + \epsilon$ with $\epsilon = \mathcal{N}(0, 0.1)$. The training data and fitted models are shown in Fig. 5.2.



(a) Training Data (b) Using 1-NN Regression (c) Using 5-NN Regression

Figure 5.2: Examples of k-NN regression. Note that the 5-NN regression model is less complex than the 1-NN regression model.

We also set aside a validation set of 50 data points (not shown), and found a validation error for 1-NN to be 0.03 and for 5-NN to be 0.01. We also tried a constant prediction (i.e. using the average of all the training points), with a validation error of 0.09. So, picking a less complex model can be advantageous, but one needs to have a sufficiently complex model in order to get good performance.

5.2 Linear Regression

The most common method of regression is *linear regression*. A *linear regression model* attempts to predict the response by taking a dot product between the feature vector and a vector of *weights* (coefficients), $\boldsymbol{\beta}$ (of the same length as the feature vectors):

$$f(\boldsymbol{x}) = \boldsymbol{x}^\top \boldsymbol{\beta}. \tag{5.3}$$

Linear regression models are quick to predict responses due to their simplicity, and are extremely powerful since one can augment the feature vectors with extra features (as in classification), such as powers or products of other features, or non-linear functions of other features (similar to the intuition behind kernel tricks), or encoding a feature that takes on a finite set of values using one-hot encoding (see Section 2.1). One can also non-linearly transform the responses to allow for better modelling (e.g. using the logarithm of the responses). For example, if we would like our linear regression model to have an intercept (which is often the case), we simply add a feature that always takes the value 1. Furthermore, a linear regression model helps us interpret the relationship between feature i and the response, based on its weight in the model, β_i. There are a number of ways to pick $\boldsymbol{\beta}$ for linear regression – we will cover three methods: (ordinary) least squares, Ridge regression and the LASSO.

5.2.1 Ordinary Least Squares

The *method of least squares* says that we pick the weights $\boldsymbol{\beta}$ to minimize the sum of the squared errors that our model makes on the training data. The sum of squared errors that our model makes on a given set of training data for a given set of weights $\boldsymbol{\beta}$ is known as the residual sum of squares (RSS):

$$\text{RSS}(\boldsymbol{\beta}) = \sum_{i=1}^{N} (y_i - \boldsymbol{x}_i^\top \boldsymbol{\beta})^2. \tag{5.4}$$

The RSS is given pictorially in Fig. 5.3. The *residual* for the training data $(\boldsymbol{x}_i, y_i)$ (as circles) is the vertical line (red) from the point $(\boldsymbol{x}_i, y_i)$ to the output of the linear regression model (black line) for that training input, $(\boldsymbol{x}_i, \boldsymbol{x_i}^\top \boldsymbol{\beta})$. The method of least squares picks $\boldsymbol{\beta}$ to minimize the sum of the norm-squared of residuals of the training data.

Let X be the (training) *data matrix* where row i is $\boldsymbol{x}_i^\top$. The data matrix has size $N \times d$. Let $\boldsymbol{y} = [y_1, \ldots, y_N]^\top$ be the vector of responses. Then, we can write the RSS more succinctly as

$$\text{RSS}(\boldsymbol{\beta}) = \|\boldsymbol{y} - X\boldsymbol{\beta}\|^2. \tag{5.5}$$

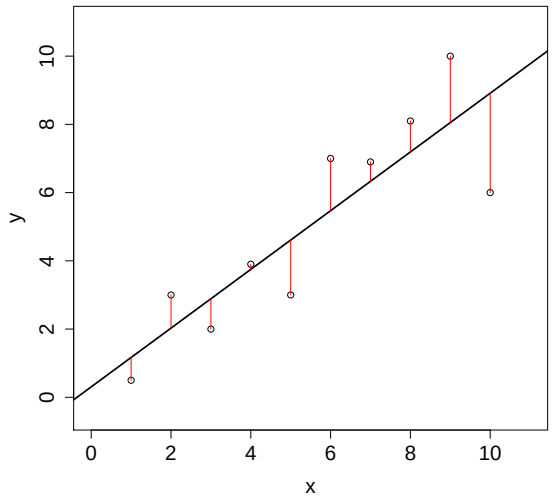


Figure 5.3: A linear regression model based on one feature (black line). The model is specified by a vector $\boldsymbol{\beta}$ that indicates the slope and intercept of the black line. The residuals are shown in red.

The *(ordinary) least squares* (OLS or LS) solution to linear regression is the model $f_{\text{LS}}(\boldsymbol{x}) = \boldsymbol{x}^\top \boldsymbol{\beta}_{\text{LS}}$ where $\boldsymbol{\beta}_{\text{LS}}$ is chosen to minimize the RSS:

$$\boldsymbol{\beta}_{\text{LS}} = \arg\min_{\boldsymbol{\beta}} \text{RSS}(\boldsymbol{\beta}). \tag{5.6}$$

We will use calculus to compute the LS solution. We expand the RSS (5.5) by writing it as a dot product:

$$\begin{aligned} \text{RSS}(\boldsymbol{\beta}) &= (\boldsymbol{y} - X\boldsymbol{\beta})^\top (\boldsymbol{y} - X\boldsymbol{\beta}) \\ &= \boldsymbol{y}^\top \boldsymbol{y} - 2\boldsymbol{y}^\top X\boldsymbol{\beta} + \boldsymbol{\beta}^\top X^\top X\boldsymbol{\beta}. \end{aligned}$$

The RSS is a differentiable function of $\boldsymbol{\beta}$, and so to minimize it, we take the gradient (derivative) with respect to $\boldsymbol{\beta}$

$$\nabla \text{RSS}(\boldsymbol{\beta}) = 2X^\top X\boldsymbol{\beta} - 2X^\top \boldsymbol{y}$$

and set it equal to zero to get the *normal equations*

$$X^\top X\boldsymbol{\beta}_{\text{LS}} = X^\top \boldsymbol{y}. \tag{5.7}$$

One can show that solving the normal equations indeed specify the minimizers of the RSS by convexity[3] (see [4] for details).

[3] A function is convex if the line joining any two points on its graph lies above the graph. This implies that all minimizers are global minimizers, and for differentiable functions, setting the derivative to zero is both necessary and sufficient for a minimizer. An example of a convex function is a quadratic that opens upwards, e.g. $g(z) = z^2$.

By solving the normal equations for $\boldsymbol{\beta}_{\text{LS}}$, we get a linear regression model. To write it down explicitly, assume that the columns of X are linearly independent (i.e. there is no vector $\boldsymbol{v}$ such that $X\boldsymbol{v} = 0$ other than $\boldsymbol{v} = \mathbf{0}$; it is necessary that we have at least as many training data points as features, i.e., that X is a *tall matrix*). Then, one can prove that $X^\top X$ is an invertible matrix and uniquely solve the normal equations as

$$\boldsymbol{\beta}_{\text{LS}} = (X^\top X)^{-1} X^\top \boldsymbol{y}. \tag{5.8}$$

The matrix $(X^\top X)^{-1} X^\top$ is a special case of the *(Moore-Penrose) pseudoinverse* of the matrix X, and is denoted by $X^\dagger$. While we cannot invert non-square matrices (or square matrices with determinant zero), the pseudoinverse has some of the same properties as the inverse. We can write the LS solution as

$$\boldsymbol{\beta}_{\text{LS}} = X^\dagger \boldsymbol{y}. \tag{5.9}$$

The pseudoinverse solution (5.9) *always* gives a correct least squares solution to linear regression, even when the columns of X are not linearly independent. The case where the features are not linearly independent is common in signal processing and big data. In these cases, X is typically a fat matrix (more columns than rows). When the columns of X are not linearly independent, there will be multiple vectors satisfying the normal equations: If you took a solution to the normal equations and added a non-zero $\boldsymbol{v}$ (which exists) such that $X^\top X\boldsymbol{v} = 0$, you would get another solution to the normal equations (and you can generate infinitely many solutions this way). In this case, (5.9) returns the least squares solution for linear regression such that the Euclidean norm of the weights, $\|\beta\|$,is minimized. The minimum norm solution is often useful in signal processing applications, as the norm squared is often interpretable as some form of energy (which we would like to make small).

In practice, one does not usually compute a linear regression model by implementing either (5.8) or (5.9), but rather uses numerical linear algebra techniques. This functionality is built into many current numerical linear algebra and machine learning packages.

Example: Assume we have a data set $\{(x_i, y_i)\}_{i=1}^N$.

We will consider three models: a linear model of the data with zero intercept, a linear model allowing for a non-zero intercept and a quadratic model using linear regression.

The first model we consider is

$$f(x) = \beta x.$$

This is already in the form of linear regression by taking $\boldsymbol{x}_i = x_i$ and $\boldsymbol{\beta} = \beta$. Our data matrix is $X = [x_1\ x_2\ \ldots\ x_N]^\top$. We can use (5.8) to solve this:

$$\begin{aligned}\beta &= (X^\top X)^{-1} X^\top \boldsymbol{y} \\ &= \frac{\sum_{i=1}^N x_i y_i}{\sum_{i=1}^N x_i^2}.\end{aligned}$$

We now consider a linear model with a possibly non-zero intercept of the form:

$$f(x) = \beta_0 + \beta_1 x. \tag{5.10}$$

To write this in the form $f(\boldsymbol{x}) = \boldsymbol{x}^\top \boldsymbol{\beta}$, we take our feature vectors to be $\boldsymbol{x}_i = \begin{bmatrix} 1 \\ x_i \end{bmatrix}$ and $\boldsymbol{\beta} = \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix}$.

Then, our least squares weights are given by (5.9):

$$\begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix} = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix}^\dagger \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}.$$

Provided that all the x_i's are not the same, the data matrix

$$X = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix}$$

has linearly independent columns and we can use (5.8) to write down the solution as well.

We now consider fitting a quadratic model of the form:

$$f(x) = \beta_0 + \beta_1 x + \beta_2 x^2. \tag{5.11}$$

To write this in the form $f(x) = \boldsymbol{x}^\top \beta$, we take our feature vectors to be $\boldsymbol{x}_i = [1 \; x_i \; x_i^2]^\top$ and $\boldsymbol{\beta} = [\beta_0 \; \beta_1 \; \beta_2]^\top$.

Then, our least squares weights are given by (5.9):

$$\begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \vdots & \vdots & \vdots \\ 1 & x_N & x_N^2 \end{bmatrix}^\dagger \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}.$$

Under similar conditions to the linear model with non-zero intercept allowed, the data matrix

$$X = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \vdots & \vdots & \vdots \\ 1 & x_N & x_N^2 \end{bmatrix}$$

has linearly independent columns and we can use (5.8) to write down the solution as well.

Now, we evaluate the behavior of these three models on a validation dataset of 100 points drawn according to drawn i.i.d. $Y = X^2 + 2X + 1 + \epsilon$ where $\epsilon \sim N(0, 9)$ and the input features are drawn uniformly in the interval $[-10, 10]$. The results are shown in Fig. 5.4. The validation RSS for the zero intercept linear model is 2×10^5, for the linear model with intercept is 9×10^4, and for the quadratic model is 975. As expected the quadratic model is the best for this example.

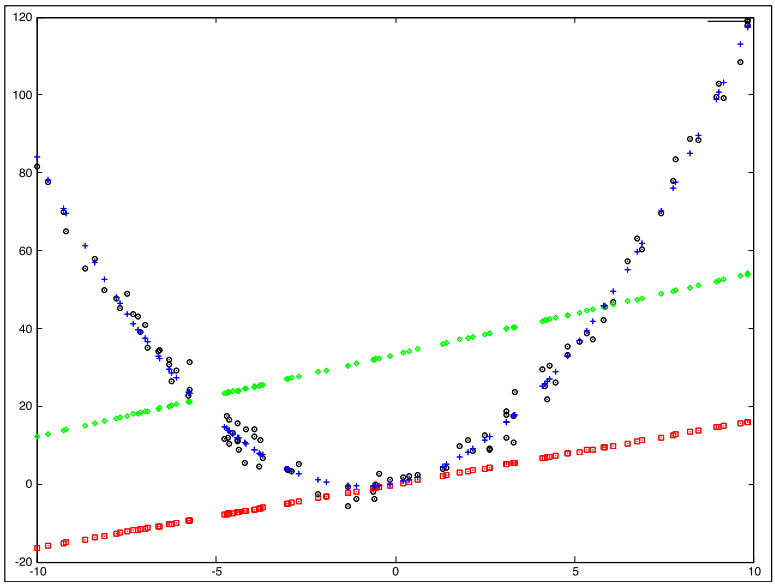


Figure 5.4: Some data drawn from a noisy quadratic and corresponding models: Original data (black), linear with zero intercept (red), linear with intercept (green), and quadratic (blue)

5.3 Model Selection for Linear Regression

A limitation of ordinary least squares as discussed in the previous section is that many of the weights may be non-zero. This creates two problems:

1. Prediction accuracy: LS solutions normally have low bias but high variance. By ignoring some of the features or making their weights small, we may increase the bias a little, but dramatically drop the variance. This can drive our prediction error down considerably.

2. Interpretability: The large weights of the linear regression model can be interpreted as which features we think have a strong influence on the response. Ideally, our model should tell us which features are important and which are not. By deliberately forcing small weights to be small (or zero), we can get a better picture of how the features affect the response.

For example, if we were to predict the temperature in Chambana today based on a snapshot of the temperature around the United States yesterday using a linear regression model, the model would be more accurate if it gave low (or zero) weight to temperatures in far away places, like Alaska or Hawaii or places with odd weather that do not tell us much about the Chambana weather, such as Embarass, MN[4]. Putting large weights on places like Ann Arbor, MI or West Lafayette, IN , which are good indicators of the temperature in Chambana, would make for a good model.

The common approaches for model selection fall into two families: subset selection and shrinkage. By using these techniques, we try to avoid overfitting the data.

A common rule of thumb for parameter tuning for linear regression is to choose the least complex

[4]According to Wikipedia, the annual average temperature is 34.5° F, which is odd, even for the midwest.

model within one standard error of the minimum RSS[5].

5.3.1 Subset Selection

The idea of *subset selection* is simple: Choose a subset of features, throw the rest of the features away and fit your regression model using least squares.

In terms of interpretability, the number of features that go into your model is determined by the subset size. Prediction accuracy depends on the choice of the particular subset.

The simplest method is *best-subset selection*: Try every subset of features up to your desired maximum number of features. The best subset is determined based on minimizing the expected prediction error (which may be done by cross-validation). The main problem with this method is that it has extremely high complexity – if you tried to do this with the maximum number of features being d, you would have to search over 2^d subsets.

Other methods include *forward (backward) stepsize selection*. These start with no (all) features, and add (remove) features one by one greedily based on the feature that improves the expected prediction error the most. These are much faster but may perform poorly when compared to best-subset selection.

Since subset selection either keeps or throws away features, it can lead to high variance leading to little improvement in the prediction error. However, if you have a small number of features, it may be worth trying.

5.3.2 Shrinkage Methods

Shrinkage methods attempt to make less important weights small in a continuous fashion, rather than simply throwing them out as in subset selection. These are also called *regularization methods*. It is important that the features are scaled (*standardized*) so that they each have mean zero and variance 1 (by subtracting the mean, and dividing by the standard deviation). This prevents features that take on large values from overshadowing those that use small coefficients, as a large feature can have a big influence on responses with small weight whereas a small feature requires a large weight to have an appreciable influence on responses. The features also become dimensionless quantities after standardization[6]. We will also assume that the responses have been centered to have mean zero, so that there is no feature corresponding to an intercept in a linear regression model (as it would not make sense to shrink the intercept)[7].

The shrinkage formulation of linear regression is to use the model $f(x) = \boldsymbol{x}^\top \boldsymbol{\beta}_s$ where

$$\boldsymbol{\beta}_s = \arg\min_{\beta} \mathrm{RSS}(\boldsymbol{\beta}) \text{ subject to } r(\boldsymbol{\beta}) \leq t. \tag{5.12}$$

[5]Using the least complex model within one standard error from the minimum predicted loss, rather than the one with minimum predicted loss is not a bad idea in general. Remember that the outputs you are trying to predict are random, so part of the estimate of the minimized loss is due to this randomness.

[6]If x has units u, then its standard deviation and mean also have units u, so the units cancel.

[7]Almost all software packages are able to deal with the intercept properly automatically, so you do not need to deal with it practice. However, you may need to standardize the features on your own.

The non-negative function $r(\boldsymbol{\beta})$ measures the size of the weights, and is often called a *regularizer* or *shrinkage penalty*. Typically, when $\boldsymbol{\beta}$ has large values, $r(\boldsymbol{\beta})$ is large, while small values give a small $r(\boldsymbol{\beta})$. Most sensible regularizers will be zero for the zeros vector and convex (typically a norm or something norm like). The parameter t is a non-negative number that controls the effect of regularization. If t is very large (e.g. $t = \infty$), then you expect to recover the least squares solution. When t is zero, all the weights will be zero.

If the regularizer is taken to be the number of non-zero coordinates in the vector $\boldsymbol{\beta}$, the shrinkage formulation of linear regression reduces to best-subset selection for linear regression with at most t features.

The two most common choices are $r(\boldsymbol{\beta}) = \boldsymbol{\beta}^\top\boldsymbol{\beta}$, which gives rise to *Ridge Regression* and $r(\boldsymbol{\beta}) = \sum_i |\beta_i|$, which gives rise to the *LASSO*.

Ridge Regression

Ridge regression goes under many names – Tikhonov or ℓ^2 regularization (numerical linear algebra, signal processing), penalized regression (statistics and machine learning), weight decay (neural networks) being the most common ones. It was originally derived to handle numerically badly behaved least squares solutions (5.8). Since (5.8) involves inverting the matrix $X^\top X$, if the determinant of this matrix is close to zero, the inverse can lose precision. This can occur if some of the features are highly correlated, among other reasons. Ridge regression uses the linear regression model specified by $\boldsymbol{\beta}_r = \left(X^\top X + \lambda I\right)^{-1} X^\top \boldsymbol{y}$, where λ is a tuning parameter ($\lambda = 0$ corresponding to LS). Increasing λ will decrease the variance but increase the bias. We will come back to how ridge regression works when we discuss the singular value decomposition.

A pictorial diagram of ridge regression is given in Fig. 5.5. The constraint on the weights, $\|\boldsymbol{\beta}\| \leq t$ specifies the weights must lie in some ball centered at the origin (in orange). The blue curves are curves of constant RSS, with the blue star being the LS solution. The ridge regression solution is the point in the blue ball that has the lowest RSS (marked as a green star). Note that the ridge regression solution is where a level curve hits the ball, since the ball does not have a corner.

The ridge regression problem can be explicitly solved by writing the shrinkage problem in a slightly different (but equivalent form [4] for convex r, known as the dual of 5.12):

$$\boldsymbol{\beta}_s = \arg\min_{\beta} \mathrm{RSS}(\boldsymbol{\beta}) + \lambda r(\boldsymbol{\beta}) \tag{5.13}$$

where λ is a parameter that controls the amount of regularization.

Substituting in the definition of RSS and the regularization function, we have

$$\boldsymbol{\beta}_r = \arg\min_{\beta} \|\boldsymbol{y} - X\boldsymbol{\beta}\|^2 + \lambda\|\boldsymbol{\beta}\|^2. \tag{5.14}$$

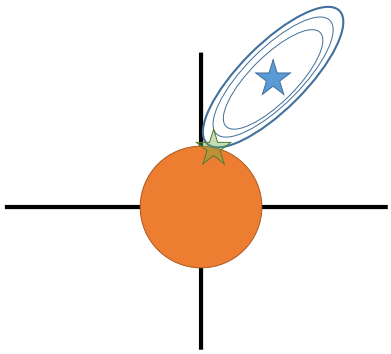


Figure 5.5: Level curves of RSS (blue) and weight constraint for LASSO (orange). Axes are coefficients. The least squares solution is marked as the blue star. The RSS level curve hits the constraint set not quite at a corner, giving the Ridge Regression solution (green star) with all coefficients non-zero, but some are small. Based on Fig. 3.11, [24]. Compare to Fig. 5.6.

By rewriting the quantity being minimized, we see

$$\begin{aligned}\left\| \begin{bmatrix} \boldsymbol{y} \\ \mathbf{0} \end{bmatrix} - \begin{bmatrix} X \\ \sqrt{\lambda} I \end{bmatrix} \boldsymbol{\beta} \right\|^2 &= \left\| \begin{bmatrix} \boldsymbol{y} - X\boldsymbol{\beta} \\ -\sqrt{\lambda}\boldsymbol{\beta} \end{bmatrix} \right\|^2 \\ &= \|\boldsymbol{y} - X\boldsymbol{\beta}\|^2 + \|-\sqrt{\lambda}\boldsymbol{\beta}\|^2 \\ &= \|\boldsymbol{y} - X\boldsymbol{\beta}\|^2 + \lambda\|\boldsymbol{\beta}\|^2\end{aligned}$$

where $\mathbf{0}$ denotes a d-dimensional zero vector, and I denotes the $d \times d$ identity matrix.

Thus, the ridge regression problem is equivalent to

$$\boldsymbol{\beta}_r = \arg\min_{\beta} \left\| \begin{bmatrix} \boldsymbol{y} \\ \mathbf{0} \end{bmatrix} - \begin{bmatrix} X \\ \sqrt{\lambda} I \end{bmatrix} \boldsymbol{\beta} \right\|^2 \tag{5.15}$$

which matches the ordinary least squares problem (5.6), except with the data matrix replaced with $\begin{bmatrix} X \\ \sqrt{\lambda} I \end{bmatrix}$ (which has linearly independent columns for $\lambda > 0$) and the responses replaced with $\begin{bmatrix} \boldsymbol{y} \\ \mathbf{0} \end{bmatrix}$.

Applying our solution to LS (5.8), we see the solution to ridge regression is thus

$$\begin{aligned}\boldsymbol{\beta}_r &= \left(\begin{bmatrix} X \\ \sqrt{\lambda} I \end{bmatrix}^\top \begin{bmatrix} X \\ \sqrt{\lambda} I \end{bmatrix} \right)^{-1} \begin{bmatrix} X \\ \sqrt{\lambda} I \end{bmatrix}^\top \begin{bmatrix} \boldsymbol{y} \\ \mathbf{0} \end{bmatrix} \\ &= \left(X^\top X + \lambda I \right)^{-1} X^\top \boldsymbol{y}\end{aligned} \tag{5.16}$$

The same discussion on computation at the end of Sec. 5.2.1 applies to the ridge regression solution (5.16) as the LS solution (5.8).

The LASSO

LASSO stands for "least absolute shrinkage and selection operator". It also has other names, such as basis pursuit (signal processing, compressive sensing) and ℓ^1 regularization.

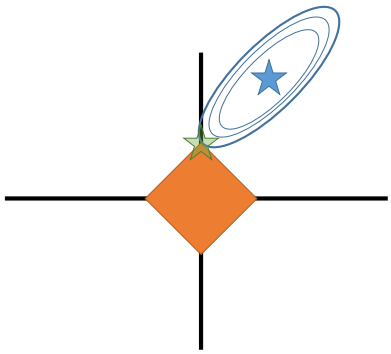


Figure 5.6: Level curves of RSS (blue) and weight constraint for LASSO (orange). Axes are coefficients. The least squares solution is marked as the blue star. The RSS level curve hits the constraint set at a corner, giving the LASSO solution (green star) with some coefficients zero and the others non-zero. Based on Fig. 3.11, [24]. Compare to Fig. 5.5.

While ridge regression shrinks weights, it cannot force a small weight to be replaced with zero. So, if your LS solution had all non-zero weights, the ridge regression solution will also have all non-zero weights (but some will be small). The LASSO minimizes the RSS with a constraint on the weights that $\|\beta\|_1 = \sum_i |\beta_i| \leq t$. The geometry of the solution is given in Fig. 5.6. Since the weight constraint set has a lot of corners (orange), the level curves of the RSS (blue; blue star denoting LS solution) can hit corners exactly, or intersect the constraint set in parts that only have a few weights. This also provides some shrinkage, based on how far into a corner the level curves for minimizing RSS intersect the weight constraint set. Thus, the LASSO regularizer is said to *promote sparsity*, since by setting t (or λ in the dual formulation as in (5.13)) appropriately, only a few weights in the model will be non-zero. Promoting sparsity has made the LASSO extremely useful for high dimensional regression problems, such as compressive sensing[8]. It is essentially the closest convex problem to subset selection, and the convexity makes solving the problem tractable through various algorithms. There is not an explicit solution for LASSO in general, unlike ridge regression or LS.

Example[9]: We consider the problem of predicting the amount of a prostate cancer antigen based on 8 features: log cancer volume (lcavol), log prostate weight (lweight), age, etc. using a linear regression model. The dataset contains 97 datapoints. We select parameters for the LASSO and Ridge regression as well as identify a subset for best subset selection via 5-fold cross-validation. The data was randomly split into a training set (70%) and validation set (30%). The features were standardized, and the responses were centered.

The weights are given in Table 5.1. All feature selection methods outperformed the LS solution on the validation set. We see both the ridge and LS solutions have all non-zero weights, though some are small. Best subset used six features in its model while LASSO used four features to predict the amount of antigen. Since both the LASSO and best subset use fewer features than the

[8] In some cases, when a signal is sparse, we can recover it from datapoints well below what Shannon-Nyquist tells us.

[9] Based on Table 3.3 in [24], and Table 13.1 in [36]. Dataset from `http://statweb.stanford.edu/~tibs/ElemStatLearn/data.html`.

	lcavol	lweigh	age	lbph	svi	lcp	gleason	pgg45	Error
Least Squares	0.66	0.323	-0.166	0.132	0.164	0.03	0.037	0.11	0.528
Best Subset Selection	0.676	0.32	-0.164	0.133	0.178	x	x	0.14	0.511
LASSO	0.59	0.178	x	x	0.075	0.059	x	x	0.506
Ridge Regression	0.514	0.283	-0.107	0.111	0.169	0.114	0.047	0.081	0.504

Table 5.1: Prostate cancer antigen prediction. x denotes a feature that is not used by that model. The error is the mean square error of the predicted responses on the validation set.

LS solution, they are easier to interpret. The LASSO had close to the best error performance as well, making it a reasonable model to use.

It is important to note that when we fit a linear regression model, we are not saying that the features cause the response – simply that we can predict the response from the features. For example, we cannot say that lcavol causes more antigen without additional knowledge.

Chapter 6

Eigendecompositions, Singular Value Decompositions, and Principal Component Analysis

In this chapter, we will focus on the most common unsupervised method of *dimensionality reduction*, Principal Component Analysis (PCA). PCA is also sometimes referred to as the Karhunen-Loève Transform (KLT)[1]. The key idea for dimensionality reduction is: while features may live in a high dimensional space, there is often an (approximately) low-dimensional structure to the data. By using this low-dimensional structure, we can create new features that approximately describe the data set in a much smaller space. This allows for more succinct descriptions of the data set (e.g. it is easier to visualize and interpret a 16-dimensional feature vector with a series of scatter plots versus a 10^6-dimensional feature vector), faster and simpler algorithms (due to the smaller number of features) and reduces the curse of dimensionality[2]. For example, a photograph may be several megapixels. One way to represent the image would be to take each pixel as a feature (leading to millions of features). However, if one transforms it using the Discrete Cosine Transform (DCT) and retains only the low frequencies, one can get a decent approximation of the original photograph with far fewer features (hundreds or thousands of DCT coefficients)[3].

Example: Assume you are buying a car. You walk into the dealership and the salesperson hits you with a storm of the features associated with each car: the price, the peak horsepower, the NHSTA safety ratings (overall, front crash, side crash, rollover), how many seats it has, how many gears it has, if its automatic or manual, and all the types of fancy expensive options you can get (there are hundreds of these – floor mats sunroof, fog lights, air conditioning, rear view camera, WiFi, GPS, etc.). As a poor student who can only drive an automatic car, you only care that the car gets you from point A to point B safely and cheaply. Therefore most of these features

[1] Many other names have been associated with this idea at some point – Kosambi, Hotelling, Pearson, etc.

[2] Many learning algorithms break down with high-dimensional feature spaces. This is extremely common for things like kNN. See Sec. 2.5 in [24] for more details.

[3] In fact, the DCT closely approximates what PCA does for many images (but in an image independent way) with fast running time, hence its use in compression algorithms like JPEG (see, e.g., Section 8.2 in [21]).

are just *noise* to you. Since you were paying attention in this class, you realize you can apply dimensionality reduction. Thus, whenever a salesperson shows you a car, you listen about the price and the NHSTA safety ratings and if the car is automatic or manual. For the rest of their spiel, you stick your fingers in your ears. Thus, you reduce the dimensionality from hundreds of features to only 6: price, overall crash safety, front crash safety, side crash safety, rollover safety and if the car is automatic. You ignore the features you do not care about, such as floor mats, if it has GPS or the horsepower. Of course, you lose some information in this approach of throwing out features – other features may carry some useful information on the decision you need to make (and maybe, by considering combinations of features, you could do better). However, the essential information is contained in the features you listened to.

6.1 Eigendecompositions

Eigendecompositions of symmetric matrices forms the mathematical basis for PCA. Here we provide an overview of eigendecomposition; more details can be found in introductory linear algebra books [20, 26, 41].

In analog signal processing (ECE210), it was shown that if you put a complex sinusoid ($e^{j\omega t}$) through a linear time-invariant (LTI) system, a scaled version of the same complex sinusoid comes out. Complex sinusoids are called *eigenfunctions* of LTI systems, and the scaling factors are called the *eigenvalues*. By looking at the response of the system to different complex sinusoids, one can characterize what the LTI system does.

In this section, we will be discussing the idea of eigenvectors and eigenvalues, which have a similar function for square matrices as complex sinusoids do for LTI systems – by studying the eigenvectors and eigenvalues, we can determine what the matrix does to vectors. In this class, we will only need to consider eigendecompositions of symmetric matrices (i.e. matrices who are their own transposes), and therefore we will focus on this case for our presentation. The eigendecomposition of a symmetric matrix n-by$-n$ gives us a new set of coordinates for $\mathbb{R}^n$ such that the matrix behaves like a diagonal matrix in these coordinates.

Let A be a n-by-n square matrix. A non-zero vector $\boldsymbol{u}$ is an *eigenvector* of A with *eigenvalue* λ if

$$A\boldsymbol{u} = \lambda\boldsymbol{u}. \tag{6.1}$$

When a matrix is applied to an eigenvector, it simply scales it. Rearranging this, we see

$$(A - \lambda I)\boldsymbol{u} = \mathbf{0}. \tag{6.2}$$

This implies that if A has eigenvalue λ, then the matrix $A - \lambda I$ is not invertible (otherwise, we would have $\boldsymbol{u} = 0$, which would be a contradiction), i.e.,

$$\det(A - \lambda I) = 0. \tag{6.3}$$

The expression $\det(A - \lambda I)$ is known as the *characteristic polynomial* of A, and has degree n. By solving the *characteristic equation* (6.3) we can find all of the eigenvalues of A. In general, there will be at most n distinct eigenvalues of A (some may be complex valued). An eigenvalue can have multiplicity greater than one, in which case we will have fewer than n distinct eigenvalues

(see Example 2 below). Once we have an eigenvalue, we can find the eigenvectors corresponding to that eigenvalue using (6.2). From the definition of an eigenvector, we can see that the sum of eigenvectors corresponding to the same eigenvalue is also an eigenvector for that eigenvalue, and a scaled version of that eigenvector is still an eigenvector for that eigenvalue.

Example 1: Consider the matrix

$$A = \begin{bmatrix} 5 & 1 \\ 1 & 5 \end{bmatrix}.$$

The characteristic polynomial is given by

$$\begin{aligned} \det(A - \lambda I) &= \begin{vmatrix} 5-\lambda & 1 \\ 1 & 5-\lambda \end{vmatrix} \\ &= (5-\lambda)^2 - 1. \end{aligned}$$

To find the eigenvalues, we solve the characteristic equation (characteristic polynomial=0). By the quadratic formula, we get that the eigenvalues are $\lambda_1 = 6, \lambda_2 = 4$.

To find the eigenvectors corresponding to the eigenvalue $\lambda_1 = 6$, we solve $(A - 6I)\boldsymbol{u}_1 = 0$, i.e.,

$$\begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} u_{11} \\ u_{12} \end{bmatrix} = \mathbf{0}$$

This gives us $u_{11} = u_{12}$, i.e.,

$$\boldsymbol{u}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

and all other solutions are scaled versions of $\boldsymbol{u}_1$ (for example, $\boldsymbol{u}_1 = [-1 - 1]^\top$ is another eigenvector). We may scale $\boldsymbol{u}_1$ to have unit norm, to get the normalized eigenvector

$$\boldsymbol{u}_1 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

To find the eigenvectors corresponding to the eigenvalue $\lambda_2 = 4$, we solve $(A - 4I)\boldsymbol{u}_2 = 0$. i.e.,

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} u_{21} \\ u_{22} \end{bmatrix} = \mathbf{0}.$$

This gives us $u_{21} = -u_{22}$, i.e.,

$$\boldsymbol{u}_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$$

and all other eigenvectors corresponding to λ_2 are scaled versions of $\boldsymbol{u}_2$. The normalized eigenvector

$$\boldsymbol{u}_2 = \frac{1}{\sqrt{2}} \begin{bmatrix} -1 \\ 1 \end{bmatrix}.$$

Notice that the eigenvectors $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$ are orthogonal to each other, i.e., $\boldsymbol{u}_1^\top \boldsymbol{u}_2 = 0$. The normalized eigenvectors $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$, are *orthonormal*, i.e., orthogonal and unit norm.

Example 2: Consider the matrix

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix}.$$

The characteristic polynomial is given by

$$\begin{aligned} \det(A - \lambda I) &= \begin{vmatrix} 1-\lambda & 0 & 0 \\ 0 & 2-\lambda & -1 \\ 0 & -1 & 2-\lambda \end{vmatrix} \\ &= (1-\lambda)(\lambda^2 - 4\lambda + 3) = -(\lambda - 1)^2(\lambda - 3). \end{aligned}$$

This means that there are two distinct eigenvalues, $\lambda_1 = 3$, and $\lambda_2 = \lambda_3 = 1$, i.e., the eigenvalue of 1 has multiplicity 2.

To find the eigenvectors corresponding to the eigenvalue $\lambda_1 = 3$, we solve $(A - 3I)\boldsymbol{u}_1 = 0$, i.e.,

$$\begin{bmatrix} 2 & 0 & 0 \\ 0 & -1 & -1 \\ 0 & -1 & -1 \end{bmatrix} \begin{bmatrix} u_{11} \\ u_{12} \\ u_{13} \end{bmatrix} = \mathbf{0}$$

This gives us $u_{11} = 0$ and $u_{12} + u_{13} = 0$, i.e.,

$$\boldsymbol{u}_1 = \begin{bmatrix} 0 \\ 1 \\ -1 \end{bmatrix}$$

and its scaled versions. The corresponding unit norm eigenvector is given by:

$$\boldsymbol{u}_1 = \frac{1}{\sqrt{2}} \begin{bmatrix} 0 \\ 1 \\ -1 \end{bmatrix}.$$

To find the eigenvectors corresponding to the eigenvalue $\lambda_2 = \lambda_3 = 1$, we solve $(A - I)\boldsymbol{u}_2 = 0$, i.e.,

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} u_{21} \\ u_{22} \\ u_{23} \end{bmatrix} = \mathbf{0}.$$

This gives us $u_{22} - u_{23} = 0$, but u_{21} can be freely chosen. Identical equations of course hold for $\boldsymbol{u}_3$. Given the flexibility in choosing the first component of the vectors $\boldsymbol{u}_2$ and $\boldsymbol{u}_3$, we can make $\boldsymbol{u}_2$ and $\boldsymbol{u}_3$ orthogonal to each other (and to $\boldsymbol{u}_1$) by choosing

$$\boldsymbol{u}_2 = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} \qquad \boldsymbol{u}_3 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}.$$

Normalizing we get

$$\boldsymbol{u}_2 = \frac{1}{\sqrt{2}} \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} \qquad \boldsymbol{u}_3 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}.$$

The properties of the eigenvalues and eigenvectors seen in the above two examples can be shown to hold more generally. In particular, for any symmetric square matrix A,

1. The eigenvalues and eigenvectors are real. (Proof: See [41])
2. If $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$ are eigenvectors corresponding to two different eigenvalues λ_1 and λ_2, then the eigenvectors are *orthogonal* $\boldsymbol{u}_1^\top \boldsymbol{u}_2 = 0$.

   Proof: Apply the fact that $A\boldsymbol{u}_1 = \lambda_1 \boldsymbol{u}_1$ and $A\boldsymbol{u}_2 = \lambda_2 \boldsymbol{u}_2$:

   $$\lambda_1 \boldsymbol{u}_1^\top \boldsymbol{u}_2 = (\lambda_1 \boldsymbol{u}_1)^\top \boldsymbol{u}_2 = (A\boldsymbol{u}_1)^\top \boldsymbol{u}_2 = \boldsymbol{u}_1^\top A^\top \boldsymbol{u}_2 = \boldsymbol{u}_1^\top A \boldsymbol{u}_2 = \boldsymbol{u}_1^\top \lambda_2 \boldsymbol{u}_2 = \lambda_2 \boldsymbol{u}_1^\top \boldsymbol{u}_2.$$

   Since $\lambda_1 \neq \lambda_2$, we must have $\boldsymbol{u}_1^\top \boldsymbol{u}_2 = 0$.
3. If an eigenvalue λ has multiplicity k, then we can find k mutually orthogonal eigenvectors for that eigenvalue. (Proof: see [26].)

Let us order the eigenvalues of A in decreasing order: $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n$. By the facts stated, we can find corresponding eigenvectors $\boldsymbol{u}_1, \ldots, \boldsymbol{u}_n$ that are *orthonormal*.

The big result of this section is that the symmetric matrix A can be written in terms of its eigenvectors and eigenvalues in a nice way:

$$A = \sum_{i=1}^{n} \lambda_i \boldsymbol{u}_i \boldsymbol{u}_i^\top. \tag{6.4}$$

This is known as the *eigendecomposition* or *spectral decomposition* of A.

Example 1: (continued) Consider the matrix

$$A = \begin{bmatrix} 5 & 1 \\ 1 & 5 \end{bmatrix}.$$

We saw earlier that the eigenvalues are $\lambda_1 = 6$, and $\lambda_2 = 4$, and the corresponding normalized eigenvectors are:

$$\boldsymbol{u}_1 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad \text{and} \quad \boldsymbol{u}_2 = \frac{1}{\sqrt{2}} \begin{bmatrix} -1 \\ 1 \end{bmatrix}.$$

We can check the eigendecomposition formula (6.4) as follows:

$$\begin{aligned} \lambda_1 \boldsymbol{u}_1 \boldsymbol{u}_1^\top + \lambda_2 \boldsymbol{u}_2 \boldsymbol{u}_2^\top &= 6 \left(\frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right) \left(\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \end{bmatrix} \right) + 4 \left(\frac{1}{\sqrt{2}} \begin{bmatrix} -1 \\ 1 \end{bmatrix} \right) \left(\frac{1}{\sqrt{2}} \begin{bmatrix} -1 & 1 \end{bmatrix} \right) \\ &= 3 \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} + 2 \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} = \begin{bmatrix} 5 & 1 \\ 1 & 5 \end{bmatrix} = A. \end{aligned}$$

Example 2: (continued) Consider the matrix

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix}.$$

We saw that the eigenvalues are $\lambda_1 = 3$, and $\lambda_2 = \lambda_3 = 1$, and the corresponding eigenvectors are:

$$\boldsymbol{u}_1 = \frac{1}{\sqrt{2}} \begin{bmatrix} 0 \\ 1 \\ -1 \end{bmatrix} \qquad \boldsymbol{u}_2 = \frac{1}{\sqrt{2}} \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} \qquad \boldsymbol{u}_3 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}.$$

We can again check the eigendecomposition formula (6.4) as follows:

$$\begin{aligned} \sum_{i=1}^{3} \lambda_i \boldsymbol{u}_i \boldsymbol{u}_i^\top &= 3\frac{1}{2} \begin{bmatrix} 0 \\ 1 \\ -1 \end{bmatrix} \begin{bmatrix} 0 & 1 & -1 \end{bmatrix} + \frac{1}{2} \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \\ &= \frac{3}{2} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & -1 \\ 0 & -1 & 1 \end{bmatrix} + \frac{1}{2} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix} = A. \end{aligned}$$

The form of the eigendecomposition given by (6.4) is often called the outer-product form the eigendecomposition. In many cases, we work with a matrix representation of the eigendecomposition. Let

$$U = [\boldsymbol{u}_1 \, \boldsymbol{u}_2 \ldots \boldsymbol{u}_n]$$

and D be the n-by-n diagonal matrix with diagonal entries $\lambda_1, \ldots, \lambda_n$, i.e.,

$$D = \begin{bmatrix} \lambda_1 & 0 & \ldots & 0 \\ 0 & \lambda_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \ldots & 0 & \lambda_n \end{bmatrix}.$$

Then, the eigendecomposition of A can be written as

$$A = UDU^\top. \tag{6.5}$$

Note that

$$U^\top = \begin{bmatrix} \boldsymbol{u}_1^\top \\ \boldsymbol{u}_2^\top \\ \vdots \\ \boldsymbol{u}_n^\top \end{bmatrix}$$

and that $U^\top U = I$, i.e., $U^{-1} = U^\top$. A square matrix with orthonormal columns is called an *orthogonal or unitary* matrix. Geometrically, it rotates and/or reflects a vector (or equivalently, the axes). The transpose of an orthogonal matrix is its inverse (and this is also an orthogonal matrix). We can invert the relationship given in (6.5) and write

$$D = U^\top AU.$$

Thus eigendecomposition can also be regarded as a way to diagonalize the matrix A.

Most computer packages will output the eigendecomposition in the form given in (6.5), by giving you a vector containing the eigenvalues and the matrix U.

Example 2: (continued) For this example, discussed above:

$$U = \begin{bmatrix} 0 & 1 & 0 \\ \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \end{bmatrix}, \qquad D = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

and it is easy to check that

$$UDU^\top = \begin{bmatrix} 0 & 1 & 0 \\ \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \end{bmatrix} \begin{bmatrix} 3 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ 1 & 0 & 0 \\ 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix} = A$$

6.1.1 The Minimax Principle

The Courant-Fischer minimax principle gives us a useful characterization of the eigenvalues and eigenvectors of a symmetric matrix. A heuristic explanation is given in [41] while the full mathematical details are given Section 4.2 of [27].

The corollary of the minimax principle[4] that we need is: Let A be a symmetric matrix with eigenvalues $\lambda_1 \geq \lambda_2 \geq \ldots$ and corresponding orthonormal eigenvectors $\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots$.

Then,

$$\lambda_1 = \max_{\boldsymbol{z}:\|\boldsymbol{z}\|=1} \boldsymbol{z}^\top A \boldsymbol{z} \tag{6.6}$$

and this maximum is achieved by $\boldsymbol{z} = \boldsymbol{u}_1$.

Also,

$$\lambda_i = \max_{\boldsymbol{x}:\|\boldsymbol{z}\|=1, \boldsymbol{z} \text{is orthogonal to } \boldsymbol{u}_1,\ldots,\boldsymbol{u}_{i-1}} \boldsymbol{z}^\top A \boldsymbol{z} \tag{6.7}$$

and this maximum is achieved by $\boldsymbol{z} = \boldsymbol{u}_i$.

These results are proved by manipulating the eigendecomposition of A.

When we discuss PCA, A will be the covariance matrix of the data, and the PCA features will be given by writing the data in terms of the coordinates provided by the eigenvectors of the covariance matrix. In turn, by this principle, we will see that the first PCA feature captures most of the variance of the data, while the second captures the most variance of the data in a direction orthogonal to the first feature, and so on.

[4] Actually, this is a statement equivalent to the minimax principle.

6.2 Singular Value Decompositions

While the eigendecomposition is one of the most powerful tools in your arsenal, it is not applicable to non-square matrices. A generalization of it, the singular value decomposition, is applicable to all matrices (even non-square ones). When we discuss PCA, we will use it to increase numerical stability over the straightforward eigendecomposition approach, by relating the singular value decomposition with the eigendecomposition of certain matrices. A more mathematical view can be found in many linear algebra books (see, e.g., [20, 41]).

Let A be an m-by-n matrix. Then, the singular value decomposition of A is given by

$$A = USV^\top \tag{6.8}$$

where U is a m-by-m matrix with orthonormal columns called the *left singular vectors*, S is a m-by-n matrix with non-negative values

$$\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_{\min(m,n)} \geq 0$$

on the leading diagonal called the *singular values* and V is a n-by-n matrix with orthonormal columns called the *right singular vectors*. The matrices U and V have the property that their transpose is their inverse.

The SVD can be considered to be a diagonalization of the matrix A, in that the orthogonality of U and V implies that

$$S = U^\top AV$$

If $m < n$ (i.e., the matrix A is fat), then

$$A = \begin{bmatrix} & & \\ \boldsymbol{u}_1 & \boldsymbol{u}_2 & \ldots \boldsymbol{u}_m \\ & & \end{bmatrix} \begin{bmatrix} \sigma_1 & 0 & \ldots & \ldots & \ldots & \ldots & 0 \\ 0 & \sigma_2 & 0 & \ldots & \ldots & \ldots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ldots & \ldots & 0 \\ 0 & \ldots & 0 & \sigma_m & 0 & \ldots & 0 \end{bmatrix} \begin{bmatrix} & \boldsymbol{v}_1^\top & \\ & \boldsymbol{v}_2^\top & \\ & \vdots & \\ & \vdots & \\ & \boldsymbol{v}_n^\top & \end{bmatrix} \tag{6.9}$$

and if $m > n$ (i.e., the matrix A is tall), then

$$A = \begin{bmatrix} & & & \\ \boldsymbol{u}_1 & \boldsymbol{u}_2 & \ldots & \boldsymbol{u}_m \\ & & & \end{bmatrix} \begin{bmatrix} \sigma_1 & 0 & \ldots & 0 \\ 0 & \sigma_2 & \ddots & 0 \\ \vdots & \ddots & \ldots & 0 \\ 0 & \ldots & 0 & \sigma_n \\ 0 & \ldots & \ldots & 0 \\ \vdots & \ldots & \ldots & \vdots \\ 0 & \ldots & \ldots & 0 \end{bmatrix} \begin{bmatrix} & \boldsymbol{v}_1^\top & \\ & \boldsymbol{v}_2^\top & \\ & \vdots & \\ & \boldsymbol{v}_n^\top & \end{bmatrix} \tag{6.10}$$

The matrices U, V come from the eigendecompositions of the matrices $AA^\top$ and $A^\top A$:

$$AA^\top = UDU^\top \tag{6.11}$$

$$A^\top A = V\tilde{D}V^\top. \tag{6.12}$$

Moreover, the non-zero entries of D and $\tilde{D}$ are the same, and these are the squares of the singular values. In particular, if $m < n$,

$$D = \begin{bmatrix} \sigma_1^2 & 0 & \dots & 0 \\ 0 & \sigma_2^2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & \sigma_m^2 \end{bmatrix}, \qquad \tilde{D} = \begin{bmatrix} \sigma_1^2 & 0 & \dots & \dots & \dots & \dots & 0 \\ 0 & \sigma_2^2 & \ddots & \dots & \dots & \dots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \dots & \dots & \vdots \\ \vdots & \dots & \ddots & \sigma_m^2 & \ddots & \dots & \vdots \\ \vdots & \dots & \dots & \ddots & 0 & \ddots & \vdots \\ \vdots & \dots & \dots & \dots & \ddots & \ddots & 0 \\ 0 & \dots & \dots & \dots & \dots & 0 & 0 \end{bmatrix}, \tag{6.13}$$

and if $m > n$, the forms of D and $\tilde{D}$ are swapped.

We can also write the SVD in an outer product form. Let $\boldsymbol{u}_1, \dots, \boldsymbol{u}_m$ be the left singular vectors and $\boldsymbol{v}_1, \dots, \boldsymbol{v}_n$ be the right singular vectors. Then,

$$A = \sum_{i=1}^{\min(m,n)} \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^\top. \tag{6.14}$$

Example 3: Consider the matrix:

$$A = \begin{bmatrix} 3 & 2 & 2 \\ 2 & 3 & -2 \end{bmatrix}.$$

It is easy to see that

$$AA^\top = \begin{bmatrix} 17 & 8 \\ 8 & 17 \end{bmatrix}.$$

The characteristic function for this (symmetric) matrix is:

$$\begin{vmatrix} 17-\lambda & 8 \\ 8 & 17-\lambda \end{vmatrix} = 0 \implies \lambda^2 - 34\lambda + 225 = 0 \implies (\lambda - 25)(\lambda - 9) = 0.$$

This means that the eigenvalues of $AA^\top$ are $\lambda_1 = 25$ and $\lambda_2 = 9$, which further implies that the singular values are

$$\sigma_1 = 5, \quad \sigma_2 = 3.$$

To find the U matrix, we need find the normalized eigenvectors of $AA^\top$, which can easily be computed to be

$$\boldsymbol{u}_1 = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad \boldsymbol{u}_2 = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ -1 \end{bmatrix}.$$

To find V, we need to do an eigendecomposition of the matrix

$$A^\top A = \begin{bmatrix} 13 & 12 & 2 \\ 12 & 13 & -2 \\ 2 & -2 & 8 \end{bmatrix}$$

It is easy to check that the eigenvalues of this matrix are 25, 9, and 0, which is as expected from (6.13). The eigenvectors corresponding to these eigenvalues can be shown to be:

$$\boldsymbol{v}_1 = \frac{1}{\sqrt{2}}\begin{bmatrix}1\\1\\0\end{bmatrix}, \quad \boldsymbol{v}_2 = \frac{1}{3\sqrt{2}}\begin{bmatrix}1\\-1\\4\end{bmatrix}, \quad \boldsymbol{v}_3 = \frac{1}{3}\begin{bmatrix}2\\-2\\-1\end{bmatrix}.$$

Thus the SVD of A can be written as:

$$\begin{bmatrix}3 & 2 & 2\\ 2 & 3 & -2\end{bmatrix} = \begin{bmatrix}\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}}\\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}}\end{bmatrix}\begin{bmatrix}5 & 0 & 0\\ 0 & 3 & 0\end{bmatrix}\begin{bmatrix}\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0\\ \frac{1}{3\sqrt{2}} & -\frac{1}{3\sqrt{2}} & \frac{4}{3\sqrt{2}}\\ \frac{2}{3} & -\frac{2}{3} & -\frac{1}{3}\end{bmatrix}.$$

It should be noted that the eigenvector $\boldsymbol{v}_3$, the one corresponding to the eigenvalue of 0, does not play a role in the SVD since it is zeroed out by the singular value matrix S in the expansion. In particular, we could have written the SVD more compactly (also see discussion below) as:

$$\begin{bmatrix}3 & 2 & 2\\ 2 & 3 & -2\end{bmatrix} = \begin{bmatrix}\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}}\\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}}\end{bmatrix}\begin{bmatrix}5 & 0\\ 0 & 3\end{bmatrix}\begin{bmatrix}\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0\\ \frac{1}{3\sqrt{2}} & -\frac{1}{3\sqrt{2}} & \frac{4}{3\sqrt{2}}\end{bmatrix}.$$

The SVD we have presented in (6.9) and (6.10) is sometimes known as the *full SVD*. Many computational packages are able to compute the *thin SVD* or *economy SVD*, which for a m-by-n matrix only computes the first m right singular vectors for a fat matrix ($m > n$), since the rest of the right singular vectors must correspond to a zero singular value. In this case S is a square matrix. A similar statement holds for tall matrices. For computational purposes, the economy SVD is often better. However, to present how algorithms work with the SVD, it is often better to use the full SVD.

6.3 Applications of the SVD

There are many applications of the SVD.

6.3.1 Low-Rank Approximation, Denoising and Compression

One of the most common uses of the SVD is to approximate a matrix in a *low-rank* sense. The rank of a matrix is the number of non-zero singular values (or equivalently, the number of linearly independent columns). Let us say that A is a $m \times n$ matrix. A *rank*-k approximation[5] of A uses the first k singular vectors and singular values as:

$$\hat{A} = \sum_{i=1}^{k} \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^\top. \tag{6.15}$$

If there are only k singular values that are large, then the singular vectors not used will have a small contribution to how A behaves, and $\hat{A}$ will be a good approximation of A (in fact, it is the

[5]There are other ways to do a rank-k approximation.

best approximation in a well defined sense via a result known as the Eckart-Young-Mirsky theorem [19]). Typically k will be much smaller than the rank of A.

Low-rank approximations are useful since they are often easier to store and use in computations than the original matrix[6], robust to noise, and the lower number of singular vectors lend themselves to easier interpretation in some cases (e.g., in PCA). In many cases, a matrix that is calculated from data does have a low rank structure, but is of full rank due to noise when collecting the data. By using an appropriate low-rank approximation, we denoise the data (such as in the K-SVD algorithm for image denoising [1], which combines many of the tools from this class). The idea of low rank approximation is what makes PCA work as well, which we will see in the next section.

Example: In Fig. 6.1, we once again have a picture of Larry which is 630x420, which we can view as a matrix with 420 rows and 630 columns. There are 264600 values in the matrix and it has rank 420. One use of the SVD is data compression. A rank-k approximation uses $420k + 630k + k = 1051k$ values (left singular vectors have length 420, right singular vectors have length 630 and there are k singular values). Using a rank 5 approximation ($\approx 1\%$ of the uncompressed image coefficients), we do not see much. However, with a rank 10 or 15 approximation ($\approx 2 - 3\%$ of the coefficients), it is clear that a cat is present. With a rank 20 approximation, it is fairly clear that the picture is indeed of Larry. One can compare the results to the vector quantization example in Fig. 4.5. This low-rank approximation gives global distortion, which can be ameliorated by partitioning the image into blocks and applying a low-rank approximation [7].

6.3.2 Linear Regression via the SVD

In the previous chapter, we studied linear regression. We saw that if we wanted to fit a model $f(\boldsymbol{x}) = \boldsymbol{x}^\top \boldsymbol{\beta}$ where the responses were given in a vector $\boldsymbol{y}$ and the input features were rows of the data matrix X that the least squares solution was given by

$$\boldsymbol{\beta}_{LS} = X^\dagger \boldsymbol{y} \tag{6.16}$$

where $X^\dagger$ is known as the *pseudoinverse*. The pseudoinverse is elegantly described in terms of the SVD as

$$X^\dagger = V S^{-1} U^\top \tag{6.17}$$

where S^{-1} is a diagonal matrix whose non-zero entries consist of the reciprocal of the non-zero singular values $(\sigma_1^{-1}, \sigma_2^{-1}, \ldots)$.

[6] The computational complexity needs some care dependent on the application; the approximation given in (6.15) will yield a dense matrix for $\hat{A}$, typically, even if A is a sparse matrix (which affords fast matrix multiplication for A but not $\hat{A}$ directly, for example).

[7] This is essentially the idea of JPEG, which partitions an image into blocks (like we did for vector quantization), transforms them via a Discrete Cosine Transform (which can be viewed in relation to PCA and the SVD), and then throws away higher frequency components to give a low rank approximation of each block. Details are given in [21]. This distortion is also related to the Gibbs Phenomenon [5].

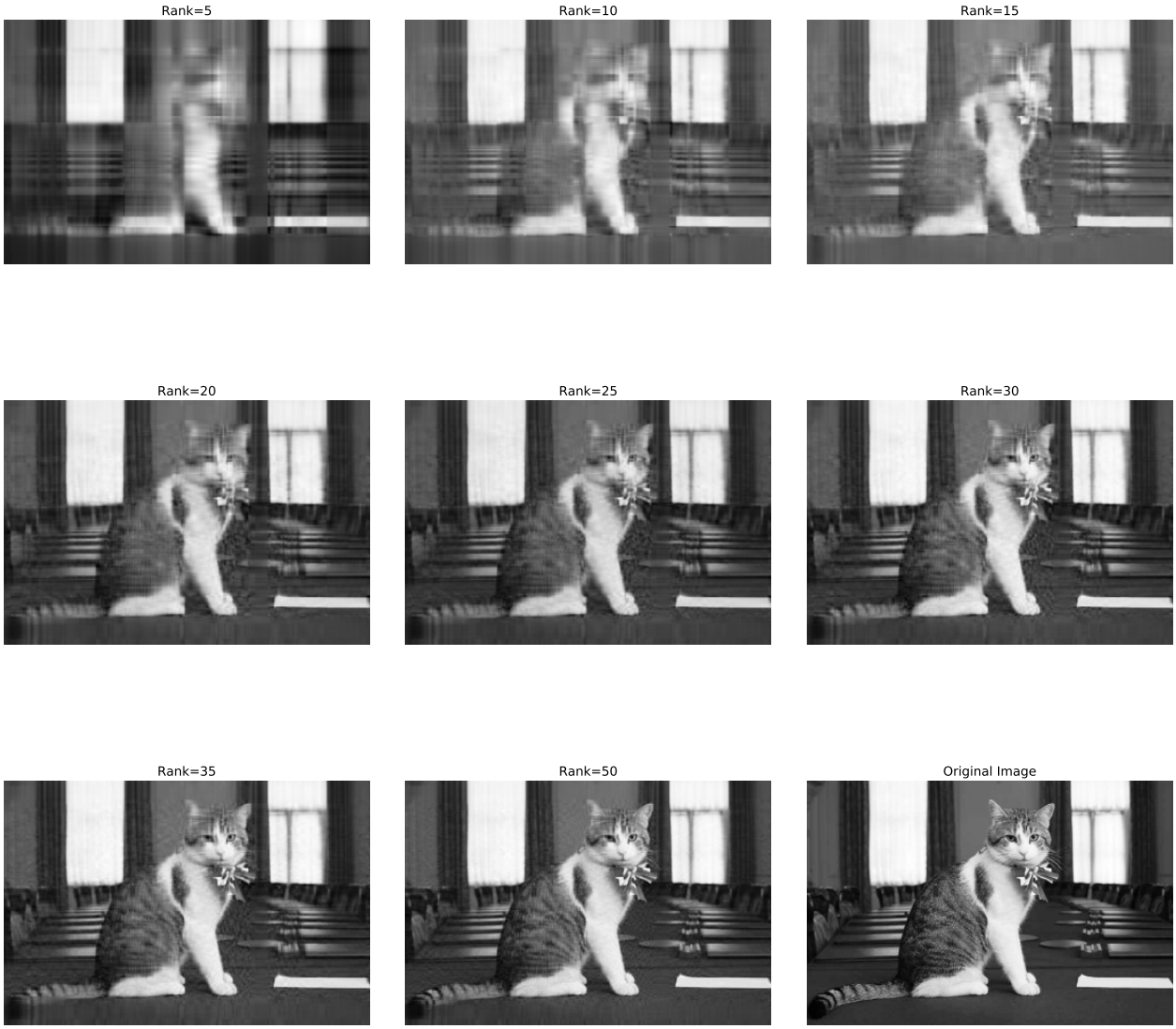


Figure 6.1: Approximating a 630 x 420 image in a low-rank manner.

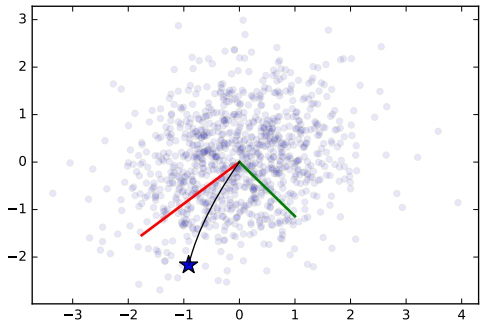


Figure 6.2: A Linear Regression Problem: The feature vectors are plotted in blue (responses are not shown), with the principal components marked in red and green. The least squares solution is shown by a star. The ridge regression solutions (by varying λ) are shown as the black curve. Note that the ridge regression solution is shrunk based on the principal directions.

The SVD also allows us to link the ridge regression solution to the least squares solution. Recall that the ridge regression solution was given as

$$\boldsymbol{\beta}_r = (X^\top X + \lambda I)^{-1} X^\top \boldsymbol{y}. \tag{6.18}$$

Substituting in $X = USV^\top$ into the ridge regression solution, we see

$$\boldsymbol{\beta}_r = V\tilde{S}^{-1}U^\top \boldsymbol{y} \tag{6.19}$$

where $\tilde{S}^{-1}$ is a diagonal matrix with $\frac{\sigma_i}{\sigma_i^2+\lambda}$ as the ith entry on the diagonal. For large values of σ_i, this is slightly smaller than $\frac{1}{\sigma}$ whereas for small values of σ this is approximately 0. Thus, the ridge regression solution takes the pseudoinverse and shrinks its behavior based on the singular values – more strong shrinkage for directions that have small singular values than those that have large singular values. It turns out that the directions with large singular values (in feature space) are the ones in which the data has highest variance, so the ridge regression solution reduces the emphasis on low variance parts directions in the data in the least squares solution. These directions are determined by the columns of V, and are called the *principal components*, which leads us nicely to our next topic, Principal Component Analysis. The shrinking is given pictorially in Fig. 6.2.

Unfortunately, the nature of the LASSO problem does not give it a nice interpretation in terms of the SVD.

6.4 Principal Component Analysis

We now discuss principal component analysis (PCA). We will assume that we have a data set $\{\boldsymbol{x}_i\}_{i=1}^N$ drawn i.i.d. from distribution $p(\boldsymbol{x})$ where the features have mean zero (if they do not, we may simply subtract the mean) and covariance matrix C. This is an unsupervised method.

In practice, it is good to standardize the features to have unit variance for similar reasons as

linear regression – certain features may unduly influence PCA simply because of their measurement scale.

Our goal in PCA is to find an orthogonal matrix (square matrix with orthonormal rows) W of the features to a new set of features that are uncorrelated and the first k features capture as much of the variance in the data as possible for each possible k. The matrix W will give us a new set of coordinates for the data such that the data is uncorrelated between the coordinates. We map the old features, $\boldsymbol{x}$ to the new features, $\tilde{\boldsymbol{x}}$, by the relationship

$$\tilde{\boldsymbol{x}} = W\boldsymbol{x}. \tag{6.20}$$

Let $\boldsymbol{x} \sim p(\boldsymbol{x})$. Mathematically, we want to find an orthogonal matrix W such that

$$\text{cov}(\tilde{\boldsymbol{x}}) = \text{cov}(W\boldsymbol{x}) = \text{E}[(W\boldsymbol{x})(W\boldsymbol{x})^\top] \tag{6.21}$$

is diagonal (so the components of $W\boldsymbol{x}$ are uncorrelated with each other. By writing $\text{cov}(W\boldsymbol{x})$ in terms of C:

$$\begin{aligned}\text{E}[(W\boldsymbol{x})(W\boldsymbol{x})^\top] &= \text{E}[W\boldsymbol{x}\boldsymbol{x}^\top W^\top] \\ &= W\ \text{E}[\boldsymbol{x}\boldsymbol{x}^\top]W^\top \\ &= W\mathsf{C}W^\top\end{aligned}$$

we are reminded of the eigendecomposition. Recall that the transpose of an orthogonal matrix is its inverse, which is also an orthogonal matrix. Then, we have

$$\mathsf{C} = W^\top \text{E}[(W\boldsymbol{x})(W\boldsymbol{x})^\top]W. \tag{6.22}$$

Comparing this to the eigendecomposition of C (which is a symmetric matrix with non-negative eigenvalues, i.e. *positive semi-definite*),

$$\mathsf{C} = UDU^\top \tag{6.23}$$

we can guess (correctly!) that $W = U^\top$. Recall that in our definition of the eigendecomposition, we assumed that the eigenvalues of C, which are the diagonal entries of D, $\lambda_1, \lambda_2, \ldots$ were in non-increasing order, and the columns of U are the corresponding eigenvectors $\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots$

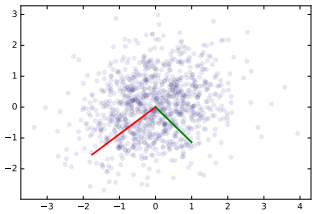
Thus, if we multiply the input vectors by $U^\top$ where U is a matrix with an orthonormal set of eigenvectors of C, we get a new set of features where the data is uncorrelated. In other words, the ith PCA feature is the projection of the data onto the eigenvector corresponding to the ith largest eigenvalue of the covariance matrix. The vector $\boldsymbol{u}_i$ is known as the ith *principal component*[8].

Now, let us look at the meaning of D. Let the total variance of the data in the original features be $\mathsf{C}_{11} + \mathsf{C}_{22} + \ldots$ (i.e. the sum of the variances of each feature). Since the covariance matrix of the PCA features is $W\mathsf{C}W^\top = U^\top UDU^\top U = D$, we can use a theorem from linear algebra[9] to see that the new features have the same total variance.

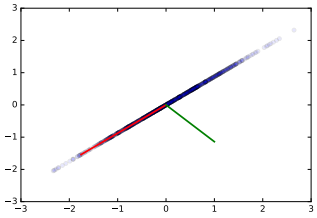
We can characterize the PCA features by their two important properties (see Section 6.1.1):

[8] Depending on the source, this may be defined as $\sqrt{\lambda_i}\boldsymbol{u}_i$.

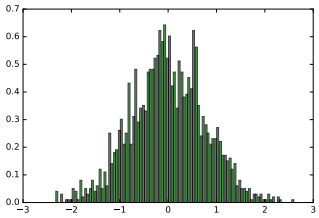
[9] The trace of a matrix is the sum of its diagonal entries, and trace is invariant under cyclic permutations: trace(AB) = trace(BA).



(a) The original data has two features (given on the horizontal and vertical axes). The first principal component is in the direction where the data has the most variance (red). The second principal component is in a direction orthogonal to the first principal component that has the most variance (green).



(b) Reducing the data to 1 dimension by projecting it along the first principal component (red; displayed in the original feature space).



(c) First PCA feature – distance from the origin of a point along the red line in Fig. 6.3b and its frequency of occurence.

- The PCA features are uncorrelated (their covariance matrix is D that has zeros on the off diagonal), and they represent orthogonal directions in the original feature space (since they are eigenvectors of a symmetric matrix).
- The ith feature captures the variance of the data in direction $\boldsymbol{u}_i$. This direction has the largest variance possible subject to being orthogonal to $\boldsymbol{u}_1, \ldots, \boldsymbol{u}_{i-1}$ (whose variance were captured by the first $i-1$ features).

  In other words, the first k PCA features capture the most variance possible in the original data by an orthogonal transformation of the data.

The second point follows from the Courant-Fischer minimax principle. Roughly speaking, the argument goes like this: the minimax principle says that the first unit eigenvector of the covariance matrix, $\boldsymbol{u}_1$, is the vector that maximizes $\boldsymbol{u}_1^\top \mathsf{C} \boldsymbol{u}_1$ (and this is the first eigenvalue of C). The first PCA feature is $\boldsymbol{u}_1^\top \boldsymbol{x}$, so its variance is $\mathrm{E}[(\boldsymbol{u}_1^\top \boldsymbol{x})(\boldsymbol{u}_1^\top \boldsymbol{x})^\top] = \boldsymbol{u}_1^\top \mathsf{C} \boldsymbol{u}_1$, so it is the direction of maximum variance. The ith PCA feature is given by $\boldsymbol{u}_i^\top \boldsymbol{x}$, and its variance (similar to the first PCA feature) is $\boldsymbol{u}_i^\top \mathsf{C} \boldsymbol{u}_i$. The minimax principle says that among all unit vectors $\boldsymbol{z}$ orthogonal to $\boldsymbol{u}_1, \ldots, \boldsymbol{u}_{i-1}$, the one which maximizes $\boldsymbol{z}^\top \mathsf{C} \boldsymbol{z}$ is $\boldsymbol{u}_i$. By induction, and the fact that the total variance of the PCA features is the same as the original features, the argument is completed.

A pictorial representation of the principal components for a distribution in $\mathbb{R}^2$ is given in Fig. 6.3a. The PCA features of a data point would be the projection of a data point along the principal components. The dimension reduced data by retaining only the first principal component in the original feature space is shown in Fig. 6.3b, and the PCA feature is visualized in Fig. 6.3c as a histogram.

An equivalent way of thinking about this is that keeping the first k PCA features, we can approximate the original data with the lowest mean square error possible among all choices of k features that are a linear transform of the original data. Details on this formulation are given in [36].

The idea of dimensionality reduction with PCA is to use the first k PCA features, since they capture the most variance of the data. We do this by multiplying the data set by the matrix consisting of the first k rows of W, which we will call W_k. To go back to the original feature space from the PCA features, simply multiply the PCA features by $W_k^\top$.

In practice, we do not have access to C, so we estimate it with the empirical covariance matrix (similar to what we did in LDA). The algorithm for PCA along with dimension reduction is given in Alg. 9. If we take the number of features to retain in PCADimensionReduction to be all of them, and then apply PCAfeaturesToOriginalFeatures, we recover the original data exactly.

Algorithm 9 Implementation of PCA using the eigendecomposition (assuming features have mean zero), along with dimensionality reduction and reconstruction of features.

function PCAWITHEIGEN DECOMPOSITION(data set $\{\boldsymbol{x}_i\}_{i=1}^N$)
  Estimate the covariance matrix: $\hat{\mathsf{C}} = \frac{1}{N}\sum_{i=1}^N \boldsymbol{x}_i\boldsymbol{x}_i^\top$
  Calculate eigendecomposition of $\hat{\mathsf{C}} = UDU^\top$
  Determine PCA transformation: $W = U^\top$
  return PCA transformation, W and amount of variance explained each feature given by the diagonal of D
end function
function PCADIMENSIONREDUCTION(data set $\{\boldsymbol{x}_i\}_{i=1}^N$,PCA Transformation W, Number of features to retain k)
  Let W_k be the matrix consisting of the first k rows of W.
  return First k PCA features, $\{W_k\boldsymbol{x}_i\}_{i=1}^N$.
end function
function PCAFEATURESTOORIGINALFEATURES(PCA features of size k, $\{W_k\boldsymbol{x}_i\}_{i=1}^N$, PCA Transformation W)
  Let W_k be the matrix consisting of the first k rows of W.
  return Reconstructed Original Features, $\{W_k^\top\boldsymbol{x}_i\}_{i=1}^N$.
end function

While Alg. 9 is correct, it is not efficient when the number of features is large, since explicitly calculating out the covariance matrix will require a considerable amount of storage and possible loss of precision (numerical instability)[10]. To alleviate this, it is preferable to use the SVD instead of the eigendecomposition[11].

Let X the N-by-d data matrix, where row i of X is $\boldsymbol{x}_i^\top$. Then,

$$\begin{aligned}\frac{1}{N}X^\top X &= \frac{1}{N}\sum_{i=1}^N \boldsymbol{x}_i\boldsymbol{x}_i^\top \\ &= \hat{\mathsf{C}}.\end{aligned}$$

By our discussion of the SVD of $X = USV^\top$, the eigendecomposition of $X^\top X$ is $VS^2V^\top$ where

[10]For those who have a numerical analysis background, the eigendecomposition approach will have a condition number that is the square of that of the SVD approach.

[11]There can be a speed difference between the two approaches, with the eigendecomposition being asymptotically more favorable if $N \gg d$ and both methods being on the same order of computational complexity in the computation model described in [20].

S^2 denotes a diagonal matrix of appropriate size containing the squared singular values of X. Thus, our PCA transformation is given by $W = V^\top$.

Algorithm 10 Implementation of PCA using SVD (assuming features have mean zero). Dimensionality reduction and reconstruction are identical to Alg. 9.

function PCAwithSVD(data set $\{\boldsymbol{x}_i\}_{i=1}^N$)
  Let X be the N-by-d data matrix (ith row is $\boldsymbol{x}_i$).
  Calculate SVD of $X = USV^\top$
  Determine PCA transformation: $W = V^\top$
  return PCA transformation, W and amount of variance explained each feature given by the diagonal of $\frac{1}{N}S^2$ where S^2 squares the non-zero entries of S
end function

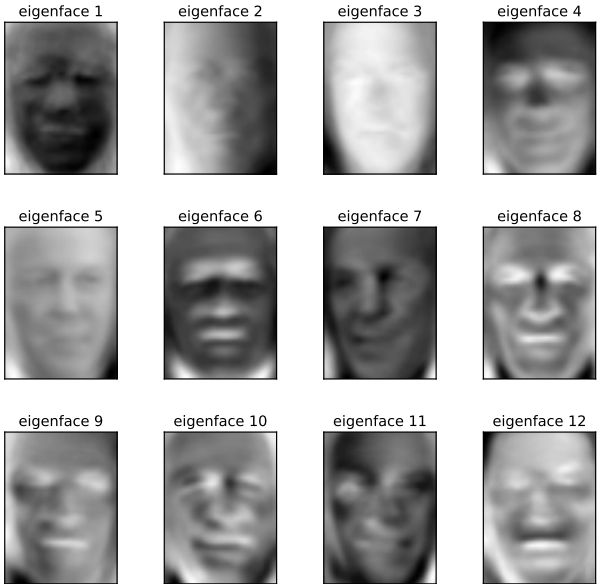
Example[12]**:** We start with a labeled data set of faces (from Ariel Sharon, Colin Powell, Donald Rumsfeld, George W Bush, Gerhard Schroeder, Hugo Chavez and Tony Blair) as 1288 images with 50x37 pixels with each pixel as a feature, and reserve 25% for testing. So, the original feature space has dimension 1850. PCA was applied to reduce the dimensionality of the data. The principal components are called *eigenfaces* [42], and 50 eigenfaces were used in this example (out of 1850 possible) as shown in Fig. 6.4a. Based on the PCA features, a 9-NN classifier was selected by cross-validation. Then, this classifier was tested on other faces as shown in Fig. 6.4b. Performance metrics are shown in Fig. 6.5b.

The confusion matrix (as described in Sec. 2.2.1) is given in Fig. 6.5a. One can see from the training data that there is a class imbalance – there are many more pictures of George W Bush than others. The precision, recall and F-measure (also described in Sec. 2.2.1, albeit in the binary setting) are reported in Fig. 6.5b. About 70% of datapoints are correctly classified overall. The confusion matrix, however, shows us more detail on the types of errors our classifier made in classifying. For example, most of the errors were towards declaring George W. Bush.

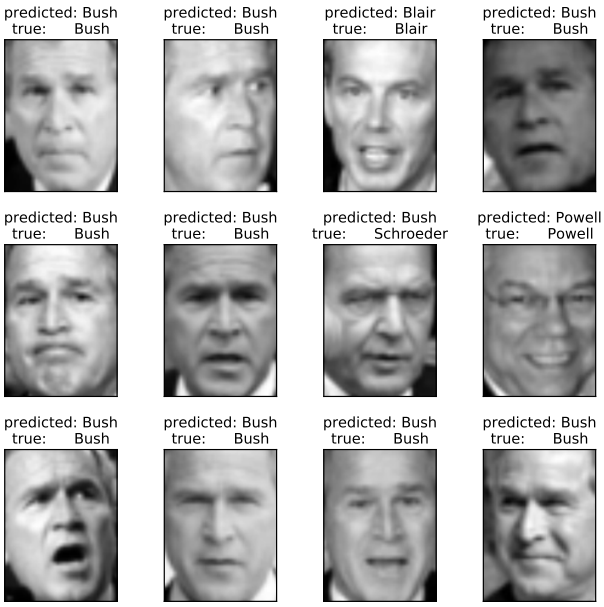
If the feature vectors follow a Gaussian distribution, the features that PCA generates are actually better than uncorrelated – they are independent. In general, the features PCA generates are not independent. A family of techniques known as Independent Component Analysis addresses this problem, but we do not have time to discuss it in this course. Further extensions such as sparse PCA, robust PCA and kernel PCA are also possible [24].

In practice, many PCA implementations use the thin SVD or randomized SVD algorithms in order to save on storage and computation, especially when the number of features or input vectors is large. This allows us to avoid storing extraneous principal components (i.e. ones we will not use for dimensionality reduction). Our presentation of PCA via the SVD holds with minor modifications in these cases. Other techniques to reduce computational costs associated with PCA are outlined in [3]. Most machine learning packages contain routines to perform PCA dimensionality reduction efficiently.

[12]This is based on an example from the Scikit-Learn documentation called "Faces recognition example using eigenfaces and SVMs" (`http://scikit-learn.org/stable/auto_examples/applications/face_recognition.html`) [38], BSD License.

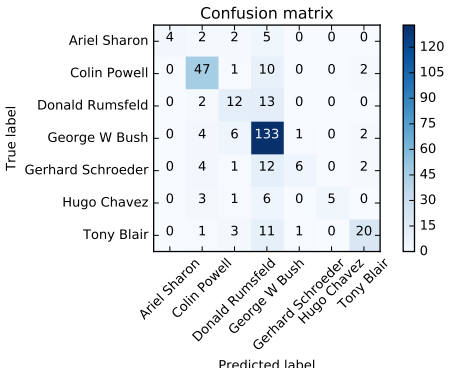


(a) Eigenfaces



(b) Classifier Results

Figure 6.4: A demonstration of PCA applied to classification on the Labeled Faces in the Wild Dataset (`http://vis-www.cs.umass.edu/lfw/`, [28]).



(a) Confusion Matrix

	precision	recall	f1-score	support
Ariel Sharon	1.00	0.31	0.47	13
Colin Powell	0.75	0.78	0.76	60
Donald Rumsfeld	0.46	0.44	0.45	27
George W Bush	0.70	0.91	0.79	146
Gerhard Schroeder	0.75	0.24	0.36	25
Hugo Chavez	1.00	0.33	0.50	15
Tony Blair	0.77	0.56	0.65	36
avg / total	0.73	0.70	0.68	322

(b) Precision, Recall and F-measure and frequency in the test set.

Figure 6.5: Error performance for the classification problem in Fig. 6.4.

6.4.1 Choosing the Number of Principal components

One way to choose the number of principal components to use is based on making the cost function small enough:

$$J(k) = \frac{1}{N}\sum_{i=1}^{N} \|\boldsymbol{x}_i - W_k^\top W_k \boldsymbol{x}_i\|^2. \tag{6.24}$$

The vector $W_k\boldsymbol{x}_i$ consists of the PCA features for input vector $\boldsymbol{x}_i$, and multiplying by $W_k^\top$ gives an approximation of the original feature vector $\boldsymbol{x}_i$. This cost function measures the mean square reconstruction error using the first k principal components. It can be proven that if the eigenvalues of the covariance matrix of the data are $\lambda_1 \geq \lambda_2 \geq \ldots$, then

$$J(k) \approx \sum_{i \geq k+1} \lambda_i. \tag{6.25}$$

Similar to the case for K-means, we can look for a kink in $J(k)$. The intuition is that for the important principal components (i.e., one that explains a large fraction of variance), we should have λ be large. So, if the k-th principal component is important, we expect a large drop between $J(k-1)$ and $J(k)$. If the principal component is not large, we expect a small drop between $J(k-1)$ and $J(k)$. Similar to K-means, if all the principal components are used, the cost function will be zero. These plots are known as *scree*[13] plots. An example is given in Fig. 6.6.

[13] A scree is a bunch of rubble at the base of a mountain. Someone thought these plots looked like that, hence the name [36].

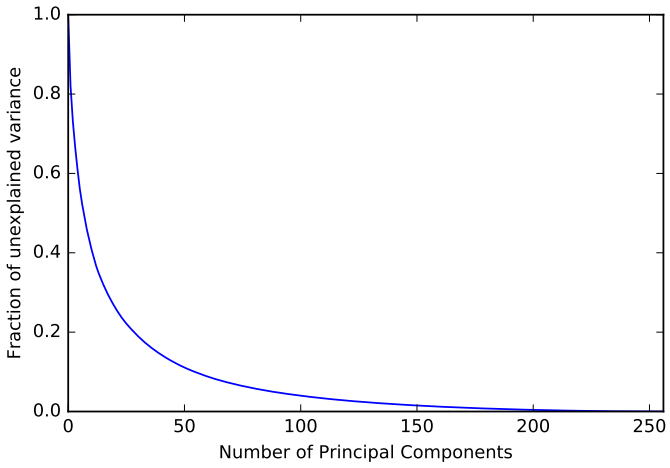


Figure 6.6: A scree plot (normalized by the total variance of the data) based on the MNIST dataset (Available at `http://yann.lecun.com/exdb/mnist/`. See [31] for a history of classification on this data set.). We see that the first few principal components explain most of the variance of the data, whereas using more principal components gives diminishing gains.

Bibliography

[1] Michal Aharon, Michael Elad, and Alfred Bruckstein. K-svd: An algorithm for designing overcomplete dictionaries for sparse representation. *Signal Processing, IEEE Transactions on*, 54(11):4311–4322, 2006.

[2] Arindam Banerjee, Srujana Merugu, Inderjit S Dhillon, and Joydeep Ghosh. Clustering with bregman divergences. *Journal of machine learning research*, 6(Oct):1705–1749, 2005.

[3] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[4] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, New York, NY, USA, 2004.

[5] James W. Brown and Ruel V. Churchill. *Fourier Series and Boundary Value Problems*. McGraw-Hill, 2006.

[6] Christopher JC Burges. A tutorial on support vector machines for pattern recognition. *Data mining and knowledge discovery*, 2(2):121–167, 1998.

[7] Chih-Chung Chang and Chih-Jen Lin. Libsvm: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):27, 2011.

[8] Thomas Cover and Peter Hart. Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27, 1967.

[9] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory (2e)*. Wiley-Interscience, 2006.

[10] Felipe Cucker and Ding-Xuan Zhou. *Learning theory : an approximation theory viewpoint*. Cambridge monographs on applied and computational mathematics. Cambridge University Press, Cambridge, New York, 2007.

[11] Pedro Domingos and Michael Pazzani. On the optimality of the simple bayesian classifier under zero-one loss. *Machine learning*, 29(2-3):103–130, 1997.

[12] David L Donoho. Aide-memoire. high-dimensional data analysis: The curses and blessings of dimensionality. *American Math. Society Lecture-Math Challenges of the 21st Century*, 2000.

[13] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification*. Wiley-Interscience, New York, 2nd edition, 2000.

[14] Charles Duhigg. How companies learn your secrets. *The New York Times Magazine*, February 2012.

[15] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. Liblinear: A library for large linear classification. *Journal of machine learning research*, 9(Aug):1871–1874, 2008.

[16] Tom Fawcett. An introduction to roc analysis. *Pattern recognition letters*, 27(8):861–874, 2006.

[17] David A. Forsyth and Jean Ponce. *Computer Vision: A Modern Approach*. Prentice Hall, 2002.

[18] Allen Gersho and Robert M. Gray. *Vector Quantization and Signal Compression*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.

[19] Gene H Golub, Alan Hoffman, and Gilbert W Stewart. A generalization of the eckart-young-mirsky matrix approximation theorem. *Linear Algebra and its applications*, 88:317–327, 1987.

[20] Gene H. Golub and Charles F. Van Loan. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[21] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[22] Bruce Hajek. *Probability with Engineering Applications*. University of Illinois, jan 2017.

[23] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.

[24] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning (2nd Edition)*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2008.

[25] Simon Haykin. *Communication Systems*. Wiley Publishing, 4th edition, 2000.

[26] K. Hoffman and R.A. Kunze. *Linear algebra*. Prentice-Hall. Prentice-Hall, 1971.

[27] Roger A. Horn and Charles R. Johnson, editors. *Matrix Analysis*. Cambridge University Press, New York, NY, USA, 1990.

[28] Gary B. Huang, Manu Ramesh, Tamara Berg, and Erik Learned-Miller. Labeled faces in the wild: A database for studying face recognition in unconstrained environments. Technical Report 07-49, University of Massachusetts, Amherst, October 2007.

[29] Zhexue Huang. Extensions to the k-means algorithm for clustering large data sets with categorical values. *Data mining and knowledge discovery*, 2(3):283–304, 1998.

[30] R. D. King, C. Feng, and A. Sutherland. Statlog: Comparison of classification algorithms on large real-world problems, 1995.

[31] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[32] David D Lewis. Naive (bayes) at forty: The independence assumption in information retrieval. In *European conference on machine learning*, pages 4–15. Springer, 1998.

[33] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*. Cambridge University Press, 2008.

[34] Vangelis Metsis, Ion Androutsopoulos, and Georgios Paliouras. Spam filtering with naive bayes-which naive bayes? In *CEAS*, volume 17, pages 28–69, 2006.

[35] Pierre Moulin and Venugopal V Veeravalli. *Statistical inference for engineers and data scientists*. Cambridge University Press, Cambridge, 2019.

[36] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, Cambridge, MA, 2012.

[37] Andrew Ng and Michael I. Jordan. On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes. *Advances in Neural Information Processing Systems*, 14:841, 2002.

[38] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[39] Lawrence R Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[40] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Transactions on pattern analysis and machine intelligence*, 22(8):888–905, 2000.

[41] Gilbert Strang. *Linear Algebra and Its Applications*. Harcourth Brace, 3 edition, 1988.

[42] Matthew Turk and Alex Pentland. Eigenfaces for recognition. *Journal of cognitive neuroscience*, 3(1):71–86, 1991.

[43] Ulrike von Luxburg, Robert C. Williamson, and Isabelle Guyon. Clustering: Science or art? *Proc. Machine Learning Research*, 27:65–79, 2012.

[44] Larry Wasserman. *All of Statistics: A Concise Course in Statistical Inference*. Springer Publishing Company, Incorporated, 2010.

[45] Harry Zhang. The optimality of naive bayes. In *Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference, Miami Beach, Florida, USA*, pages 562–567, 2004.

