Machine Learning

Onkar Pandit

14 April 2022

Last updated on November 3, 2022.

Contents

1 Introduction

A machine learning algorithm is an algorithm that is able to learn from data. Mitchell provides the definition "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."

Task T can be any generic task, such as classification of spam mails, or prediction of the stock values, or translation of text from one language to other, and many more. The performance measure P is the measures the extent up to which the task is performed well, accuracy, $F1$-score, error rate are the few examples of it. Finally, the experience E is the data provided to the learning algorithm based on which algorithms can learn, for example in case of supervised learning algorithms the experience is labelled data.

As described by Mitchell, in machine learning following points are quite important.

1. **Choice of data** We want to train a machine and for that we need an input data, what kind of data we want to input to a machine learning algorithm is important for the success of the system.

2. **Choice of the problem** Suppose we need to play a chess game. Then there are many problems which can be solved in order to play the chess successfully. But most effectively, we can formulate it as a problem of choosing a best move given a current state of the board from a legal moves which can be played. This is the stage, where we actually decide, what kind of knowledge we want machine to learn.

3. **Function representation** Now, once we decide on the problem we intend to solve, we can choose a representation function. For example, linear function is one of the choice as well as polynomial function etc. Here, we can also decide on the type of features we want to input to the system. The kind of features.

4. **Selection of target function**: Once we decide on the function representation of the system, we have to measure the system performance. Generally this is called a loss function which measures the difference between system predicted output and true output. This loss function is considered as a minimization objective over the parameters. Thus, parameters are learned which can reduce the loss as much as possible. The optimization problem is solved with different approaches, thus choice of loss function as well as optimization technique are closely related and are chosen at tandem.

5. **Inference**: Inference is overloaded in machine learning. It means that using a learned (or trained) machine learning model to predict output for new inputs. Also, it concerns with actually making predictions with decision theory so as to reduce the cost function. In NLP it has different

meaning, where expected output which is text and not numbers. Thus, in NLP different inference algorithms are designed so as to efficiently go from predicted probability numbers to NLP output.

2 Machine Learning: Data, Model, and Learning

There are three major components of a machine learning system: data, models, and learning. A data is the experience from the real world which the model uses to learn the intricacies to solve the complex task. One of the guiding principles of machine learning is that good models should perform well on unseen data. We elaborate about these three components in the following sections

2.1 Data

Data is crucial for machine learning, and without data, machine learning is not possible. It requires data in one form or the other to grow its experience and ability to make decisions based on the data fed to it. It is crucial to have data with which it is possible to solve the task. This means the features in each instance of the data should have meaningful information. It is important and challenging to identify good features. Many of these aspects depend on domain expertise and require careful engineering, and, in recent years, deep learning approaches have been used to represent the basic data with useful information. In addition to that, it is equally important to have correct data, otherwise making useful conclusions based on the data is difficult.

Machine learning algorithms accept numerical data which is arranged in a tabular format, where we think of each row of the table as representing a particular instance or example, and each column to be a particular feature. Even when we have data in tabular format, there are still choices to be made to obtain a numerical representation. For example, a categorical variable may be converted into any distinct numbers but the right choice will depend on the empirical performance. Furthermore, it is often important to use domain knowledge when constructing the representation.

In recent years, machine learning has been applied to many types of data that do not obviously come in the tabular numerical format, for example genomic sequences, text and image contents of a webpage, and social media graphs. In these domains it is important to represent the non-numerical data into numerical data while keeping the original information intact.

2.2 Model

Once we have data in an appropriate vector representation, we can get to the business of constructing a predictive function, also known as a predictor. A model describes data that one could observe from a system. Thus, when we get the data, we make educational guess, as to which model to use for fitting the

data at hand. For example, in linear regression, we assume that the relation between the output and input is linear, therefore we employ a linear model. in other cases, we might use different models. There are two major approaches in machine learning for models: a predictor as a function, and a predictor as a probabilistic model. We describe both in the next subsection.

2.2.1 Models as Functions

A predictor is a function that, when given a particular input example, in our case, a vector of features, produces an output. For now, consider the output to be a single number, i.e., a real-valued scalar output. This can be written as

$$f : R^D \rightarrow R \tag{1}$$

Linear regression, that we see in the subsequent sections is one of the examples of model as a function.

2.2.2 Models as Probability Distributions

We often consider data to be noisy observations of some true underlying effect, and hope that by applying machine learning we can identify the signal from the noise. This requires us to have a language for quantifying the effect of noise. We often would also like to have predictors that express some sort of uncertainty, e.g., to quantify the confidence we have about the value of the prediction for a particular test data point. Instead of considering a predictor as a single function, we could consider predictors to be probabilistic models, i.e., models describing the distribution of possible functions.

2.2.3 Discriminative vs Generative Models

Discriminative Models For many applications of machine learning the goal is to predict the value of a vector y given the value of a vector x of input features. In a classification problem y represents a discrete class label, whereas in a regression problem it corresponds to one or more continuous variables. From a probabilistic perspective, the goal is to find the conditional distribution $p(y|x)$. The most common approach to this problem is to represent the conditional distribution using a parametric model, and then to determine the parameters using a training set consisting of pairs $\{x_n, y_n\}$ of input vectors along with their corresponding target output vectors. The resulting conditional distribution can be used to make predictions of y for new values of x. This is known as a *discriminative* approach, since the conditional distribution discriminates directly between the different values of y.

In the discriminative models define the conditional probability $p(y|x)$ for any x, y pair. The parameters of the model are estimated from the training examples. Given a new test example x, the output from the model is

$$f(x) = \arg\max_{y \in \mathcal{Y}} p(y|x)$$

Thus we simply take the most likely label y as the output from the model. If our model $p(y|x)$ is close to the true conditional distribution of labels given inputs, the function $f(x)$ will be close to optimal.

Generative Models An alternative approach is to find the joint distribution $p(x, y)$, expressed for instance as a parametric model, and then subsequently uses this joint distribution to evaluate the conditional $p(y|x)$ in order to make predictions of y for new values of x. This is known as a generative approach since by sampling from the joint distribution it is possible to generate synthetic examples of the feature vector x.

As opposed to discriminative models that directly estimate the conditional distribution $p(y|x)$, in generative models instead model the joint probability $p(x, y)$ over (x, y) pairs. The parameters of the model $p(x, y)$ are again estimated from the training examples $(x^(i), y^(i))$ for $i = 1, \cdots, n$. In many cases we further decompose the probability $p(x, y)$ as follows:

$$p(x, y) = p(y)p(x|y)$$

and then estimate the models for $p(y)$ and $p(x|y)$ separately. These two model components have the following interpretations:

- $p(y)$ is a *prior* probability distribution over labels y.
- $p(x|y)$ is the probability of generating the input x, given that the underlying label is y.

We will see that in many cases it is very convenient to decompose models in this way; for example, the classical approach to speech recognition is based on this type of decomposition.

Given a generative model, we can use Bayes rule to derive the conditional probability $p(y|x)$ for any (x, y) pair:

$$p(y|x) = \frac{p(x|y) \times p(y)}{p(x)} \tag{2}$$

where

$$p(x) = \sum_{y \in \mathcal{Y}} p(x, y) = \sum_{y \in \mathcal{Y}} p(x|y) \times p(y)$$

Thus the joint model is quite versatile, in that we can also derive the probabilities $p(x)$ and $p(y|x)$. We use Bayes rule directly in applying the joint model to a new test example. Given an input x, the output of our model, $f(x)$, can be derived as follows:

$$
\begin{aligned}
f(x) &= \arg\max_{y \in \mathcal{Y}} p(y|x) \\
&= \arg\max_{y \in \mathcal{Y}} \frac{p(x|y) \times p(y)}{p(x)} \\
&= \arg\max_{y \in \mathcal{Y}} p(x|y) \times p(y)
\end{aligned}
$$

The last equation follows because the denominator, $p(x)$, does not depend on y, and hence does not affect the arg max. This is convenient, because it means that we do not need to calculate $p(x)$, which can be an expensive operation.

Models that decompose a joint probability into into terms $p(y)$ and $p(x|y)$ are often called *noisy-channel models*. Intuitively, when we see a test example x, we assume that has been generated in two steps: first, a label y has been chosen with probability $p(y)$; second, the example x has been generated from the distribution $p(x|y)$. The model $p(x|y)$ can be interpreted as a "channel" which takes a label y as its input, and corrupts it to produce x as its output. Our task is to find the most likely label y, given that we observe x.

In practice, the generalization performance of generative models is often found to be poorer than than of discriminative models due to differences between the model and the true distribution of the data. When labelled training data is plentiful, discriminative techniques are widely used since they give excellent generalization performance. However, although collection of data is often easy, the process of labelling it can be expensive. Consequently there is increasing interest in generative methods since these can exploit unlabelled data in addition to labelled data.

There are compelling reasons to use discriminative models than generative models as it is better to solve the classification problem directly than learning more general joint probability as intermediate step. Thus, discriminative models are always preferred to generative models in practice.

Naive Bayes classifier is an example of generative classifier which first finds out the joint distribution $p(x, y)$ via calculations of conditional probabilities as $p(x|y)$ and $p(y)$. In contrast to that logistic regression is considered to be discriminative as it directly models $p(y|x)$. In addition to naive bayes algorithms, Bayesian Networks, Markov Random Fields, Hidden Markov Models are considered to be generative models. Whereas, Neural Networks, SVMs are considered to be discriminative models.

Concretely, suppose we are solving a classification problem. We have a joint model over labels $Y = y$, and features $X = \{x_1, x_2, \cdots x_n\}$. The joint distribution of the model can be represented as $p(X, Y) = P(x_1, x_2, \cdots x_n, y)$. Once we learn these probabilities, our goal is to estimate the probability: $P(Y = 1|X)$. Both generative and discriminative models can solve this problem, but in different ways.

To get the conditional probability $P(Y|X)$, generative models estimate the prior $P(Y)$ and likelihood $P(X|Y)$ from training data and use Bayes rule to calculate the posterior $P(Y|X)$:

$$posterior \propto \frac{likelihood \times prior}{evidence} \tag{3}$$

$$P(Y|X) = \frac{P(X|Y) \times P(Y)}{P(X)} \tag{4}$$

Now from the above equation it is to see why Naive Bayes is called generative model. The likelihood probability is calculated $P(X|Y)$ by making the assumption that the features are conditionally independent given the label y, and the prior $P(Y)$ is calculated from the training data. Then the model selects the label

$$\hat{y} = \arg\max_{y \in Y} \frac{P(X|Y) \times P(Y)}{P(X)} \tag{5}$$

The calculation $P(X)$ is computationally intractable. It is not needed to calculate as the $\arg\max$ does not depend on x and it is going to be constant for all the values of y so it is neglected and prediction is done as follows:

$$\hat{y} = \arg\max_{y \in Y} P(X|Y) \times P(Y) \tag{6}$$

2.3 Learning

The goal of learning is to obtain optimal values of model parameters such that the resulting predictor will perform well on unseen data. It is important to note that we are not only concenerned about the model's performance on the seen data but more on the unseen data.

There are conceptually three distinct algorithmic phases when discussing machine learning algorithms:

1. Prediction or inference
2. Training or parameter estimation
3. Hyperparameter tuning or model selection

2.3.1 Prediction

The prediction phase is when we use a trained predictor on previously unseen test data. In other words, the parameters and model choice is already fixed and the predictor is applied to new vectors representing new input data points. When we have a probabilistic model the prediction phase is called *inference*.

Unfortunately, there is no agreed upon naming for the different algorithmic phases. The word "inference" is sometimes also used to mean parameter estimation of a probabilistic model, and less often may be also used to mean prediction for non-probabilistic models.

2.3.2 Parameter Estimation

The training or parameter estimation phase is when we adjust our predictive model based on training data. We would like to find good predictors given training data, and there are two main strategies for doing so: finding the best predictor based on some measure of quality, sometimes called finding a point estimate, or using Bayesian inference. Finding a point estimate can be applied to both types of predictors, but Bayesian inference requires probabilistic models.

For the non-probabilistic model, we follow the principle of empirical risk minimization, where the parameters are adjusted so to minimize the difference between desired output and predicted output. This difference between desired and predicted output is quantified with **loss function**, which is a empirical risk. Then this value is minimized where this minimization is formed as **optimization** problem with loss function as an objective function. In general, numerical methods are used to optimize the parameters so that minimum loss is attained by the model.

2.3.3 Hyperparameters

We often need to make high-level modeling decisions about the structure of the predictor, such as the number of components to use or the class of probability distributions to consider. The choice of the number of components is an example of a hyperparameter, and this choice can affect the performance of the model significantly but it comes out of the actual adjustment of parameters.

The distinction between parameters and hyperparameters is somewhat arbitrary, and is mostly driven by the distinction between what can be numerically optimized versus what needs to use search techniques. Another way to consider the distinction is to consider parameters as the explicit parameters of a probabilistic model, and to consider hyperparameters, higher-level parameters, as parameters that control the distribution of these explicit parameters.

2.4 Building Machine Learning Algorithm

Nearly all deep learning algorithms can be described as particular instances of a fairly simple recipe: combine a specification of a dataset, a cost function, an optimization procedure and a model.

By realizing that we can replace any of these components mostly independently from the others, we can obtain a very wide variety of algorithms.

The cost function typically includes at least one term that causes the learning process to perform statistical estimation. The most common cost function is the negative log-likelihood, so that minimizing the cost function causes maximum likelihood estimation. The cost function may also include additional terms, such as regularization terms. This term is typically included to make the model generalize over the unseen data. it is any modification done in learning algorithms so as to reduce the generalization error but not its training error.

If we change the model to be nonlinear, then most cost functions can no longer be optimized in closed form. This requires us to choose an iterative numerical optimization procedure, such as gradient descent.

In some cases, the cost function may be a function that we cannot actually evaluate, for computational reasons. In these cases, we can still approximately minimize it using iterative numerical optimization so long as we have some way of approximating its gradients.

Most machine learning algorithms make use of this recipe, though it may not immediately be obvious. If a machine learning algorithm seems especially unique or hand-designed, it can usually be understood as using a special-case optimizer. Some models such as decision trees or k-means require special-case optimizers because their cost functions have flat regions that make them inappropriate for minimization by gradient-based optimizers. Recognizing that most machine learning algorithms can be described using this recipe helps to see the different algorithms as part of a taxonomy of methods for doing related tasks that work for similar reasons, rather than as a long list of algorithms that each have separate justifications.

3 Parameter Estimation

Let us start the discussion with a simple problem where we have n observations $\mathbf{x} = \{x_1, x_2, \cdots, x_n\}$ and corresponding values $\mathbf{y} = \{y_1, y_2, \cdots, y_n\}$. Assume for now, the input x is a scalar value and not a vector. Given these values, we intend to find a function $f(x, w)$ which predicts $\hat{y}$ for an input x and parameters w. The whole problem boils down to searching a hypothesis space of parameters and estimating values for w which sufficiently predicts the output given an input.

3.1 Maximum Likelihood Estimation

Let us see how we can efficiently search for values of w. The value of $\hat{y}$ depends on function $f(x, w)$ and we assume that the distribution is $\mathcal{N}(f(x, w), \sigma)$. Then the likelihood of observing the n points with the independent and identically distributed assumption,

$$p(\mathbf{y}|x, w) = \prod_{i=1}^{n} \frac{1}{\sigma\sqrt{2\pi}} \exp^{-\frac{(y_i - f(x_i, w))^2}{2\sigma^2}} \tag{7}$$

In practice, it is more convenient to maximize the log of the likelihood function. Because the logarithm is strictly increasing function of its argument (if for all x and y such that $x \leq y$ one has $f(x) \leq f(y)$). Thus, maximization of the log of a function is equivalent to maximization of the function itself. But for practical purposes it is more convenient to work with the log-likelihood function in maximum likelihood estimation, in particular since most common probability distributions–notably the exponential family–are only logarithmically concave, and concavity of the objective function plays a key role in the maximization.

Taking the log not only simplifies the subsequent mathematical analysis, but it also helps numerically because the product of a large number of small probabilities can easily underflow the numerical precision of the computer, and this is resolved by computing instead the sum of the log probabilities.

Given the independence of each event, the overall log-likelihood of intersection equals the sum of the log-likelihoods of the individual events. This is analogous to the fact that the overall log-probability is the sum of the log-probability of the individual events. In addition to the mathematical convenience from this, the adding process of log-likelihood has an intuitive interpretation, as often expressed as "support" from the data. When the parameters are estimated using the log-likelihood for the maximum likelihood estimation, each data point is used by being added to the total log-likelihood. As the data can be viewed as an evidence that support the estimated parameters, this process can be interpreted as "support from independent evidence adds", and the log-likelihood is the "weight of evidence". Interpreting negative log-probability as information content or surprisal, the support (log-likelihood) of a model, given an event, is the negative of the surprisal of the event, given the model: a model is supported by an event to the extent that the event is unsurprising, given the model.

Log-likelihood is given as:

$$\log p(\mathbf{y}|x, w) = -n \log(\sigma\sqrt{2\pi}) - \sum_{i=1}^{n} \frac{(y_i - f(x_i, w))^2}{2\sigma^2} \tag{8}$$

Now, in maximum likelihood estimation, we intend to maximize the above mentioned log-likelihood function. We intend to maximize this with respect to parameters w. The first term in the equation does not depend on weights w, so we have to concentrate on the square term which depends on w. This term is multiplied by -1 so we have to minimize the square term, which is nothing but the minimizing squared loss function.

Variance Estimation So far, we have dealt with the mean of the likelihood function. Now, if we want to estimate the variance as well as want to reduce it, we derive the equation to reduce it.

$$\begin{aligned}
\log p(\mathbf{Y}|\mathbf{X}, \beta, \sigma^2) &= \sum_{n=1}^{N} \log \mathcal{N}(\mathbf{x}_i^\mathsf{T}\boldsymbol{\beta}, \sigma^2) \\
&= -N \log(\sigma^2\sqrt{2\pi}) - \sum_{n=1}^{N} \frac{(y_n - \mathbf{x}_i^\mathsf{T}\boldsymbol{\beta})^2}{2\sigma^2} \\
&= -\frac{N}{2} \log(\sigma^2) - \frac{1}{2\sigma^2} \sum_{n=1}^{N} (y_n - \mathbf{x}_i^\mathsf{T}\boldsymbol{\beta})^2
\end{aligned}$$

Taking the partial derivative with respect to σ^2 and equating it to 0.

$$\frac{\partial \log p(\mathbf{Y}|\mathbf{X}, \beta, \sigma^2)}{\partial \sigma^2} = -\frac{N}{\sigma^2} + \frac{1}{2\sigma^4} \sum_{n=1}^{N} (y_n - \mathbf{x}_i^\mathsf{T} \boldsymbol{\beta})^2 = 0$$

$$\Rightarrow \frac{N}{\sigma^2} = \frac{1}{2\sigma^4} \sum_{n=1}^{N} (y_n - \mathbf{x}_i^\mathsf{T} \boldsymbol{\beta})^2$$

$$\Rightarrow \sigma^2_{ML} = \frac{1}{N} \sum_{n=1}^{N} (y_n - \mathbf{x}_i^\mathsf{T} \boldsymbol{\beta})^2$$

Therefore, the maximum likelihood estimate of the noise variance is the empirical mean of the squared distances between the noise-free function values $\mathbf{x}_i^\mathsf{T} \boldsymbol{\beta}$ and the corresponding noisy observations y_n at input locations x_n.

Connection between MLE, cross entropy and KL divergence Maximum likelihood estimation of parameters θ for m iid instances is obtained by solving following optimization problem

$$\theta_{MLE} = \arg\max_{\theta} \sum_{i=1}^{m} \log p_{model}(\boldsymbol{x}_i; \boldsymbol{\theta}) \tag{9}$$

Because the *arg max* does not change when we rescale the cost function, we can divide by m to obtain a version of the criterion that is expressed as an expectation with respect to the empirical distribution $\hat{p}_{data}$ defined by the training data:

$$\theta_{MLE} = \arg\max_{\theta} \mathbb{E}_{\mathbf{x} \sim \hat{p}_{data}} \log p_{model}(\boldsymbol{x}; \boldsymbol{\theta}) \tag{10}$$

One way to interpret maximum likelihood estimation is to view it as minimizing the dissimilarity between the empirical distribution $\hat{p}_{data}$ defined by the training set and the model distribution, with the degree of dissimilarity between the two measured by the KL divergence. The KL divergence is given by

$$D_{KL}(\hat{p}_{data} \parallel p_{model}) = \mathbb{E}_{\mathbf{x} \sim \hat{p}_{data}} [\log \hat{p}_{data}(\boldsymbol{x}) - \log p_{model}(\boldsymbol{x})] \tag{11}$$

The term on the left is a function only of the data generating process, not the model. This means when we train the model to minimize the KL divergence, we need only minimize

$$-\mathbb{E}_{\mathbf{x} \sim \hat{p}_{data}} [\log p_{model}(\boldsymbol{x})] \tag{12}$$

which is similar to maximization in eq. 10.

Minimizing this KL divergence corresponds exactly to minimizing the cross-entropy between the distributions. Many authors use the term "cross-entropy" to identify specifically the negative log-likelihood of a Bernoulli or softmax distribution, but that is a misnomer. Any loss consisting of a negative log-likelihood

is a cross-entropy between the empirical distribution defined by the training set and the probability distribution defined by model. For example, mean squared error is the cross-entropy between the empirical distribution and a Gaussian model.

We can thus see maximum likelihood as an attempt to make the model distribution match the empirical distribution $\hat{p}_{data}$. Ideally, we would like to match the true data generating distribution p_{data}, but we have no direct access to this distribution.

While the optimal θ is the same regardless of whether we are maximizing the likelihood or minimizing the KL divergence, the values of the objective functions are different. In software, we often phrase both as minimizing a cost function. Maximum likelihood thus becomes minimization of the negative log-likelihood (NLL), or equivalently, minimization of the cross entropy. The perspective of maximum likelihood as minimum KL divergence becomes helpful in this case because the KL divergence has a known minimum value of zero. The negative log-likelihood can actually become negative when $\boldsymbol{x}$ is real-valued.

Advantages Maximum likelihood provides a consistent approach to parameter estimation problems. This means that maximum likelihood estimates can be developed for a large variety of estimation situations.

Maximum likelihood methods have desirable mathematical and optimality properties. Specifically, They become minimum variance unbiased estimators as the sample size increases. By unbiased, we mean that if we take (a very large number of) random samples with replacement from a population, the average value of the parameter estimates will be theoretically exactly equal to the population value. By minimum variance, we mean that the estimator has the smallest variance, and thus the narrowest confidence interval, of all estimators of that type.

They have approximate normal distributions and approximate sample variances that can be used to generate confidence bounds and hypothesis tests for the parameters.

Disadvantages The problem comes about because no matter how many parameters you add to the model, the MLE technique will use them to fit more and more of the data (up to the point at which you have a 100% accurate fit), and a lot of that "fit more and more of the data" is fitting randomness – i.e., overfitting. For example, if I have 100 data points and am fitting a polynomial of degree 99 to the data, MLE will give me a perfect in-sample fit, but that fit won't generalize at all well - I really cannot expect to achieve anywhere near a 100% accurate prediction with this model. Because MLE is not regularized in any way, there's no mechanism within the maximum likelihood framework to prevent this overfitting from occurring. You have to do that yourself, by hand, by structuring and restructuring your model, hopefully appropriately. It does not regulate the number of parameters, i.e. suppose we add regularization term to the loss function which is explained in the next section, the overall estimation

will try to reduce those parameters which lead to overfitting thus regulating the number of parameters indirectly.

MLE is intuitive/naive in that it starts only with the probability of observation given the parameter (i.e. the likelihood function) and tries to find the parameter best accords with the observation. But it take into no consideration the prior knowledge.

3.2 Maximum A posterior Estimation

We showed that minimizing squared error is equivalent of MLE, but this can lead to overfitting. Therefore, we go for Bayesian estimation. In Bayesian approach in addition to likelihood, we also consider a prior probability distribution over parameters w.

A MAP estimated is the choice that is most likely given the observed data. In contrast to MLE, MAP estimation applies Bayes's Rule, so that our estimate can take into account prior knowledge about what we expect our parameters to be in the form of a prior probability distribution.

Remember that Bayesian approach maximizes following posterior probability:

$$p(w|y) = \frac{p(y|w)p(w)}{p(y)} \tag{13}$$

where $p(y)$ is probability of seeing y which is normalizing factor. Therefore, we can write following:

$$\text{posterior } \propto \text{likelihood } \times \text{prior}$$

Suppose, we assume parameters $\mathbf{w}$ are distributed normally as follows:

$$\begin{aligned} p(\mathbf{w}) &= \mathcal{N}(0, \alpha^{-1}I) \\ &= (\frac{\alpha}{2\pi})^{M/2} \exp\big\{ -\frac{1}{2}(\mathbf{w}-\mathbf{0})^T(\alpha^{-1}I)^{-1}(\mathbf{w}-\mathbf{0})\big\} \\ &= (\frac{\alpha}{2\pi})^{M/2} \exp\big\{ -\frac{\alpha}{2}\mathbf{w}^T\mathbf{w}\big\} \end{aligned}$$

where α is a precision of parameters.

Therefore, we can combine all these distributions to produce posterior as:

$$p(w|y,x) \propto p(y|w,x)p(w) \tag{14}$$

$$\log p(w|y,x) \propto \log p(y|w,x) + \log p(w) \tag{15}$$

$$\propto -n\log(\sigma\sqrt{2\pi}) - \sum_{i=1}^{n}\frac{(y_i - f(x_i,w))^2}{2\sigma^2} + \log(\frac{\alpha}{2\pi})^{M/2}) - \frac{\alpha}{2}\mathbf{w}^T\mathbf{w} \tag{16}$$

$$\propto -\big(\sum_{i=1}^{n}\frac{(y_i - f(x_i,w))^2}{2\sigma^2} + \frac{\alpha}{2}\mathbf{w}^T\mathbf{w}\big) - \big(n\log(\sigma\sqrt{2\pi}) - \log(\frac{\alpha}{2\pi})^{M/2})\big) \tag{17}$$

The second term does not depend on w, which means we have to minimize the first term. The first term is nothing but the mean squared error and regularization.

$$\sum_{i=1}^{n} \frac{(y_i - f(x_i, w))^2}{2\sigma^2} + \frac{\alpha}{2}\mathbf{w}^T\mathbf{w}$$

3.3 Estimation for classification problems

So far, we looked at the problem where the output was continuous. Let us see, if we want to predict discrete output $y \in \{0, 1\}$. Let us consider, the y is distributed with Bernoulli distribution with probability $f(x, w)$. Note that, as opposed to regression problem as in previous section where we considered Gaussian Distribution we consider Bernoulli here. The reason is simple, in the previous problem we modelled continuous output variable and here we are modelling discrete random variable. Then the likelihood is given as:

$$p(\mathbf{y}|x, w) = \prod_{i=1}^{n} f(x_i, w)^{y_i}(1 - f(x_i, w))^{(1-y_i)} \tag{18}$$

Taking log of the likelihood:

$$\log p(\mathbf{y}|x, w) = \sum_{i=1}^{n} y_i \log f(x_i, w) + (1 - y_i) \log(1 - f(x_i, w)) \tag{19}$$

Maximizing this log-likelihood is nothing but minimizing entropy as the likelihood function is in the form of entropy.

$$-\sum_{i=1}^{n} y_i \log f(x_i, w) + (1 - y_i) \log(1 - f(x_i, w)) \tag{20}$$

Now, Bayesian treatment to the classification problem can be again done as similar to previous problem. Here again we consider a normal distribution for parameters w. So in this case of classification as well, the loss function becomes of the form:

$$\Big(-\sum_{i=1}^{n} y_i \log f(x_i, w) + (1 - y_i) \log(1 - f(x_i, w))\Big) + \frac{\alpha}{2}\mathbf{w}^T\mathbf{w}$$

4 Evaluating Hypotheses

In many cases it is important to evaluate the performance of learned hypotheses as precisely as possible to understand whether to use the hypothesis. Estimating the accuracy of a hypothesis is relatively straightforward when data is plentiful. However, when we must learn a hypothesis and estimate its future accuracy given only a limited set of data, two key difficulties arise:

- Bias in the estimate. First, the observed accuracy of the learned hypothesis over the training examples is often a poor estimator of its accuracy over future examples. Because the learned hypothesis was derived from these examples, they will typically provide an optimistically biased estimate of hypothesis accuracy over future examples. This is especially likely when the learner considers a very rich hypothesis space, enabling it to overfit the training examples. To obtain an unbiased estimate of future accuracy, we typically test the hypothesis on some set of test examples chosen independently of the training examples and the hypothesis.

- Variance in the estimate. Second, even if the hypothesis accuracy is measured over an unbiased set of test examples independent of the training examples, the measured accuracy can still vary from the true accuracy, depending on the makeup of the particular set of test examples. The smaller the set of test examples, the greater the expected variance.

4.1 Sample Error and True Error

Sample Error is the error rate of the hypothesis over the sample of data that is available. On the other hand, True Error is the error rate of the hypothesis over the entire unknown distribution $\mathcal{D}$ of examples.

Definition 4.1. The sample error (denoted $error_s(h)$) of hypothesis h with respect to target function f and data sample S is

$$error_s(h) = \frac{1}{n} \sum_{x \in S} \delta(f(x), h(x)) \tag{21}$$

Where n is the number of examples in S, and the quantity $\delta(f(x), h(x))$ is 1 if $f(x) \neq h(x)$, and 0 otherwise.

The true error of a hypothesis is the probability that it will misclassify a single randomly drawn instance from the distribution $\mathcal{D}$.

Definition 4.2. The true error (denoted $error_{\mathcal{D}}(h)$) of hypothesis h with respect to target function f and distribution P, is the probability that h will misclassify an instance drawn at random according to $\mathcal{D}$.

$$error_{\mathcal{D}}(h) \equiv \Pr_{x \in \mathcal{D}}[f(x) \neq h(x)] \tag{22}$$

Here the notation $\Pr_{x \in \mathcal{D}}$ denotes that the probability is taken over the instance distribution $\mathcal{D}$.

What we usually wish to know is the true error $error_{\mathcal{D}}(h)$ of the hypothesis, because this is the error we can expect when applying the hypothesis to future examples. All we can measure, however, is the sample error $error_s(h)$ of the hypothesis for the data sample S that we happen to have in hand. The true

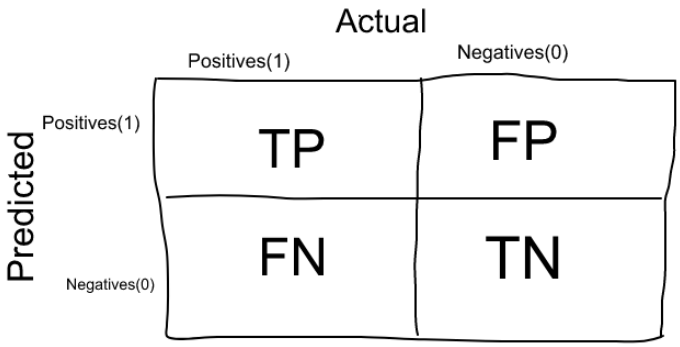


Figure 1: Confusion Matrix.

error is estimated with the use of observed true error. The 95% confidence interval for n samples is given as

$$error_s(h) \pm 1.96\sqrt{\frac{error_s(h)(1 - error_s(h))}{n}} \tag{23}$$

4.2 Validation Set

4.3 Performance Metrics

After doing the usual Feature Engineering, Selection, and of course, implementing a model and getting some output in forms of a probability or a class, the next step is to find out how effective is the model based on some metric using test datasets. Different performance metrics are used to evaluate different Machine Learning Algorithms. For now, we will be focusing on the ones used for Classification problems. We can use classification performance metrics such as Log-Loss, Accuracy, AUC(Area under Curve) etc. Another example of metric for evaluation of machine learning algorithms is precision, recall, which can be used for sorting algorithms primarily used by search engines

4.3.1 Confusion Matrix

The Confusion matrix (Fig. 1) is one of the most intuitive and easiest metrics used for finding the correctness and accuracy of the model. It is used for Classification problem where the output can be of two or more types of classes. The Confusion matrix in itself is not a performance measure as such, but almost all of the performance metrics are based on Confusion Matrix and the numbers inside it.

Let us see the different performance metrics summarized in confusion matrix.

- True Positives (TP): True positives are the cases when the actual class of the data point was 1 (True) and the predicted is also 1 (True).

- True Negatives (TN): True negatives are the cases when the actual class of the data point was 0 (False) and the predicted is also 0 (False).
- False Positives (FP): False positives are the cases when the actual class of the data point was 0 (False) and the predicted is 1 (True). False is because the model has predicted incorrectly and positive because the class predicted was a positive one.
- False Negatives (FN): False negatives are the cases when the actual class of the data point was 1 (True) and the predicted is 0 (False). False is because the model has predicted incorrectly and negative because the class predicted was a negative one.

Based on thse premitive metrics other important metrics are calculated as follows.

- Accuracy: Accuracy in classification problems is the number of correct predictions made by the model over all kinds predictions made.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{24}$$

- Precision: Precision is a measure that tells us what proportion of samples predicted as positive are actually positive.

$$\text{Precision} = \frac{TP}{TP + FP} \tag{25}$$

- Recall: Recall is a measure that tells us what proportion of samples which had positive class i.e. 1 and predicted correctly. It also called as True Positive Rate (TPR) or Sensitivity. This metric is useful for plotting ROC and understanding the effect of different threshold values.

$$\text{Recall} = \frac{TP}{TP + FN} \tag{26}$$

- F1-Score: It is calculated as a harmonic mean of precision and recall.

$$\text{F1-score} = \frac{2PR}{P + R} \tag{27}$$

- Specificity or True Negative Rate: It is calculated as follows:

$$\text{Specificity} = \frac{TN}{TN + FP} \tag{28}$$

- False Positive Rate: It is calculated as $1 - \text{Specificity}$ as:

$$\text{FPR} = \frac{FP}{TN + FP} \tag{29}$$

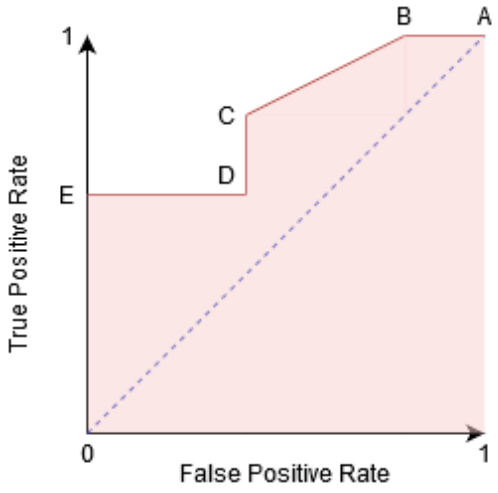


Figure 2: ROC.

4.3.2 ROC

The Receiver Operator Characteristic (ROC) curve is an evaluation metric for binary classification problems. It is a probability curve that plots the TPR against FPR at various threshold values and essentially separates the 'signal' from the 'noise'. The Area Under the Curve (AUC) is the measure of the ability of a classifier to distinguish between classes and is used as a summary of the ROC curve. The higher the AUC, the better the performance of the model at distinguishing between the positive and negative classes.

We have already seen how to calculate TPR in the previous section, here we understand how to calculate FPR.

When AUC = 1, then the classifier is able to perfectly distinguish between all the Positive and the Negative class points correctly. If, however, the AUC had been 0, then the classifier would be predicting all Negatives as Positives, and all Positives as Negatives.

In a ROC curve (Fig. 2), a higher X-axis value indicates a higher number of False positives than True negatives. While a higher Y-axis value indicates a higher number of True positives than False negatives. So, the choice of the threshold depends on the ability to balance between False positives and False negatives.

4.4 Cross-validation

When we are building a machine learning model using some data, we often split our data into training and validation/test sets. The training set is used to train the model, and the validation/test set is used to validate it on data it has never seen before. The classic approach is to do a simple 80%-20% split, sometimes

with different values like 70%-30% or 90%-10%. The problem arises if the data is not enough for such split. This means there is either not enough data to train the model or to validate the model or we can not have good estimate of the model as to how it is going to perform in the future.

When we have very little data, splitting it into training and test set might leave us with a very small test set. Say we have only 100 examples, if we do a simple 80–20 split, we will get 20 examples in our test set. It is not enough. We can get almost any performance on this set only due to chance. The problem is even worse when we have a multi-class problem. If we have 10 classes and only 20 examples, It leaves us with only 2 examples for each class on average. Testing anything on only 2 examples can not lead to any real conclusion.

In cross-validation, we do more than one split. We can do 3, 5, 10 or any K number of splits. Those splits called Folds, and there are many strategies we can create these folds with.

- **Simple K-Folds** We split our data into K parts, let's use K=3 for a toy example. If we have 3000 instances in our dataset, We split it into three parts, part 1, part 2 and part 3. We then build three different models, each model is trained on two parts and tested on the third. Our first model is trained on part 1 and 2 and tested on part 3. Our second model is trained to on part 1 and part 3 and tested on part 2 and so on.

- **Leave One Out** This is the most extreme way to do cross-validation. For each instance in our dataset, we build a model using all other instances and then test it on the selected instance.

- **Stratified Cross Validation** When we split our data into folds, we want to make sure that each fold is a good representative of the whole data. The most basic example is that we want the same proportion of different classes in each fold. Most of the times it happens by just doing it randomly, but sometimes, in complex datasets, we have to enforce a correct distribution for each fold.

When we create K different models using our learning algorithm and test it on K different test sets, we can be more confident in our algorithm performance. When we do a single evaluation on our test set, we get only one result. This result may be because of chance or a biased test set for some reason. By training five (or ten) different models we can understand better what is going on. Say we trained five models and we use accuracy as our measurement. We could end up in several different situations. The best scenario is that our accuracy is similar in all our folds, say 92.0, 91.5, 92.0, 92.5 and 91.8. This means that our algorithm (and our data) is consistent and we can be confident that by training it on all the data set and deploy it in production will lead to similar performance.

However, we could end up in a slightly different scenario, say 92.0, 44.0, 91.5, 92.5 and 91.8. These results look very strange. It looks like one of our folds is

from a different distribution, we have to go back and make sure that our data is what we think it is.

The worst scenario we can end up in is when we have considerable variation in our results, say 80, 44, 99, 60 and 87. Here it looks like that our algorithm or our data (or both) is no consistent, it could be that our algorithm is unable to learn, or our data is very complicated.

By using Cross-Validation, we are able to get more metrics and draw important conclusion both about our algorithm and our data.

5 Supervised Learning

Supervised learning (SL)[1] is the machine learning task of learning a function that maps an input to an output based on example input-output pairs. It infers a function from labelled training data consisting of a set of training examples. In supervised learning, each example is a pair consisting of an input object (typically a vector) and a desired output value (also called the supervisory signal). A supervised learning algorithm analyses the training data and produces an inferred function, which can be used for mapping new examples. An optimal scenario will allow for the algorithm to correctly determine the class labels for unseen instances. This requires the learning algorithm to generalize from the training data to unseen situations in a reasonable way. This statistical quality of an algorithm is measured through the so-called generalization error.

Formally, given a set of N training examples of the form $\{(x_1, y_1), \cdots, (x_N,\ y_N)\}$ such that x_i is the feature vector of the i^{th} example and y_i is its label (i.e., class), a learning algorithm seeks a function $g : X \to Y$, where X is the input space and Y is the output space. The function g is an element of some space of possible functions G, usually called the hypothesis space. It is sometimes convenient to represent g using a scoring function $f : X \times Y \to \mathbb{R}$ such that g is defined as returning the y value that gives the highest score: $g(x) = \arg\max_y f(x, y)$. Let F denote the space of scoring functions.

Although G and F can be any space of functions, many learning algorithms are probabilistic models where g takes the form of a conditional probability model $g(x) = P(y|x)$, or f takes the form of a joint probability model $f(x, y) = P(x, y)$. For example, naive Bayes and linear discriminant analysis are joint probability models, whereas logistic regression is a conditional probability model.

There are two basic approaches to choosing f or g: empirical risk minimization and structural risk minimization.

Empirical risk minimization seeks the function that best fits the training data.

$$R_{emp}(g) = \frac{1}{N} \sum_i \mathcal{L}(y_i, g(x_i)). \tag{30}$$

[1] In human and animal psychology, a parallel task is termed as *concept learning*.

where $\mathcal{L}$ is a loss function which measures the deviation from the desired output and predicted output for each sample.

Structural risk minimization includes a penalty function that controls the bias/variance tradeoff.

$$J(g) = R_{emp}(g) + \lambda C(g) \tag{31}$$

where λ is a penalty controlling constant and $C(g)$ is a penalty term over the model parameters.

5.1 Decision Trees Classifier

Decision trees classify instances by sorting them down the tree from the root to some leaf node, which provides the classification of the instance. Each node in the tree specifies a test of some attribute of the instance, and each branch descending from that node corresponds to one of the possible values for this attribute. An instance is classified by starting at the root node of the tree, testing the attribute specified by this node, then moving down the tree branch corresponding to the value of the attribute in the given example. This process is then repeated for the subtree rooted at the new node.

Decision tree learning is one of the predictive modelling approaches used in statistics, data mining and machine learning. It uses a decision tree (as a predictive model) to go from observations about an item (represented in the branches) to conclusions about the item's target value (represented in the leaves). Tree models where the target variable can take a discrete set of values are called *classification trees*; in these tree structures, leaves represent class labels and branches represent conjunctions of features that lead to those class labels. Decision trees where the target variable can take continuous values (typically real numbers) are called *regression trees*. Decision trees are among the most popular machine learning algorithms given their intelligibility and simplicity.

In general, decision trees represent a disjunction of conjunctions of constraints on the attribute values of instances[2]. Each path from the tree root to a leaf corresponds to a conjunction of attribute tests, and the tree itself to a disjunction of these conjunctions. For example, the decision tree shown in Figure 3 corresponds to the expression

$$\begin{aligned} &(Outlook = Sunny \ \wedge \ Humidity = Normal) \\ \vee\, &(Outlook = Overcast) \\ \vee\, &(Outlook = Rain \ \wedge \ Wind = Weak) \end{aligned}$$

5.1.1 Algorithms to construct decision tree

Most algorithms that have been developed for learning decision trees are variations on a core algorithm that employs a top-down, greedy search through the

[2]Disjunction is logical *or* operation whereas conjunction is logical *and*.

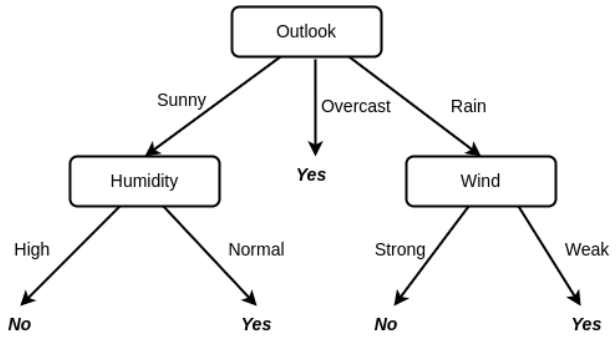


Figure 3: A decision tree for the concept $PlayTennis$. An example is classified by sorting it through the tree to the appropriate leaf node, then returning the classification associated with this leaf (in this case, *Yes* or *No*). This tree classifies Saturday mornings according to whether or not they are suitable for playing tennis.

space of possible decision trees. A basic algorithm, ID3, learns decision trees by constructing them top-down, beginning with the question "which attribute should be tested at the root of the tree?" To answer this question, each instance attribute is evaluated using a statistical test to determine how well it alone classifies the training examples. The best attribute is selected and used as the test at the root node of the tree. A descendant of the root node is then created for each possible value of this attribute, and the training examples are sorted to the appropriate descendant node (i.e., down the branch corresponding to the example's value for this attribute). The entire process is then repeated using the training examples associated with each descendant node to select the best attribute to test at that point in the tree. This forms a greedy search for an acceptable decision tree, in which the algorithm never backtracks to reconsider earlier choices.

5.1.2 Information Gain to Select Attributes

The central choice in various algorithms is selecting which attribute to test at each node in the tree. We would like to select the attribute that is most useful for classifying examples. What is a good quantitative measure of the worth of an attribute? We will define a statistical property, called information gain, that measures how well a given attribute separates the training examples according to their target classification. Information gain measure is used to select among the candidate attributes at each step while growing the tree.

Let us first see how entropy is calculated. Entropy is defined as

$$Entropy(S) \equiv -\sum_{i=1}^{N} p_i \log p_i \tag{32}$$

Given entropy as a measure of the impurity in a collection of training examples, we can now define a measure of the effectiveness of an attribute in classifying the training data. The measure we will use, called information gain, is simply the expected reduction in entropy caused by partitioning the examples according to this attribute. More precisely, the information gain, $Gain(S, A)$ of an attribute A relative to a collection of examples S, is defined as

$$Gain(S, A) \equiv Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v) \tag{33}$$

Let us contemplate how this gain is calculated for certain attribute. First, we calculate total entropy of the given data, this is calculated based on the class distribution of the output variable (y). If the distribution is skewed then the entropy is less, intuitively this means that there is a less uncertainty in the dataset. On the contrary, the entropy is higher if there is more evenness in the distribution, which means more uncertainty which output we will see.

Then we select attributes one by one and calculate the relative entropy. Suppose, we select an attribute A to see what will be the information gain if we choose A. Suppose, it takes two values, a_1 and a_2, then we calculate how many samples are of a_1 value and how many are of a_2. After that, we select a value a_1 of attribute A and see what is the distribution of output variable y only with these values, based on that entropy is calculated. Similarly, for a_2 value also entropy is calculated. Then both these entropies are multiplied with the proportion of those values from the total samples: $\frac{|S_{a_1}|}{|S|}$ and $\frac{|S_{a_2}|}{|S|}$. This value is subtracted from the total entropy to get the information gain.

Let us think about this for a moment, what are we actually gaining with this calculation. Suppose, there is only one attribute and one output variable. Suppose, the attribute takes value a_1 when the output is $True$ and a_2 when output is $False$. This means that the relative entropy will be 0 as entropy is zero when there is no uncertainty. This means the second term in Eq. 33 is 0, that means there is maximum information gain with this single attribute, thus we can decide the value of the output with only this attribute.

5.1.3 Gini Index

It is an alternative measure to understand the importance of the predictor at any given branch. It is also known as Gini impurity, calculates the amount of probability of a specific feature that is classified incorrectly when selected randomly. If all the elements are linked with a single class then it can be called pure.

Let's perceive the criterion of the Gini Index, like the properties of entropy, the Gini index varies between values 0 and 1, where 0 expresses the purity of classification, i.e. All the elements belong to a specified class or only one class exists there. And 1 indicates the random distribution of elements across various classes. The value of 0.5 of the Gini Index shows an equal distribution of elements over some classes.

While designing the decision tree, the features possessing the least value of the Gini Index would get preferred. You can learn another tree-based algorithm(Random Forest). The Gini Index is determined by deducting the sum of squared of probabilities of each class from one, mathematically, Gini Index can be expressed as:

$$\text{Gini Index} = 1 - \sum_{i=1}^{n} p_i^2 \tag{34}$$

Where p_i denotes the probability of an element being classified for a distinct class.

Classification and Regression Tree (CART) algorithm deploys the method of the Gini Index to originate binary splits. In addition, decision tree algorithms exploit Information Gain to divide a node and Gini Index or Entropy is the passageway to weigh the Information Gain.

Gini Index vs Information Gain

- The Gini Index facilitates the bigger distributions so easy to implement whereas the Information Gain favors lesser distributions having small count with multiple specific values.
- The method of the Gini Index is used by CART algorithms, in contrast to it, Information Gain is used in ID3, C4.5 algorithms.
- Gini index operates on the categorical target variables in terms of "success" or "failure" and performs only binary split, in opposite to that Information Gain computes the difference between entropy before and after the split and indicates the impurity in classes of elements.

5.1.4 Inductive Bias in Decision Trees

What is the policy by which ID3 generalizes from observed training examples to classify unseen instances? In other words, what is its inductive bias? Inductive bias is the set of assumptions that, together with the training data, deductively justify the classifications assigned by the learner to future instances.

Given a collection of training examples, there are typically many decision trees consistent with these examples. Describing the inductive bias of ID3 therefore consists of describing the basis by which it chooses one of these consistent hypotheses over the others. Which of these decision trees does ID3 choose? It chooses the first acceptable tree it encounters in its simple-to-complex, hill-climbing search through the space of possible trees. Roughly speaking, then, the ID3 search strategy (a) selects in favor of shorter trees over longer ones, and (b) selects trees that place the attributes with highest information gain closest to the root. Because of the subtle interaction between the attribute selection heuristic used by ID3 and the particular training examples it encounters, it is difficult to characterize precisely the inductive bias exhibited by ID3. However,

we can approximately characterize its bias as a preference for short decision trees over complex trees.

Disadvantages

- Trees fail to deal with linear relationships. Any linear relationship between an input feature and the outcome has to be approximated by splits, creating a step function. This is not efficient.
- This goes hand in hand with lack of smoothness. Slight changes in the input feature can have a big impact on the predicted outcome, which is usually not desirable. Imagine a tree that predicts the value of a house and the tree uses the size of the house as one of the split feature. The split occurs at 100.5 square meters. Imagine user of a house price estimator using your decision tree model: They measure their house, come to the conclusion that the house has 99 square meters, enter it into the price calculator and get a prediction of 200 000 Euro. The users notice that they have forgotten to measure a small storage room with 2 square meters. The storage room has a sloping wall, so they are not sure whether they can count all of the area or only half of it. So they decide to try both 100.0 and 101.0 square meters. The results: The price calculator outputs 200 000 Euro and 205 000 Euro, which is rather unintuitive, because there has been no change from 99 square meters to 100.
- Trees are also quite unstable. A few changes in the training dataset can create a completely different tree. This is because each split depends on the parent split. And if a different feature is selected as the first split feature, the entire tree structure changes. It does not create confidence in the model if the structure changes so easily.
- Decision trees are very interpretable – as long as they are short. The number of terminal nodes increases quickly with depth. The more terminal nodes and the deeper the tree, the more difficult it becomes to understand the decision rules of a tree. A depth of 1 means 2 terminal nodes. Depth of 2 means max. 4 nodes. Depth of 3 means max. 8 nodes. The maximum number of terminal nodes in a tree is 2 to the power of the depth.

5.2 Linear Regression

It is a linear approach for modelling the relationship between a scalar response and one or more explanatory variables (also known as dependent and independent variables). The case of one explanatory variable is called simple linear regression; for more than one (fitting vector valued output variable), the process is called multiple linear regression (not to be confused with multivariate linear regression).

In linear regression, the relationships are modelled using linear predictor functions whose unknown model parameters are estimated from the data. Such

models are called *linear models*. It it important to note that the "linear" term in this algorithm comes from the linear parameters and not from the linear "features". In fact, it is common to use non-linear features such as n^{th} power of the features or some non-linear function of the features. And, because the prediction from the model depends linearly on their unknown parameters, it is easier to fit than models which are non-linearly related to their parameters. In turn, it is easier to determine the statistical properties of the resulting estimators. This ease of training and inference lead to extensive use of linear regression analysis.

Linear regression has been mainly used for following two purposes:

1. If the goal is prediction, forecasting, or error reduction, linear regression can be used to fit a predictive model to an observed data set of values of the response and explanatory variables. After developing such a model, if additional values of the explanatory variables are collected without an accompanying response value, the fitted model can be used to make a prediction of the response.

2. If the goal is to explain variation in the response variable that can be attributed to variation in the explanatory variables, linear regression analysis can be applied to quantify the strength of the relationship between the response and the explanatory variables, and in particular to determine whether some explanatory variables may have no linear relationship with the response at all, or to identify which subsets of explanatory variables may contain redundant information about the response.

Linear regression models are often fitted using the least squares approach, but they may also be fitted in other ways, such as by minimizing the "lack of fit" in some other norm (as with least absolute deviations regression), or by minimizing a penalized version of the least squares cost function as in ridge regression with L2-norm penalty and lasso with L1-norm penalty. Conversely, the least squares approach can be used to fit models that are not linear models. Thus, although the terms "least squares" and "linear model" are closely linked, they are not synonymous.

Given a data set $\{y_i, , x_{i1}, \ldots, x_{ip}\}_{i=1}^{n}$ of n statistical units, a linear regression model assumes that the relationship between the dependent variable y and the p-vector of regressors x is linear. This relationship is modelled through a disturbance term or error variable ϵ– an unobserved random variable that adds "noise" to the linear relationship between the dependent variable and regressors. Thus the model takes the form

$$y_i = \beta_0 + \beta_1 x_{i1} + \cdots + \beta_p x_{ip} + \epsilon_i = \mathbf{x}_i^{\mathsf{T}} \boldsymbol{\beta} + \epsilon_i, \qquad i = 1, \ldots, n \tag{35}$$

where T denotes the transpose, so that $\mathbf{x}_i^{\mathsf{T}} \boldsymbol{\beta}$ is the inner product between vectors $\mathbf{x}_i$ and $\boldsymbol{\beta}$. Note, that the true value of y_i is difficult to obtain even though we perfectly obtain the parameters because of the error term. The terms $\mathbf{x}$ and β are considered to be known once we train the model, the only uncertainty is from the unknown error which is a variance σ as stated in the next equation.

The linear regression model also has probabilistic perspective, we assume that the likelihood is distributes normally as

$$P(y_i|\mathbf{x}_i;\beta) = \mathcal{N}(\mathbf{x}_i^\mathsf{T}\boldsymbol{\beta}, \sigma^2) \tag{36}$$

For training, to fit the data to the model, the parameters can be estimated with maximum likelihood principle by reducing the following least squared loss function.

$$\mathcal{L}(\beta) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \mathbf{x}_i^\mathsf{T}\boldsymbol{\beta})^2 \tag{37}$$

In the least-squares setting, the optimum parameter is defined as such that minimizes the sum of mean squared loss:

$$\vec{\hat{\beta}} = \arg\min_{\vec{\beta}} L\left(D, \vec{\beta}\right) = \arg\min_{\vec{\beta}} \sum_{i=1}^{n}\left(\vec{\beta}\cdot\vec{x_i} - y_i\right)^2 \tag{38}$$

This can be solved with gradient descent algorithm where the parameter β are directed towards optimal values.

Analytical Solution In the case of linear regression we can derive an analytic solution as well to get the β values

$$\begin{aligned} L\left(D, \vec{\beta}\right) &= \|X\vec{\beta} - Y\|^2 \\ &= \left(X\vec{\beta} - Y\right)^\mathsf{T}\left(X\vec{\beta} - Y\right) \\ &= Y^\mathsf{T}Y - Y^\mathsf{T}X\vec{\beta} - \vec{\beta}^\mathsf{T}X^\mathsf{T}Y + \vec{\beta}^\mathsf{T}X^\mathsf{T}X\vec{\beta} \end{aligned}$$

As the loss is convex the optimum solution lies at gradient zero. The gradient of the loss function is (using Denominator layout convention):

$$\begin{aligned} \frac{\partial L\left(D, \vec{\beta}\right)}{\partial \vec{\beta}} &= \frac{\partial\left(Y^\mathsf{T}Y - Y^\mathsf{T}X\vec{\beta} - \vec{\beta}^\mathsf{T}X^\mathsf{T}Y + \vec{\beta}^\mathsf{T}X^\mathsf{T}X\vec{\beta}\right)}{\partial\vec{\beta}} \\ &= -2X^\mathsf{T}Y + 2X^\mathsf{T}X\vec{\beta} \end{aligned} \tag{39}$$

Setting the gradient to zero produces the optimum parameter:

$$\begin{aligned} &-2X^\mathsf{T}Y + 2X^\mathsf{T}X\vec{\beta} = 0 \\ &\Rightarrow X^\mathsf{T}Y = X^\mathsf{T}X\vec{\beta} \\ &\Rightarrow \vec{\hat{\beta}} = \left(X^\mathsf{T}X\right)^{-1}X^\mathsf{T}Y \end{aligned} \tag{40}$$

This gives a closed form solution for the parameters of linear regression. In practical implementations the gradient decent approach is used instead of the closed form analytic approach because of the computational complexity. The matrix X is going to be huge of size $n \times m$ where n are number of samples which

can be in the range of million and the features can be thousand. The transpose and multiplication of them leads to 10^12 operations and inverting this matrix is again going to lead computational overhead. Thus in reality gradient based optimization approach is used instead of the analytical solution.

5.2.1 Assumptions

- Linear Assumption: We assume that the input features and output labels are related linearly, thus the function which we are learning is linear in the parameters. If they are not linear then we are not going to find such function with the model. However the neat trick is to use higher powers of the features as augmented features to make them rich and then use a linear regression model over it.

- IID assumption: It is assumed that the samples on which the model is trained comes from the some unknown but identical distribution. If the data samples are derived from the different distributions then there is a no way to model such data. Other assumption is that each data sample is independent of each other unlike the time series data where the samples depend on the previous samples. **Now, if independent assumption is broken, then what happens?**

- Homodescacity: Suppose we calculate *residuals* at each given input points after training the linear regression model. The *residual* is a difference between the actual prediction and the true value of the output at that point. Now, imagine line on the graph and residuals around them. Homodescacity says that these residuals are going to be bounded for all the labels i.e. these are not going to diverge as the samples increase. In another sense they are going to be follow a normal distribution. If this condition is not met this means that the model is underfit and we have to add some more non-linear features in the model to learn more effectively.

- No missing feature values: We assume that all the training samples as well as the test samples are going to have all the features values in them.

- No correlation between features otherwise it is better to do the feature selection.

- Fixed features :The input features are considered "fixed". Fixed means that they are treated as given constants and not as statistical variables. This implies that they are free of measurement errors. This is a rather unrealistic assumption. Without that assumption, however, you would have to fit very complex measurement error models that account for the measurement errors of your input features. And usually you do not want to do that.

- Absence of multicollinearity: You do not want strongly correlated features, because this messes up the estimation of the weights. In a situation where two features are strongly correlated, it becomes problematic to estimate the

weights because the feature effects are additive and it becomes indeterminable to which of the correlated features to attribute the effects.

5.3 Logistic Regression

The linear regression model can work well for regression, but fails for classification. Why is that? In case of two classes, you could label one of the classes with 0 and the other with 1 and use linear regression. Technically it works and most linear model programs will spit out weights for you. But there are a few problems with this approach:

A linear model does not output probabilities, but it treats the classes as numbers (0 and 1) and fits the best hyperplane (for a single feature, it is a line) that minimizes the distances between the points and the hyperplane. So it simply interpolates between the points, and you cannot interpret it as probabilities.

A linear model also extrapolates and gives you values below zero and above one. This is a good sign that there might be a smarter approach to classification.

Since the predicted outcome is not a probability, but a linear interpolation between points, there is no meaningful threshold at which you can distinguish one class from the other.

Linear models do not extend to classification problems with multiple classes. You would have to start labeling the next class with 2, then 3, and so on. The classes might not have any meaningful order, but the linear model would force a weird structure on the relationship between the features and your class predictions. The higher the value of a feature with a positive weight, the more it contributes to the prediction of a class with a higher number, even if classes that happen to get a similar number are not closer than other classes.

The logistic model is a statistical model that models the probability of one event, out of two alternatives, taking place by having the log-odds for the event be a linear combination of one or more independent variables (predictors). In regression analysis, logistic regression (logit regression) is estimating the parameters of a logistic model, the coefficients in the linear combination. Formally, in binary logistic regression there is a single binary dependent variable, coded by a indicator variable, where the two values are labeled "0" and "1", while the independent variables can each be a categorical variables or a continuous variable. To get the legitimate probability for output "1", after the linear or affine transformation of features, the function that converts real values to probability is the logistic function, hence the name.

Logistic function is a type of sigmoid function (see Appendix A) which converts real values into the range of [0,1], therefore, different sigmoid functions instead of logistic function can be used to model the probability i.e. to convert the linear combination to a probability. Though, the logistic function is the most commonly used sigmoid function to model the probability as it is a natural parameter for the Bernoulli distribution and it maximizes entropy (minimizes added information), and in this sense makes the fewest assumptions of the data being modeled.

The parameters of a logistic regression are most commonly estimated by maximum-likelihood estimation (MLE). This does not have a closed-form expression, unlike linear least squares. Logistic regression by MLE plays a similarly basic role for binary or categorical responses as linear regression by ordinary least squares (OLS) plays for scalar responses: it is a simple, well-analyzed baseline model.

The basic setup of logistic regression is as follows. We are given a dataset containing N points. Each point i consists of a set of m input variables $x_{1,i}, \cdots, x_{m,i}$ also called independent variables, predictor variables, features, or attributes, and a binary outcome variable y_i also known as a dependent variable, response variable, output variable, or class, i.e. it can assume only the two possible values 0 or 1. The goal of logistic regression is to use the dataset to create a predictive model of the outcome variable.

Logistic regression uses the mechanism already developed for linear regression by modeling the probability p_i using a linear predictor function, i.e. a linear combination of the explanatory variables and a set of regression coefficients that are specific to the model. The *linear predictor function* $f(x_i)$ for a particular data point x_i is written as:

$$f(x_i) = \beta_0 + \beta_1 x_{1,i} + \cdots + \beta_m x_{m,i} = \mathbf{x}_i^{\mathsf{T}} \boldsymbol{\beta} \tag{41}$$

where βs are regression coefficients indicating the relative effect of a particular explanatory variable on the outcome.

Further, this real value is converted to probability by passing it through sigmoid function.

$$p_i = \sigma(f(x_i)) = \frac{1}{1 + e^{-\mathbf{x}_i^{\mathsf{T}} \boldsymbol{\beta}}} \tag{42}$$

Next to learn these linear parameters, the usual measure of goodness of fit for a logistic regression uses logistic loss (or log loss), the negative log-likelihood.

$$\mathcal{L}(\beta) = -\sum_{j=1}^{N} (\, y_j \ln(p_j) + (1 - y_j) \ln(1 - p_j)) \tag{43}$$

The log loss can be interpreted as the "surprisal" of the actual outcome y_k relative to the prediction p_k, and is a measure of information content. Note that log loss is always greater than or equal to 0, equals 0 only in case of a perfect prediction: $p_k = 1 if y_k = 1, p_k = 0 if y_k = 0$, and approaches infinity as the prediction gets worse, meaning the actual outcome is "more surprising". Since the value of the logistic function is always strictly between zero and one, the log loss is always greater than zero and less than infinity. Note that unlike in a linear regression, where the model can have zero loss at a point by passing through a data point (and zero loss overall if all points are on a line), in a logistic regression it is not possible to have zero loss at any points, since y_k is either 0 or 1, but $0 < p_k < 1$.

5.3.1 Multinomial Logistic Regression

Let us consider a more general problem than binary response variable. Suppose, the number of categories over which we have to classify is K, then the current model is extended in following ways.

Specifically, it is assumed that we have a series of N observed data points, $\{\mathbf{y_i}, x_{1,i}, \ldots, x_{m,i}\}_{i=1}^{n}$, where x_{ij} are features and $\mathbf{y_i}$ is a vector associated with the point i. It contains 1 at certain class k out of possible K classes. Then the previous discussion of binary logistic regression is applied for each class, the linear predictor function is given as:

$$f(k, x_i) = \beta_{0,k} + \beta_{1,k}x_{i1} + \beta_{2,k}x_{2,i} + \cdots + \beta_{m,k}x_{m,i} = \mathbf{x}_i^T \boldsymbol{\beta}_k \tag{44}$$

where a linear predictor function $f(k, i)$ produces real valued score that indicates instance has outcome k and the coefficients β are similar to previous discussion. Note that, here we will learn K sets of β parameters, each for the class.

Now, to convert real values corresponding to all the classes, *softmax* function is used as follows:

$$p_{i,k} = \text{softmax}(f(k, x_i)) = \frac{e^{-\mathbf{x}_i^\intercal \boldsymbol{\beta}_k}}{\sum_{k'=1}^{K} e^{-\mathbf{x}_i^\intercal \boldsymbol{\beta}_{k'}}} \tag{45}$$

Then, generalized formula of negative log-loss, cross entropy is minimized to obtain optimal regression coefficients.

$$\mathcal{L}(\beta) = -\sum_{j=1}^{N}\sum_{i=1}^{K} y_{j,k} \ln(p_{j,k}) \tag{46}$$

5.4 Naive Bayes Classifier

One highly practical Bayesian learning method is the naive Bayes learner, often called the naive Bayes classifier. In some domains its performance has been shown to be comparable to that of neural network and decision tree learning.

The naive Bayes classifier applies to learning tasks where each instance x is described by a conjunction of attribute values and where the target function $f(x)$ can take on any value from some finite set V. A set of training examples of the target function is provided, and a new instance is presented, described by the tuple of attribute values $(a_1, a_2, \cdots, a_n)$. The learner is asked to predict the target value, or classification, for this new instance. The Bayesian approach to classifying the new instance is to assign the most probable target value v_{map} given the attribute values $(a_1, a_2, \cdots, a_n)$ that describe the instance.

$$v_{map} = \underset{v_j \in V}{\arg\max} P(v_j | a_1, a_2, \cdots, a_n) \tag{47}$$

We can use Bayes theorem to rewrite this expression using joint probability of input features and output labels as

$$v_{map} = \arg\max_{v_j \in V} \frac{P(a_1, a_2, \cdots, a_n, v_j)}{P(a_1, a2, \cdots, a_n)} \tag{48}$$

$$v_{map} = \arg\max_{v_j \in V} \frac{P(a_1, a_2, \cdots, a_n | v_j) P(v_j)}{P(a_1, a_2, \cdots, a_n)} \tag{49}$$

Since the denominator in the equation 49 does not depend on the maximizing factors we can neglect that term. This leads to the following equation

$$v_{map} = \arg\max_{v_j \in V} P(a_1, a_2, \cdots, a_n | v_j) P(v_j) \tag{50}$$

Now we could attempt to estimate the two terms in Equation 50 based on the training data. It is easy to estimate each of the $P(v_j)$ simply by counting the frequency with which each target value v_j occurs in the training data. However, estimating the different $P(a_1, a2, \cdots, a_n | v_j)$ terms in this fashion is not feasible unless we have a very, very large set of training data. The problem is that the number of these terms is equal to the number of possible instances times the number of possible target values. Therefore, we need to see every instance in the instance space many times in order to obtain reliable estimates.

Note from the formulation that to model the conditional probability of labels given the input features, we are modelling the joint probability of features and output label. This shows Naive Bayes is a *generative model*.

The naive Bayes classifier is based on the simplifying assumption that the attribute values are conditionally independent given the target value. In other words, the assumption is that given the target value of the instance, the probability of observing the conjunction is just the product of the probabilities for the individual attributes. Substituting this into Equation 50, we have the approach used by the naive Bayes classifier.

$$v_{NB} = \arg\max_{v_j \in V} \prod_{i=1}^{n} P(a_i | v_j) P(v_j) \tag{51}$$

To summarize, the naive Bayes learning method involves a learning step in which the various $P(v_j)$ and $P(a_i|v_j)$ terms are estimated, based on their frequencies over the training data. The set of these estimates corresponds to the learned hypothesis. This hypothesis is then used to classify each new instance by applying the rule in Equation 51. *Whenever the naive Bayes assumption of conditional independence is satisfied, this naive Bayes classification v_{NB} is identical to the MAP classification.*

One interesting difference between the naive Bayes learning method and other learning methods we have considered is that there is no explicit search through the space of possible hypotheses (in this case, the space of possible hypotheses is the space of possible values that can be assigned to the various $P(v_j)$ and $P(a_i|vj)$ terms). Instead, the hypothesis is formed without searching, simply by counting the frequency of various data combinations within the training examples.

5.4.1 Difficulty in Estimating Probabilities

The probability for any attribute given the output is given by counting the number of instances of the attribute (n_c) divided by the total number instances of producing the output (n).

Suppose, in our sample there is no instance of the attribute and the output combination. This means we have $n_c = 0$ which a very biased estimate. This dominates the total decision process of naive bayes classifier as multiplying any term with this 0 probability will cancel out the whole term. To mitigate the issue, probability is estimated with adjusted formula as

$$\frac{n_c + mp}{n + m} \tag{52}$$

Here, n_c and n are defined as before, p is our prior estimate of the probability we wish to determine, and m is a constant called the equivalent sample size, which determines how heavily to weight p relative to the observed data. A typical method for choosing p in the absence of other information is to assume uniform prior; that is, if an attribute has k possible values we set $p = \frac{1}{k}$.

5.5 SVM

Support vector machines (SVMs) are supervised learning models with associated learning algorithms that analyze data for classification and regression analysis. SVM maps training examples to points in space so as to maximise the width of the gap between the two categories. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall. In addition to performing linear classification, SVMs can efficiently perform a non-linear classification using what is called the kernel trick, implicitly mapping their inputs into high-dimensional feature spaces.

The SVM allows for a geometric way to think about supervised machine learning. The SVM view of machine learning is subtly different from the maximum likelihood view i.e. probabilistic view of the data distribution, from which an optimization problem is derived. In contrast, the SVM view starts by designing a particular function that is to be optimized during training, based on geometric intuitions.

Separating Hyperplanes The main idea in most of the machine learning algorithms is to come up with such a hyperplane which can divide the R^D dimensional data into two parts so as to assign either one of the label.

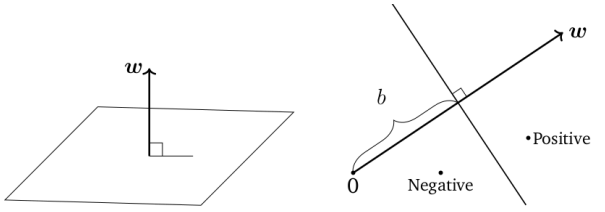
Let example $\mathbf{x} \in R^D$ be an element of the data space. Consider a function

$$\begin{aligned} f &: R^D \to R \\ \mathbf{x} &\mapsto f(\mathbf{x}) := \langle \mathbf{w}, \mathbf{x} \rangle + b \end{aligned}$$

parametrized by $\mathbf{w} \in R^D$ and $b \in R$. Hyperplanes are affine subspaces. Therefore, the hyperplane that separates the two classes in our binary classification

problem as

$$\mathbf{x} \in R^D : f(\mathbf{x}) = 0 \tag{53}$$



(a) Separating hyperplane in 3D

(b) Projection of the setting in (a) onto a plane

Figure 4: Separating hyperplane: two views.

An illustration of the hyperplane is shown in Figure 4, where the vector $\mathbf{w}$ is a vector normal to the hyperplane and b the intercept. We can derive that $\mathbf{w}$ is a normal vector to the hyperplane by choosing any two examples $\mathbf{x}_1$ and $\mathbf{x}_2$ on the hyperplane and showing that the vector between them is orthogonal to $\mathbf{w}$. In the form of an equation,

$$\begin{aligned} f(\mathbf{x}_1) - f(\mathbf{x}_2) &= (\langle \mathbf{w}, \mathbf{x}_1 \rangle + b) - (\langle \mathbf{w}, \mathbf{x}_2 \rangle + b) \\ &= \langle \mathbf{w}, \mathbf{x}_1 - \mathbf{x}_2 \rangle \end{aligned}$$

Since we have chosen $\mathbf{x}_1$ a and $\mathbf{x}_2$ to be on the hyperplane, this implies that $f(\mathbf{x}_1) = 0$ and $f(\mathbf{x}_2) = 0$ and hence $\langle \mathbf{w}, \mathbf{x}_1 - \mathbf{x}_2 \rangle = 0$. Recall that two vectors are orthogonal when their inner product is zero. Therefore, we obtain that $\mathbf{w}$ is orthogonal to any vector on the hyperplane.

Inference With this hyperplane intuition, let us see how we classify the test examples. The examples are classified as positive or negative depending on the side of the hyperplane on which it occurs. Concretely, to classify a test example $\mathbf{x}_{test}$, we calculate the value of the function $f(\mathbf{x}_{test})$ and classify the example as +1 if $f(\mathbf{x}_{test}) > 0$ and -1 otherwise. Thinking geometrically, the positive examples lie "above" the hyperplane and the negative examples "below" the hyperplane.

This inference idea gives the intuition behind the training of SVMs. When training the classifier, we want to ensure that the examples with positive labels are on the positive side of the hyperplane, i.e.,

$$\langle \mathbf{w}, \mathbf{x}_n \rangle + b \geqslant 0 \qquad \text{when } y_n = +1 \tag{54}$$

Conversely for negative example,

$$\langle \mathbf{w}, \mathbf{x}_n \rangle + b < 0 \qquad \text{when } y_n = -1 \tag{55}$$

These two conditions are often presented in a single equation

$$y_n(\langle \mathbf{w}, \mathbf{x}_n \rangle + b) \geqslant 0 \tag{56}$$

5.5.1 Concept of Large Margin

Based on the concept of distances from points to a hyperplane, we now are in a position to discuss the support vector machine. For a linearly separable dataset, we have infinitely many candidate hyperplanes, and therefore classifiers, that solve our classification problem without any training errors. To find a unique solution, one idea is to choose the separating hyperplane that maximizes the margin between the positive and negative examples. In other words, we want the positive and negative examples to be separated by a large margin.

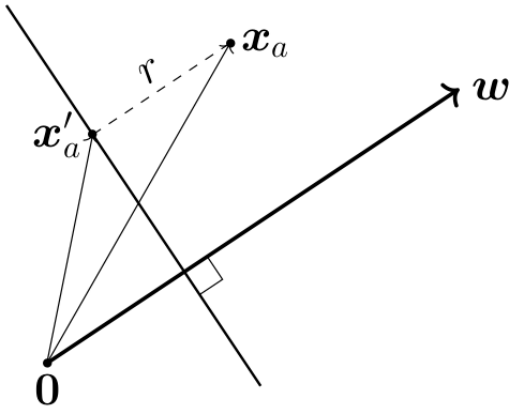


Figure 5: Distance of the point from the hyperplane.

Consider a hyperplane $\langle \mathbf{w}, \mathbf{x}_n \rangle + b$, and an example $\mathbf{x}_a$ as illustrated in Figure 5. Without loss of generality, we can consider the example $\mathbf{x}_a$ to be on the positive side of the hyperplane, i.e., $\langle \mathbf{w}, \mathbf{x}_a \rangle + b > 0$. We would like to compute the distance $r > 0$ of $\mathbf{x}_a$ from the hyperplane. We do so by considering the orthogonal projection of $\mathbf{x}_a$ onto the hyperplane, which we denote by $\mathbf{x}'_a$. Since $\mathbf{w}$ is orthogonal to the hyperplane, we know that the distance r is just a scaling of this vector $\mathbf{w}$. If the length of $\mathbf{w}$ is known, then we can use this scaling factor r factor to work out the absolute distance between $\mathbf{x}_a$ and $\mathbf{x}'_a$. For convenience, we choose to use a vector of unit length (its norm is 1) and obtain this by dividing $\mathbf{w}$ by $||\mathbf{w}||$. Using vector additions, we obtain

$$\mathbf{x}_a = \mathbf{x}'_a + r\frac{\mathbf{w}}{||\mathbf{w}||} \tag{57}$$

Recall that we would like the positive examples to be further than r from the hyperplane, and the negative examples to be further than distance r in

the negative direction from the hyperplane. Combination of these conditions, produces following objective

$$y_n(\langle \mathbf{w}, \mathbf{x}_n \rangle + b) \geqslant r \tag{58}$$

In other words, we combine the requirements that examples are at least r away from the hyperplane, both in the positive and negative direction, into one single inequality.

Collecting these requirements into a single constrained optimization

$$\begin{aligned}
&\underset{\mathbf{w}, b, r}{\text{maximize}} \quad r && \text{(59a)}\\
&\text{subject to} \\
&y_n(\langle \mathbf{w}, \mathbf{x}_n \rangle + b) \geqslant r, && \text{(59b)}\\
&||\mathbf{w}|| = 1 \qquad \forall n = 1, \cdots, N && \text{(59c)}
\end{aligned}$$

which says that we want to maximize the margin r while ensuring that the data lies on the correct side of the hyperplane.

5.5.2 Max-margin derivation

We derive the margin maximization problem by making a assumption that the closest point from the hyperplane in the positive direction is at 1 unit distance. This means that $\langle \mathbf{w}, \mathbf{x}_i \rangle + b = 1$, for points $\mathbf{x}_i$s.

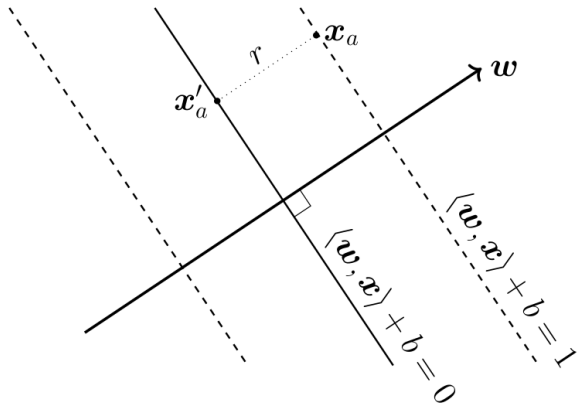


Figure 6: Geometric representation of SVM margin with $r = \frac{1}{||\mathbf{w}||}$.

As shown in Figure 6 $\mathbf{x}_a$ lies exactly on the margin, i.e.,

$$\langle \mathbf{w}, \mathbf{x}_a \rangle + b = 1 \tag{60}$$

Since $\mathbf{x}'_a$ is the orthogonal projection of $\mathbf{x}_a$ onto the hyperplane, it must by definition lie on the hyperplane, i.e.,

$$\langle \mathbf{w}, \mathbf{x}'_a \rangle + b = 0 \tag{61}$$

Then we can solve this,

$$\langle \mathbf{w}, (\mathbf{x}_a - r\frac{\mathbf{w}}{||\mathbf{w}||}) \rangle + b = 0 \quad \cdots \text{ from Eq. 57} \tag{62}$$

$$\langle \mathbf{w}, \mathbf{x}_a \rangle + b - r\frac{\langle \mathbf{w}, \mathbf{w} \rangle}{||\mathbf{w}||}) = 0 \quad \cdots \text{ follows from linearity of dot product} \tag{63}$$

$$1 - r||\mathbf{w}|| = 0 \quad \cdots \text{ substituting Eq. 60} \tag{64}$$

$$r = \frac{1}{||\mathbf{w}||} \tag{65}$$

This gives us margin in the form of the hyperplane. Then, the new optimization object is defined as:

$$\underset{\mathbf{w}, b}{\text{maximize}} \quad \frac{1}{||\mathbf{w}||} \tag{66a}$$

subject to

$$y_n(\langle \mathbf{w}, \mathbf{x}_n \rangle + b) \geq 1 \qquad \forall n = 1, \cdots, N \tag{66b}$$

Instead of maximizing the reciprocal of the norm, we often minimize the squared norm. We also often include a constant $1/2$ that does not affect the optimal $\mathbf{w}, b$ but yields a tidier form when we compute the gradient. Then, our objective becomes

$$\underset{\mathbf{w}, b}{\text{minimize}} \quad \frac{1}{2}||\mathbf{w}||^2 \tag{67a}$$

subject to

$$y_n(\langle \mathbf{w}, \mathbf{x}_n \rangle + b) \geq 1 \qquad \forall n = 1, \cdots, N \tag{67b}$$

This is known as the hard margin SVM. The reason for the expression "hard" is because the formulation does not allow for any violations of the margin condition.

5.5.3 Soft margin SVM

In the case where data is not linearly separable, we may wish to allow some examples to fall within the margin region, or even to be on the wrong side of the hyperplane as illustrated in Figure 7.

The key geometric idea is to introduce a slack variable ξ_n corresponding to each example-label pair $(\mathbf{x}_n, y_n)$ that allows a particular example to be within

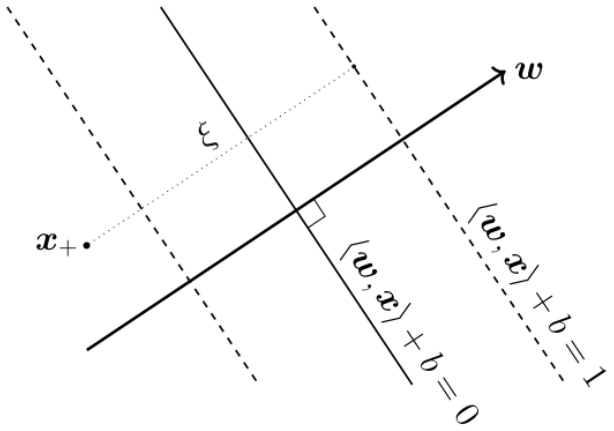


Figure 7: Soft margin SVM.

the margin or even on the wrong side of the hyperplane (refer to Figure 7). We subtract the value of ξ_n from the margin, constraining ξ_n to be non-negative. To encourage correct classification of the samples, we add ξ_n to the objective

$$\underset{\mathbf{w}, b}{\text{minimize}} \quad \frac{1}{2}||\mathbf{w}||^2 + C\sum_n \xi_n \tag{68a}$$

subject to

$$y_n(\langle \mathbf{w}, \mathbf{x}_n \rangle + b) \geq 1 - \xi_n \qquad \forall n = 1, \cdots, N, \tag{68b}$$

$$\xi_n \geq 0 \qquad \forall n = 1, \cdots, N \tag{68c}$$

This is called a soft margin objective and the parameter $C > 0$ trades off the size of the margin and the total amount of slack that we have.

5.5.4 SVM loss function

The ideal loss function in the binary classification problem would be measure the number of wrong predictions, however, differentiating that function is difficult. Therefore, we look for a different loss function by looking at the soft-margin svm objective and equivalent way is to have a *hinge loss*:

$$\max\{0, 1 - y_n(\langle \mathbf{w}, \mathbf{x}_n \rangle + b)\} \tag{69}$$

This loss is easy to understand. Suppose, the prediction is accurate which means $y_n(\langle \mathbf{w}, \mathbf{x}_n \rangle + b) \geqslant 1$, then the second term will become ≤ 0, so the overall loss become 0. On the contrary, the prediction is incorrect, then $y_n(\langle \mathbf{w}, \mathbf{x}_n \rangle + b) < 1$ i.e. lesser than the margin. This in turn will make the second term positive, which will contribute to the loss.

Alternative way of putting the loss function

$$\ell(t) = \begin{cases} 0 & if\ t \geq 1 \\ 1-t & if\ t < 1 \end{cases}$$

The loss corresponding to hard margin SVM is given as:

$$\ell(t) = \begin{cases} 0 & if\ t \geq 1 \\ \infty & if\ t < 1 \end{cases}$$

With that intuition and definition of hinge loss, we write the SVM loss function as:

$$\min_{\mathbf{w},b} \frac{1}{2}||\mathbf{w}||^2 + C\sum_n \max\{0, 1 - y_n(\langle \mathbf{w}, \mathbf{x}_n \rangle + b)\} \tag{70}$$

Observe that Eq. 70 and objective function mentioned in the optimization Eq. 68c are equivalent.

Support vectors in SVM are those vectors which are closest to the decision boundary. These are the only points which are necessary for the decision in a way, because the other points which are further from the decision plane will not contribute in the loss function therefore the support vectors are the most important. Mathematically they are on 1 unit distance from the decision boundary and are given by the equation: $y_n(\langle \mathbf{w}, \mathbf{x}_n \rangle + b) = 1$

5.5.5 Multi-class classification with SVM

In binary classification with SVM first we find out the vector which can produce a score for the given input vector. In case of multi-class classification, instead of multiplying with vector we multiply it with matrix which is going to produce m scores for each class.

Formally, consider $x \in R^d$ which is a d-dimensional feature vector. Then SVM learns matrix $W \in R^{d \times m}$ and $b \in R^m$ bias to get the score vector as

$$S = W^T + b \tag{71}$$

The loss function is defined for pair x_i, y_i where x_i is a input feature vector and y_i is a true label.

$$\ell_i = \sum_{j \neq y_i} \max\left(0, S_j - S_{y_i} + 1\right) \tag{72}$$

The idea is that the score for the wrong labels should be lesser than the true labels other wise there will be loss incurred.

5.5.6 Ranking SVM

Consider for given input x we have four options $o = \{a, b, c, d\}$ and we know there ranking. Then the idea is to find the score for each $z_i = [x, o_i]$ pair and assign scores depending on there ranks.

The scores are calculated as earlier where separate parameters are learned for each option as:

$$
\begin{aligned}
s_a &= w_a^T \ z_a \\
s_b &= w_b^T \ z_b \\
s_c &= w_c^T \ z_c \\
s_d &= w_d^T \ z_d
\end{aligned}
$$

Suppose the rankings are $a > b > c > d$ then the scores should be in the same order $s_a > s_b > s_c > s_d$. The loss is calculated on each pair.

5.5.7 Kernels

5.6 K-nearest neighbours

K-nearest neighbours is a family of techniques that can be used for classification or regression. As a non-parametric learning algorithm, k-nearest neighbours is not restricted to a fixed number of parameters. We usually think of the k-nearest neighbours algorithm as not having any parameters, but rather implementing a simple function of the training data. In fact, there is not even really a training stage or learning process. Instead, at test time, when we want to produce an output y for a new test input x, we find the k-nearest neighbours to x in the training data $\mathbf{X}$. We then return the average of the corresponding y values in the training set. In the case of classification, we can average over one-hot code vectors c with $c_y = 1$ and $c_i = 0$ for all other values of i. We can then interpret the average over these one-hot codes as giving a probability distribution over classes.

One simple refinement over the k-nearest neighbour is to weight the output of each of the neighbours according to the inverse of the distance. Therefore, in this type of refinement, higher weight is given to the output of the closest point. Another, important tweak can be to use different distance measure than the euclidean distance to know the nearest k-neighbours.

As a non-parametric learning algorithm, k-nearest neighbour can achieve very high capacity. For example, suppose we have a multiclass classification task and measure performance with 0-1 loss. In this setting, 1 -nearest neighbour converges to double the Bayes error as the number of training examples approaches infinity. The error in excess of the Bayes error results from choosing a single neighbour by breaking ties between equally distant neighbours randomly. When there is infinite training data, all test points x will have infinitely

many training set neighbours at distance zero. If we allow the algorithm to use all of these neighbours to vote, rather than randomly choosing one of them, the procedure converges to the Bayes error rate.

The high capacity of k-nearest neighbours allows it to obtain high accuracy given a large training set. However, it does so at high computational cost, and it may generalize very badly given a small, finite training set. One weakness of k-nearest neighbours is that it cannot learn that one feature is more discriminative than another. For example, imagine we have a regression task with $x \in R^{100}$ drawn from an isotropic Gaussian distribution, but only a single variable x_1 is relevant to the output. Suppose further that this feature simply encodes the output directly, i.e. that $y = x_1$ in all cases. Nearest neighbour regression will not be able to detect this simple pattern. The nearest neighbour of most points x will be determined by the large number of features x_2 through x_{100}, not by the lone feature x_1. Thus the output on small training sets will essentially be random.

The value of k is decided based on the performance measure on the validation set. We select k which produces the highest performance over validation set.

One of the difficulty wit this algorithm is that at the inference we have to compare each test sample with all the n training samples. Suppose each data sample is of d-dimension then for each data sample in the test set the time complexity will be $O(nd)$ just to get the distance of the test sample from the n samples. Then to get the nearest k samples, we can again iterate all the distances k times which gives total complexity of $O(nd + nk)$. The other way is that, we can store these distances and sort them which will be $O(n \log n)$, so total complexity will be $O(nd + n \log n)$.

This is not a efficient way of applying the KNN algorithm. The trick is to create K-d tree at the training time and then at the inference time calculation of distances is more efficient than the linear time complexity which we saw. In the training phase, the creation K-d tree will be more computationally complex than just storing the values. K-d tree is generic version of Binary tree of K dimension which is stored based on each dimension of the vector of K-d. At each level we decide which data samples to put either to left of the tree or right of the tree. At the inference time finding k nearest neighbor is going to take $O(k \log n)$, i.e. $\log n$ time for each k.

6 Unsupervised Learning

Unsupervised learning is a type of algorithm that learns patterns from untagged data. The hope is that through mimicry, which is an important mode of learning in people, the machine is forced to build a compact internal representation of its world and then generate imaginative content from it. In contrast to supervised learning where data is tagged by an expert, unsupervised methods exhibit self-organization that captures patterns as probability densities or a combination of neural feature preferences. The other levels in the supervision spectrum are reinforcement learning where the machine is given only a numerical performance

score as guidance, and semi-supervised learning where a smaller portion of the data is tagged.

Dimensionality reduction is one type of unsupervised learning where unlabelled data is used. These algorithms convert the high dimensional data into low dimensions while keeping as much information intact in the data. Working directly with high-dimensional data, such as images, comes with some difficulties: It is hard to analyze, interpretation is difficult, visualization is nearly impossible, and from a practical point of view storage of the data vectors can be expensive. However, high-dimensional data often has properties that we can exploit. For example, high-dimensional data is often overcomplete, i.e., many dimensions are redundant and can be explained by a combination of other dimensions. Furthermore, dimensions in high-dimensional data are often correlated so that the data possesses an intrinsic lower-dimensional structure. Dimensionality reduction exploits structure and correlation and allows us to work with a more compact representation of the data, ideally without losing information. We can think of dimensionality reduction as a compression technique, similar to jpeg or mp3, which are compression algorithms for images and music.

6.1 Principal Component Analysis

The principal components of a collection of points in a real coordinate space are a sequence of p unit vectors, where the i^{th} vector is the direction of a line that best fits the data while being orthogonal to the first $i-1$ vectors. Here, a best-fitting line is defined as one that minimizes the average squared distance from the points to the line. These directions constitute an orthonormal basis in which different individual dimensions of the data are linearly uncorrelated. Principal component analysis (PCA) is the process of computing the principal components and using them to perform a change of basis on the data, sometimes using only the first few principal components and ignoring the rest.

In data analysis, the first principal component of a set of p variables, presumed to be jointly normally distributed, is the derived variable formed as a linear combination of the original variables that explains the most variance. The second principal component explains the most variance in what is left once the effect of the first component is removed, and we may proceed through p iterations until all the variance is explained. PCA is most commonly used when many of the variables are highly correlated with each other and it is desirable to reduce their number to an independent set.

It is commonly used for dimensionality reduction by projecting each data point onto only the first few principal components to obtain lower-dimensional data while preserving as much of the data's variation as possible. The first principal component can equivalently be defined as a direction that maximizes the variance of the projected data. The i i^{th} principal component can be taken as a direction orthogonal to the first $i-1$ principal components that maximizes the variance of the projected data.

For either objective, it can be shown that the principal components are eigenvectors of the data's covariance matrix. Thus, the principal components

are often computed by eigendecomposition of the data covariance matrix or singular value decomposition of the data matrix. PCA is the simplest of the true eigenvector-based multivariate analyses and is closely related to factor analysis.

6.1.1 PCA as Eigendecomposition

PCA learns a representation that has lower dimensionality than the original input. It also learns a representation whose elements have no linear correlation with each other. This is a first step toward the criterion of learning representations whose elements are statistically independent. To achieve full independence, a representation learning algorithm must also remove the nonlinear relationships between variables.

PCA learns an orthogonal, linear transformation of the data that projects an input x to a representation z. We could learn a one-dimensional representation that best reconstructs the original data in the sense of mean squared error, and that this representation actually corresponds to the first principal component of the data. Thus we can use PCA as a simple and effective dimensionality reduction method that preserves as much of the information in the data as possible, again, as measured by least-squares reconstruction error.

Mathematical Treatment Suppose we are given $\{\mathbf{x}_i\}_{i=1}^n$ vectors such that $x_i \in R^d$. The goal is to project each vector onto a m-dimensional space where $m \lll d$. Now there can be many vector spaces which will have m-dimensions, the constraint is that there should be maximum variation maintained from the data otherwise if all the projected points have less variation then there will be information loss.

Suppose u is the unit vector where we want to project the given vectors with the constraint that the projection should have maximum variance. Formally we can write proejction vector as $p_i = u^T x_i u$ and calculate mean of these projections as

$$\bar{p} = \frac{1}{n}\sum_{i=1}^{n} u^T x_i u \tag{73}$$

$$= \big(\frac{1}{n}\sum_{i=1}^{n} u^T x_i\big) u \tag{74}$$

$$= u^T(\frac{1}{n}\sum_{i=1}^{n} x_i) u \tag{75}$$

$$\bar{p} = u^T \bar{x} u \tag{76}$$

The first equation is the definition of calculation of sample mean. In equation 75 we do some clever manupluation. Imagine multiplying each vector with u and then summing up them all, in this case we can take the u components out leading to addition of all x_i components which is written in vector form as in the equation.

Now let us calculate variance of the projection

$$Var = \frac{1}{n}\sum_{i=1}^{n}(u^T x_i - u^T \bar{x})^2 \tag{77}$$

$$= \frac{1}{n}\sum_{i=1}^{n} u^T (x_i - \bar{x})^2 \tag{78}$$

$$= \frac{1}{n}\sum_{i=1}^{n} u^T (x_i - \bar{x})(x_i - \bar{x})^T u \tag{79}$$

$$= u^T \frac{1}{n}\sum_{i=1}^{n} \big((x_i - \bar{x})(x_i - \bar{x})^T\big) u \tag{80}$$

$$Var = u^T S u \tag{81}$$

where S is a variance matrix.

After writing the projection equation, I realized that the projection of vector x_i is going to be just some scalar multiplied vector itself as the first term of dot product is going to produce a scalar value. This makes as we are projecting some vector on the other that vector is going to be just scalar multiplied vector! This took me some time to realise it!

The next step is to solve this where we want to maximize the variance of projection which leads to the following formulation.

$$\underset{u}{\text{maximize}} \quad u^T S u \tag{82a}$$

$$\text{subject to}$$

$$u^T u = 1 \tag{82b}$$

Because of the constraint of the unit vector, it is not easy to solve the optimization function. Therefore we apply Lagrangian multiplier to get the following formulation

$$\max_u \; u^T S u + \lambda(1 - u^T u)$$

To get the maximum value of the above equation, we take the gradient and equate it to 0.

$$\frac{\partial\big(u^T S u + \lambda(1 - u^T u)\big)}{\partial u}$$
$$= 2Su - \lambda 2u$$

Equating this to 0

$$Su = \lambda u \tag{83}$$

This looks similar to eigenvalue formulation where λ is nothing but the eigenvector of matrix S which can also be written as

$$\lambda = u^T S u$$

This is similar to the objective we wanted to maximize which means if we maximize the eigenvalue of the covariance matrix or select the highest eigenvalue then we are going to get the projection which will have the highest variance.

Let us examine how the PCA representation decorrelates the original data representation $\mathbf{X}$. Let us consider the $m \times n$-dimensional design matrix $\mathbf{X}$. We will assume that the data has a mean of zero, $E[x] = 0$. If this is not the case, the data can easily be centred by subtracting the mean from all examples in a preprocessing step. Because we assume that the data is centered we can write the covariance matrix as follows:

$$\begin{aligned}\text{Var}(\mathbf{x}) &= \frac{1}{m-1}\sum_i (\mathbf{x}_i - \mu)^2 = \frac{1}{m-1}\sum_i (\mathbf{x}_i)^2 = \frac{1}{m-1}\sum_i \mathbf{x}_i^T \mathbf{x}_i \\ &= \frac{1}{m-1}\mathbf{X}^T\mathbf{X}\end{aligned}$$

PCA finds a representation through linear transformation $z = x^T W$ where Var$[z]$ is diagonal. The diagonal Var matrix means that there is no correlation between the distinct features of z as the fundamental goal of the PCA algorithm is to find such features which are linearly uncorrelated.

We know that the principal components of a design matrix $\mathbf{X}$ are given by the eigenvectors of $\mathbf{X}^T\mathbf{X}$. From this view,

$$\mathbf{X}^T\mathbf{X} = \mathbf{W}\Lambda\mathbf{W}^T \tag{84}$$

The principal components may also be obtained via the singular value decomposition. Specifically, they are the right singular vectors of $\mathbf{X}$. To see this, let $\mathbf{W}$ be the right singular vectors in the decomposition $\boldsymbol{X = U\Sigma W^T}$. We then recover the original eigenvector equation with $\mathbf{W}$ as the eigenvector basis:

$$\boldsymbol{X^T X = (U\Sigma W^T)^T (U\Sigma W^T) = W\Sigma^2 W^T} \tag{85}$$

The SVD is helpful to show that PCA results in a diagonal Var$[z]$. Using the SVD of $\mathbf{X}$, we can express the variance of $\mathbf{X}$ as:

$$\text{Var}(\mathbf{x}) = \frac{1}{m-1}\mathbf{X}^T\mathbf{X} = \frac{1}{m-1}\boldsymbol{W\Sigma^2 W^T}$$

where we use the fact that $\mathbf{U}^T\mathbf{U} = \mathbf{I}$ because the $\mathbf{U}$ matrix of the singular value decomposition is defined to be orthogonal. This shows that if we take $z = x^T\mathbf{W}$, we can ensure that the covariance of z is diagonal as required:

$$
\begin{aligned}
\text{Var}(\mathbf{x}) &= \frac{1}{m-1}\mathbf{Z}^T\mathbf{Z} \\
&= \frac{1}{m-1}\mathbf{W}^T\mathbf{X}^T\mathbf{X}\mathbf{W} \\
&= \frac{1}{m-1}\mathbf{W}^T\boldsymbol{W}\boldsymbol{\Sigma^2}\boldsymbol{W^T}\mathbf{W} \\
&= \frac{1}{m-1}\boldsymbol{\Sigma^2}
\end{aligned}
$$

where this time we use the fact that $\boldsymbol{W^TW} = \mathbf{I}$, again from the definition of the SVD.

The above analysis shows that when we project the data x to z, via the linear transformation $\mathbf{W}$, the resulting representation has a diagonal covariance matrix as given by Σ^2 which immediately implies that the individual elements of z are mutually uncorrelated.

This mutually uncorrelated representation attempts to disentangle the unknown factors of variation underlying the data. In the case of PCA, this disentangling takes the form of finding a rotation of the input space described by $\mathbf{W}$ that aligns the principal axes of variance with the basis of the new representation space associated with z .

While correlation is an important category of dependency between elements of the data, we are also interested in learning representations that disentangle more complicated forms of feature dependencies. For this, we will need more than what can be done with a simple linear transformation.

6.1.2 Key Steps of PCA in Practice

To obtain the lower dimensional projections of the data, we can also use the iterative optimization technique where following least squared loss can be reduced:

$$
\boldsymbol{J}(B) = \frac{1}{m}\sum_{i}^{m}||\mathbf{x}_i - \tilde{\mathbf{x}}_i||^2 = \frac{1}{m}\sum_{i}^{m}||\mathbf{x}_i - \mathbf{B}\mathbf{B}^T\mathbf{x}_i||^2 \tag{86}
$$

where $\boldsymbol{J}(B)$ is a reconstruction error with projection $\mathbf{z}_i = \mathbf{B}^T\mathbf{x}_i$ and reconstruction as $\tilde{\mathbf{x}}_i = \mathbf{B}\mathbf{z}_i = \mathbf{B}\mathbf{B}^T\mathbf{x}_i$.

Generally in practice, instead of solving this iterative optimization problem, SVD or Eigenvalue approach is used. Following are the steps followed in practice to obtain lower dimensional projection with PCA.

1. **Mean subtraction** We start by centring the data by computing the mean μ of the dataset and subtracting it from every single data point. This ensures that the dataset has mean 0. Mean subtraction is not strictly necessary but reduces the risk of numerical problems.

2. **Standardization** Divide the data points by the standard deviation σ_d of the dataset for every dimension $d = 1, \cdots, D$. Now the data is unit free,

and it has variance 1 along each axis. This step completes the standardization of the data.

3. **Eigendecomposition of the covariance matrix** Compute the data covariance matrix and its eigenvalues and corresponding eigenvectors. Since the covariance matrix is symmetric, the spectral theorem states that we can find an Orthogonal Normal Basis (ONB) of eigenvectors. responding eigenvalue. The longer vector spans the principal subspace, which we denote by U . The data covariance matrix is represented by the ellipse.

4. **Projections** The lower dimensional vector representation for x_* is obtained as

$$z_* = \mathbf{B}^T x_* \tag{87}$$

Here, $\mathbf{B}$ is the matrix that contains the eigenvectors that are associated with the largest eigenvalues of the data covariance matrix as columns.

6.2 Linear Discriminant Analysis

6.3 Gaussian Mixture Model

When we apply machine learning to data we often aim to represent data in some way. In density estimation, we represent the data compactly using a density from a parametric family, e.g., a Gaussian or Beta distribution. For example, we may be looking for the mean and variance of a dataset in order to represent the data compactly using a Gaussian distribution. The mean and variance can be found using tools we discussed earlier: maximum likelihood or maximum a posteriori estimation. We can then use the mean and variance of this Gaussian to represent the distribution underlying the data, i.e., we think of the dataset to be a typical realization from this distribution if we were to sample from it.

In practice, the Gaussian or similarly all other distributions we encountered so far have limited modeling capabilities. For example, a Gaussian approximation of the density that generated the data in Figure 8 would be a poor approximation. Rather it will be more accurate if we assume the underline density to be mixture of multiple distributions. In the following sections we elaborate the technique of Gaussian mixture models where we assume that the data is generated by combining multiple Gaussian densities.

A Gaussian mixture model is a density model where we combine a finite number of K Gaussian distributions $\mathcal{N}(\mathbf{x}|\mu_k, \mathbf{\Sigma}_k)$ so that

$$p(\mathbf{x}|\theta) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\mu_k, \mathbf{\Sigma}_k) \tag{88}$$

where we defined $\theta := \{\mu_k, \mathbf{\Sigma}_k, \pi_k : k = 1, \cdots, K\}$ as the collection of all parameters of the model. This convex combination of Gaussian distribution gives us significantly more flexibility for modeling complex densities than a simple Gaussian distribution.

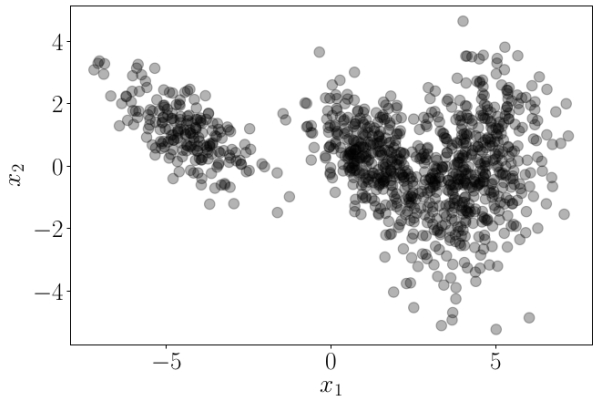


Figure 8: A data with combination of two Gaussian distributions.

6.3.1 Parameter Learning via Maximum Likelihood

Assume we are given a dataset $\mathcal{X} = \{x_1, \cdots, x_N\}$, where $x_n, n = 1, \cdots, N$, are drawn i.i.d. from an unknown distribution $p(x)$. Our objective is to find a good approximation/representation of this unknown distribution $p(x)$ by means of a GMM with K mixture components. The parameters of the GMM are the K means μ_k, the covariances $\mathbf{\Sigma}_k$, and mixture weights π_k. We summarize all these free parameters in $\theta \coloneqq \{\mu_k, \mathbf{\Sigma}_k, \pi_k : k = 1, \cdots, K\}$.

By exploiting i.i.d. assumption of the data, we can write the likelihood as:

$$p(\mathcal{X}|\theta) = \prod_{n=1}^{N} p(\mathbf{x}_n|\theta), \qquad p(\mathbf{x}_n|\theta) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n|\mu_k, \mathbf{\Sigma}_k) \tag{89}$$

Then, the log-likelihood is calculated as

$$\mathcal{L} \coloneqq \log p(\mathcal{X}|\theta) = \sum_{n=1}^{N} \log p(\mathbf{x}_n|\theta) = \sum_{n=1}^{N} \log \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n|\mu_k, \mathbf{\Sigma}_k) \tag{90}$$

We aim to find the parameters θ_{MLE} that maximize the log-likelihood $\mathcal{L}$. However, obtaining closed form solution as we did earlier is not possible. Hence, iterative *Expectation Maximization* algorithm is used to attain optimal values.

Partial derivative with the parameter values is given as:

$$\frac{\partial \log p(\mathbf{x}_n|\theta)}{\partial \theta} = \frac{1}{p(\mathbf{x}_n|\theta)} \frac{\partial p(\mathbf{x}_n|\theta)}{\partial \theta} \tag{91}$$

where

$$\frac{1}{p(\mathbf{x}_n|\theta)} = \frac{1}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_n|\mu_j, \mathbf{\Sigma}_j)} \tag{92}$$

Before, looking at the approach of updates of parameters θ, we first familiarize with a term: responsibilities.

Responsibilities A responsibility of the k^{th} mixture component for n^{th} data sample is defined as:

$$r_{nk} \coloneqq \frac{\pi_k \mathcal{N}(\mathbf{x}_n|\mu_k, \mathbf{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_n|\mu_j, \mathbf{\Sigma}_j)} \tag{93}$$

The mixture components have a high responsibility for a data point when the data point could be a plausible sample from that mixture component. Note that $r_n \coloneqq [r_{n1}, \cdots, r_{nK}]^T \in R^K$ is a (normalized) probability vector, i.e. $r_{nk} \geq 0$ and $\sum_k r_{nk} = 1$. This is possible because of the normalizing term in the denominator.

Update rule for μ Partial derivative of the probability of $\mathbf{x}_n$ w.r.t. μ_k is calculated as:

$$\begin{aligned}
\frac{\partial \log p(\mathbf{x}_n|\theta)}{\partial \mu_k} &= \sum_{j=1}^{K} \pi_j \frac{\partial \mathcal{N}(\mathbf{x}_n|\mu_j, \mathbf{\Sigma}_j)}{\partial \mu_k} \\
&= \pi_k \frac{\partial \mathcal{N}(\mathbf{x}_n|\mu_k, \mathbf{\Sigma}_k)}{\partial \mu_k} \qquad \cdots \text{ for other terms derivative will be 0} \\
\frac{\partial \log p(\mathbf{x}_n|\theta)}{\partial \mu_k} &= \pi_k (\mathbf{x}_n - \mu_k)^T \mathbf{\Sigma}_k^{-1} \mathcal{N}(\mathbf{x}_n|\mu_k, \mathbf{\Sigma}_k)
\end{aligned}$$

Now, calculating the partial derivative over all the data points, i.e. derivative of log-likelihood is given as:

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \mu_k} &= \sum_{n=1}^{N} \frac{\partial \log p(\mathbf{x}_n|\theta)}{\partial \mu_k} \\
&= \sum_{n=1}^{N} \frac{1}{p(\mathbf{x}_n|\theta)} \frac{\partial p(\mathbf{x}_n|\theta)}{\partial \mu_k} \\
&= \sum_{n=1}^{N} (\mathbf{x}_n - \mu_k)^T \mathbf{\Sigma}_k^{-1} \frac{\pi_k \mathcal{N}(\mathbf{x}_n|\mu_k, \mathbf{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_n|\mu_j, \mathbf{\Sigma}_j)} \\
\frac{\partial \mathcal{L}}{\partial \mu_k} &= \sum_{n=1}^{N} r_{nk} (\mathbf{x}_n - \mu_k)^T \mathbf{\Sigma}_k^{-1}
\end{aligned}$$

Equating this partial derivative to 0, we get,

$$\sum_{n=1}^{N} r_{nk}\mathbf{x}_n = \sum_{n=1}^{N} r_{nk}\mu_k^{new} \tag{94}$$

$$\iff \mu_k^{new} = \frac{1}{\sum_{n=1}^{N} r_{nk}} \sum_{n=1}^{N} r_{nk}\mathbf{x}_n \tag{95}$$

$$\mu_k^{new} = \frac{1}{N_k} \sum_{n=1}^{N} r_{nk}\mathbf{x}_n \tag{96}$$

where $N_k \coloneqq \sum_{n=1}^{N} r_{nk}$ is total responsibility for k^{th} mixture component for the whole dataset.

Intuitively, the update rule can be interpreted as an importance-weighted Monte Carlo estimate of the mean, where the importance weights of data point $\mathbf{x}_n$ are the responsibilities r_{nk} of the k^{th} cluster for $\mathbf{x}_n$, $k = 1, \cdots, K$.

Therefore, the mean μ_k is pulled toward a data point $\mathbf{x}_n$ with strength given by r_{nk}. The means are pulled stronger toward data points for which the corresponding mixture component has a high responsibility, i.e., a high likelihood.

The update of the mean parameters in Eq. 96 look fairly straightforward. However, note that the responsibilities r_{nk} are a function of $\pi_j, \mu_j, \mathbf{\Sigma}_j \; \forall j = 1, \cdots, K$, such that the updates in Eq. 96 depend on all parameters of the GMM, and a closed-form solution, which we obtained for linear regression cannot be obtained.

Update rule for $\mathbf{\Sigma}$ We start with calculating partial derivative w.r.t. $\mathbf{\Sigma}_k$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{\Sigma}_k} = \sum_{n=1}^{N} \frac{1}{p(\mathbf{x}_n|\theta)} \frac{\partial p(\mathbf{x}_n|\theta)}{\partial \mathbf{\Sigma}_k}$$

The second term is calculated as:

$$\frac{\partial p(\mathbf{x}_n|\theta)}{\partial \mathbf{\Sigma}_k} = \pi_k \mathcal{N}(\mathbf{x}_n|\mu_k, \mathbf{\Sigma}_k).[-\frac{1}{2}(\mathbf{\Sigma}_k^{-1} - \mathbf{\Sigma}_k^{-1}(\mathbf{x}_n - \mu_k)(\mathbf{x}_n - \mu_k)^T\mathbf{\Sigma}_k^{-1})] \tag{97}$$

Substituting it in the previous equation, we get

$$\frac{\partial \mathcal{L}}{\partial \mathbf{\Sigma}_k} = \sum_{n=1}^{N} \frac{\pi_k \mathcal{N}(\mathbf{x}_n|\mu_k, \mathbf{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_n|\mu_j, \mathbf{\Sigma}_j)}.[-\frac{1}{2}(\mathbf{\Sigma}_k^{-1} - \mathbf{\Sigma}_k^{-1}(\mathbf{x}_n - \mu_k)(\mathbf{x}_n - \mu_k)^T\mathbf{\Sigma}_k^{-1})] \tag{98}$$

$$= -\frac{1}{2}\mathbf{\Sigma}_k^{-1} \sum_{n=1}^{N} r_{nk} + \frac{1}{2}\mathbf{\Sigma}_k^{-1} \Big(\sum_{n=1}^{N} r_{nk}(\mathbf{x}_n - \mu_k)(\mathbf{x}_n - \mu_k)^T \Big) \mathbf{\Sigma}_k^{-1} \tag{99}$$

Equating this to 0, we obtain the update rule as:

$$\boldsymbol{\Sigma}_k^{new} = \frac{1}{N_k} \sum_{n=1}^{N} r_{nk}(\mathbf{x}_n - \mu_k)(\mathbf{x}_n - \mu_k)^T \tag{100}$$

We can interpret the update of the covariance in Eq. 100 as an importance-weighted expected value of the square of the centered data.

Updating Mixture Weights The partial derivative of the log-likelihood with respect to the weight parameters $\pi_k, k = 1, \cdots, K$, we account for the constraint $\sum_k \pi_k = 1$ by using Lagrange multipliers. The Lagrangian is

$$\begin{aligned}
\mathfrak{L} &= \mathcal{L} + \lambda\Big(\sum_{k=1}^{K} \pi_k - 1\Big) \\
&= \sum_{n=1}^{N} \log \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}_n | \mu_k, \boldsymbol{\Sigma}_k) + \lambda\Big(\sum_{k=1}^{K} \pi_k - 1\Big)
\end{aligned}$$

Partial derivate is given as

$$\frac{\partial \mathfrak{L}}{\partial \pi_k} = \sum_{n=1}^{N} \frac{\mathcal{N}(\mathbf{x}_n | \mu_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_n | \mu_j, \boldsymbol{\Sigma}_j)} + \lambda \tag{101}$$

$$= \frac{1}{\pi_k} \sum_{n=1}^{N} \frac{\pi_k \mathcal{N}(\mathbf{x}_n | \mu_k, \boldsymbol{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_n | \mu_j, \boldsymbol{\Sigma}_j)} + \lambda \tag{102}$$

$$= \frac{1}{\pi_k} \sum_{n=1}^{N} r_{nk} + \lambda \tag{103}$$

$$0 = \frac{N_k}{\pi_k} + \lambda \tag{104}$$

$$\iff \quad \pi_k = -\frac{N_k}{\lambda} \tag{105}$$

Now partial derivative with λ is

$$\frac{\partial \mathfrak{L}}{\partial \lambda} = \sum_{k=1}^{K} \pi_k - 1 \tag{106}$$

$$\iff \quad \sum_{k=1}^{K} \pi_k = 1 \tag{107}$$

$$\sum_{k=1}^{K} -\frac{N_k}{\lambda} = 1 \qquad \text{from eq. 105} \iff \qquad -\frac{N}{\lambda} = 1 \tag{108}$$

$$\iff \quad \lambda = -N \tag{109}$$

Combining Equations 105 and 109, we get

$$\pi_k^{new} = \frac{N_k}{N} \tag{110}$$

6.3.2 Expectation Maximization

Unfortunately, the updates for the parameters $\mu_k, \mathbf{\Sigma}_k, \pi_k$ do not constitute a closed-form solution because the responsibilities r_{nk} depend on those parameters in a complex way. However, the results suggest a simple iterative scheme for finding a solution to the parameters estimation problem via maximum likelihood. The expectation maximization algorithm (EM algorithm) is a general iterative scheme for learning parameters (maximum likelihood or MAP) in mixture models and, more generally, latent-variable models.

Every step in the EM algorithm increases the log-likelihood function. For convergence, we can check the log-likelihood or the parameters directly. A concrete instantiation of the EM algorithm for estimating the parameters of a GMM is as follows:

1. Initialize $\mu_k, \mathbf{\Sigma}_k, \pi_k$.
2. *E-step*: Evaluate responsibilities r_nk for every data point $\mathbf{x}_n$ using current parameters $\mu_k, \mathbf{\Sigma}_k, \pi_k$:

$$r_{nk} \coloneqq \frac{\pi_k \mathcal{N}(\mathbf{x}_n | \mu_k, \mathbf{\Sigma}_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(\mathbf{x}_n | \mu_j, \mathbf{\Sigma}_j)} \tag{111}$$

3. *M-step*: Re-estimate parameters $\mu_k, \mathbf{\Sigma}_k, \pi_k$ using the current responsibilities r_nk evaluated from the previous step:

$$\mu_k^{new} = \frac{1}{N_k} \sum_{n=1}^{N} r_{nk} \mathbf{x}_n \tag{112}$$

$$\mathbf{\Sigma}_k^{new} = \frac{1}{N_k} \sum_{n=1}^{N} r_{nk} (\mathbf{x}_n - \mu_k)(\mathbf{x}_n - \mu_k)^T \tag{113}$$

$$\pi_k^{new} = \frac{N_k}{N} \tag{114}$$

6.4 K-means clustering

Another example of a simple representation learning algorithm is k-means clustering. The k-means clustering algorithm divides the training set into k different clusters of examples that are near each other. We can thus think of the algorithm as providing a k-dimensional one-hot code vector h representing an input

Algorithm 1 K-means Clustering Algorithm

Require: Number of clusters K
Require: Training data $\{x_1, \cdots, x_n\}$
Initialize K **centroids**: $\mu_1, \cdots, \mu_K \leftarrow$`ChooseKRadom(`$\{x_1, \cdots, x_n\}$`)`
while stopping criteria not met **do**
    Initialize: $w_k \leftarrow \{\}$ $\quad \forall k = 1, \cdots, K$
    $i = 1$
    while $i \leqslant n$ **do**
        $j \leftarrow \arg\min_j |\mu_{j'} - x_i|$
        $w_j \leftarrow w_j \cup x_i$
        $i \leftarrow i + 1$
    $k = 1$
    while $k \leqslant K$ **do**
        $\mu_k \leftarrow \frac{1}{|w_k|} \sum w_k$
        $k \leftarrow k + 1$

x. If x belongs to cluster i, then $h_i = 1$ and all other entries of the representation h are zero. The one-hot code provided by k-means clustering is an example of a sparse representation, because the majority of its entries are zero for every input. Algorithm 1 details all the steps of this algorithm.

There is a possibility that we do not know the value of k before executing the algorithm. One of the way of solving such issue is to start with linear search with some values of k. Then with a single value of k we create clusters and measure the appropriateness of the clusters. We measure the distance between clusters and understand how far they are how different they are. We can put some threshold to know if these are good clusters. If those clusters are better we can stop at that value of k.

With most of the k-means algorithms the distance measure is Euclidean Distance, does this mean we are going to find out only hyperspherical boundary for the samples to put into the clusters. It makes sense also, we do not want to create some other boundaries around the centroid, this means we accept any sample in any direction if the distance is nearer to the centroid.

One difficulty pertaining to clustering is that the clustering problem is inherently ill-posed, in the sense that there is no single criterion that measures how well a clustering of the data corresponds to the real world. We can measure properties of the clustering such as the average Euclidean distance from a cluster centroid to the members of the cluster. This allows us to tell how well we are able to reconstruct the training data from the cluster assignments. We do not know how well the cluster assignments correspond to properties of the real world. Moreover, there may be many different clusterings that all correspond well to some property of the real world. We may hope to find a clustering that relates to one feature but obtain a different, equally valid clustering that is not relevant to our task. For example, suppose that we run two clustering algorithms

on a dataset consisting of images of red trucks, images of red cars, images of gray trucks, and images of gray cars. If we ask each clustering algorithm to find two clusters, one algorithm may find a cluster of cars and a cluster of trucks, while another may find a cluster of red vehicles and a cluster of gray vehicles. Suppose we also run a third clustering algorithm, which is allowed to determine the number of clusters. This may assign the examples to four clusters, red cars, red trucks, gray cars, and gray trucks. This new clustering now at least captures information about both attributes, but it has lost information about similarity. Red cars are in a different cluster from gray cars, just as they are in a different cluster from gray trucks. The output of the clustering algorithm does not tell us that red cars are more similar to gray cars than they are to gray trucks. They are different from both things, and that is all we know.

7 Graphical Models

A graphical model or probabilistic graphical model (PGM) or structured probabilistic model is a probabilistic model for which a graph expresses the conditional dependence structure between random variables.

Generally, probabilistic graphical models use a graph-based representation as the foundation for encoding a distribution over a multi-dimensional space and a graph that is a compact or factorized representation of a set of independences that hold in the specific distribution. Two branches of graphical representations of distributions are commonly used, namely, Bayesian networks and Markov random fields. Both families encompass the properties of factorization and independences, but they differ in the set of independences they can encode and the factorization of the distribution that they induce.

- **Undirected Graphical Model**:The undirected graph shown in Fig. 9 may have one of several interpretations; the common feature is that the presence of an edge implies some sort of dependence between the corresponding random variables. From this graph we might deduce that B, C, D are all mutually independent, once A is known, or (equivalently in this case) that

$$P[A, B, C, D] = f_{AB}[A, B] \cdot f_{AC}[A, C] \cdot f_{AD}[A, D] \tag{115}$$

  for some non-negative functions f_{AB}, f_{AC}, f_{AD}.

- **Bayesian Network**: If the network structure of the model is a directed acyclic graph, the model represents a factorization of the joint probability of all random variables. More precisely, if the events are $X_1, \ldots, X_n$ then the joint probability satisfies

$$P[X_1, \ldots, X_n] = \prod_{i=1}^{n} P[X_i | \text{pa}(X_i)] \tag{116}$$

  where $\text{pa}(X_i)$ is the set of parents of node X_i, nodes with edges directed towards X_i. In other words, the joint distribution factors into a product of

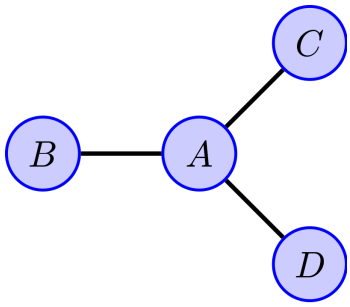


Figure 9: Undirected graphical model with random variables as nodes.

conditional distributions. For example, in the directed acyclic graph shown in the Figure 10 this factorization would be

$$P[A, B, C, D] = P[A] \cdot P[B|A] \cdot P[C|A] \cdot P[D|A, C]$$

Any two nodes are conditionally independent given the values of their parents. In general, any two sets of nodes are conditionally independent given a third set if a criterion called d-separation holds in the graph. Local independences and global independences are equivalent in Bayesian networks.

This type of graphical model is known as a directed graphical model, Bayesian network, or belief network. Classic machine learning models like hidden Markov models, neural networks and newer models such as variable-order Markov models can be considered special cases of Bayesian networks.

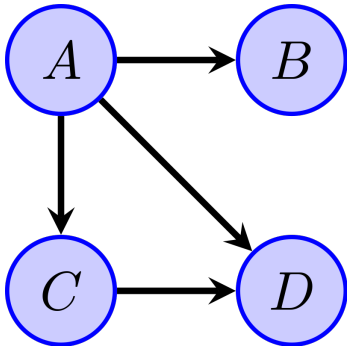


Figure 10: Directed acyclic graphical model: Bayesian Network.

- **Directed Cyclic Graphical Models**: The Figure 11 depicts a graphical model with a cycle. This may be interpreted in terms of each variable 'depending' on the values of its parents in some manner. The particular graph shown suggests a joint probability density that factors as

$$P[A, B, C, D] = P[A] \cdot P[B] \cdot P[C, D|A, B]$$

but other interpretations are possible.

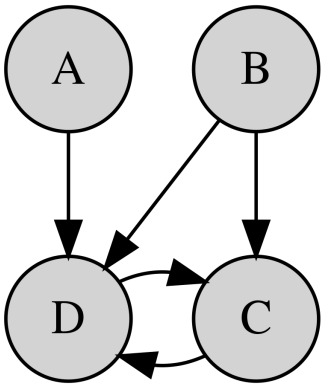


Figure 11: Directed cyclic graphical model.

7.1 Hidden Markov Models

A hidden Markov model (HMM) is a statistical Markov model in which the system being modeled is assumed to be a Markov process – call it X–with unobservable ("hidden") states. As part of the definition, HMM requires that there be an observable process Y whose outcomes are influenced by the outcomes of X in a known way. Since X can not be observed directly, the goal is to learn about X by observing Y. HMM has an additional requirement that the outcome of Y at time $t = t_0$ must be "influenced" exclusively by the outcome of X at $t = t_0$ and that the outcomes of X and Y at $t < t_0$ must not affect the outcome of Y at $t = t_0$.

Before looking at the formal definition of the HMM, let us first understand the pre-requisite concepts.

Markov Property A stochastic process has the Markov property if the conditional probability distribution of future states of the process, conditional on both past and present values, depends only upon the present state. In other words, given the present, the future does not depend on the past. A process with this property is said to be Markov or Markovian and known as a Markov process. The most famous Markov process is a Markov chain. Brownian motion is another well-known Markov process.

Formally it can written as:

$$P(X_n = x_n \mid X_{n-1} = x_{n-1}, \ldots, X_0 = x_0) = P(X_n = x_n \mid X_{n-1} = x_{n-1}). \tag{117}$$

Markov Process or Markov Chain A Markov process is a stochastic process that satisfies the Markov property–sometimes characterized as "memorylessness". In simpler terms, it is a process for which predictions can be made regarding future outcomes based solely on its present state and these predictions are just as good as the ones that could be made knowing the process's full history.

A Markov chain is a type of Markov process that has either a discrete state space or a discrete index set (often representing time), but the precise definition of a Markov chain varies. For example, it is common to define a Markov chain as a Markov process in either discrete or continuous time with a countable state space, thus regardless of the nature of time, but it is also common to define a Markov chain as having discrete time in either countable or continuous state space, thus regardless of the state space.

Definition : Let X_n and Y_n be discrete-time stochastic processes and $n \geq 1$. The pair (X_n, Y_n) is a hidden Markov model if

- X_n is a Markov process whose behavior is not directly observable: hidden;
- $\mathbf{P}\left(Y_n \in A \mid X_1 = x_1, \ldots, X_n = x_n\right) = \mathbf{P}\left(Y_n \in A \mid X_n = x_n\right)$, for every $n \geq 1$, $x_1, \ldots, x_n$, and every Borel set A.

We now describe how the form for HMMs can be derived: in particular, we describe the independence assumptions that are made in the model. Consider a pair of sequences of random variables $x_1, \cdots, x_n$, and $y_1, \cdots, y_n$, where n is the length of the sequences. Suppose that we are solving a part-of-speech tagging problem, where x is sequence of words and y is a corresponding POS tag for each word in the sentence. Specifically, we assume that each x_i can take any value in a finite set of words. For example, it might be a set of possible words in English, for example {the, dog, saw, cat, laughs, $\cdots$}. Each y_i can take any value in a finite set of possible tags. For example, it might be the set of possible part-of-speech tags for English, e.g. {D, N, V, $\cdots$}.

Now let us see how the joint probability of the input and output sequence is modelled with HMM generative model.

$$P(x_1, \cdots, x_n, y_1, \cdots, y_n) = P(y_1, \cdots, y_n) P(x_1, \cdots, x_n, |y_1, \cdots, y_n) \tag{118}$$

$$= \prod_{i=1}^{n} P(y_i|y_1, \cdots, y_{i-1}) P(x_i|x_1, \cdots, x_{i-1}, y_1, \cdots, y_{i-1}, y_i) \tag{119}$$

$$= \prod_{i=1}^{n} P(y_i|y_{i-1}) P(x_i|y_i) \tag{120}$$

In the Eq. 118, we use the exact formula to get the joint probability. This is nothing but a noisy channel modelling, where we are expanding the joint

probability into symbol y generation and then finding probability of getting x if the generated symbol is y. The next Eq. 119 is just an expansion of the probability formula. It is just a conditional probability of getting output y_i given all the previous outputs and seeing token x_i given both previous output labels as well as input tokens. The last equation 120 is where markov assumption is used, the assumption is that the output label y_i depends on the previous output label y_{i-1} and given this state it is independent of the previous labels. Similarly, we made an assumption that given a label y_i, the token x_i is independent of the previous words as well as output labels. The dependance of these variables can be graphically shown as Fig. 12.

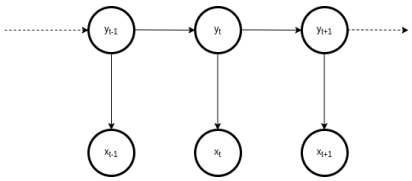


Figure 12: Directed cyclic graphical model.

In the last Equation 120, the output label y_i assumed to be depending only on the previous label, this is a *bigram* Markov assumption. We can improve on this with more relaxed assumption as trigram assumption or further more dependance for example in triagram assumption we say that y_i is going to depend on the previous two labels.

7.2 Markov Random Fields

7.3 Log-Linear Models

The major drawback of HMM is that the markov assumptions which we made are quite restrictive. First we assume that the output is going to depend only on the previous output as well as the input symbol generation is going to depend only on the current output label. Though with these assumption, fairly decent results have been obtained, there is a room for improvement. Consider the POS tagging example, we assumed that the current POS tag is going to depend only on the previous POS tag and the current word is going to depend only on the current POS. These are quite simplistic assumptions and do not simulate the actual POS process where the current POS can depend on the previously occured any of the label as well as current word for surely depends on some of the previous tokens. To mitigate these restrictive assumptions log-linear models are used which are unlike HMM are discriminative i.e instead of modeling joint probability $p(x, y)$, model conditional probability $p(y|x)$.

Formally, we have sets $\mathcal{X}$ and $\mathcal{Y}$: we will assume that $\mathcal{Y}$ is a finite set. Our goal is to build a model that estimates the conditional probability $p(y|x)$ of a label $y \in \mathcal{Y}$ given an input $x \in \mathcal{X}$. For example, x might be a word, and y might be a candidate part-of-speech (noun, verb, preposition etc.) for that word. We

have a feature-vector definition $\phi : \mathcal{X} \times \mathcal{Y} \rightarrow R^d$. We also assume a parameter vector $w \in R^d$. Given these definitions, log-linear models take the following form:

$$p(y|x; w) = \frac{\exp(w.\phi(x, y))}{\sum_{y' \in \mathcal{Y}} \exp(w.\phi(x, y'))} \tag{121}$$

This is a conditional probability of y given x and parametrized over w. The equation can be intuitively understood in the form of scoring function: $\exp(w.\phi(x, y))$. The parameters w are found such that we get higher score from this function for true x, y pair. The denominator is nothing but a normalizing factor so as to get a legitimate probability value.

Log-likelihood function : To estimate parameters w we assume that we are given a training set $\{(x_i, y_i)\}_{i=1}^n$ and from that we find a likelihood function as:

$$L(w) = \sum_{i=1}^{n} \log p(y|x; w) \tag{122}$$

We can think of $L(w)$ as being a function that for a given w measures how well w explains the labeled examples. A "good" value for w will give a high value for $p(y_i|x_i; w) \qquad \forall i = 1, \cdots, n$, and thus will have a high value for $L(w)$.

The maximum-likelihood estimates are

$$w^* = \arg\max_{w \in R^d} \sum_{i=1}^{n} \log p(y|x; w) \tag{123}$$

The maximum-likelihood estimates are thus the parameters that best fit the training set, under the criterion $L(w)$. Unfortunately, an analytical solution does not in general exist. Instead, people generally use gradient-based methods to optimize $L(w)$. In many applications, it has been shown to be highly beneficial to modify the log-likelihood function to include an additional regularization term. The modified criterion is then

$$w^* = \arg\max_{w \in R^d} \sum_{i=1}^{n} \log p(y|x; w) - \frac{\lambda}{2}||w||^2 \tag{124}$$

Note that we are here maximizing the log-likelihood directly, i.e. we do not have a loss function to minimize. It is shown earlier that when we get the MSE loss or Cross-entropy loss underneath that we are actually looking to maximize the log-likelihood, but we assume that the probability $p(y|x)$ follows some distribution i.e. in regression models we assume that the distribution is normal and hence we derive MSE whereas in the case of classification we assume that the probability is Bernoulli and derive cross entropy. However, here we are not making any assumptions on the probability and are just maximizing the loglikelihood, that is the reason that the regularization

term is in negative whereas as generally the sign in positive in loss functions.

7.3.1 Maximum Entropy Markov Models

We will now return to sequence labelling tasks, and describe maximum-entropy Markov models (MEMMs), which make direct use of log-linear models. MEMMs will be a useful alternative to HMMs as they can get rid of the strict assumptions made by HMMs.

Our goal will be to model the conditional distribution

$$p(y_1, y_2, \cdots, y_m | x_1, \cdots, x_m)$$

where each x_j for $j = 1 \cdots m$ is the j^{th} input symbol (for example the j^{th} word in a sentence), and each y_j for $j = 1 \cdots m$ is the j^{th} state. We will use $\mathcal{Y}$ to denote the set of possible states; we assume that $\mathcal{Y}$ is a finite set.

In a first step, MEMMs use the following decomposition:

$$p(y_1, y_2, \cdots, y_m | x_1, \cdots, x_m) = \prod_{i=1}^{m} p(y_i | y_1, y_2, \cdots, y_{i-1}, x_1, \cdots, x_m) \quad (125)$$

$$= \prod_{i=1}^{m} p(y_i | y_{i-1}, x_1, \cdots, x_m) \quad (126)$$

First Eq. 125 is exact formula for the conditional probability, followed by Markov assumption similar to HMM in the second Eq. 126. The assumption is that the current output is going to depend on the previous output and not on any other historical output. But the major difference in HMM and MEMM formulation is that the presence of input terms $x_1, \cdots, x_m$ in this conditional probability which means there is a dependence on the input sequence which was not there in HMM.

Having made these independence assumptions, we then model each term using a log-linear model:

$$\prod_{i=1}^{m} p(y_i | y_{i-1}, x_1, \cdots, x_m) = \frac{\exp(w.\phi(x_1, \cdots, x_m, i, y_{i-1}, y_i))}{\sum_{y' \in \mathcal{Y}} \exp(w.\phi(x_1, \cdots, x_m, i, y_{i-1}, y'))} \quad (127)$$

Decoding with MEMMs The decoding problem is as follows. We are given a new test sequence $x_1, \cdots, x_m$. Our goal is to compute the most likely state sequence for this test sequence,

$$\arg\max_{y_1, y_2, \cdots, y_m \in \mathcal{Y}^m} p(y_1, y_2, \cdots, y_m | x_1, \cdots, x_m)$$

There are $|\mathcal{Y}|^m$ possible state sequences, so for any reasonably large sentence length m brute-force search through all the possibilities will not be possible. Intelligent decoding algorithms such as Viterbi algorithms are used for decoding.

The key advantage of MEMMs is that the use of feature vectors ϕ allows much richer representations than those used in HMMs. For example, the transition probability can be sensitive to any word in the input sequence $x_1, \cdots, x_m$. In addition, it is very easy to introduce features that are sensitive to spelling features (e.g., prefixes or suffixes) of the current word x_i, or of the surrounding words. These features are useful in many NLP applications, and are difficult to incorporate within HMMs in a clean way.

In NLP, decoding or inference are important steps as getting the output is not straightforward as in the cases of other domains. This special need for sophisticated inference techniques arises because of the nature of problems solved in NLP, the problems are structured prediction where the output also depend on each other. Therefore, different efficient decoding algorithms have been proposed.

7.3.2 Conditional Random Fields

Conditional random fields (CRFs) are a class of statistical modeling methods often applied in pattern recognition and machine learning and used for structured prediction. Whereas a classifier predicts a label for a single sample without considering "neighbouring" samples, a CRF can take context into account. To do so, the predictions are modelled as a graphical model, which represents the presence of dependencies between the predictions. What kind of graph is used depends on the application. For example, in natural language processing, "linear chain" CRFs are popular, for which each prediction is dependent only on its immediate neighbours. In image processing, the graph typically connects locations to nearby and/or similar locations to enforce that they receive similar predictions.

Other examples where CRFs are used are: labeling or parsing of sequential data for natural language processing or biological sequences, part-of-speech tagging, shallow parsing, named entity recognition, gene finding, peptide critical functional region finding, object recognition and image segmentation in computer vision.

It is also another log-linear model and more poweful than MEMM. A first key idea in CRFs will be to define a feature vector

$$\phi(x_1, \cdots, x_m, y_1, y_2, \cdots, y_m)$$

which considers whole input sequence and whole output sequence altogether. We will often refer to ϕ as being a "global" feature vector (it is global in the sense that it takes the entire state sequence into account). We then build a giant log-linear model,

$$p(y_1, y_2, \cdots, y_m | x_1, \cdots, x_m) = \frac{\exp(w.\phi(x_1, \cdots, x_m, y_1, y_2, \cdots, y_m))}{\sum_{(y'_1, y'_2, \cdots, y'_m) \in \mathcal{Y}^m} \exp(w.\phi(x_1, \cdots, x_m, y'_1, y'_2, \cdots, y'_m))} \tag{128}$$

This is another log-linear model, but it is is "giant" in the sense that: 1) the space of possible values for y, i.e., $|\mathcal{Y}|^m$, is huge. 2) The normalization constant (denominator in the above expression) involves a sum over the set $|\mathcal{Y}|^m$. At first glance, these issues might seem to cause severe computational problems, but we will soon see that under appropriate assumptions we can train and decode efficiently with this type of model.

8 Ensemble Learning

These are not the separate learning algorithms but the set of meta-learning techniques which are used to improve the accuracy of machine learning algorithms. The basic idea is to train multiple machine learning models and then combine the results with some intuition to produce the higher results. The difference in the combination of the results produce three different algorithms.

8.1 Bagging

This is an abbreviation for Bootstrap aggregating. Multiple learners are trained on the boostrapped samples from the training data, and then finally averaging or voting is done to get the final prediction. To get the training data for each learner instead of repeating the same dataset, predefined number of samples are randomly selected with replacement. This is done so as to make the learning as uncorrelated as possible as if we use same training data, the models will same things, therefore it is beneficial to use different samples randomly.

Random forest is slight variation from applying bagging over decision trees. Suppose we use Bagging over decision trees, then the idea will be train multiple decision trees over bootstrapped data and then to aggregate the results to get final prediction. However, in random forest, one more level of randomness is introduced where at the splitting of nodes only fewer features than total number of features are used randomly. This means that the trees are generated differently with least correlation between them. This also means that the trained trees are weak learners as they are trained with fewer features. But, counter intuitively it produces stronger results.

8.2 Boosting

This is an another approach of using multiple learners to get the stronger prediction, however, unlike the previous way of aggregating them with simple averaging or voting the learners are learned in sequential manner where the classification of the pevious learner is taken into consideration while training the next learner. The next learner assigns higher weightage to the samples which are wrongly classified so the next learners puts stronger parameters updates and learn to classify more difficult points. Thus, weak learners are combined to get the stronger learning.

At the inference step the averaging of the predictions done to get the final prediction. One approach of doing this aggregation can be to weigh the predictions from each learner according to their overall accuracy of the predictions on the validation set.

There are different meta-learning algorithms have been based on the boosting concept. Let us look at them in following paragraphs.

8.2.1 Gradient Boosting

Like other boosting methods, gradient boosting combines weak learners into a single strong learner in an iterative fashion. It is easiest to explain in the least-squares regression setting, where the goal is to teach a model F to predict values of the form $\hat{y} = F(x)$ by minimizing the mean squared error $\frac{1}{n}\sum_i (\hat{y}_i - y_i)^2$, where i indexes over some training set of size n of actual values of the output variable y:

Now, let us consider a gradient boosting algorithm with M stages. At each stage m ($1 \leq m \leq M$) of gradient boosting, suppose some imperfect model F_m (for low m, this model may simply return $\hat{y}_i = \bar{y}$, where the RHS is the mean of y). In order to improve F_m, our algorithm should add some new estimator, $h_m(x)$. Thus,

$$F_{m+1}(x_i) = F_m(x_i) + h_m(x_i) = y_i$$

or, equivalently,

$$h_m(x_i) = y_i - F_m(x_i)$$

Therefore, gradient boosting will fit h_m to the residual $y_i - F_m(x_i)$. As in other boosting variants, each F_{m+1} attempts to correct the errors of its predecessor F_m. A generalization of this idea to loss functions other than squared error, and to classification and ranking problems, follows from the observation that residuals $h_m(x_i)$ for a given model are proportional to the negative gradients of the mean squared error (MSE) loss function with respect to $F(x_i)$:

$$L_{\text{MSE}} = \frac{1}{n}\sum_{i=1}^{n} (y_i - F(x_i))^2$$
$$\frac{\partial L_{\text{MSE}}}{\partial F(x_i)} = \frac{2}{n}(y_i - F(x_i)) = \frac{2}{n}h_m(x_i)$$

So, gradient boosting could be specialized to a gradient descent algorithm, and generalizing it entails "plugging in" a different loss and its gradient.

Algorithm 2 Gradient Boosting Algorithm

1: **Require:** training set $\{(x_i, y_i)\}_{i=1}^n$, a differentiable loss function $L(y, F(x))$, number of iterations M.

2: **Initialize model with a constant value**: $F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma)$.

3: **while** $m \leq M$ **do**

4: Compute so-called pseudo-residuals:

$$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n. \tag{129}$$

5: Fit a weak learner $h_m(x)$ to pseudo-residuals with training set:

$$\{(x_i, r_{im})\}_{i=1}^n$$

6: Compute multiplier γ_m by solving optimization problem:

$$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$

7: Update the model: $F_m(x) = F_{m-1}(x) + \gamma_m h_m(x)$.

8: Output $F_M(x)$..

8.2.2 XGBoost

8.2.3 AdaBoost

8.3 Stacking

In stacking instead of doing the simple averaging or voting another learner is trained to combine the results from the previous stage. The models are trained at level-0 are combined at level-1 by employing some other model to produce stacking kind of structure.

9 Anomaly Detection

Anomaly detection, a.k.a. outlier detection or novelty detection, is referred to as the process of detecting data instances that significantly deviate from the majority of data instances. Anomaly detection has been an active research area for several decades, with early exploration dating back as far as to 1960s. Due to the increasing demand and applications in broad domains, such as risk management, compliance, security, financial surveillance, health and medical risk, and AI safety, anomaly detection plays increasingly important roles, highlighted in various communities including data mining, machine learning, computer vision and statistics.

Algorithm 3 XGBoost Algorithm

1: **Require:** training set $\{(x_i, y_i)\}_{i=1}^{n}$, a differentiable loss function $L(y, F(x))$, number of iterations M.

2: **Initialize model with a constant value**: $F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma)$.

3: **while** $m \leq M$ **do**

4: Compute the gradients and hessians:

$$\phi_m(x_i) = -\left[\frac{\partial L(y_i, F_{m-1}(x_i))}{\partial F_{m-1}(x_i)}\right]$$
$$\psi_m(x_i) = -\left[\frac{\partial^2 L(y_i, F_{m-1}(x_i))}{\partial F_{m-1}(x_i)^2}\right]$$

5: Fit a weak learner $h_m(x)$ to pseudo-residuals with training set:

$$\left\{x_i, -\frac{\phi_m(x_i)}{\psi_m(x_i)}\right\}_{i=1}^{N}$$

by solving following optimization problem

$$\theta_m = \arg\min_{\theta \in \mathbf{\Theta}} \sum_{i=1}^{N} \frac{1}{2} \psi_m(x_i) \left[-\frac{\phi_m(x_i)}{\psi_m(x_i)} - \theta(x_i)\right]^2$$
$$\hat{F}_m(x) = \alpha\, \theta_m(x)$$

6: Update the model: $F_m(x) = F_{m-1}(x) + \hat{F}_m(x)$

7: Output $F(x) = \sum_{m=0}^{M} F_m(x)$.

Major Problems Unlike those problems and tasks on majority, regular or evident patterns, anomaly detection addresses minority, unpredictable/uncertain and rare events, leading to some unique problem complexities to all (both deep and shallow) detection methods:

- **Unknownness** Anomalies are associated with many unknowns, e.g., instances with unknown abrupt behaviors, data structures, and distributions. They remain unknown until actually occur, such as novel terrorist attacks, frauds and network intrusions.

- **Heterogeneous anomaly classes** Anomalies are irregular, and thus, one class of anomalies may demonstrate completely different abnormal characteristics from another class anomalies. For example, in video surveillance, the abnormal events robbery, traffic accidents and burglary are visually highly different.

- **Rarity and class imbalance** Anomalies are typically rare data instances, contrasting to normal instances that often account for an overwhelming proportion of the data. Therefore, it is difficult, if not impossible, to collect a large amount of labeled abnormal instances. This results in the unavailability of large-scale labeled data in most applications. The class imbalance is also due to the fact that misclassification of anomalies is normally much more costly than that of normal instances.

- **Diverse types of anomaly** Three completely different types of anomaly have been explored.

  → *Point anomalies* are individual instances that are anomalous w.r.t. the majority of other individual instances. Consider, for instance, a $10,000 transaction from your bank account when you normally only make $10-$100 transactions. This may also be known as an additive outlier.

  → *Conditional anomalies*, a.k.a. contextual anomalies, also refer to individual anomalous instances but in a specific context, i.e., data instances are anomalous in the specific context, otherwise normal. The contexts can be highly different in real-world applications, e.g., sudden temperature drop/increase in a particular temporal context, or rapid credit card transactions in unusual spatial contexts. Other example can be, ice cream sales remaining constant throughout hot and cold months.

  → *Group anomalies*, a.k.a. collective anomalies, are a subset of data instances anomalous as a whole w.r.t. the other data instances; the individual members of the collective anomaly may not be anomalies, e.g., exceptionally dense subgraphs formed by fake accounts in social network are anomalies as a collection, but the individual nodes in those subgraphs can be as normal as real accounts.

  → *Temporal anomalies* A temporal anomaly is when the number of users nears zero or very low for a short period of time. This could be, as an example, the result of a server going down.

Unsolved Challenges

- **Low recall** One of the most challenging issues is the difficulty to achieve high anomaly detection recall rate. Since anomalies are highly rare and heterogeneous, it is difficult to identify all of the anomalies. Many normal instances are wrongly reported as anomalies while true yet sophisticated anomalies are missed.

- **High-dimensional Anomaly Detection** Anomalies often exhibit evident abnormal characteristics in a low-dimensional space yet become hidden and unnoticeable in a high-dimensional space. High-dimensional anomaly detection has been a long-standing problem. Subspace/feature selection-based methods may be a straightforward solution. However,

identifying intricate (e.g., high-order, nonlinear and heterogeneous) feature interactions and couplings may be essential in high-dimensional data yet remains a major challenge for anomaly detection.

- **Data efficient models** Due to the difficulty and cost of collecting large-scale labeled anomaly data, fully supervised anomaly detection is often impractical as it assumes the availability of labeled training data with both normal and anomaly classes. The unsupervised methods do not have any prior knowledge of true anomalies. They rely heavily on their assumption on the distribution of anomalies. On the other hand, it is often not difficult to collect labeled normal data and some labeled anomaly data.

  Semi-supervised anomaly detection, which assumes a set of labeled normal training data, is a research direction dedicated to this problem. Another research line is weakly-supervised anomaly detection that assumes we have some labels for anomaly classes yet the class labels are partial/incomplete (i.e., they do not span the entire set of anomaly class), inexact (i.e., coarse-grained labels), or inaccurate (i.e., some given labels can be incorrect).

  Two major challenges are how to learn expressive normality/abnormality representations with a small amount of labeled anomaly data and how to learn detection models that are generalized to novel anomalies uncovered by the given labeled anomaly data.

- **Noise-resilient Anomaly Detection** Many weakly/semi-supervised anomaly detection methods assume the given labeled training data is clean, which can be highly vulnerable to noisy instances that are mistakenly labeled as an opposite class label. One main challenge here is how to develop noise-resilient anomaly detection.

- **Different Anomaly Types** Most of existing methods are for point anomalies, which cannot be used for conditional anomaly and group anomaly since they exhibit completely different behaviors from point anomalies. One main challenge here is to incorporate the concept of conditional/-group anomalies into anomaly measures/models for the detection of those complex anomalies.

- **Explanable Model** In many critical domains there may be some major risks if anomaly detection models are directly used as black-box models. For example, the rare data instances reported as anomalies may lead to possible algorithmic bias against the minority groups presented in the data, such as under-represented groups in fraud detection and crime detection systems. An effective approach to mitigate this type of risk is to have anomaly explanation algorithms that provide straightforward clues about why a specific data instance is identified as anomaly. Providing such explanation can be as important as detection accuracy in some applications. To derive anomaly explanation from specific detection methods is still a largely unsolved problem, especially for complex models. Developing inherently interpretable anomaly detection models is also crucial, but

it remains a main challenge to well balance the model's interpretability and effectiveness.

Now let us see the different approaches proposed to detect anomalies. First we will look at the traditional machine learning approaches and then we describe deep learning based approaches.

9.1 Traditional ML approaches

9.1.1 KNN

KNN for Anomaly Detection is a repurposing of the original K-Nearest Neighbors algorithm, which determines a neighbors class by aggregating the classes of the K-Nearest Neighbors in terms of Euclidean distance. Because KNN attempts to make minor generalizations in recognition of underlying relationships in the data (increasing as k grows), errors in predictions are signs that data points may not be conforming to overall trends. Using this approach, any algorithm that uses a predictive model can be used to detect anomalies. Any data point that is an anomaly will, in theory, reduce the predictive power of the model. Using these labels, another KNN model can be trained to strictly identify anomalies / not anomalies. This method, which can be repurposed with other predictor methods, is, in a sense, a semi-supervised approach.

9.1.2 Local Outlier Factor

It is a density-based algorithm for anomaly detection. Using Euclidean distances between the k-nearest neighbors, the local density can be estimated. Local Outlier Factor compares the local density of one item to the local density of its neighbors, thus determining areas of similar density and other items of much lower density than their neighbors (outliers). LOF relies on the idea that the density around an outlier is much different than the density around its neighbors. Local Outlier Factor is computed on the base of the average ratio of the local reachability density of an item and its k-nearest neighbors. While it is a nearest-neighbor based approach, it is unsupervised and does not need any labels.

9.1.3 Isolation Forest

It attempts to isolate observations by randomly choosing a feature and randomly splitting the feature at some split value between the maximum and minimum value. The algorithm relies on the idea that isolating anomalies are easier because few conditions are needed to separate the cases from normal observations. On the other hand, isolating a normal observation requires more conditions. Hence, an anomaly score can be calculated as the number of conditions required to separate an observation.

The algorithm first constructs the separation by creating several isolation trees (random decision trees), calculating the path length to isolate each observation. Because the method relies on randomness, bagging is implemented

by aggregating the results of many isolation trees. Isolation forest is a clever method that relies on a heavy assumption, which may have various degrees of truth, depending on the context.

9.1.4 Classification Model

This method requires a labeled dataset containing both normal and anomalous samples to construct a predictive model to classify future data points. The core idea is that we are going to classify whether a given point is anamolous or not. For that we take training data containing normal samples and anomolous data and learn a classifier to predict it. The most commonly used algorithms for this purpose are supervised Neural Networks, Support Vector Machine learning, K-Nearest Neighbors Classifier, etc.

The main problem with these approaches is that of the class imbalance. There will be plenty of data points of normal instances whereas the number of anamolous instances will be quite less. This makes the training part quite difficult. We have to rely on either sampling approaches or to other techniques to reduce class imbalance issue like weighted loss function where we weight the errors from the low numbered class higher than the higher numbered class.

9.2 Deep Learning for Feature Extraction

This category of methods aims at leveraging deep learning to extract low-dimensional feature representations from high-dimensional and/or non-linearly separable data for downstream anomaly detection. The feature extraction and the anomaly scoring are fully disjointed and independent from each other. Thus, the deep learning components work purely as dimensionality reduction only.

Formally, the approach can be represented as

$$\mathcal{Z} = \Phi(\mathbf{x}; \Theta) \tag{130}$$

where $\Phi : \mathcal{X} \rightarrow \mathcal{Z}$ is a deep neural network-based feature mapping function, with $\mathbf{x} \in R^D, Z \in R^K$ and normally $\ll D$. An anomaly scoring method f that has no connection to the feature mapping Φ is then applied onto the new space to calculate anomaly scores.

Compared to the dimension reduction methods that are popular in anomaly detection, such as principal component analysis (PCA) and random projections, deep learning techniques have been demonstrating substantially better capability in extracting semantically rich features and non-linear feature relations. Here the assumption is that the feature representations extracted by deep learning models preserve the discriminative information that helps separate anomalies from normal instances.

One research line is to directly uses popular pre-trained deep learning models, such as AlexNet, VGG and ResNet, to extract low-dimensional features. This line is explored in anomaly detection in complex high-dimensional data such as image data and video data.

Advantages The advantages of this approach are as follows. (i) A large number of state-of-the-art (pre-trained) deep models and off-the-shelf anomaly detectors are readily available. (ii) Deep feature extraction offers more powerful dimensionality reduction than popular linear methods. (iii) It is easy-to-implement given the public availability of the deep models and detection methods.

Disadvantages Their disadvantages are as follows. (i) The fully disjointed feature extraction and anomaly scoring often lead to suboptimal anomaly scores. (ii) Pre-trained deep models are typically limited to specific types of data.

9.3 Generic Normality Feature Learning

This category of methods learns the representations of data instances by optimizing a generic feature learning objective function that is not primarily designed for anomaly detection, but the learned representations can still empower the anomaly detection since they are forced to capture some key underlying data regularities.

9.3.1 Autoencoders

These models aims to learn some low-dimensional feature representation space on which the given data instances can be well reconstructed. This is a widely-used technique for data compression or dimension reduction. The heuristic for using this technique in anomaly detection is that the learned feature representations are enforced to learn important regularities of the data to minimize reconstruction errors; anomalies are difficult to be reconstructed from the resulting representations and thus have large reconstruction errors.

The core assumption while using AE for anomaly detection is that, the normal instances can be better restructured from compressed space than anomalies.

An AE is composed of an encoding network and an decoding network. The encoder maps the original data onto low-dimensional feature space, while the decoder attempts to recover the data from the projected low-dimensional space. The parameters of these two networks are learned with a reconstruction loss function. A bottleneck network architecture is often used to obtain low-dimensional representations than the original data, which forces the model to retain the information that is important in reconstructing the data instances.

To minimize the overall reconstruction error, the retained information is required to be as much relevant as possible to the dominant instances, e.g., the normal instances. As a result, the data instances such as anomalies that deviate from the majority of the data are poorly reconstructed. The data reconstruction error can therefore be directly used as anomaly score. The basic formulation of this approach is given as follows.

$$\mathbf{z} = \Phi_e(\mathbf{x}; \Theta_e), \;\; \hat{\mathbf{x}} = \Phi_d(\mathbf{z}; \Theta_d) \tag{131}$$

$$\{\Theta_e^*, \Theta_d^*\} = \arg\max_{\Theta_e, \Theta_d} \frac{1}{n} \sum_{i=1}^{n} ||\mathbf{x} - \Phi_d(\Phi_e(\mathbf{x}; \Theta_e); \Theta_d)||^2 \tag{132}$$

$$s_{\mathbf{x}} = ||\mathbf{x} - \Phi_d(\Phi_e(\mathbf{x}; \Theta_e^*); \Theta_d^*)||^2 \tag{133}$$

where Φ_e is the encoding network with the parameters Θ_e and Φ_d is the decoding network with the parameters Θ_d. The encoder and the decoder can share the same weight parameters to reduce parameters and regularize the learning. $s_{\mathbf{x}}$ is a reconstruction error-based anomaly score of $\mathbf{x}$.

Advantages The advantages of data reconstruction-based methods are as follows. (i) The idea of AEs is straightforward and generic to different types of data. (ii) Different types of powerful AE variants can be leveraged to perform anomaly detection.

Disadvantages Their disadvantages are as follows. (i) The learned feature representations can be biased by infrequent regularities and the presence of outliers or anomalies in the training data. (ii) The objective function of the data reconstruction is designed for dimension reduction or data compression, rather than anomaly detection. As a result, the resulting representations are a generic summarization of underlying regularities, which are not optimized for detecting irregularities.

9.3.2 Generative Adversarial Networks (GAN)

GAN-based anomaly detection emerges quickly as one popular deep anomaly detection approach after its early use. This approach generally aims to learn a latent feature space of a generative network G so that the latent space well captures the normality underlying the given data. Some form of residual between the real instance and the generated instance are then defined as anomaly score.

Assumption Normal data instances can be better generated than anomalies from the latent feature space of the generative network in GANs.

9.3.3 Predictability Modeling

Predictability modeling-based methods learn feature representations by predicting the current data instances using the representations of the previous instances within a temporal window as the context. In this section data instances are referred to as individual elements in a sequence, e.g., video frames in a video sequence. This technique is widely used for sequence representation learning and prediction.

To achieve accurate predictions, the representations are enforced to capture the temporal/sequential and recurrent dependence within a given sequence length. Normal instances are normally adherent to such dependencies well and can be well predicted, whereas anomalies often violate those dependencies and

are unpredictable. Therefore, the prediction errors can be used to define the anomaly scores.

Assumption Normal instances are temporally more predictable than anomalies.

This research line is popular in video anomaly detection. Video sequence involves complex high-dimensional spatial-temporal features. Different constraints over appearance and motion features are needed in the prediction objective function to ensure a faithful prediction of video frames.

Formally, given a video sequence with consecutive t frames $x_1, x_2, \cdots, x_t$, then the learning task is to use all these frames to generate a future frame $\hat{x}_{t+1}$ so that it is as close as possible to the ground truth x_{t+1}. Its general objective function can be formulated as

$$\alpha \ell_{pred}(\hat{x}_{t+1}, x_{t+1}) + \beta \ell_{adv}(\hat{x}_{t+1}) \tag{134}$$

where $\hat{x}_{t+1} = \Psi(\Phi(x_1, x_2, \cdots, x_t; \Theta); W)$, ℓ_{pred} is loss measured between predicted frame and actual frame whereas ell_{adv} is adversarial loss. After training, for a given video frame x, a normalized Peak Signal-to-Noise Ratio based on the prediction difference $||x_i \hat{x}_i||^2$ is used to define the anomaly score. Under the same framework, an additional autoencoder-based reconstruction network can also be added to further refine the predicted frame quality, which helps to enlarge the anomaly score difference between normal and abnormal frames.

9.4 Anomaly Measure-dependent Feature Learning

Anomaly measure-dependent feature learning aims at learning feature representations that are specifically optimized for one particular existing anomaly measure. Different from the generic feature learning approach that calculates anomaly scores based on some heuristics after obtaining the learned representations, this research line incorporates an existing anomaly measure f into the feature learning objective function to optimize the feature.

9.4.1 Distance-based Measure

Deep distance-based anomaly detection aims to learn feature representations that are specifically optimized for a specific type of distance-based anomaly measures. Distance-based methods are straightforward and easy-to-implement. There have been a number of effective distance-based anomaly measures introduced, e.g., DB outliers, k-nearest neighbor distance, average k-nearest neighbor distance, relative distance, and random nearest neighbor distance.

One major limitation of these traditional distance-based anomaly measures is that they fail to work effectively in high-dimensional data due to the curse of dimensionality. Since deep distance-based anomaly detection techniques project data onto low-dimensional space before applying the distance measures, it can well overcome this limitation.

Assumption Anomalies are distributed far from their closest neighbors while normal instances are located in dense neighborhoods.

The random neighbor distance-based anomaly measure is leveraged to drive the learning of low-dimensional representations out of ultrahigh-dimensional data. The key idea is that the representations are optimized so that the nearest neighbor distance of pseudo-labeled anomalies in random subsamples is substantially larger than that of pseudo-labeled normal instances. The pseudo labels are generated by some off-the-shelf anomaly detectors. Let $\mathcal{S} \in \mathcal{X}$ be a subset of data instances randomly sampled from the dataset $\mathcal{X}$, $\mathcal{A}$ and $\mathcal{N}$ respectively be the pseudo-labeled anomaly and normal instance sets, with $\mathcal{X} = \mathcal{A} \cup \mathcal{N}$, its loss function is built upon the hinge loss function:

$$L_{query} = \frac{1}{|\mathcal{X}|} \sum_{x \in A, x' \in N} \max\{0, m + f(x, \mathcal{S}; \Theta) - f(x', \mathcal{S}; \Theta)\} \tag{135}$$

where m is a predefined constant for the margin between two distances, which is a random nearest neighbor distance function operated in the representation space:

$$f(x, \mathcal{S}; \Theta) = \min_{x' \in \mathcal{S}} ||\Phi(x; \Theta)\Phi(x'; \Theta)|| \tag{136}$$

Minimizing the loss in Eq. 135 guarantees that the random nearest neighbor distance of anomalies are at least m greater than that of normal instances in the Φ-projected representation space. At the evaluation stage, the random distance in Eq. 136 is used directly to obtain the anomaly score for each test instance.

Following this approach, we might also derive similar representation learning tailored for other distance-based measures by replacing Eq. 136 with the other measures, such as the k-nearest neighbor distance or the average k-nearest neighbor distance. However, these measures are significantly more computationally costly. Thus, one major challenging for such adaptions would be the prohibitively high computational cost.

This approach is similar to contrastive loss based approach. There we provide triplet to the network: one two are normal samples and one is anomaly. The latent representation is made to be nearer for the normal samples whereas anomaly is tried to push away with the hinge loss.

9.4.2 One-class classification

This category of methods aims to learn feature representations customized to subsequent one-class classification-based anomaly detection. One-class classification is referred to as the problem of learning a description of a set of data instances to detect whether new instances conform to the training data or not. Most one-class classification models are inspired by Support Vector Machines (SVM, such as the two widely-used one-class models: one-class SVM (or v-SVC) and Support Vector Data Description (SVDD).

Assumption All normal instances come from a single (abstract) class and can be summarized by a compact model, to which anomalies do not conform.

In general one-class SVM is combined with neural networks, conventional one-class SVM is to learn a hyperplane that maximize a margin between training data instances and the origin. The key idea of deep one-class SVM is to learn the one-class hyperplane from the neural network-enabled low-dimensional representation space rather than the original input space. A generic formulation of the key ideas in can be represented as

$$\min_{r,\Theta,w} \frac{1}{2}||\Theta||^2 + \frac{1}{vN}\sum_{i=1}^{N} \max\{0, r - w^T\Phi(x;\Theta)\} \tag{137}$$

where r is the margin parameter, Θ are the parameters of a representation network.

Any instances that have $r - w^T\Phi(x;\Theta) > 0$ can be reported as anomalies.

9.5 End-to-End Anomaly Score Learning

This research line aims at learning scalar anomaly scores in an end-to-end fashion. Compared to anomaly measure-dependent feature learning, the anomaly scoring in this type of approach is not dependent on existing anomaly measures; it has a neural network that directly learns the anomaly scores. Novel loss functions are often required to drive the anomaly scoring network.

9.5.1 Ranking Models

10 Recommender Systems

11 Domain Adaptation

A Sigmoid Function

A sigmoid function is a mathematical function having a characteristic "S"-shaped curve or sigmoid curve. In some fields, most notably in the context of artificial neural networks, the term "sigmoid function" is used as an alias for the logistic function.

A wide variety of sigmoid functions including the logistic and hyperbolic tangent functions have been used as the activation function of artificial neurons. Sigmoid curves are also common in statistics as cumulative distribution functions (which go from 0 to 1), such as the integrals of the logistic density, the normal density, and Student's t probability density functions. The logistic sigmoid function is invertible, and its inverse is the logit function. The sigmoid function could generate some non-zero values, resulting in a dense representation.

- **Logistic function**:

$$f(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{1+e^x} \tag{138}$$

Derivative The standard logistic function has an easily calculated derivative. The derivative is known as the density of the logistic distribution:

$$\frac{d}{dx} f(x) = \frac{e^x \cdot (1+e^x) - e^x \cdot e^x}{(1+e^x)^2} = \frac{e^x}{(1+e^x)^2} = f(x)\big(1-f(x)\big) \tag{139}$$

Integral Conversely, its antiderivative can be computed by the substitution $u = 1 + e^x$, since $f(x) = \frac{e^x}{1+e^x} = \frac{u'}{u}$

$$\int \frac{e^x}{1+e^x}\,dx = \int \frac{1}{u}\,du = \ln u = \ln(1+e^x). \tag{140}$$

In artificial neural networks, this is known *softplus* function.

- **Hyperbolic Tangent**:

$$f(x) = \tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{141}$$

- **Arctangent function**:

$$f(x) = \arctan x \tag{142}$$

- **Generalized Logistic function**:

$$f(x) = (1+e^{-x})^{-\alpha} \qquad \alpha > 0 \tag{143}$$

B Derivative of Absolute Function

$$
\begin{aligned}
\frac{d}{dx} f(x) &= \frac{d}{dx}|x| \\
&= \frac{d}{dx}\sqrt{(x^2)} \\
&= \frac{d}{dx}(x^2)^{\frac{1}{2}} \\
&= \frac{1}{2}(x^2)^{-\frac{1}{2}} 2x \\
&= \frac{x}{(x^2)^{\frac{1}{2}}} \\
&= \frac{x}{|x|} \\
\frac{d}{dx}|x| &= \operatorname{sign}(x)
\end{aligned}
$$