Usually, the performance measure P is specific to the task T
0-1 loss is 0 if correctly classified and 1 if not
Most common metric is reporting average log probability the model assigns to examples

If measuring a quantity is hard to define or intractable, an alternative criterion which is a good approximation of the desired criterion must be employed

Supervised learning attempts to estimate p(y | x) whereas unsupervised learning attempts to learn an implicit probability distribution
Chain rule of probability for a vector x in R^n

$$p(\mathbf{x}) = \prod_{i=1}^{n} p(\mathrm{x}_i \mid \mathrm{x}_1, \dots, \mathrm{x}_{i-1}).$$

An unsupervised task, modeling p(x) , can be split into n supervised learning problems by this chain rule
Or, the supervised problem of learning p(y | x) can be solved by using unsupervised technologies to learn the joint distribution p(x, y) and inferring

$$p(y \mid \mathbf{x}) = \frac{p(\mathbf{x}, y)}{\sum_{y'} p(\mathbf{x}, y')}.$$

Multi-instance / semi-supervised learning - a collection of examples has an attribute, but individual members are not labeled

Design matrix - rows are examples and columns are features
Not all photographs can be measured with the same # of features
{x^1, x^2 … x^j } does not imply any two example vectors have the same size

$$\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}$$

Linear regression has a weight vector where w is a vector of parameters
w_i is the feature x_i is multiplied by before summing up the contributions
w is a set of weights determining how each feature affects the prediction

$$\mathrm{MSE}_{\mathrm{test}} = \frac{1}{m} || \hat{\boldsymbol{y}}^{(\mathrm{test})} - \boldsymbol{y}^{(\mathrm{test})} ||_2^2,$$

Error increases when the Euclidean distance between the targets and the prediction increase
Design an algorithm that improves the weights w in a way that reduces $\mathrm{MSE}_{\mathrm{test}}$ when algorithm gains experience by observing (X^{train}, y^{train})
We can minimize $\mathrm{MSE}_{\mathrm{train}}$

Minimize by solving for where the gradient is 0

$$\nabla_{\boldsymbol{w}} \mathrm{MSE}_{\mathrm{train}} = 0$$

$$\Rightarrow \nabla_{\boldsymbol{w}} \frac{1}{m} ||\hat{\boldsymbol{y}}^{(\mathrm{train})} - \boldsymbol{y}^{(\mathrm{train})}||_2^2 = 0$$

$$\Rightarrow \frac{1}{m} \nabla_{\boldsymbol{w}} ||\boldsymbol{X}^{(\mathrm{train})}\boldsymbol{w} - \boldsymbol{y}^{(\mathrm{train})}||_2^2 = 0$$

Solve for where the

$$\Rightarrow \nabla_{\boldsymbol{w}} \left(\boldsymbol{X}^{(\mathrm{train})}\boldsymbol{w} - \boldsymbol{y}^{(\mathrm{train})}\right)^{\top} \left(\boldsymbol{X}^{(\mathrm{train})}\boldsymbol{w} - \boldsymbol{y}^{(\mathrm{train})}\right) = 0 \qquad (5.9)$$

$$\Rightarrow \nabla_{\boldsymbol{w}} \left(\boldsymbol{w}^{\top}\boldsymbol{X}^{(\mathrm{train})\top}\boldsymbol{X}^{(\mathrm{train})}\boldsymbol{w} - 2\boldsymbol{w}^{\top}\boldsymbol{X}^{(\mathrm{train})\top}\boldsymbol{y}^{(\mathrm{train})} + \boldsymbol{y}^{(\mathrm{train})\top}\boldsymbol{y}^{(\mathrm{train})}\right) = 0 \qquad (5.10)$$

$$\Rightarrow 2\boldsymbol{X}^{(\mathrm{train})\top}\boldsymbol{X}^{(\mathrm{train})}\boldsymbol{w} - 2\boldsymbol{X}^{(\mathrm{train})\top}\boldsymbol{y}^{(\mathrm{train})} = 0 \qquad (5.11)$$

$$\Rightarrow \boldsymbol{w} = \left(\boldsymbol{X}^{(\mathrm{train})\top}\boldsymbol{X}^{(\mathrm{train})}\right)^{-1}\boldsymbol{X}^{(\mathrm{train})\top}\boldsymbol{y}^{(\mathrm{train})} \qquad (5.12)$$

where we are taking the gradient with respect to the weight vector (which is the same as w^T for our purposes)

The system of equations whose solution is given by 5.12 is known as the normal equations
Lin reg is frequently used with another parameter, an intercept b

$$\hat{y} = \boldsymbol{w}^{\top}\boldsymbol{x} + b$$

The mapping from features to predictions is an affine function, meaning that the plot of a model's predictions don't need to pass through the origin
Instead of adding the bias parameter b, add an extra element to x which is always set to 1 ; the extra weight corresponding to this element is the bias parameter

Capacity, Overfitting, and Underfitting
Optimization problems only minimize error on the training set
Generalization error / test error is defined as the expected value of the error of a new input which may be encountered in practice
In linreg, we care about test error which can be measured by

$$\frac{1}{m^{(\mathrm{test})}} ||\boldsymbol{X}^{(\mathrm{test})}\boldsymbol{w} - \boldsymbol{y}^{(\mathrm{test})}||_2^2.$$

Statistical learning theory deals with the issue of affecting test set performance when only observing the training set

The train and test data are generated over by a probability distribution over datasets called the data generating process
i.i.d assumptions - examples in each dataset are independent, and that the train and test set are identically distributed / drawn from the same probability distribution as each other

Data generating process over a single example, which is then used to generate every train example and test example
This underlying distribution is the data generating distribution, p_{data}

Expected training error equals expected test error
If the probability distribution p(x, y) is sampled to generate a train and test set, for some fixed weight vector, we should have the same error since the same dataset is sampled the same way

Capacity = ability of a model to fit to a wide variety of functions
Models with high capacity can overfit by memorizing properties of the training set which do not hold for the test set

We can limit a model by choosing its hypothesis space - the set of functions the learning algorithm is allowed to select as being the solution
We can generalize linear regression to include polynomials in its hypothesis space

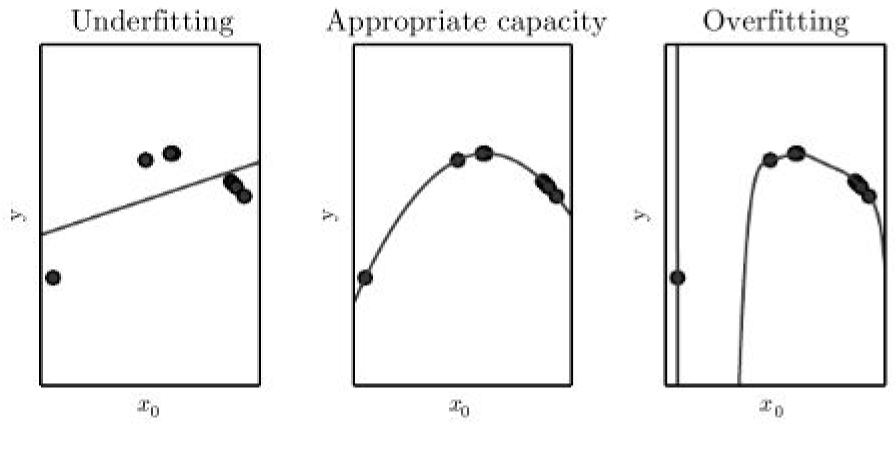
By introducing a new feature x^2 to a linreg model, we can learn a model which is quadratic in x

$$\hat{y} = b + w_1 x + w_2 x^2 .$$

Though the model implements a quadratic function of its input, the output is a linear function of the parameters / weights, so the normal equations can be used to train in closed form

We can get a polynomial of degree 9

$$\hat{y} = b + \sum_{i=1}^{9} w_i x^i .$$



Degree 9 polynomial overfits since its large capacity gives it nearly infinite solutions passing through the few points

Representational capacity - family of functions the learning algorithm can choose from when varying parameters to reduce a training objective
There are many functions which perform well on a set, so in practice, it is unlikely the best one is found
The effective capacity, therefore, is less than the representational capacity of the model family

VC dimension measures the capacity of a binary classifier
VC dimension is the largest possible value of m such that there exists a training set of m different x points the classifier can label arbitrarily

Error bounded from above by a quantity which is proportional to model capacity and inversely proportional to # of examples
Deep learning model capacities are hard to quantify ; the effective capacity is limited by a convex optimization problem, even though deep learning is non-convex
Generalization error typically has a parabolic shape curve as a function of model capacity

Non-parametric models are not limited by a finite sized parameter vector
Practical non-parametric models can be implemented by making their complexity a function of training set size
For example, nearest neighbor regression where the predicted value is the label of the closest point in the training set
$\hat{y} = y_i$ where $i = \arg\min ||\boldsymbol{X}_{i,:} - \boldsymbol{x}||_2^2.$
If the label values are averaged in order to break ties, then this algorithm is able to achieve the minimum possible error on any regression dataset
Non-parametric models can also be created by wrapping a parametric learning algorithm inside another algorithm increasing the number of parameters as needed

The Bayes error is the error incurred by an oracle (perfect model) making predictions from the true probability distribution $p(\boldsymbol{x}, y)$
It is a result of noise / external variables
Training and generalization error vary as the size of the training set varies ; for non- models, more data yields better generalization until the best error (Bayes error) is reached

The No Free Lunch Theorem
The no free lunch states that for all possible data generating distributions / all possible tasks, each classification algorithm has the same error rate when classifying previously unobserved points
However, we can narrow the distributions used in practice down to real world distributions so we can design learning algorithms performing well on this distribution

Design a machine learning algorithm to perform well on a specific task, by building a set of preferences into the learning algorithm

When these preferences are aligned with / factored into the learning problem the algorithm must solve, it performs better
We can control the performance of our algorithms by choosing what kind of functions they may draw solutions from, as well as the amount of these functions

If we give a learning algorithm a preference for one solution in its hypothesis space over another, the unpreferred solution will only be chosen if it fits the data significantly better

For example, we can modify the criterion linear regression to include weight decay
We add the squared L^2 norm as our preference

$$J(\boldsymbol{w}) = \text{MSE}_{\text{train}} + \lambda \boldsymbol{w}^\top \boldsymbol{w},$$

In general, we add a regularizer to the cost function
In the above example, the regularizer is $\Omega(\boldsymbol{w}) = \boldsymbol{w}^\top \boldsymbol{w}.$
Regularizing is a more general way of excluding a solution from our hypothesis space ; functions that are not in the hypothesis space can be considered to have an infinitely strong preferene against them
Regularization tries to reduce generalization error but not training error

Hyperparameters and Validation Sets
Hyperparameters are not learned by the learning algorithm itself
The validation set is used to learn hyperparamters
Since the validation set "trains" the hyperparameters, the validation set underestimates the generalization error, though typically by a smaller amount by the training set
Benchmarks on a test set can become stale, so data scientists move onto new ones

A small test set has sampling error around the estimated average test error
This leads to people using more computation to evaluate the mean test error
k-fold cross validation - partition of dataset is formed by splitting into k non-overlapping datasets
Test error is estimated by taking the average dataset across k trials
On trial i (where i ranges from 1 to k), the i-th subset is used as the test set

Estimators, Bias, and Variance
Point estimation tries to provide the single best prediction of some quantity of interest
It can be a single parameter or vector of parameters in some model

A point estimate of a parameter theta is given by $\hat{\boldsymbol{\theta}}.$
Let x be a set of m vectors {x^1 .. x^m} which are independent and identically distributed
A point estimator or statistic is any function of the data

$$\hat{\boldsymbol{\theta}}_m = g(\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}).$$

Note that g does not have to return an accurate value
The true parameter value theta is fixed but unknown

$\hat{\boldsymbol{\theta}}$ is a random variable since data is drawn from a random process

In function estimating, we are interested in estimating $\hat{f}$.
The function estimator is simply a point estimator in the function space

The bias of an estimator is defined as $\text{bias}(\hat{\boldsymbol{\theta}}_m) = \mathbb{E}(\hat{\boldsymbol{\theta}}_m) - \boldsymbol{\theta}$ where the expectation is over a random sample of data and theta is the true underlying weight vector used to generate the data distribution

Unbiased if $\text{bias}(\hat{\boldsymbol{\theta}}_m) = \mathbf{0}$ and $\mathbb{E}(\hat{\boldsymbol{\theta}}_m) = \boldsymbol{\theta}$.

Asymptotically unbiased if $\lim_{m\to\infty} \text{bias}(\hat{\boldsymbol{\theta}}_m) = \mathbf{0}$ which implies that $\lim_{m\to\infty} \mathbb{E}(\hat{\boldsymbol{\theta}}_m) = \boldsymbol{\theta}$.

Consider a set of m samples independent and identically distributed from a Bernoulli distribution with mean theta

$$P(x^{(i)};\theta) = \theta^{x^{(i)}}(1-\theta)^{(1-x^{(i)})}.$$

A common estimator for the theta parameter is the mean of the training samples

$$\hat{\theta}_m = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}.$$

To determine whether the estimator is biased, substitute the above equation into our bias equation

$$\begin{aligned}
\text{bias}(\hat{\theta}_m) &= \mathbb{E}[\hat{\theta}_m] - \theta \\
&= \mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m} x^{(i)}\right] - \theta \\
&= \frac{1}{m}\sum_{i=1}^{m} \mathbb{E}\left[x^{(i)}\right] - \theta \\
&= \frac{1}{m}\sum_{i=1}^{m} \sum_{x^{(i)}=0}^{1} \left(x^{(i)}\theta^{x^{(i)}}(1-\theta)^{(1-x^{(i)})}\right) - \theta \\
&= \frac{1}{m}\sum_{i=1}^{m} (\theta) - \theta \\
&= \theta - \theta = 0
\end{aligned}$$

Since the bias = 0, we can say our estimator is unbiased

Consider a set of m samples independently and identically distributed according to a Gaussian distribution $p(x^{(i)}) = \mathcal{N}(x^{(i)}; \mu, \sigma^2),$ where i is from 1 to m

The gaussian PDF is given by
$$p(x^{(i)}; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2}\frac{(x^{(i)}-\mu)^2}{\sigma^2}\right).$$
Sample mean is a common estimator of the Gaussian mean parameter (mu)

$$\hat{\mu}_m = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$$

To determine the bias of the sample mean, calculate its expectation as done above

$$\begin{aligned}
\text{bias}(\hat{\mu}_m) &= \mathbb{E}[\hat{\mu}_m] - \mu \\
&= \mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m} x^{(i)}\right] - \mu \\
&= \left(\frac{1}{m}\sum_{i=1}^{m} \mathbb{E}\left[x^{(i)}\right]\right) - \mu \\
&= \left(\frac{1}{m}\sum_{i=1}^{m} \mu\right) - \mu \\
&= \mu - \mu = 0
\end{aligned}$$

The expected value of x according to the distribution is mu, so there is no bias when the sample mean is the estimator

We will consider two estimators of the variance of a Gaussian distribution
The first estimator of variance is the sample variance

$$\hat{\sigma}_m^2 = \frac{1}{m}\sum_{i=1}^{m}\left(x^{(i)} - \hat{\mu}_m\right)^2,$$

where mu is the sample mean

We want to compute $\text{bias}(\hat{\sigma}_m^2) = \mathbb{E}[\hat{\sigma}_m^2] - \sigma^2$

$$\begin{aligned}
\mathbb{E}[\hat{\sigma}_m^2] &= \mathbb{E}\left[\frac{1}{m}\sum_{i=1}^{m}\left(x^{(i)} - \hat{\mu}_m\right)^2\right] \\
&= \frac{m-1}{m}\sigma^2
\end{aligned}$$

The bias is therefore $\hat{\sigma}_m^2$ is $-\sigma^2/m.$
The sample variance is a biased estimator

The unbiased sample variance estimator $\tilde{\sigma}_m^2 = \frac{1}{m-1}\sum_{i=1}^{m}\left(x^{(i)} - \hat{\mu}_m\right)^2$ is an alternate approach

The expected value is therefore sigma^2

$$\begin{aligned}\mathbb{E}[\tilde{\sigma}_m^2] &= \mathbb{E}\left[\frac{1}{m-1}\sum_{i=1}^{m}\left(x^{(i)} - \hat{\mu}_m\right)^2\right] \\ &= \frac{m}{m-1}\mathbb{E}[\hat{\sigma}_m^2] \\ &= \frac{m}{m-1}\left(\frac{m-1}{m}\sigma^2\right) \\ &= \sigma^2.\end{aligned}$$

The bias is therefore 0

However, unbiased estimators are not always the best estimators

Variance is how much we expect the estimator to vary as a function of the sample

The variance of an estimator is its variance

$$\mathrm{Var}(\hat{\theta})$$

The standard error is the square root of variance

$$\mathrm{SE}(\hat{\theta})$$

These measure how much a random sample from the underlying generator would vary

$$\mathrm{SE}(\hat{\mu}_m) = \sqrt{\mathrm{Var}\left[\frac{1}{m}\sum_{i=1}^{m}x^{(i)}\right]} = \frac{\sigma}{\sqrt{m}},$$

Standard error of the mean is given by

where sigma^2 is the true variance of the samples x^i

The square root of the sample variance nor the square root of the unbiased estimator of the variance provide an unbiased estimate of the standard deviation - they are slightly lower

They are used in practice, and are good for large values of m

Standard error of the mean is useful

We can estimate generalization error by computing sample mean of error on the test set

The central limit theorem tells us that the sample mean will be approximately distributed with a normal distribution (after many re-samples)

- We can use this to compute the probability that the true expectation falls in any chosen interval

The 95% confidence interval centered at the mean is

$$(\hat{\mu}_m - 1.96\mathrm{SE}(\hat{\mu}_m), \hat{\mu}_m + 1.96\mathrm{SE}(\hat{\mu}_m)),$$

Algorithm A is better than algorithm B if the upper bound of the 95% confidence interval for the error of algorithm A is less than the lower bound of the 95% of the confidence interval for algorithm B

Once again, we look at a Bernoulli distribution, wanting to compute the estimator's variance
Consider a set of m examples {x^1...x^m}

$$P(x^{(i)};\theta) = \theta^{x^{(i)}}(1-\theta)^{(1-x^{(i)})}$$

$$\begin{aligned}
\mathrm{Var}\left(\hat{\theta}_m\right) &= \mathrm{Var}\left(\frac{1}{m}\sum_{i=1}^{m} x^{(i)}\right) \\
&= \frac{1}{m^2}\sum_{i=1}^{m}\mathrm{Var}\left(x^{(i)}\right) \\
&= \frac{1}{m^2}\sum_{i=1}^{m}\theta(1-\theta) \\
&= \frac{1}{m^2}m\theta(1-\theta) \\
&= \frac{1}{m}\theta(1-\theta)
\end{aligned}$$

The variance decreases as a function of m, the number of examples in the dataset
Notice how a small theta decreases variance

Bias measures the expected deviation from the true value of the parameter, whereas variance measures the deviation from the expected estimator value that a particular sampling method may cause

How do we choose between bias and variance when choosing an estimator?

- Empirically effective - cross validation
- We can also compare the mean squared error of the estimates
- $$\begin{aligned}\mathrm{MSE} &= \mathbb{E}[(\hat{\theta}_m - \theta)^2] \\ &= \mathrm{Bias}(\hat{\theta}_m)^2 + \mathrm{Var}(\hat{\theta}_m)\end{aligned}$$ Recall MSE measures overall expected deviatoin between the estimator and true value of the parameter theta

Generally, increasing capacity increases variance and decreases bias

We prefer that $\mathrm{plim}_{m\to\infty}\hat{\theta}_m = \theta.$ where m is the number of training examples plim indicates convergence in probability, meaning for any epsilon greater than 0

$$P(|\hat{\theta}_m - \theta| > \epsilon) \to 0 \text{ as } m \to \infty.$$

The probability that the distance between the two is greater than some threshold approaches 0 as the number of training examples increases

The conditions above are known as consistency
Strong consistency refers to the almost sure convergence of predicted theta to theta
Almost sure convergence of a sequence of random variables {x_1 .. x_m} to a vector x occurs when

$$p(\lim_{m\to\infty} \mathbf{x}^{(m)} = \boldsymbol{x}) = 1.$$

Consistency ensures that bias decreases as number of data exapmles grows
Asymptotic unbiasedness does not imply consistency
Consider estimating the mean of a normal distribution with a dataset of m samples {x_1 .. x_m }

The first example is an unbiased estimator $\hat{\theta} = x^{(1)}$. and $\mathbb{E}(\hat{\theta}_m) = \theta$ so the estimator is always unbiased, and therefore asymptotically unbiased

However, this is not a consistent estimator since the following is not true $\hat{\theta}_m \to \theta \text{ as } m \to \infty.$

Maximum Likelihood Estimation

We want to derive good estimators, rather than guessing and checking variance / bias

Consider a set of m examples $\mathbb{X} = \{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ drawn from $p_{\text{data}}(\mathbf{x})$.

$p_{\text{model}}(\mathbf{x}; \boldsymbol{\theta})$ is a parametric group of probability distributions over the same space ; it maps any configuration of x to a real number estimating its probability

The max likelihood estimator is defined as

$$\begin{aligned}\boldsymbol{\theta}_{\text{ML}} &= \arg\max_{\boldsymbol{\theta}} p_{\text{model}}(\mathbb{X}; \boldsymbol{\theta}) \\ &= \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{m} p_{\text{model}}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta})\end{aligned}$$

The theta for which the probability of the entire set X is maxed

This is computationally inefficient due to things like underflow, so taking the log (which keeps the argmax the same) while using a summation is more efficient

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log p_{\text{model}}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}).$$

The ML estimation minimizes the dissimilarity between the empirical distribution $\hat{p}_{\text{data}}$ and the model distribution, with dissimilarity being measured through KL divergence

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

The expected value of their difference where x is distributed with respect to p_{data}

Since the term on the left is not affected by the model's probability, all we need to minimize is

$$-\mathbb{E}_{\mathbf{x}\sim\hat{p}_{\text{data}}}[\log p_{\text{model}}(\boldsymbol{x})]$$ which is the same as maximizing the positive version

Minimizing the KL divergence corresponds to minimizing the cross entropy between the distributions ; any loss consisting of a negative log-likelihood is a cross entropy between the empirical distribution and the probability distribution

- Mean squared error is the cross entropy between the empirical distribution and a Gaussian model

Therefore, max likelihood is an attempt to make the model distribution match the empirical distribution (as seen by random sampling / however we chose the training data)

While theta is the same whether we maximize the likelihood of theta or minimize the KL divergence, their objective functions have different values - i.e we minimize a cost function

Maximum likelihood becomes minimization of cross entropy or min. of negative log-likelihood

- KL divergence has a known minimum of zero
- Negative log likelihood can become negative when x is real-valued

Use maximum likelihood estimator to estimate a conditional probability P(y | x ; theta) in order to predict y given x

If X is all the inputs and Y is the observed targets, the conditional max likelihood estimator becomes

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} P(\boldsymbol{Y} \mid \boldsymbol{X}; \boldsymbol{\theta}).$$

which can be decomposed to

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log P(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta}).$$

if they are i.i.d (independent & identically distributed)

The model for linear regression now produces a conditional distribution p(y | x) instead of a prediction y-hat

With an infinitely large training set, we have the same x but different y (empirically likely)

The goal of the learning algorithm is now to fit the distribution p(y | x) to all the different y-vaues which make sense / are compatible with x

Define $p(y \mid \boldsymbol{x}) = \mathcal{N}(y; \hat{y}(\boldsymbol{x}; \boldsymbol{w}), \sigma^2).$ Generate a distribution of possible y

The function $\hat{y}(\boldsymbol{x}; \boldsymbol{w})$ predicts the mean Gaussian (so we can parametrize the normal model) and the variance is fixed to some value chosen by the user
This choice of the functional form of p(y | x) causes the maximum likelihood to yield the same linear regression algorithm as before
The conditional log likelihood equation is given by

$$\sum_{i=1}^{m} \log p(y^{(i)} \mid \boldsymbol{x}^{(i)}; \boldsymbol{\theta})$$

$$= -m \log \sigma - \frac{m}{2} \log(2\pi) - \sum_{i=1}^{m} \frac{\left\|\hat{y}^{(i)} - y^{(i)}\right\|^2}{2\sigma^2},$$

where y-hat of i is the prediction of the i-th input and m is the number of training examples

Comparing log likelihood with MSE $\mathrm{MSE}_{\mathrm{train}} = \frac{1}{m} \sum_{i=1}^{m} ||\hat{y}^{(i)} - y^{(i)}||^2,$, it is evident that maximizing the log likelihood w/ respect to the weight vector w (or theta in this case) gives the same estimate as minimizing the mean-squared error
They have different values, but the same optima

The max likelihood estimator is the best, considering rate of convergence, as the number of examples approaches infinite
The max likelihood converges to the true value of the parameter (weights in this case) if the following conditions are satisfied:

- The true distribution must lie in the model family $p_{\text{model}}(\cdot; \boldsymbol{\theta})$
- The true distribution must have only one value of theta / 1 weight vector

Consistent estimators vary in their statistic efficiency
- More statistically efficient if an estimator obtains lower generalization error for a fixed number of samples m or requires fewer examples to achieve a certain low error
- Statistic efficiency is used for the parametric case, where we want the weight vector

No consistent estimator has a lower MSE than the max likelihood estimator for large m

Bayesian Statistics
A different approach is to consider all possible values of theta when predicting ; this is known as Bayesian statistics
The frequentist view of optimization is that the true theta is unknown while predicted theta is random since it is a function of random sampling of the dataset
Bayesian uses probability to reflect degrees of certainty of states of knowledge
The dataset is not random, and theta is unknown so it is represented as a random variable

Before observing the data, we represent our knowledge of theta using the prior probability distribution $p(\boldsymbol{\theta})$, called the prior

We use a prior distribution with high entropy to reflect uncertainty in theta before observing data

Some priors assume theta lies in a finite range with a uniform distribution, others make the coefficients very small in magnitude

Consider a set of m examples

We can recover the effect of the data on our belief / constraint on theta by combining the data likelihood with the prior using Bayes' rule

$$p(\boldsymbol{\theta} \mid x^{(1)}, \ldots, x^{(m)}) = \frac{p(x^{(1)}, \ldots, x^{(m)} \mid \boldsymbol{\theta}) p(\boldsymbol{\theta})}{p(x^{(1)}, \ldots, x^{(m)})}$$

When Bayesian estimation is used, the prior begins as a Gaussian distribution with high entropy and observation of the data causes a loss in entropy and the estimates concentrate around a few highly likely value of the parameter

Bayesian approach makes predictions using a full distribution of estimated values of theta

$$p(x^{(m+1)} \mid x^{(1)}, \ldots, x^{(m)}) = \int p(x^{(m+1)} \mid \boldsymbol{\theta}) p(\boldsymbol{\theta} \mid x^{(1)}, \ldots, x^{(m)}) \, d\boldsymbol{\theta}.$$

After having observed m examples, if we are uncertain about theta, this is incorporated into any predictions which we make

The frequentist approach addresses uncertainty by estimating variance

Bayesian integrates over uncertainty, which uses the laws of probability to protect against overfit

Another key difference is caused by the Bayesian prior distribution

In practice, the prior causes bias towards models which are smooth and simple

Bayesian methods are good at generalizing when training data is limited, but are computationally inefficient when the # of training examples is large

Bayesian estimation can be used for learning linreg parameters

$$\hat{y} = \boldsymbol{w}^\top \boldsymbol{x}.$$

Given m training examples, we can express the prediction of y over the entire set as

$$\hat{\boldsymbol{y}}^{(\text{train})} = \boldsymbol{X}^{(\text{train})} \boldsymbol{w}.$$

Expressing as a conditional distirbution on y

$$\begin{aligned} p(\boldsymbol{y}^{(\text{train})} \mid \boldsymbol{X}^{(\text{train})}, \boldsymbol{w}) &= \mathcal{N}(\boldsymbol{y}^{(\text{train})}; \boldsymbol{X}^{(\text{train})} \boldsymbol{w}, \boldsymbol{I}) \qquad (5.71) \\ &\propto \exp\left(-\frac{1}{2}(\boldsymbol{y}^{(\text{train})} - \boldsymbol{X}^{(\text{train})}\boldsymbol{w})^\top (\boldsymbol{y}^{(\text{train})} - \boldsymbol{X}^{(\text{train})}\boldsymbol{w})\right), \end{aligned}$$

Assume the Gaussian variance on y is one, to determine the posterior (after the prior is considered) distribution, we first need to specify a prior distribution

We use the Gaussian as the prior distribution

It is sometimes difficult to express prior beliefs in terms of the parameters of the model

$\boldsymbol{\mu}_0$ and $\boldsymbol{\Lambda}_0$ $$p(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w}; \boldsymbol{\mu}_0, \boldsymbol{\Lambda}_0) \propto \exp\left(-\frac{1}{2}(\boldsymbol{w} - \boldsymbol{\mu}_0)^\top \boldsymbol{\Lambda}_0^{-1}(\boldsymbol{w} - \boldsymbol{\mu}_0)\right),$$

where are the prior distribution mean vector and covariance matrix

We can now find the posterior distribution

$$p(\boldsymbol{w} \mid \boldsymbol{X}, \boldsymbol{y}) \propto p(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{w}) p(\boldsymbol{w}) \tag{5.74}$$

$$\propto \exp\left(-\frac{1}{2}(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})^\top (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})\right) \exp\left(-\frac{1}{2}(\boldsymbol{w} - \boldsymbol{\mu}_0)^\top \boldsymbol{\Lambda}_0^{-1}(\boldsymbol{w} - \boldsymbol{\mu}_0)\right) \tag{5.75}$$

$$\propto \exp\left(-\frac{1}{2}\left(-2\boldsymbol{y}^\top \boldsymbol{X}\boldsymbol{w} + \boldsymbol{w}^\top \boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{w} + \boldsymbol{w}^\top \boldsymbol{\Lambda}_0^{-1}\boldsymbol{w} - 2\boldsymbol{\mu}_0^\top \boldsymbol{\Lambda}_0^{-1}\boldsymbol{w}\right)\right).$$

$$\boldsymbol{\Lambda}_m = \left(\boldsymbol{X}^\top \boldsymbol{X} + \boldsymbol{\Lambda}_0^{-1}\right)^{-1}$$

$$\boldsymbol{\mu}_m = \boldsymbol{\Lambda}_m \left(\boldsymbol{X}^\top \boldsymbol{y} + \boldsymbol{\Lambda}_0^{-1}\boldsymbol{\mu}_0\right).$$

Using these variables, rewrite the posterior as a Gaussian distribution

$$p(\boldsymbol{w} \mid \boldsymbol{X}, \boldsymbol{y}) \propto \exp\left(-\frac{1}{2}(\boldsymbol{w} - \boldsymbol{\mu}_m)^\top \boldsymbol{\Lambda}_m^{-1}(\boldsymbol{w} - \boldsymbol{\mu}_m) + \frac{1}{2}\boldsymbol{\mu}_m^\top \boldsymbol{\Lambda}_m^{-1}\boldsymbol{\mu}_m\right)$$

$$\propto \exp\left(-\frac{1}{2}(\boldsymbol{w} - \boldsymbol{\mu}_m)^\top \boldsymbol{\Lambda}_m^{-1}(\boldsymbol{w} - \boldsymbol{\mu}_m)\right).$$

All terms which don't include the vector w have been omitted => implied by the fact that the PDF must integrate to 1

If we set $\boldsymbol{\Lambda}_0 = \frac{1}{\alpha}\boldsymbol{I}$, then μ_m gives the same estimate of w as frequentist linear regression with weight decay penalty

Alpha cannot be 0 for bayesian learning

Bayesian estimate provides a covariance matrix showing how likely different values of w are

While in theory the best approach is to use the full posteiror distribution over theta, it is often desirable to have a single point estimate as it provides a tractable approximation

We can still allow the prior to influence the choice of the point estimate, through the maximum a posteriori point (MAP) estimate

The MAP estimate chooses the point of maximal posterior probability

$$\boldsymbol{\theta}_{\text{MAP}} = \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{\theta} \mid \boldsymbol{x}) = \arg\max_{\boldsymbol{\theta}} \log p(\boldsymbol{x} \mid \boldsymbol{\theta}) + \log p(\boldsymbol{\theta}).$$

log p(x | theta) gives the standard log likelihood term and log p(theta) corresponds to the prior distribution

Consider a linear regression model with Gaussian prior on the weights given by

$\mathcal{N}(\boldsymbol{w}; \mathbf{0}, \frac{1}{\lambda}\boldsymbol{I}^2)$

The log prior term is proportional to the weight decay penalty
Therefore, MAP Bayesian inference with a Gaussian prior on the weights is weight decay
Regularized estimation strategies can be interpreted as making the MAP approximation to Bayesian inference

- This applies when the regularization consists of adding another term to the objective function where the term is related to log p(thets)
- Regularizer terms which depend on the data don't correspond to MAP Bayesian inference which cannot be modeled by a prior probability distribution

MAP Bayesian allows us to create complicated regularization terms

Supervised Learning Algorithms
Most supervised learning algorithms are based on estimating a probability distribution p(y | x) ; we can do this using maximum likelihood estimation to find the best parameter theta for a parametric family of distributions p(y | x ; theta)

Linear regression corresponds to $p(y \mid \boldsymbol{x}; \boldsymbol{\theta}) = \mathcal{N}(y; \boldsymbol{\theta}^\top \boldsymbol{x}, \boldsymbol{I}).$
Linear regression's prob. distributions can be generalized to the classification scenario by defining a different family of probability distributions which sum up to a probability of 1

A distribution over a binary variable has a mean that must be between 0 and 1, so we cannot use any real valued number as the mean
We can squash the output of the linear function into the interval (0, 1) and interpret that value as the probability $p(y = 1 \mid \boldsymbol{x}; \boldsymbol{\theta}) = \sigma(\boldsymbol{\theta}^\top \boldsymbol{x}).$
This is known as logistic regression
In logistic regression, there is no closed form solution to find the optimal weights, so we must max the log likelihood, or minimize the negative log likelihood with gradient descent

Support vector machine is driven by a linear function $w^T x + b$
SVM predicts the positive class is present when $w^T x + b$ is positive
SVMs are known for their kernel trick, meaning that many ML algorithms can be written exclusively in terms of dot products between examples

The linear function can be re-written as $$\boldsymbol{w}^\top \boldsymbol{x} + b = b + \sum_{i=1}^{m} \alpha_i \boldsymbol{x}^\top \boldsymbol{x}^{(i)}$$ where x^i is a training example and a_i is a vector of coefficients

Rewriting the function in this way lets us replace x by the output of a feature function $\phi(\boldsymbol{x})$ and the dot product with a function $k(\boldsymbol{x}, \boldsymbol{x}^{(i)}) = \phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{x}^{(i)})$ called a kernel

The dot product is an inner product analogous to $\phi(\boldsymbol{x})^\top \phi(\boldsymbol{x}^{(i)})$.
For some feature spaces, we may have to integrate instead of summate to find the inner prod

Replacing the dot product with kernel evaluations, we can make predictions using

$$f(\boldsymbol{x}) = b + \sum_i \alpha_i k(\boldsymbol{x}, \boldsymbol{x}^{(i)}).$$

The function is nonlinear with respect to x but the relationship between $\phi(\boldsymbol{x})$ and f(x) is linear ; the relationship between alpha and f is also linear
The kernel based function is the same as preprocessing the inputs using phi(x), and learning a linear model with respect to phi in the new space
Kernel trick allows us to learn models that are nonlinear as a function of x using convex optimization techniques which are guaranteed to converge efficiently since phi is fixed and alpha is the only thing being optimized (this allows for a linear decision function)
The kernel function is implemented more efficiently than constructing two phi(x) vectors and taking their x product

In some cases, phi(x) can be infinite dimensional resulting in an infinite computational cost
k(x, x') is a nonlinear, but tractable function of x even when phi(x) is intractable

- e.g a kernel function in infinite dimensions can be equivalent to the minimum

Gaussian kernel is most commonly used
$k(\boldsymbol{u}, \boldsymbol{v}) = \mathcal{N}(\boldsymbol{u} - \boldsymbol{v}; 0, \sigma^2 \boldsymbol{I})$ where $\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$ is standard normal density
This kernel is also known as radial basis function (RBF) since its value decreases along lines in v space radiating outward from u
The RBF corresponds to a dot product in an infinite dimensional space

Gaussian kernel performs template matching - a training example x associated with a label y becomes a template for class y
When a test point x' is near x by Euclidean distance, the Gaussian kernel has a large response, putting a large weight on the associated training label y
The prediction combines many training labels weighted by similarity of the training examples

Kernel machines or kernel methods are the category of algorithms using the kernel trick
The cost of evaluating the decision function is linear in the number of training examples since each i-th example contributes a term to the decision function
SVMs mitigate this by learning an alpha vector consists of mostly 0s, so the kernel function only needs to be evaluated for terms with non-zero alpha
These training examples are known as support vectors

Unsupervised Learning

In unsupervised learning, we are generally interested in looking for a representation which preserves as much information about x as possible while obeying a constraint which keeps the data simpler than x itself

Simpler representation can be defined by lower dimensional representations, sparse representations, and independent representations

Sparse representations results in an overall structure which distributes data along the axis of the representation space

Independent representations disentangle sources of variation / randomness / noise so that the dimensions of the representation are statistically independent

PCA can be viewed as an unsupervised learning algorithm which learns a representation of the data, using lower dimensionality than the input and a representation whose elements have no correlation with one another (first step towards statistical independence)

PCA preserves as much info as possible as measured by least squares reconstruction error with a orthogonal, linear transformation of the data

How does PCA decorrelate the original data representation X?

Assume that the data in our m x n dimensional design matrix X has a mean of 0 E[x] = 0 (subtract the mean from all the samples)

The unbiased sample covariance matrix (x) associated with X is given by

$$\mathrm{Var}[\boldsymbol{x}] = \frac{1}{m-1}\boldsymbol{X}^\top \boldsymbol{X}.$$

PCA finds a representation through a linear transformation $\boldsymbol{z} = \boldsymbol{x}^\top \boldsymbol{W}$ where Var[z] is diagonal

As proved in earlier chapters, the principal components of a design matrix X are given by eigenvectors of $\boldsymbol{X}^\top \boldsymbol{X}$, and therefore:

$$\boldsymbol{X}^\top \boldsymbol{X} = \boldsymbol{W}\boldsymbol{\Lambda}\boldsymbol{W}^\top.$$

The principal components may be obtained through singular value decomposition ; they are the right singular vectors of X

Let W be the right singular vectors in the decomposition $\boldsymbol{X} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{W}^\top.$

Recover the original eigenvector equation with W as the eigenvector basis

$$\boldsymbol{X}^\top \boldsymbol{X} = \left(\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{W}^\top\right)^\top \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{W}^\top = \boldsymbol{W}\boldsymbol{\Sigma}^2\boldsymbol{W}^\top.$$

The SVD shows us that PCA results in a diagonal Var[z]

Using the SVD of X, we can express its variance

$$\begin{aligned}
\operatorname{Var}[\boldsymbol{x}] &= \frac{1}{m-1}\boldsymbol{X}^\top \boldsymbol{X} \\
&= \frac{1}{m-1}(\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{W}^\top)^\top \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{W}^\top \\
&= \frac{1}{m-1}\boldsymbol{W}\boldsymbol{\Sigma}^\top \boldsymbol{U}^\top \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{W}^\top \\
&= \frac{1}{m-1}\boldsymbol{W}\boldsymbol{\Sigma}^2 \boldsymbol{W}^\top,
\end{aligned}$$

where $U^T U$ is the identity matrix I
If $z = x^T W$, we can ensure the covariance of z is diagonal as required

$$\begin{aligned}
\operatorname{Var}[\boldsymbol{z}] &= \frac{1}{m-1}\boldsymbol{Z}^\top \boldsymbol{Z} \\
&= \frac{1}{m-1}\boldsymbol{W}^\top \boldsymbol{X}^\top \boldsymbol{X}\boldsymbol{W} \\
&= \frac{1}{m-1}\boldsymbol{W}^\top \boldsymbol{W}\boldsymbol{\Sigma}^2 \boldsymbol{W}^\top \boldsymbol{W} \\
&= \frac{1}{m-1}\boldsymbol{\Sigma}^2,
\end{aligned}$$

$W^T T = 1$
This shows us that when we project the data from x to z through the linear transformation W, the resulting representation has a diagonal covariance matrix implying that the elements of z are uncorrelated with one another
By doing this, PCA disentangles unknown factors of variation underlying the data
In PCA, this disentangling occurs by finding a rotation of the input space that aligns the principal axes of variance with the basis of the new lower dimensionality representation space and projecting onto these principal axes

K-means clustering divides the training set into k different clusters of examples near each other
It produces a k-dimensional one hot code vector h representing an input x ; if x belongs to cluster i, then the element $h_i = 1$ and all other entries are 0 (sparse representation)
The representation is not distributed, but it is computationally efficient since it only uses 1 int

K-means starts by initializing k different centroids $\{u^1 \ldots u^k\}$ to different values and performing two steps until convergence

- Each training example is assigned to cluster i, where i is the index of the nearest centroid
- Each centroid u^i is updated to the mean of all training examples in its cluster

The average Euclidean distance from a cluster centroid to the members of the cluster tells us how well we can reconstruct the data, but we do not know how well cluster assignments correspond to real world properties

We can hope to find a clustering which identifies a particular feature, but instead, the algorithm finds a valid feature which is not relevant to our task
This is why a distributed representation may be preferred - it gives a more precise analysis of correlation / similarity

Stochastic Gradient Descent
The cost function often decomposes into a sum over training examples of a per-example loss function

$$J(\boldsymbol{\theta}) = \mathbb{E}_{\mathbf{x},\mathrm{y}\sim\hat{p}_{\text{data}}} L(\boldsymbol{x}, y, \boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta})$$

where L is the per-example loss

$$L(\boldsymbol{x}, y, \boldsymbol{\theta}) = -\log p(y \mid \boldsymbol{x}; \boldsymbol{\theta}).$$

Gradient descent requires computing

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}).$$

The gradient is an expectation which may be approximated using a minibatch of examples

$$\mathbb{B} = \{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m')}\}$$

The gradient estimate is

$$\boldsymbol{g} = \frac{1}{m'}\nabla_{\boldsymbol{\theta}}\sum_{i=1}^{m'} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}).$$

which then follows the estimated gradient downhill

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \boldsymbol{g},$$

Though grad. descent is regarded as slow, it works very well with neural nets

SGD is used outside of deep learning to train linear models on large datasets
The cost per update does not depend on the training set size m for a fixed model ; larger models are used as training set size increases
The asymptotic cost of training a model is O(1) as a function of m when it approaches infinite

Prior to deep learning, the best way to learn a nonlinear function was to use the kernel trick combined with linear model algorithms which required constructing an m x m matrix

$$G_{i,j} = k(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}).$$

Constructing this matrix has a high computational cost
Deep learning provided a scalable way of training nonlinear models on large datasets

Building a Machine Learning Algorithm

Deep learning algorithms consist of combining a specification of dataset, a cost function, and an optimization procedure

Linreg combines a dataset consisting of $\boldsymbol{X}$ and $\boldsymbol{y}$, the cost function of $J(\boldsymbol{w}, b) = -\mathbb{E}_{\mathbf{x},\mathrm{y}\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(y \mid \boldsymbol{x}),$ the model specification $p_{\text{model}}(y \mid \boldsymbol{x}) = \mathcal{N}(y; \boldsymbol{x}^{\top}\boldsymbol{w} + b, 1)$. and an optimization algorithm which found where the gradient of the cost function is 0

The most common cost function is negative log likelihood so that minimizing the cost function maximizes
The cost function may also include additional terms for regularization
If the model is nonlinear, the solution is typically no longer closed form so we choose an iterative numerical optimization procedure like grad. descent

The view of combining a specification of data, cost function, and optimization procedure cna be applied to unsupervised learning algorithms such as PCA where the data is X and the unsupervised cost / model can be represented with $J(\boldsymbol{w}) = \mathbb{E}_{\mathbf{x}\sim\hat{p}_{\text{data}}} ||\boldsymbol{x} - r(\boldsymbol{x}; \boldsymbol{w})||_2^2$
where the model has w with norm 1 and reconstruction function $r(\boldsymbol{x}) = \boldsymbol{w}^{\top}\boldsymbol{x}\boldsymbol{w}.$

Recognizing the view of models this way lets us see that different algorithms are a part of a taxonomy of methods for doing related tasks through similar methods

Challenges Motivating Deep Learning
Deep learning is used to overcome the intense computational requirements of learning complicated functions in high dimensional spaces

The curse of dimensionality is the problem which arises with the number of configurations increasing exponentially as the number of variables increases
In high dimensional spaces, since the # of configurations is huge, a typical data point may not be close to others

We can express priors as directly influencing the function itself and only indirectly acting on the parameters (as opposed to affecting them directly through a probability distribution)
Priors can be biased / expressed implicitly, by using the algorithm in such a way to choose a particular class of functions over another (which is not always possible to do through prior distributions)

An implicit prior example is the smoothness prior or local constancy prior which states a function should not change very much in a small region
These work for simple algorithms, but do not generalize well to AI-level statistical challenges

The model is encouraged to learn the given f* when dealing with local constancy
$f^*(\boldsymbol{x}) \approx f^*(\boldsymbol{x} + \epsilon)$ for most configurations x and small changes e
If we know a good answer for x, then that answer works for other x' values nearby that x
We average if we have several labeled examples close to the point we are estimating

One example to local constancy is KNN, which are constant over each region containing all points x which have the same set of k nearest neighbors
While KNN directly copies the output of most nearby examples, other algorithms interpolate
An important class of kernels is the local kernels where $k(\boldsymbol{u}, \boldsymbol{v})$ is large as the distance between u and v increases
A local kernel can be thought of as a similarity function which performs template matching by measuring how closely a test example x resembles each training example x^i
Deep learning was made to get past the failures of template matching

Generally, to distinguish O(k) regions in input space, you need O(k) examples for methods like KNN and decision trees
There are typically O(k) parameters with O(1) parameters associated with each of the regions

To represent a complex function which has more regions to be distinguished then the number of examples, you need more than just an assumption of smoothness
The smoothness assumption only works if there are enough examples to observe maxima and minima of the true underlying function

If the function behaves differently in a wide number of regions (much greater than # of training examples), then we can add some extra assumptions about the underlying data distribution which will let us define $O(2^K)$ regions with O(k) examples
We can then generalize non-locally

We could add constraints which we know based on our task, but this usually is not good for AI since we want to generalize to a large variety of related tasks
We assume the data was generated by a composition of the given features
These mild assumptions allow an exponential increase in the relationship between the # of examples and # of regions to be distinguished, thereby countering the curse of dimensionality

Manifold Learning
A manifold is a connected region
In math, it is a set of points associated with a neighborhood around each point
We experience the surface of the world is a 2D plane, but it is in fact a spherical manifold

Existence of transformations which can be applied to move on the manifold from one position to a neighboring position (similar to North, west, east and south in our real life manifold)

In ML, manifold tends to be used loosely to designate a connected set of points that can be approximated well by considering only a small # of dimensions embedded in a high dim. space
Each dimension corresponds to a local direction of variation
We allow the dimensionality of the manifold to vary from one point to another, which often happens when a manifold intersects itself

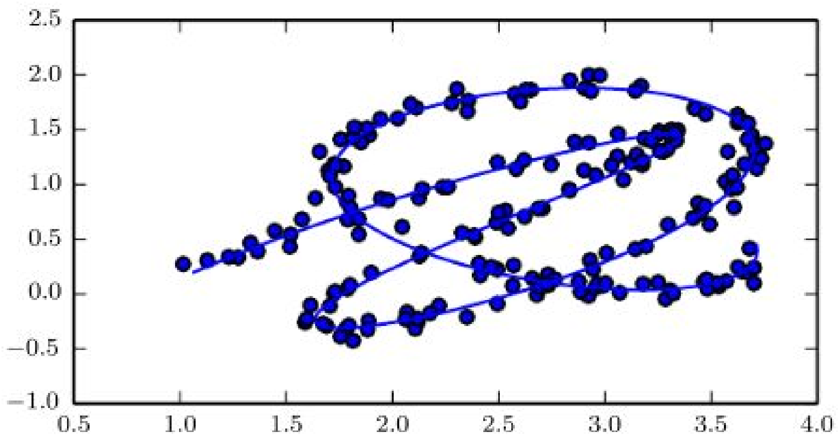


Figure 5.11: Data sampled from a distribution in a two-dimensional space that is actually concentrated near a one-dimensional manifold, like a twisted string. The solid line indicates the underlying manifold that the learner should infer.

One problem in ML is attempting to learn functions with variations throughout all of R^n
Manifold learning surmounts this by assuming most of R^N has useless inputs, and that interesting inputs only occur along a collection of manifolds containing a subset of points with interesting variations in the output occuring only along directions that lie on the manifold or with interesting variations occuring when moving between manifolds

The key assumption of manifold learning is that probability mass is highly concentrated
The assumption that the data lies along a low dimensional manifold may not always be correct
In AI tasks such as image processing, however, manifold learning is at least approximately correct

- The first observation in favor of this manifold hypothesis is that probability distribution over images is highly concentrated in real life
- Uniform random noise almost never represents structured input from real data
- If you generate a document for example, the chance that it will be natural language like how we speak in English is virtually nothing
- Images encountered in AI applications occupy a negligible proportion of the image space's total volume (which can be attained by picking a random pixel from every image to generate each pixel in a new image, which gives a static / noise pattern)

Concentrated probability distributions are not sufficient to show that the data lies on a small number of manifolds

The examples we encounter must be connected to other examples, with each example being surrounded by highly similar examples which may be reached by applying transformations to traverse the manifold

The second argument in favor of the manifold hypothesis is that we can imagine such transformations (e.g dimming lights, changing rotation), allowing us to trace out a manifold in image space which can traverse between images used in a particular application

When the data lies in a low dimension manifold, it is natural to represent data in manifold coordinates rather than in terms of R^N

- e.g roads are 1D manifolds embedded in 3D space, and manifold coordinates could be thought of as addresses instead of 3D coordinates