ML Lecture Notes: Wk 1 — Intellectual History of AI/ML

1 Computational Theory of the Mind

Hobbes - "reason is nothing but reckoning"

+ others: idea that brains compute –> can be approximated

2 Cybernetics - 1950s

3 Early AI - 1956-70s

Perceptron

Supervised learning: Given examples X with labels y, predict:

- Classification
- regression

4 AI Inter (mid 70s-80s

- lack of funding
- hardware not ready
- expectations about scaling were unreasonable
-

5 Unsupervised learning

6 AI Thaw - 1980s

6.1 Knowledge based systems

- “Don’t tell the program what to do, tell it what to know.”
- Knowledge-base
- Inference engine

6.2 Expert System

- For a very specific domain
- Intended to emulate experts

6.3 Neural Nets (returned)

- back propagation

7 AI Winter II: late 80s-early 90s

- again funding dried up
- commercial busts + much of the hardware undergirding was killed by IBM and Apple (i.e. the move to personal computing)
- issues with neural nets: vanishing gradient problem
- embodied reason???

8 Reinforcement Learning

- given a set of states s, with actions a and rewards r
- Action
- learn $s_t(a)$ to max $\sum r_t$

9 Maturing ML: mid 90s - early 2010s

- Rigor
  - Probability, decision theory etc
  - Judea Pearl
- Evaluation
  - Cross-validation
  - challenges
- Replicability
  - code & data sharing
  - open source & open access
- Bagging
- Boosting
- Stacking
- Support Vector Machines
- Neural Nets

10 Deep Learning & AI: early 2010s-now

- use of GPUs -> computation performance of matrix multiplication
- frictionless reproducibility
- neural nets
- Transformers: GPT

ML Lecture Notes: Wk 2 — Training, Divergence, Loss & Optimisation

11 Fundamental: Loss Function

$$\hat{\theta} = \arg\min_{\theta} \mathcal{L}(\theta)$$

Where:

- $\hat{\theta}$ = a point estimate
- $\mathcal{L}(\%)$ = Loss (distance)
- min = optimization

12 Maximum Likelihood Estimation (MLE)

MLE is just a method just chooses a particular loss function (the Negative Log Likelihood: NLL)

$$\hat{\theta}_{MLE} = \arg\max_{\theta} p(\mathcal{D}|\theta)$$

= seeks to find the particular model as close as possible to the data.

- Modeling is just empirical risk minimization.
- We define that risk using the loss function.
- We can optimise to minimise a bunch of different aspects.
- MLE is one modeling option which takes the distance between the data and the model as the loss function
- think of 1) the KL divergence, or 2) the negative logged PDF into which you would plug your parameters and it would penalise you from diverging from (?).
- MLE's loss function is the distance from the data.

In MLE, the loss function is the Negative Log-Likelihood (NLL). While MLE focuses on maximizing the likelihood function, in optimization terms, we minimise, so we take the negative.

The objective in MLE is to maximize the likelihood function:

$$L(\theta; x_1, x_2, \ldots, x_n) = \prod_{i=1}^{n} p(x_i|\theta)$$

This can be equivalently framed as minimizing the negative of the log-likelihood function, which serves as the loss function:

$$\mathcal{L}(\theta) = -\ell(\theta; x_1, x_2, \ldots, x_n) = -\sum_{i=1}^{n} \log p(x_i|\theta)$$

The NLL provides a measure of how well the probabilistic model, parameterized by θ, fits the observed data. A lower NLL indicates a better fit because it corresponds to a higher likelihood of observing the given data under the model.

We must then think about an appropriate **probabilistic model,** p.

$p(D|\theta)$ is the **likelihood**

NB: the **MLE** $p(D|\theta)$ is **frequentist**, because it takes θ as fixed/given, with the data being the r.v.

cf.**Maximum a Posterior (MAP)** $p(\theta|D)$ by adding a prior, we are being , because we are taking the data as fixed, with the parameters as r.v.s

12.1 NLL: Optimizers: minimize loss functions

Optimizers typically (by convention) minimize things, so:

$$\hat{\theta}_{mle} = \arg\min_{\theta} \text{NLL}(\theta)$$

Where:

$$NLL(\theta) = -\sum_{i=1}^{n} \log p(y_i|x_i, \theta)$$

12.2 MLE Assumption 1) i.i.d.

- units don't interact with each other
- units have same assumed model
- *this is a strong and* ***substantively meaningful*** *assumption*

$$p(\mathcal{D}|\theta) = \prod_{i=1}^{n} p(y_i|x_i, \theta)$$

Products are messy, so we work with log-sums:

$$\log p(\mathcal{D}|\theta) = \sum_{i=1}^{n} \log p(y_i|x_i, \theta)$$

12.3 MLE Assumption 2) a model for p = normally distributed

12.3.1 Modeling y_i with Normal Distribution

Suppose

- $y_i \sim \mathcal{N}(\mu, \sigma)$, then $\theta = (\mu, \sigma)$
- $y_i \sim \mathcal{N}(\mu^*, \sigma)$ and $\mu^* = x_i\beta$, then $\theta = (\beta, \sigma)$

Simple Normal Model: If y_i is assumed to be normally distributed with mean μ and standard deviation σ, denoted as $y_i \sim \mathcal{N}(\mu, \sigma)$, the parameters of interest are $\theta = (\mu, \sigma)$. This model assumes that all observations come from a normal distribution with the same mean and variance.

Normal Model with Linear Predictor: If y_i is normally distributed with mean μ_i and standard deviation σ, where $\mu_i = x_i\beta$, denoted as $y_i \sim \mathcal{N}(\mu_i, \sigma)$, then the model parameters are $\theta = (\beta, \sigma)$. This introduces a linear relationship between predictors x_i and the mean of the response variable, **making it a linear regression model**.

Linear Regression = Normal Model with Linear Predictor The response variable y_i is assumed to be normally distributed with a mean (μ_i) that is a linear function of predictors (x_i) and a constant standard deviation (σ). **This forms the basis of linear regression models, where we assume normally distributed errors**.

1. **Normal Distribution Assumption**: Each observation y_i is assumed to come from a normal distribution, which is a common assumption in regression analysis.

2. **Linear Relationship**: The mean of the normal distribution for each y_i, denoted as μ_i, is not a fixed value but is instead determined by a linear combination of predictors x_i and their corresponding coefficients β.

3. **Model Parameters** θ: In this context, the model parameters are $\theta = (\beta, \sigma)$. The vector β contains the coefficients that describe how the mean response y_i changes with the predictors x_i, and σ is the common standard deviation of the response variable across all observations, assuming homoscedasticity (constant variance).

4. **Implications**:
- **Linear Regression**: This model is a form of linear regression because it models the mean of the response variable as a linear function of the predictors.
- **Parameter Estimation**: Estimating the model parameters involves finding the values of β and σ that best fit the observed data. This is typically done using Maximum Likelihood Estimation (MLE) or, equivalently for linear models, Ordinary Least Squares (OLS) when focusing solely on the β coefficients.
- **Statistical Inference**: With the estimated parameters, one can conduct hypothesis tests to assess the significance of predictors, construct confidence intervals, and make predictions.

12.3.2 Probability of Observing y_i

The probability of observing y_i from one of these models is expressed through the probability density function (PDF) of the normal distribution. For a given observation y_i, the PDF is:

$$p(y_i \; x_i, \theta) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{y_i - \mu_i}{\sigma}\right)^2\right)$$

Here, μ_i can either be a constant μ or $x_i\beta$ depending on the model.

But, we prefer this as the NLL (i.e. the negative, logged PDF):

$$\begin{aligned} NLL(\theta) &= -\log p(y_i|x_i, \theta) \\ &= \frac{1}{2}\log(2\pi\sigma) + \frac{1}{2}\frac{(y_i - \mu_i)^2}{\sigma^2} \end{aligned}$$

The goal in MLE is to find the parameter values (in this case of linear regression, the coefficients in $\mu_i = x_i\beta$ and the standard deviation σ) that minimize the sum of these negative log-likelihoods across all observations.

- This process effectively fits the linear regression model to the data by balancing the model's fit (how close the predicted means are to the actual data points) and the model's complexity (reflected in part by σ).
- Interpretation of the Terms: The first term is constant for a given σ and affects all observations equally, acting as a scaling factor.
- The second term is crucial for model fitting, as it penalizes deviations of the observed values from their expected means, adjusted for the scale of the data (as represented by σ).
- The balance between these terms ensures that the estimated model accurately captures the central tendency and variability of the data.

12.4 KL Divergence

Maybe we just want to choose a model that is maximally "close" to the data.
Let's define "close" as Kullback-Leibler (KL) divergence:

$$\begin{aligned} D_{KL}(p \parallel q) &= \sum_y p(y) \log \frac{p(y)}{q(y)} \\ &= \sum_y p(y) \log p(y) - \sum_y p(y) \log q(y) \end{aligned}$$

- Negative Entropy = green
- Cross Entropy = orange

There's some stuff in the slides I skipped here.

Overall intuition: the cross-entropy (which measures divergence of the model from the data) is mathemtically the same as the NLLif you set $p(y)$ (i.e. the data) to the uniform distribution -

which is to say you assume the data is ...? So minimising the NLL is the same as minimising KL divergence, which is trying ot get our model maximally close to the data.

I skipped next slide

13 MLE for Linear Regression

I DONT FULLY UNDERSTAND HOW MLE AND RSS CONNECT... WHY DID WE JUST DO ALL THAT MLE WORK, IF WE CAN JUST TAKE THE RSS GRADIENT???

Maximum Likelihood Estimation (MLE) for linear regression and the Residual Sum of Squares (RSS) are both methods used to estimate the parameters of a linear regression model, but they approach the problem from different statistical perspectives.

MLE for Linear Regression
In MLE, we assume that the residuals (differences between observed values and values predicted by the model) follow a normal distribution with a mean of zero and some variance σ^2. The likelihood function for linear regression under the normal distribution assumption is a function of the parameters of the regression line (usually denoted as β) and the variance of the error term σ^2.

The goal of MLE is to find the parameter values that maximize the likelihood of observing the data given the model. In the case of linear regression with normally distributed errors, this is equivalent to minimizing the negative log-likelihood, which, due to the properties of the logarithm, turns out to be equivalent to minimizing the sum of squared residuals.

Residual Sum of Squares (RSS) RSS is a measure of the discrepancy between the data and an estimation model. It's the sum of the squares of the residuals:

$$RSS = \sum_i (y_i - x_i\beta)^2$$

Mean Squared Error (MSE) MSE is simply the average of the RSS:

$$MSE = \frac{1}{n}\sum_i (y_i - x_i\beta)^2$$

Where n is the number of observations, y_i is the observed value, x_i is the feature vector for the i-th observation, and β represents the regression coefficients.

Maximum Likelihood Estimation (MLE) and Residual Sum of Squares (RSS) are related concepts in regression analysis, but they are not entirely separate methods. They are connected through the assumption about the distribution of errors in the regression model.

In ordinary linear regression, we assume that the response variable Y is linearly related to the predictors X with an added error term ϵ that follows a normal distribution with mean zero and some variance σ^2, often expressed as $\epsilon \sim N(0, \sigma^2)$.

RSS (Residual Sum of Squares):

- RSS is a measure of the model's fit to the data. It is calculated by summing the squares of the differences between the observed responses y_i and the responses $\hat{y}_i$ predicted by the linear model.
- The method of least squares finds the coefficients β that minimize the RSS.

MLE (Maximum Likelihood Estimation):

- MLE is a method used to estimate the parameters of a statistical model. In the context of linear regression, MLE seeks to find the coefficients β that maximize the likelihood of observing the sample data, given a specific set of parameters.
- Under the assumption of normally distributed errors, maximizing the likelihood is equivalent to minimizing the RSS. This is because the likelihood function for a normal distribution is a function of the squared differences between observed and predicted values (which is RSS), scaled by the variance of the errors.

Relationship:

- In the specific case of linear regression with normal errors, minimizing RSS is equivalent to performing MLE because the likelihood function for a normal distribution is proportional to the exponential of the negative RSS.
- The difference is conceptual: RSS is a direct measure of the fit of the model to the observed data, while MLE is a probabilistic approach that assumes a specific distribution of the error terms.
- In practice, for linear regression under the assumption of normal errors, both methods will yield the same estimates of the coefficients β.

Therefore, while MLE and RSS come from different theoretical foundations—MLE from probability theory and RSS from geometric considerations—they converge to the same solution in the context of ordinary linear regression with normally distributed errors.

From before:

$$\begin{aligned} NLL(\theta) &= -\log p(y_i|x_i;\theta) \\ &= \frac{n}{2}\log(2\pi\sigma) + \frac{n}{2\sigma^2}(y_i - x_i\beta)^2 \end{aligned}$$

Now we compare to the Residual Sum of Squares (RSS)

$$\begin{aligned}\text{RSS}(\beta) &= \frac{1}{2}\left(\sum_{i=1}^{n}(y_i - x_i\beta)^2\right) \\ &= \frac{1}{2}\|X\beta - y\|_2^2 \\ &= \frac{1}{2}\left((X\beta - y)^T(X\beta - y)\right)\end{aligned}$$

This also gives us the Mean square error (MSE):

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - x_i\beta)^2$$

Where is σ^2 go?

Where Did σ^2 Go?
In the context of MLE for linear regression, the σ^2 term is a part of the likelihood function when modeling the residuals as a normal distribution. However, when we minimize the negative log-likelihood to perform MLE, the σ^2 term becomes a scaling factor that does not affect the estimation of β since it is constant with respect to β. As a result, when we derive the MLE solution, we are left with an optimization problem that looks identical to minimizing RSS or MSE, where the σ^2 term is not present.

Analytic Solution
For linear regression, we can find an analytic solution to minimize the RSS or MSE, which will also give us the MLE of β under the assumption of normally distributed residuals. This is done by taking the derivative of the RSS with respect to β, setting it equal to zero, and solving for β. This process yields the normal equations, which can be solved to find the best-fitting linear regression coefficients:

$$\frac{\partial}{\partial \beta}\text{RSS} = \frac{\partial}{\partial \beta}\sum(y_i - x_i\beta)^2 = 0$$

Solving the normal equations:

$$X^T X\beta = X^T y$$

Where X is the matrix of input features (with each row corresponding to an observation and each column to a feature), and y is the vector of observed values. The solution to these equations gives us the least squares estimates for the coefficients β:

$$\beta = (X^T X)^{-1} X^T y$$

This is known as the Ordinary Least Squares (OLS) estimator. It is called "ordinary" to distinguish it from other variations that might put constraints or have different assumptions on the coefficients.

Conclusion
In summary, while MLE for linear regression involves maximizing the likelihood of the observed data given the model parameters, assuming normally distributed errors, minimizing the RSS or MSE is a specific application of this method when the errors are normally distributed with a constant variance σ^2. The analytic solution for the OLS estimator, which is derived from setting the derivative of the RSS with respect to β to zero, does not involve σ^2 because it is a scale factor that does not influence the estimation of β. The solution obtained is the set of coefficients that minimize the average squared difference between the observed and predicted values.

How do we find an analytic solution?
Take the derivative, set it equal to 0.

13.0.1 Analytic Solution

1) Find the derivative of the RSS:

$$\nabla_\beta \text{RSS} = \frac{1}{2}\nabla_\beta(\mathbf{X}\beta - \mathbf{y})^T(\mathbf{X}\beta - \mathbf{y})$$

2) Expand it out:

$$= \frac{1}{2}\nabla_\beta(\mathbf{X}\beta)^T\mathbf{X}\beta - (\mathbf{X}\beta)^T\mathbf{y} - \mathbf{y}^T\mathbf{X}\beta + \mathbf{y}^T\mathbf{y}$$

3) Rewrite:

$$= \frac{1}{2}\nabla_\beta \beta^T \mathbf{X}^T\mathbf{X}\beta - \beta^T\mathbf{X}^T\mathbf{y} - \mathbf{y}^T\mathbf{X}\beta + \mathbf{y}^T\mathbf{y}$$

4) Derive:

$$\begin{aligned}
&= \frac{1}{2}\mathbf{X}^T\mathbf{X} + \mathbf{X}^T\mathbf{X}\beta - \mathbf{X}^T\mathbf{y} - \mathbf{X}^T\mathbf{y} \\
&= \frac{1}{2}2\mathbf{X}^T\mathbf{X}\beta - 2\mathbf{X}^T\mathbf{y} \\
&= \mathbf{X}^T\mathbf{X}\beta - \mathbf{X}^T\mathbf{y}
\end{aligned}$$

So we just showed that: $\nabla_\beta = \mathbf{X}^T\mathbf{X}\beta - \mathbf{X}^T\mathbf{y}$

Now, we can find an optimum by setting to:

$$\hat{\beta} = \beta : \nabla_\beta \text{RSS} = 0$$

Which would mean:

$$X^T X\beta = X^T y$$

Or

$$\beta = (X^T X)^{-1} X^T y$$

So the RSS-derived optimisation problem for linear regression is: $\beta = (X^T X)^{-1} X^T y$

13.1 What does this give us?

13.1.1 Prediction

$$\hat{y} = X\hat{\beta}$$

13.1.2 Slope wrt each feature

$$\frac{X\hat{\beta} - X'\hat{\beta}}{X - X'} = \hat{\beta} \approx \frac{dy}{dX}$$

13.2 $Var(\hat{\beta})$?

The calculation of this variance is fundamental in statistical inference, as it allows us to **assess the precision** of the estimated coefficients and to **construct confidence intervals and hypothesis tests**.

First, rewrite $\hat{\beta}$ on the assumed linear model:

$$\begin{aligned}
\hat{\beta} &= (X^T X)^{-1} X^T y \\
&= (X^T X)^{-1} X^T (X\beta + \epsilon) \\
&= \beta + (X^T X)^{-1} X^T \epsilon
\end{aligned}$$

Next, write the variance and use this identity:

$$\text{Var}(\hat{\beta}) = \mathbb{E}[(\hat{\beta} - \beta)(\hat{\beta} - \beta)^T]$$

Substituting the expression for $\hat{\beta}$, and then we get:

$$\begin{aligned}\text{Var}(\hat{\beta}) &= \mathbb{E}[((X^TX)^{-1}X^T(X\beta + \epsilon) - \beta)((X^TX)^{-1}X^T(X\beta + \epsilon) - \\ &= \mathbb{E}[((X^TX)^{-1}X^T\epsilon - \beta)((X^TX)^{-1}X^T\epsilon - \beta)^T] \\ &= \mathbb{E}[((X^TX)^{-1}X^T\epsilon\epsilon^TX(X^TX)^{-1}] \\ &= (X^TX)^{-1}X^TE[\epsilon\epsilon^T]X(X^TX)^{-1}\end{aligned}$$

This provides the **'sandwich' form of the variance of** $\hat{\beta}$:

$$\text{Var}(\hat{\beta}) = (X^TX)^{-1}X^T)E[\epsilon\epsilon^T]X(X^TX)^{-1}$$

NB: this sandwich estimator is also known as the **robust standard error estimator**

Under Homoskedasticity assumption:

$\sigma^2 = \frac{1}{n}RSS(\beta) = MSE(\beta)$:

Error terms are normally distributed:

$\rightarrow$ ϵ has a mean of 0 and variance $\sigma^2\mathbf{I}$

$\rightarrow$ $\text{E}[\epsilon\epsilon'] = \sigma^2\mathbf{I}$

$\rightarrow$ expected value (of $Var\hat{\beta}$) simplifies to:

$$\text{Var}(\hat{\beta}) = (X^TX)^{-1}X^T\sigma^2\mathbf{I}X(X^TX)^{-1}$$

Giving us the **Homoskedastic "sandwich" estimator of the variance of** $\hat{\beta}$**:**

$$Var(\hat{\beta}) = \sigma^2(X^TX)^{-1}$$

This is often referred to as the BREAD of the "sandwich" in econometric parlance, with the MEAT being the middle term $\sigma^2\mathbf{I}$ in more complex models that account for heteroskedasticity or other forms of non-constant variance. In the standard OLS setting, the MEAT simplifies to $\sigma^2\mathbf{I}$, so the variance of the coefficient estimates depends on the inverse of the matrix $(X^TX)^{-1}$ (the BREAD) and the variance of the errors σ^2.

Under Heteroskedastic Assumptions: We can get unbiased estimates as $\hat{\epsilon}^2 = (y - \hat{y})^2$.

We assume: $E[\epsilon] = E[\text{diag}(\hat{\epsilon}^2)]$
Giving us:

$$\text{Var}(\hat{\beta}) = (X^TX)^{-1}X^T\text{diag}(\hat{e}^2)X(X^TX)^{-1}$$

This gives us the variance-covariance matrix, which we then take the diagonals of to get the variance.

The significance of the variance of the estimated coefficients $\hat{\beta}$:

1. **Statistical Significance**: The variance of $\hat{\beta}$ allows us to test hypotheses about the regression coefficients. Specifically, it enables us to determine if the estimated coefficients are significantly different from zero (or any other value), which helps to conclude whether there is a statistically significant relationship between the predictor variables and the response variable.
2. **Confidence Intervals**: Using the variance of $\hat{\beta}$, we can construct confidence intervals for the regression coefficients. These intervals provide a range of values within which we expect the true coefficient value to lie with a certain level of confidence (e.g., 95
3. **Precision of Estimates**: The variance provides a measure of the precision of the estimated coefficients. A smaller variance indicates that the estimator is more precise, meaning there is less uncertainty around the estimate of the coefficient.
4. **Model Diagnostics**: The variance can be used to assess the goodness of fit of the model. If the variance of the error term ϵ is high, it may indicate that the model does not fit the data well, or there are omitted variables that are influential in explaining the variation in the response variable.
5. **Influence of Data**: The matrix $(X'X)^{-1}$ in the variance expression reflects how the design matrix X influences the precision of the estimated coefficients. The spread and collinearity of the data points in the predictor variables affect $X'X$, and thus the variance of the estimates.

13.2.1 Now, how do we use that?

TL;DR;
Homoskedasticity –> can estimate coefficient variance using $\sigma^2(X'X)^{-1}$ **(take diagonals of resultant matrix)**
Heteroskedasticity –> can estimate coefficient variance using $(X^TX)^{-1}X^T\text{diag}(\hat{e}^2)X(X^TX)^{-1}$ **(take diagonals of the resultant variance-covariance matrix**

- we need more assumptions on the 'meat'
- if assume σ^2 is constant (= homoskedastic)
  - then $\sigma^2 = \frac{1}{n}RSS(\beta) = MSE(\beta)$
  - we can then assume $E[\epsilon\epsilon'] = \sigma^2 I$ - this is due to the normality assumption: Under normality, errors are expected to have a mean of zero and a constant variance σ^2, and the covariance between any two error terms is zero, which is represented by the identity matrix I scaled by σ^2
  - we can then substitute this term into the 'sandwich' formula for the variance of $\hat{\beta}$, to get $\text{Var}(\hat{\beta}) = \sigma^2(X'X)^{-1}$
  - Thus $E[\epsilon\epsilon'] = \sigma^2 I$ allows for simplifying the estimation of the variance-covariance matrix of the estimated coefficients ($\hat{\beta}$). This assumption allows for straightforward calculation of standard errors, confidence intervals, and hypothesis tests using the formula: $\text{Var}(\hat{\beta}) = \sigma^2(X'X)^{-1}$
  - TL;DR: Homoskedasticity –> can estimate coefficient variance using $\sigma^2(X'X)^{-1}$
- if we assume σ^2 varies (=heteroskedastic)
  - We can get unbiased estimates.

- We retain our assumption of independence.
- We assume: $\mathbb{E}[\varepsilon\varepsilon'] = \mathbb{E}[\text{diag}(e)]$. This means we can estimate it!
- Substitute into the sandwich: $(X^TX)^{-1}X^T\text{diag}(\hat{e}^2)X(X^TX)^{-1}$
- Called "robust standard errors". In particular "HC0". (there are others)
- 1) Calculate the residuals for each observation: as $e = y - \hat{y}$
- 2) Calculate the Estimator for the Variance-Covariance Matrix using sandwich estimator" $\hat{V}(\hat{B}) = (X^TX)^{-1}X^T\text{diag}(\hat{e}^2)X(X^TX)^{-1}$
- 3) **Compute Robust SEs: square roots of the diagonal elements of variance-covariance matrix** $\hat{V}(\hat{B})$
- TL;DR: Heteroskedasticity –> can estimate coefficient variance using $X'XX\text{diag}(e^2)X'XX$

NB Q 4 & 5 OF THE PREOBLEM SHEET 1 UNPACK THIS WELL

1. **Robustness to Heteroskedasticity**: It allows for valid standard errors, confidence intervals, and hypothesis tests for the coefficients even in the presence of heteroskedasticity. This means that the error variances ϵ^2 can vary with the level of the independent variables, and the estimator will still provide consistent estimates of the variance of $\hat{\beta}$.
2. **Individual Variance Estimates**: The diagonal of this variance-covariance matrix gives the variance of each estimated coefficient $\hat{\beta}$.
3. **Covariance Estimates** While the diagonal elements give the variances of individual coefficients, the off-diagonal elements provide the covariances between pairs of coefficients. This information is useful for understanding how the uncertainty of one coefficient estimate might be related to the uncertainty of another.
Crucial for making reliable inferences about the importance and impact of each feature in your regression model, especially in real-world scenarios where the assumption of homoskedastic errors (constant variance across observations) often does not hold.

Homoskedasticity Assumption
where errors are assumed to have a constant variance across observations:

$$\text{Var}(\hat{\beta}) = \sigma^2(X^TX)^{-1}$$

From this formula, $(X^TX)^{-1}$ is computed, and σ^2 is often estimated from the data. **The diagonal elements of the resulting matrix** provide the variances of each estimated coefficient, giving one value per feature.

Heteroskedasticity Consideration
adjusts for the possibility that the error variances might not be constant:

$$\text{Var}(\hat{\beta}) = (X^TX)^{-1}X^T\text{diag}(\epsilon^2)X(X^TX)^{-1}$$

- $\text{diag}(\epsilon^2)$ represents a **diagonal matrix with elements corresponding to the squared residuals from the model**, allowing for varying error variances.
With both estimators, the diagonals of the resulting matrices provides the variance for each coefficient estimates.

14 Bayes Rule

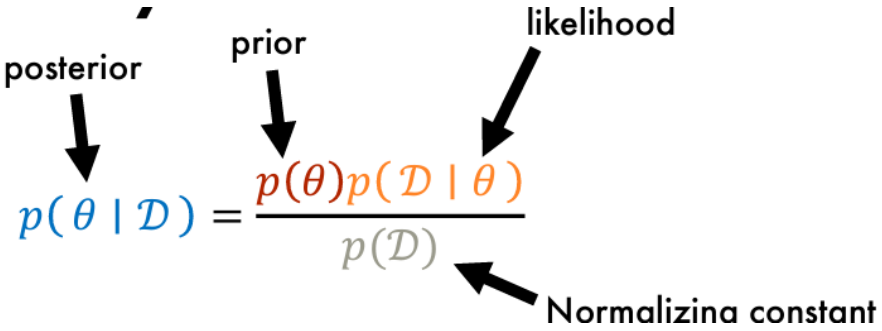


Figure 1: Bayes Rule

By incorporating a prior, we can transitioning from MLE to Maximum A Posteriori (MAP) estimation:

- Likelihood = MLE (from Frequentist) - prob of data given parameters.
- Prior = our beliefs about the parameters before observing any data
- (normalising constant = the marginal prob of the data; ensures the posterior probabilities sum to 1)
- combination of MLE with a prior leads to MAP estimation.
- MAP estimation aims to find the parameter values that maximize the posterior distribution, $p(\theta|D)$, which represents the probability of the parameters given the observed data

Since $p(D)$ is constant with respect to the parameters θ, it does not affect the argmax operation when we seek to maximize the posterior.

Therefore, the MAP estimation can be simplified to finding the parameters that maximize the numerator of the posterior distribution:

$$\text{MAP} = \arg\max_{\theta} p(\theta|D) = \arg\max_{\theta} \left(p(D|\theta) \cdot p(\theta)\right)$$

For computational ease: logarithm of the posterior:

$$\arg\max_{\theta} \log p(\theta|D) = \arg\max_{\theta} \left(\log p(D|\theta) + \log p(\theta)\right)$$

= maximizing the sum of the log-likelihood ($\log p(D|\theta)$) and the log-prior ($\log p(\theta)$).

This approach effectively balances fitting the model to the data (through the likelihood) with maintaining consistency with prior beliefs (through the prior).

NB, if prior is a constant, the MLE = MAP

14.1 A Moment on Probability

Objective Probability:

- flipping a coin

- based on facts about the properties of the world
- frequentist

Subjective Probability:

- making fair bets
- based only on our beliefs about the world
- bayesian

14.2 What can we do with $Var(\hat{\beta})$?

14.2.1 Bayesian

- Assume that β are random, but data isn't.
- Make statements like:
  - There is a 90% chance $\beta \in [-1, 1]$.
  - There is only a 5% chance $|\beta| < 1$.
- Is based on an assumed model of the world.

So:

- Construct a credible interval $p(\beta) = N(\beta, \text{Var}(\beta))$ so $p(\beta \in [-1, 1]) = 1 - \alpha$ is a meaningful statement.

14.3 Frequentist

- Assume that β are fixed, but data is random.
- Make statements like:
  - If we were to "rerun this experiment", 90% of the time, $\beta \in [-1, 1]$.
    - or, 90% of our data is between these point
  - If we were to "rerun this experiment", $|\hat{\beta}| < 1$ less than 5% of the time.
- Can be based (almost) entirely on specifics of study design (sometimes).

So:

- Construct a confidence interval.
- We do not assume a distribution of β.
- Instead, we think about a process for constructing an interval, $[l, u]$.
- Usually, $[l, u] = \beta \pm k\sqrt{\text{Var}(\hat{\beta})}$ where $k \approx z_{\frac{\alpha}{2}}$ when $\alpha = 0.05$, $k \approx 1.96$.
- We can then say that, if we constructed that right, $p(\beta \in [l_{\text{cb}}, u_{\text{cb}}]) \approx 0.95$.

I skipped slide 22 - where he uses the non-informative prior to make the MAP the same as the MLE

15 Prediction

15.1 Empirical Risk Minimization

MLE is just minimisation of a loss function, but we can change that loss

- Generic Loss function: $\mathcal{L}(\theta) = \ell(y, \theta; x)$
- MLE: $NLL(\theta) = -\sum_{i=1}^{n} \log p(y_i|x_i, \theta)$
- Mean Squared Error: $\mathcal{L}(\theta; \lambda) = \frac{1}{n}\sum_{i}^{n}(y_i - x_i\beta)^2$
- 1-0 Misclassification rate: $\ell(y, \theta; x) = \begin{cases} 0 & \text{if } y = f(x; \theta) \\ 1 & \text{if } y \neq f(x; \theta) \end{cases}$ so that: $\mathcal{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n} \ell(y, \theta; x)$

But issue with 1-0 loss function, is it is undifferentiable -> Surrogate loss functions (requirements: 1) upper bound of 0-1 loss; 2)v.close to the 0-1 loss)

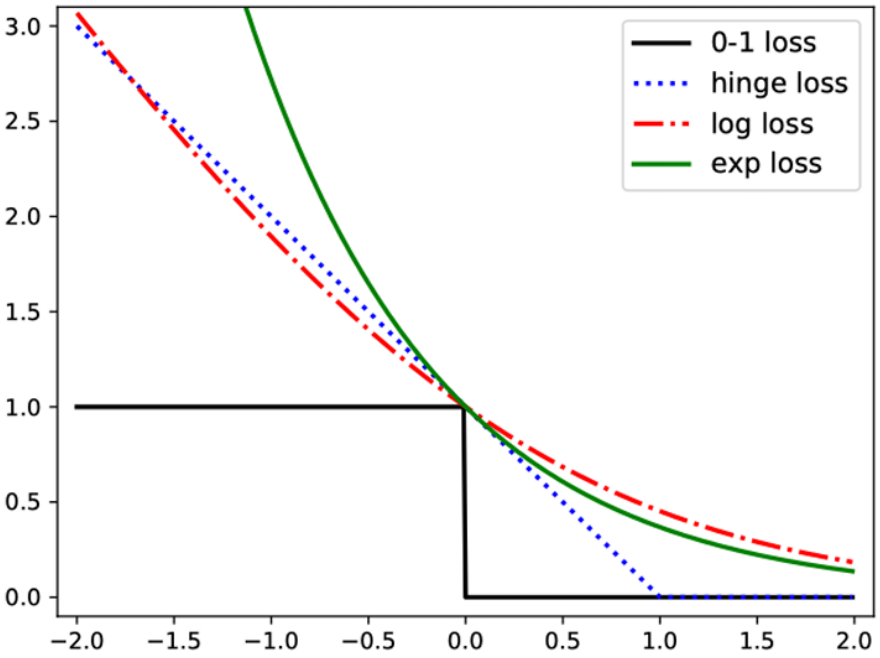


Figure 2: Enter Caption

15.2 Confusion Matrix

How much do we care about different kinds of errors?

- Type 1: False positive
- Type 2: False negative

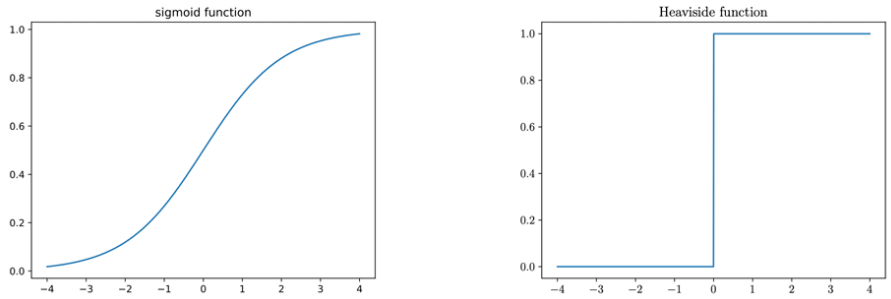


Figure 4: Enter Caption

		True	
		0	1
Predicted	0	✓	Type II
	1	Type I	✓

Figure 3: Enter Caption

16 Logistic Regression

= the classification version of linear regression

$$p(y_i|x_i, \theta) = Bern(y_i|\sigma(x_i\beta))$$

$$\begin{aligned} p(y = 1|x_1, \theta) &= \mu_i \\ &= \sigma_x\beta \\ &= \frac{1}{1 + e^{-x_i\beta}} \end{aligned}$$

16.1 Training Logistic Regression

Determine the NLL -> the avg binary cross-entropy / 'log loss':

$$NLL_\beta = -\frac{1}{n} \log \prod_{i=1}^{n} \text{Ber}(y_i|\mu_i)$$

Simplify:

$$NLL_\beta = -\frac{1}{n}\sum_{i=1}^{n} \log(\mu_i^{y_i} \times (1-\mu_i)^{1-y_i})$$

$$NLL_\beta = -\frac{1}{n}\sum_{i=1}^{n} y_i \log(\mu_i) + (1-y)\log(1-\mu_i)$$

$$NLL_\beta = -\frac{1}{n}\sum_{i=1}^{n} y_i \log \mu + (1-y_i)\log(1-\mu_i)$$

Find the gradient:

$$\nabla_\beta NLL(\beta) = -\nabla_\beta \frac{1}{n}\sum_{i=1}^{n} y_i \log \mu + (1-y_i)\log(1-\mu_i)$$

Simplifies to:

$$\nabla_\beta NLL(\beta) = \frac{1}{n}\sum_{i=1}^{n} (\mu_i - y_i)x_i$$

The problem is that μ_i is funky, so there's no easy closed form...

17 Linear Classification

Decision boundary = where your classification changes

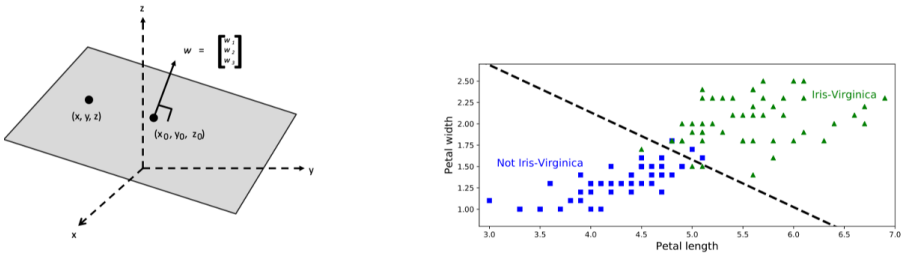


Figure 5: Enter Caption

18 Bias/Variance tradeoff

Bias:

$$bias(\hat{\theta}) = E[\hat{\theta}] - \theta$$

Central decomposition: MSE = Bias + Variance:

$$MSE(\hat{\theta}) = E[(\hat{\theta} - \theta)^2]$$
$$\dots$$
$$= Var(\hat{\theta}) + bias(\theta)^2$$

Significance: if our use of an unbiased estimator leads to too much variance, we might prefer to use a biased one.

18.1 Example

E,g, using a Bayesian prior to introducing some bias, reduce variance.

This technique is useful in scenarios where the **sample size is small**, or there is substantial **uncertainty about the sample mean**, and we wish to incorporate external information of beliefs into our estimation process.

In the below Bayesian approach, the adjusted estimator $\bar{y}(k)$ represents a compromise between the data's evidence (the sample mean) and the prior belief (the mean is near zero).

The parameter k controls this compromise: large k means more weight on prior belief, means greater bias towards zero, lower variance of the estimate.

- Suppose we have n i.i.d. copies from $y_i \sim N(1, \sigma^2)$
  we're dealing with a set of $\bar{y}$ independent and identically distributed (i.i.d.) samples from a normal distribution
- **Sample mean:**
  let $\bar{y} = \frac{1}{n}\sum_i y_i$
  this just denotes the sample mean definition
- **Variance of sample mean:**
  we know that $V(\bar{y}) = \frac{\sigma^2}{n}$
  this is just the definition of the variance of a sample mean
- **Now, we place Bayesian prior to adjust the estimator:** think mean will be near zero:
  - Adjusted estimator: $\widetilde{y} = \frac{n}{n+k}\bar{y}$
    scales the estimate of y
- **Impact on bias & variance of the Bayesian estimator:**
  - **Bias** of $\widetilde{y} = 1 - \frac{n}{n+k}$ *Increases (tends towads 0) as k increases*
    since 1 is the Expected Mean, subtract (what?)
  - **Variance** of $\widetilde{y} = (\frac{n}{n+k})\frac{\sigma^2}{n}$ *Decreases as k increases*
    scales the variance

The notation describes an approach to estimating the mean of this distribution, incorporating a prior belief about the mean being close to zero, and adjusting the estimate based on this prior.

Sample Mean ($\bar{y}$): The notation $\bar{y} = \frac{1}{n}\sum_i y_i$ denotes the sample mean.

Variance of $\bar{y}$ - i.e. ($Var(\bar{y})$): For samples drawn from a normal distribution, the variance of the sample mean $\bar{y}$ is $\frac{\sigma^2}{n}$. This follows from the properties of the normal distribution, where the variance of the sum (or average) of iid normal variables is the sum (or average) of their variances.

Prior Belief and Adjusted Estimator - i.e. ($\bar{y}(k)$): When you mention placing a prior that assumes the mean will be near zero, this introduces a Bayesian element into the estimation. The adjusted estimator $\bar{y}(k)$ seems to represent a shrinkage estimator that pulls the sample mean $\bar{y}$ towards zero (the prior belief) by a factor that depends on k, a parameter that quantifies the

strength of this belief or the weight of the prior relative to the data.

Bias of $\bar{y}(k)$: The formula $\text{bias}(\bar{y}) = 1 - \frac{n+k}{n}$ quantifies the bias introduced by adjusting the estimator towards zero. This shows that as k increases (placing more emphasis on the prior belief), the bias towards zero increases.

Variance of $\bar{y}(k)$: The formula $Var(\bar{y}) = \frac{n}{n+k}\frac{\sigma^2}{n}$ represents the variance of the adjusted estimator. It shows that the variance is reduced from $\frac{\sigma^2}{n}$ (the variance of the unadjusted sample mean) by a factor of $\frac{n}{n+k}$.

This reflects a trade-off: while introducing bias by incorporating the prior belief, the variance of the estimate is reduced.

ML Lecture Notes: Wk 3 — High-dimensional methods and Regularisation

19 Supervised Learning

- Given X with labels y: prediction
- Classification: learn $f(x) : \mathcal{X} \to 0, 1$
- Regression: learn $f(x) : \mathcal{X} \to \mathbb{R}$
- Performance based on some distance between predicted vs actual labels: $d(\hat{f}(x), y)$ - for example in OLS $(\hat{f}(x) - y)^2$

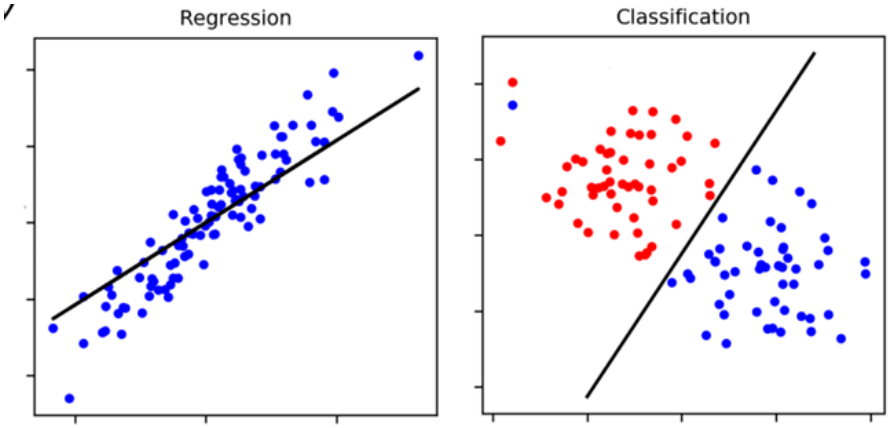


Figure 6: Regression vs classification in supervised learning

20 OLS

- optimal solution of OLS regression:

$$\hat{\beta} = (X^T X)^{-1} X^T y$$

- this gives us the optimal coefficients; we then plug these back (take matrix product) to get predictions:

$$\hat{y} = X\hat{\beta}$$

- in general, we assume first column of X is a column of 1s - the intercept. This gives us matrix $n \times (p + 1)$
- prediction result: $\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 X_1 + \cdots + \hat{\beta}_p X_p$

21 Polynomial regression

- Allows us to introduce non-linearity; *NB: the 'linear' in linear regression, refers to linearity in parameters, NOT the produced line*
- polynomial order = measure of complexity of the model = how well it can fit the training data
- we can encode non-linearity into OLS with feature expansion through polynomials
  - assume we have a single feature:
  - feature expansion: $f(X) = \beta_0 + \beta_1 X + \beta_2 X^2 + \beta_3 X^3 \cdots + \beta_M X^M$
- BUT this creates a new question: how to choose appropriate M

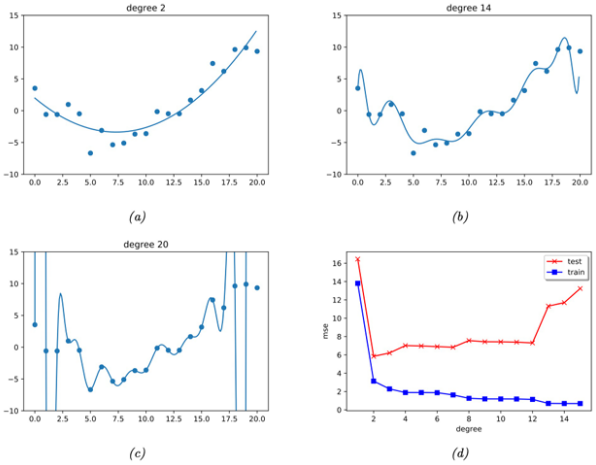


Figure 7: High degree polynomials: overfitting

21.1 Challenges with polynomial regression: Numerical Instability

Polynomials can represent a huge class of functions (highly flexible + mathematically straightforward): Taylor series / Weierstrass approx theorem (anything can be closely approx'd)

BUT, Polynomials are **global approximates** -> problems with **edges** and **variance**:

- Runge's Phenomenon. This phenomenon occurs when using high-degree polynomials to interpolate a set of points, leading to large oscillations at the edges of the interval. The polynomial fits the given points well in the middle of the domain but exhibits large and erratic fluctuations at the ends (or edges) of the domain.
- **High Variance at Edges**: The edges or extremities of the domain cause a lot of variance in the polynomial approximation. As you increase the degree of the polynomial to achieve a better fit through the dataset, the polynomial's behavior becomes increasingly erratic near the boundaries. This is because polynomials are global approximates, meaning changes to the polynomial to improve the fit in one part of the domain can have far-reaching effects throughout the entire domain, including unwanted oscillations at the edges.
- **Sensitivity to Data (Variance)**: High-degree polynomials are very sensitive to the data they are fitting. Small changes in the data points (input) can lead to significant changes in the polynomial (radically large differences in output estimations), particularly affecting its behavior at the domain's edges. This sensitivity is a direct consequence of the polynomial trying to accommodate all data points in a global manner, leading to a lack of robustness.

- = While polynomial functions are powerful tools for function approximation due to their global nature, they can be **problematic when it comes to accurately representing functions with sharp edges or rapid changes**. The global nature of polynomial approximation means that **trying to fit parts of the function with high curvature or discontinuities can lead to high variance and oscillatory behavior, especially at the domain's boundaries**. This issue necessitates careful consideration when choosing the degree of the polynomial for approximation tasks and often encourages the use of alternative approaches, such as piecewise polynomials or spline functions, which can provide local approximations that mitigate some of these challenges.

For polynomials of large degrees (x^k where k is large), two main issues contribute to **numerical instability**:

1. **Magnitude of Polynomial Terms**: As the degree of the polynomial increases, the magnitude of x^k (where k is the degree of the polynomial) grows rapidly for values of x greater than 1 or less than -1. This can lead to extremely large values that are difficult to manage computationally.
   i.e. when k gets large, x^k gets super large

2. **Magnitude of Coefficients**: To compensate for the large values produced by high-degree terms, the fitting process often results in very small (or very large, in magnitude) coefficients (β). These coefficients attempt to scale the polynomial terms back down to a reasonable size to fit the data accurately.
   i.e. as x^k gets super large, in order to smooth things out, β_k then has to shrink: to return to fit the line, we have to multiply our super large value by something small

=> **Numerical Instability**: The combination of very large polynomial terms and small coefficients can lead to numerical instability. This is because small changes in the data or in the coefficients can result in disproportionately large changes in the polynomial's value, making the polynomial approximation sensitive and unpredictable.

- **Condition Number = measure of Numerical Instability**: condition number of the matrix X^TX, where X is the matrix of polynomial basis functions. The condition number, $\kappa(X^TX) = \frac{\text{max eigenvalue}}{\text{min eigenvalue}}$, measures the sensitivity of the polynomial fit to errors in the data.

  - *A high condition number indicates that the polynomial fit is likely to be numerically unstable, as it is highly sensitive to small changes in the input data.*
  - *think of SVD: condition number is how hard it is to invert this matrix whether it is hard to inverse is based around the difference between the max and the min*

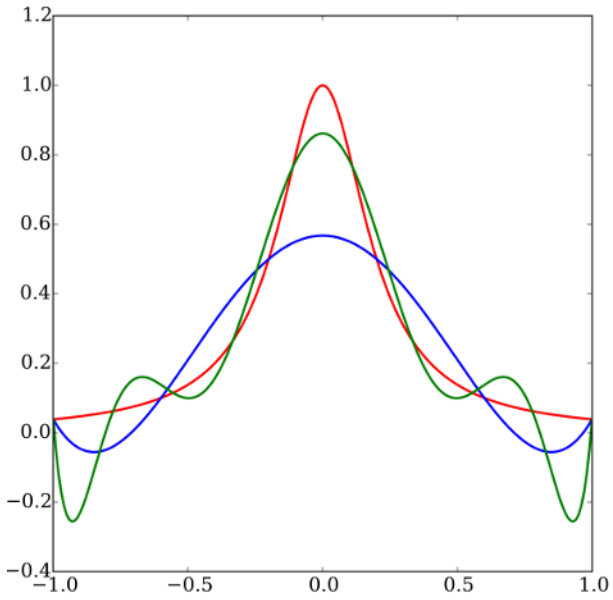


Figure 8: Numerical Instability

22 Bias-Variance tradeoff

$$bias(\hat{\theta}) = E[\hat{\theta}] - \theta$$
$$MSE(\hat{\theta}) = Var(\hat{\theta}) + bias(\theta)^2$$

- as complexity increases, the model becomes more expressive:
  - bias decreases: *the function can now 'express' the data better*
  - variance increases: *the function is harder to estimate*

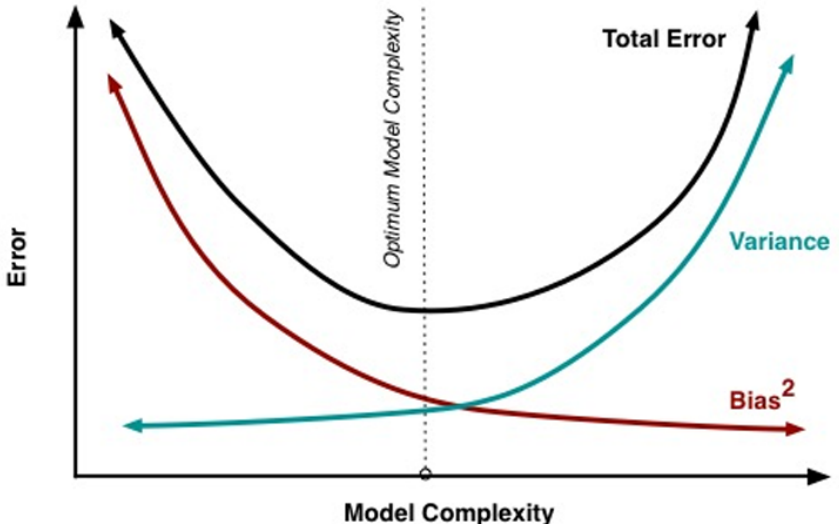


Figure 9: Bias Variance Trade off

23 Evaluating Model Performance

This process typically involves defining the criteria of interest, estimating population risk, and employing strategies like Empirical Risk Minimization (ERM) to approximate this risk based on available data.

23.1 Defining Criteria of Interest -> "Risk"

1. **Model**: how you predict $f(x)$ from X
2. **A measure of distance... or is this the loss function???** dependent on selected feature space: eg euclidean distance vs ..?
3. **Loss function**: quantifies the discrepancy or error between the actual outcome y and predicted outcome $f(x)$. Gives you errors / losses:

$$(d(f(x), y)$$

4. **Risk**: measure of expected loss of a model. The metric you minimise in ERM.

"Risk" is a generalized concept that refers to the **expected loss or error of a model with respect to its predictions on new data**; essentially an expected loss quantifying how much, on average, the model's predictions deviate from the actual values according to the chosen metric (i.e. the loss function).

- **Accuracy** - measures the proportion of all predictions that the model gets right. *Used when the costs of false positives and false negatives are roughly equivalent.*

...In classification problems, precision and recall are often more insightful than accuracy, especially in imbalanced datasets. **These metrics can serve as the "risk" by framing the loss in terms of incorrect predictions**: false positives for precision and false negatives for recall....

- **Precision** - (Positive Predictive Value) measures the proportion of positive identifications that were actually correct. *Used when the cost of false positives is high*
- **Recall** - (Sensitivity) measures the proportion of actual positives that were identified correctly. *Used when the cost of false negatives is high.*
- **Area Under the ROC Curve (AUC)** provides a single measure summarizing the performance of a classifier across all possible threshold values, reflecting the model's ability to discriminate between positive and negative classes.
  - A higher AUC indicates a better model.
  - In this context, risk could be considered as 1 - AUC, representing the model's inability to discriminate correctly.
- **Mean Squared Error (MSE)** - average squared difference between the observed actual outcomes and the outcomes predicted by the model. It is commonly used in regression problems. The MSE directly quantifies the risk as the expectation of squared errors.
- Other relevant metrics

23.2 Population Risk ($R^*_{f,p}$)

- Population risk is what we really care about, but we can't observe it directly!
- A theoretical measure of the expected loss or error of a model f over the entire population of interest.
- Often denoted as $E_{P^*}[\ell(y, f(x))]$...
- ... where $\ell(y, f(x))$ is a loss function measuring the discrepancy between the true outcomes y and the model predictions $f(x)$.
- = a gold standard for model performance: indicates how well model would perform in general beyond just the observed data.
- BUT not directly observable (we don't have access to entire population of data / all possible scenarios) -> have to estimate it.
- We sample from the full population, and then we have labels for those observations in the sample: ERM...

Population Risk:

$$R_{f,p^*} = R_f = \mathbb{E}_{p^*}[\ell(y, f(x))]$$

This notation represents the concept of the **population risk** or **expected risk** for a predictive model f. Let's break down the meaning and components of this expression:

1. R_{f,p^*} **or** R_f: This denotes the population risk associated with the predictive model f. The population risk is a theoretical measure of the average loss or error of the model f when predicting outcomes (y) from inputs (x) across the entire population distribution p^*.
2. $\mathbb{E}_{p^*}[\cdot]$: This is the expectation operator, indicating that the following expression is to be averaged over the probability distribution p^*, which represents the true underlying distribution of the data. The subscript p^* emphasizes that the expectation is with respect to the true population distribution of the input-output pairs (x, y).
3. $\ell(y, f(x))$: This is the loss function, which quantifies the discrepancy or error between the actual outcome y and the predicted outcome $f(x)$ for a given input x. The choice of loss function (ℓ) depends on the specific problem and goals of the model (e.g., mean squared error for regression tasks, or cross-entropy loss for classification tasks).
4. **Interpretation**: The expression as a whole, $\mathbb{E}_{p^*}[\ell(y, f(x))]$, represents the expected value of the loss $\ell(y, f(x))$ when the model f is used to predict y from x, averaged over all possible input-output pairs (x, y) according to their true distribution in the population (p^*). This expected loss is what the model aims to minimize; however, since the true distribution p^* is usually unknown, direct computation of R_{f,p^*} is not feasible in practice.

In summary, $R_{f,p^*} = \mathbb{E}_{p^*}[\ell(y, f(x))]$ encapsulates the goal of supervised learning: to find a model f that minimizes the average loss incurred when predicting outcomes from inputs, as averaged over the true, but unknown, distribution of the data. This concept underpins the rationale for using empirical risk minimization (ERM) as an approach to approximate and minimize the population risk based on a sample from the population.

23.3 Empirical Risk Minimization (ERM)

- ERM **approximates the (theoretical) Population Risk** using the available sample data
- **Empirical Risk = average loss over the sample set** (using a given loss function): $\frac{1}{n}\sum_{i=1}^{n}\ell(y_i, f(x_i))$
- ERM Process: aims to find the function (from a hypothesis space H), that minimises Empirical Risk $f^* = \arg\min_{f\in\mathcal{H}}\frac{1}{n}\sum_{i=1}^{n}\ell(y_i, f(x_i))$
- = minimising loss on the observed dataset (as proxy for minimising true population risk)
- The empirical risk is based on the empirical distribution of the sample data, which represents an approximation of the true underlying distribution of the population. By minimizing the loss over the empirical distribution, ERM seeks to approximate the best possible model performance on the population level.

Empirical Risk Minimisation:

$$\hat{f}_{ERM} = \arg\min_{f\in\mathcal{H}}\frac{1}{n}\sum_{i=1}^{n}\ell(y_i, f(x_i))$$

Here's what each part means:

- $\hat{f}_{ERM}$: the **predictive model or function** that we are trying to find. It's the function that, given input x, produces an output $f(x)$ which is the prediction of the true output y.
- $\arg\min_{f\in\mathcal{H}}$: the argument of the **minimization problem**. It means we are looking for the function f within a hypothesis space $\mathcal{H}$ that minimizes the following expression. The hypothesis space $\mathcal{H}$ is the set of all models or functions that we are considering as potential solutions.
- $\frac{1}{n}\sum_{i=1}^{n}\ell(y_i, f(x_i))$: the **empirical risk**, which is the average loss over the sample dataset consisting of n data points. Each data point consists of an input x_i and the corresponding true output y_i. The function $\ell(y, f(x))$ is a loss function that measures the discrepancy or error between the predicted output $f(x_i)$ and the true output y_i for each data point. The empirical risk is thus the average of these individual losses over all data points in the sample.
- **Empirical Distribution of Your Data**: The empirical risk is calculated based on the empirical distribution of the data, which is the distribution represented by the sample data points (x_i, y_i). Unlike the true underlying distribution p^* of the population, the empirical distribution is what we have access to through the collected data.

Overall Explanation: The process of Empirical Risk Minimization involves selecting the best predictive model f from a set of possible models ($\mathcal{H}$) based on how well they perform on the available data. Specifically, ERM seeks the model that minimizes the average loss (as measured by some loss function ℓ) incurred on the sample data. This approach is fundamental in machine learning for fitting models to data, with the ultimate goal of finding a model that not only performs well on the sample data but also generalizes well to new, unseen data.

NB!

BUT, if you a choose a model based on your ERM alone, you will ovefit to the training data and end up with a high order polynomial - which is problematic! We need to include other considerations...

23.4 Different Kinds of Error: Approximation vs Estimation

1. **Best possible function:**

$$f^{**} = \arg\min_f \mathcal{R}(f)$$

   - represents the theoretical best possible predictive model across all conceivable models; minimizes the true risk $R(f)$, which is the expected loss over the entire population distribution.
   - This function serves as a benchmark for the best performance that could be achieved in theory, regardless of any constraints or limitations.
   - this is an ideal function; maybe we can't estimate it with our chosen class - eg. it's more complex than a polynomial, maybe it has discontinuities etc
   - can never be observed

2. **Best function within a considered hypothesis space:**

$$f^* = \arg\min_{f\in\mathcal{H}} \mathcal{R}(f)$$

   - best function within a specific hypothesis class H (i.e. the set of all models we are considering)
   - **minimises true risk**$R(f)$
   - represents the best performance we can hope to achieve given the constraints of our model class or hypothesis space
   - we can't get to f^{**} if the actual function is not in the class of functions we are using

3. **Our empirical best guess:**

$$\begin{aligned} f_n^* &= \arg\min_{f\in\mathcal{H}} \mathcal{R}(f, \mathcal{D}) \\ &= \arg\min_{f\in\mathcal{H}} \frac{1}{n}\sum_{i=1}^{n} \ell(y_i, f(x_i) \end{aligned}$$

   - **minimises empirical risk** $\mathcal{R}(f, \mathcal{D})$
   - Empirical risk is the average loss computed over a specific dataset $\mathcal{D}$
   - Empirical best guess aims to approximate the true best function f^* within $\mathcal{H}$ by minimising the observed errors on the training data $\mathcal{D}$
   - this is our best estimate, based on the available data - trained on n samples - a subset of the full population

... this allows us to rethink what our error is...
Decomposing the difference in risk (expected loss) between (1) the theoretical best possible model (Rf^{**}) and (3) the best empirical model (Rf_n^*):

*NB we are dealing with the risks associated with each of these 3 model types; the E_{p^*} deals with the **empirical** risk of our best guess model, whereas the other two are associated with the (theoretical) **true** risk.*

$$R3 - R1 = (R2 - R1) + (R3 - R2)$$

$$\mathbb{E}_{p^*}\mathcal{R}(f_n^*) - \mathcal{R}(f^{**}) = \underbrace{\mathcal{R}(f^*) - \mathcal{R}(f^{**})}_{\text{Approximation Error}} + \underbrace{\mathbb{E}_{p^*}\mathcal{R}(f_n^*) - \mathcal{R}(f^*)}_{\text{Estimation/Generalization Error}}$$

Approximation error: how much worse our chosen model class is compared to the best possible model class. We can never eradicate this, just do a better job of selecting functions, but we will always pay some cost based on our modelling choice. *(Theory-based)*

- R(1) - R(2)
- Measures the gap between (1) the best possible function f^{**} and (2) the best function we consider f^*.
- Reflects the inherent limitations of the hypothesis space in approximating the true best model.
- Quantifies how much performance is lost simply because our model class cannot capture the true underlying relationship in the data perfectly.
- Quantifies how well the best theoretical model in our chosen hypothesis space can approximate the true best model.
- This error is inherent to the choice of the model class and does not decrease with more data.

Estimation / Generalization error: errors from estimating the model from finite data. We CAN do something about this; this is (1) sampling variation and (2) modelling problems *(Empirically-based)*

- R(3) - R(2)
- This error measures the difference between the expected risk of (2) the best model within the hypothesis space ($R(f^*)$), vs the risk of (3) the empirically best model $R(f_n^*)$ - which is the model obtained by training on n samples.
- Captures the error introduced by the process of estimating the model from a finite sample size n, rather than having access to the entire population.
- Refers to the model's performance on new, unseen data compared to the training data.

NB!

- The estimation error is influenced by (1) the size of the training data, and (2) the model's capacity;
  - Decreases as the sample size n increases.
  - Increases if the model complexity is too high relative to the amount of data (overfitting).

Generalization Error = total error of our best guess, including errors from both (1) model class limitations, and (2) from estimating the model from finite data.

Balancing Approximation error vs Generalization error:

- The total difference in risk between the theoretical best possible model and our empirically best model from a sample can be understood in terms of two fundamental challenges in machine learning: choosing the right model class (approximation error) and accurately estimating the best model within that class from limited data (estimation error).
- Minimizing the total error involves balancing these two sources of error. **Improving the model class to reduce approximation error might increase the complexity of the model, potentially increasing the estimation error if additional data is not available.**
- This equation highlights the trade-off in machine learning between the complexity of the hypothesis space (which can reduce approximation error but increase estimation error) and the amount of data needed to effectively estimate models within that space.

Similar to bias-variance trade off: could make $f*$ a huge class of functions, but this makes it more complex so we want to choose a model that balances the trade off between approximation vs estimation errors.

23.4.1 Estimating the Generalisation Error

Generalization error refers to the model's performance on new, unseen data compared to the training data on which it was learned. This can be **estimated using test data** - providing insight into the model's performance in real-world or unseen scenarios.

We take 2 samples of population:

- **Training data**: $p_{train}(x, y)$ - do ERM: minimize the loss (or error) on these data points, effectively learning the underlying pattern or relationship between features (X) and targets (y)
- **Testing data**: $p_{test}(x, y)$ - *again/also* do ERM: **comparing the ERM between the testing and the training (= Generalisation error)**. To evaluate the model's performance, specifically its ability to generalize the learned patterns to new, unseen data

By evaluating the model on the testing data, we can estimate the generalization error:

$$\underbrace{\mathbb{E}_{p^*}\mathcal{R}(f_n^*) - \mathcal{R}(f^*)}_{\text{Estimation/Generalisation Err}} \approx \underbrace{\mathbb{E}_{p_{\text{train}}}[\ell(y, f_n^*)]}_{\text{Training Loss}} - \underbrace{\mathbb{E}_{p_{\text{test}}}[\ell(y, f_n^*)]}_{\text{Test Loss}}$$

we are taking a new sample, and comparing:

- Training Loss: how well we thought we did - represents the average loss of the model on the training dataset. It measures how well the model fits the data it was trained on.
- Test Loss: how well we actually did - represents the average loss of the model on a separate test dataset. It measures the model's ability to generalize to new, unseen data.

NB!

Generalisation/Estimation error is a concept how overly-optimistic we were: the optimism of pure-ERM.

Expected risk is a measure of generalisation error. We can actually observe how overly optimistic we were.

NB!

Generalisation/Estimation error: quantifies the **error introduced by estimating the model from finite data**.

It represents **difference between the expected risk of the model trained on** n **samples** and the **best possible model within the hypothesis space**.

The ERM makes us overly optimistic, because we are overfitting to the data (will reduce the Training Loss to 0, but will increase the Test Loss

In fact, we need to consider how well it generalises - again it is always a balance (no free lunch).

Trade offs:

- Approximation loss vs Estimation loss
- ($>$ within that $>$) Training Loss vs Test Loss
- wait, actually these are measuring the smae thing (overfit?)

In the over fitted model: the Approximation Error is basically 0, but will give us high Estimation / Generalisation error (the difference between the training loss (towards 0) and the Test Loss (high!) will be much increased)

1. **Generalisation/Estimation error ($\mathbb{E}_{p^*}\mathcal{R}(f_n^*) - \mathcal{R}(f^*)$):**

   - represents the difference in expected risk (or performance) between the best empirical model f_n^* obtained from training on n samples and the best possible model f^* within the hypothesis space $\mathcal{H}$ across the true distribution p^* of the data.
   - Essentially, it quantifies how much worse our model f_n^*, trained on finite data, performs compared to the theoretically optimal model f^* that could be achieved with complete knowledge of the distribution p^*.

2. **Training Loss ($\mathbb{E}_{p_{train}}[\ell(y, f_n^*)]$):** This is the expected loss (average loss) of the model f_n^* on the training data. It represents how well the model fits the data it was trained on.

3. **Test Loss ($\mathbb{E}_{p_{test}}[\ell(y, f_n^*)]$):** This is the expected loss of the model f_n^* on a separate test dataset that was not used during training. This measures how well the model generalizes to new, unseen data.

The approximation essentially states that the difference in expected risk between the empirically best model and the optimal model can be estimated by the difference in the model's performance (in terms of loss) on the training set versus its performance on the test set.

Key Takeaways: - **Generalization Gap:** The difference between the model's performance on the training data and the test data = practical estimate of the generalization gap.

- small gap indicates that the model generalizes well
- large gap suggests that the model may be overfitting to the training data and not performing as well on unseen data.

Estimating Model Performance: Since we cannot directly observe the true expected risk over the entire data distribution (p^*), we use the training and test datasets to estimate how well our model is likely to perform in practice. This approach allows us to gauge the model's ability to generalize beyond the specific examples it has seen during training.

It's important to note that while this method can provide a good estimate of generalization error, it's not perfect. The testing data can only offer an estimate, and the model's true performance could vary in completely new contexts. Furthermore, factors like overfitting to the training data or biases in the data collection process can affect the accuracy of the generalization error estimate.

24 Regularisation: as viewed from multiple angles

24.1 Regularisation overview

24.1.1 Mechanics

Regularization prevents models from overfitting to the training data, thereby improving their generalization to unseen data. Overfitting occurs when a model **learns patterns specific to the training data, including noise**, to the extent that it performs poorly on new data. Regularization addresses this by introducing additional information or constraints into the model to discourage overly complex models without compromising the model's ability to learn from the

data.

Regularization works by adding a **penalty on the size of the model parameters** to the loss function that the model optimizes. There are two common types of regularization:

1. **L1 Regularization (Lasso Regression)**: Adds the sum of the absolute values of the coefficients to the loss function. This can lead to coefficients being exactly zero, thus performing feature selection.
2. **L2 Regularization (Ridge Regression)**: Adds the sum of the squares of the coefficients to the loss function. This tends to distribute the penalty among all coefficients, shrinking them but rarely making them exactly zero.

24.1.2 Uses

1. **Prevent Overfitting**: By penalizing large coefficients, regularization reduces the model's complexity, leading to lower variance and less overfitting.
2. **Improve Generalization**: A simpler model with smaller coefficients is less sensitive to the noise in the training data, making it better at predicting outcomes for unseen data.
3. **Feature Selection (L1 Regularization)**: By driving some coefficients to zero, L1 regularization can help in identifying the most important features of the data.

24.1.3 Comprehensive View

Regularization extends beyond L1 and L2 methods. For example, Elastic Net combines the penalties of L1 and L2 regularization, enjoying the benefits of both feature selection and coefficient shrinkage. Dropout, primarily used in deep learning, is another form of regularization where randomly selected neurons are ignored during training, preventing the network from becoming too dependent on any one neuron and thus reducing overfitting.

24.1.4 Formally

Where:

- $J(\theta)$ is regularized objective function
- $Loss(\theta)$ original loss function (eg MSE for regression)
- λ is regularisation strength parameter

L1 Regularisation (Lasso)

$$J(\theta) = \text{Loss}(\theta) + \lambda \sum_{j=1}^{n} |\theta_j|$$

$\sum_{j=1}^{n} |\theta_j|$ adds absolute values of the model coefficients, encouraging sparsity.

L2 Regularisation (Ridge)

$$J(\theta) = \text{Loss}(\theta) + \lambda \sum_{j=1}^{n} \theta_j^2$$

And when we manipulate this to get the minimisation objective function for beta:

$$\hat{\beta}_{ridge} = (X^T X + \lambda I_p)^{-1} X^T y$$

NB when Lambda = 0, it is just the same as the normal linear regression

Elastic Net:

$$J(\theta) = \text{Loss}(\theta) + \lambda_1 \sum_{j=1}^{n} |\theta_j| + \lambda_2 \sum_{j=1}^{n} \theta_j^2$$

where λ_1 and λ_2 are parameters that control impact of L1 and L2 terms respectively

24.2 Regularisation: as necessity

- OLS: $\hat{\beta} = (X^T X)^{-1} X^T y$
- **BUT, only calculable when $X^T X$ is inevitable**
  - non-zero determinant
  - full rank: each dimension / column of X is linearly independent (cannot predict a column of X based on a linear combination of another)
  - equivalent to an eigen value condition

  In cases where features are linearly dependent...(?)
- **$\rightarrow$ regularisation can guarantee $(X^T X)^{-1}$ to exist**
  - The easiest way to make sure that the inverse exists: to ensure the diagonal > off-diagonals
  - i.e.: if $\sum_j |A_{ij}| < A_{ij}$ for all i, then is A invertible
  - i.e. the **ridge** dominates the matrix.
  - This make $(X^T X)_{ii}$ bigger $\rightarrow$ eventually make it convertible.
  - $\hat{\beta}_{ridge} = (X^T X + \lambda I_p)^{-1} X^T y$ - here the scaling factor makes $(X^T X)_{ii}$ term bigger.

24.3 Regularisation: as optimisation

- We can also express this directly as a restriction on our coefficients
- **Standard regularisation** adds a penalty in our basic ERM setup:

$$\mathcal{L}(\theta; \lambda) = \underbrace{\left[\frac{1}{n} \sum_{i}^{n} \ell(y_i, \theta; x_i)\right]}_{\text{Loss}} + \lambda \underbrace{C(\theta)}_{\text{complexity}}$$

  This penalises complexity: lambda manages the trade off between complexity and loss.
- **Ridge Regression**: MSE for loss function and C is the squared sum of coefficients

$$\mathcal{L}_{ridge}(\theta; \lambda) = \underbrace{\frac{1}{n} \sum_{i}^{n} (y_i - x_i \beta)^2}_{\text{squared loss}} + \lambda \underbrace{\beta^T \beta}_{\text{sq sum coefs}}$$

  this penalises large β coefficients, lambda manages trade off.

I skipped the next slide - not sure what it's demonstrating.

24.4 Regularisation: the intuition behind it

NB!

TL;DR: Regularisation **introduces bias** in order to **reduce variance**.

- Suppose X is orthonormal (each column is independent, and normalised so var = 1)
- then $\hat{\beta}_{ridge} = \frac{\hat{\beta}_{ols}}{1+\lambda}$
- regularisation is re-scaling the OLS coefficients downwards.
- it introduces bias in order to reduce variance

NB: social scientists use un-regularised regression because they care about

1. Bias
2. Interpretability

Bias is first order concern in social science: you first find an unbiased estimator, and only then do you work to improve it. (Whereas ridge introduces bias (scaling down) for good reason: reduce variance)

24.5 Regularisaion: SVD

I skipped the SVD slide - I think basic point was that this is how python does regression; intuition ridge is scaling up the singular values, and this has significance for something (?) i think it makes it invertible?

24.6 Regularisation: geometrically

skipped - basically the prior pulls it towards it –> balances out

24.7 Regularisation as measurement error

[skipped]

if you add gaussian corruption to every one of your features this is just the same as ridge regression

we used to have one row of y and X now we have a 100 rows of X
Intuition: you add infinite Gaussian noise to each x feature –> you will get minimal coefficients because it will all be random: there will be no linear relationship between the X features and the y output. Less linear relationship means smaller coefficients. So, adding noise effectively shrinks the coefficients

simplifies to, $R(\theta) = \frac{1}{n}\sum_i^n (X\beta - y_i)^2 + \sigma^2\|\beta\|^2$

24.8 Regularisation: as a Bayesian

NB!

TL;DR: Bayesian MAP estimator *is* ridge regression. There's a 1:1 correspondence.

Regularization, a method to prevent overfitting by penalizing large model coefficients, has a direct analogy in Bayesian statistics through the concept of priors, which encode prior beliefs about parameters before observing the data.

Regularization as Implicit Priors

In a Bayesian framework, priors are assumptions about the distribution of model parameters before any data is observed. When you apply regularization in a machine learning model, you're implicitly making assumptions about the distribution of the parameters you're trying to estimate, similar to specifying a prior in Bayesian terms:

- **L1 Regularization (Lasso)** corresponds to assuming a **Laplace distribution** (a double exponential distribution) as the prior for the parameters. This creates a preference for solutions where many parameters are exactly zero, mirroring the sparsity induced by a Laplace prior.
- **L2 Regularization (Ridge)** corresponds to assuming a **Gaussian (normal) distribution** as the prior for the parameters. This encourages the parameters to be small overall but does not necessarily force them to zero, reflecting the properties of a Gaussian distribution.

Intuitive Understanding

The idea can be understood intuitively: If you believe that the true parameters of your model should be small (to avoid overfitting), you can express this belief by imposing a penalty on the size of the parameters (regularization) or by choosing priors that favor smaller values (Bayesian priors). Regularization in the optimization problem then acts to incorporate this belief directly into the model fitting process, similar to how Bayesian inference would update the prior beliefs in the light of observed data to arrive at the posterior distribution of parameters.

Mathematical Formulation

In both cases, you are adding extra information to guide the solution of the optimization problem towards certain properties:

- In regularization, this guidance comes in the form of an added penalty term to the loss function, which directly influences the optimization process.
- In Bayesian statistics, the guidance comes through the prior distribution, which is mathematically combined with the likelihood of the observed data to form the posterior distribution of the parameters.

Practical Implications

This relationship highlights a beautiful cross-over between frequentist (regularization) and Bayesian approaches. It suggests that by choosing a specific form of regularization, you're implicitly making assumptions akin to choosing a prior in Bayesian analysis. This perspective can help in selecting the appropriate regularization technique based on prior knowledge about the data or the domain.

- Take Normal MLE model, add a prior on β
- suppose $p(\beta = N(0, \tau, I)$ (tau is stan.dev)
  - $\tau -> \inf$ = weak regularisation: indifferent to prior = MLE
  - $\tau -> 0$ = strong regularisation: ignores the data = assumes $\beta = 0$
- MAP = ridge regression
- there is a relationship between tau and lambda

24.9 Regularisation in Polynomial Regression

Polynomial regression is a way to get high dimensional model, but we want some regularisation to penalise the complexity. This wasy we can get the best of both worlds by having high dimensionality (expressive), but regularised. This allows for complex, expressive models which do not overfit.

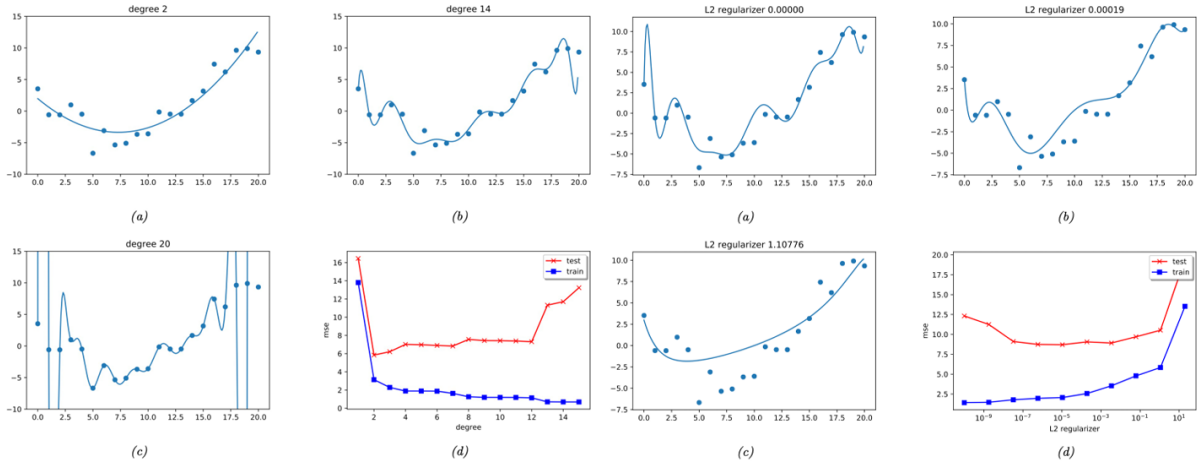


Figure 10: Right 4: all with L2 Regularisation. Especially (c): this is a smooth model in the way that a polynomial doesn't normally look. In this way, regularization allows us to get best of all

24.10 Validation Sets & Hyperparameter selection (λ)

To choose the correct λ value (a hyperparameter) for regularisation $\rightarrow$ Validation sets

$$\hat{\beta}_{ridge} = (X^T X + \lambda I_p)^{-1} X^T y$$

- We want to choose hyperparameter values to **minimise Generalisation Error**:

$$\mathbb{E}_{p^*}\mathcal{R}(f_n^*) - \mathcal{R}(f^{**}) = \underbrace{\mathcal{R}(f^*) - \mathcal{R}(f^{**})}_{\text{Approximation Error}} + \underbrace{\mathbb{E}_{p^*}\mathcal{R}(f_n^*) - \mathcal{R}(f^*)}_{\text{Estimation/Generalization Error}}$$

  Remembering the following approximation:

$$\underbrace{\mathbb{E}_{p^*}\mathcal{R}(f_n^*) - \mathcal{R}(f^*)}_{\text{Estimation/Generalisation Err}} \approx \underbrace{\mathbb{E}_{p_{\text{train}}}[\ell(y, f_n^*)]}_{\text{Training Loss}} - \underbrace{\mathbb{E}_{p_{\text{test}}}[\ell(y, f_n^*)]}_{\text{Test Loss}}$$

  We can say that decreasing the Training Loss/Risk, at the expense of increasing the Test Loss/Risk, reduces the Generalisation Error.

- We parition data into additional validation set:
  1. Train model on p_{train}
  2. Choose hyperparameters & model selection with $p_{validation}$
  3. Measure generalisation error with p_{test}

25 Lasso regression

- Whereas Ridge gives us small $\beta^T\beta$, Lasso makes it **sparse** (betas to 0)
- we replace $\beta^T\beta = \|\beta\|_2 \Rightarrow |\beta| = \|\beta\|$
- Beta-squared: quadratic $\Rightarrow$ absolute value of the coefficient vector: linear

$$\mathcal{L}_{lasso}(\theta;\lambda) = \frac{1}{n}\sum_{i}^{n}(y_i - x_i\beta)^2 + \lambda\|\beta\|_1$$

So, **with orthonormal X: Lasso becomes a threhold function:**

- MLE: $\hat{\beta}_{mle}$
  - *linear regression w/o regularization: least squares = equivalent to MSE if the errors are assumed to be normally distributed*
- Ridge: $\frac{\hat{\beta}_{mle}}{1+\lambda}$
  - *scaled down uniformly from the MLE estimate*
- Lasso: $(sign(\hat{\beta}_{mle})(|\hat{\beta}_{mle}| - \lambda)_+$
  - Where:
    - $sign(\hat{\beta}^{\text{mle}})$ determines the direction of the coefficient (+ or -).
    - $|\hat{\beta}^{mle}| - \lambda$ subtracts a constant λ (the regularization strength) from the absolute value of the MLE estimate.
    - The result is set to zero if it is negative (denoted by $(\cdot)_+$, which means taking the positive part), effectively performing variable selection by setting small coefficients to exactly zero.
  - *Thresholding function: take the magnitude of normal regression, subtract lambda, then take the*
    so when lambda is ? then this is just 0
    basically we don;'t need to know, just take it as a given

ML Lecture Notes: Wk 4 — Generalization and Complexity

Content

- practical advice on validation
- theoretical bounds on generalization
- conceptualising hardness of a problem: VC dimension
- generalization performance of linear models: simple low/high dimensional setting

26 Cross-validation

26.1 Overview

- Cross-validation assesses how results of a statistical analysis will generalize to an independent data set → allows us to pick model with best tuned hyperparameters.
- involves partitioning a sample of data into complementary subsets, performing the analysis on one subset (called the training set), and validating the analysis on the other subset (called the validation set or testing set).
- To reduce variability, multiple rounds of cross-validation are performed using different partitions, and the validation results are averaged over the rounds.

A specific method for calculating the risk (/error) of a predictive model during cross validation:

$$R_\lambda^{\text{cv}} = \frac{1}{K}\sum_{k=1}^{K} R_k(\hat{\theta}(D_{-k}^{\text{train}}), D_k^{\text{test}})$$

Where:

- R_i denotes model's risk (error rate) of the model on the kth fold, where
  - the model parameters θ are learned using...
  - ...training datasets (ie the full set without the kth validation set...
  - ... and evaluated on the kth test dataset
- R_{cv} is the cross-validation estimate of the risk (prediction error)
- K represents number of groups that a given sample is split into (number of folds)

When $K = n$: this is called the "leave-one-out CV (LOOCV). The model is trained on all data points except one and the prediction error is computed on the left-out observation. This process is repeated for each data point in the dataset, and the results are averaged to get the final estimate of the model's prediction error.

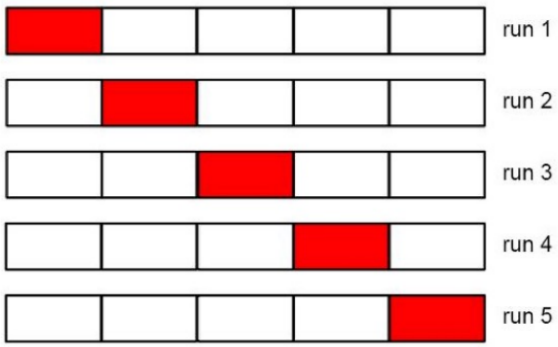


Figure 11: $K = 5$

After completing the cross-validation process and obtaining an estimate of the model's prediction error, the final step is to fit the model again on the entire dataset to obtain the final model estimates. This step is crucial because it allows the model to learn from the entire dataset, maximizing its performance and ensuring that no data is wasted. This final model, trained on all available data, is then used for making predictions on new, unseen data.

26.2 LOOCV in linear regression

aka Linear regression is amazing.

The expression for the prediction error in LOOCV for linear regression can be simplified due to the properties of linear models, leading to a more efficient computation that doesn't require refitting the model n times.

Deriving formula for the LOOCV error (MSE_{cv}) in linear regression:

MSE / prediction risk calculated using OLS loss function for cross validation:

$$MSE_{(cv)} = \frac{1}{n}\sum_{i=1}^{n}\epsilon_{-i}^2$$

And we have the 'Hat matrix' H, which projects the vector of observed dependent variables (y) onto the vector of predictions ($\hat{y}$). Hat matrix is defined as:

$$H = X(X^TX)^{-1}X^T$$

Then

$$\begin{aligned}\text{MSE}_{\text{cv}} &= \frac{1}{n}\sum_{i=1}^{n}\left(\frac{\epsilon_i}{1-h_{ii}}\right)^2 \\ &= \frac{1}{n}\sum_{i=1}^{n}\left(\frac{y_i-\hat{y}_i}{1-h_{ii}}\right)^2\end{aligned}$$

Where:

- y_i = actual value of dependent var for ith observation
- $\hat{y}_i$ = predicted value (based on model trained without the ith observation)

- h_{ii} is the ith diagonal element of the hat matrix H
- $1 - h_{ii}$ adjusts the errors for the ith observation, accounting for the leverage (influence) of the observation on its own prediction

26.2.1 Significance

- This formula allows for the direct calculation of the LOOCV error without the need to explicitly leave out each observation and retrain the model each time.
- It leverages the mathematical properties of linear regression (specifically utilizing the leverage values from the hat matrix)...
- making LOOCV particularly efficient and appealing for linear models (without the computational burden of refitting the model n times).

26.3 One Standard Error Rule

- A heuristic that helps for model selection, balancing the trade-off between model complexity and generalization performance.
- Based on the principle of Occam's razor: preferring simpler models that are less likely to overfit the data.
- When selecting models based on cross-validation (CV) risk estimates, it's tempting to just choose the model with the lowest CV risk.
- However, rather than looking at point estimates of CV risk, we should look at the standard error whiskers.
- by ackowledging uncertainty in our risk estimates, we are basically saying that a slightly moe regularised model (simpler) is within reasonable margin of error as good as the very lowest –> since we have uncertainty here, we should favour the simpler one...
- this model may be more complex than necessary and more prone to overfitting.

26.3.1 Process

1. **Calculate CV risk for each model**: perform K-fold CV for each model and calculate prediction error (risk) for each fold. Then compute the mean CV risk for each model.
2. **Calculate the standard error of the CV risks**: from the variation in the risk estimates across the K folds.
    - Specifically, SE can be computed as the standard deviation of the K CV risk estimates divided by the square root of K
    - $SE_{cv} = \frac{\sigma CV}{\sqrt{(K)}}$
3. **Apply One Standard Error Rule:**
    - identify the model with the lowest CV risk: $R_\lambda(\theta, D_{validation})$
    - then select a simpler model within one SE of the lowest CV risk.
    - formally:
        - if $\hat{R}_{min}$ is the model with the lowest CV risk observed
        - and SE_{min} is the SE of the cross validation risk estimate

- then we consider any model with a CV risk of $\hat{R} \leq \hat{R}_{min} + SE_{min}$ as candidate
- And among these candidates, we prefer the simpler models (higher lambda coefficients

By allowing for the selection of a model within one SE of the lowest error, it incorporates a margin of uncertainty in the model selection process, preventing the selection of overly complex models that might not significantly outperform simpler models on new data.

This encourages the selection of the simplest model that is within an acceptable range of performance, thus promoting better generalization to unseen data.

26.3.2 Optimism of the training error (overfitting)

The optimism of the training error refers to the tendency of a model's performance on the training data to be more optimistic (i.e., better) than its performance on unseen data. This phenomenon arises because the model θ are optimized on the training data, making the model potentially too complex and sensitive to the idiosyncrasies of the training set, a situation known as overfitting.

Idea that the training error will be over optimisitic of a model's perfomance on real world data, and so **will lead us to select an overly complex model**.

If we just optimise based on the training error, we would select our regularisation parameter λ to be 0 (no regularisation).

This is why we need to optimise (/select model) based on different criteria than just minimising the training error. This is where the One Standard Error Rule comes in.

Formally, if we were to optimise our lamba value based on minimising the training error:

$$\begin{aligned}\hat{\lambda} &= \text{argmin}_\lambda \min_\theta R_\lambda(\theta, D) \\ &= \text{argmin}_\lambda \min_\theta R_\lambda(\theta, D) + \lambda C(\theta) \\ &= 0\end{aligned}$$

Where

- the $\hat{\lambda}$ is the value of λ that minimises the risk $R_\lambda(\theta, D)$
- the inner min_θ indicates process of find the model parameters θ that minimise the risk for a given λ
- the outer $argmin_\lambda$ is finding the value of λ that leads to the lowest prossible risk after θ has been optimised.

This 0 suggestes optimal value of reglarisation parameter is 0, under the defined optimisation problem. This shows the optimism of the training error. Using this as our metric would be overoptimistic of our model's ability, so instead we need to optimise (select model) on different criteria.

26.4 Grouping considerations for K-fold CV

26.4.1 i.i.d violated: info leakage between observations

Assume a simple linear generative model:

$$y_i = X_i\beta + \epsilon_i$$

Where

- all the features are neatly normally distributed (X_{ij} $N(0,1)$ for each i, j
- BUT the collective noise term $\epsilon_i = N(0, \Sigma)$: meaning the collective noise for a given unit i follows a multivariate distribution (covariance condition)
- this implied dependencies among the observations, within a unit i
- each unit we observe is within a country, $c(i)$ (ie data points grouped by country)
- $\Sigma_{ij} = 0$ if $c(i) \neq c(j)$ = non i.i.d
  - this covariance condition specifies that observations from different countries are uncorrelated
  - however, observations within same country (where $c(i) = c(j)$) may have non-zero covariance
  - i.e. they are not independent of each other

$\rightarrow$ this introduction of country-specific covariance = structured dependencies and non-i.i.d

In this case, how do we evaluate models?

Intuition:

When assigning data points into K folds for CV, we need to account for the structured dependencies among observations: ensure that info from one observation does not bleed into another observation, by rethinking how we sample from the population

Cross-validation adjustments that respect the country grouping.

Ensure that observations from the same country are not split between training and testing sets inappropriately

26.4.2 Group K-fold & Timeseries Split

- **Examples:**

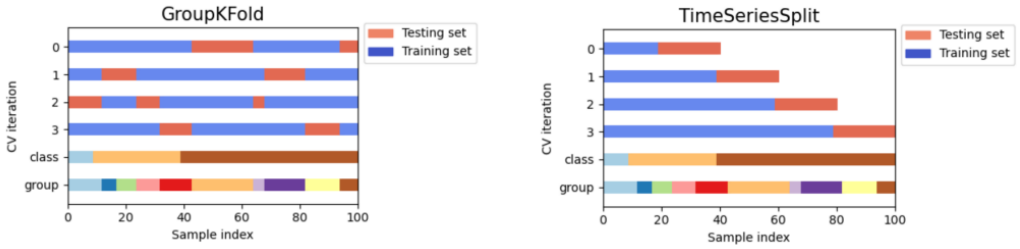


Figure 12: CV Grouping. On Left: see that data from a given group remains in groups (ie groups not split across folds at any point) across all CV iterations. On Right: when there's a time series, use future data to validate models trained on past data

Group K-Fold

- When data contain groups that are expected to have similar properties, it's important to ensure that observations from the same group are **not** spread across both training and validation sets. This prevents **information leakage, where the model indirectly learns about the validation set during training**.
- Each fold is a cross-validation iteration, and within each fold, **all observations from a particular group are contained either entirely within the training set or entirely within the testing set**.
- This respects the group structure, ensuring that when a model is trained, it does not see any data from the group that is in the testing set for that fold.

Timeseries Split

- For time series data, the temporal order of the observations must be preserved. **Data from the future should not be used to predict past events** as this would constitute temporal leakage.
- This strategy involves training on an initial segment of the time series data and using the subsequent segment for validation. **With each fold, the training set grows**, always including the data up to the next testing period but **never mixing future data into the training set**.

General Cross-Validation Principle

- **Information Bleed:** Cross-validation should be designed in such a way that information from one observation doesn't "bleed" into another, meaning that the training data should not contain information that could give away the answers for the validation data.
- **Same Fold Requirement:** Observations that are related or grouped by some inherent connection (like being from the same country, belonging to the same subject, or being sequentially related in time) should be placed within the same fold of the cross-validation process to avoid information leakage.
- **Sampling Process:** When designing the folds for cross-validation, the sampling process that generated the data should be considered. If there's a structure or dependency in the sampling process, it needs to be accounted for in the way the data is split for training and testing.

27 Bayes Risk

Definitions of risk so far have been **frequentest risk**.

Frequentist risk

- **Risk = expected loss / errors** of a decision function or estimator across different data samples, assuming that the true parameter θ is fixed.
- Defined as a function of a loss function (e.g. MSE / other)
- Quantifies how far our estimates / decisions are from the true value of θ
- **Treats θ as a constant** that we aim to estimate as accurately as possible.
- **Data is the r.v. (where the errors are)**

Bayes risk

- **Treats parameters θ as r.v.** (not the data)
- Integrates over all possible values of these parameters, weighted by their probability, as described by a prior distribution $\pi_0(\theta)$.
  - NB integration is same as summing - integration is mathematical way to take the expectation of the risk with respect to the joint distribution of the parameters θ and the data x:

$$\begin{aligned} R\pi_0(f) &= E\pi_0(\theta)\left[R(\theta, f)\right] \\ &= \int d\theta dx \pi_0(\theta) p(x|\theta) \ell(\theta, f(z)) \end{aligned}$$

Where

- $R\pi_0(f)$ = Bayes risk for a decision function f. Represents:
  - expected value of the risk $R(\theta, f)$

  - with respect to a prior distribution $\pi_0(\theta)$

- the expectation $(E\pi_0(\theta)\left[R(\theta, f)\right])$ is taken over the distribution of θ, considering all possible values that θ could take, each weighted by its prior probability
- the integral $\int d\theta dx \pi_0(\theta) p(x|\theta) l(\theta, f(z))$ gives formal expression for this expectation.
  - it is integrating over all values of θ and the data x, where:
    * $\pi(\theta)$ is the prior distribution of θ
    * $P(x|0)$ is the likelihood function of observing the data x, given parameter θ
    * $l(\theta, f(z))$ is the loss function that quantifies cost of decisions of estimates made by f when the true state of nature is θ and the observed data is z

A note on use of integration:

- **Expectation over continuous r.v.s:** integration is mathematical way to take the expectation of the risk with respect to the joint distribution of the parameters θ and the data x:
- When dealing with continuous random variables, expectations are calculated using integrals. This is **analogous to taking a weighted average, where the weights are given by the probability density function** of the variable.
- *Think about taking a slice of the PDF: you are taking the whole area under the curve, which is higher (denser) at points more likely - thus weighting them more*
- **Prior distribution:** $\pi_0(\theta)$ represents our beliefs re: possible values of θ before observing any data. Since it is an r.v. we need to account for all possible values it can take. The integral computes the weighted avg of the risk over all these possible values.
- **Likelihood**: $p(x|\theta)$ is the likelihood of observing the data x given a particular value of θ. For each value of θ, the likelihood can be different, and it indicates how well the data agree with the parameter.
- **Loss Function**: $\ell(\theta, f(z))$ quantifies the "cost" of using the decision function f when the true parameter is θ and the observed data is z. This function is what we aim to minimize in making decisions or estimations.
- **Joint Distribution**: The product $\pi_0(\theta)p(x|\theta)$ gives us a joint distribution over θ and x. This joint distribution is what we integrate over to calculate the expected loss.
- **The Integral**: The integral $\int d\theta dx \pi_0(\theta)p(x|\theta)\updownarrow(\theta, f(z))$ essentially sums up the weighted loss across all combinations of θ and x, where the weights are given by the joint probability of θ and x. This is why it's a double integral: it integrates over both the parameter space and the data space.

Here is a step-by-step breakdown of what the integral does:

- For each potential value of θ, determine the likelihood of the observed data x.
- Compute the loss for the decision function f based on that θ and x.
- Weight this loss by the joint probability of θ and x (which comes from the prior and the likelihood).
- Sum (integrate) this weighted loss across all possible values of θ and all possible observations x to get the average (expected) loss.

Bayes risk thus reflects the avg performance of f when both θ and x are r.vs. (???? does it)

It accounts for uncertainty about θ by averaging over its distribution.

Key difference is how uncertainty re: parameter θ is handled:

- **Frequentest: assumes θ FIXED but unknown**. Does not use prior info. Risk is calculated relative to this fixed θ

- **Bayes: assumes θ is r.v, with its own distribution (the prior)**. The risk is calculated by averaging over all possible values of θ given by this prior distribution

28 Generalisation Bounds

NB: all of this is using bound for a simple binary classification problem.

Reminder: generalization error

- **Best possible function:**

$$f^{**} = \arg\min_f \mathcal{R}(f)$$

- **Best function within a considered hypothesis space:**

$$f^* = \arg\min_{f \in \mathcal{H}} \mathcal{R}(f)$$

- **Our empirical best guess:**

$$f_n^* = \arg\min_{f \in \mathcal{H}} \mathcal{R}(f, \mathcal{D}) = \arg\min_{f \in \mathcal{H}} \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, f(x_i))$$

$$\mathbb{E}_{p^*} \mathcal{R}(f_n^*) - \mathcal{R}(f^{**}) = \underbrace{\mathcal{R}(f^*) - \mathcal{R}(f^{**})}_{\text{Approximation Error}} + \underbrace{\mathbb{E}_{p^*} \mathcal{R}(f_n^*) - \mathcal{R}(f^*)}_{\text{Estimation/Generalization Error}}$$

$$R3 - R1 = (R2 - R1) + (R3 - R2)$$

Concrete example of generalisation error: truncating polynomials

- **True model:** $x \sim \text{Unif}(0,1)$; $y = \beta_0 + \beta_1 x + \beta_2 x^2 + \epsilon$
- **Best possible function:**

$$\begin{aligned} f^{**} &= \arg\min_f \mathcal{R}(f) \\ &= \beta_0 + \beta_1 x + \beta_2 x^2 \end{aligned}$$

- **Best function within a considered hypothesis space:**
  NB we are truncating the polynomial here

$$\begin{aligned} f^{*} &= \arg\min_{f \in \mathcal{H}} \mathcal{R}(f) \\ &= \beta_0^* + \beta_1^* x \end{aligned}$$

- **Our empirical best guess:**

$$\begin{aligned} f_n^{*} &= \arg\min_{f \in \mathcal{H}} \mathcal{R}(f, \mathcal{D}) \\ &= \hat{\beta}_0 + \hat{\beta}_1 x \end{aligned}$$

We could then work out by hand:

$$\mathbb{E}_{p^*}\mathcal{R}(f_n^*) - \mathcal{R}(f^{**}) = \underbrace{\mathcal{R}(f^*) - \mathcal{R}(f^{**})}_{\text{Approximation Error}} + \underbrace{\mathbb{E}_{p^*}\mathcal{R}(f_n^*) - \mathcal{R}(f^*)}_{\text{Estimation/Generalization Error}}$$

$$R3 - R1 = (R2 - R1) + (R3 - R2)$$

This gives us:

- Approximation error = neglecting the x^2 term
- Estimation/Generalisation error = Estimation error in $\hat{\beta}$ (due to modeling deficiencies, or insufficient data)

To know *a-priori* how well a model will perform on unseen data: we seek a bound on generalisation error

The tighter the bound, the more confident we can be about the performance of our model on unseen data.

If we restrict ourselves to classification tasks $y \in 0, 1$, bound on the generalization error for classification problems might look as follows:
NB this is the generic framework for setting up a bound

$$p\big(\max_{f^* \in \mathcal{H}} \big[\mathcal{R}(f_n^*) - \mathcal{R}(f^*)\big] \text{ is big}\big) \text{ is small}$$

Where

- $p(\cdot)$ is small: the chance is small over datasets we might see; stating that with high probability (e.g., 95%), the true generalization error will be below a certain threshold;

- $\max_{f^* \in \mathcal{H}}$: considers the worst case over scenario across all models within a hypothesis space
- $[\mathcal{R}(f_n^*) - \mathcal{R}(f^*)]$ is big: generalisation error = the difference between the empirical and the true risk: we don't want big generalization error

NB this is crude, loose bound. We can do better - but this is nice and simple as a first pass.

28.1 Uses of bounds

NB bounds are NOT a replacement for estimate of the error.

- Typically, we estimate the error of a model by measuring its performance on a validation set or using cross-validation (Empirical Error Estimation) .
- This gives us an empirical error rate that indicates how well the model is doing on data it hasn't seen during training.
- However, this approach has limitations, especially when the available data is limited or not representative of the full range of scenarios the model may encounter.

Instead, bounds they help us to:

- **Reason over models** - especially when empirical estimates of error are close or not available. They provide a way to judge which models might be more robust or likely to generalize well.
- **Reason how well models performance varies with sample size** - theoretical bounds often include terms that relate to the sample size, allowing us to predict how adding more data may impact model performance. This is crucial for planning data collection and understanding the trade-offs between data quantity and model complexity.
- **Understand what assumptions we need for good performance** - for instance, they may highlight the need for certain data distribution properties, the importance of model assumptions, or the influence of parameter settings.
- **Understand, will we be safe for the worst class** - can offer confidence that, even under unfavorable conditions, the model's error won't exceed a certain threshold with high probability.

To arrive at the bound statement given above, we have probability tools we need in place: (1) Hoeffding's Inequality; (2) Boole's Inequality.

28.2 Composite Tool 1: Hoeffding's inequality

- provides a way to quantify the uncertainty of empirical estimates.
- Gives us a way to bound the probability that the sum of bounded independent random variables deviates from its expected value.
- Useful for understanding how far estimates based on sample data (like the sample mean) are likely to be from the true population parameter (like the population mean).

For $X_1, \cdots, X_n$ $Bern(\theta)$ (ie independent r.v.s drawn from Bernoulli distr), we're interested in the probability that the sample mean $\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$ of these random variables **deviates** from the expected value $\mathbb{E}[\bar{X}] = \mu$ by more than some small positive amount t.

Deviation:

- This is the absolute difference between the observed statistic (like a sample mean) and its expected value (population mean).
- Hoeffding's inequality tells us that the probability of this deviation is bounded above by $2\exp(-2\sigma^2)$.

Formally, for any $\epsilon > 0$:
where ϵ is a threshold of deviation

$$P(|\bar{X} - \mu| > \epsilon) \leq 2\exp(-2n\epsilon^2)$$

NB $\theta = \mu$ - μ is just the mean as a possible parameter, whereas θ is generic complete set of parameters for a model
Where

- $P(\cdot)$: the chance if small
- $|\bar{X} - \mu| > \epsilon$: the deviation is large(r than the threshold ϵ
- $2\exp(-2n\epsilon^2)$: gets smaller quickly with sample size <- NB the negative n term

Implications:

- **The Chance is Small**: As n increases, the probability of a large deviation becomes exponentially smaller, meaning that with more observations, the sample mean is very likely to be close to the true mean μ.
- **The Deviation is Large**: We are bounding the probability of a deviation that is more significant than our threshold ϵ.
- **Exponential Decrease**: The bound $2\exp(-2\sigma^2)$ decreases exponentially with the number of samples n, which reassures us that the law of large numbers holds – as we get more data, our sample mean gets closer to the expected value.

NB!

TL;DR: Hoeffding's inequality tells us that as n increases, the sample parameter estimates converge v closely to their true population estimates

Proof is in Markov's inequality...

28.3 Composite Tool 2: Boole's Inequality / Union Bound

Provides an upper bound on the probability of the union of multiple events.

If $\mathcal{E}_1, \ldots, \mathcal{E}_d$ are events:

- $P(\mathcal{E}_1 \cup \ldots \cup \mathcal{E}_d) \leq \sum_{i=1}^{d} P(\mathcal{E}_i)$
- $P(\mathcal{E}_1 \text{ OR } \mathcal{E}_2) \leq P(\mathcal{E}_1) + P(\mathcal{E}_2) - P(\mathcal{E}_1 \cup \mathcal{E}_2)$

And so, doing away with the 'complex' RHS term about the intersection/union, we can say:

$$P(A_1 \text{OR} A_2) \leq P(A_1) + P(A_2)$$

NB!

Boole's Inequality states that the probability of the union of a finite number of events is less than or equal to the sum of the probabilities of each event.

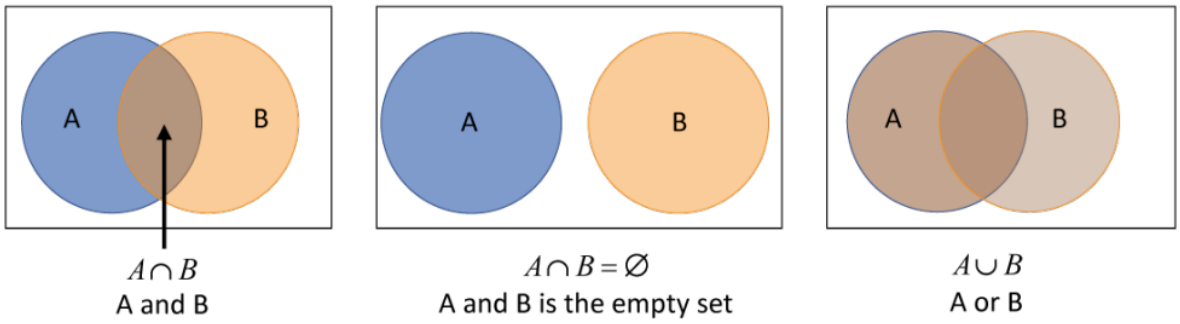


Figure 13: Enter Caption

The Union Bound is often used in scenarios where calculating the exact probability of the union of events is complex or intractable due to intersections between events. It provides a quick way to get a conservative estimate.

In machine learning, for example, it's commonly used to derive bounds on generalization error, as it simplifies the analysis by avoiding the need to directly calculate the intersections between events.

Offers an easy way to estimate the probability of any one of a set of events happening, without needing to know the detailed relationship between those events.

28.4 Combining Hoeffding & Boole's inequality together → First Generalization bound

The generalization error of a binary classifier will be more than ϵ, in the worst case is upper bounded by:

$$p\Big(\max_{f^* \in \mathcal{H}} \big[\mathcal{R}(f_n^*) - \mathcal{R}(f^*) > \epsilon\big]\Big) \leq 2\dim(\mathcal{H}) \cdot \exp(-2n\epsilon^2)$$

Where

- $p(\cdot)$: **probability over different datasets.** It is the chance that the generalization error will be more than ϵ across different potential training datasets.
- $\max_{f^* \in \mathcal{H}}$: **Worst case over hypothesis space (of size $\dim \mathcal{H}$).** This denotes taking the maximum over all hypotheses in the hypothesis space. It's looking at the worst possible model that we could obtain from our learning algorithm.
- $\mathcal{R}(f_0^*) - \mathcal{R}(f^*)$: **Generalization error is worse than ϵ** This denotes the actual generalization error, which is the difference between the true error rate (often the best error rate we could achieve on the distribution from which the data is drawn) and the empirical error rate (the error rate on the training set)
- $2 \cdot dim(\mathcal{H}$: a factor that scales with a measure of the complexity or size of the hypothesis space
- $\exp(-2n\epsilon^2)$: **A chance that disappears as $n \to \infty$** - This is the bound itself, an exponentially decreasing function of the number of sample and the square of the threshold. Shows that the probability decreases exponentially as the number of samples n increases, or as the threshold ϵ becomes tighter.

NB!

Indicates that as we increase the number of samples n, the probability of the generalization error being larger than threshold ϵ becomes exponentially smaller.

Gives us confidence in the reliability of our binary classifier: as our sample size grows, the "chance" that our classifier's generalization error will be worse than some threshold becomes very small, thus indicating that with sufficient data, our classifier is likely to perform well on unseen data.

28.5 Proof:

THESE COLOURS AREN'T RIGHT - LOOK BACK AT SLIDES

$$\Pr\left(\max_{f^* \in \mathcal{H}} [\mathcal{R}(f_n^*) - \mathcal{R}(f^*) > \epsilon]\right) = \Pr\left(\bigcup_{I^* \in \mathcal{H}} [\mathcal{R}_{I^*}(f_n^*) - \mathcal{R}(f^*) > \epsilon]\right)$$

$$\text{(Union bound)} \leq \sum_{I^* \in \mathcal{H}} \Pr\left([\mathcal{R}_{I^*}(f_n^*) - \mathcal{R}(f^*) > \epsilon]\right)$$

$$\text{(Hoeffding Inequality)} \leq \sum_{f^* \epsilon \mathcal{H}} 2exp(-2n\epsilon^2)$$

$$\text{(Finiteness of} \mathcal{H}) \leq 2\dim \mathcal{H} \exp\left(-2n\epsilon^2\right)$$

28.6 Implications

$$p\left(\max_{f^* \in \mathcal{H}} \left[\mathcal{R}(f_n^*) - \mathcal{R}(f^*) > \epsilon\right]\right) \leq 2\dim(\mathcal{H}) \cdot \exp(-2n\epsilon^2)$$

This inequality is making a statement about the probability of the generalization error of the best (worst???) hypothesis in the hypothesis space H being greater than some threshold ϵ

- **Optimism in our training error:** the tendency of the empirical risk R_S^* to underestimate the true risk $R_{D_0}^*$, because its measured on the same data that was used to train the model
- **Error in the population...**: ie the true risk $R_{D_0}^*$, which is the expected performance of the model on the entire distribution of data, not just the training set.

NB!

– **...increases with size of hypothesis space** - larger hypothesis space, the more complex the models it contains, greater chance of overfitting. The factor $2 \cdot dim(H)$ reflects this by increasing the bound on the generalization error accordingly

NB!

– **... decreases with sample size** as we collect more data, less likely model will have large generalisation error (ie helps model learn true underlying patterns, rather than noise / idiosyncrasies). The term $exp(-2n\epsilon^2)$ reflects this.

This generalization bound provides a probability measure that tells us how confident we can be that our model, selected from a set of possible models, will perform well on new data. It balances the complexity of the model (as represented by the size of the hypothesis space) with the amount of data we have, giving us a way to predict and control for overfitting.

28.7 Issues with this bound

$$p\big(\max_{f^* \in \mathcal{H}} \big[\mathcal{R}(f_n^*) - \mathcal{R}(f^*) > \epsilon\big]\big) \leq 2\dim(\mathcal{H}) \cdot \exp(-2n\epsilon^2)$$

1. we have a finite hypothesis set
2. assumes are data i.i.d
3. is the Hoeffding the best possible bound?

29 Intrinsic Dimensionality

= *the minimum number of parameters needed to accurately describe every point within a dataset or space.*

- reflects the true 'complexity' or 'structure' of the data,
- as opposed to the ambient dimension, which is the space the data is embedded in.

Dimensionality of a circle; the earth etc

- A circle, despite being embedded in a 2D space, is essentially 1-dimensional if you consider only the path around it.
- Earth's surface, while part of a 3D space, can be thought of as 2-dimensional because you only need two dimensions (latitude and longitude) to describe any location on it.
- For the circle and the Earth, their intrinsic dimensions are 1 and 2, respectively, irrespective of the higher-dimensional space they may reside in.

Manifold

- manifold is a space that might be complex globally but resembles simpler Euclidean spaces locally
- "a surface in which every local area looks only n dimensional
- Manifolds are used to model complex shapes and structures in higher-dimensional spaces by considering them as collections of simpler, lower-dimensional pieces

Manifold hypothesis

- **High Ambient Dimensionality Masks Low Underlying Intrinsic Dimensionality**
- while data may exist in a high-dimensional space (high ambient dimensionality), the 'real' structure of the data occupies a much lower-dimensional space (low intrinsic dimensionality).
- eg **images of faces might exist in a space of thousands of dimensions (each pixel being a dimension), but the variations between faces (such as the shape of the nose, the distance between the eyes) can be described with far fewer dimensions**
- applied to ML: even though we often deal with high-dimensional data, the effective complexity of our data might be much lower → development of algorithms that aim to uncover and exploit these lower-dimensional structures (e.g., dimensionality reduction techniques like PCA, t-SNE, autoencoders)

29.1 Example: modeling how location predicts binary vote choice

Location Features: as predictors for voting preferences

- latitude
- longitude
- height

Restricted linear hypothesis space:

- **hypothesis space constraints:** each dimension of the location features X_i can take on values from the set $-1, 0, 1$ - i.e. simplified model where each feature can have a neg, pos or no effect. ($\hat{\beta} \in -1, 0, 1$ for every dimension$i$)
- **Model complexity of n features**: the number of possible models in the restricted space is 3^n - as each feature can take on 3 possible values. For K features we have 3^K possible models $\rightarrow$ exponential growth in number of models as more features added.

Example Models & Bounds Bound for binary classification given by:

$$2 \cdot \dim(\mathcal{H}) \cdot \exp(-2n\epsilon^2)$$

- **Model 1: w/ all 3 features (lat, long, height)**:
  - $k = 3 \rightarrow dim(\mathcal{H}) = 3^3$
  - bound $= 2 \cdot 3^3 \cdot \exp(-2n\epsilon^2)$
- **Model 2: w/ 2 features (lat, long)**:
  - $k = 2 \rightarrow dim(\mathcal{H}) = 3^2$
  - bound $= 2 \cdot 3^2 \cdot exp(-2n\epsilon^2)$

- **pos exponential relationship between features and bound** - as we add features -> exponential growth in hypothesis space (*this in turn gets put through linear scalor factor in the bound definition*)
- **neg exponential relationship between observations and bound** - as we add observation -> exponential decrease in bound term

NB!

Significance: model with only 2 features has a tighter bound on the generalization error, indicating potentially better generalisability...

But remember, there's a trade off here between generalisation error vs approximation error: while fewer features might make the model more generalisable to new data, it might make the model itself less expressive, increasing the approximation error.

29.2 ... Question on feature redundancy

In the above example: **is height redundant?**

- is $height = f(latitude, longitude)$?
- if so, potentially redundant for the model.

- If height does **not provide additional, independent information** beyond what latitude and longitude already offer, excluding it might **simplify the model (reducing variation / generalisation error) without significantly sacrificing predictive power**.
- Tighter bound on generalizability error, without a trade off for predictive power!

The complexity of a model (in terms of the number of features) affects the theoretical bounds on its generalization error, with implications for model selection and feature inclusion.

30 On Complexity

How can we define the **expansiveness of a hypothesis class**?

When discussing the complexity of a hypothesis class, the aim is to quantify **how "rich" or "flexible" the set of functions (or models) within** that class is.

A more complex hypothesis class can fit a wider variety of data patterns, but it also has a higher risk of overfitting to the training data. There are several ways to define or measure the complexity of a hypothesis class:

- **Count Parameters**, or degrees of freedom within it.
- **Measuring 'Wiggliness' (derivatives)** - smoothness of a function quantified using derivatives.
  - For a given function, areas where the derivative (rate of change) is large indicate rapid changes in the function's output, contributing to "wiggliness."
  - In some contexts, the total variation, which is a measure of how much a function varies, can be used. For smoother functions, this variation would be lower.
  - eg Sobolev spaces.
- **VC-Dimension (Vapnik-Chervonenkis Dimension)** - a more sophisticated measure of hypothesis class complexity; quantifies the capacity of a hypothesis class based on the largest set of points that the class can shatter:
  - "Shattering" refers to the hypothesis class's ability to correctly classify all possible label configurations for a given set of points, indicating a high level of flexibility or complexity.
  - A higher VC-dimension indicates a more complex hypothesis class. However, with increased complexity comes a higher risk of overfitting.
- **Rademacher Complexity** - quantifies hypothesis class complexity by measuring the class's ability to fit random noise:
  - It is defined as the expectation of the supremum (over the hypothesis class) of the average error on a dataset, where the labels are randomly assigned.
  - This complexity measure helps in understanding how well the hypothesis class can generalize, with higher values indicating a greater ability to fit random patterns (and thus, potentially, a higher risk of overfitting).

30.1 Vapnik-Chervonenkis (VC) Dimension

Allows for more rigorous generalization error bound.

TL;DR - VC's bound simplifies or is approximated to a form that indicates the generalization error:

- decreases as the number of samples increases
- increases with the complexity of the hypothesis class (where V - the VC dimension - represents that complexity); when V is big -> big error

The VC dimension itself:

$$\text{VC}(\mathcal{H}) \leq \frac{1}{\epsilon}\left(\log_2\left(\frac{2}{\epsilon}\right)\log_2(|\mathcal{H}|) + 1 - \log_2(\delta)\right) \approx 1 - \epsilon$$

= VC dimension is a measure of the capacity of a hypothesis class, denoting the largest number of points that can be shattered (correctly classified in all possible ways) by the hypotheses in $\mathcal{H}$.

It's a measure of complexity

The VC dimension is used to determine a model's bound (Vapnik's bound interpretation), which is a formal way to quantify how the complexity of a hypothesis class impacts its ability to generalize from training data to unseen data.

$$p\Big(\max_{f^* \in \mathcal{H}}\big[\mathcal{R}(f_n^*) - \mathcal{R}(f^*)\big] \leq \sqrt{\frac{1}{n}\Big[V\big(\log\frac{2n}{V} + 1\big) - \log\frac{\epsilon}{4}\Big]}\Big) \approx 1 - \epsilon$$

Where:

- V represents complexity of $\mathcal{H}$ (i.e. the VC dimension0

- bound quantifies the likelihood that the generalization error exceeds a certain threshold..
- ... which depends on number of samples
- ...and complexity of the hypothesis class
- = provides a probabilistic guarantee about the worst-case generalization error across all hypotheses in H.

Nb we are still in the same kind of framework as before:

$$p\Big(\max_{f^* \in \mathcal{H}}\big[\mathcal{R}(f_n^*) - \mathcal{R}(f^*)\big] \text{ is big}\Big) \text{ is small}$$

30.1.1 VC Dimension & "shattering"

How many points can functions in a hypothesis class perfectly predict?

For how many points can ERM drive training error to zero?

For linear classifiers: this is often 1 + number of parameters:

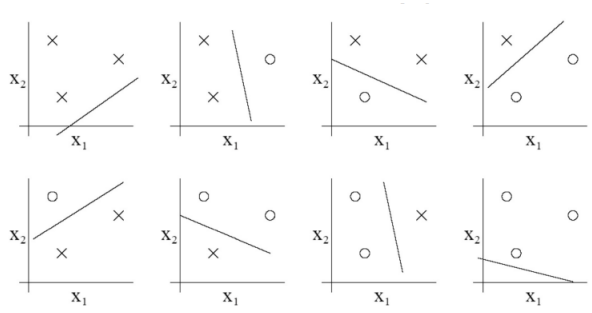


Figure 14: 3 points can be perfectly classified using 2 features (X_1, X_2)

BUT this is not true for all functions: VC does not *always correspond to parameter counting*

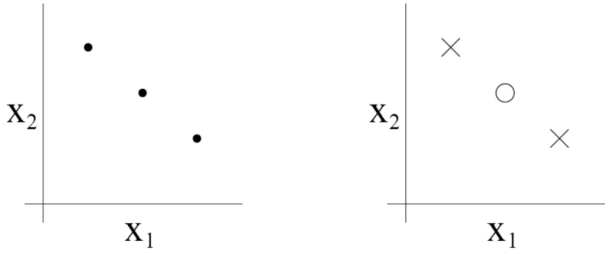


Figure 15: This configuration: 3 observations cannot be classified with decision boundary across / as function of 2 features

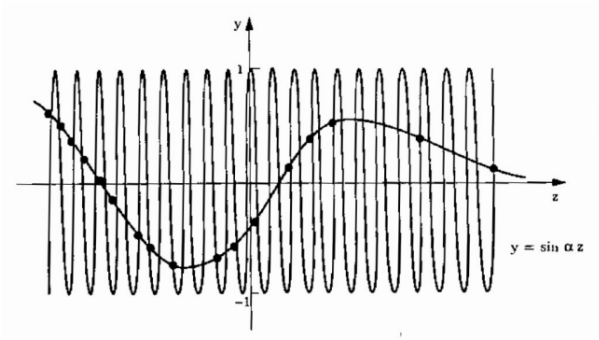


Figure 16: frequency of cosine: even w/ 1 param it can perfectly interpolate!

31 Structural Risk Minimisation

- avoid cross-validation
- just minimise an upper bound on test error
- i.e. $R(f_n^*) + bound$

Sure, it's an option...
BUT... just do cross validation

32 Generalisation in OLS

I didn't get these slides

Basically saying that OLS specifically perfectly generalises.

32.1 OLS Estimation & Generalization Error

OLS estimator: minimizes sum of sq errors - $\beta = argmin_\beta(Y \hat{} - X\beta)'(Y - X\beta)$

Generalization error: how well the model, with parameters estimated from the training data, performs on new, unseen data.

It's captured by the difference between the predicted and true values, considering the true model parameters.

Decomposition ... end up with:

$$\hat{\beta} = \beta^* + (X'X)^{-1}X'e$$

Which shows that the OLS estimator is the sum of the true parameters and a term that depends on the error e.

Specifically, this decomposition highlights how the estimation error $(\hat{\beta} - \beta^*)$ is influenced by the error term e, through the term $(X'X)^{-1}X'e$

Essentially, the estimation error depends on the variability of the design matrix X and the error term e: deviations from the true model arise due to the error in the data.

In an OLS setting, understanding the estimation error and its components is crucial for assessing how well the model can generalize. If the error term is small and the design matrix is well-conditioned (not too collinear), the OLS estimator will be close to the true parameters, indicating good generalization capability.

OLS is BLUE: Best Linear Unbiased Uestimator, under the Guass-Markov theorem.

***

...the finally derived line in this slide:

NB!

$$\text{Expected error} = \mathbb{E}[X(X^TX)^{-1}X^Te)^2]$$

Where

- X^TX represents the covariance matrix of the predictor variables.
- $(X^TX)^{-1}$ is its inverse; is used to estimate the coefficients such that the squared residuals are minimized
- NB the hat matrix: $H = (X^TX)^{-1}X^T$ - this projects the observed Y values onto the space spanned by the columns of X, resulting in the fitted values.
- however, here we are multiplying all that by error term e and a squared opertion outside the expected value –> captures expected squared length (or squared norm) of some vector related to residuals = an analysis of predictive performance

ASK ABOUT THIS- ITS IMPORTANT AND USED RIGHT AT THE END OF THE LECTURE, BUT I DON'T UNDERSTAND WHAT IS IS.
if it is expected error, then does that mean it is both generalisation and approximation error?
NB: Generalization Error = $E[(Y_{new} - X_{new}\beta)^2]$
I think it connects to that?

33 Detour: Singular Value Decomposition

NB this is general context needed for the maths behind estimating the generalisation error in final Low vs High dimensional feature spaces (i.e. next section)

Recap: $X = U\Sigma V^T$ is an $n \times p$ matrix, of rank r

- U: columns are unit vector, and orthogonal = orthonormal*; $U^TU = I_n$ if $r = p$
- V: is the correlation (feature vectors??) = orthonormal*; $V^TV = I_p$ if $r = p$
- Σ is the scale; a diagonal** matrix of singular values of X in descending order; these are scale factors that stretch-compress unit vectors in U. *NB rank of X is also # of non-zero singular values*

**orthonormal: means that they preserve the length of vectors upon transformation, making them particularly useful for rotations and reflections in geometry and for orthogonal transformations in linear algebra.*

*** The diagonal nature of Σ (also D: only the singular values (the entries of Σ) affect the scaling in the transformation $X = U\Sigma V^T$, separating the rotation/reflection effects (handled by U and V V) from scaling (handled by Σ).*

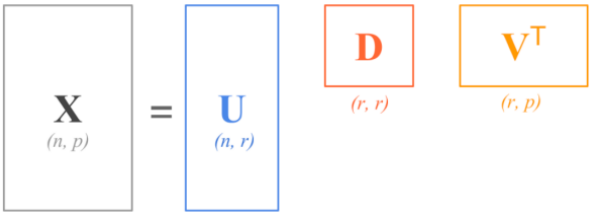


Figure 17: SVD dimensions

33.1 Properties of SVD

- if $U, D, V = svd(X)$, then $svd(X^TX) = VD^2V^T$
- if $U, D, V = svd(X)$, then $svd(XX^T) = UD^2U^T$
  This intuitively makes sense: X multiplied by its tranpose is sort of like squaring it.
- these provide the eigenvalue decompositons of X^TX and XX^T

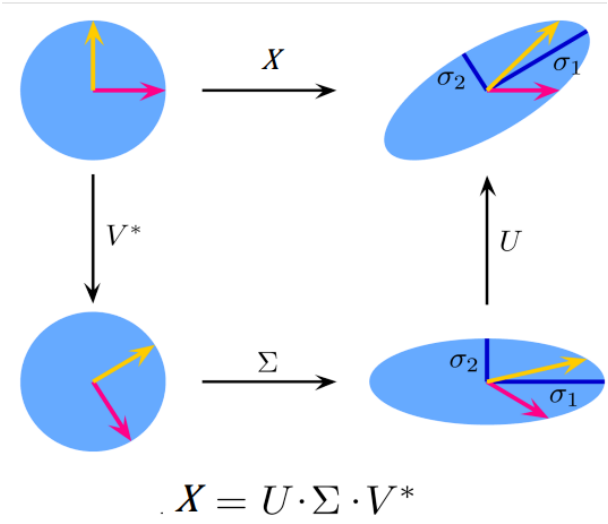


Figure 18: SVD transformations decomposed

The properties of SVD facilitate the eigenvalue decomposition of matrices like $X^T X$ and XX^T..

- revealing the underlying structure of the data and the relationships between variables
- but also provide a robust method for computing solutions in linear regression problems

By leveraging these properties, we can efficiently compute predictions for new data in a way that is stable **and resistant to issues like multicollinearity or singular matrices**.

We use these properties to show that:

$$X\beta_{ols} = \hat{} UU^T y$$

.

- i.e. that the predictions of a model...
- ... $= UU^T y$
- WHAT'S INTUITION HERE?

Also, that the prediction for a new $\tilde{X}$ is:

NB!

$$\tilde{X}\hat{\beta}_{ols} = \tilde{X}V^T D^{-1} U^T y$$

The SVD of X can facilitate this by providing a stable way to compute the **pseudo-inverse of** X, especially when $X^T X$ is not invertible or poorly conditioned.

34 Expected error in low vs high dimensional feature spaces

34.1 Standard Statistics: Bias-variance trade off

NB!

All of this is in under parameterised regime, before the interpolation threshold, where traditional statistical rules apply:

Low Dimensional Space: $p \ll n$ = OLS is BLUE

- Risk (Expected Error): $R(\hat{\beta}) - R(\beta^*) = \frac{p}{n}\sigma^2$.

As p increases, expected error increases.
As n increases, expected RAPIDLY error decreases.

So, we can reduce Generalisation Risk easily, by adding data.

Risk can be decomposed into:

- Bias: exact composition depends on loss function??? $\cdots \rightarrow$ low
- Variance exact composition depends on loss function ??? $\cdots \rightarrow$ low

High Dimensional Space: $p \gg n$ = OLS struggles (overfitting & poor generalisation)

- Risk (Expected Error): $R(\hat{\beta}) - R(\hat{\beta^*}) \approx (1 - \frac{n}{p})\|\beta^*\|^2$.

As p increases, expected error increases.
AS n increases, expected error SLIGHTLY decreases.

Risk can be decomposed into:

- Bias: $\approx (1 - \frac{n}{p})\|\beta^*\|^2 \cdots \rightarrow$ very large! (overfitting)
- Variance $\approx \frac{n}{p}???? \cdots \rightarrow$ very small HOW/WHY I THOUGHT COMPELX MODELS HAD HIGH VAR??????

So, adding further n doesn't do much. (Need regularization in high dimensional contexts: BUT THESE INTRO BIAS TO REDUCE VARIANCE.... SURELY THAT MAKES IT ALL WORSE???)

So, the same general dynamics in both, since we are not yet past the interpolation threshold: danger of too many predictors... but the *rates/ratios* between p and n are different

- low dim ($n \gg p$): increasing n RAPIDLY decreases generalisation error ($\frac{n}{p}\sigma^2$)
- high dim($p \gg n$): increasing n on the margin has MINOR effect $(1 - \frac{n}{p})$

At the interpolation threshold: dynamics change: $\frac{p}{n} \rightarrow \frac{n}{p}$

Under Paramaterised

- Low dimensional feature space: ↑ bias, ↓ variance
- High dimensional feature space: ↓ bias, ↑ variance.

Double Descent Curve @ perfect interpolation:

Over paramaterised

- High dimensional feature space:
  - ↓ bias, ↓ variance ??????????

Expected error over a new test point x_{new} can be analyzed in terms of bias and variance.

NB!

If

$$MSE = variance(\theta) + bias(\theta)^2$$

Is all error decomposible into variance & bias, but only the MSE loss function gives that specific decomposition????

34.1.1 Expected Error Decomposition

Expected prediction error for a new test point can often be decomposed into:

- **bias** - error introduced by approximating the real-world problem (which may be complex and nuanced) with a simpler model. A high-bias model makes strong assumptions about the form of the underlying function that generates the data, leading it to systematically misrepresent the data. For example, using a linear model for a relationship that is inherently non-linear would introduce bias.
- **variance** - error introduced by sensitivity to small fluctuations in the training set. A model with high variance pays a lot of attention to the training data and may capture noise as if it were a real signal, leading to poor performance on new test points.
- **irreducible** - error inherent in the problem itself, due to noise or other factors in the data generation process that cannot be eliminated by any model. It represents the lower bound on the error that any predictive model for the given task could achieve.

.
In ML: trade off between bias and variance:

- model with too much bias may not capture the important patterns in the data, leading to underfitting
- model with too much variance may capture noise rather than signal, leading to overfitting.

(From decomposition above): variance component in particular can be represented as a sum over the variance of the predicted values, which depends on the covariance of the feature vectors. ?????

34.1.2 Variance of New Test Point

NB!

Expected Error: $R(\hat{\beta}) - R(\beta^*) = \frac{p}{n}\sigma^2$

If x_{new} is drawn from the same distribution as the training data X, then the variance of the predicted values for x_{new} can be linked to the covariance matrix of the feature vectors (Σ).

Formally:

- consider the expected error over a new test point, $\tilde{x}$:
- from above: expected error given as $E[(\tilde{x}(X^TX)^{-1}X^Te)^2]$?????????
- so the expected error over new test point: $E[(\tilde{x}(X^TX)^{-1}X^Te)^2]$
- using SVD, this can be written as $E[(\tilde{x}V^TD^{-1}U^Te)^2]$
- ...there's a bunch more work where they all get vectorized(??)...
- ... final derivation: $R(\hat{\beta}) - R(\beta^*) = \frac{p}{n}\sigma^2$
  - this is relationship between expected risk of estimated coefficients in linear regression model vs the true coefficients,
  - and that relationship expressed as ratio of predictors vs observations (ie model complexity vs data)
  - and variance of the error terms
- $\frac{p}{n}\sigma^2$ essentially states that the difference in expected risk between using the estimated coefficients and the true coefficients is proportional to the ratio of the number of predictors to the number of observations, multiplied by the variance of the error terms.
  - increase complexity (features): increase difference in risk of estivated vs true model = overfitting.
  - increasing sample size: reduces risk difference = improving generalization capability. i.e. importance of sufficient data, esp as model complexity increases.
  - impact of model complexity and sample size on the difference in risk is scaled by the variance of the error. Higher error variance means that the discrepancy in risk due to estimation inaccuracies will be more pronounced.

NB!

$\frac{p}{n}\sigma^2$ provides a concise quantification of how the expected performance of a linear regression model degrades with increasing complexity (more features) relative to the amount of data available, and how this degradation is influenced by the inherent noise in the data (σ^2).

34.2 Low Dimensional Features $n \gg p$:*

IS 'LOW DIMENSIONALITY THE STANDARD FOR MOST STATS, SO THIS IS THE WORLD WE'VE BEEN IN SO FAR, WHERE INCREASING COMPLEXITY THROUG ADDING PARAMETER VS AMOUNT OF DATA IS PROBLEMATIC AND LEADS TO OVERFITTING; WHEREAS THE 'HIGH DIMENSIONALITY' WORLD IS THE WORLD OF DEEP

LEARNING, AND WE GET THE DOUBLE DESCENT - IE THE USUAL RULES/RELATIONSHIP BETWEEN P & N REGARDING GENERALIZATION STOP APPLYING????
*$\gg$ = *'much greater than'*

low # of features cf # of observations:

Due to $\frac{p}{n}\sigma^2$, in low dimensional settings, models tend to have **higher bias** but **lower variance**

- prediction error primarily reflects the model's simplicity,
- which might not capture all complexities of the data (leading to higher bias)
- but shows stability across different test points (leading to lower variance).

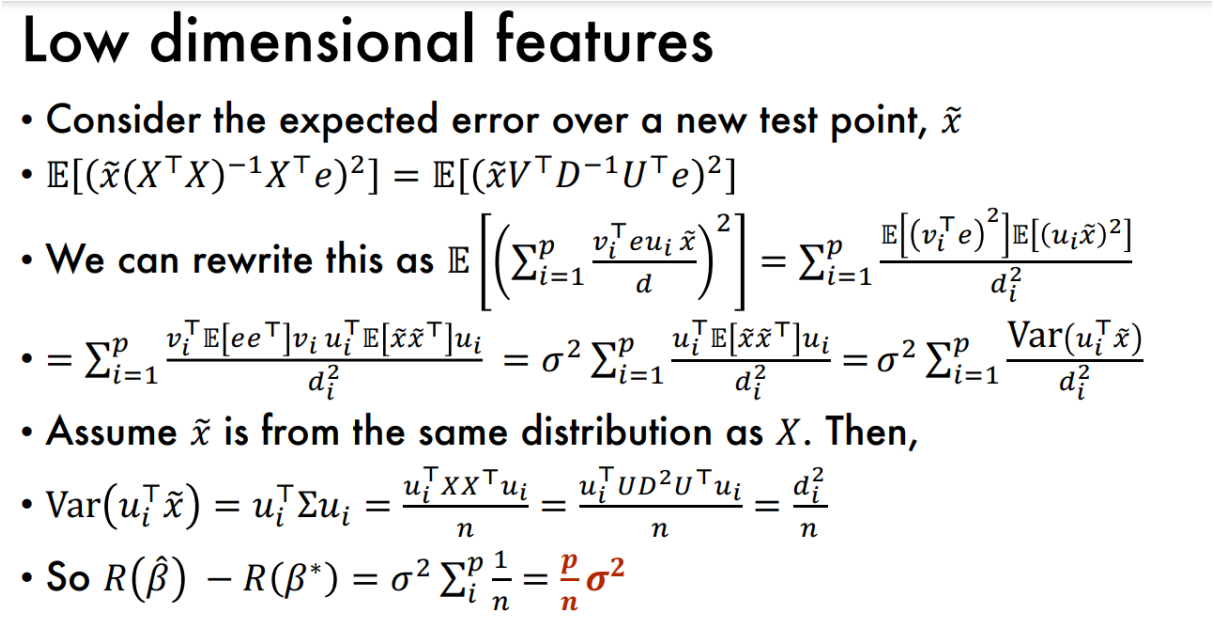


Figure 19: Enter Caption

34.3 High Dimensional Features ($p \gg n$):

In high-dimensional settings, traditional intuitions about model performance and generalization can fail. High dimensionality poses unique challenges, such as increased risk of overfitting, as models have enough parameters to perfectly fit the training data, potentially capturing noise as if it were a true signal.

Bias models extremely flexible $\rightarrow$ v low training error, but v large bias: I THOUGHT BIAS WAS LOW????? BUT THAT VARIATION WAS HIGH

- formally: $(1-\frac{n}{p})\|\beta^*\|^2$ = v large
- overfitting noise $\rightarrow$ generalization error
- *Bias refers to the error introduced from erroneous assumptions in the learning algorithm. High bias can cause an algorithm to miss the relevant relations between features and target outputs (underfitting(????)).*

Variance: NB THIS IS CHAT GPT, DREW'S SLIDES ACTUALLY SAY VARIANCE: 'VERY SMALL' high - small changes in training data → big changes in model

- Refers to the error introduced by sensitivity to small fluctuations in the training set. High variance can cause an algorithm to model the random noise in the training data (overfitting)
- a reflection of model's overfitting to the training data
- Bias $(1 - \frac{n}{p})\|\beta^*\|^2$ = v large
- Variance $\frac{n}{p}$ = v small

I THINK THE IDEA IS THAT IN THE DOUBLE DESCENT THE BASIC RELATIONSHIPS BETWEEN N & P CHANGE FROM $\frac{n}{p} \to \frac{p}{n}$

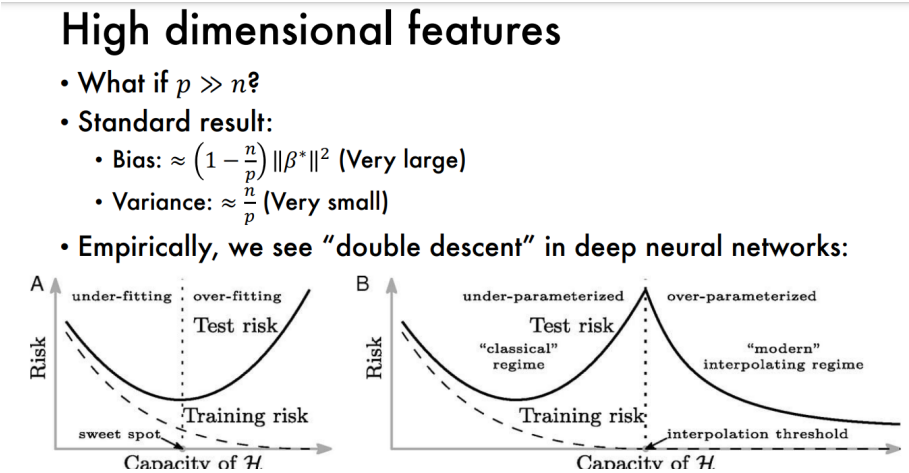


Figure 20: Enter Caption

34.3.1 Double Descent Phenomenon

Empirical observation: in deep netural nets: double descent curve.

Occurs when increasing model's complexity beyond point of interpolation.

ML Lecture Notes: Wk 5 — Benign Overfitting & Kernels

35 Recap

Relationship between risk, loss, bias:

- Risk = expected loss
- Loss function determines risk
- when we take MSE as our risk definition (L2 norm as loss function), then ouR risk (errors) decomposes nicely into bias and variance.
- But this might not always be the case

The generalization OLS: how well the OLS estimator performs in terms of predicting new, unseen data. This performance is measured through the lens of bias and variance, as well as the risk or the expected prediction error.

35.1 $p \ll n$

Many more observations than predictors: OLS works great, just add n and risk rapidly approaches 0.

It represents a more traditional case where OLS estimates are often unbiased and the estimators have the least variance among all linear unbiased estimators (thanks to the Gauss-Markov theorem).

- $R(\hat{\beta}) - R(\hat{\beta}^*) = \frac{p}{n}\sigma^2$
- The risk $R(\hat{\beta}) - R(\hat{\beta}^*)$, which represents the difference in prediction error between the estimated coefficients ($\hat{\beta}$) and the true coefficients (β^*), rapidly approaches 0 as n increases
- NB: an assumption - requires that SVD of training X = SVD of test $\tilde{X}$
  - implies that the geometry or structure of the data (in terms of its variance-covariance matrix) is consistent between training and testing sets.
  - This is an idealized assumption and often not met in real scenarios,
  - indicating that in practice, some form of regularization or model validation is necessary to ensure good generalization (ie that the rate of error reduction against n given above does not hold).

35.2 $p \gg n$

This scenario is more common in modern machine learning problems, where the dimensionality of the data (number of predictors) is much larger than the number of observations. This situation poses unique challenges for OLS, leading to overfitting and poor generalization if not addressed properly.

- Risk (Expected error): $R(\hat{\beta}) - R(\hat{\beta}^*) \approx (1 - \frac{n}{p})\|\beta^*\|^2$
  This can be decomposed into Bias + Variance:
- Bias $\approx (1 - \frac{n}{p})\|\beta^*\|^2$ (very large)
  - in high-dimensional settings, OLS estimates can be biased.
  - This is contrary to the low-dimensional case.
  - The bias can be very large because the model tries to fit the noise in the training data due to having too many degrees of freedom.
- Variance $\approx \frac{n}{p}$ (Very small)
  - because the model is highly constrained by the data—it has many predictors to "explain" the variability, leaving little room for the estimates to vary across different samples.
- **On the margin, sample size doesn't help that much:** when p is large, just boosting n a bit (on the margin) does not help.
  - prediction error doesn't significantly improve with more data because the model's complexity (number of parameters) is too high relative to the amount of available data.
  - The approximation $(1 - \frac{n}{p})$ shows that as the ratio of the number of observations to the number of predictors decreases, the benefit of additional data diminishes.

35.3 Implications

- The generalization of OLS is fundamentally affected by the ratio of predictors to observations.
- In low-dimensional settings ($p \ll n$), OLS generally performs well with risks decreasing as more data becomes available.
- However, in high-dimensional settings ($p \gg n$), traditional OLS struggles due to large bias and a lack of significant improvement from additional data.
- This has led to the development of techniques such as regularization (e.g., ridge regression, lasso) to address overfitting and improve model generalization in high-dimensional contexts.
- These methods introduce bias deliberately to reduce variance and improve the model's predictive performance on new data. BUT THESE INTRO BIAS TO REDUCE VARIANCE.... SURELY THAT MAKES IT ALL WORSE???

36 Double Descent Puzzle

- As the complexity of deep neural networks increases, we eventually see decreases in test-error!
- Intuitively, we might expect that at some point, increasing complexity could lead to overfitting, where the network memorizes the training data, including noise, resulting in poor generalization to unseen data (high test error).
- This counterintuitive behavior suggests that very deep and complex networks have a regularizing effect that improves their ability to generalize, rather than merely memorizing the training data.

- One explanation: **manifold hypothesis**: structure in high dimensional data makes it behave low dimensionality
  - high-dimensional data (such as images, text, or audio) often lies on or near a much lower-dimensional manifold.
  - A manifold is a mathematical space that might locally resemble Euclidean space but has a more complex, curved structure when viewed as a whole.
  - For instance, the surface of the Earth is a two-dimensional manifold that exists in three-dimensional space.
  - the intrinsic dimensionality of the data is much lower than the ambient (high) dimensionality.
  - This lower-dimensional structure implies that the data points are not randomly dispersed throughout the high-dimensional space but are constrained in a way that reflects meaningful relationships (e.g., similarities or categories) within the data.
- Deep neural networks, especially those with high complexity, are exceptionally good at discovering and exploiting the low-dimensional manifolds in high-dimensional data spaces.
- Through the process of training, these networks learn to ignore the irrelevant dimensions (noise) and focus on the manifold's structure, effectively regularizing themselves.
- it is the underlying structure that gives us the ability to learn
- This ability to recognize and adapt to the data's manifold structure is what enables complex DNNs to achieve lower test errors, as they're not merely fitting to noise but are capturing the underlying patterns that generalize well to unseen data.

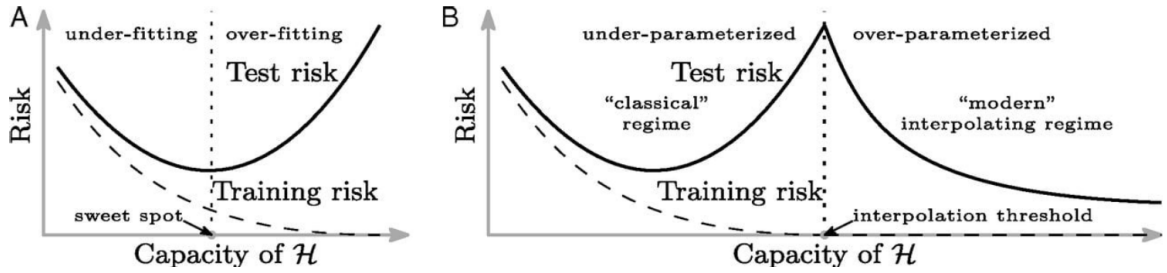


Figure 21: Double Descent

37 Modern Analysis of OLS k splits -> "Benign Overfitting"

= a sophisticated strategy for dealing with high-dimensional data in linear regression contexts.

- Conceptually we are decomposing into 2 distinct parts to the problem, based on the dimensionality of the feature space):
- $\beta^* = \left[\beta^*_{0:k}, \beta^*_{k:p}\right]$ where k "splits" our features into low vs high dimensional parts
- when $p \approx n$ this is the **interpolation threshold: 0 training error**

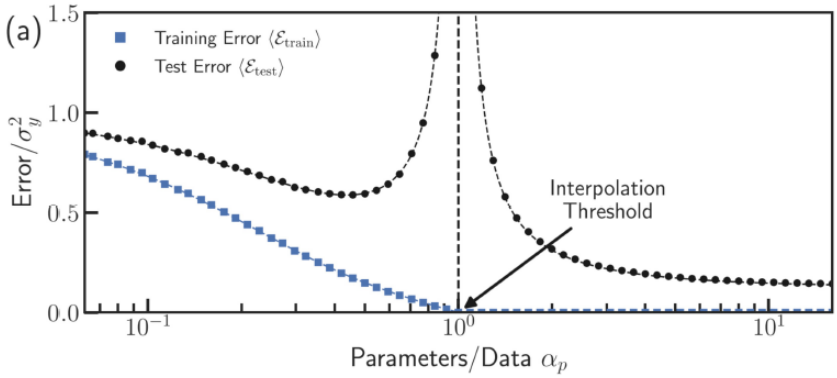


Figure 22: Interpolation Threshold

37.0.1 Low Dimensional Part: $\beta^*_{0:k}$

This part represents the coefficients of the first k features in your model. The idea here is that these features are in a "low-dimensional" space relative to the total number of observations (n).

In traditional statistical settings, having a lower number of features compared to the number of observations helps in achieving models that are generalizable and less prone to overfitting.

37.0.2 High Dimensional Part: $\beta^*_{k+1:p}$

This part represents the coefficients of the features from $k+1$ to p where p is the total number of features.

This segment of the model dwells in the "high-dimensional" space, especially when p is close to, or exceeds, n

.

The concept of an "interpolation threshold" becomes particularly relevant when $p \approx n$. At or near this threshold, every observation can potentially be "perfectly" fit by the model, leading to a scenario where the training error can be zero.

This is sometimes referred to as the model entering an "interpolation regime" or "overparameterized regime," where the number of parameters (or model complexity) is high enough to fit all training points perfectly.

when models are sufficiently complex to enter the interpolation regime, they may still capture underlying patterns that are generalizable, rather than merely memorizing the training data

In high-dimensional settings, the behavior of OLS and other statistical estimators needs to be reconsidered. Traditional assumptions and interpretations may not directly apply. For example, the bias-variance tradeoff, a cornerstone in understanding model performance, has nuances in high-dimensional contexts. Regularization techniques, dimensionality reduction strategies, and other methodological adaptations become crucial in managing the challenges posed by high-dimensional data.

37.0.3 How does this work? i.e. how is this conceptual split engineered? What is the mechanism?

When $p \gg n$, we cannot estimate $\beta = (X^T\hat{X})^{-1}X^Ty$.
when $p > n$

- the covariance matrix X^TX is not invertible
- the covariance matrix X^TX has singularity problems

Instead, we take the SVD of X, which is guaranteed to exist:

$$X\beta = \hat{U}U^Ty$$

- NB: the above is specifically on OLD data X; you don't actually have to calculate the $\hat{\beta}$, instead SVD derives a linear project onto X (or is this y???), which is similar to what regression does. So for any sample points, you don't have to derive the data(???)
- We take the **pseudoinverse** of X (?), leading to alternative formula:

$$X = V\Sigma^{-1}U^Yy$$

- This circumvents the problem of X^TX being singular.

Then, we truncate dimensions of U associated with v small singular values = introduces a regularization-like approach:

$$\beta^* = [\beta^*_{0:k}, \beta^*_{k:\infty}]$$

- by retaining only first k singular values (and corresponding vectors), we essentially perform dimensionality reduction, focusing on the components of X that capture the most variance in the data.

This is a kind of practical application of the manifold hypothesis:

- that the essential info resides in lower-dimensional structure within the high-dimensional space.
- that the first k singular values explain everything important.

The decomposition of β into low-dimensional and high-dimensional parts distinguishes between signal and noise in the data.

- Features in the low-dimensional part may be associated with stronger signals *(high structure)*,
- while the high-dimensional part may include features that add complexity but not necessarily predictive power, unless carefully managed. *(There isn't much structure there; it doesn't explain much)*

This works well / is possible for highly structured data. If highly structured:

- then the low dimensional values have high singular values, meaning they matter
- we **reshuffle** the data using SVD, so that the most important components/features are at the front
- SVD just allows us to look at out data so that each column is independent of all others, and the most important are queued first

- SVD is central to the regression, it is doing the shuffling and splitting for us

So these are two complimentary formal approximations/interpretations of the mainifold hypothesis:

1. **Decomposition of β^* (conceptual)** (into $\beta^*_{1:k}, \beta^*_{k+1:p}$) mirrors idea of splitting solution into parts that correspond to retained and discarded singular values.
    - a prioritization of features or dimensions that are deemed most informative or relevant
    - the parts of the model residing in low dimensionality can behave like OLS when $p \ll n$
2. **Truncation of U (actual mechanics)**: keeping only first k singular values (and correpsonding columns of U and V implies approximating X by its most significant components.
    - this reduces complexity of the model, mitigating overfitting
    - by disregarding dimensions that contribute less to the variance in the data

the split is *within the SVD*

The idea

- $\beta^* = [\beta^*_{0:k}, \beta^*_{k:\infty}]$
- $\beta^*_{0:k}$
    - The low dimensional part
    - This can behave like OLS when $p \ll n$
- $\beta^*_{k:\infty}$
    - The high dimensional part is assumed to be zero
    - This can be excluded as unimportant through the manifold hypothesis
    - It won't overly bias results because it behaves like white noise
- Critical: this split is *within the SVD*
    - Large singular values are in $\beta^*_{0:k}$, while small ones are in $\beta^*_{k:\infty}$

Figure 23: Enter Caption

*NB: for this to work, we have to make the the **assumption** that the high dimensional features behaves like white noise: moving in all directions $\rightarrow$ won't overly bias the data.

In the context of regression, the SVD of X allows us to rewrite the regression equation as $\beta^* = UU^T y$. This has 2 benefits:

- allows us to to run regression when $p > n$ (bypasses singularity of $X^T X$)
- also allows us to do dimensionality reduction

So SVD opens up double pathway for dealing with high dimensionality data effectively.

The critical insight here is the role of SVD in enabling a systematic way to identify and retain the most informative aspects of the data while discarding the rest as noise. This technique leverages the inherent structure of the data, as revealed by the SVD, to make intelligent decisions about which dimensions (i.e., singular values and their corresponding vectors) are worth keeping. This approach not only helps in managing the curse of dimensionality but also in potentially enhancing the model's interpretability and generalizability by focusing on what truly matters in the data.

37.1 "Benign overfitting" for $p \gg n$

"Benign overfitting" suggests a scenario where, despite overfitting, a model can still generalize well on unseen data.

This is how the split described above actually is operationalised.

Risk is bounded by 1) residual variance and 2) a constant penalty:

1. residual variance = inherent noise in the data that cannot be explained by the model.
2. a constant penalty = reflects the complexity of the model. However, unlike traditional settings where complexity grows with more features, leading to increased risk of overfitting, here the penalty can remain manageable. This suggests that the additional complexity introduced by many features does not necessarily compromise the model's predictive performance

this is risk???:

$$\frac{\sigma^2}{c}\left(\frac{k^*}{n}+\frac{n}{R_{k^*(\Sigma)}}\right)$$

Where:

- residual variance
- some constant penalty
- the parametric rate: ie. the $\frac{p}{n}$ before for the $0:k$ part
- the effective number of dimensions on the $k:\infty$ part. It decides the ideal split for us(??).
  - $R_{k^*}(\Sigma)$ is a technical rank condition of Σ - a measure of the spread of the singular values.
  - It is a measure of the effective number of dimensions in this high-dimensional part
  - as each dimension is orthogonal in SVD, when we have a high number here, it means the high dimensional part moves in many different directions; ie. it looks random / it lacks structure.
  - this is the white noise assumption from earlier
  - if we have a lot of different dimensions, then we can effectively estimate 0 for the coefficients in this part

NB: in low dimensional OLS risk is given as $\frac{p}{n}\sigma^2$; which is here given by the green and the red components.

37.1.1 Parametric Rate for the $0 : k$ part

The notion of a parametric rate being essentially maintained for the initial $0 : k$ part of the model indicates that for the lower-dimensional, significant aspects of the model (those associated with the largest singular values), the traditional understanding of risk and generalization holds.

This part of the model behaves in a predictable manner, where increasing model complexity (in a controlled fashion) does not unduly inflate the risk.

??

37.1.2 Effective Number of Dimensions in the $k : \infty$ part

- so long as the high dimensional part of X hits a lot of different directions, then it won't ruin the model.
- This is influenced by the rank condition on Σ, which is a measure of the spread of the singular values.
- If Σ (representing the structure of the data in feature space) has a wide range of singular values, this indicates a diverse spread in the directions covered by the features.
- The spread of singular values $R^*(\Sigma)$ captures how well the high-dimensional space is 'utilized' by the data.
- a broad spread suggests that the data, despite being high-dimensional, does not concentrate in a few directions but rather spans multiple dimensions effectively.

37.1.3 Implications for High-Dimensional Models

As long as the high-dimensional part of X covers a wide array of directions, the risk of overfitting, though present, does not necessarily impair the model's ability to generalize.

Because the diversity in directions can help in distributing the model's complexity across many dimensions, rather than concentrating it in a way that makes the model sensitive to noise in the training data.

Benign overfitting offers a counterintuitive perspective where high-dimensional models can still perform well, provided that the data's structure and the model's complexity are managed appropriately. This underscores the importance of understanding the underlying geometry and distribution of the feature space in high-dimensional data analysis.

37.2 Demo

See script

My notes:

- the problem is that all the singular values are basically the same -> so we are not finding the structure -> so when we clip it aggressively, we haven't isolated the structure
- the weighting: the high frequency features of the cosine function are weighted down -> we are saying we think the low dimensionality features are more important

- now function: we are perfectly fitting to the white noise - the 1/c cost (???)... the point is that it is fitting to the noise, but it is NOT ruining the structure...rather it is is finding the structure here
- where the c constant comes from is deep maths, don't bother trying to work out where this comes form.
- for the weighted features comparison:
  - the dotted line is the minimal error: it is if you had 3 features
  - - if you had
  - in the low dimensional case it doesn't matter if you do this feature weighting
  - - in the higher dimensions it matters
- by weighting the features you are putting in a preference for smoother functions
- you are saying you prefer functions that use lower dimensionality features

37.3 Takeaway from "Benign Overfitting"

NB!

1) IF your data is well explained by low intrinsic dimension
2) AND the high dimensional part behaves like white noise (that the high dimensional points are pointing all different directions)
3) THEN your generalization in the interpolating regime can be "as if" you just ran the the parametric model on the intrinsic dimension.
+ this happens automatically when you fit OLS with SVD

- In contexts where the feature space dimensionality $p \gg n$:
- **Low Intrinsic Dimension if Key** -> model can generalise!
  - despite existing in high-dimensional space p,
  - the meaningful variation can be captured by a much smaller number of dimensions k,
  - data's essential structure can be effectively summarized without resorting to the full complexity of the high-dimensional space.
- **High-dimensional part as White Noise**:
  - non-meaningful variance
  - adds randomness w/o affecting underlying patterns that the model aims to learn
- **Generalization in the Interpolating Regime**
  - where the model fits the training data perfectly ($p \gg n$): the ability to generalize **"as if"** the model were trained only on the intrinsic dimensions is a powerful outcome.
  - implies that despite the high-dimensional nature of the feature space, the model can achieve generalization performance comparable to what would be expected if it were operating in a lower-dimensional space defined by the data's intrinsic dimensions.
  - This phenomenon underscores the importance of the structure and quality of the data over sheer dimensionality.
- **OLS with SVD as an Automatic Mechanism**

- fitting OLS using SVD does all this automatically.
- SVD allows the model to effectively discern and leverage the intrinsic dimensionality of the data by focusing on the significant singular values and their corresponding singular vectors.
- method essentially filters out the "noise" represented by the lesser singular values
- model benefits from the reduced complexity and enhanced generalization capability that comes from operating within the data's intrinsic dimensional space.

In summary, the ability to generalize well in high-dimensional settings, even when models fit the training data perfectly, is significantly influenced by the intrinsic dimensionality of the data and the distribution of variance across dimensions.

ML Lecture Notes: Wk 6-I — Kernels

Kernels: new way to think about measuring similarity

Feature expansion: the concept of expanding data into a space where linear separability is achievable suggests that the space is highly dimensional (??)

38 Alternative view of regression

- **Trad**: we ascribe coefficients to features –> linear combination / projection of features, weighted by parameter coefficients
- **Alt**: considering weighting the units based on their proximity (determined by their covariance) to the point of interest (or target variable). These weights are basically coefficients for units, providing a linear combination to create a new synthetic unit based on features.
  - This approach hints at techniques like k-nearest neighbors (kNN) or kernel smoothing, where the idea is to weight observations (or "units") based on how similar (or "close") they are to the point where we're making a prediction.
  - This similarity can be measured in various ways, such as Euclidean distance or, as mentioned, covariance.
  - The alternative perspective highlights geometric and relational aspects of the features and units: understanding the data in terms of its structure and relationships among observations.
  - Such a perspective is crucial in methods that rely on distance or similarity measures, as it considers the arrangement and interaction of data points within the feature space.
- **Alt prediction:** in weighted regression / kernel smoothing - the prediction at a new location $\tilde{X}$ is a linear combination of all training labels weighted by their similarity/distance to $\tilde{X}$

Here we take ridge regr through the lens of similarities within units, rather than between features. Rewriting ridge regression using identity:

$$\begin{aligned} X\hat{\beta}_{ridge} &= X\underbrace{(X^TX + \lambda I_p)}_{p\times p}{}^{-1}X^Ty \\ &= XX^T\underbrace{(XX^T + \lambda I_n)}_{n\times n}{}^{-1}y \end{aligned}$$

If you have low set of features (p is small), there's no reason you'd want to use the red formulation here - you'd stick with the trad blue.

But if we do feature expansion, then we get big p -> then we might want to work with n matrix instead.

Implications of X^TX and XX^T:

- X^TX: similarity between features - dot product between column vectors.
- XX^T: similarity between units - dot product between row vectors. Important to understand similarity / distance between data points.

38.1 Why are X^TX and XX^T similarities?

Define Euclidean Distance, given by:

$$\begin{aligned} d^2(x,y) &= \sum_{i=1}^{n}(x_i - y_i)^2 \\ expand &= \sum_{i=1}^{n} x_i^2 - 2x_iy_i + y_i^2 \\ separate\ sums &= \sum_{i=1}^{n} x_i^2 - 2\sum_{i=1}^{n} x_iy_i + \sum_{i=1}^{n} y_i^2 \\ in\ linear\ algebra &= x^Tx - 2x^Ty + y^Ty \end{aligned}$$

Relationship between Distance & Similarity Distance and similarity are inversely related: the smaller the distance, the greater the similarity, and vice versa.

By manipulating the formula for squared Euclidean distance, we can express a relationship that looks similar to the formula for calculating cosine similarity, which is a direct measure of similarity between two vectors.

Dot product refresher:

$x \cdot y =< x, y >= x^Ty = \sum x_i, y_i$

- $x \cdot y$ (also $< x, y >$) takes 2 equal-length vectors -> yields 1 scalar that represents magnitude of one vector projected onto another
- helps understand the relationship between two vectors in terms of their magnitude (length) and direction.
- $x \cdot y = \sum x_i, y_i$ - i.e. summing the products of corresponding elements.
- $x \cdot y = x^Ty$ in linear algebra terms

dot product is crucial to computing squared Euclidean distance & cosine similarity.

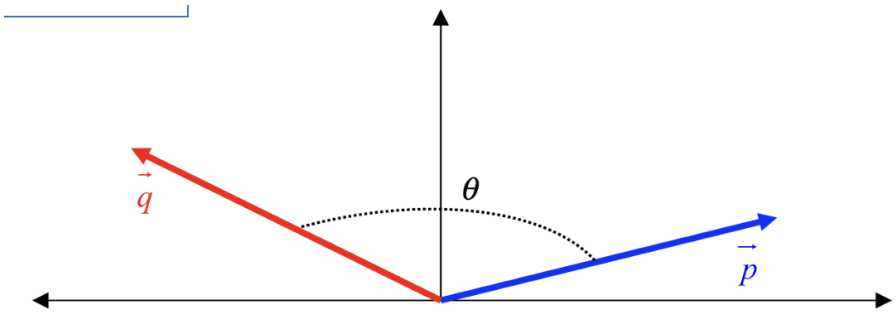


Figure 24: dot product

Cosine Similarity
Cosine similarity is a metric used to measure how similar two vectors are, regardless of their magnitude. It is calculated as the cosine of the angle between the two vectors in an n-dimensional space. The formula for cosine similarity is:

$$\text{cosine similarity} = \frac{x \cdot y}{||x||\,||y||}$$

where:

- $x \cdot y$ is the dot product of vectors x and y,
- $||x||$ and $||y||$ are the norms (magnitudes) of vectors x and y, respectively.

Cosine similarity ranges from -1 to 1, where 1 indicates that the vectors are in the same direction (very similar), 0 indicates that they are orthogonal (no similarity), and -1 indicates that they are in opposite directions (very dissimilar). This measure is widely used in text analysis, recommendation systems, and other applications where the orientation of vectors (indicating similarity in terms of direction) is more important than their magnitude.

Dot product as measure of vector similarity (magnitude AND orientation)

$$x \cdot y = ||x||\,||y|| \cos(\theta)$$

explains dot product in terms of vector magnitudes and the cosine of the angle between them, where:

- $||x||$ and $||y||$ are the norms (magnitudes or lengths).
- $\cos(\theta)$ is the cosine of the angle θ between the two vectors.

Breaking Down the Formula

1. **Norm of a Vector ($||x||$ and $||y||$):**
   - measure of its length.
   - For a vector x in an n-dimensional space, represented as $x = (x_1, x_2, ..., x_n)$,
   - its norm is calculated using the formula $||x|| = \sqrt{x_1^2 + x_2^2 + ... + x_n^2}$.
2. **Cosine of the Angle ($\cos(\theta)$):**
   - cosine function of the angle θ between vectors x and y.
   - a measure of the directional similarity between the two vectors.
   - When angle is 0 degrees: vectors are pointing in the same direction: $\cos(0) = 1$.
   - When angle is 90 degrees: vectors are orthogonal: $\cos(90°) = 0$.
3. This shows that the dot product can indicate not only the magnitude of the vectors but also how they are oriented with respect to each other.

Interpretation

- **When vectors are parallel**
  - $\theta = 0$, and $\cos(\theta) = 1$.
  - dot product $x \cdot y = ||x||\,||y||$ reaches its maximum value, equal to the product of their lengths, indicating maximum similarity.
- **When vectors are perpendicular**
  - $\theta = 90°$, and $\cos(\theta) = 0$.
  - dot product $x \cdot y = 0$, indicating that the vectors are orthogonal and have no 'overlap' in any dimension.
- **When vectors are anti-parallel**, $\theta = 180°$, and $\cos(\theta) = -1$. The dot product $x \cdot y = -||x||\,||y||$, which is the negative of the product of their lengths, indicating that the vectors are in opposite directions.

TL;DR:

- **dot product is the length of q vec and p vec times *cosin* of angle between them.**
- **if similar: if angle is small -> cosin large -> dot product large**
- **if different: as angle grows -> cosin shrinks -> dot product shrinks**

38.2 X^TX & Cosine Similarity

- X^TX is a matrix where each element represents dot product between pairs of column vectors in X
  - dot product: $x \cdot y = ||x||\,||y|| \cos(\theta)$ - measures similarity (magnitude AND direction)
  - cosine similarity: $= \frac{x \cdot y}{||x||\,||y||}$ - measures directional similarity (regardless of their magnitude)
- taking the reformulated definition of Euclidean distance from above: $(= x^Tx - 2x^Ty + y^ty)$, where
  - $-2x^Ty$ is the dot product between x, y
  - if we normalised our columns before feeding into this formula: the x^Tx and y^Ty values become 1
  - so the only thing we are left with is the inner dot product (WHY????)
- putting these together, we get:

$$\begin{aligned}(X^TX)_{ij} &= -\frac{1}{2}\left(d^2(x_i, x_j) - \|x_i\|^2 + \|x_j\|^2\right) \\ &= x_i^T x_j \\ &= \|x_i\| \cdot \|x_j\| \cdot \cos(\theta_{ij})\end{aligned}$$

NB the $-\frac{1}{2}$ comes from the fact similarity is the inverse of distance

- this rephrases the squared distance in terms of vector norms and their dot product...
- ...showing that the dot product (and thus cosine similarity) can be interpreted through the lens of modifying the Euclidean distance formula.

For some reason, this means that:

- $(XX^T)_{ij}$ = covariance between units (matrix of dot products of the row vectors)
- $(X^TX)_{ij}$ = covariance between features (matrix of dot products of the column vectors)

38.3 Similarity's Relationship to Covariance: differentiated by structure

Identity: $cov(x, y) = E[xy] - E[x]E[y]$

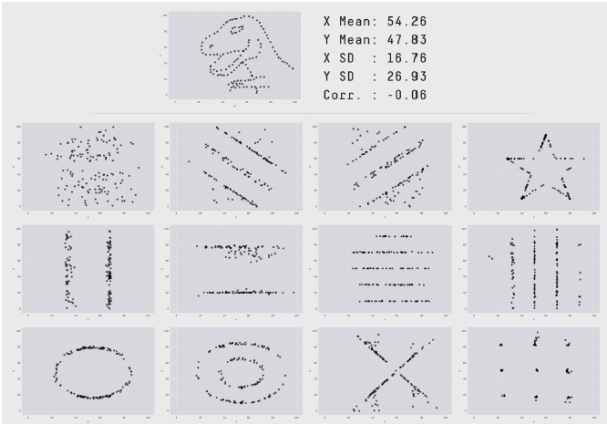


Figure 25: Covariance Issues

- All the above have same correlation, but very different structures
- Our measure of similarity determines the kinds of functions we can learn
- is linear correlation (over X) the right way to think about similarity? if all you take is correlations, then you will be estimating the same covariance between Xs for all of the above images
- most important thing in ML is to define a good definition / conception of distance for the problem.

  OR IS COVARIANCE ALWAYS ASSOCIATED WITH EUCLIDEAN DISTANCE, SO THE HIGHER LEVEL OF ABSTRACTION IS TO THINK ABOUT SIMILARITY RATHER THAN DISTANCE?

NB!

IF YOU DEFINE DISTANCE DIFFERENTLY -> WILL GET DIFFERENT MEASURES OF COVARIANCE BETWEEN UNITS / FEATURES...

... 2 DATA POINTS THAT ARE CLOSE IN EUCLIDEAN DISTANCE (CO-VARY TOGETHER), WHEN YOU REENGINEER THE FEATURE SPACE THRU KERNELS / CONSIDER DIFFERENT MEASURES OF DISTANCE, MIGHT NO LONGER COVARY IN THE SAME WAY (??)

COVARIANCE = A MEASURE OF HOW CLOSE THEY ARE, BUT IT IS BASED UPON DEFINITION OF DISTANCE, IF YOU CHANGE THAT DEFINITION, THE MEASURED RESULT WITH CHANGE

38.4 Definitions of distance in ML

We need to think about how to define a suitable measure of distance to model relationships within data effectively.

This "distance" helps the algorithm determine how similar or different the data points are from one another.

38.5 Distance in OLS

- OLS finds the line (or hyperplane in higher dimensions) that best fits the data by minimizing the sum of the squares of the vertical distances (residuals) of the points from the line.
- This method implicitly uses Euclidean distance, which is the straight-line distance between two points in space
- so the similarity matrixes (in OLS' case: covariance matrixes) give you squared distance
- In our alternative view of regression (see above), when we make prediction at a new location $\tilde{X}$, OLS essentially uses a linear combination (a weighted average) of all training labels (known outputs in the training data).
  - re writing ridge regression estimator: $\tilde{X}X^T(XX^T + \lambda I_N)^{-1}y$
    - This expression gives you new prediction point
    - But, we are turning the idea of linear regression on its head, and thinking of the weights over units, rather than the weights over features
    - $\tilde{X}X^T$ - similarity between test & training rows
    - XX^T - similarity between rows
    - so with $\tilde{X}X^T$ we are in effect taking a walk to find the Xs that are most similar to the new $\tilde{X}$ - this weights all the labels for each observation in the training rows (X), based on its similarity/distance to the test rows ($\tilde{X}$).
  - this can be intepreted as a weighting estimator:
    - $W = \tilde{X}X^T(XX^T + \lambda I_N)^{-1}$(NB: no y)
    - W is an $n \times n$ matrix (where $\tilde{n}$ is the number of test points)
    - each row of W represents the weights for one label in the test data
    - This approach can be seen as weighting the training labels (y) by how similar the test point ($\tilde{X}$) is to each training point.
    - The weights (W) are determined by how similar the training data points are to each other (X^TX) and to the new data point ($\tilde{X}$), WHY IS IT IMPORTANT HOW SIMILAR THEY ARE TO EACH OTHER, WHY NOT JUST HOW SIMILAR THEY ARE TO THE NEW DATA????
    - (adjusted by the regularization term (λ), which prevents overfitting).
  - $\hat{y}(\tilde{x}_i = \sum_{j=1}^{n} W_{ij}y_j$
    - the weight here is large when we are using a lot of information from a particular training point.

we're not limited to linear models or Euclidean distance. By choosing different ways to measure distance or similarity (e.g., through kernels or other distance metrics), we can create models that capture more complex relationships in the data.

Constraint of OLS
In OLS, the effective weight of training points evolves purely linearly with respect to X.

- in a linear model, the weights evolve linearly with distance from the test point
- because of this linearity, in the demo: for $\tilde{x} = 2$ the unit $x = 3$ is weighted more heavily than $x = 2$...
- ... if we could incorporate flexibility to emphasise weights around $x = 2$

38.6 Non-linear distance measures

This is done through feature expansions (WHAT IS CONNECTION BETWEEN NON-LINEAR DISTANCE MEAUSRES AND FEATURE EXPANSIONS?

- Feature expansion, refers to the process of increasing the dimensionality of the feature space by creating new features from the existing ones, making the model capable of capturing more complex patterns within the data.
- Essentially, feature expansion allows linear models to fit non-linear relationships by transforming the original feature space into a higher-dimensional space where a linear separation (in the case of classification) or a linear relationship (in the case of regression) becomes feasible.

We can refer to these functions as the base features, $\phi(X)$, where ϕ is a function that expands a 1D feature space into an nD space.

We've been doing this already:

- Polynomial:
  - $\phi(X) = [1, X, X^2, \cdots, X^d]$ for some degree d
- Trigonometric:
  - $\phi(X) = [1, cos(X), cos(2X), \cdots, cos(dX)]$ for some degree d
  - NB as d increases for trigonometric functions, the frequency increases.

When we expand the feature space of our data through transformations like polynomial or trigonometric expansions, we're essentially creating more complex models to capture the underlying patterns within the data more effectively.

However, as the degree of these expansions (d) increases, especially as $d \to \infty$, computational complexity of calculating dot products in this expanded feature space becomes prohibitively expensive and unwieldy.

- $X^T X$ is already a $p \times p$ matrix
- calculating dot products of this expanded feature set is $\phi(x)^T \phi(x)$
- any expansion means a whole new set of columns needs to be generated
- v quickly becomes massive

This challenge is where kernels and the kernel trick come into play, offering a powerful solution.

38.7 Using Kernels to handle infinite dimensional spaces

- The kernel trick is a method that allows us to compute the dot products in these high-dimensional feature spaces without explicitly performing the feature expansion.
- This is possible because many machine learning algorithms, including those for classification and regression, can be rewritten in a way that they only depend on the dot products between data points, rather than the data points themselves.

- Kernels effectively compute these dot products in the transformed feature space ($\phi(x)^T\phi(x')$) directly from the original input space, significantly reducing computational complexity.
- Types of Feature Expansion
  - **The Polynomial Kernel** computes the dot product in the polynomial feature space as:
    $$k(x, x') = (\gamma \mathbf{x}^\top \mathbf{x}' + r)^d,$$
    where γ, r, and d are kernel parameters. This kernel encapsulates the essence of polynomial expansions without the need for explicit computation of the expanded features.
  - **Trigonometric Expansion**: In cases where the data exhibits periodic behavior, trigonometric expansions can be useful. This might involve adding features like $sin(x), cos(x)$, and higher-order terms for each original feature x. This type of expansion is particularly useful in time-series analysis or when modeling cyclical phenomena.
  - **The RBF kernel**, often referred to as the Gaussian kernel, is defined as:
    $$k(x, x') = \exp(-\gamma \|\mathbf{x} - \mathbf{x}'\|^2),$$
    where γ is a parameter that determines the spread of the kernel. This kernel can handle infinite-dimensional spaces because it effectively measures similarity in a space that considers all possible polynomial terms of all degrees, weighted by their exponential decay.
- Advantages of Feature Expansion:
  - **Efficiency** - Kernels allow the computation to remain tractable even as the complexity of the feature space grows. This makes it possible to work with very high-dimensional spaces, including infinite-dimensional spaces, without direct computation in those spaces.
  - **Flexibility** - Kernels allow the computation to remain tractable even as the complexity of the feature space grows. This makes it possible to work with very high-dimensional spaces, including infinite-dimensional spaces, without direct computation in those spaces.
  - **Implicit Feature Expansion** - Different kernels can model different types of relationships and patterns in the data. For instance, while a polynomial kernel can capture polynomial relationships, an RBF kernel can model more complex, non-linear relationships that might not be well-described by polynomials.

39 Defining a kernel

Call $k(x, x') = \phi(x)^T\phi(x')$. A kernel is a just a function of x, x'

A kernel is a similarity metric between vectors: it's the similarity of this polynomial space rather than the feature space

- a kernel is just a function: a dot product of 2 vectors
- dot products measure the similarity between two vectors in a high-dimensional feature space
- the most basic kernel possible is the dot product of x^Tx' - this measures the similarity of 2 vectors in feature space.

- more complex kernels include feature expansions
- kernels are specifically dot products of feature expansions: they also transforms the feature space to one where linear separability might be more achievable.
- they thus measure the similarity of two sets of features across all the dimensions that the expanded features represent.
- in this way a polynomial regression is a kernel regression

$\phi(x)$ is a feature expansion, such as $\phi(x) = [x, x^2, x^3]$

- = transforming input data (x) into a higher-dimensional space using a function ϕ
- e.g: if x is a 2D vector: $[x_1, x_2] \rightarrow$ a possible feature expansion could be $\phi(x) = [x_1, x_2, x_1^2, x_2^2, x_1 \cdot x_2]$
  - includes their original features, their squares, and their product, increasing the dimensionality of the data.

A kernel function : $k(x, x') = \phi(x) \cdot \phi(x')$:

- where the dot product is calculated in the higher-dimensional feature space.
- kernel trick computes the dot product of two vectors in the feature space without explicitly performing the feature expansion.

Gram matrix: K_{ij}

- composed of all the pairwise dot products between the feature-expanded vectors, represented as $K_{ij} = k(x_i, x_j)$ for vectors x_i, x_j
- where an element of k is the kernel function of the feature x_i and x_j
- dot products between x and x' : $K_{xx'}$
- A valid kernel leads to a positive semi-definite Gram matrix (see slides)
- i.e the determinant has to be ≥ 0
- all this is really saying, is that it needs to look like a dot product

Kernel Trick: replacing $x^T x'$ with $k(x, x')$

- it means doing a dot-product in the kernel space
- Complex kernels from simpler kernels
- They let us define complex **implicit** feature expansions: creation of more sophisticated feature mappings and decision functions.
- = the implicit representation of $\phi(x) \cdot \phi(x')$
- Kernels enable the learning algorithms to operate in high-dimensional feature spaces without explicitly computing the coordinates of the data in these spaces.
- the beauty of kernels methods is their ability to *implicitly* compute the dot product in high-dimensional feature space w/o ever explicitly computing the feature vectors $\phi(x)$ and $\phi(x')$
- We don't have to actually write out every element of $\phi(\cdot)$ - rather than building out full feature expansion, we could just work out the kernel, which might be easier.

40 Manipulating Kernels

To construct complex kernels - i.e. complex measures of similarity. We can construct a lot of complex kernels which have intuitive meaning about how we think of similarity.

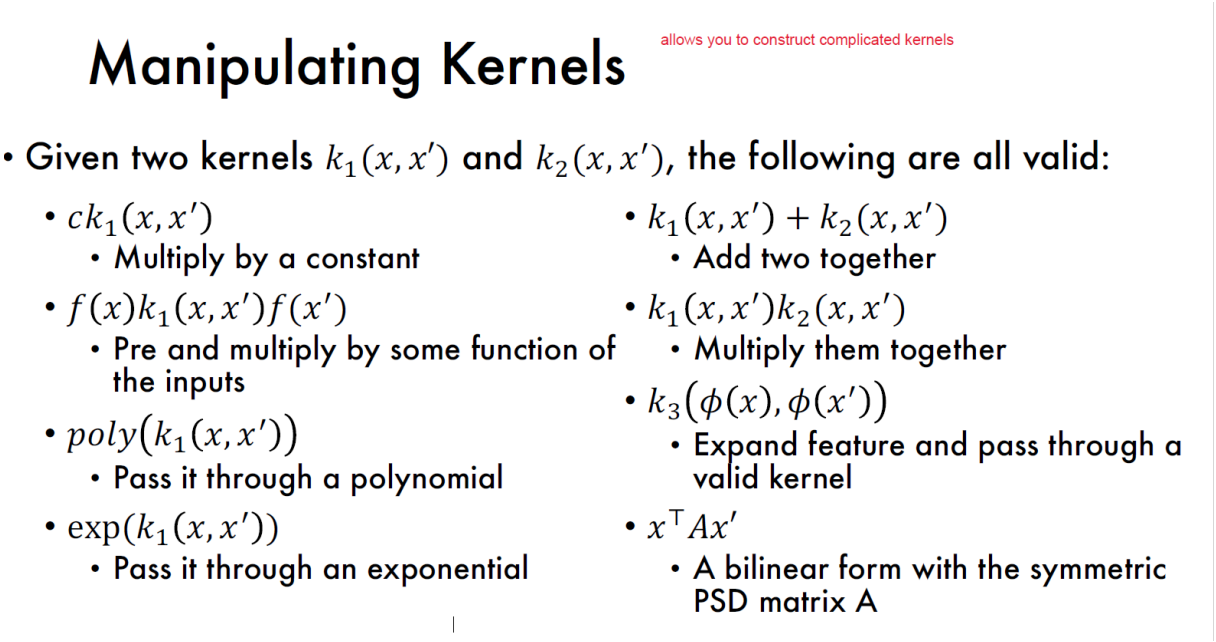


Figure 26: Enter Caption

Given a basic kernel, we can create more complex kernels by manipulating the simple kernel. This allows for the flexibility and adaptability of kernels to specific problems or datasets in machine learning, especially in support vector machines and kernel methods.

For example, a more complex kernel can be defined as:

$$\alpha k_{linear}(x, x')^{+}(1 - \alpha)k_{complex}(x, x')$$

Here, α serves as a hyperparameter, adjusting the balance between the linear kernel and the complex kernel based on the Euclidean distance between x and x'.

41 Kernel examples

Assume we have a two-dimensional input space with input vectors x and z. The polynomial kernel of degree 2 can be represented as:

Breaking this down further, we get:

$$\begin{aligned} k(x, z) &= (x^T z)^2 \\ &= (x_1 z_1 + x_2 z_2)^2 \\ &= x_1^2 z_1^2 + 2x_1 z_1 x_2 z_2 + x_2^2 z_2^2 \\ \text{sep } x \text{ and } z \text{ terms} &= (x_1^2, \sqrt{(2)} x_1 x_2 x_2^2)(z_1^2, \sqrt{(2)} z_1 z_2 z_2^2) \\ &= \phi(x)^T \phi(z) \end{aligned}$$

Kernels are functions used to compute the similarity or a measure of distance between inputs in a transformed feature space.

- This representation shows that the polynomial kernel is a dot product of the transformed feature vectors in the expanded feature space,
- where $\phi(\cdot)$ represents the feature transformation function.
- The transformed features include the original features squared and the interaction terms, effectively allowing the modeling of non-linear relationships in the input space.

Manipulations in feature space (not sure if i need this)

- When working directly in the feature space, especially with polynomial kernels, the computational cost is related to the dimensionality of the feature space ($p \times p$),
- which might be less than the cost of computing pairwise distances or dot products in the original high-dimensional space ($n \times n$),
- especially when n, the number of samples, is very large.
- This highlights the efficiency of kernel methods for handling high-dimensional data.

General Polynomial kernel
The polynomial kernel for any degree M is typically represented as:

$$k(x, x') = (x \cdot x' + c)^M$$

Where,

- c is a constant term that allows adjustment of the influence of higher-degree terms in the polynomial.
- This form of the kernel allows for the explicit computation of the dot product in the transformed feature space without needing to calculate the transformation $\phi(\cdot)$ explicitly.

The key takeaway from these examples is that polynomial kernels allow us to implicitly work in a high-dimensional feature space, enabling linear algorithms to capture non-linear relationships. This is done without the computational burden of explicitly mapping input vectors into this high-dimensional space

Thus far, we've expressed $k(x, x')$ fully explicitly through $\phi(\cdot)$... but Gaussian kernel....

42 Gaussian kernel - ie. Radial Basis Function

Infinite dimensional!

Produces a polynomial expansion of our features, where each term is weighted by a constant (in which it "prefers" lower degree polynomials)

Allows us to do regression with infinite parameters - lets us be much more expressive in your models

$$k(x, x') = \exp\left(-\frac{\|x - x^2\|^2}{2\sigma^2}\right)$$

1. take the difference
2. square it
3. sum them up
4. normalise

Calculates the similarity between two points, where:

- x and x', where $\|x - x'\|^2$ is the squared Euclidean distance between points
- σ is a parameter that controls the spread of the kernel.

Indicates how similarity between 2 points decreases as their distance increases.

42.1 Expanding the square

$$\begin{aligned} k(x, x') &= \exp\left(-\frac{x^2}{2\sigma^2}\right) \exp\left(-\frac{-2xx'}{2\sigma^2}\right) \exp\left(-\frac{x'^2}{2\sigma^2}\right) \\ &= \exp\left(-\frac{x^T x}{2\sigma^2}\right) \exp\left(-\frac{-2x^T x'}{2\sigma^2}\right) \exp\left(-\frac{x'^T x'}{2\sigma^2}\right) \\ &= \exp\left(-\frac{x^T x}{2\sigma^2}\right) \exp\left(\frac{x^T x'}{\sigma^2}\right) \exp\left(-\frac{x'^T x'}{2\sigma^2}\right) \end{aligned}$$

where:

- $\exp\left(-\frac{x^T x}{2\sigma^2}\right)$ - is the exponential of the negative squared norm of x scaled by $2\sigma^2$
- $\exp\left(-\frac{x'^T x'}{2\sigma^2}\right)$ - is the exponential of the negative squared norm of x' scaled by $2\sigma^2$
- $\exp\left(\frac{x^T x'}{\sigma^2}\right)$ - is the exponential of the dot product of x and x', scaled by σ^3. Note that the negative sign in the exponent has been removed compared to the other two components because this term actually reflects the interaction (or similarity) between x and x'

This expansion separates expression into parts that depend solely on x, x' and the interaction between the two.

When you multiply these three components, the negative squared norms of x and x' can be seen as normalization factors, and the positive dot product reflects the closeness of the two vectors x and x'.

The parameter σ controls the width of the Gaussian kernel, determining how far the influence of a single training example reaches, with low values meaning 'far' and high values meaning 'close'.

42.2 Infinite-dimensionality of Gaussian kerne's $\phi(x)$

The Gaussian kernel function $\phi(x)$ represents the mapping of input x to an infinite-dimensional space.

- it does this by leveraging the properties of the exponential function to measure similarity in a way that accounts for the distance between points, scaled by σ

Unlike polynomial kernels, where $\phi(x)$ leads to a finite number of polynomial terms, the Gaussian kernel's mapping is not as straightforward to visualize because it involves an infinite series expansion.

From the above expression we get 3 elements resembling this expression:

$$\phi(x) = \exp\left(-\frac{x^2}{2\sigma^2}\right)$$

To get the *kth* element of this expression we get:

$$\phi(x)_k = \exp\left(-\frac{x^2}{2\sigma^2}\right)\frac{x^k}{\sigma^k\sqrt{k!}}$$

Suggests that $\phi(x)$ involves components that are exponentially scaled by the squared distance of x from the origin, normalized by $2\sigma^2$. This results from the kernel's nature, mapping inputs into an infinite-dimensional space.

42.3 Intuitive heuristic to understand property of exponential function: Taylor Series approximation

Intuitively, the red highlights above represent a connection to the Talor series of

$$\exp(x) = \sum_{n=0}^{\infty}\frac{x^n}{n!}$$

This series expands $exp(x)$ into an infinite sum of terms, each of which is x raised to the power of n, divided by $n!$

The Gaussian kernel shares the property of being an exponential function - which involves rapid decay as the argument increases.

This provides insight into how the Gaussian kernel can be thought of as **incorporating an infinite number of polynomial terms, where each term is weighted by a constant derived from its order in the series.**

- **The higher the order, the smaller the weight - as $k!$ increases VERY rapidly**
- **This means it "prefers" lower degree polynomials**
- High degree polynomials are "weighted down" in similar way to what we demoed with high dimensional Fourier series ($\frac{\cos mx}{m}$ for large m)

This highlights the Gaussian kernel's capacity to capture a vast range of non-linear relationships by implicitly considering all possible polynomial interactions in an infinite-dimensional space.

- Each component of the input vector contributes to the feature mapping,
- but unlike explicit polynomial expansion, this happens in a way that's computationally feasible due to the kernel trick,
- which allows the computation of dot products in this high-dimensional space without directly calculating the feature map $\phi(x)$

By considering all possible polynomial terms through its exponential function and the Taylor series, the Gaussian kernel offers a remarkably flexible approach to model various data structures and relationships, all while maintaining computational efficiency through the kernel trick.

In the Gaussian kernel, the feature space is implicitly an infinite-dimensional space, and the kernel function provides a similarity measure between any two instances. The fact that this calculation only depends on the distance between x and x' means it is computationally efficient despite the high dimensionality of the feature space.

43 Kernel adv I: expressive, non-linear models

Kernels are functions that implicitly map data into a high-dimensional space to find patterns or features that are not readily apparent in the original space.

Kernels allow for non-linear models: let you be much more expressive in your models:

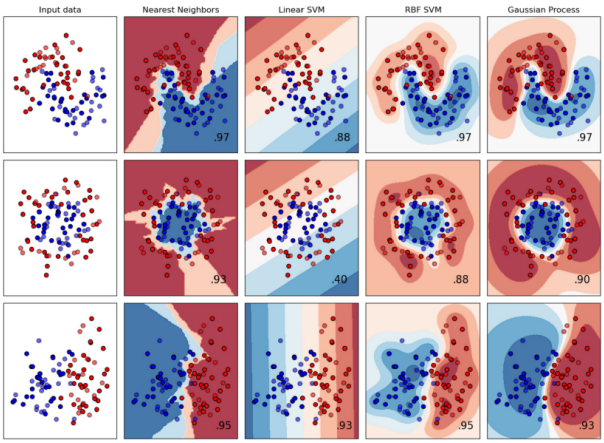


Figure 27: right-most = expressive kernel-based models due to locally weighted similarity

Here the Radial Base Function delineates based on locally-weighted similarity, in which the outcome is predicted based on outcomes that are 'close'; *we define close through different kernels*

44 Kernel adv II: model different similarities

Kernels are functions that implicitly map data into a high-dimensional space to find patterns or features that are not readily apparent in the original space.

44.1 Different kernel → different similarity measure.

Kernels allow you to express things you believe about your data (eg that there's a cyclical time trend, of that one part of the data you think can learn well with highly local models, whereas there's another part of the model that is much more global. This flexibility allows us to push a little bit further through the bias-variance trade off.

Can mix & match to define the kinds of models you can learn

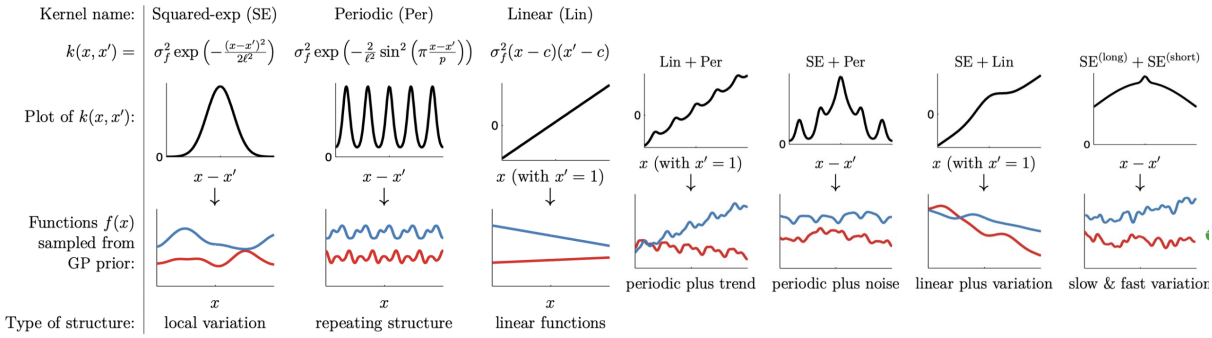


Figure 28: Combining Kernels

Above:

- Gaussian kernel: takes points nearby - idea of locality in similarity.
- Periodic kernel: for cyclical time series - another kind of similarity is that data of the week have similarity to each other.
- far right: adjusting the σ^2 hyper parameter - combines a low frequency, wider global model, with a much more high-frequency local model.

44.2 Or you can model other kinds of structure:

- **Inducing Low-Rank Global Structure:**
  - A "low-rank" structure implies that there is some underlying simplicity or pattern in the data.
  - When a kernel induces a low-rank structure, it means that the data can be represented in a lower-dimensional space effectively.
  - This is a global structure because it applies across the entire dataset.
  - For example, Principal Component Analysis (PCA) is a technique that finds a low-rank approximation of the data.
- **But also allow for local variation:**
  - Despite the global structure, it's important for a model to capture the local nuances and variations in the data.
  - Kernels can cater to this by allowing for non-linear boundaries that adapt to local properties of the data.
  - The RBF kernel, for example, can create complex regions that closely fit the distribution of each class in the data.

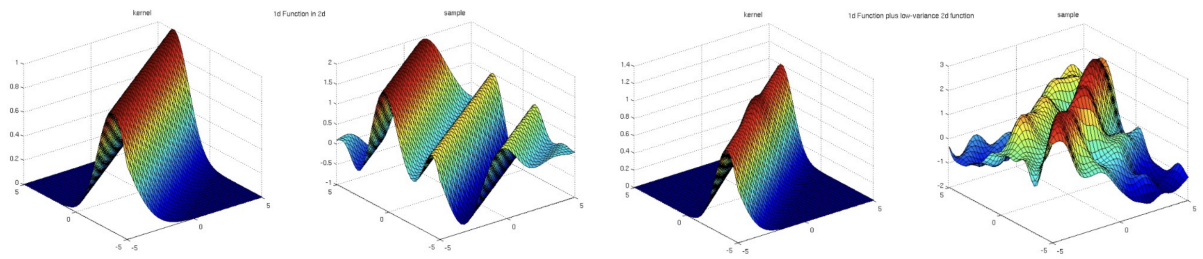


Figure 29: Enter Caption

Taking first few low-rank singular values (i.e. the main features), but also allows for local variation. Allows things to be smooth, but not too complicated.

- **Or allow for Learning Different Features Demand Different Ideas of Similarity:**
  - Not all features contribute equally to the patterns in the data.
  - Some features might be more important than others, and some pairs of data points might be considered similar based on one subset of features but different based on another.
  - By using or combining different kernels, or by learning the parameters of a kernel (like in multiple kernel learning), the model can learn which features (or combinations thereof) are most indicative of similarity for a particular task.
  - For example, in image processing, color and texture might contribute differently to the classification of objects within an image, and different kernels or kernel parameters might be better at capturing the importance of each.

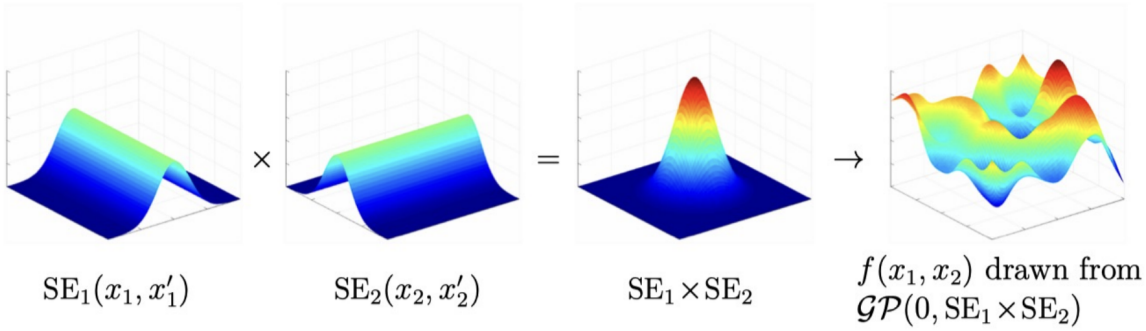


Figure 30: Enter Caption

- By combining these three elements, you can build machine learning models that are capable of capturing both the broad, global trends in the data (low-rank structure) and the fine-grained, local patterns (local variation), as well as learning task-specific notions of similarity among the data points (different features demanding different ideas of similarity). This balance is crucial for creating models that are both accurate and robust to new, unseen data.
  - There are lots of ways to construct custom kernels for your application
  - More structure can make it easier to learn

When NOT to use kernels:

- when your sample size is constrained
- when your data is high dimensional (in high dimensions everything is far apart)

45 Use of flexible similarity-based models:

Here are two approaches to building models that are based on the concept of similarity between data points, rather than relying solely on linear correlations among features.

45.1 Global: Kernel Ridge Regression

$$\tilde{X}X^T(XX^T + \lambda I_n)^{-1}y$$
$$\downarrow$$
$$\phi(\tilde{X}, X)(\phi(X, X) + \lambda I_n)^{-1}y$$
$$\downarrow$$
$$K_{\tilde{x}x}(K_{xx} + \lambda I_n)^{-1}$$

- idea is to extend the concept of linear models to accommodate non-linear relationships between the variables
- achieved by replacing the linear correlation term (XX^T) in ridge regr, with kernel function of our choosing $k(X, X')$
- The kernel function computes the similarity between data points in a potentially high-dimensional space without explicitly mapping the data to that space, thanks to the kernel trick.
- $K(X, X)$ represents the kernel matrix computed from the training matrix data.
  - each element is the result of applying the kernel function to a pair of data points.

Evaluation

- **'weird' properties of a global model**: farther away points can feed into the predictions
- **Fixed size kernel across the space** - same "lens" or weighting across the entire space, which may not always be ideal. Different regions of the data space might benefit from different levels of sensitivity or types of kernels.
- **Computational complexity** requires a big $n \times n$ matrix
  - why?

= quite flexible, but needs tuning of hyper parameters, incl:

- choice of kernel
- regularization parameter
- distance metric to measure similarity between points

As we add flexibility, tuning becomes increasingly important

45.2 Local: K Nearest Neighbours Regression

Instead of averaging across the whole space, we truncate to only use the K most similar units, based on $k(x, x')$

- making predictions based on a subset of the most similar data points to the point of interest, rather than using the entire dataset
- where K represents number of similar (nearest) data points to consider
- In KNN regression, similarity is usually measured using a distance metric (e.g., Euclidean distance), but the concept of kernels can also be applied to define similarity.

- The prediction for a new data point is made by averaging the target values of the K nearest neighbors to this point, effectively using a similarity-weighted average.
- inherently non-linear and can adapt to local data patterns, making it highly flexible.
- unlike kernel ridge regression, which incorporates all data points into a global model, KNN regression focuses on local information, potentially leading to very different behavior, especially in regions of the input space where data is sparse or patterns change abruptly

Evaluation

- ***Exclusively* local** - (unlike KNN) see demo graphs
- **Simple averaging** - prediction for a new point is average target value of its K nearest neighbors - both a strength and a weakness. Assumes equal importance of all neighbors and ignores the possibility of varying relationships across the input space.
- **Bias-variance tradeoff** - choice of K (the number of neighbors) is crucial: it directly affects the bias-variance tradeoff.
  - smaller $K \rightarrow$ models that capture noise in the data (high variance),
  - larger K might oversmooth and miss important patterns (high bias).

= quite flexible, but needs tuning of hyperparameters, incl:

- choice of kernel
- regularization K
- distance metric to measure similarity between points

46 Demo of the models

- linear weighting is extremely constrained - esp in the middle, it's just the avg value (specifically with RBF)
- as σ^2 increases –> spread gets wider –> fits more of a global mean model
- our standard kernel is the euclidean distance = squared something????

47 Kernels in High Dimensions

TL;DR: kernel models are very flexible, but they don't always workwell as features become high dimensional

- kernels are functions which describe (?) distance.
- concepts of 'distance' break down in high dimensional spaces.

= **'curse of multi dimensionality': Points tend to become equidistant from each other in high dimensions**, affecting the performance of algorithms reliant on distance or similarity measures.

NB: much of this from ChatGPT - goes beyond what we need
In high-dimensional spaces, the intuition from low-dimensional spaces often does not apply, particularly regarding distances and volume distribution.

This affects the performance of models like KNN and Kernel Ridge Regression. Consider a scenario where:

47.1 Distance in High Dimensional Space

In high dimensions, nothing is close together!

- X $Uniform(-1,1)^d$,
- Volume of the domain of X : 2^d
- X is uniformly distributed across a d-dimensional hypercube, with each dimension ranging from -1 to 1. Thus, the volume of this hypercube is 2^d.
- For a test point at the origin, we wish to calculate the fraction of points within a certain distance $\epsilon > 0$.

The volume of a sphere (or its high-dimensional equivalent) in a d-dimensional space does not scale straightforwardly with dimensionality.

47.2 Volume in High Dimensional Space

The volume of a sphere in d dimensions within a radius ϵ is given by the formula:

$$V_d(\epsilon) = \frac{\pi^{d/2}}{\Gamma\left(\frac{d}{2}+1\right)}\epsilon^d$$

where Γ is the Gamma function, a generalization of factorial.

As the dimensionality d increases, the volume of the sphere becomes a smaller fraction of the hypercube's volume. This means that in high-dimensional space, even a small ϵ encompasses a diminishing fraction of the total volume, making the concept of "nearest neighbors" less meaningful.

47.3 Implications for Machine Learning

The peculiar behavior of distance and volume in high-dimensional spaces has significant implications for machine learning:

- **Curse of Dimensionality:** Points tend to become equidistant from each other in high dimensions, affecting the performance of algorithms reliant on distance or similarity measures.
- **Dimensionality Reduction:** Techniques like PCA, t-SNE, or autoencoders are employed to project data into a lower-dimensional space, where distances are more meaningful.
- **Careful Feature Selection:** It becomes crucial to select or engineer features carefully to avoid the curse of dimensionality in high-dimensional machine learning tasks.

d	Volume of domain of X	Volume of sphere	Fraction of data nearby	$\epsilon = \frac{1}{2}$
1	2^1	2ϵ	ϵ	$\frac{1}{2} = 0.5$
2	2^2	$\pi\epsilon^2$	$\frac{\pi}{4}\epsilon^2$	$\frac{\pi}{16} \approx 0.20$
3	2^3	$\frac{4\pi}{3}\epsilon^3$	$\frac{\pi}{6}\epsilon^3$	$\frac{\pi}{48} \approx 0.07$
4	2^4	$\frac{\pi^2}{2}\epsilon^4$	$\frac{\pi^2}{32}\epsilon^4$	$\frac{\pi^2}{512} \approx 0.02$

Figure 31: NB: how quickly ϵ decreases as dimensions increase

In high dimensions, the amount of your data that is in a small local area becomes v small.
In high dimensions, everything becomes extrapolation, rather than interpolation.

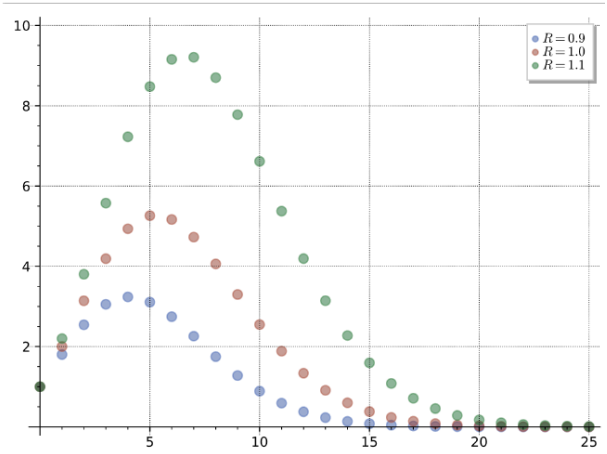
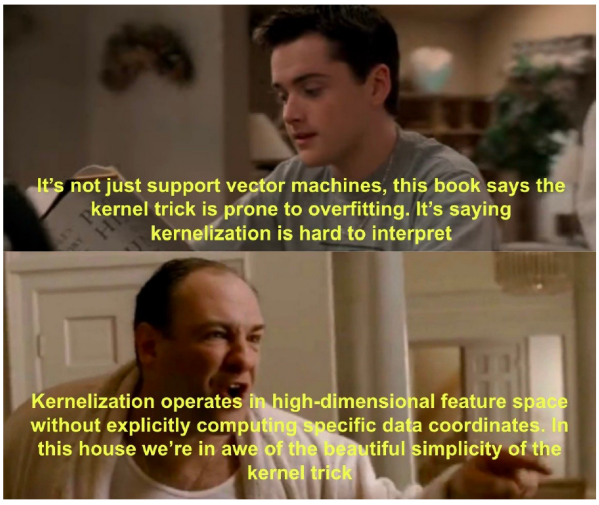


Figure 32: Enter Caption

48 Summary

- kernels as a new way to think about measuring similarity
- There are lots of ways to construct custom kernels for your application
- More structure can make it easier to learn - we have the ability to use what we know about where the data comes from, to build the right kind of structure
- Their reliance on distance means that they can struggle in high dimensions without further work



ML Lecture Notes: Wk 6-II — Qualitative Fairness

49 Machine Learning in Context

Idea of bias in data → running models exacerbates those biases.

Sometimes you might not want to include data you considered biased / isn't relevant to the purpose of the model.

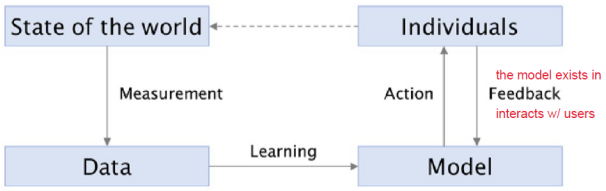


Figure 33: Enter Caption

Example of encoded bias: which patterns do we want to mimic, and which not?

Figure 34: Enter Caption

Feedback loops E.g. Predictive policing - self-fulfilling prophecy.
see Lum & Isaac 2016 - more crime observed when cop location determined based on observed crime rates
... Also, consider social scienc hat: construct validity of arrest as measures of crime committed....

50 Types of harm

Allocative Harm
A system withholds a resource or opportunity based on group membership.

Representational Harm
A system reinforces subordination of some group along the lines of identity

51 What is 'Fairness' (Qualitative)

Legitimacy
Is there any sense in which such a system should be deployed? (*precedes* a discussion of the specific allocative risks/harms.
E.g. predicting crime from faces - is there any legitimate reason to be doing so?

Relative treatement of groups
How does the system allocate resources? Can be measured rigorously and quantitatively
Procedural Fairness... as we work with more and more complex models, how can we know if it is rational

52 Types of Automation

1. **Rules previously defined by hand / process**
   - benefits eligibility / minimum requirements for jobs
   - loses flexibility of discretion
2. **Rules previously only informally defined**
   - automated grading
   - expert systems: try to machine-learn decision
   - BUT, may not rely on what we want it to! - may learn in a different way
     - for us, often the process is important
     - reasoning matters
3. **Fully learning rules from data**
   - who should lenders grant loans to?
   - where should police patrol the streets?
   - choose a proxy to machine learn as the basis for the decision
   - BUT once again we're in feedback loop territory..see next

52.1 Problems in Type-3 automation

- **Bias**: how do you create the appropriate rules from data when the data you're training on is problematic? For many sensitive domains there is no unbiased data
- **Proxy Mismatch/construct validity** - data often chosen based on convenience (/ppl with power collect data (an expensive exercise) -> biased
  - E.g. Entrance into a "high risk care management system" in the US - model based on predicted health-care costs (as a proxy health-care needs, with the logic being that high health care costs mean high need).
  - Problem: by defining target as payment for service, you are excluding those who were not prev able to pay for healthcare; ie. the most vulnerable
- **Feature Failing** - Failing to consider relevant info → Mistakenly think two people are alike because you didn't measure how they were different
  - if you don't measure across features, you can't see that there's difference between them

  - this is where human discretion is important: they can have a conversation / other, and see how units differ across variables / for reasons not included in the input form

- **Distribution Shifts** - data we want to apply our decisions on differs systematically from what we trained / evaluated on
  - eg medical trials predominantly middle class, white, educated, male.

53 Agency, Recourse, Culpability

Models on immutable characteristics

- people cannot change their fate

Models on mutable characteristics

- then an obligation to empower individuals so they know how to change their fate
- BUT then they can 'game' they system..?

ML Lecture Notes: Wk 7-I — Quantitative Fairness

54 Classification model set up

Binary classification: learn $f(X) : \mathcal{X} \to (0, 1)$

Suppose we want to take actions based on resulting model, $\hat{f}(X)$

54.1 Risk Scores and Estimation

The risk score, denoted as $r(x)$, is essentially an estimation of the probability $E[y|X = x]$ that $Y = 1$ given the features $X = x$. This expectation is the probability of the positive class conditioned on the features.

54.2 Ideal Model and Reality

Ideal scenario: if we had perfect knowledge, we could define our classifier's action as:

$$\hat{y}(x) = I(E[y|X = x] > 0.5)$$

Which is to say:

$$\hat{y}(x) = \begin{cases} 1 & \text{if } E[Y \mid X = x] > 0.5 \\ 0 & \text{otherwise} \end{cases}$$

where I is the indicator function that outputs 1 if its argument is true and 0 otherwise.

This means if the expected probability of $Y = 1$ given $X = x$ is greater than 0.5, we predict 1 (the positive class); otherwise, we predict 0.

However, **the reality** is that we don't know $E[Y \mid X = x]$ a priori. Thus, we estimate it using our data and a heavy-side function.

This estimation is where regression models are 'hiding' under the hood of classification models - specifically logistic regression in many binary classification tasks. Logistic regression models the probability that $Y = 1$ as a function of X, providing us with an estimate of $r(x)$ or $\hat{E}[Y \mid X = x]$.

> Risk score is probability prediction from a regression model.
>
> It is the expectation (the prob) of the positive class, conditioned on the features

54.3 Regression Under the Hood of Classification

In binary classification, the regression model (like logistic regression) helps estimate the risk score $r(x)$ or the probability $E[Y \mid X = x]$. This estimated probability is then used to make a classification decision (e.g., if the estimated probability > 0.5, classify as 1; otherwise, 0).

55 Accuracy

Accuracy is a metric for evaluating classification models, defined as the ratio of correctly predicted observations (both true positives, TP, and true negatives, TN) to the total observations

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

or

$$\text{Accuracy} = \frac{1}{n}(TP + TN)$$

How accurate? The required level of accuracy depends on the application and its implications.

Accuracy for Whom? highlights importance of considering the impact of model predictions on different stakeholders. A medical diagnostic test's false negatives might have severe implications for patients, whereas false positives might burden the healthcare system with unnecessary costs.

56 Cost-Sensitive Learning (alt approach 1)

$$\text{Accuracy} = \frac{1}{n}(TP + TN)$$

... makes implicit assumption that Type I and Type II errors are equally as costly...

	$y = 0$	$y = 1$
$\hat{y} = 0$	c_{00}	c_{01}
$\hat{y} = 1$	c_{10}	c_{11}

Figure 35: Confusion Matrix

We want c_{00} and c_{11}; but how do we balance c_{10} and c_{01} against eachother

- Type-I errors (false positives) = c_{01}
- Type-II errors (false negatives) = c_{10}

We have to explicitly code our values into the system to calculate and optimise towards a cost-sensitive loss. These are utility values.

$$CostSensitiveLoss = \frac{1}{n}(FN \times c_{FN} + FP \times c_{FP})$$

$$CostSensitiveLoss = \frac{1}{n}(FN \times c_{01} + FP \times c_{10})$$

This approach aims to minimize a weighted sum of errors, allowing for a more nuanced optimization that reflects the actual costs (financial, ethical, etc.) associated with different errors.

Libraries like scikit-learn (sklearn) implement this concept through mechanisms like "class weights."

57 Receiver Operating Characteristic (ROC) Curves - alt approach II

Receiver Operating Characteristic (ROC) curve is a tool in evaluating the performance of binary classification models.

A graphical representation of a classifier's ability to distinguish between the two classes at various threshold settings.

57.1 ROC & Confusion Matrix

ROC curve plots the True Positive Rate (TPR, or sensitivity) against the False Positive Rate (FPR, or 1 - specificity) at different threshold levels.

Here, aforementioned **risk $\hat{r}(x)$ acts as the threshold**. Risk/threshold is the point at which the predicted probability is considered to classify an observation into the positive class.

At any given threshold:

$$\text{True Positive Rate (TPR)} = \frac{TP}{TP + FN}$$

$$\text{False Positive Rate (FPR)} = \frac{FP}{FP + TN}.$$

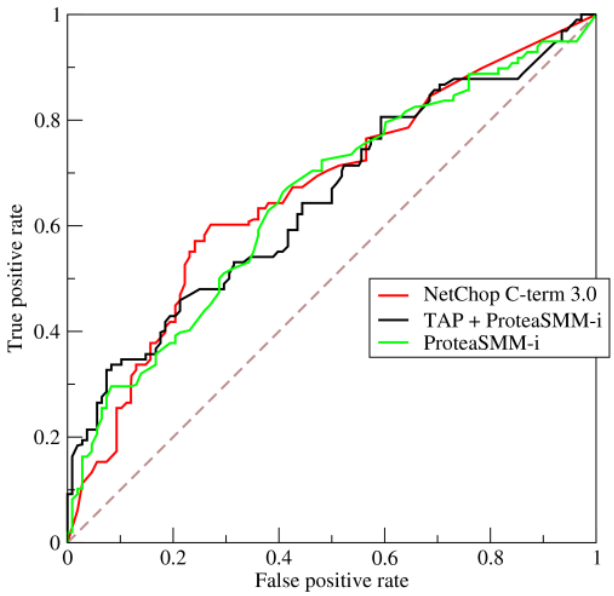


Figure 36: ROC curve

Where:

- bottom left: nothing predicted positive
- top right: everything predicted positive
- diagonal: completely random .5 classifier results
- closer to y-axis the better

57.2 Model Comparison & ROC

If one model's ROC curve is consistently above another's across the FPR range, it indicates that the former model has a better balance of true positives and false positives for all threshold settings.

For any 'reasonable loss', that model is preferred.

A "reasonable" loss function, is one that adheres to Proper Scoring Rules, which are criteria ensuring that the predicted probabilities accurately reflect the true underlying probabilities. Proper Scoring Rules encourage models to estimate true probabilities as accurately as possible (is 'treat the problem like regression'), rather than merely optimizing for classifications. This approach aligns with treating the problem somewhat like regression, where the goal is to accurately predict numerical values (in this case, probabilities) rather than discrete classes.

This motivates the **Area Under Curve (AUC)** measure.

- An AUC of 1 indicates a perfect model;
- an AUC of 0.5 suggests a model no better than random chance.

The AUC is particularly useful because it is independent of the classification threshold and provides an aggregate measure of performance across all possible thresholds.

NB: **AUC is an overall evaluation of a model**, but often we are interested in specific areas of the curve (e.g. 95%+ True Positive rate etc), whereas AUC tells you over ALL thresholds.

AUC allows us to select, **both the model, and the threshold**.

- it shows us how well each model performs at different thresholds.
- the model gives you the line
- each point on the curve gives you the threshold rate

There are different ways to chose the model based on this:
Threshold-led:

1. you might specify that you can't accept a False Positive rate $> 20\%$
2. so now we're only looking at the FP $<20\%$ section on the x axis
3. then you chose the curve that's best (ie closest to y-axis) between 0-20%

Or, you might be model-led: evaluate model as a whole (see AUC motivation)

58 Discrimination via classification

Problem: it is very possible that X encodes sensitive features about group membership.
Explicit Use of Sensitive Features

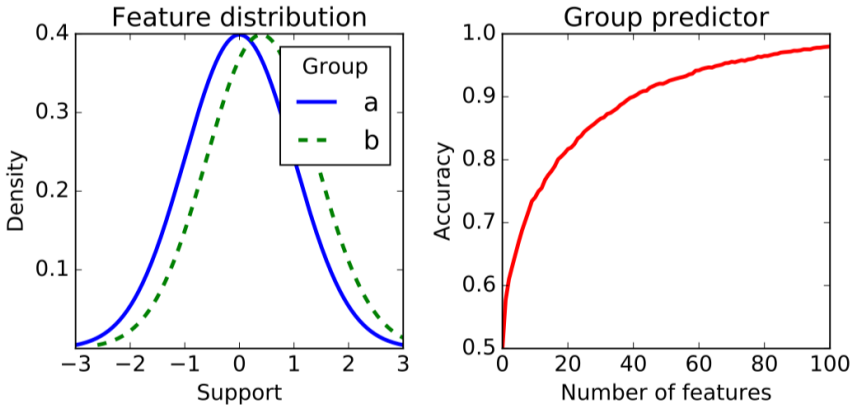


Figure 37: group a and b similar, so if we only have one feature its hard to predict group membership. But if we have many predictors, we are able to predict group membership extremely well

When features explicitly encode sensitive information, such as "self-reported race," using these features directly in a machine learning model can lead to discriminatory outcomes.

Models may learn to make decisions based on these sensitive attributes, perpetuating or even exacerbating existing biases present in the data or societal structures.
Implicit Encoding of Sensitive Features

Models can still learn discriminatory patterns through features that are correlated with group membership.

Socioeconomic factors do a good job of predicting race despite not directly using race.

58.1 Accumulation of Slight Predictivity

A set of features, each with slight predictivity for a sensitive group, can collectively enable a model to classify individuals into groups with high accuracy.

This phenomenon is related to the concept of "redundant encodings," where combinations of non-sensitive features effectively encode sensitive information.

As a result, models can exhibit discriminatory behavior based on learned patterns that align closely with sensitive group characteristics, even if no single feature strongly predicts group membership

58.2 Addressing Discrimination in Classification

Fairness Metrics and Objectives: Various metrics and definitions of fairness (e.g., demographic parity, equal opportunity) can guide the evaluation and adjustment of models to reduce discrimination.

Feature Selection and Engineering: Carefully reviewing and selecting features to minimize the risk of encoding sensitive information, either directly or indirectly.

Bias Mitigation Techniques: Applying pre-processing, in-processing, and post-processing techniques to reduce bias. These include methods to alter training data to be less biased, adjust the learning algorithm itself, or modify the model's predictions to achieve fairness objectives.

Enhancing the transparency and interpretability of models can help identify and understand potential sources of bias, leading to more informed adjustments and improvements.

59 Quantitative Fairness: Independence, Separation, Sufficiency

- R **is the risk score** - e.g. $\hat{r}(x)$
- A **is the sensitive attribute** - e.g. race / gender
- Y **is the outcome / label** - e.g. good / bad job candidate
- $\hat{Y}$ **is the predicted label** - e.g. the decision to hire
- X **are the features** of the prediction model

Our goal is to understand restrictions on R which would lead to "fair" results.

59.1 Independence

$$R \perp A$$

The risk score is independent of the sensitive attribute

This implies:

$$p(R|A = 1) = p(R|A = 0)$$

- probability distribution of the risk scores R given the sensitive attribute A is the same in both groups
- This implies that the model's assessment of risk is not influenced by the sensitive attribute

$$p(Y|A = 1) = p(Y|A = 0)$$

- probability distribution of the predicted probabilities $P(\hat{Y})$ i.e., the model's estimated probabilities of the positive class), given the sensitive attribute A is the same across groups.
- This means that the model's predictions are made independently of the sensitive attribute.

Risk scores should look the same for members of each group.Implies acceptance rates should be the same across groups (if we are making some sort of decision off risk score).

59.1.1 Implications for Fairness

Risk Scores and Group Membership: If risk score is independent of sensitive attribute, model evaluates risk based solely on relevant factors that do not include the sensitive attribute. Referred to as "demographic parity" or "statistical parity" - decision-making process is fair across different groups.

Acceptance Rates Across Groups: A direct implication of the above is that acceptance rates—the proportion of individuals from each group who are predicted to be in the positive

class (e.g., receiving a loan, being hired)—should be the same across groups.

Partialling out / orthogonal projection

You could also measure for race, then strip out all components of variables that are related to the feature you measure using a technique similar to partialling out or orthogonal projection. This process doesn't remove the feature itself but instead removes the portion of variation in the feature which corresponds to the protected feature.

This is achieved by regressing each predictor on the sensitive attributes and then using the residuals from these regressions as new predictors. These residuals represent the original predictors with the influence of the sensitive attributes removed. So, when you run the regression model using these residuals, you have predictors that are uncorrelated with the sensitive attributes.
This method aims to mitigate the impact of the sensitive attribute on the model's predictions, thereby reducing bias related to that attribute.

59.2 Separation (Conditional Independence/Equalised Odds)

$$R \perp A|Y$$

The risk score is independent of the sensitive attribute, within strata defined by the label
This implies:

1. **Error Rate is the same between groups**

$$TPR_{A=1} = TPR_{A=0}$$

$$FPR_{A=1} = FPR_{A=0}$$

2. **Equal Treatment Among Similarly Situated Individuals:** The risk score for individuals within the same outcome category (good or bad candidates) should be independent of A.

    - e.g. men and women identified as good candidates should have similar risk scores, irrespective of the gender difference.
    - This does not imply, however, that the proportions of good and bad candidates need to be the same between these groups.

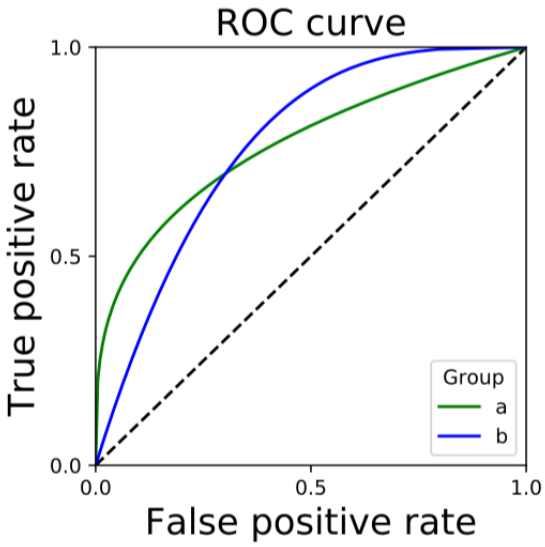


Figure 38: Enter Caption

Only the intersection is the model we select

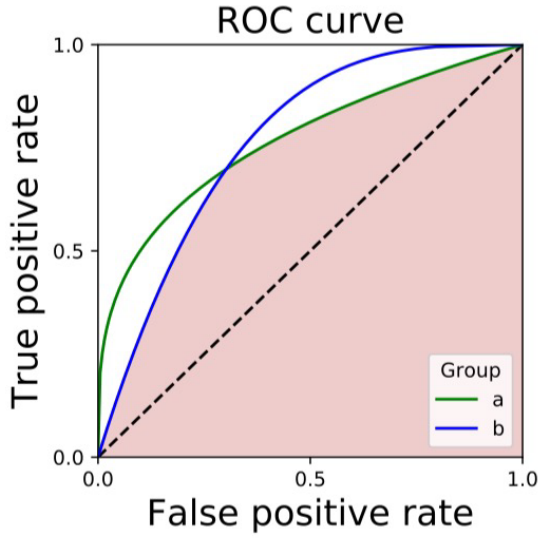


Figure 39: Enter Caption

This approach to fairness—ensuring equal error rates across groups—aligns with the concept of **equalized odds**, a fairness criterion demanding that a classifier's TPR and FPR be equal across groups defined by a sensitive attribute.

59.3 Sufficiency

$$Y \perp A|R$$

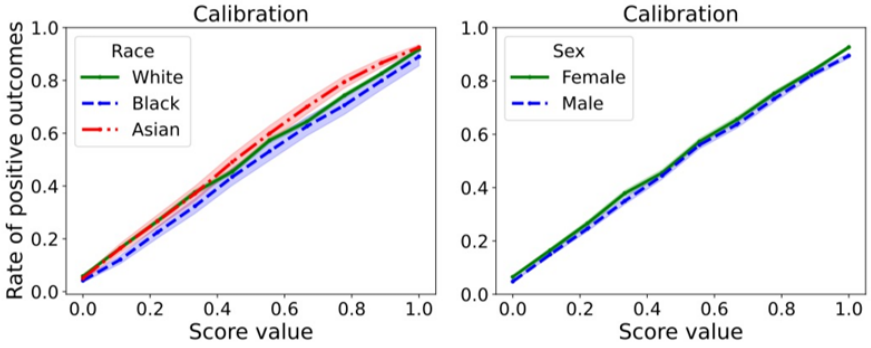
Outcome frequency given risk score, is equal across groups

Given the same risk score, the probability of a certain outcome should be equal regardless of the group membership.

$$P(Y = 1 \mid R = r, A = 1) = P(Y = 1 \mid R = r, A = 0)$$

Each group is **'well calibrated'**: for each group, the model's predicted probabilities match the observed probabilities.

Calibration is a measure of how well the probabilities of a predictive model correspond to the actual outcomes. A well-calibrated model means that if the model predicts an event with 70% probability, then that event should indeed happen approximately 70% of the time



On average the risk score is right in each group.

- This doesn't necessarily mean the model is perfect on an individual level
- but indicates that, on balance, the risk scores are a reliable indicator of the true risk within each group.
- it does not guarantee that the model is highly accurate on an individual level. A model can be fair in terms of sufficiency but still have room for improvement in accuracy and precision, especially in predicting individual outcomes.

59.4 Independence & Sufficiency

if A is related to Y - i.e. group membership gives some information about outcomes:
Sufficiency $(Y \perp A|R)$ and independence $(R \perp A)$ cannot both hold.

Independence: indicates that the probability distribution of the risk scores is the same across groups -> aims for selection rates equal across groups.

Sufficiency: ensures that, for any given risk score, the outcome probabilities are equal across different groups.

Good calibration means unequal acceptance rates:

- Good calibration within groups means that for individuals with the same risk score, the probability of the outcome is consistent across different groups.
- However, achieving good calibration does not necessarily lead to equal acceptance rates across groups.
- If the underlying distributions of risk are different between groups (which they often are when A is related to Y), then a well-calibrated model can result in unequal acceptance rates.

- the model accurately reflects the differing distributions of risk, which naturally leads to different rates of positive outcomes between the groups.

Equal acceptance rates means bad calibration

59.5 Independence & Separation

Separation: $R \perp A|Y$ - for individuals with the same outcome, the prediction of the model should be the same regardless of the group membership - aims for equal error rates across groups, such as equal false positive rates and equal false negative rates for binary outcomes.

Equal Acceptance Rates means Unequal Error Rates:

- Achieving independence by ensuring equal acceptance rates across groups inevitably leads to unequal error rates.
- the model disregards the actual distribution of the outcomes across different groups in favor of equalizing decision rates, which can amplify the impact of underlying disparities in the base rates of the outcomes.

Equal Error Rates means Unequal Acceptance Rates:

- separation (equalized odds) by equalizing error rates across groups results in unequal acceptance rates.
- the model attempts to adjust its predictions to compensate for differences in outcome distributions, leading to decisions that reflect these underlying disparities.

59.6 Fairness is not a technical problem

You simply cannot have it all - reasonable quantitative measures of fairness conflict with one another. You must think through what matters in your particular case

Fairness is a problem of expressing the values that your system should embody - you have to *explicitly encode that (see above)*

These criteria result in different models:

- **Max profit:** No fairness constraints; just an accuracy trade off. Delivers wildly different rates by race.
- **Single threshold:** one threshold for all groups
- **Independence:** equal acceptance rates
- **Separation:** equal error ates

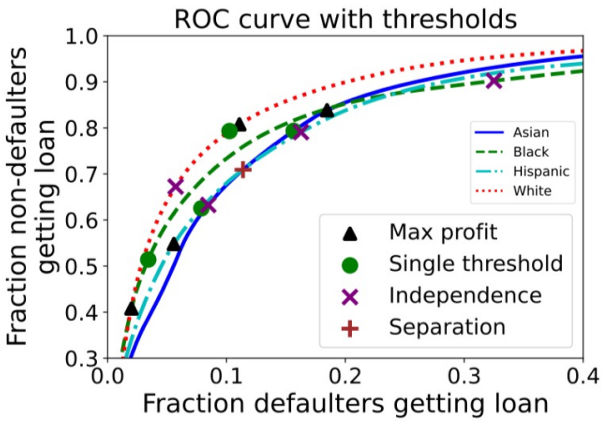


Figure 40: Enter Caption

60 POSIWID

"The Purpose of a System is What it Does" - Stafford Beer (management consultant; cybernetician

When dealing with complex systems, focus on the results they generate.

Don't focus narrowly on "the algorithm" (means you ignore its wider purpose and get hung up on technical details), try to evaluate the larger system it is part of.

See FB slides.

ML Lecture Notes: Wk 7-II — Tree-based methods

61 Decision Tree as new basis for constructing functions

For both classification & regression.

Involve segmenting the predictor space into a number of simple regions.

To make a prediction for a given observation, one typically uses the mean or mode of the training observations in the region to which it belongs.

The decision tree is the most fundamental of these methods and serves as the building block for more advanced techniques like random forests and gradient boosting machines.

61.1 Constructing a Decision Tree

1. **Choose a Feature**
    - At each node of the tree, the algorithm selects one feature that best splits the set of items.
    - The criterion for "best" can depend on the task at hand (classification or regression)
    - and is determined by measures such as the Gini impurity, entropy for classification tasks, or variance reduction for regression.
2. **Splitting Data Based on Feature Values**
    - dataset is split into subsets based on the values of that feature.
    - can be a binary split (e.g., high vs. low) or more granular, depending on the feature and splitting criteria used.
3. **Predictions at Terminal Nodes:**
    - process of splitting continues recursively, creating a tree structure with nodes and branches, until a stopping criterion is reached (such as a minimum number of samples per node).
    - terminal nodes, or leaves, represent the segments of the dataset that are as homogeneous as possible.
    - In classification tasks, the prediction for each leaf node is usually the most common label (class) of the training samples in that node.
    - For regression tasks, it's common to use the average of the training samples' target values in the node.
4. **Labels = Good, Bad:**
    - In the context of a binary classification problem where the outcomes are labeled as "good" or "bad," the decision tree will aim to separate the data such that each leaf node is as pure as possible with respect to these labels.

- The "average" mentioned refers to the proportion of each class in classification tasks, serving as the basis for making predictions for new data points that end up in each leaf.

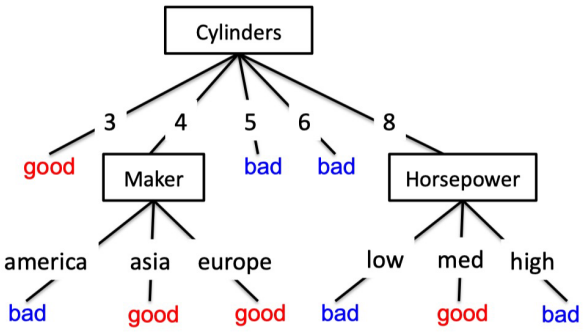


Figure 41: NB sort of interaction effect along branches: '4 cylinders x continent'; 'horsepower x low-med-high'

61.2 Properties of Decision Trees - adv vs disadv

61.2.1 Advantages

- **Flexibility:** - v general hypothesis class
  - can handle any input
  - can capture complex nonlinear relationships between the features and the response.
  - can easily handle qualitative features without the need for dummy variables.
- **Interpretability:**
  - One of the main advantages of decision trees is their ease of interpretation. They can be visualized graphically and understood by non-experts, making them useful in decision-support systems.
  - you just traverse down the tree, at each node ask a question then at the end of asking these questions you have a prediction
- **Non-parametric Nature:**
  - do not assume any specific functional form between the features and the response, making them adaptable to various scenarios.
  - Standardisation doesn't matter
- **Capability:**
  - With sufficient depth, a decision tree can theoretically capture any functional relationship between input features and the output.
  - fast and scalable (as they are greedy)
  - This flexibility comes at the cost of a higher risk of overfitting, especially as the depth of the tree increases.
- Robust to outliers
  - 1) an outlier only affects the node it is in

  - 2) a single outlier wont make changes to the internal structure of how you would make these splits

- Missingness is handled naturally - just split on whether the feature value is missing

61.2.2 Disadvantages

- **Overfitting:** Without constraints, a decision tree can grow to perfectly fit the training data,
  - To mitigate this, techniques such as pruning (reducing the size of the tree), setting a maximum depth, or requiring a minimum number of samples to split a node are commonly used.
  - Additionally, ensemble methods like Random Forests and Gradient Boosting Machines aggregate the predictions of multiple trees to improve predictive performance and generalization to unseen data.
- **Suboptimal Predictive Performance - Simplicity Leading to Inaccuracy**
  - do not provide the same level of predictive accuracy as other, more complex models.
  - partly because the simple, hierarchical decision-making process might not capture all the nuances in the data, especially in cases where relationships between variables are complex and non-linear.
- **Difficulty in Optimizing for 'best tree;**
  - as cannot differentiate -> have to be greedy -> Combinatorial Problem
  - The space of all possible trees for a given dataset is vast, and finding the optimal tree (i.e., the tree that minimizes some loss function over all possible trees) is computationally infeasible for all but the simplest datasets.
  - The greedy algorithms used to build trees can only assure finding locally optimal solutions at each step, which may not be globally optimal.
- **Computationally Intensive:** Although greedy recursive splitting is efficient compared to searching all possible trees, it can still be computationally intensive, especially with large datasets and a high number of features.
- **Local Optima:** The greedy approach does not guarantee finding the globally optimal tree. It makes the best decision at each step, which may not lead to the best overall tree.
- **Instability to Data Changes**
  - very sensitive to small changes in the training data.
  - Adding or removing a few data points, especially if they are outliers or near decision boundaries, can lead to a significantly different tree structure.
  - This sensitivity is indicative of high variance in the model, meaning that small changes in the data can lead to large changes in the model outcome.
- **Instability to Feature Changes**
  - model's reliance on hierarchical decisions based on feature values -> small changes in the feature set can lead to different decision trees.
  - adding new features, removing existing ones, or even slight modifications to the values of the features.

= **High Variance Learners / overfitting**

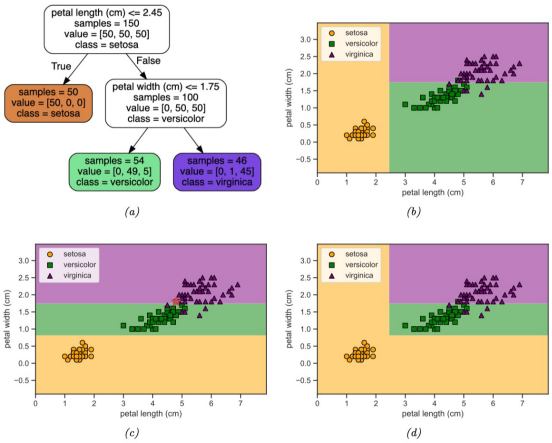


Figure 42: High Variance Learners

To mitigate: pruning + ensemble methods + careful cross validation

- **Pruning**
- **Baggning** - turns downside into a positive!
- **Random Forests and Boosting:** To mitigate some of these limitations and improve prediction accuracy, ensemble methods like Random Forests and Gradient Boosting Machines aggregate the predictions of multiple trees, each built on a subset of the data or with different initial conditions, to produce a more robust model.

61.3 How should we split nodes

depends on the type of the feature (categorical or continuous) and the specific goals of the modeling task (classification or regression)

61.3.1 Categorical: Splitting by Unique value

- **Applicability:** This method is typically used for categorical variables. The idea is to partition the data into subsets based on the unique values of the selected feature. For a feature with k unique values, this approach could potentially split a node into k branches, each corresponding to one of the feature's values.
- **Advantages:** This approach is straightforward and ensures that the model captures the effect of each category of the variable on the outcome.
- **Disadvantages:** It can lead to very complex trees, especially if the categorical variable has many levels. This complexity can make the model prone to overfitting, as it may capture noise in the training data. Moreover, with a large number of splits, some branches might end up with very few data points, making the model's predictions less reliable.

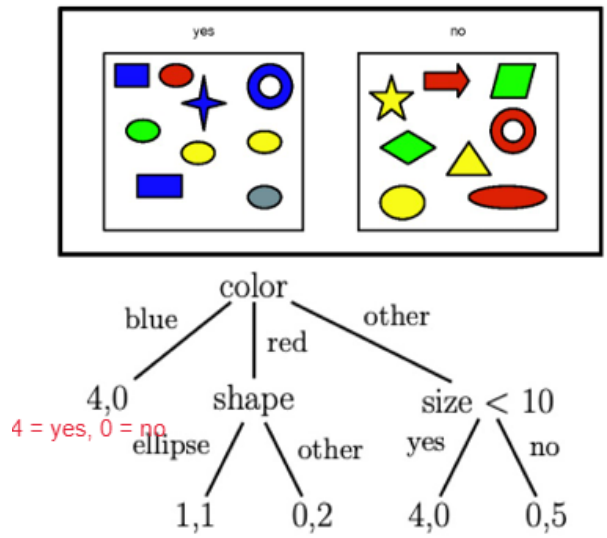


Figure 43: Enter Caption

61.3.2 Continuous: Splitting by Thresholding

- **Applicability:** Thresholding is predominantly used for continuous variables. The data at each node is split into two groups based on whether their values are above or below a certain threshold. This method can also be adapted for ordinal categorical variables by treating their ordered categories as continuous values.
- **Advantages:** This method is more scalable and generally leads to simpler trees. It's particularly effective for handling continuous variables, allowing the model to make splits that best separate the data with respect to the target variable. Thresholding helps in identifying critical values that differentiate outcomes, which can be valuable for interpretation and understanding the decision-making process of the tree.
- **Disadvantages:** The main challenge with thresholding is determining the optimal threshold for each split. This typically requires computational algorithms to search over many possible thresholds to find the one that best separates the data according to a criterion (like Gini impurity or information gain for classification, and variance reduction for regression).

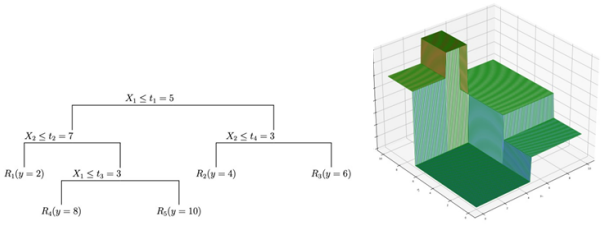


Figure 44: this is the shape we build up; ts flat within terminal nodes, discontinuous between nodes

62 How to choose this split - how to optimise a tree?

There's no efficient way to construct the best tree (computationally intractable) → in practice, we greedily recurse down the tree.

- Involves finding the partition that optimizes a certain criterion, aiming to best separate the data with respect to the target variable.
- Since the decision tree is built in a top-down manner, starting from the root and expanding down to the leaves, a greedy approach is employed at each step.
- However, **identifying the globally optimal decision tree—that is, the smallest tree that perfectly or best represents the data—is computationally infeasible for all but the simplest datasets due to the combinatorial explosion of possible trees**.

Here's how the process typically works in practice:

Greedy Recursive Splitting

1. **Start at the Root:** Begin with the entire dataset, considering all features and their possible values (or thresholds) for splitting.
2. **Evaluate All Possible Splits:** For each feature, calculate the split criterion (e.g., Gini impurity, information gain, or variance reduction) for all possible values. This involves partitioning the data according to each split and evaluating how well it separates the data with respect to the target.
3. **Choose the Best Split:** Select the feature and value that provide the best split according to the chosen criterion. This is the "greedy" part of the algorithm, as it chooses the best split at this particular step without regard to future splits.
4. **Recursively Apply to Each Child Node:** Apply steps 2 and 3 recursively to each child node created by the split. Continue this process until a stopping criterion is met (e.g., maximum tree depth, minimum node size, or if no further improvement can be made).

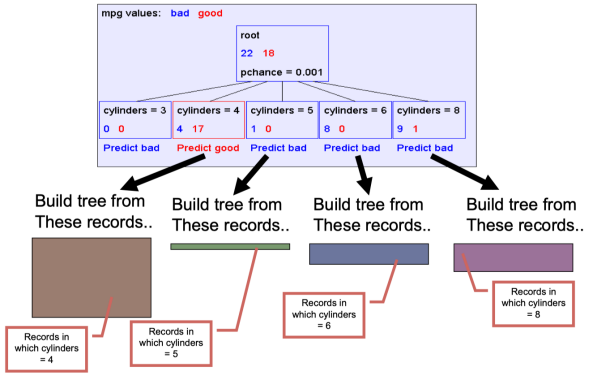


Figure 45: Enter Caption

NB: this gives us a general framework but it still doesn't tell us what is the optimum split at each location

62.1 How can we fit a tree smartly?

We are partitioning the data space into regions R_i, based on certain conditions (thresholds for the features $\theta \rightarrow$ then making predictions based on avg outcome within those regions.

Node Definition A node is defined as $R_1 = \{x : x_1 \leq t_1, x_1 \leq t_2\}$

- Units with first feature less than or equal to t_1, etc
- parameters θ define the model

Prediction Predictions for a test point x are: $f(\tilde{x};\theta) = \sum_{j=1}^{J} \hat{y}(R_j) I\hat{x} \in R_j)$.

Where $\hat{y}(R_j) = \frac{\sum_{i=1}^{n} y_i Ix \in R_j)}{\sum_{i=1}^{n} Ix \in R_j)}$

Intuitively: summing hte outcomes y of the training data that fall within the same region R_i as new prediction point x.

= just hte avg value of outcome within the node

Loss function measures measures the discrepancy between the actual outcomes y_i and the predictions $f(x_i;\theta$.

$$\begin{aligned}\mathcal{L}(\theta) &= \sum_{i}^{n} \ell(y_i, f(x_i)) \\ &= \sum_{j}^{J} \sum_{x_i \in R_j} \ell(y_i, \hat{y}(R_j))\end{aligned}$$

It is often defined as the sum of losses over all data points, where the loss for a single data point could be squared error for regression or log loss for classification, for example.
Challenge: Non-Differentiability

One major issue with decision trees, as highlighted, is that the process of selecting the best splits (i.e., optimizing the thresholds θ is inherently non-differentiable.

The thresholds define boundaries that abruptly change the membership of data points to nodes, and consequently, the predictions.

So, how to optimise?

- Greedy Recursive Splitting - At each step, the feature and threshold that result in the "best" split (according to some criterion like information gain or variance reduction) are chosen. This process is repeated recursively for each branch until a stopping condition is met
- Dynamic Programming - For a fixed tree depth, one could, in theory, use dynamic programming to explore all possible trees. However, this is often computationally infeasible except for very small trees.
- Randomization: Techniques like Random Forests introduce randomness into the tree building process by randomly selecting a subset of features to consider at each split or by bootstrapping the training data. This can help in exploring a larger space of possible models and can lead to better generalization.
- Pruning: To address overfitting and reduce the complexity of the tree, post-hoc pruning methods can be applied. These methods simplify the tree after it has been fully grown, based on criteria such as the cost-complexity trade-off.

62.2 1) Greedy Recursive Splitting - If you can't be smart, be greedy

To be greedy means to do the best thing at each step - Without thinking about the future or what might be best elsewhere
FINISH THIS NOTING

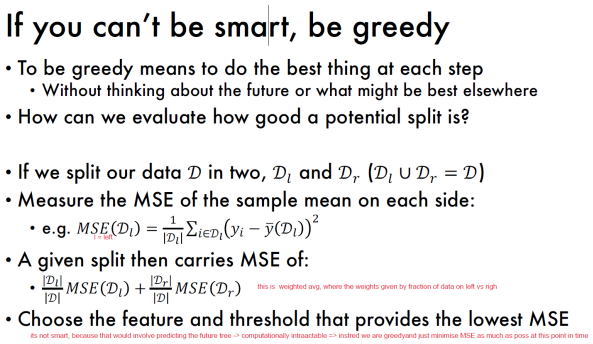


Figure 46: Enter Caption

How can we evaluate how good a potential split is?

In the context of decision trees, especially for regression tasks, evaluating the quality of a potential split involves calculating how well the split organizes the data in terms of reducing variability (or error) within each resulting node. The Mean Squared Error (MSE) is a common metric for this purpose, reflecting the average squared difference between the observed actual outcomes and the predicted values, which, at any node of the tree, is typically the mean of the outcomes in that node. Here's how this process works in a greedy algorithm framework:
Evaluating a Potential Split

1. **Splitting the Data:** Consider a dataset $\mathcal{D}$ that you want to split into two subsets, $\mathcal{D}_{\text{left}}$ and $\mathcal{D}_{\text{right}}$, based on a certain feature's threshold. The goal is to find the threshold that best reduces the overall MSE.

2. **Calculating MSE:** For each subset ($\mathcal{D}_{\text{left}}$ and $\mathcal{D}_{\text{right}}$), you calculate the MSE as follows:

   - For each point in $\mathcal{D}_{\text{left}}$ or $\mathcal{D}_{\text{right}}$, compute the difference between the point's outcome y and the average outcome $\bar{y}_{\mathcal{D}}$ of the subset.
   - Square these differences and average them over all points in the subset to get the MSE.

3. **Weighted Average of MSE:** Since $\mathcal{D}_{\text{left}}$ and $\mathcal{D}_{\text{right}}$ may not contain the same number of points, you compute a weighted average of their MSEs to account for their relative sizes. This is given by:

$$\text{MSE}_{\text{split}} = \frac{|\mathcal{D}_{\text{left}}|}{|\mathcal{D}|} \cdot \text{MSE}_{\mathcal{D}_{\text{left}}} + \frac{|\mathcal{D}_{\text{right}}|}{|\mathcal{D}|} \cdot \text{MSE}_{\mathcal{D}_{\text{right}}}$$

   Here, $|\mathcal{D}_{\text{left}}|$ and $|\mathcal{D}_{\text{right}}|$ are the sizes of the left and right subsets, respectively, and $|\mathcal{D}|$ is the size of the original dataset. The weights are thus the fractions of the dataset that go to the left and right, respectively.

4. **Choosing the Best Split:** You evaluate this weighted average MSE for potential splits across all features and possible thresholds. The best split is the one that results in the lowest $\text{MSE}_{\text{split}}$, indicating that it most effectively reduces variability within each of the resulting subsets.

Greedy Nature of the Algorithm
The decision to choose the split that minimizes the MSE at each step, without considering future splits or the overall structure of the tree that these choices will lead to, embodies the greedy nature of the algorithm. This approach ensures that, at each step, the model is optimized to reduce error as much as possible given the current structure. However, it does not guarantee that the final tree will be the optimal structure for minimizing error across the entire dataset, due to the algorithm's local, rather than global, optimization focus.
Despite this limitation, greedy algorithms are widely used for decision tree construction because they offer a practical balance between computational efficiency and model effectiveness. While they may not always find the absolute best model, they can still produce highly effective and interpretable models that perform well on a wide range of tasks.

62.2.1 Greed eventually overfits

Intuitive: if you keep splitting, you will keep reducing MSE in the training data... eventually will have $p = n$ where you perfectly predict training data....model increasingly adapts to the noise rather than the underlying pattern

62.3 2) Pruning

62.3.1 Setting Tuning Parameter to Limit Tree Growth

- **Max Depth of the Tree:** Restricting the depth prevents the tree from creating highly specific rules that only apply to small portions of the data.
- **Minimum Number of Samples per Leaf:** each leaf node represents a reasonably large subset of the data, discouraging overly specific splits.
- **Minimum Number of Samples to Consider a Split:**

Tuned through CV

62.3.2 Cost-Complexity Pruning

also known as weakest link pruning.

First allowing the decision tree to overfit the data and then pruning back the tree to improve its generalization capabilities.

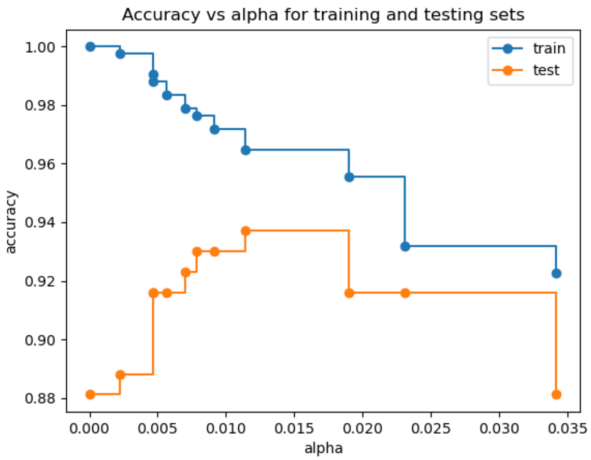


Figure 47: CCP starts out with perfect test accuracy, then slowly prunes back based on

1. **Build the Whole Tree:** Initially, allow the decision tree to grow until it perfectly fits the training data.
2. **Prune the Tree:** Evaluate each node starting from the leaves and measure the increase in error (or decrease in accuracy) that would result from pruning (removing) that node and replacing it with its most common class (for classification) or average outcome (for regression). The increase in error is weighed against a complexity parameter (α), which balances the trade-off between the tree's accuracy and its simplicity.
3. **Cutting Out Non-Beneficial Nodes:** If the error reduction (or accuracy gain) offered by a node is not greater than the complexity parameter, that node is pruned. This process is repeated recursively for each node in the tree until only nodes that provide a substantial benefit, according to the complexity parameter, remain.

Tuning the Complexity Parameter: (α) is tuned using cross-validation, where different values of α are tested to find the one that results in the best performance on a validation set or through out-of-bag error when using ensemble methods like Random Forests.

62.4 3) Randomization through Ensemble Techniques

Ensemble technique improve stability & accuracy of ML algos - esp decision trees.

62.4.1 Bagging (Bootstrap Aggregating)

Turns downside into a positive! Leverages the "wisdom of the crowd," turning the instability of individual decision trees into a collective strength that achieves higher performance and reliability.

Addresses the high variance and overfitting problems associated with decision trees by averaging multiple trees that individually may have high variance.

A foundational technique for more complex ensemble methods like Random Forests.

How Bagging Works

1. **Botostrapping = Create Multiple Datasets:** create M new datasets by sampling from the original dataset with replacement. Each bootstrapped dataset is likely to be different from the others.
2. **Fit Separate Decision Trees:** For each of the M bootstrapped datasets, fit a separate decision tree. Since each tree is trained on a slightly different dataset, they will differ from one another, capturing different aspects of the data.
3. **Sampling Characteristics:** On average, each bootstrapped dataset contains about 63% of the unique instances from the original dataset. The key to this is the bootstrapping process: there's about a 63% chance (see slides for mathematical proof)
4. **Out-of-Bag (OOB) Estimates:** For each tree, the remaining $\approx$ 37% of the training instances that were not included in its bootstrapped dataset (the out-of-bag samples) can be used as a validation set to estimate the model's performance.
5. **Aggregation:** To make a prediction for a new instance, bagging takes the predictions from all M trees and then aggregates them to form a final prediction.

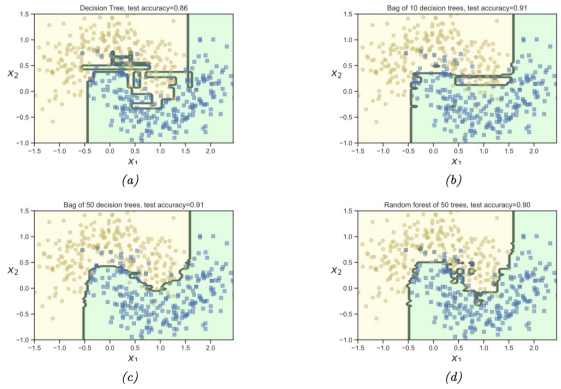


Figure 48: a) has a lot of weird shapes (overfit)... but as we start to use a 'bag' of boostrapped models, it smooths out

Advantages of Bagging

- **Reduction in Variance:** By averaging multiple trees, bagging can significantly reduce the variance without increasing bias, leading to a more robust and accurate model overall.
- **Improved Accuracy:** Aggregating the predictions of multiple trees generally leads to higher accuracy than any of the individual trees.
- **OOB Error as an Estimator:** The OOB error, calculated by using each tree's OOB samples, provides a handy and unbiased estimate of the model's performance, similar to cross-validation but without the need for a separate validation set.
- **Parallelization:** Since each tree is built independently of the others, bagging can be easily parallelized, leading to computational efficiency.

Bagging gives us useful random variation in the inclusion of samples -> beyond improving prediction 2 major advs: 1) Provides better estimates of error; 2) Allows estimation of variance of predictions

1) Averaging over multiple poor models can be very good! Each one needs to be better than useless, but very different from one another. **Injects randomness**

NB - ideal number of models needed isn't always clear: tune with cv.

2) Using bagging to estimate variance

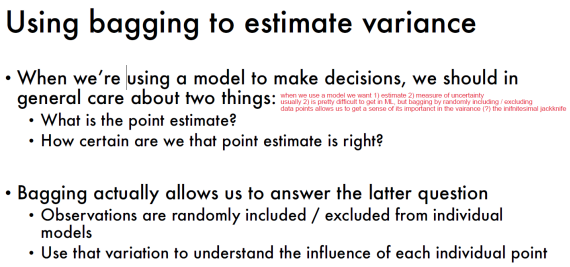


Figure 49: Enter Caption

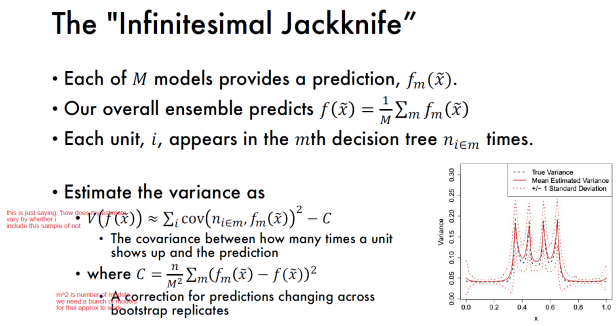


Figure 50: Enter Caption

62.4.2 Random Forests

= bagging on speed: add further improvements such as **feature subsampling** to increase the diversity among the trees.

Random Forests extend the bagging technique by adding an extra layer of randomness to the tree-building process.

- not only involves creating multiple trees on bootstrapped datasets (bagging)
- but also randomly selects a **subset of features** at each split in the construction of the trees (we "hold out" features at each node)

introduces more variance/diversity among the learners/trees in the forest -> stronger overall model.

Individual trees are worse - not always considering all features for every split, each tree in the Random Forest is likely to have a higher bias....

... means that the trees are less correlated with each other. **When their predictions are aggregated, the variance of the combined model is reduced**. (Aggregated by averaging for regression or voting for classification)
Random forests can represent complex relationships very well

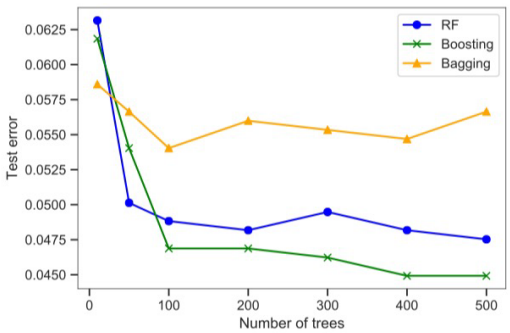


Figure 51: as you add models random forest starts to do better: this is because these trees are all injecting a degree of randomness; if there's good structure in your data and your features are meaningful,then you can avg over a bunch of them and you get good results

Random forests are basically an **atheoretical heuristic** that just happens to work really well.

63 Demonstration - Random Forests & power of out-of-bag

in all of those rectangles: it takes avg within them decision tree function is basically a pixelated version

out of bag estimate - almost as good as a test set, or perhaps a more accurate comparison is to a CV set

the out of bag sample allows you to work out how each data point influences the model (?)

infintesimal boostrap: if you were to remove this one point in this area, how would hte prediction change.... sothis means that our stnadard error is not constant across the functiin„„ in areas where we have less/more data?, we will get inflated variance

i think it's areas where the slopw is steeper we have higher variance because things are changing a lot so small changes in data point will have big impact whereas where slopw is flat, variance in predictions will be less

looks at correlation between trees - if highly correlated then in the data we have high correlations iun preductions between trees - this is not great we want less correlation between the predictions ... this motivates what we're about to look at nxt

ML Lecture Notes: Wk 8-I — Boosting

64 Boosting/Ensemble Learning & Correlated Errors

64.1 Motivation: errors of trees are correlated!

The main issue encountered in ensemble learning, particularly with methods that rely on decision trees, like Random Forests and Gradient Boosting Machines (GBMs): is the correlation of errors among the trees in the ensemble.

Despite efforts to diversify the trees—through methods like bootstrapping the data (as in Random Forests) or by manipulating input features — the errors made by individual trees can still be correlated.

This correlation can diminish the ensemble's ability to reduce overall error through averaging or combining individual tree predictions.

64.1.1 Core issue in Ensemble Learning Basics

combine the predictions of several base estimators (like decision trees) to improve generalizability and robustness over a single estimator.

Core issue: if the base learners (trees) make similar errors, then these errors will reinforce each other when their predictions are combined, instead of cancelling out.

Ensemble model's main strength is its ability to reduce the variance component of the error by averaging out uncorrelated errors from diverse models. However, if errors are correlated, this variance reduction is less effective.

64.1.2 Ensemble Prediction Function

Formalising ensemble method in prediction function:

NB!

For each tree (weak learners):

$$f(x;\theta,w) = \sum_{m=1}^{M} w_m F_m(x;\theta)$$

Where

- $F_m(x;\theta)$ represents the individual tree predictions
- w_m are the weights assigned to each tree's prediction
- $w_m = \frac{1}{M}$ - in simple ensemble methods like bagging, each tree has equal weight.

However, when we fit each $F_m(\cdot)$ independently, we can't ensure they do different things; despite being trained on different samples or with different features, they end up making similar mistakes due to overlapping regions of error.

64.1.3 Solution

1. Fit $F_1(\cdot)$ as normal,
2. Fit $F_2(\cdot)$ on reweighted (or residual-focused) data that emphasizes the errors made by $F_1(\cdot)$.

i.e. subsequent trees focus on learning different things than (i.e. correcting the errors of) the prev tree.

This is the core of Boosting algorithms, such as AdaBoost and Gradient Boosting. These involve a sequential learning process.

- Step 1 is always same
- How subsequent steps are handled depends on method: each reweight or remodel the data to emphasize the errors made by the previous models in the sequence.
  - **AdaBoost** - this involves increasing the weights of the instances that were misclassified by the previous models, making them more 'important' for the next model to correct.
  - **Gradient Boosting** - each new model is trained on the residuals (the differences between the observed and predicted values) of the previous models, effectively focusing on correcting the errors made by the ensemble so far.

This reduces correlation between the errors of the individual trees -> more robust & accurate ensemble methods.

65 Intuition of Boosting

An ensemble technique that focuses on minimizing a loss function by sequentially adding models that predict the residuals or errors of the ensemble thus far.

1. Fit a model
2. Calculate residuals
3. Fit a model to the residuals
4. Repeat

The residuals are our mistakes. So at each iteration, we are fitting a model that will correct the mistakes we made.

Mathematically we can directly add these together (although in practice we don't actually directly add them together: we use β parameter as a learning rate to avoid fitting too closely to the data in each model.

By down weighting a single decision tree, we are...(?)

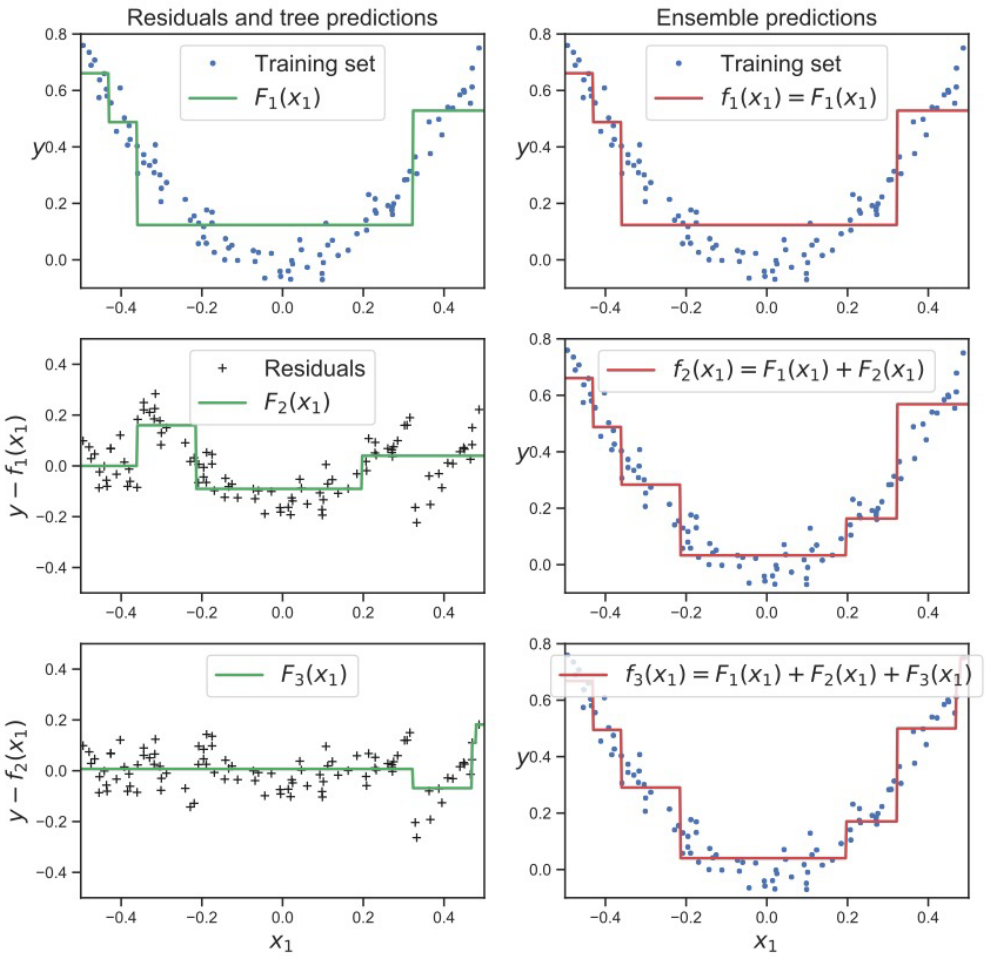


Figure 52: Green line is a single decision tree; Red line is combined trees. NB Green line is not that deep-expressive: you might only have a depth of 1-2; we're not doing so much learning with each decision tree, but when we combine them together we have depth. Cross validation: decide how deep each tree, how many iterations.

65.1 Generic Loss function at iteration m

$$\beta_m, \theta_m = \arg\min_{\beta,\theta} \sum_{i=1}^{N} \mathcal{L}\left(y_i, f_{m-1}(x_i) + \beta F(x_i;\theta)\right)$$

Where:

- $f_{m-1}(x_i)$ - previous prediction (in iteration $m-1$)
- β - Tradeoff between previous and current prediction
- $F(x_i;\theta)$ - current function (tree) being optimizes

What's going on:

- $\ell(y, \hat{y})$ is the loss function comparing the true labels y with the predictions $\hat{y}$
- $\hat{y} = f_{m-1}(x_i)$ is the prediction from the previous iteration,
- the minimization argument is minimising over two sets of parameters

Take the prediction from the previous set of trees, then add that into the prediction of the current tree, but weight it by β

Big F the **Strong learner** = all trees.I DON'T THINK THIS IS RIGHT?

Small f the **Weak learner** = individual trees.

We are not trying to fit a perfect model each time; rather we are trying to reduce the error a bit each time...

... if we make M big enough, we will reduce a lot of error: intuitively, if each of your weak learners is able to do a bit to help you, then eventually you can combine to get a strong learner.

The logic of boosting: you can do a better job by iteratively stacking weak learners, than fitting just one strong learner.

This works extremely well. In practice, most models are boosted models - they are extremely robust.

65.2 Double Optimization Process

The optimization process involves two key steps at each iteration:

Fitting the New Model to Residuals: For each observation i, calculate the residual from the previous iteration's prediction. Then, fit a new model $f(x;\theta)F(x;\theta)$ to these residuals. This step focuses on learning from the mistakes of the ensemble thus far.

Finding the Optimal β: Once $f(x;\theta)F(x;\theta)$ is fitted to predict the residuals, the next step is to find the optimal scaling factor β that minimizes the overall loss when the predictions of $f(x;\theta)F(x;\theta)$ are added to the previous ensemble's predictions. This involves a line search to find the value of β that best reduces the loss.

66 Least Squares Boosting

A specific Gradient Boosting Machine (GBM) - common for regression problems.

Squared Loss Function: For least squares boosting, the loss function is specifically the squared error between the true and predicted values,

$$\ell(y, \hat{y}) = (y - \hat{y})^2$$

Insert this into the prev generic loss function at iteration m:

$$\mathcal{L}\left(y_i, f_{m-1}(x_i) + \beta F(x_i;\theta)\right) = \left((y_i - f_{m-1}(x_i)) - \beta F(x_i;\theta)\right)^2$$

In which $(y_i - f_{m-1}(x_i))$ is the residual from the previous iteration = *essentially predicting the error of the previous model, thereby correcting it.*

The goal at each step is to reduce the discrepancy between the observed values and the ensemble's predictions by adding a new predictor $\hat{y}(x;\theta)F(x;\theta)$ that corrects the residuals from the previous step.

Take the residual from the previous prediction, then fit a model on that.

NB the previous model is *all of the previous trees* - it is combining them recursively

67 AdaBoost

NB!

Essentially: if model misclassifies something, weight up next time (using exponential loss function)!

67.1 Binary Classification & Encoding

Not so important
AdaBoost typically deals with labels y that are encoded as $\{-1, 1\}$.

This encoding facilitates mathematical manipulation, especially in the context of loss functions that compare predictions ($F(x)$) to actual labels (y).

To transform binary labels $\{0, 1\}$ into a $\{-1, 1\}$ encoding, we use a transformation: $y = (\hat{2y} - 1)$:

$$\begin{aligned} y = 0 \qquad & \Rightarrow \hat{y} = (2 \times 0 - 1) = -1 \\ y = 1 \qquad & \Rightarrow \hat{y} = (2 \times 1 - 1) = 1 \end{aligned}$$

AdaBoost models produce predictions $F(x)$ that can take any real value ($-\infty$ to $+\infty$). To interpret these predictions as probabilities, a logistic function (or sigmoid function) σ is used:

$$\sigma(F(x)) = \frac{1}{1 + \exp\{2F(x)\}}$$

This function maps the real-valued predictions to the $(0, 1)$ interval, providing a probability measure.

67.2 Use of Exponential Loss Function

AdaBoost uses Exponential Loss Function

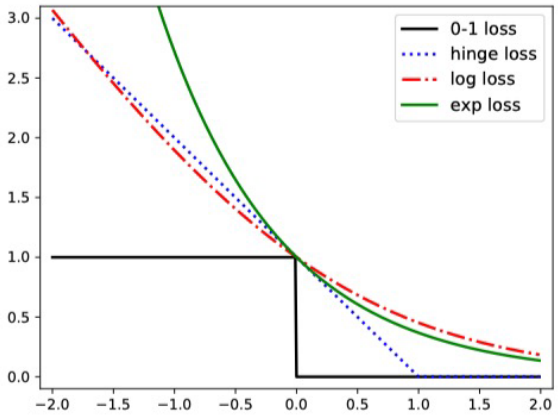


Figure 53: Shows exponential loss function is closest to trie loss function??? wile remaining differentiable?

67.2.1 Possible Loss Functions

Log-Loss (or Logistic Loss): This is commonly used in logistic regression and measures the discrepancy between the actual labels and predicted probabilities. It is given by:

$$\ell_{\log}(y, F(x)) = \log(1 + \exp(-2yF(x)))$$

The factor of 2 is included to adjust for the $y \in \{-1, 1\}$ encoding. This loss encourages predictions $F(x)$ that are consistent with the actual labels y, penalizing incorrect predictions more heavily as the discrepancy increases.

Exponential Loss: Used by AdaBoost, the exponential loss function is:

$$\ell_{\exp}(y, F(x)) = \exp(-yF(x))$$

This loss function similarly aims to penalize discrepancies between the labels and predictions but does so in an exponential manner. The key characteristic of the exponential loss is that it heavily penalizes models that are confident and wrong. Unlike log-loss, which asymptotically approaches its maximum penalty, the exponential loss can grow without bound as the prediction moves away from the actual label.

The magnitude of this loss depends on the loss function being used. For both the exponential loss and log loss functions, incorrect predictions are penalized, but the severity of the penalty differs:

1. **Exponential Loss:** This function imposes a higher penalty on incorrect predictions compared to log loss. The penalty increases exponentially with the degree of incorrectness of the prediction, thus heavily penalizing predictions that are far from the true outcome. This characteristic of exponential loss drives the model to prioritize the correction of incorrect predictions to a greater extent.

2. **Log Loss**: While also penalizing incorrect predictions, log loss does so in a manner that is less aggressive than exponential loss. The penalty increases logarithmically, meaning that while incorrect predictions are discouraged, the model is not penalized as harshly for predictions that are wrong.

Both serve as effective surrogate loss functions; minimizing either loss function over a sufficiently large dataset tends to lead the model towards similar performance outcomes.

However, from a mathematical optimization perspective, log loss is generally considered easier to minimize due to its smoother gradient.

67.2.2 AdaBoost and Exponential Loss

The use of the exponential loss in AdaBoost is strategic:

- **Weight Update Mechanism:** The exponential loss directly influences the way weights are updated for each observation in the training dataset. Observations that are misclassified by the current ensemble are assigned exponentially higher weights, making them more critical for the next model to get right.

- **Model Fitting:** Each new model in the AdaBoost sequence is fitted to correct the errors of the ensemble thus far, with a focus on the observations that the current ensemble finds most challenging. This is operationalized by the weights, which are updated according to the exponential loss.

- **Error Correction Focus:** The exponential nature of the loss ensures that as the sequence progresses, the boosting algorithm focuses more on the hardest to classify observations. This focus enables AdaBoost to achieve high accuracy, even when individual models in the ensemble are weak learners.

The exponential loss function is fundamental to how AdaBoost iteratively improves the ensemble model, emphasizing the algorithm's adaptiveness and its capacity to concentrate on the most challenging aspects of the training data.

67.3 (discrete) AdaBoost

An iterative process of minimizing the exponential loss for binary classification tasks.

Iteration m: AdaBoost aims to find a weak learner (model) $F(x)$ that, when added to the ensemble of previous learners, minimally increases the overall exponential loss:

$$L_m(y, f_m) = \sum_{i=1}^{N} \exp(-y_i(f_{m-1}(x_i) + \beta F(x_i))$$

This can be rewritten to highlight the weights w_{im} at iteration m, emphasizing that each sample is weighted based on the ensemble's performance on it up to the last iteration:

$$= \sum_{i=1}^{N} \omega_{im} \exp\{-\beta \tilde{y}_i F(x_i)\}$$

Weights ω_{im}: The weight for each sample i at iteration m, $\omega_i^{(m)}$, is defined as

$$\exp(-\beta \tilde{y}_i f_{m-1}(x_i))$$

This represents the loss due to the previous iteration's ensemble prediction. This makes samples that were harder to classify in the previous iterations more significant in the current iteration.

NB!

weights = loss due to prev iteration's ensemble prediction.

It's nice because they are a direct function of the (prev) loss, so we only have to minimize one argument F_m.

Loss Simplification: The loss can be further simplified by separating it into parts based on whether the prediction $f_{m-1}(x_i)$ matches the actual label y_i.

...Incorrect predictions are penalized more heavily, leading to the update rule for the ensemble model that focuses on minimizing the weighted sum of incorrectly classified samples.

Choosing F_m: The model F_m for iteration m is chosen such that it minimizes the weighted error of misclassification, effectively focusing on the samples that the previous model found challenging:

$$F_m = \text{argmin}_F \sum_{i=1}^{N} \omega_i^{(m)} I(y_i \neq F(x_i))$$

Update Rule for β_m: The amount β_m by which to adjust the contribution of F_m is determined by its weighted classification accuracy, with

$$\beta_m = \frac{1}{2} \log\left(\frac{1 - \text{acc}_m}{\text{acc}_m}\right)$$

where acc_m is the weighted accuracy of F_m.

slides ?:

$$\beta_m = \frac{1}{2}\log\left(\frac{\%}{\%'}\right)$$

Where

$$\text{acc}_m = \sum_{i=1}^{N} 2^{\omega_i^{(m)}\mathbb{I}(y_i \neq F(x_i))} \Big/ \sum_{i=1}^{N} 2^{\omega_i^{(m)}}$$

We are weighting samples up by omega, so that the implicit beta is the ratio of the accuracy: misclassification rate (we then take the log and halve to get the learning rate).
This formula ensures that more accurate models have a greater influence on the ensemble prediction.

Weight Update for Samples: After determining F_m and β_m, the weights of the samples are updated to reflect the new ensemble's performance, making the next iteration focus on samples that are still misclassified.

In sum, for our weaker learner we reweight the loss function by the exponential loss of the prev iteration.

So where we had large loss before, we weight that up by te size of that loss.

This is why the exponential loss function is nice - it reweights it up

Discrete AdaBoost effectively combines multiple weak learners into a strong ensemble classifier by iteratively focusing more on the samples hardest to classify.

By adjusting weights based on previous errors and choosing each subsequent model to minimize these weighted errors, AdaBoost dynamically adapts to the challenges presented by the training data.

This process not only improves the ensemble's overall accuracy but also provides a robust mechanism against overfitting, given the emphasis on correcting misclassified samples.

68 Gradient Boosting

NB!

More generalizable! Doing gradient descent on functions - focusing on the residuals or the gradients of a wide range of loss functions

NB

- Least Squares Boosting is a form of gradient descent that specifically uses the squared error loss function (esp suited to regression)
- AdaBoost is another specific instance of gradient boosting primarily focused on classification tasks and specifically designed to increase the weight of training instances that are

misclassified by the previous models (using exponential loss function), thereby focusing subsequent models on the hard cases.

- Gradient boosting is more generalizable: its use of gradient descent on the loss function provides a systematic and generalizable approach to minimizing prediction error, which contrasts with AdaBoost's focus on correcting misclassified instances through weight adjustments

68.0.1 Conceptual Overview

- **Objective:** The goal is to find a function f^* that minimizes the loss function $\mathcal{L}(f)$. This is akin to seeking the best approximation of the target variable as a function of the input variables.
- **Gradient of Loss Function:** The gradient of the loss function with respect to the predictions, denoted g_m, points in the direction of the steepest increase in the loss. By moving in the opposite direction, we can reduce the loss. The gradient is calculated as:

  $$g_m = \nabla_{f_{m-1}} \ell(y_i, f_{m-1}(x_i))$$

  This gradient represents the direction in which we should adjust our predictions to minimize the loss.
- **Function Update with Gradient Descent:** The estimated function f_m is updated by taking a step in the opposite direction of the gradient:

  $$f_m = f_{m-1} - \beta_m g_m$$

  Here, β_m is the step length or learning rate, which determines the size of the step taken in the direction of the negative gradient.
- **Step Length (Learning Rate) β_m:** In the ideal theoretical setup, β_m is chosen to minimize the loss given the current direction of the step, g_m:

  $$\beta_m = \text{argmin}_\beta \mathcal{L}(f_{m-1} - \beta g_m)$$

  However, in practice, β_m is often treated as a hyperparameter that is set prior to the start of training. It controls the speed of convergence and can help prevent overfitting by making smaller adjustments to the model at each step.

68.0.2 Practical Implementation

In practice, each g_m is represented by a new model (usually a decision tree) that is fitted to the current residuals (or gradients). The ensemble is updated by adding this new model, scaled by the learning rate β_m, to the existing ensemble. This process is repeated for a specified number of iterations or until convergence.

- **Residual Fitting:** Instead of directly computing and working with gradients, gradient boosting typically involves fitting new models to the residuals of the previous model predictions. For regression problems, these residuals are the differences between the observed values and the ensemble predictions. For classification, they are related to the gradient of the loss function with respect to the ensemble predictions.
- **Hyperparameter Tuning:** The learning rate β_m, along with other parameters like the depth of the trees and the number of trees in the ensemble, are crucial hyperparameters that need to be tuned to achieve the best performance.

Gradient boosting leverages the concept of gradient descent in function space by iteratively reducing the loss, leading to a highly flexible and powerful modeling approach that can capture complex relationships in the data.

69 XGBoost - eXtreme Gradient Boosting

Quasi-gradient descent

NB!

Adds a bunch of hyperparameters for regularization tuning + that allow us to do Cross Validation easily

- **Hyperparameters Tuning:** XGBoost provides a wide range of hyperparameters that can be finely tuned to optimize performance.
- **Regularization:** Unlike traditional gradient boosting, XGBoost incorporates regularization terms directly into the objective function to control overfitting. The regularization term is given by:

$$\gamma J + \frac{1}{2}\lambda \sum_{j=1}^{J} w_j^2$$

  where

  - J is the number of leaves in a tree
  - w_j are the leaf weights,
  - γ and λ are regularization parameters that penalize the complexity of the model.

  So, in addition to defining the depth of the tree - it also allows us to add a regularizer that adds a penalty for each split.

  - this is more continuous
  - whereas before it was a binary decision: split or not

- **Second-Order Approximation:** XGBoost uses a second-order approximation of the loss function, rather than just the first-order. This allows for a more accurate estimation of the optimal step size and direction when optimizing the loss function.

  - curvature of the function as well as gradient
  - better understands the loss landscape

  And integration of this into the splitting criteria...

- **Enhanced Splitting Criteria:** The integration of the second-order approximation into the splitting criteria for the decision trees leads to more optimal splits, further improving model accuracy and training speed.

- **Feature Sampling at Nodes:** Similar to the strategy employed by Random Forests: sampling features at each node before determining best split.

  - contributes to diversity f the models in the ensemble
  - reduces overfitting
  - speeds up computation -> scalable

- **Comparison with Random Forests:** Radom forests is to bagging, what XGBoost is to boosting.

  - While Random Forests build trees independently using bagging,
  - XGBoost sequentially builds trees that complement each other using boosting.

70 Introspection into Complex Models

70.1 Feature Importance

> Which features are used to make predictions?

Quantifies the contribution of each feature to the predictive power of the model, providing insights into which features are most informative for making predictions.

One common method to compute feature importance in tree-based models is by using the gain in model accuracy attributed to the kth feature. This can be represented as:

$$R_k(T) \sum_{j=1}^{J-1} G_j I(v_j = k)$$

where:

- $R_k(T)$ is the feature importance of feature k in tree T,
- J is the number of nodes in the leaves/tree,
- Gain_n is the gain in accuracy at node j,
- $(v_j = k)$ indicates that node j splits on feature k,
- $\text{Indicator}(v_j = k)$ is an indicator function that is 1 if node j splits on feature k and 0 otherwise,
- T is the total number of trees in the ensemble,

?
Then we avg over trees, and nromalise to sum to 100.

$$R_k = \frac{1}{M} \sum_{m=1}^{M} R_k(T_m)$$

Here R_k is the average importance of feature k across all trees.

This gives us a rank of feature importance.

NB - Feature Importance does NOT tell you about direction, it just tells you how the accuracy went up when it split on 'George' (see above): when split on "George" the accuracy went up, but we can't say inclusion of George is more or less likely to make it spam. There's lots of interaction effects going on in this model - could be that in some cases inclusion of "George makes it much less likely to be spam, whereas other times it is not - it depends on interaction effects

70.2 Partial Dependency

> How does the prediction change as this feature varies?"

A way to visualize the effect of a single feature on the predicted outcome, holding all other features constant.

Particularly useful for interpreting the behavior of complex models.

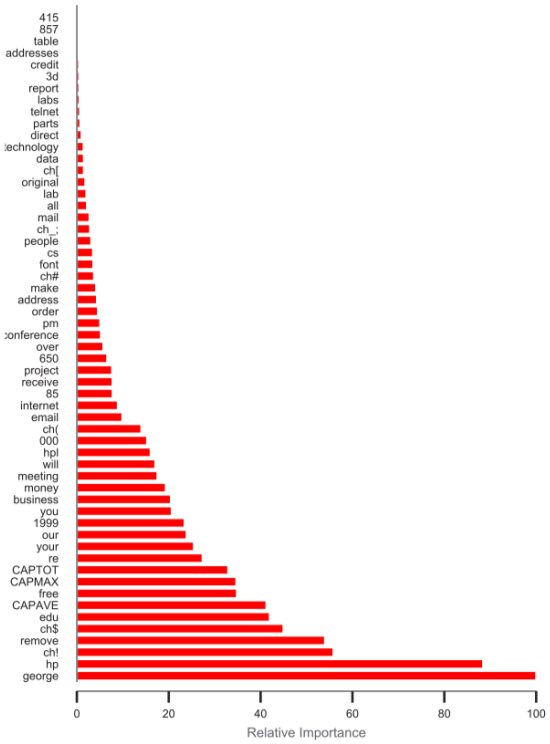


Figure 54: Feature Importance for spam filter, where features are words

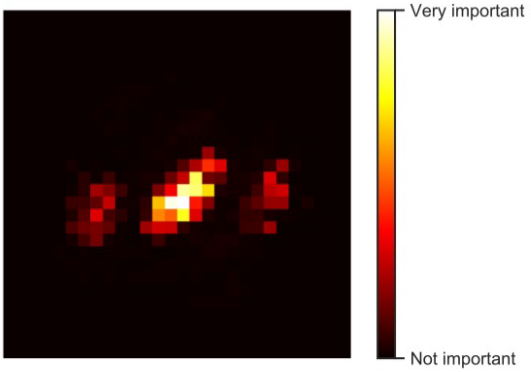


Figure 55: Feature Importance for number classifiction: 3 vs 8. Intuitively, it's the middle pixels which are most useful

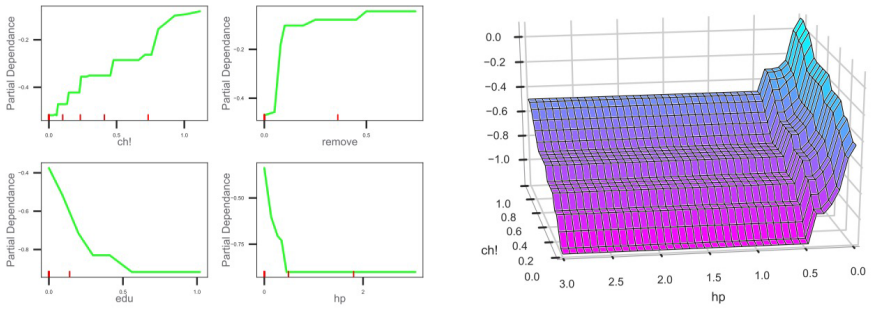


Figure 56: Partial Dependency visualised

$$\bar{f}(x_k = x) = \frac{1}{N} \sum_{i=1}^{N} f(x_i \mid do(x_k^{(i)} = x))$$

Prediction of the model when the feature x_k is held constant at a specific value, and the other features $x^{(i)}$ take on their values from the dataset.

This function averages the predictions over all data points, with the feature of interest set to a particular value, effectively tracing out the relationship between that feature and the predicted outcome.

The only thing we're changing in the the feature vectoris the open feature -> what is the effect of this feature on the prediction.

71 Demonstration - Boosting

72 Takewaway

Tree-based models are robust, scalable and amenable to tuning (lots of useful hyperparameters).

They are one of the most important workhorse models you will learn about

Rarely should you use something more complex than them without benchmarking against them first.

Many models are just gradient boost models. Unlike neural nets they are much easier to tune: even if you get the learning rate a little wrong they still work - whereas a neural net if you get the learning parameter a little wrong, it will just fail to converge.

ML Lecture Notes: Wk 8-II — Sampling

How do we choose the data we collect?

3 Different angles on sampling

1. Sampling to get a better model:
    - Active learning
    - Leverage score sampling
2. Sampling to improve rewards
    - Bandit algorithms
3. Sampling to measure prevalence (eg we want to know how much hate speech is on the internet - this is actually a hard question to answer (cannot measure it all) - might want to use a model
    - AIPW

73 Explanation vs Prediction

Explanation Objective: to understand the impact of a feature(s), A on an outcome, Y.

About uncovering causal relationships or explaining variations in the outcome based on changes in the features.

In sampling, this might involve selecting samples that provide the best information about these causal relationships, often requiring careful design to avoid confounding factors.

How?

1. Improve the causal inferences you can make with a dataset: Observational methods
2. Improve the dataset you will use to make causal inferences: Experimentation

Prediction Objective: To predict an unseen outcome Y based on observed features X.

Aim is to build models that can generalize well to new, unseen data.

How?

1. Improve our models for prediction: Most of this class
2. Improve the dataset you will use to predict: this lecture...

74 Set up

We have N observations of features X.

How should we choose n observations to measure the label?

> Assumption: it is easy for us to measure on X, but measuring on y is hard.
>
> eg detecting hate speech online - the X part is easy, but the y part requires a human to look at it and make a judgement.
>
> Often y is just harder

Or, perhaps we have n_0 observations of (X, y).

How should we choose the next n_1 observations, based only on X?
...3 Different angles on sampling

1. Sampling to get a better model:
    - Active learning
    - Leverage score sampling
2. Sampling to improve rewards
    - Bandit algorithms
3. Sampling to measure prevalence
    - AIPW

75 Caution

Taking a non-random sample to collect your training data may be dangerous...

- might miss certain areas of the feature space
- can introduce bias into the training data, meaning the model might learn patterns that are not generally applicable to the broader population or dataset. This can occur if the sample over-represents certain groups or characteristics while under-representing others.

> On the training data we're a bit more free...
>
> ... whereas for the testing data, we really need it to be a random sample - no leeway

Taking a non-random sample for your testing data is, in general, a very bad idea - You might have no idea how poorly your model behaves!

- skewed representation: might not accurately reflect the diversity or distribution of the broader dataset or population

76 Data Leakage

76.1 What it is

When you mistakenly use ”too much” information in predictions.

When information from outside the training dataset is used to create the model.

This information inadvertently informs the model about the target variable, leading to predictions that are not based on the inherent patterns in the training data but on this "leaked" information.

As a result, the model may appear to perform exceptionally well during training and testing phases but perform poorly on truly unseen data.

76.2 How to protect: understand the ’production’ task you are solving

A thorough understanding of how the model will be used in production is crucial. Knowing the data available at prediction time helps ensure that only appropriate features are used during training.

The link between training and test should be the same as in the applied problem you are solving

Features that will not be available at prediction time should not be included in the model.

is the way that i am defining features respecting the features i will actually have with me in production?

76.3 Examples

Using data (labels) from the future.

Using repeated measures of the same unit (e.g. pupils in the same school)

77 Random Sampling

77.1 Why random sampling for training sets?

We care about population risk:

$$\mathbb{E}\left[L(y, f(x))\right]$$

- Expected loss over the entire distribution of the data population.
- It quantifies how well a model $f(x)$ predicts outcomes $y$across all possible inputs x, weighted by their likelihood of occurrence.
- measures the average performance of a model on the entire data population, not just on a sample.

77.2 Approx through ERM

Population risk is a theoretical measure, which we can't know (don't have access to whole population) -> we approximate when we estimate ERM on samples of the population.

$$\frac{1}{n}\sum_{i=1}^{n}\ell(y_i, f(x_i))$$

Minimise avg loss over training set.
But what if some parts of the model are harder to learn?

77.3 Challenge of Heteroskedastic Noise

Some parts of the model are harder to learn

$$y_i = \beta X_i + \epsilon_i$$

where $\epsilon_i \sim N(0, \sigma^2(X_i)))$
Variance of the error terms (ϵ_i) varies across observations -> makes it more difficult to learn certain parts of the model.

Areas of X with higher noise ($\sigma^2(X_i)$ is large) are harder to predict accurately compared to areas with lower noise.

Random sampling:

- **Bias Reduction:** It helps in minimizing bias by giving all data points an equal chance of being included in the sample, thus reflecting the true distribution of the population.
- **Variance Understanding:** It enables the model to learn from various parts of the data distribution, including both high-noise and low-noise regions. This enhances the model's ability to generalize from the training data to unseen data.
- **Model Robustness:** It ensures that the model is exposed to and learns from the diversity present in the entire dataset, including any underlying patterns or relationships, making it more robust and accurate.

77.4 The big idea behind non-uniform (adaptive) sampling strategies

Density of data points in different areas of the covariate (feature) space impacts model accuracy –> why uniform sampling may not always be the optimal approach!

Areas with More Data

Improved Accuracy/Better Kearbubg: In areas of the feature space where data is abundant/dense, models tend to have higher accuracy. This is because more data provides a clearer signal for the model to learn from, reducing the uncertainty and variance in model predictions for these areas.

Unequal Sampling Consideration

Non-Uniform Distribution of Information: Not all regions of the feature space contribute equally to model performance. Some areas might be critical for understanding complex phenomena or capturing rare events, even if they are less densely populated with data points.

Heteroskedasticity and Noise: In the presence of heteroskedastic noise, where the variability of errors differs across the feature space, uniform sampling might not capture the full spectrum of variability effectively. Areas with higher noise might require more data to achieve a comparable level of model accuracy to lower-noise areas.

Sampling Strategy: Recognizing these aspects, it might be beneficial to adopt a non-uniform sampling strategy. This could involve oversampling in less populated but critical areas or areas with higher noise to ensure the model learns these parts of the feature space effectively.

Adaptive Sampling Techniques

Several adaptive sampling techniques can be used to address these considerations, such as:

- **Stratified Sampling:** Dividing the feature space into strata based on certain characteristics (e.g., density, noise level) and sampling more from underrepresented or critical strata.
- **Importance Sampling:** Weighting data points based on their importance or contribution to model learning, which can help focus model training on more challenging or informative parts of the feature space.
- **Active Learning:** Dynamically selecting data points for labeling and inclusion in the training set based on the model's current performance and uncertainties, focusing on areas that would most improve the model.

Conclusion

The recognition that not all areas of the covariate space contribute equally to model accuracy—and that uniform sampling may not always be optimal—underscores the importance of thoughtful sampling strategies in model development.

78 Active Learning

Optimizing the training process by selectively choosing the most informative data points for labeling and inclusion in the training set.

Particularly valuable when labeling data is expensive or time-consuming

78.1 Process

1. **Initial Model Training** - preliminary model trained on a small set of labeled data: $\{(X_i, y_i)\}_{i=1}^{N}$.
    - his initial model serves as the basis for making decisions on what data to label next.
2. **Receiving Labels** - labels for data points come from a distribution $p(X, y)$. In practical scenarios, this often involves requesting labels from human experts or through experiments.
3. **Model Update** - updated or retrained with the newly labeled data, improving its understanding and predictions
4. **Selecting next X to Label** - updated or retrained with the newly labeled data, improving its understanding and predictions

78.2 Criteria for Selecting Data Points

Selection of X in active learning is guided by the goal of reducing population risk, meaning we aim to improve the model's accuracy across the entire data distribution.

- **Uncertainty Sampling:** Choose data points for which the current model has the highest uncertainty in its predictions. This often involves selecting points closest to the decision boundary of a classifier.
- **Query by Committee:** Maintain multiple models (a committee) and choose data points where there is the most disagreement among the committee members. This approach is based on the premise that areas of high disagreement are likely to be the most informative for training.
- **Expected Model Change:** Select data points that, when labeled and added to the training set, are expected to result in the most significant change or improvement in the model.
- **Expected Error Reduction:** Choose data points that are expected to most reduce the overall error of the model on the unlabeled dataset.
- **Density-Weighted Methods:** Combine uncertainty with the density of data points in the feature space, preferring data points that are not only uncertain but also representative of the data distribution.

78.3 Uncertainty Sampling

Uncertainty sampling is a strategy used in active learning to select the most informative data points for labeling based on the model's uncertainty about its predictions.
Suppose multi-class classification -> focusing on data points that the model finds difficult to classify.

3 ways to determine uncertainty:

1. **Maximum Entropy**
   - **Formula:** $-\sum_c p_c \log p_c$
   - **Description:** Entropy measures the uncertainty or disorder within a system. In the context of multi-class classification, it quantifies the model's uncertainty across all possible classes. A higher entropy value indicates greater uncertainty, meaning the model's predictions are spread out over several classes rather than concentrated on one.
     - a measure of how widely dispersed our are predictions x log probability of being in that class
     - we would sample examples with high values
   - **When to Use:** This method is most effective when you want a holistic measure of uncertainty that considers all classes equally. It's particularly useful when any misclassification is equally undesirable, and you're interested in data points where the model lacks clear direction.
2. **Margin Sampling**
   - **Formula:** $\min(p_c - p^*)$ where p_c is the probability of the most probable class.

- **Description:** The margin measures the difference between the model's confidence in the most probable class and the second most probable class. A smaller margin means the model finds it difficult to distinguish between the top two classes, indicating higher uncertainty.
- **When to Use:** Margin sampling is useful when the decision boundary between classes is of particular interest. It focuses on examples where the model is nearly equally split between two classes, making it valuable for refining class boundaries.

3. **Least Confident**

- **Formula:** $1 - p^*$
- **Description:** This method considers only the model's confidence in its most probable class. The less confident the model is about its top choice (meaning p^* is low), the higher the uncertainty. Essentially, it measures how much the model's confidence falls short of absolute certainty.
- **When to Use:** Least confident sampling is straightforward and effective when the main concern is with the model's top prediction. It's best used in situations where the priority is to boost the model's confidence in its predictions, focusing on data points where the model is most tentative about its top choice.

Uncertainty sampling is used to determine which examples to send to a human to encode.

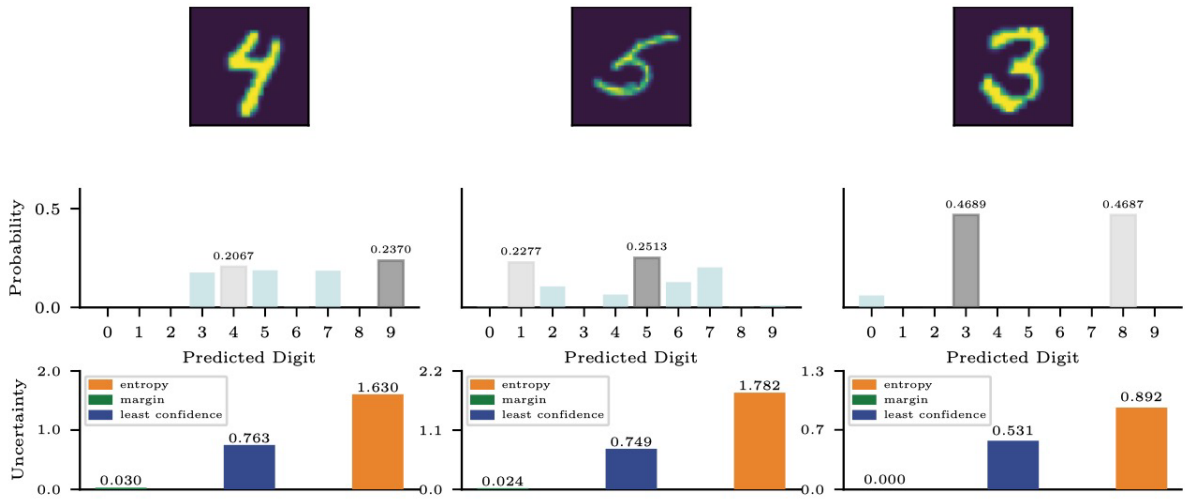


Figure 57: Enter Caption

78.4 Bayesian Active Learning by Disagreement

In response to objections to other active learning strategies.

Leverages Bayesian inference to quantify uncertainty in a model's predictions, focusing on selecting data points that, once labeled, are expected to yield the largest reduction in this uncertainty.

Fundamental aim is to select data points for which the model's prediction would most reduce the model's overall uncertainty.
Operationalized by identifying points where the model parameters θ, conditioned on the current data D and a potential new data point (x, y), show the greatest variability or disagreement.

- **Expected Information Gain:**
  - focuses on the expected information gain about the model parameters θ given a new data point,
  - measured by the Kullback-Leibler (KL) divergence between the before/after posterior distribution of the model parameters before and after observing the new data point:

$$\mathbb{E}_{y|x,D}\left[D_{KL}\left[p(\theta|D,x,y)||p(\theta|D)\right]\right]$$

  Where
  - expectation over possible outcomes
  - KL divergence + the double line represents the KL divergence point between the two arguments
  - model parameters after new point
  - model parameters before new point

  This expression quantifies how much observing a new data point y at x is expected to change the distribution of the model parameters θ, hence indicating the data point's potential to reduce uncertainty.

  Before/after the new 'hallucinated' point - we 'hallucinate' a point to see how much the model would learn if we updated on this point.

- **KL Divergence:**
  - a measure of how one probability distribution diverges from a second, expected probability distribution.
  - In BALD, it's used to compare the model's parameter distribution before and after incorporating the new data point
  - providing a mathematical framework to capture the notion of "disagreement" or information gain.

- **Uncertainty Decomposition:** The approach can be conceptualized as decomposing the total uncertainty into two parts:

$$H(y|x,\mathcal{D}) - \mathbb{E}_{p\theta|\mathcal{D}}H(y|x,\theta,\mathcal{D})$$

  - **Uncertainty sampling** uncertainty in the predicted label (which can be reduced by adding more data).
  - **Irreducible uncertainty** inherent in the data distribution (which cannot be easily mitigated). This term penalizes places where we just can't learn the answer

  BALD aims to minimize the former while acknowledging the latter.

79 How to Ensure Learning Doesn't Suffer

Ensuring that learning doesn't suffer when examples are sampled from the population with varying probabilities involves adjusting the empirical risk minimization (ERM) process to account for the sampling scheme. This adjustment ensures the model training process is unbiased and reflective of the true population risk, despite the non-uniform sampling probabilities.

79.1 Non-Uniform Sampling & Poplation Risk

As always, our goal is to understand population risk, and minimize it.

Suppose examples are sampled from the population with probability p_i: dataset might not represent the true distribution of the population, especially if certain units are more likely to be sampled than others.

To approximate Population risk (which was the purpose of using random sampling in ERM), we need to adjust sampling strategy in ERM.

79.2 Adjusting for Sampling in ERM

Observed Expectation: The expectation of the loss $\mathbb{E}[\ell(y_i, f(x_i))s_i]$ considering the sampling indicator s_i, which equals 1 if the example is sampled and 0 otherwise, must be adjusted to reflect the true population risk.

Adjustment for Sampling Probability: Since $\mathbb{E}[s_i] = p_i$, the empirical risk must be recalculated to counteract the bias introduced by the varying sampling probabilities. This ensures that units with a higher likelihood of being sampled don't disproportionately influence the model's understanding of the population risk.

79.3 Solution: Reweighting the Sample

Reweighting Based on Sampling Probabilities: To correct for non-uniform sampling, each sampled unit's contribution to the empirical risk is weighted by the inverse of its sampling probability. This reweighting balances the dataset, making it as if each unit were sampled uniformly from the population.

Empirical Risk Estimates Calculation: The empirical risk is recalculated using the reweighted samples as follows:

$$\frac{1}{N}\sum_{i=1}^{N}\frac{1}{p_i}\ell(y_i, f(x_i))$$

above was ChatGPT; NB Slides slightly different:

$$\frac{1}{\sum_i \frac{1}{p_i}}\sum_{i=1}^{N}\frac{1}{p_i}\ell(y_i, f(x_i))$$

This formula ensures that each data point contributes to the loss in proportion to its true prevalence in the population, not its prevalence in the sample.

79.4 Implications

Fair Representation: By reweighting the samples according to their sampling probabilities, the training process more accurately reflects the true distribution of the population. This leads to a model that is better tuned to the nuances of the data it is intended to represent.

Reduced Bias & Improved Generalisation: reduces the bias in model training that results from non-uniform sampling, ensuring that the model's performance metrics are more indicative of its expected performance across the entire population/ generalizes better to unseen data, thereby enhancing their real-world applicability and reliability.

In summary, when samples are drawn from the population with varying probabilities, adjusting the empirical risk to account for these differences ensures that learning accurately reflects the entire population, minimizing risk and improving model performance.

> do does this mean we might employ non-uniform sampling strategies due to whatever reason (eg heteroskedastic errors where we want more data in one area of feature space), but that we can then reweight our ERM to get to approximate the population risk again?

80 Leverage Score (OLS)

In OLS: insights into the influence individual data points have on the model's estimates.

> Leverage score sampling lets you choose parts of the space to oversample (+ if we use Random Fourier Features, we can also substitute in kernels easily)
>
> We use it when getting labels is expensive and we need good predictions.

80.1 Definition

OLS regression: leverage score for a given data point quantifies the influence of that point on the fitted values:

$$h_{ii} = x_i(X^T X)^{-1} x_i^T$$

or

$$\frac{\delta \hat{y}_i}{\delta y_i} = X(X^T X)^{-1} X^T)_{ii}$$

This is the 'hat' matrix, where:

- x_i is the vector of predictor values for the ith observation;
- $x_i i$ are the diagonals, which are equivalent of the rate of change of this predictions wrt y. These diagons tells us how our model changes if we consider this particular point
- -> sample units proportionally to this

Leverage scores are always between 0 and 1. High leverage scores indicate points that have a substantial influence on the regression line's slope and position. These points can unduly affect the model's performance and interpretation.

These give us more bang for buck in prediction think of them as the outliers in a regression - they really affect the predictions.

By sampling units with a probability **proportional to their leverage scores**, you focus on including points in your sample that have the most substantial influence on the model's fit.

Cohen and Peng (2014)

When samples are selected based on leverage scores, predictions can achieve a desired accuracy level (within a factor of $1+\epsilon$) with significantly fewer samples.

Specifically, they found that $plogn/\epsilon^2$ samples are sufficient to ensure model predictions are within this accuracy factors.

NB this number does not include a population n term - you can get good population predictions even if you sample a relatively small sample. By focusing on the important points, it's almost as if we had access to the whole population.

This allows us to greatly speed up the convergence of the model
NOTES

81 Leverage Scores for Kernels & Scaling: Random Fourier Features

NB!

Running kernel regression on a full dataset computationally expensive ($O(n^3)$) -> instead run 1000s of linear regressions instead ((R^3)) (where R = features)

Kernel methods transform the input data into a higher-dimensional space where linear separation of the data is easier. However, this transformation often involves computing the Gram matrix, which is computationally expensive for large datasets.

The Challenge with the Gram Matrix

Computational Complexity: The Gram matrix for a dataset of n points is $n \times n$, requiring $O(n^2)$ space and, for some kernel computations, up to $O(n^3)$ time, making it impractical for large n.

Leverage Score Sampling in Kernels: To mitigate this, leverage score sampling can be applied to kernel methods. It allows for the selection of a representative subset of the data that captures the essential characteristics of the full dataset, reducing the size of the Gram matrix needed for computation.

81.1 Random Fourier Features (as approx of kernels

Approximation Technique: Random Fourier features provide a means to approximate kernel functions, like the Gaussian kernel, efficiently.

RFF uses a transformation that maps input data into a new feature space where linear algorithms can efficiently operate.

It leverages the fact that any shift-invariant kernel (including the RBF kernel) can be represented as an inner product in a transformed feature space.

Particularly valuable because it enables the use of kernel methods on large datasets by significantly reducing computational complexity.

Implementation: For a Gaussian kernel, random vectors W_r are drawn from a normal distribution $\mathcal{N}(0,1)$, and random biases B are drawn from a uniform distribution $\mathcal{U}(0,2\pi)$. The transformed features Z are then created for R features using the formula $Z = \sqrt{\frac{2}{R}}\cos(WX+B)$, where σ is the scale parameter of the Gaussian kernel.

Computational Efficiency: This transformation reduces the computational complexity from $O(n^3)$ to $O(R^3)$, where R is typically much smaller than n, significantly speeding up the computation without a substantial loss in accuracy.

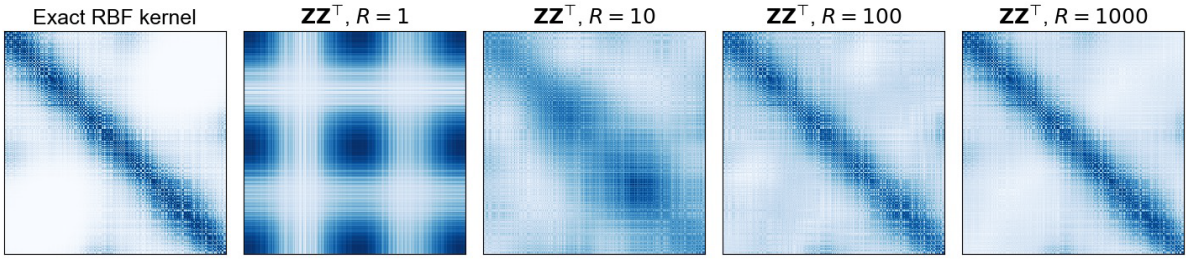


Figure 58: 1. is the RBF we're trying to approximate; then we see how increasing number of features

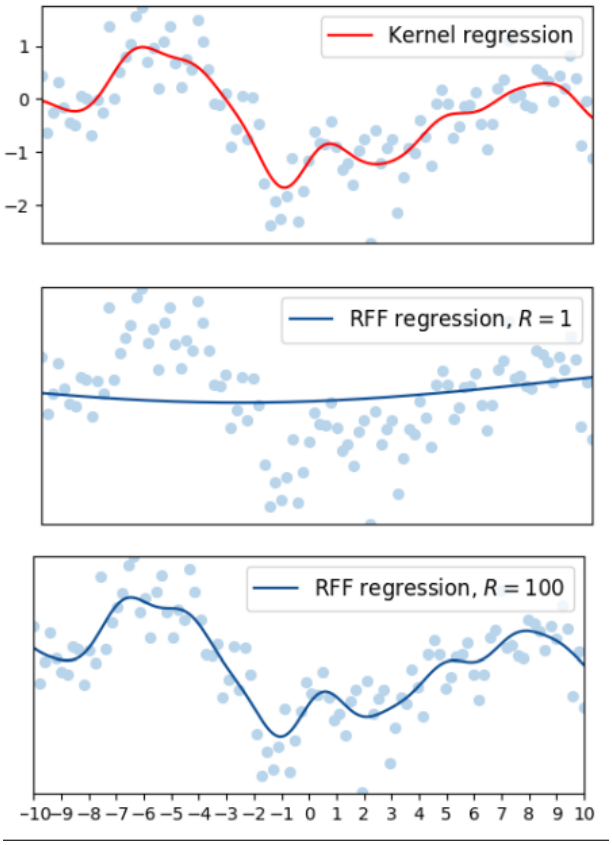


Figure 59: Enter Caption

82 Multi-Armed Bandits - decision making under uncertainty

Above has been about (1) Sampling to get a better models. Here we are applying Bandit algorithms for (2) sampling to improve rewards.

MAB = A restricted, simpler form of reinforcement learning (i.e. ML where an agent learns to make decisions by performing actions in an environment to achieve some goal.)

Use bandit approaches if the model is a nuisance and you just care about making a decision in the best possible way.

82.1 About

82.1.1 Basic Idea

Choose actions (from a set of possible actions) that maximize the cumulative reward over time. The decision-making process at each step is independent of previous steps (unlike in more complex reinforcement learning scenarios where the state of the environment changes based on the sequence of actions taken.) MAB: we don't repeatedly interact with units over and over

Single armed bandit = a slot machine (idea of getting robbed by lottery).

In a multi-armed bandit problem, imagine having several slot machines, each with different payout rates. The goal is to identify and play those that offer the best return rates. Initially, you won't know which machines are most profitable, necessitating experimentation to discover them.

This scenario represents a fundamental exercise in decision-making under uncertainty, laying the groundwork for reinforcement learning. Here, the act of choosing a slot machine to play parallels taking actions in the world to achieve a desired outcome, such as distributing different advertisements to see which performs best.

Essentially, the multi-armed bandit problem is a streamlined version of reinforcement learning focused on optimizing a series of actions for maximum reward, using the slot machine metaphor to illustrate the strategy of selecting the most beneficial actions.

82.1.2 Formally

Optimise sequence of actions so that you get the highest cumulative reward, based on the info available at each decision point:

$$max_{a_1,\cdots a_n} \sum_i r(a_i)$$

Where

- r = rewards
- a = actions

82.1.3 'Contextual Bandits'

$$\underset{max}{a_1,\cdots a_n} \sum_i r(a_i, x_i)$$

Decision-making process can also take into account additional information or features x, about the environment at each step.

'Conditional on X' - we have to solve the same thing, but just for units that look a particular way based on x

This context can be anything relevant to the decision-making process, such as user preferences, environmental conditions, or any other situational data.

82.2 Central Tradeoff

Exploration vs Exploitation
Exploration

- Experimenting with different actions to identify those that yield the highest rewards.
- Gathering more information and understanding the environment or the problem space better.

Exploitation

- Leveraging/exploit the information already acquired to make the best possible decisions.
- maximize the immediate reward based on current knowledge.

82.3 Epsilon-Greedy (Solution 1)

Simplest solution to tradeoff:

- For a proportion $(1-\epsilon)$ of the time, choose the action believed to be the best based on current information = exploitation.
- For the remaining ϵ potion of the time, select an action at random = exploration.

E.g. $\epsilon = 5\% \rightarrow 95\%$ of the time you play action you think is best; 5% of the time you choose completely random action.

BUT, the problem with epsilon greedy, is that every period we are paying the cost of possibly paying an arm that is very bad (5% of the time).

Instead, we want to do a lot of exploration early on in the game, then as we progressively get better at understanding which of the arms are good, we want to play those arms more and more... Upper Confidence Bound Algorithms....

82.4 Upper Confidence Bound Algorithm (Solution 2)

"Optimism under uncertainty" - more intelligent that Epsilon Greedy.

Main idea: maximize rewards over time, it's crucial not just to exploit what you currently believe to be the best option but also to explore seemingly suboptimal choices (i.e. to be **optimistic** that given your uncertainty in current estimates of what actions produce what value, these seemingly **suboptimal** choices might actually prove to be more rewarding upon further exploration!)

$$\bar{r_a} + \sqrt{\frac{2\log t}{n_{at}}}$$

where

- t is the total number of plays,
- n_a is the number of times action t has been played.

Sometimes written as

$$\bar{r_i} + \sqrt{\frac{\log n}{n_i}}$$

Mechanism: incorporating BOTH:

1. **(Exploitation)** current best estimate of value reward of an action: $\bar{r_i}$ (usually the highest value of the sum of its avg reward)
2. **(Exploration)** uncertainty or variability in that estimate: $\sqrt{\frac{\log n}{n_i}}$

NB it is this confidence/uncertainty term that makes it 'optimism under uncertainty'

82.4.1 Intuition

By including the uncertainty term, we are adopting a stance of "optimism under uncertainty" which encourages giving seemingly suboptimal actions more chances (based on the premise that they might yield better results than currently estimated)

UCB algorithms select actions based on a calculated upper bound of the potential reward, factoring in both the average reward and the uncertainty (or variance) of that reward.The decision rule incorporates Hoeffding's Inequality or an explicit confidence interval to balance between exploiting known good actions and exploring uncertain ones.

82.4.2 Sublinear Regret

The Upper Confidence Bound (UCB) algorithm effectively minimizes the cumulative regret over time, which is the opportunity loss from not consistently selecting the optimal action. The scaling of regret with the UCB algorithm is expressed as $O(\sqrt{n \log n})$, indicating a sublinear increase in regret over time.

- In the initial stages, there is a significant emphasis on exploration, with the uncertainty component being comparable in magnitude to the expected reward. This facilitates gathering information about the reward distribution of each action.
- As more trials are conducted (*i.e.*, as n increases), the algorithm gradually shifts its focus towards exploitation, where the uncertainty term diminishes relative to the expected reward. This shift underscores an increased confidence in the action value estimates, reducing the need for exploration.

The nature of the $\log n$ term is to increase sublinearly – it grows more slowly than n yet asymptotically continues to rise. This characteristic implies that while the exploration component decreases over time, it never completely ceases, ensuring that all actions are periodically revisited to account for potential changes in their reward distributions.

As the UCB algorithm progresses, the likelihood of selecting suboptimal actions (those with a lower expected reward) gradually decreases. This reduction is due to the algorithm's capability to distinguish more effectively between the actions' reward potentials as more data are accumulated.

The exploration cost in UCB, which pertains to the potential loss from exploring less promising strategies, diminishes relative to strategies like epsilon-greedy. In the epsilon-greedy strategy, a fixed proportion of decisions are made randomly, incurring a constant exploration cost. In contrast, UCB's exploration cost decreases as the algorithm becomes more discriminating in its exploration, focusing on actions that, despite appearing suboptimal, have not been adequately explored. This selective exploration results in a more efficient regret minimization strategy, where the likelihood of choosing less optimal actions – and thus the exploration cost – increases at a logarithmic rate, ensuring a more effective allocation of exploration efforts over time.

82.4.3 Practical Considerations

Success of this approach depends on a stable environment where the reward probabilities of actions do not change over time, allowing for a correct model to be built with accumulated data.

In cases where UCB algo assumptions don't hold, epsilon-greedy remains a viable strategy, particularly for continuous learning and adaptation, by ensuring some degree of exploration is **always** present, irrespective of confident / amount of learning done.

82.4.4 Incorporating other measures of uncertainty

NB: if we have a good measure of uncertainty (see next topic notes) of each arm, we could use that in the confidence / uncertainty term here. The idea would be that be that this red part would grow as sample size gets larger, so eventually it grows until the point that it is used.

82.5 Thompson Sampling (Solution 3)

An alternative approach to the Upper Confidence Bound (UCB) algorithms used in multi-armed bandit problems. Relying on a **probabilistic model** rather than a deterministic one.

Consider Thompson Sampling as the default option for managing exploration-exploitation trade off.

82.5.1 Thompson Sampling vs UCB Algorithms

UCB algos select actions deterministically, choosing the arm with the highest upper confidence bound on the expected reward. Balances exploration vs exploitation by mathematically calculating which arm is believed to offer the best reward, considering both the average reward and the uncertainty or variability in that estimate.

Thompson Sampling - selects actions based on the probability that each action is the best choice among all available options.

By sampling from a probabilistic model, Thompson Sampling directly builds in the exploration driven by uncertainty (that UCB has to mathematically model / approximate).

82.5.2 Mechanism

Probabilistic Action Selection: Actions are chosen according to the likelihood (p) that they are the best option:

$$p(a) = p(r_a = max_a r_i)$$

Where

- a = action selected based on prob that...
- ...its reward r_a is the max among...
- ... all r_i other actions

"The probability of choosing an arm = the probability that the reward for that arm is the best action"

Probability model allows for the calculation of the probability that any given action is the optimal one.

Through Bayesian inference or other probabilistic methods, it's possible to update the model with each action taken and its observed reward, thereby refining the probabilities over time.

Maintaining Good Error Estimated:

One of the strengths of Thompson Sampling is its inherent ability to maintain accurate estimates of the selection error.

Since actions are chosen based on probabilistic beliefs about their efficacy, and since the algorithm continuously updates these beliefs with each new piece of data, it naturally keeps track of the uncertainty associated with each action.

This ongoing adjustment ensures that the algorithm remains aware of its own performance and adjusts its action selection strategy accordingly.

82.5.3 Key Advantages

- Inherently balances exploration and exploitation by considering the uncertainty in its estimates of each action's reward.
- particularly effective in environments where the reward distributions are not static but may evolve over time.
- Computationally efficient and can be more intuitive to implement, especially when a probabilistic model of the environment already exists or can be easily constructed.

We don't derive this here, but it can be proved that Thompson Sampling also has sublinear regret; it's as good as UCB algos, but it has the nice properties of a probabilistic model.

83 Estimating Prevalence of a Trait - AIPW

Above we have covered (1) Sampling to get a better model; (2) sampling to improve rewards; (3) sampling to measure prevalence: AIPW.

83.1 The Problem

There is a trait we can measure, but it's expensive. We want to know how common it is. E.g. measuring what fraction of online comments are hate speech.

Random Sampling Solution - randomly sample units then take an average of sampled units.

- **However**, the **variance** of our estimate is contingent on the sample size, necessitating a power analysis to determine the necessary number of units for a reliable estimate.

Model Solution - an alternative method is to employ Empirical Risk Minimization (ERM) to train a model, using the model's predictions on the trait to compute an average.

- **Process**
  1. Random sampling (representative)
  2. Trait measurement
  3. Model training with ERM - create a model that accurately predicts the trait based on the input data.
  4. Avg model predictions
- This allows us to avoid having to collect the expensive y label, and instead focus on the easy X features.
- **However** while efficient, introduces significant **bias**, particularly in applications like estimating the probability of hate speech where model calibration must be precise—a challenging and often unrealistic assumption.

AIPW Solution - combine these methods, acknowledging that trait prevalence varies across categories and is often highly concentrated in certain topics. This variability suggests the utility of unequal sampling strategies, where more samples are drawn from units exhibiting characteristics associated with higher likelihoods of the trait of interest.

Prevalence itsn't evenly distributed

Non-uniform distribution of a particular trait across a population, as traits often cluster within specific subsets of units (i.e. among specific groups or under certain conditions). This poses challenges for accurately estimating the prevalence of the trait.

Solution: targeted sampling and adjusting sampling probabilities based on model predictions to achieve more representative and unbiased estimates:

- **Targeted Sampling** intentionally sample more frequently from units known to have characteristics associated with a higher likelihood of exhibiting the trait. (Ensures that the sample more accurately reflects the variance within the population).
- **Model-based Sampling Adjustment** - Scale up the strategy by using predictions from a previously trained model to guide the sampling process: adjust the sampling probability based on the model's output.
  - e.g. assigning prob p for units where model score $f(x) \leq 0.5$
  - doubling prob to $2p$ for units scoring $f(x) \geq 0.5$
- **Correction for oversampling** - to normalize the influence of the oversampled groups -> derive unbiased estimates from the skewed sample

Applied to hate speech example: this would give more units with hate speech in your sample, so we'd have more labels. We then correct for this oversampling to get to perfectly unbiased estimates.

83.2 Augmented Inverse Propensity Weights (AIPW)

1. **Augmented** - utilizes model estimates, often denoted as Q, which predict the likelihood or presence of the trait in each unit.
   - These predictions help in adjusting for the fact that some units were more likely to be sampled than others due to their characteristics or the model's targeting.
2. **Inverse Propensity Weights**
   - propensity score in this context is the probability of a unit being sampled, given its characteristics.
   - IPW corrects for unequal sampling proportions by weighting each unit inversely to its propensity score.
   - This means that units that were less likely to be sampled (but ended up in the sample) are given more weight in the analysis,
   - while those that were more likely to be sampled are given less weight.
   - This adjustment aims to mimic a random sample where every unit had an equal chance of being included.

Combining random sampling & model predictions solutions to the sampling problem (see above)

AIPW combines the strengths of predictive modeling (Q) with the robustness of IPW adjustments. By incorporating both the model's estimates and correcting for sampling bias, AIPW provides a more accurate and unbiased estimate of trait prevalence. This is particularly useful in complex sampling scenarios where direct measurement and simple corrections may not be sufficient to account for all sources of bias.

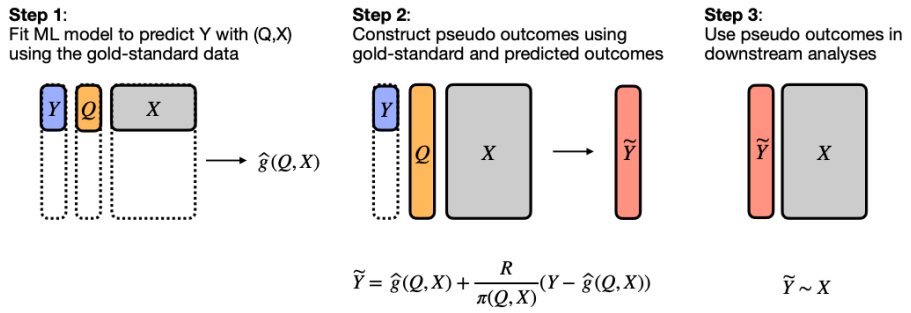


Figure 60: AIPW - process for estimating outcomes using machine learning models and propensity scores

Step 1: Model Fitting

- **Objective**: Fit a Machine Learning (ML) model to predict a target variable Y using predictors Q and X, where Q could be treatment indicators or other key features, and X represents covariates or control variables.
- **Process**: The ML model is trained on a "gold-standard" dataset, which means the data is considered to be of high quality and reliability.
- **Outcome**: The fitted model, represented as $\hat{g}(Q, X)$, is the model's estimation function that predicts Y from Q and X.

Step 2: Pseudo Outcomes Construction

- **Objective**: Generate pseudo outcomes $\hat{Y}$ that adjust for the sampling design or treatment assignment mechanism.
- **Process**:
  - **Original Outcomes**: Y from the gold-standard data are taken as the true outcomes.
  - **Predicted Outcomes**: The model's predictions $\hat{g}(Q, X)$ are the estimated outcomes based on Q and X.
  - **Propensity Scores**: $\pi(Q, X)$ is the propensity score, which is the probability of receiving the treatment or being sampled, given Q and X.
  - **Pseudo Outcome Formula**: The pseudo outcome $\hat{Y}$ is calculated using the formula:

$$\hat{Y} = \hat{g}(Q, X) + \frac{R}{\pi(Q, X)}(Y - \hat{g}(Q, X))$$

    Here, R is an indicator variable that is 1 if the unit is sampled or treated and 0 otherwise. This formula adjusts the predicted outcome $\hat{g}(Q, X)$ with a correction term that accounts for the discrepancy between the predicted and observed outcomes, weighted by the inverse of the propensity score.

Step 3: Utilization of Pseudo Outcomes

- **Objective**: Use the pseudo outcomes $\hat{Y}$ in subsequent analyses.
- **Process**: The pseudo outcomes are utilized as the dependent variable in further analysis models where X is the set of covariates or independent variables.
- **Implication**: This approach allows for the correction of potential biases in estimates due to non-random sampling or treatment assignment, leading to more accurate analyses that reflect the true effect of variables of interest.

In essence, this process demonstrates a method for creating a corrected outcome variable that accounts for potential biases in the data collection or experimental design, enabling more accurate statistical inferences in downstream analyses.

NB!

Crucially we are getting the **variance-reducing** properties of using the model on every single unit, AND we are getting the **unbiasedness** of pseudo-random sampling. We get the best of both worlds.

ML Lecture Notes: Wk 9 — Uncertainty

2 ways to formalise predictive uncertainty:

1. **Gaussian Process** - assume a model, get a Bayesian probabilistic model for your data
2. **Conformal Inference** - assume nothing about your model, get a particular measure of uncertainty about your predictions for future points.

84 A Primer on Gaussian Processes (GPs)

A Gaussian process is a collection of random variables, any finite number of which have a joint Gaussian distribution. This means that every finite set of points sampled from a GP can be described by a mean vector and a covariance matrix, creating a multivariate Gaussian distribution.

GPs are characterized by their mean function (often assumed to be zero in practice) and a covariance function or kernel, which describes the relationship or similarity between different points in the input space.

A GP is a probabilistic model that's used in scenarios where we want to make predictions about uncertain quantities. It's essentially a way to define a 'space' of possible functions that could fit our data, and then, using the observed data, we find the most likely function from this space.

Intuition:
Given some data points, a GP helps you predict where other points might lie, along with how certain it is about those predictions.

At its core, a Gaussian process generalizes the Gaussian probability distribution (bell curve). A single Gaussian distribution defines the probability of a single random variable. In contrast, a GP deals with functions, which you can think of as an infinite collection of random variables, one for each input point.

84.1 Components

A GP is fully specified by its mean function and a covariance function (a kernel).

1. **Mean Function** ($m(x)$)
   - Describes the average value of the function you're trying to predict at point x.
   - commonly set to zero since the GP can model the data well without specifying a mean function, thanks to the flexibility of the covariance function.
   - setting to 0 is just a simplification, saying that before seeing any data, we don't prefer one function's value over another.
2. **Covariance Function / Kernel** ($k(x, x')$ - embodies our assumptions about the function we want to learn.

- A function that defines the covariance between any two points x and x' in the input space.
- It encapsulates assumptions about the function such as smoothness, periodicity, and how quickly it can vary.
- It answers questions like: How does the output value at one point (say, temperature at one location) relate to the output at another point (temperature at a nearby location)? Do we expect smooth variations or abrupt changes? Are there repeating patterns?
- choice of the kernel function is critical and heavily influences the model's performance. It determines the shape and smoothness of the functions in our 'function space.
  - Squared Exponential Kernel: Assumes the function is very smooth. Close points will have similar values.
  - Periodic Kernel: Assumes the function repeats itself over time.
  - Linear Kernel: Assumes the function has a linear relationship.

  The GP uses the kernel to measure the similarity between points. Points that are 'close' according to the kernel will have similar outputs.

Inference: When you want to make a prediction at a new point, the GP looks at the points you've already observed and uses the kernel to weigh their influence on the new point. It combines these influences to give you **not just a single predicted value, but a probability distribution for that value**.

84.2 Properties

- **Non-parametric** - don't assume a fixed form of the function being modeled; instead define a distribution over possible functions that fit the data.
- **Incorporation of prior knowledge** - through choice of kernel; e.g. if we know the target function is smooth, we might choose a squared exponential kernel.
- **Uncertainty Quantification** - built-in measure of uncertainty: places where we don't have much data will have wider confidence intervals, reflecting that we're less sure about those areas. Crucial for applications like optimization, where understanding the confidence in predictions can guide decision-making.
- **Flexibility** - ability to work well with a small amount of data, thanks to the strong prior assumptions encoded in the kernel. Particularly good when you have a small amount of data but believe that the data has a rich structure.

84.3 The Challenge

Main downside is computational. For n data points, GP inference can require operations that scale with n^3, which becomes prohibitive for large n. This is because it involves inverting a large matrix that grows with the number of data points.

84.4 Summary

a Gaussian process is like a very smart, very flexible line that we fit through data. It doesn't just go straight, or curve in preset ways like traditional lines or curves. It can adopt an infinite number of shapes, guided by the properties of the data and the assumptions we encode in the kernel.

Plus, it gives us a shaded area that grows and shrinks, showing where we can trust its predictions and where we're guessing.

85 Conceptual (Bayesian) Shift: Probability Distributions of Functions

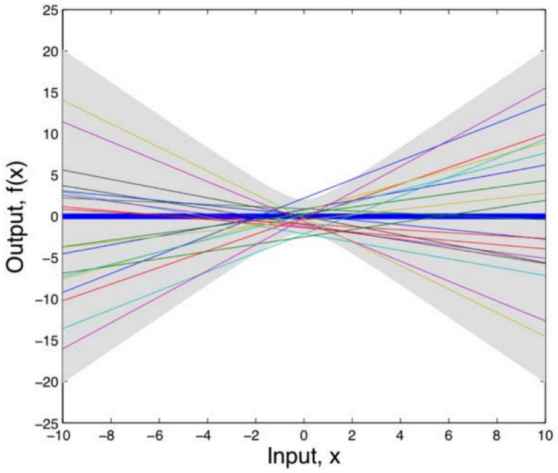


Figure 61: Enter Caption

85.1 The Idea

Probability distributions, commonly applied to random variables, can also be extended to functions.

Hypothesis class $\mathcal{H}$ - the set of all functions that your algorithm can possibly learn. Each element f of this set $\mathcal{H}$ is a candidate function for the 'true' function you aim to discover.

Prob Distribution Over Functions - Instead of picking one single function as our guess, we can take a probabilistic approach and assign a probability distribution over the entire hypothesis class.

- Every function f in $\mathcal{H}$ is assigned a prob $p(f)$ that reflects our belief in how likely it is to be the true function.

= Bayesian.

- Bayesian approach allows us to update our beliefs about the functions' probabilities as we observe more data.

85.2 Ridge Regression as applied example

RR can be thought of as putting a probability distribution over the coefficients β of linear functions.

$$p(f) = p(\beta)$$

In RR coefficients β close to zero are considered more likely than large β values. This reflects a prior belief that the true relationship is likely to be simple (with small coefficients) rather than complex (with large coefficients).

We're effectively placing a Gaussian (or normal) prior on the coefficients β: functions that correspond to these coefficients inherit this probability distribution, meaning we're indirectly placing

a distribution on functions.

we express our prior belief about the simplicity of the function by assuming a probability distribution over the function's parameters, which by extension, is a distribution over the space of possible functions.

85.3 Conceptual Shift

Thinking about probabilities not just in terms of random variables but in terms of entire functions.

In this framework, supervised learning now involves updating our beliefs about which functions are likely to be good explanations for the data, rather than searching for a single 'best' function. This perspective allows for a more nuanced understanding of model selection and the inherent uncertainties in the process of learning from data.

86 Gaussian Processes

NB!

when the goal is not just to make predictions, but also to understand the uncertainty associated with those predictions.

86.1 The Process

1. **Prior Distribution over Functions in $\mathcal{H}$** - this prior encapsulates our beliefs about the functions before observing any data.
2. **Condition on Observed Data** - We then update our beliefs (i.e., condition our prior) based on the data we've observed.
   - observed data: $y = f(x) + \epsilon$
   - assumed to be normally distributed, with mean 0, and variance σ^2: $\epsilon\ N(0, \sigma^2$
3. **Gaussian Process Model** - GPs model any collection of function values (labels) as a multivariate normal distribution, with means and covariances.
   - if you take any set of points from the function f, their corresponding outputs will be distributed according to a multivariate Gaussian.
   - How to construct a prob distribution: you need to derive a mean μ and covariance K based on inputs X: these two measures allow you to construct a prob distribution rather than just a point estimate.
4. **Conditional Distribution** - distribution of observed and predicted values is jointly Gaussian:

$$\begin{bmatrix} y \\ y_* \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mu_X \\ \mu_* \end{bmatrix}, \begin{bmatrix} K_{X,X} + \sigma^2 I & K_{X,*} \\ K_{*,X} & K_{*,*} \end{bmatrix} \right)$$

   where:

   - y = **observed training points**
   - y_* = **predictions (test points)**
   - $K_{X,X}$ = **covariance matrix of the training points**
   - $K_{X,*}$ = **covariance between the training and test points**

  - $K_{X,X}$ and $K_{X,*}$ tell how much the function values at the training points inform us about the function values at the test points.
- $K_{*,*}$ = **covariance among the test points themselves**
  - how much we expect the function values at one test point to vary with the function values at another test point, before seeing the data.
- $\sigma^2 I$ **noise associated with the training data observations** - adds variance to the observed data to account for noise.

5. **Bayesian Optimization** - GPs can be applied to continuous optimization problems via Bayesian Optimization: a GP is used to model the function we're optimizing, and an acquisition function is used to make decisions about where to sample next, considering both the GP's predictions and the uncertainty around those predictions.

NB!

= any collection of labels are distributed multi-variate normal, With means and covariances

Conditional Normality Assumption:

The fundamental assumption in Gaussian Processes (GPs) is that all points are conditionally normal; this means that for any set of inputs, the corresponding outputs will follow a multivariate normal distribution. We fit the GP model by conditioning on observed data, effectively updating our beliefs about the function we're modeling in light of new evidence.

86.2 Applications for Sampling & Decision Making

In contrast to bandit algorithms, which are designed for making discrete choices, Bayesian optimization is an approach for making continuous choices. This optimization often involves GPs due to their ability to model the uncertainty in function values effectively.

The GP learns from the data, and we utilize an acquisition function to determine the next point to sample. This acquisition function balances exploitation, where we sample points in regions with high predicted values (suggesting high reward or low cost), against exploration, where we sample points in regions with high uncertainty (where there's more to learn about the function's behavior).

Bayesian optimization with GPs is a sequential, iterative process. At each iteration, the GP model incorporates new observations, and the acquisition function determines the most informative next sample point based on the current model. This strategy is particularly advantageous for optimizing functions that are costly to evaluate because it strives to achieve the best possible results with as few evaluations as possible, thereby conserving resources and time.

86.3 Fit a model by condition on data: output statistics

Making predictions using a Gaussian Process model after the model has been conditioned on observed data, (training set $\mathcal{D}$).

When we condition on the data, we update our belief about the distribution of functions based on the evidence provided $\mathcal{D}$

3.2.3 Marginals and conditionals of an MVN *

Suppose $\boldsymbol{y} = (\boldsymbol{y}_1, \boldsymbol{y}_2)$ is jointly Gaussian with parameters

$$\boldsymbol{\mu} = \begin{pmatrix} \boldsymbol{\mu}_1 \\ \boldsymbol{\mu}_2 \end{pmatrix}, \quad \boldsymbol{\Sigma} = \begin{pmatrix} \boldsymbol{\Sigma}_{11} & \boldsymbol{\Sigma}_{12} \\ \boldsymbol{\Sigma}_{21} & \boldsymbol{\Sigma}_{22} \end{pmatrix}, \quad \boldsymbol{\Lambda} = \boldsymbol{\Sigma}^{-1} = \begin{pmatrix} \boldsymbol{\Lambda}_{11} & \boldsymbol{\Lambda}_{12} \\ \boldsymbol{\Lambda}_{21} & \boldsymbol{\Lambda}_{22} \end{pmatrix}$$

where $\boldsymbol{\Lambda}$ is the **precision matrix**. Then the marginals are given by

$$\begin{aligned} p(\boldsymbol{y}_1) &= \mathcal{N}(\boldsymbol{y}_1 | \boldsymbol{\mu}_1, \boldsymbol{\Sigma}_{11}) \\ p(\boldsymbol{y}_2) &= \mathcal{N}(\boldsymbol{y}_2 | \boldsymbol{\mu}_2, \boldsymbol{\Sigma}_{22}) \end{aligned}$$

and the posterior conditional is given by

$$\begin{aligned} p(\boldsymbol{y}_1 | \boldsymbol{y}_2) &= \mathcal{N}(\boldsymbol{y}_1 | \boldsymbol{\mu}_{1|2}, \boldsymbol{\Sigma}_{1|2}) \\ \boldsymbol{\mu}_{1|2} &= \boldsymbol{\mu}_1 + \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}(\boldsymbol{y}_2 - \boldsymbol{\mu}_2) \\ &= \boldsymbol{\mu}_1 - \boldsymbol{\Lambda}_{11}^{-1}\boldsymbol{\Lambda}_{12}(\boldsymbol{y}_2 - \boldsymbol{\mu}_2) \\ &= \boldsymbol{\Sigma}_{1|2}\left(\boldsymbol{\Lambda}_{11}\boldsymbol{\mu}_1 - \boldsymbol{\Lambda}_{12}(\boldsymbol{y}_2 - \boldsymbol{\mu}_2)\right) \\ \boldsymbol{\Sigma}_{1|2} &= \boldsymbol{\Sigma}_{11} - \boldsymbol{\Sigma}_{12}\boldsymbol{\Sigma}_{22}^{-1}\boldsymbol{\Sigma}_{21} = \boldsymbol{\Lambda}_{11}^{-1} \end{aligned}$$

Figure 62: Enter Caption

86.3.1 Predictive Distribution

for outputs y^* at new input points X^*, given observed data $\mathcal{D}$): is a normal distribution, with mean μ^* and variance Σ^*:

$$p(f_*|X_*, \mathcal{D}\, \mathcal{N}(\mu_*, \Sigma_*)$$

To construct this predictive distribution, we need:

1. Mean μ^*
2. Variance Σ_*

86.3.2 Mean μ^*:

$$\mu_* = m(X_*) + K_{X,*}^T K_{X,X}(\sigma^2 I)^{-1}(y_* - m(X_*))$$

Where, you take the

- $m(X^*)$= mean function at new input points (X^*). It represents our **prior mean** for these points. It provides a baseline expectation for the function's output before considering the observed data.
- Then adjust based on:
  - $K_{X,*}$: **covariance matrix between the training inputs X and the test inputs X_***
    - It measures how much each of the new input points X_* is expected to covary with the training inputs.
    - Essentially, it captures the influence of the observed data on the predictions at new points based on the similarity (as defined by the kernel) between the training and new input points.
    - reflects how changes in the training inputs inform the outputs at the test inputs.

- $K_{X,X}^{-1}(\sigma^2 I)^{-1}$: **inverse of the covariance matrix of the training inputs X, scaled by noise σ^2**, accounting for the uncertainty due to noise in the observed outputs.
  - inverse covariance matrix of the training inputs X (often regularized by adding $\sigma^2 I$ to ensure it's invertible) reflects how the training points relate to each other.
  - serves as a normalization factor that weights the influence of each training point based on its relationship to the rest of the training data.
- $(y_* - m(X))$: **difference between the observed outputs y and the prior mean function evaluated at the training points $m(X)$.**
  - represents how the actual observations deviate from our prior beliefs.
- $\sigma^2 I$ = **noise in the observations**
  - is added to the diagonal of $K_{X,X}$ before inversion.
  - It accounts for the uncertainty in the observed outputs y due to noise, ensuring that the model does not overfit the training data by assuming the observed outputs are perfectly accurate.

Interpretation of the mean μ_*
represents our best guess for the function's output at the new input points X_*, adjusted for the information gained from the observed data $\mathcal{D}$.

This adjustment adjusts the prior mean $m(X_*)$ based on

1. how similar (according to the kernel) the new inputs X_* are to the training inputs X
2. how the observed outputs y deviate from what the mean function predicts for the training data.

This results in a prediction that is informed by both the prior knowledge encoded in the mean function and the observed data, weighted by the similarities between input points.

86.3.3 Variance Σ_*:

reflects our uncertainty re: predictions

$$\Sigma_* = K_{*,*} - K_{X,*}^T K_{X,X}(\sigma^2 I)^{-1} K_{X,*}$$

Where, you take the

- $K_{*,*}$ = **covariance matrix of the test points** - how much we expect the function values at one test point to vary with the function values at another test point, before seeing the data.
- $K_{X,*}^T K_{X,X} \cdot$ = a term based on how the test and training points relate to each other / how much the function values at the training points inform us about the function values at the test points.
- $\sigma^2 I$ = **Noise Term** - This represents the noise associated with the training data observations. It's added to the diagonal of the covariance matrix of the training points, which ensures that we're accounting for the uncertainty in our measurements of the function's output.

Intuitive Step-by-step:

1. **Start With Test Covariance:** We begin with our initial uncertainty about the function values at the test points, which is the covariance matrix of the function values at the test points itself, denoted as $K_{*,*}$. This represents our best guess about how the function values might vary together before we consider the data.
2. **Adjust by Training-Test Relationship:** We then adjust this by how much the test points are expected to vary with the training points, represented by the covariance matrix between the training points and the test points, denoted as $K_{X,*}$. Specifically, we're looking at the covariance between our test predictions and the actual observed training data.
3. **Apply Noise Adjustment:** The term $(K + \sigma^2 I)^{-1}$ adjusts for the uncertainty in our training data due to noise. This adjustment is essential because it prevents us from being too confident in our predictions when our observations are noisy.
4. **Subtract to Get Final Variance:** The subtraction essentially narrows down our initial uncertainty by removing the part that can be explained by the relationship with the training data. The result, Σ_*, is a matrix that gives us variances and covariances of the predictions.

Interpretation of Σ_* The diagonal elements of Σ_* give us the variances at each test point. A higher variance indicates greater uncertainty in the prediction at that point.

Off-diagonal elements represent covariances between the predictions at different test points. These help understand how the uncertainty about one prediction relates to the uncertainty about another.

Σ_* captures what we don't know about our test predictions after taking into account what we do know from the training data and the model's structure (as specified by the covariance function). This computed variance Σ_* is what allows GPs not just to make predictions, but to provide a confidence measure for those predictions, which is invaluable in decision-making processes where uncertainty needs to be considered.

NB!

GPs makes predictions that are:

1. informed by the observed data
2. carry a measure of uncertainty.

86.4 GP as probabilistic reinterpretation of KRR

Note to self: these calculations are VERY similar to how we reconcptualised Kernel Ridge Regression as being about how similar units are to each other, with similarity measures defined by a kernel.

The GP goes a step further by taking the KRR idea of modeling similarities (via the kernel) between points, then uses it to define a distribution over functions. Instead of merely providing a single point estimate for the predictions (as in KRR), a GP provides a joint probability distribution for all possible predictions. This distribution is defined by means (which can be

seen as the point predictions) and variances/covariances (which encode the uncertainty and the relationships between different predictions).

The construction of a GP involves computing a mean and a covariance matrix from the data, informed by the chosen kernel.

A GP can be conceptualized as a probabilistic reinterpretation of kernelized ridge regression, where the emphasis is on the entire distribution of possible outcomes rather than just the expected value, allowing for the modeling of uncertainty and the correlation of predictions.

86.5 When we assume mean function to be 0 -> identical to KRR

When the mean function in a GP model is set to zero, and the model is conditioned on observed data points, the resulting posterior mean function of the GP can be seen as equivalent to the solution provided by KRR.

86.5.1 Gaussian Processes and Kernel Ridge Regression:

Gaussian Process: In a GP, when we assume a zero mean function, the focus shifts entirely to the covariance (kernel) function to model the data. When we The GP predicts new outputs f_* based on observed data ($\mathcal{D}$) and inputs X_* as a multivariate normal distribution with mean μ_* and covariance Σ_*.

Kernel Ridge Regression: KRR is a form of ridge regression that uses a kernel to map input data into a higher-dimensional space, where linear regression is then applied. The regularization in KRR adds a penalty for large weights in the solution, similar to how the noise term $\sigma^2 I$ regularizes the GP's covariance matrix.

86.5.2 Connecting GPs and KRR:

Posterior Mean Function: When conditioning a GP (with a zero mean function) on observed data, the posterior mean for the outputs f_* at new inputs X_* is given by:

$$\mu_* = K_{X,*}^T (K_{X,X} + \sigma^2 I)^{-1} f$$

This formula directly parallels the solution in KRR, where:

- $K_{X,*}$ is the covariance between the training inputs and new inputs.
- $K_{X,X}$ is the covariance among the training inputs.
- $\sigma^2 I$ represents the noise term, analogous to the regularization in KRR.

Intuition: The essence here is that both GP (with zero mean) and KRR use the kernel to measure similarities between points and regularize these relationships to predict new values. The GP's posterior mean incorporates the observed data (similarly to how KRR does) by weighting the influence of each training point based on the kernel-defined similarity and regularization term. Essentially, data points that are 'similar' (according to the kernel) to the test points have a greater influence on the prediction.

HERE IS WHERE GPs EXTEND KRR - BY NOT ONLY COMPUTING THE MEANS OF THE DISTRIBUTIONS, BUT ALSO ALSO COMPUTING THE COVARIANCE Σ_* WHICH QUANTIFIES UNCERTAINTY - THIS ALLOWS GPs TO BUILD A FULL POSTERIOR PREDICTIVE DISTRIBUTION (BEYOND KRR'S POINT ESTIMATE)

- In KRR, you get point estimates of the function at new inputs. These estimates are based on a deterministic function learned from the data.
- GPs, on the other hand, provide a full probabilistic model. For any set of inputs, a GP gives you a mean prediction and a covariance matrix:
  - the diagonal elements of Σ_* give the variances at each predicted point, representing the model's confidence in its own predictions.
  - The off-diagonal elements represent the covariances between predictions at different points, indicating how much the uncertainty in one prediction is correlated with the uncertainty in another.
- his additional information from Σ_* is particularly valuable when you need to make decisions under uncertainty, as it provides insight into the reliability of the predictions and how they might vary together. For instance, in Bayesian optimization, the uncertainty (quantified by Σ_*) is used to balance exploration and exploitation when choosing the next points to evaluate.

Posterior Predictive Distribution: To obtain the full posterior predictive distribution in a GP, including the uncertainty of the predictions, we also compute Σ_*, which reflects the variance and covariance of the predictions. If one desires to include the observational noise in the predictive distribution, $\sigma^2 I$ is added to the diagonal of Σ_*, mirroring how in KRR, predictions are made within the context of inherent data noise.

In summary, assuming a zero mean function in a GP and conditioning on data leads to a posterior mean that is conceptually and mathematically similar to the solution obtained through KRR. This connection beautifully illustrates how GPs extend the principles of kernel methods and regularization from KRR to a probabilistic framework, enabling not just predictions but also quantification of uncertainty around those predictions.

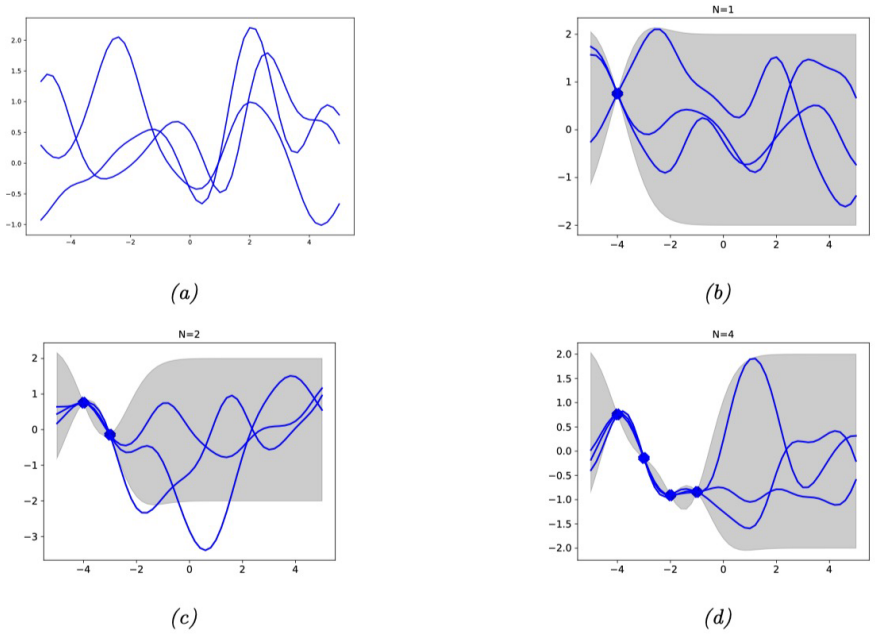


Figure 63: progression of GP as more data points observed: all of the function values that are not consistent with that point at X_* get thrown out / down weighted

(a) No Data Points: The first plot (top left) shows several functions **sampled from a GP prior**. Without any observed data, the functions are quite varied, reflecting the high level of **uncertainty** about the true function form. The GP prior gives us a wide distribution of possible functions that could fit future observed data.

(b) One Data Point ($N = 1$): The second plot (top right) introduces a single observed data point, as indicated by the 'x' marker. The shaded area represents the confidence interval, or the uncertainty around the predicted function values. Notice how the uncertainty (the width of the shaded area) is smallest near the observed data point and grows as we move away from it. The blue lines are **functions that are drawn from the posterior distribution of functions given the observed data**. They **all go through the observed data point** due to the high certainty about the function value at that point.

(c) Two Data Points ($N = 2$): The third plot (bottom left) now includes two observed data points. The confidence interval is narrower around the observed points where the model is more certain, and wider in areas far from these points where the model has more uncertainty. The sample functions from the GP reflect this by varying less near the observed points and more in areas with no data.

(d) Four Data Points ($N = 4$): The final plot (bottom right) shows the situation after observing four data points. The confidence interval has tightened significantly around these points, indicating a much stronger belief in the function values there. The sample functions from the GP pass through all the observed points and have less variance around them, illustrating that the GP has learned the trend from the data and has less uncertainty where the data is dense.

This illustrates the GP's capability to both interpolate and extrapolate based on the data, providing a probabilistic framework for regression that naturally incorporates uncertainty.

86.6 GPs have Good Variance Properties

Increase in Variance Away from Data: As a test point gets further from the observed (training) data points, the GP model's uncertainty about the prediction at that test point increases. This is reflected in an increase in variance. As the prediction moves further away from where we have data, the model has to rely more on extrapolation, which is inherently less certain.

Dependence on Kernel: The rate at which the variance increases as we move away from the data depends on the choice of the kernel function in the GP. Some kernels will cause the variance to increase rapidly, while others might result in a more gradual increase.

Contrast w/ other methods:

- linear regression or polynomial regression, provide point estimates without a measure of uncertainty, so there's no concept of increased variance with distance from the training data.
- Even within the subset of methods that do provide some measure of uncertainty, such as some Bayesian methods, they may not always show an increase in variance with distance from the data points

Variance property of GPs is extremely valuable in

1. quantifying the reliability of predictions (eg for policy / decision making)

2. guiding decisions in areas like active learning and Bayesian optimization, where you need to weigh the benefit of exploring unknown regions against exploiting known areas.

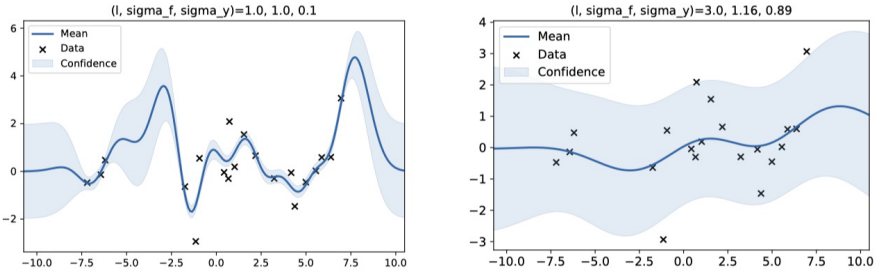


Figure 64: crosses = training data; every point on x axis = a potential test point. Can see certainty follows training data

- solid blue line is the GP's mean prediction of the function that generated the data.
- shaded area represents the confidence interval / indication of the uncertainty of the GP's predictions.
- RHS: confidence intervals are consistently wider across the entire range of inputs - due to both the larger length-scale, indicating that observations influence the prediction over a larger range, and the higher noise variance, which indicates less confidence in the precision of the observations.

86.7 Posterior of the function vs Posterior Predictive

distinction between two types of uncertainties that can be modeled with GPs

1. Uncertainty of the function itself (posterior of the function)
2. Uncertainty in future observations (posterior predictive)

86.7.1 Posterior of the Function

The uncertainty in the function values that the GP has learned based on the observed data.

It's about the shapes or forms the true function could take within the bounds of the data and prior assumptions (encoded in the kernel).

Credible interval: a range of values for the function at a given point that, given the prior and observed data, are believed to contain the true function value with a certain probability.

- A 90% credible interval for the conditional mean would mean that there is a 90% probability that the true function value lies within that interval.

86.7.2 Posterior Predictive

reflects the uncertainty about future observations (labels) that would be seen if we were to sample new data from the function at new input points.
Prediction Interval: a range of values that are believed to contain the true observed labels with a certain probability, taking into account the noise in the observations.

- If we say there's a 90% prediction interval, we mean that there's a 90% probability that a future observation will fall within this interval.

86.8 Implication for Kernels

Adding the noise variance to the kernel is a methodological step for shifting from a focus on the posterior of the function -> posterior predictive.
When we're interested in the posterior predictive distribution, which includes the variability due to noise, we add the noise variance term (σ_y^2) to the diagonal of the kernel covariance matrix K.

This adjustment accounts for the additional uncertainty that comes from noisy observations when making predictions about future observed labels.

86.9 Web Demo

http://www.infinitecuriosity.org/vizgp/

86.10 Takeaways

GPs

1. are a prob distribution over functions
2. create new distributions by conditioning on data
3. have good variance properties

Flexible models that provide extremely well behaved estimates of uncertainty • Scaling them to large datasets is challenging, and an active area of ML research • GPs are commonly used as part of Bayesian Optimization: • Learn a GP as a response function • Choose an acquisition function that manages exploration/exploitation • Sample new units at which to measure outcomes • It's one particular form of the larger RL paradigm

87 Conformal Inference

87.1 A primer on Comformal Inference

Create rigorous prediction intervals (for future observations) that are valid under a certain confidence level, regardless of the underlying distribution of the data.

It is based on the idea of nonconformity measures that assess how well new observations conform to past observations.

1. **Nonconformity Measure** - define a nonconformity measure, which is a function that assigns a score to each example based on how different or "nonconforming" it is relative to a set of examples.
    - in a regression setting, a nonconformity score might be the absolute residual of each point — how far away the actual y value is from the predicted y value.
2. **Train** - train your model on a proper training set as usual.
3. **Calibration** - apply your model to a separate calibration set and calculate nonconformity scores for each point in this set. These scores indicate how well the model's predictions match the actual values.
4. **Prediction Interval** - When you have a new point you'd like to predict, you use your model to predict it, and then you use your nonconformity measure and the scores from the calibration set to determine how "strange" this new prediction is.

- You calculate a prediction interval for the new point that, if the method is correctly calibrated, will contain the true value with a specified probability (such as 95

5. **Validity** - Under mild conditions, conformal prediction intervals are provably valid, meaning that if you say you're 95% confident the true value lies within the interval, it will indeed lie within the interval 95% of the time in the long run.

Distribution free! Does not rely under assumptions about the underlying distribution of the data.

A way to generate prediction intervals that are theoretically guaranteed to contain the true prediction a certain percentage of the time (known as the coverage probability). Adds interpretability / reliability.

87.2 Process

Set Target Coverage Probability

- choose a significance level α which is related to the desired coverage probability (eg $\alpha = 0.05$ for 95% coverage).
- this gives prediction interval (l, u) we will be calculating, such that $P(l \leq y \leq u) \leq 1 - \alpha$
  - a 'prediction' interval for future prediction
  - for a given level of confidence

α corresponds to the Type I error rate (false positive)

Calculate Prediction Interval based on Target Coverage Probability

1. **Heuristic Notion of Uncertainty** - identifying a measure that reflects the confidence in your predictions.
2. **Score Function** $s(x, y)$ - heuristic from step 1 is formalized into a score function.
   - score function assigns a numerical value indicating the error or nonconformity of the predicted value $f(x)$ relative to the true value y
   - $s(x, y) = |y - \hat{f}(x)|$ is a common choice (where larger score = worse)
3. **Quantile** T_q - calculate the $(1 - \frac{\alpha}{2}$ quantile of these scores based on your calibration of validation data
   - This quantile represents the threshold that a certain proportion (e.g., 95% for a 5% significance level) of the scores fall below.
   - compute $\hat{q}$ as the $\frac{\lceil (n+1)(1-\alpha) \rceil}{n}$ quantile of yur data
4. **Prediction Set** $\mathcal{C}(X_{test})$ - use quantile to form a prediction set:

$$\mathcal{C}(X_{test} = \{y : s(X_{test}, y) \leq \hat{q}\}$$

   Any valye y that when paired by X_{test} results in a score $s(X_{test}, y)$ less than or equal to T_q is included in the set
   This set is computed using out-of-sample data to ensure it accurately reflects the model's ability to generalize.

87.3 Example

We have DGP $y = \beta X + \epsilon$, where ϵ is very non-normal.

We can 'conformalise' our predictions, $\hat{y} = \hat{\beta} X$ to get a brediction interval that is still valid (ie has 95% coverage) no matter what - it might not be a good interval (i.e. not tight at all), but it is valid.

The tightness of this interval will depend on the appropriateness of the score function and the variability of the data.

87.3.1 Steps for Conformal Inference

1. **Fit Your Model:** First, fit the linear model to your data, $y = \beta X$, even though you know the error distribution is non-normal. **Initial (Incorrect) Assumption:** Temporarily assume that $\epsilon \sim \mathcal{N}(0, \sigma^2)$), acknowledging that this assumption is incorrect but necessary for the next step.

2. **Calculate Nonconformity Scores:** Define your nonconformity score $s(x, y)$ as the standardized absolute residuals. This score represents how unusual or 'nonconforming' each observation is relative to the model prediction. It's calculated by dividing the absolute residual by an estimate of the standard deviation of the residuals, σ:

$$s(x, y) = \frac{|y - \hat{y}|}{\hat{\sigma}}$$

$$s(x, y) = \frac{|y - \beta X|}{\hat{\sigma}}$$

   where

   - $\hat{y}$ is the prediction from your linear model
   - $\hat{\sigma}$ is the estimated standard deviation of the residuals

3. **Adjust for Heteroskedasticity:** If you suspect that the residuals are heteroskedastic (the variance of the residuals changes with X), then you need to model $\sigma(\hat{X})$.

   - You can train a separate model that predicts the standard deviation of the residuals for each X, thus allowing for different variances at different points in the predictor space.

4. **Determine the Prediction Interval:** Using the nonconformity scores from your calibration or validation set, calculate the appropriate quantile for your desired coverage probability (e.g., the 95th percentile for a 95% prediction interval).

   - for each new X value where you predict y, need to compute a range around the predicted value $\hat{y} = \beta X$ that corresponds to the quantil of the nonconformity score.

5. **Form Prediction Interval:** conformal prediction interval for a new observation at X_{new} would be:

$$\mathcal{C}(X_{new}) = \hat{y} \pm \text{quantile value}$$

The key property of conformal prediction intervals is their validity: if you create a 90% prediction interval using this method, then, in the long run, 90% of the true y values will fall within the intervals you produce, regardless of the underlying distribution of ϵ

87.4 Marginal vs Conditional Coverage

Our guarnatee from conformal inference is marginal. We might want it to be conditional.

Marginal Coverage

The coverage guarantee is **averaged over all points** in the test set. For example:

X	$f(X)$	(l, u)	$P(y \in (l, u))$
1	0.1	$(0, 0.2)$	99%
2	0.2	$(0.1, 0.3)$	90%
3	0.3	$(0.2, 0.4)$	81%
		Overall:	90%

Even though for some X the probability of y falling in the interval is much higher than the target $1 - \alpha$, overall, when averaged across all points, the coverage meets the target rate.

Conditional Coverage

This refers to the desired property where the **coverage guarantee holds within subgroups** defined by X. For each value of X, the interval would correctly contain the true y 90% of the time.

X	$f(X)$	(l, u)	$P(y \in (l, u))$
1	0.1	$(0.02, 0.18)$	90%
2	0.2	$(0.1, 0.3)$	90%
3	0.2	$(0.18, 0.42)$	90%
		Overall:	90%

Here, each value of X has a tailored interval that maintains the coverage rate specifically for that value.

However, achieving conditional coverage is much more complex because it requires intervals that adapt to the distribution of y at each X value, accounting for all potential variations within the data.

The challenge with conditional coverage is how to define and measure "similarity" or "closeness" in X values and how to adjust intervals accordingly to maintain the desired coverage rate within each subgroup.

There isn't a straightforward method to achieve exact conditional coverage in all cases. Conformal inference typically provides marginal coverage guarantees, and while there are advanced methods attempting to approach conditional coverage, it remains an active area of research.

88 Uncertainty Takeaways

Motivations/purposes - with good measures of uncertainty you can:

- **Improved Decision Making**: Understanding the uncertainty can improve decision-making in uncertain environments. For instance, in Thompson sampling, the variance of predictions is used to balance exploration and exploitation in multi-armed bandit problems.

- **Outlier Detection**: A good measure of uncertainty can help in identifying outliers by flagging data points that have high uncertainty in their predictions, which could indicate that they deviate significantly from the pattern captured by the model.
- **Active Learning**: In active learning, models can use uncertainty estimates to determine which new data points would be most informative to learn from if they were labeled. This is especially useful in scenarios where labeling data is expensive or time-consuming.

Tools / How? - 2 tools!

- **Gaussian Processes (GPs)**: excellent, but computational expensiv e
  - **Versatility**: GPs can provide various types of uncertainty measures, not just for the predictions but also for the function that generates the data. This includes both the posterior of the function and the posterior predictive distribution.
  - **Challenges**: While GPs are powerful, they struggle with scaling to very large datasets due to their computational complexity, which typically involves matrix operations that are cubic in the number of data points.
- **Conformal Inference**: extremely scalable, but only gives us preciction intervals, and only gives us marginal guarantees not the conditional intervals
  - **Scalability**: Conformal inference is highly scalable and can be applied to very large datasets.
  - **Limitations**: It primarily provides prediction intervals with marginal coverage guarantees. While these intervals are valid under the model's assumptions, conformal inference does not typically give conditional coverage, which would ensure that the coverage guarantee holds within specific subgroups of the data.
  - **Compatibility**: A major advantage of conformal inference is that it can be applied on top of any predictive model, from simple linear regressions to complex machine learning models.

GPs are powerful for smaller datasets where the modeling of detailed uncertainties is crucial, while **conformal inference** shines in providing scalable and model-agnostic prediction intervals.

ML Lecture Notes: Wk 10 — Neural Nets

89 Perceptrons

89.1 The Algorithm

A Linear classifer. Initially proposed as a model of biological neuron function.

Formally: binary classifier: maps inputs x (feature vector) to output value $f(x;\theta$ based on linear prediction functions combining weights β w/ feature vector:

A non-probabilistic version of logistic regression

$$f(x;\theta) = \mathbb{I}(X\beta \geq 0)$$

Where

- $\mathbb{I}$ is an indicator function that outputs 1 if the argument is true (i.e. that the linear combo of $X\beta$ is non-negative) and 0 otherwise.

This heavy-side step function is what makes the peceptron a non-probabilistic classifier (unlike logistic regr, which outputs probabilities).

Learning Algo - updates weights to reduce classification errors.

Update rule (for a single dimension) at each step t given by:

$$\beta_i(t+1) = \beta_i(t) + r(y_i - \hat{f}_j(t))(\mathbf{x}_{i,j}$$

Where

- in the case of an error...
- ...updates each coefficient
- ...in the direction of the covariate
- r = learning rate (hyperparam)

Process:

1. Calculate output based on given β: $\hat{f}_j(t) = f(X_j\beta(t))$
2. Update: $\beta_i(t+1) = \beta_i(t) + r(y_i - \hat{f}_j(t))(\mathbf{x}_{i,j}$
3. Calculate output based on updated β
4. ...

This update is performed iteratively over the training set until the algorithm converges (i.e., no further errors are made on the training set) or a maximum number of iterations is reached.

The Perceptron's weight update is reminiscent of the gradient descent used in logistic regression, where the gradient of the negative log-likelihood (NLL) with respect to β

$$\nabla_\beta NLL(\beta) = \frac{1}{n}\sum_{j=1}^{n}(y_j - \hat{f}_j(x+j)) \cdot x_j$$

The Peceptron is updating like in logistic regression - but LogRegr updates weights based on the gradient of a probabilistic loss, the Peceptron update rate:

- for a single observation/error at a time (whereas LogRegr: for the whole dataset, hence the sum)
- Perceptron updates weights directly based on misclassifications, making it suitable for datasets that are linearly separable (whereas LogReg uses probabilistic loss)

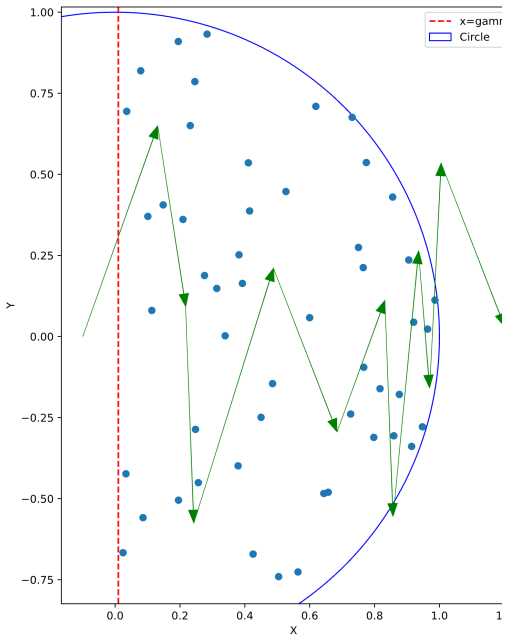


Figure 65: I don't fully understand this, but the point is that the algo bounces around a log, but generally moves in the right direction; the decision boundary is the red dashed line - if the decision vector was pointing all the way to the right, then the red line is the decision boundary?????

This does converge (in datasets that are linearly separable), but there are issues:

89.2 Problems!

89.2.1 1. Fundamental Linearity of Perceptron

If not linearly separable - Peceptron algorithm will continue updating without reaching a point where all points are correctly classified

the XOR function:

The XOR (exclusive OR) function outputs true (or 1) if the number of true inputs is odd; in a two-input scenario, it gives 1 when either x_1 or x_2 is 1, but not both.

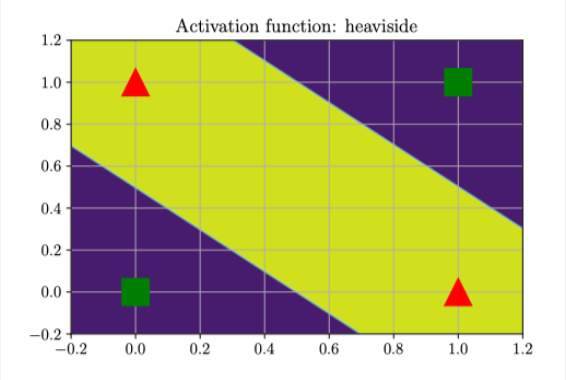


Figure 66: The XOR data points are not linearly separable. That is, we cannot draw a straight line to separate the class of points with output 1 from those with output 0

A single-layer Perceptron is essentially a linear classifier.

- It makes its decisions by weighing input features with certain coefficients, summing them up, and applying a thresholding step function.

Because the decision boundary of a Perceptron is a hyperplane (a line in two dimensions), it cannot solve problems where a non-linear decision boundary is required, such as the XOR problem.

89.2.2 2. Inability to Choose Between Solutions

When a dataset is linearly separable, there could be infinitely many hyperplanes that correctly separate the classes. However, single-layer Perceptron does not have a mechanism to prefer one over another.

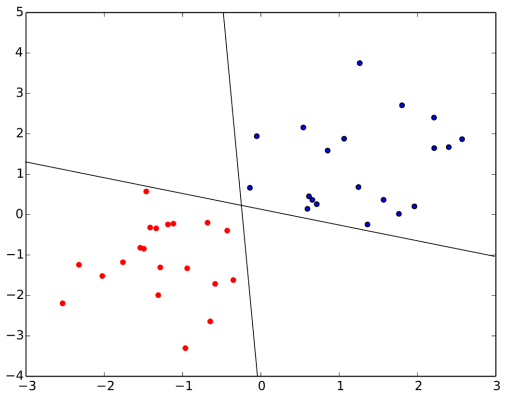


Figure 67: Enter Caption

Ideally, one might want to choose the hyperplane that maximizes the margin between the classes for better generalization (as done by Support Vector Machines), but the Perceptron does not

inherently have this capability.

89.2.3 AI Winter

These issues (esp XOR) publicized by Minsky and Papert in their book "Perceptrons" in the late 1960s.

This finding led to the first AI winter. Many in the field turned their attention to other areas, considering neural networks to be a dead end.

This perspective persisted until the 1980s and 1990s, when the development of multi-layer networks and the backpropagation algorithm revived interest in the field. This revival was based on the realization that with one or more hidden layers, neural networks could indeed model non-linear functions, effectively addressing the shortcomings of the Perceptron.

90 Feed-Forward Neural Networks (FFNN) or Multi-Layer Perceptron (MLP)

The single-layer Perceptron can be thought of as the simplest FFNN with no hidden layers.

What we've done thus far is learn parameters for a larger model:

$$f(x;\theta) = \theta\phi(x)$$

Where $\phi(x)$ represents some transformatuon of the inputs, potentially non-linear (kernels etc).

But what if we could also learn ϕ....

90.0.1 Hierarchical Learning of Feature Representations

The feed-forward architecture is built on the premise that more complex representations can be learned by composing simpler ones.

This is embodied in the structure of the network, where each layer's output serves as the input to the subsequent layer.

$$f(x;\theta_1,\theta_2) = \theta_1\phi(x;\theta_2)$$

- $\phi(x)$ represents some transformatuon of the inputs, potentially non-linear
- $\theta_1, \theta_2, \cdots \theta_n$ represent the parameters at each layer of the network

Formally:

$$f(x;\theta = (\theta_1,\ldots,\theta_L)) = f_L(f_{L-1}(\ldots(f_1(x))))$$

Where $f_l(x) = f(x;\theta_l)$

$$f(x;\theta) = \theta_n\phi(\ldots(\theta_2\phi(\theta_1\phi(x)))\ldots)$$

Demonstrates that each layer of a FFNN applies a set of weights θ_i to its input, and then an activation function ϕ to introduce non-linearity

90.0.2 Learning ϕ

The ability to learn ϕ (the transformation at each layer) is crucial.

In traditional machine learning models, feature transformation is often a manual process based on domain knowledge.

With FFNNs, these transformations are learned from data.

The function ϕ at each layer can be seen as a feature extractor that transforms the data into a more abstract representation, which should ideally be more useful for the task at hand, such as classification or regression.

Deep Hierarchies in FFNN

A deep FFNN, which has many layers of these transformations, can learn very complex patterns. Each layer's output (its feature representation) is used by the next layer as its input. Through training, the network adjusts its weights (the parameters θ) across all layers to minimize some loss function.

Parameters Learning In the context of FFNN, we are not just learning a single set of parameters, but a series of parameter sets $(\theta_1, \theta_2 \ldots)$, each corresponding to a different layer in the network. The learning process involves finding the best values for all these parameters so that the network can accurately map inputs to outputs.

91 Composing Functions

91.1 Assuming only Linear Stacking

Linear Transformations

In a feed-forward network, each layer performs a linear transformation of its input data.

E.g. if a layer has n_1 hidden units (also called neurons) and receives d features from the input or the previous layer, the transformation it performs can be represented by a matrix θ_1 of size $n_1 \times d$

Composing Linear Layers

If we have multiple layers $\theta_1, \theta_2, \ldots \theta_L$, each one taking output of prev layers as its input we have a sequence of matrix multiplications.

Suppose we want m outputs, and we have $dfeatures$:

Composite function:

$$f(x; \theta = (\theta_1, \ldots, \theta_L)) = f_L(f_{L-1}(\ldots f_2(f_1(x)) \ldots))$$

Where $f_l(x) = \theta_l x$

- θ_1 is $n_1 \times d$ for n_1 hidden units
- θ_l is $n_l \times n_{l-1}$
- θ_L is $m \times n_{L-1}$

Limitations of Linear Stacking

If all functions f_i are purely linear, then stacking them together doesn't give us the benefits of a deep network, as the compisiton of linear functions is still a linear function.

Mathematically, if you only have linear transformations without any non-linearity between them, the entire network's effect is equivalent to a single linear transformation from the input to the output.

= flattens all intermediate layers. You effectively cannot throw more info into this model than what you can put into an $m \times d$ matrix...

Taking the above example, where $f_l(x) = \theta_l x$...

- θ_1 is $n_1 \times d$ for n_1 hidden units
- θ_l is $n_l \times n_{l-1}$
- θ_L is $m \times n_{L-1}$

...then $f(x;\theta) = \theta_L \theta_{L-1} \cdot \theta_1 x$

If θ_1 is a $d \times n_1$ matrix; θ_2 is an $n_2 \times n_1$ matrix; and so on.
The output θ_L would be an $m \times n_{L-1}$ matrix for m outputs.

If you multiply all these matrices together, you get an effective transformation matrix M of dimension $d \times m$.

This is not very helpful: when everything is linear we end up with a single weight matrix, which is necessarily linear.

Non-Linearity To make a deep network powerful and able to capture complex relationships in data, we introduce non-linear activation functions after each linear transformation. Common choices are sigmoid, hyperbolic tangent (tanh), and rectified linear unit (ReLU). This non-linearity is essential because it allows the network to learn and represent more than just a simple linear function of the input data. Without non-linearity, no matter how deep the network is, it would be functionally no different than a single-layer Perceptron.

NB!

Without non-linear activation functions, no matter how many layers the network has, it would still behave like a linear model because the composition of linear functions is linear

91.2 Activation Function

Introduce non-linearity, allowing the network to learn and model complex relationships between the inputs and outputs. (prevents collapse into a single uninteresting linear model)

Combined into alternating layers: linear > non linear > linear...

- Linear layer = where we are learning parameters
- Non-linear layers = to prevent collapse

91.2.1 Step Function

Used in the original Perceptron model; a simple threshold function that activates (outputs 1) if the weighted sum of inputs is greater than zero and deactivates (outputs 0) otherwise.

$$f(x;\theta) = \mathbb{I}(X\beta \geq 0)$$

It's a binary function and lacks the nuance needed for modeling the complex, graded behaviors seen in real-world data.

91.2.2 Softmax / Logit

The softmax function is often used in the final layer of a neural network classifier to represent a categorical distribution – that is, the probability that an input belongs to each of several categories.

For binary classification, the softmax function reduces to the logistic sigmoid (logit) function:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

This function takes a real-valued number and squashes it into a range between 0 and 1, which is interpretable as a probability.

A nice property is that it is non-parameterised - there's nothing that needs to be learned, we can just throw it in

91.2.3 Rectified Linear Units (ReLUs)

The ReLU function is given by:

$$\text{ReLU}(x) = \max(x, 0)$$

This activation function has become very popular due to its simplicity and effectiveness. It introduces non-linearity while being computationally efficient and facilitating the optimization process, especially in deep networks.

It's linear (and so preserves the properties of linear models) for positive values and zero for negative values.

91.2.4 Variations on ReLUs

- **Swish**: The Swish function is a smooth, non-monotonic function defined as:

  $$\text{Swish}(x) = x \cdot \sigma(x)$$

  It has been found to sometimes outperform ReLUs because it has a non-zero gradient for all input values, which helps mitigate the "dying ReLU" problem where neurons can become inactive and only output zero.

- **Leaky ReLU**: To address the issue of dying ReLUs, Leaky ReLU allows a small, non-zero gradient when the unit is not active:

  $$\text{LeakyReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases}$$

  where α is

This maintains some kind of relationship between inputs and outputs at all levels of inputs; it's just different for high vs low values.

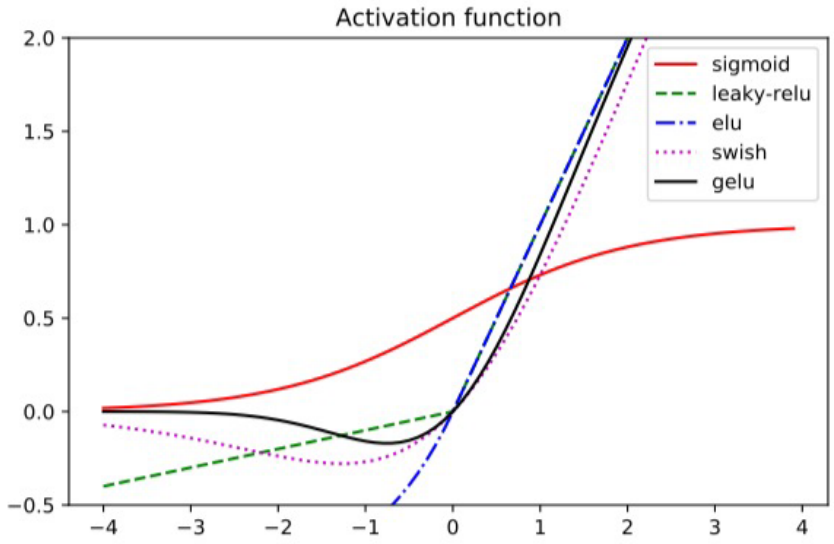


Figure 68: Enter Caption

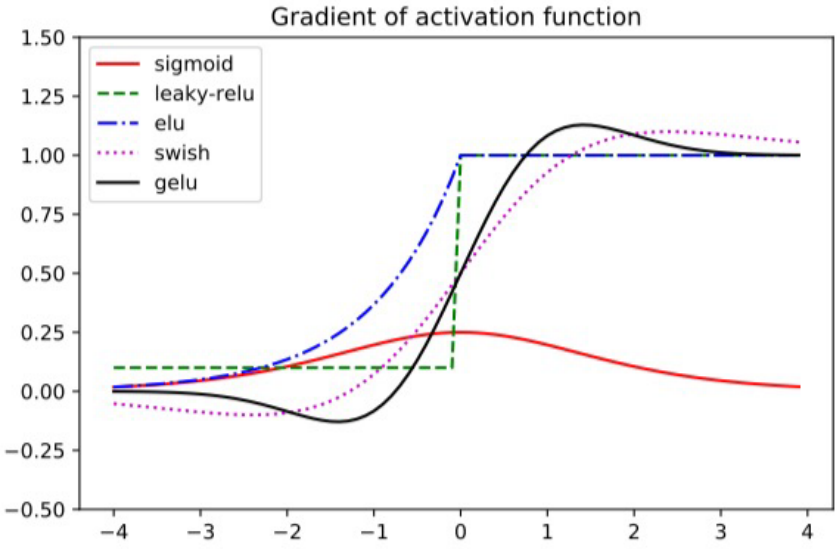


Figure 69: Enter Caption

92 SGD (Training an NN 1)

In simplest case, done using a method called stochastic gradient descent. SGD is an optimization algorithm used to minimize a (loss) function.

92.1 Update rule:

$$\theta(t+1) = \theta(t) - \frac{\eta_t}{B} \sum_b \nabla_\theta \ell(y_b, f(x_b, \theta))$$

Where

- η = learning rate (scalar)
- $\nabla_\theta \ell(y_b, f(x_b, \theta))$ = gradient of the loss function wrt parameter

92.2 Mini-batch Gradient Descent

- a variant of SGD.
Instead of calculating the gradient of the loss function using the entire dataset (which can be computationally expensive), we use a small random subset of the data called a mini-batch.

Reduces variance in the parameter updates and can lead to faster convergence.

92.3 Challenge of Gradient Computation

NN = extremly complex functions (that's the point)

Computing the gradient of the loss function with respect to each parameter isn't straightforward because it's not always clear how each parameter affects the loss.

"Credit assignment": determining how much each parameter contributed to the final outcome.

Solution comes down to lots of applications of the Chain Rule for Derivatives

= **Backpropagation**

= Chain rule application to calculate gradients efficiently for networks with many layers (deep networks).

2 passes through the network:

1. **Forward Pass**: Inputs are passed through the network to compute the outputs and the loss.
2. **Backward Pass**: The gradient of the loss is propagated back through the network to compute the gradients of the loss with respect to each parameter.

We compute the gradient of the loss function at the final layer and then use the chain rule to successively calculate the gradients for each layer from the output end to the input end.
Credit Assignment & Taylor Series The credit assignment problem involves figuring out which weights and biases to adjust to decrease the loss.

Conceptually, you can think of optimizing the network's parameters as a process of making small, iterative improvements based on a local linear approximation of the loss function around the current parameters (hence the reference to a Taylor series expansion).

Back-propagation helps in assigning "credit" by quantifying how much a small change in each parameter will impact the loss.

93 Back-propagation (Training an NN 2)

Calculates the gradient of the loss function with respect to each weight in the network

This allows us to update the weights in the direction that minimizes the loss, using gradient-based optimization methods such as Stochastic Gradient Descent (SGD). Let's dissect the provided process.

93.1 Function composition in NNs / Layers as Functions

A neural network can be thought of as a composition of functions, like:

$$\mathbf{o} = f_L \circ f_{L-1} \circ \cdots \circ f_1(\mathbf{x})$$

where f_i represents the function computed by layer i, $\mathbf{x}$ is the input, and $\mathbf{o}$ is the output.

Each layer f_i typically consists of a linear transformation followed by a non-linear activation. For instance:

- $f_i : \mathbb{R}^{n_{i-1}} \to \mathbb{R}^{n_i}$ maps the input of that layer from a space of n_{i-1} dimensions to a space of n_i dimensions.
- $n > m$ signifies there are more input features n than output features m, which is common in many real-life problems.

93.2 Chain Rule & Jacobians

The gradient of the loss with respect to a certain layer's inputs can be computed by taking the product of the gradients of all subsequent layers' transformations.

In matrix form, this is often represented by Jacobians J, which are matrices of partial derivatives.

If we have a sequence of functions $f_1, f_2, \ldots, f_L$, the gradient with respect to the inputs of f_1 involves the Jacobians of all the functions up to the output:

$$\frac{\partial \mathbf{o}}{\partial \mathbf{x}_1} = J_{f_L}(\mathbf{x}_L) \cdot J_{f_{L-1}}(\mathbf{x}_{L-1}) \cdot \ldots \cdot J_{f_1}(\mathbf{x}_1)$$

where each $J_{f_i}(\mathbf{x}_i)$ represents the Jacobian matrix of partial derivatives of the function f_i with respect to its input $\mathbf{x}_i$.

93.3 Gradient Computation

During the backward pass of backpropagation, we start at the output and work our way back, layer by layer, to the input.

At each step, we compute the gradient of the loss with respect to the layer's output and then use the chain rule to "backpropagate" this gradient through the layer.

This requires computing the gradient of the layer's activation function and then the gradient of its linear transformation.

For a neural network with parameters θ, the backpropagation algorithm systematically applies the chain rule to compute the gradient $\nabla_\theta L$ of the loss function L with respect to the parameters θ.

These gradients are then used to update the parameters in the direction that minimizes the loss.

Example: suppose we have few outputs relative to input features: $x > m$:

Our function of interest is o:

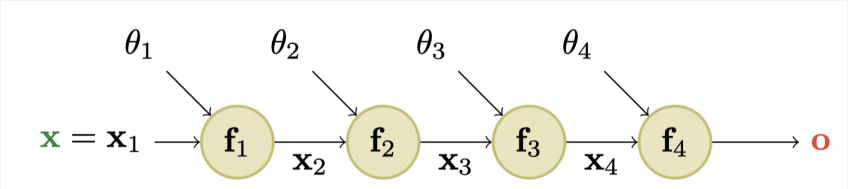


Figure 70: NB: 4th function here is loss function

$$o = f_4(f_3(f_2)f_1(x)))) = f_4 \circ f_3 \circ f_2 \circ f_1(\mathbf{x})$$

In terms of numbers of features at each layer (neurons?):

- $f_1 : \mathbb{R}^n \to \mathbb{R}^{m_1}$
- $f_2 : \mathbb{R}^{m_1} \to \mathbb{R}^{m_2}$
- $f_3 : \mathbb{R}^{m_2} \to \mathbb{R}^{m_3}$
- $f_4 : \mathbb{R}^{m_3} \to \mathbb{R}^{m}$

Where n = number of features; m_1 = number of features in 1st hidden layer....

In terms of gradients of o can be expressed as: (??)

$$\begin{aligned}
\frac{\partial \mathbf{o}}{\partial \mathbf{x}_1} &= \\
\textit{how fns change wrt change in input:} &= \frac{\partial \mathbf{o}}{\partial \mathbf{x}_4}\frac{\partial \mathbf{x}_4}{\partial \mathbf{x}_3}\frac{\partial \mathbf{x}_3}{\partial \mathbf{x}_2}\frac{\partial \mathbf{x}_2}{\partial \mathbf{x}_1} \\
\textit{part derives of features wrt its input:} &= \frac{\partial f_4(x_4)}{\partial \mathbf{x}_4}\frac{\partial f_3(\mathbf{x}_3)}{\partial \mathbf{x}_3}\frac{\partial f_2(\mathbf{x}_2)}{\partial \mathbf{x}_2}\frac{\partial f_1(\mathbf{x})}{\partial \mathbf{x}} \\
\textit{as Jacobians:} &= J_{f_4}(x_4) \cdot J_{f_3}(x_3) \cdot J_{f_2}(x_2) \cdot J_{f_1}(x)
\end{aligned}$$

Where Jacobian is a matrix of the gradient of each function, such that the overall function Jacobian:

$$J_f(\mathbf{x}) = \begin{pmatrix} \nabla f_1(\mathbf{x})^T \\ \vdots \\ \nabla f_m(\mathbf{x})^T \end{pmatrix} \in \mathbb{R}^{m \times n}$$

= how changes of input affect changes in output

93.4 Backpropagation for an MLP

Defining the MLP Structure

The MLP consists of

- **hidden layers** where each layer performs a specific transformation of the data passing through it. (Allows it to hierarchically learn complex feature transformations)

- **At the final layer, a loss function** $\mathcal{L}$ measures how far the network's predictions ($f_B(x_B)$) is from the true value (y).
- For regression tasks: MSE $\mathcal{L}(x_B, y) = \frac{1}{2}(x\beta - y)^2$

So, we define an MLP:

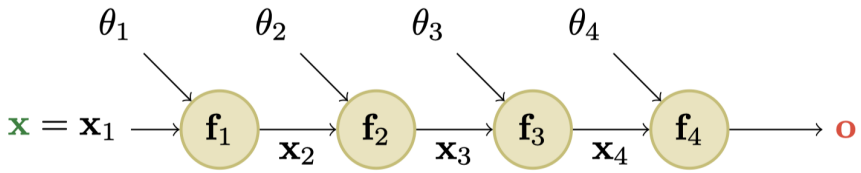


Figure 71: Enter Caption

Layers in the MLP

- **Final loss layer:** $\mathcal{L}(x_4, y) = \frac{1}{2}\|(x_x\beta - y)\|^2$
- **Linear layer:** $x_4 = f_3(x_3, \theta_3) = W_2 x_3$
  - takes inputs, applies weights
  - W_2 is the weight matrix associated with this layer
- **Nonlinear activation:** $x_3 = f_2(x_2), \varnothing = \phi(x_2)$
  - activation function: there are no parameters to optimise; we don't have to worry about it in back propagation
- **Linear layer:** $x_2 = f_1(x), \theta_1 = W_1 x$
  - takes inputs, applies weights
  - W_1 is the weight matrix associated with this layer

Gradient Calculation for Back-propagation Computing the gradient of the loss with respect to each weight in the network.

This requires applying the chain rule in reverse order, from the output layer back to the input layer.

For a given weight matrix W, the gradient is denoted as $\nabla_W \mathcal{L}$.

- Calculate the gradient of the loss with respect to the output of the final layer:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}_B} \frac{\partial \mathbf{x}_B}{\partial \mathbf{L}}$$

- Use the chain rule to find the gradient with respect to the weights of the last layer $\mathbf{W}_2$:

$$\frac{\partial \mathcal{L}}{\partial \theta_2} \frac{\partial \theta_2}{\partial \mathbf{W}_2}$$

- Propagate the gradient backward to the output of the non-linear activation layer:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}_C} \frac{\partial \mathbf{x}_C}{\partial \mathcal{L}}$$

- Continue applying the chain rule to find the gradient with respect to the weights of the preceding layer $\mathbf{W}_1$:

$$\frac{\partial \mathcal{L}}{\partial \theta_1} \frac{\partial \theta_1}{\partial \mathbf{W}_1}$$

NB: stacking - The partial derivatives needed at each step are often stored and reused, which is an efficient implementation known as dynamic programming.

$$\frac{\partial \mathcal{L}}{\partial \theta_3} = \frac{\partial \mathcal{L}}{\partial x_4} \cdot \frac{\partial x_4}{\partial \theta_3}$$

$$\frac{\partial \mathcal{L}}{\partial \theta_2} = \frac{\partial \mathcal{L}}{\partial x_4} \cdot \frac{\partial x_4}{\partial x_3} \cdot \frac{\partial x_3}{\partial \theta_2}$$

$$\frac{\partial \mathcal{L}}{\partial \theta_1} = \frac{\partial \mathcal{L}}{\partial x_4} \cdot \frac{\partial x_4}{\partial x_3} \cdot \frac{\partial x_3}{\partial x_2} \cdot \frac{\partial x_2}{\partial \theta_1}$$

The computation of the gradient at each layer accounts for the influence of each parameter (weight and bias) on the output error. This systematic approach allows us to "assign credit" or "blame" to the parameters for their contribution to the error.

Since neural networks represent a complex, non-linear mapping, there's no closed-form solution for the optimal weights. Hence, iterative optimization methods like SGD are employed.

SGD makes a local linear approximation of the loss function around the current parameter values (considering the Taylor series expansion) and updates the parameters to decrease the loss, aiming to converge to a set of parameters that minimize the loss function over time.

93.5 Algorithmically

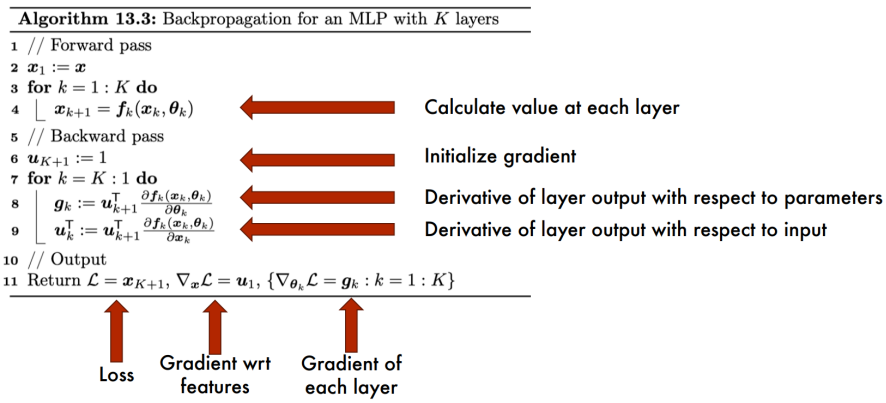


Figure 72: Enter Caption

You can build up this gradient iteratively if you understand each individual layer...

93.6 We still need each layer's derivatives!

When training neural networks, we need to compute the derivatives of each layer with respect to its inputs and weights to update the parameters.

93.6.1 I: Numerical Approximation of derivatives

although it's not typically used for neural networks due to its inefficiency and approximation error.

Involves calculating the gradient by estimating how the function value changes with a very small change in the input.

$$lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

In practice, you pick a small value for h and compute the above quotient to approximate the derivative.
BUT issues w/ Numerical Approximation

- **Computational Cost:** requires at least two evaluations of the network function per parameter to estimate its derivative.
- **Choice of h:** if h is too large, the approximation may be poor. If it's too small, we may encounter rounding errors due to the limitations of floating-point arithmetic in computers.
- **No Structural Advantage:** Neural networks are structured in layers with differentiable activation functions, which can be efficiently exploited using analytical gradients via back-propagation. Numerical methods do not use this structural information, leading to inefficient computations and potential loss of precision.

So we want an analytical solution...

93.6.2 II: Auto-differentiation

Enables the calculation of derivatives through a computational graph of the function.

Decomposes complex functions into simpler, elementary operations whose derivatives are known (hence the term "atomic functions"). Then, by applying the chain rule, it computes the derivatives of the entire function automatically and accurately.
Atomic Functions & Operations
Atomic functions are the basic building blocks of more complex functions.

- addition,
- multiplication,
- trigonometric functions,
- exponentials,
- logarithms,

where we already know how to compute the derivatives.

Combing Operations

Auto-differentiation systems construct a computational graph where nodes represent atomic functions or operations, and edges represent dependencies between these operations. The derivatives are calculated using the following rules:

- **Addition Rule:** If $f(x)$ and $g(x)$ are functions, then the gradient of their sum with respect to x is the sum of their individual gradients:

$$\nabla_x(f(x) + g(x)) = \nabla_x f(x) + \nabla_x g(x)$$

- **Multiplication Rule:** The gradient of the product of two functions $f(x)$ and $g(x)$ with respect to x involves the product rule:

$$\nabla_x(f(x) \cdot g(x)) = f(x) \cdot \nabla_x g(x) + g(x) \cdot \nabla_x f(x)$$

- **Composition Rule:** For a composite function $h(x) = f(g(x))$, the chain rule gives the gradient of $h(x)$ with respect to x as the product of the gradient of $f(u)$ with respect to u and the gradient of $g(x)$ with respect to x:

$$\nabla_x f(g(x)) = \nabla_u f(u)\Big|_{u=g(x)} \cdot \nabla_x g(x)$$

Need for Full Frameworks
While basic numerical libraries like NumPy support a wide range of mathematical operations, they do not inherently support auto-differentiation. This is why full-fledged deep learning frameworks like TensorFlow, PyTorch, and JAX are necessary for neural network development. These frameworks have auto-differentiation capabilities built-in and can automatically calculate the gradients required for training neural networks.

- TensorFlow and PyTorch build dynamic computation graphs that represent the operations of the neural network. When the network performs a forward pass, these frameworks keep track of the operations and are then able to traverse the graph in reverse to compute gradients using backpropagation.
- JAX provides automatic differentiation capabilities through its jax.grad and related functions. It transforms Python and NumPy code into functions that can be auto-differentiated and optimized.

94 MLP Design Recipe

Design an Architecture

Number of Layers

- **Single-Layer Perceptron (SLP):** This configuration has no hidden layers, only an input layer and an output layer. It is only suitable for linearly separable problems.
- **One Hidden Layer:** An MLP with a single hidden layer can approximate any function that contains a continuous mapping from one finite space to another. It is more capable of handling non-linear data compared to an SLP.
- **Multiple Hidden Layers (Deep Learning):** More layers allow the network to learn more complex representations, but they also make the network more computationally intensive and potentially harder to train.

Number of Units in Each Layer

- The number of units in each hidden layer is a parameter that may require tuning. A larger number of units can increase the model's capacity but may lead to overfitting.

- Generally, the size of the input layer corresponds to the dimensionality of the data, and the size of the output layer corresponds to the number of output classes (for classification) or one (for regression).
- The number of units in hidden layers is often determined through experimentation, although common starting points are numbers that form a pyramid shape (decreasing or increasing).

Function Definition at Each Layer
Each layer typically performs a linear transformation followed by a non-linear activation:

- **Linear Transformation:** Each neuron in a layer computes a weighted sum of its inputs, which are the outputs of the previous layer.
- **Activation Function:** This function introduces non-linear properties to the model, allowing it to learn more complex patterns. Common choices include ReLU (Rectified Linear Unit), sigmoid, and tanh.

Design a Loss Function

The choice of a loss function depends on the specific task:

- **Regression:** Mean Squared Error (**MSE**) is commonly used. It measures the average squared difference between the estimated values and the actual value.
- **Classification:** Cross-entropy loss (also known as **Log Loss**) is typical. It measures the performance of a classification model whose output is a probability value between 0 and 1.

Choose an Optimizer

The optimizer is the algorithm used to update weights in the network based on the gradients of the loss function:

- **Stochastic Gradient Descent (SGD):** A simple yet effective approach. Variants like mini-batch gradient descent are commonly used in practice.
- **Adam (Adaptive Moment Estimation):** Combines the advantages of two other extensions of SGD, namely AdaGrad and RMSProp. Adam is popular due to its effective handling of sparse gradients and adaptive learning rate techniques.
- **BFGS (Broyden–Fletcher–Goldfarb–Shanno Algorithm):** A more sophisticated optimization algorithm that uses second-order derivatives and maintains a full approximation of the inverse Hessian matrix. It's more common in smaller, less complex models due to its computational expense.

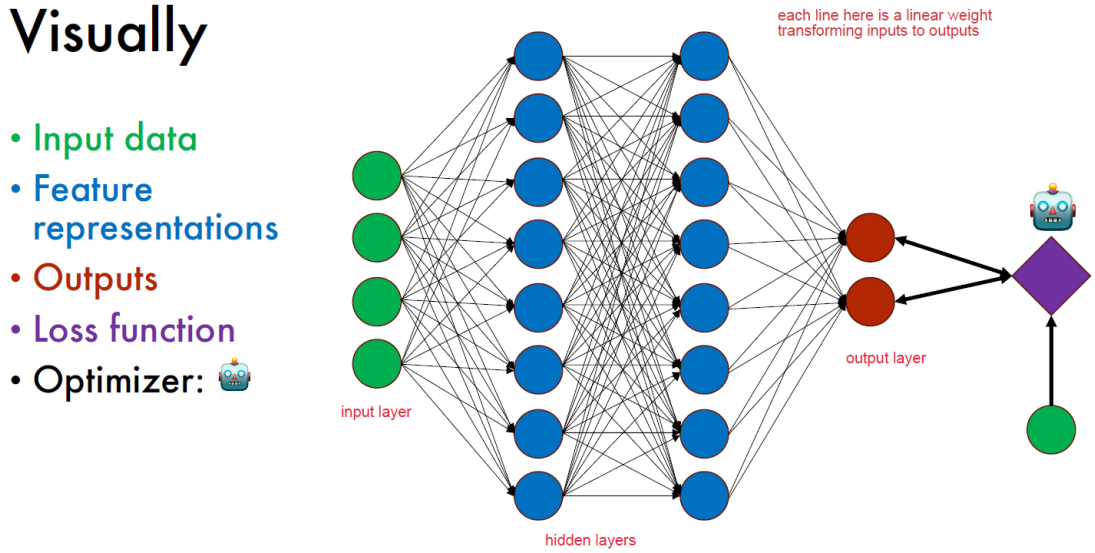


Figure 73: Enter Caption

95 A Simple MLP

1 hidden layer, with lots of hidden units

Useful for both regression and classification tasks depending on the loss function and final layer activation used.

95.1 Architecture of the MLP

95.1.1 Hidden Layer (f_1)

This layer takes an input x with d features.

It transforms the input using a weight matrix θ_1 of dimensions $d \times k$, where k is the number of hidden units.

After the linear transformation, a non-linear activation function (typically sigmoid or ReLU) is applied. Here, a sigmoid (σ) is used, which squashes the output to the range (0, 1).

The output of this layer is a k-dimensional vector representing learned features from the input.

- **Function:** $f_1(x; \theta_1) = [\sigma(\theta_1^1 x), \sigma(\theta_1^2 x), \ldots, \sigma(\theta_1^k x)]$
- **Parameters:** θ_1 contains $d \times k$ parameters, where d is the dimensionality of the input vector x, and k is the number of hidden units.
- **Role: just learns a feature representation** of the input data by applying a non-linear transformation (usually sigmoid) to a linear combination of the inputs. The sigmoid activation ensures that the output of each neuron is between 0 and 1, making the model capable of capturing non-linear relationships between the input features.

95.1.2 Output Layer (f_2)

Takes the k-dimensional output from the hidden layer.

Transforms it using a weight vector θ_2 of dimensions $k \times 1$, thus producing a single output value for regression.

For classification, the output would pass through a sigmoid function to represent a probability.

- **Function:** $f_2(x; \theta_2) = \theta_2 x$
- **Parameters:** θ_2 contains $k \times 1$ parameters, assuming a single output unit for regression.
- **Role: performs a linear regression on the feature representation** provided by the hidden layer f_1. It maps the transformed features to a single continuous output, which can be the predicted value in the case of regression.

95.2 Function Composition

The overall network function f is the composition of f_2 and f_1

$$f(x; \theta) = f_2(f_1(x; \theta_1); \theta_2)$$

The total parameter count is the sum of elements in θ_1 and θ_2:

- $(d \times k) + (k \times 1)$ parameters.

95.3 Loss Functions

For Regression: Mean Squared Error (MSE) is used, defined as:

$$\ell(y, f(x; \theta)) = (y - f(x; \theta))^2$$

For Classification: Log Loss (or Cross-Entropy Loss) can be used when the output is passed through a sigmoid function in the output layer to ensure the outputs are probabilities:

$$\ell(y, f(x; \theta)) = -y \log(f(x; \theta)) + (1 - y) \log(1 - f(x; \theta))$$

This loss function is suitable for binary classification tasks, where y is either 0 or 1.

96 Optimizer

BFGS: Broyden–Fletcher–Goldfarb–Shanno (BFGS) is an optimizer that uses second-order derivatives (the Hessian matrix of second derivatives) to guide the optimization process. It is more sophisticated than simple gradient descent and can converge faster for small to medium-sized problems.

However, its memory requirement, which grows with the number of parameters squared, makes it less feasible for very large models.

97 Practical Considerations

- **Capacity of the Model:** The number of hidden units k affects the model's ability to learn from the data.
  - Too few units may lead to underfitting,
  - Too many can cause overfitting.
- **Transition from Regression to Classification:** By changing the output layer to include a sigmoid activation and adjusting the loss function to log loss, the same architecture can be used for binary classification tasks. This flexibility makes MLPs a popular choice in many different machine learning applications.

98 Choosing an Optimizer

SGD

1.
   - **Basic SGD** - updates weights using a portion of the training data rather than the whole dataset to compute the gradient, which is computationally efficient and reduces memory requirements.
   - **With Momentum** - helps accelerate the gradient vectors in the right direction, thereby leading to faster converging. It does this by adding a fraction of the update vector of the past step to the current step's gradient vector.

2. **BFGS**
   - Suitable for smaller datasets as it involves computation of the Hessian matrix, which can be very large for models with many parameters. BFGS is a quasi-Newton method that approximates the Hessian matrix, necessary for the Newton's method updates.
   - tries to find the exact solution of the gradients equation $\nabla f = 0$ in fewer iterations, making it powerful for small to medium-sized problems.

Adam (Adaptive Moment Estimation)

Update rules:

$$\text{A} : m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$\text{B} : s_t = \beta_2 s_{t-1} + (1-\beta_2) g_t^2$$
$$\text{C - combo} : \theta_t = \theta_{t-1} - \frac{\eta_t}{\sqrt{s_t} + \epsilon} m_t$$

= SGD, with

3.
   - Momentum (exponential weighted avg of the gradient)
   - Normalization (exponential weighted average of the gradient magnitude)
   - Learning rate
   - Avoid division by 0

   1. **A - Updating the running average of the gradients (m_t):**

- m_t is the first moment estimate, which is essentially the mean (hence the symbol m) of the gradients.
- β_1 is a decay rate parameter for the first moment estimate. This is similar to the momentum parameter, controlling the degree to which the previous gradients influence the current update.
- g_t is the gradient at time step t.
- This equation computes an exponentially decaying average of past gradients. The decay rate is controlled by β_1, with typical values around 0.9.

2. **B - Updating the running average of the squared gradients (s_t):**

- s_t is the second moment estimate, corresponding to the uncentered variance of the gradients (hence the symbol s which is standard for variance).
- β_2 is a decay rate parameter for the second moment estimate. It determines how quickly the moving average forgets the earlier observed squared gradients.
- This equation computes an exponentially decaying average of past squared gradients, where β_2 typically has a value around 0.999.

3. **C - Updating the parameters (θ_t):**

- θ_{t-1} is the parameter vector at the previous time step.
- η_t is the learning rate at time step t. It can be fixed or adaptive.
- $\sqrt{s_t + \epsilon}$ is the element-wise square root of the second moment estimate, with ϵ added to improve numerical stability (preventing division by zero). ϵ is a small constant, often around 10^{-8}.
- This equation updates the parameters in the direction that minimizes the loss. It scales the gradient inversely proportional to the square root of the moving average of the squared gradients, which has an effect of adapting the learning rate for each parameter.

Together, these three components of the Adam algorithm adapt the learning rate based on the first and second moments of the gradients. This property makes Adam particularly suitable for problems with sparse gradients and/or noisy data, and it is one of the reasons for Adam's popularity in training deep neural networks.

- Adam = SGD, but with momentum
- β_1 and β_2: These are the exponential decay rates for moment estimates; β_1 is typically around 0.9, and β_2 around 0.999, which are the defaults in most frameworks.
- Adam stores an exponentially decaying average of past squared gradients (s) and past gradients (m).
- **Learning rate (η)**: This is a crucial hyperparameter for Adam, which might need adjustments based on the model's performance during training.
- ϵ: A very small number (like 10^{-8}) to prevent any division by zero in the implementation.

Default Adam Initialization:

- $\beta_1 = 0.9$
- $\beta_2 = 0.999$
- $\epsilon = 10^{-6}$
- $\eta_t = 0.003$

Of these, the learning rate is the main think we'd want to tweak.

98.1 Choosing an Optimizer

For Small Problems: BFGS - as it can rapidly converge to the *exact* solution, provided the problem size is manageable.

- an Analytical solution (?)

For Large-Scale Problems: Adam - efficiently handles large datasets and converges faster than SGD due to its adaptive learning rate mechanisms.

- a numerical approximation ?

99 MLPs are Universal Approximators

Universal Approximation Theorem:
A feedforward network with a single hidden layer containing a finite number of neurons can approximate continuous functions on compact subsets of $\mathcal{R}^n$, under mild assumptions on the activation function.

This means that theoretically, one hidden layer is sufficient to approximate any smooth function to any desired level of accuracy, provided the activation function is non-constant, bounded, and monotonically-increasing.

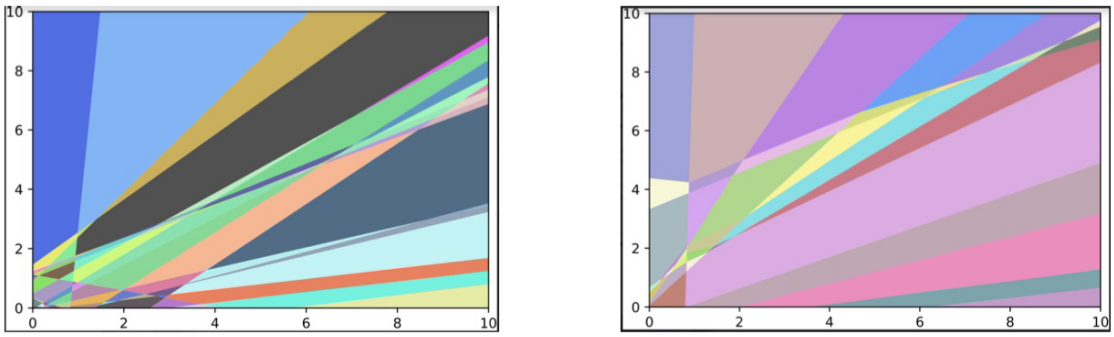


Figure 74: These are the piecewise linear functions learnt using ReLUs; we can carve up the areas of the space

Empirical Performance of Deeper Networks:
While a single hidden layer can approximate any function given sufficient neurons, in practice, using multiple layers often yields better performance.

This is not well supported by the theoretical maths, but it seems deeper architectures are more efficient at learning complex patterns due to their hierarchical structure. Each layer can learn different levels of abstraction, which is particularly useful in tasks like image and speech recognition, where data exhibit hierarchical patterns.

Expressing Compositional Structures:

Hierarchical or compositional structures in data are effectively captured by deeper networks.

For instance, in image processing, lower layers might learn to detect edges, while higher layers may interpret these features to recognize more complex shapes or objects. This compositional learning is a key reason why deep learning has been successful in various complex tasks.

Thus, even though a single hidden layer MLP is theoretically sufficient for universal approximation, deeper networks tend to perform better in practice, particularly in handling complex data sets and learning tasks.

100 NNs as GPs

MLP Converging to a GP: As the number of hidden nodes in an MLP increases to infinity, under certain conditions, the distribution of the MLP's outputs converges to a Gaussian process.

This relationship between deep neural networks and Gaussian processes has been formalized in the study of neural network Gaussian processes (NNGPs) and the neural tangent kernel (NTK).

Neural Tangent Kernel (NTK): The NTK is a concept that has emerged from the analysis of how neural networks evolve during training under gradient descent. It essentially quantifies the change in the outputs of a neural network with respect to infinitesimal changes in its parameters.

The kernel $k(x, y) = g(x)^\top K g(y)$, where $g(x) = \nabla_A f(x; \theta)$, relates to how the outputs for inputs x and y co-vary in the infinite-width limit.

- Here, K is typically the Gram matrix composed of inner products of gradients of the network's output with respect to its parameters.

Kernel Stability During Training: In the infinite width limit, the NTK remains constant during training. This constancy implies that the training dynamics of wide neural networks can be predicted and analyzed using kernel methods, and the network behaves as if it were a linear model in the function space defined by the NTK.

Implications of Architecture on the NTK: The structure of the MLP (e.g., depth, width, activation functions) determines the specific form of the NTK. This kernel, therefore, encapsulates how architecture choices impact learning dynamics and capabilities.

Random Sampling with SGD and GP Analogies: When training an MLP with stochastic gradient descent (SGD), you effectively sample from the function space defined by the network's initialization and architecture. In the infinite-width scenario, this sampling mirrors drawing functions from a Gaussian process defined by the NTK. Each training trajectory (using a different minibatch sequence) may yield slightly different models (like individual MLPs shown in red), but all are samples from the underlying distribution described by the NTK (the GP in blue).
Feature Representations and Regression: In this framework, the MLP learns feature representations, and training essentially becomes a form of regression on these learned features. The

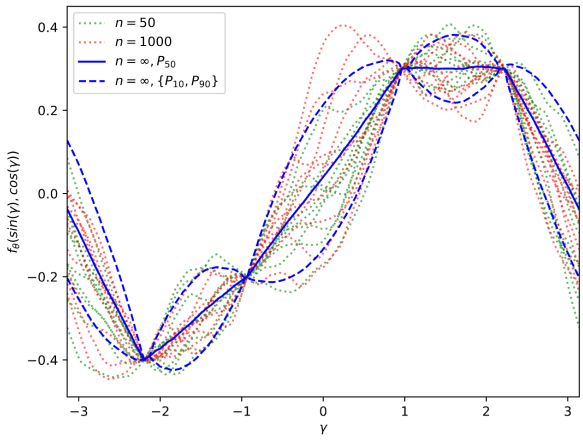


Figure 75: Enter Caption

specifics of these representations depend on the network's architecture, initialization, and the nature of the training data.

This framework of understanding deep learning models using kernel methods and Gaussian processes provides powerful insights into how neural networks learn and how their architecture affects their learning dynamics and capabilities. It also bridges the gap between non-parametric Bayesian methods (like GPs) and parametric models like neural networks.

101 Vanishing / Exploding Gradient

Affects the efficiency of training neural networks, as the number of layers (depth) increases.

Logic of Gradients' Leverage by the Chain Rule

The derivative of the loss function with respect to the weights in earlier layers is calculated using the chain rule.

In a network with L layers, the gradient of the loss with respect to the weights of the first layer involves the product of derivatives across all L layers.

Mathematically, we can represented the separation of the overall gradient by the chain rule:

$$\frac{\partial \mathcal{L}}{\partial z_l} = \frac{\partial \mathcal{L}}{\partial z_L} \frac{\partial z_L}{\partial z_{L-1}} \cdots \frac{\partial z_{l+2}}{\partial z_{l+1}} \frac{\partial z_{l+1}}{\partial z_l}$$

And given the simplifying assumptions below, we can write that as follows I THINK THIS ISN'T QUITE RIGHT?:

$$\frac{\partial \mathcal{L}}{\partial W_1} \approx \left(\prod_{i=1}^{L} J_i \right) \frac{\partial \mathcal{L}}{\partial W_L}$$

- where J_i denotes the Jacobian matrix of the partial derivatives from layer i.

Simplifying Assumption

If we assume that the derivatives between each layer are roughly the same, say J...

- this makes sense: we're assuming layers have similar relationships to each other throughout.

$$\frac{\partial z_{l+1}}{\partial z_l} \approx J$$

...then the gradient of the first layer can be approximated as J^L times the gradient at the last layer. This results in:

$$\frac{\partial \mathcal{L}}{\partial W_1} \approx J^L \frac{\partial \mathcal{L}}{\partial z_L}$$

More generally, this can be written for any layer l as:

$$\frac{\partial \mathcal{L}}{\partial z_l} \approx J^{L-l} \frac{\partial \mathcal{L}}{\partial z_L}$$

Here, we take the matrix of partial derivatives (the Jacobian) and raise it to a power representing the number of layers from l to L, effectively capturing the multiplicative effect of the gradients through the network.

101.1 This repetitive multiplication magnifies any small deviations in the values of J.

Behaviour at the limits

For $\lim_{k\to\infty} p^k = 0$ if $p \in [0, 1]$, but $\lim_{k\to\infty} p_k = \infty$ (??) if $p \in [1, \infty]$.

For a matrix, a similar property holds based on the eigenvalues.

Behavior of the Gradients as L increases

$$\lambda_{max} < 1 \text{ gradient converges to } 0$$

Vanishing Gradient: If the norm of J (or any of its eigenvalues, λ) is less than 1, the product J^L approaches zero as L becomes large. Gradients become too small for effective learning, causing training to stagnate.

$$\lambda_{max} > 1 \text{ gradient diverges}$$

Exploding Gradient: Conversely, if the norm of J is greater than 1, J^L grows exponentially with L. This can cause learning to diverge wildly or result in numerical instability (like NaN values).

Matrix Properties and Gradient Dynamics
The characteristics of the matrix J are crucial. The eigenvalues of J determine how the gradients behave:

1. **Exploding gradients**: Occur if any eigenvalue $\lambda > 1$. This means the gradients increase exponentially as they propagate back through the network. -
2. **Vanishing gradients**: Occur if any eigenvalue $\lambda < 1$. Here, gradients diminish exponentially, making it very hard to update weights in earlier layers, particularly in deep networks.

101.2 Gradient Clipping for Exploding Gradients

limiting (or "clipping") the gradients during backpropagation to prevent them from becoming too large, which could cause numerical instability or poor convergence due to overly large updates to the weights.

During the backpropagation process, before the gradient descent update, the gradients are clipped if they exceed a specified threshold c:

Clipping Rule

$$g = \min\left(1, \frac{c}{\|g\|}\right) g$$

Where

- g = gradient vector
- $\|g\|$ = norm of the gradient vector
- c = threshold

This scales down all components of the gradient equally if its norm is greater than c. *Normalises the gradients*

Direction Preservations

Gradient clipping preserves the direction of the gradient vector. This is crucial as it ensures that the optimization process still proceeds in the direction of steepest descent, albeit with a shortened step.

Adaptive Learning Rate Behaviour

Conceptually, gradient clipping can be thought of as adaptively modifying the learning rate. When the gradients are clipped, the learning rate effectively reduces for steps where the gradient norm exceeds the threshold.

Even if they are clipped, while we maybe not be learning everything, we are still just ensuring that we are moving in the right direction, while keeping amount we are moving within each gradient under control.

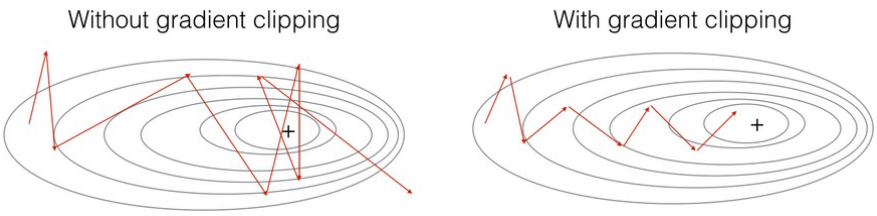


Figure 76: Enter Caption

101.3 Non-saturating Activation Functions for Vanishing Gradients

The usual sigmoid activation function can lead to vanishing gradients, so we use ReLu and its variants.

101.3.1 Intuitive Understanding

- **Non-saturating Nature**: Non-saturating activation functions like ReLU and its variants (like Leaky ReLU and Parametric ReLU) are generally preferred in deep learning architectures to avoid vanishing gradients. They are particularly effective in deeper networks where

the depth could exacerbate the vanishing gradient problem with saturating activations like sigmoid or tanh.

- **Implications of Large Inputs**: While ReLU does not suffer from vanishing gradients for large positive inputs, it still faces issues with negative inputs. Leaky ReLU, by providing a pathway for gradient flow even for negative inputs, helps in maintaining the activation across the network.

101.3.2 Sigmoid Activation Function

- **Function:** $z = \sigma(\beta x)$ where $\sigma(x) = \frac{1}{1+e^{-x}}$.
- **Gradient:** $\frac{d\mathcal{L}}{dz} = z(1-z) \cdot x$.
- **Behavior:** The gradient of the sigmoid function becomes very small when the output z is close to 0 or 1. This is because $z(1-z)$ approaches 0 as z approaches 0 or 1, leading to a vanishing gradient problem. This problem occurs particularly when the inputs are large in magnitude (either positive or negative), causing the sigmoid function to saturate at these extreme values.

101.3.3 ReLU (Rectified Linear Unit) Activation Function

- **Function:** $z = \text{ReLU}(\beta x) = \max(0, \beta x)$.
  - More simply: $\text{ReLu}(x) = max(0, x)$
- **Gradient:** $\frac{d\mathcal{L}}{dz} = \mathbf{1}(z > 0) \cdot x$, where $\mathbf{1}(z > 0)$ is an indicator function that is 1 if $z > 0$ and 0 otherwise.
- **Behavior:** The ReLU function helps mitigate the vanishing gradient problem for positive inputs because the gradient is either 0 (for negative inputs) or equal to the input (for positive inputs). It does not saturate for large positive inputs, which helps maintain healthy gradients during training. However, for very negative inputs, the gradient is zero, which can lead to "dead neurons" where neurons never activate.
- **Benefits**:
  - Simple and computationally efficient.
  - Does not saturate for positive inputs, which helps mitigate the vanishing gradient problem.
  - Generally performs well in many applications and is widely used as a default activation function in many types of neural networks.
- **Drawbacks**:
  - **Dead Neurons Problem**: For inputs less than zero, the gradient is zero, which means that during training, any neuron that outputs a negative value will not update its weights. Over time, if a large number of neurons only output negative values, they stop contributing to the learning process—effectively "dying." This can limit the network's capacity to learn complex patterns.

101.3.4 Leaky ReLU Activation Function

- **Function:** $z = \text{Leaky ReLU}(x) = \max(\alpha x, x)$ where α is a small positive constant (e.g., 0.01).
  - More simply: $\text{Leaky ReLU}(x) = \max(\alpha x, x)$
- **Gradient:** $\frac{d\mathcal{L}}{dz} = \mathbf{1}(x > 0) \cdot x + \alpha \cdot \mathbf{1}(x \leq 0) \cdot x$.
- **Behavior:** The Leaky ReLU addresses the issue of dead neurons by allowing a small, non-zero gradient when x is less than 0 (hence "leaky"). This ensures that there is always some gradient flowing through the network, which prevents neurons from becoming inactive.
- **Benefits**:
  - Addresses the dead neurons problem of ReLU: Even for negative inputs, Leaky ReLU allows a small, non-zero gradient (αx when $x < 0$), which keeps all neurons "alive" and updating throughout the training process.
  - Can lead to better performance on tasks where maintaining a richer representation in the network is beneficial.
  - Often helpful in deeper networks where the dead neuron problem is more likely to impair learning.
- **Drawbacks**:
  - The non-zero slope for negative inputs can introduce a risk of exploding gradients, although this is less common compared to the benefits of preventing dead neurons.
  - The choice of α is critical and can vary depending on the specific application; finding the optimal α might require cross-validation.

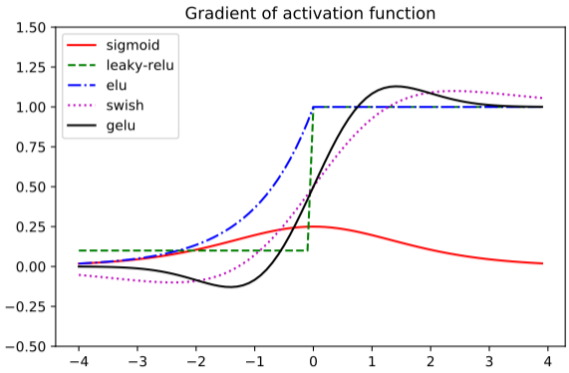


Figure 77: Enter Caption

101.3.5 Choosing Between ReLU and Leaky ReLU

The choice between ReLU and Leaky ReLU may depend on specific task characteristics and empirical performance:

- **ReLU** might be sufficient for many applications, especially if the network is not very deep or if training data is plentiful and well-preprocessed.
- **Leaky ReLU** is often preferred in scenarios where you suspect that the network suffers from the dead neurons problem, particularly in deeper or more complex networks where preserving the flow of gradients through all parts of the network is crucial.

Summary

Sigmoid - goes to zero if z is near 0 or 1.

ReLU - is zero for v neg inputs, but doesn't disappear on the high end.

Leaky ReLu - is never zero.

102 Batch Normalisation

Introduced by Sergey Ioffe and Christian Szegedy in 2015.

Addresses the problem of internal covariate shift, where the distribution of each layer's inputs changes during training, as the parameters of the previous layers change.

This can make training slow and requires careful initialization and a small learning rate.
Idea: Ensure the distribution of activations within a layer have zero mean and unit variance.

102.1 Components of Batch Normalization

1. **Standardization**:
   - **Goal**: Normalize the activations of a layer for each mini-batch to have zero mean and unit variance.
   - **Process**: For each feature, subtract the mini-batch mean and divide by the mini-batch standard deviation.
   - **Math**: Given inputs x from a mini-batch, compute:

$$y = \frac{x - \bar{x}_b}{\sqrt{\sigma_b^2 + \epsilon}}\gamma + \beta$$

     where
     - $\gamma + \beta$ are two learnable parameters.
     - $\sigma_b^2 + \epsilon$ batch-specific variance
     - $\bar{x}_b$ barch-specific mean
     - ϵ is a small constant added for numerical stability

2. **Scaling and Shifting**:
   - After standardization, the outputs are scaled and shifted using parameters that are learned during training. This step ensures that the batch normalization layer can represent the identity transformation and can scale and shift the normalized data as needed for the network to learn effectively.
   - **Math**: The normalized $\hat{x}$ is transformed as:

$$y = \gamma\hat{x} + \beta$$

     where γ and β are parameters learned during training. γ is the scale factor, and β is the shift factor.

102.2 Implementation Details

- **Learnable Parameters**: The scale (γ) and shift (β) parameters are learned during the backpropagation, just like any other learnable parameters in the network.
- **Mini-batch Statistics**: During training, the mean and variance used for normalization are computed on each mini-batch. This ensures that the model remains robust to changes in data distribution during training.

102.3 Behavior at Test Time

On test-set, set $\bar{x_b}$ and $\hat{\sigma_b^2}$ to their expoential weighted moving avg.

- **Fixed Statistics**: At test time, you cannot compute the mean and variance for each mini-batch because the mini-batch size might be different, or you might be running inference on a single example. Instead, the mean and variance computed during training are used.
- **Moving Average**: During training, the exponential moving averages of the batch means and variances are maintained. These averages are then used during inference to normalize the test data. This method ensures consistency between the statistics seen during training and testing, stabilizing the model's output.

102.4 Benefits of Batch Normalization

- **Faster Convergence**: By reducing internal covariate shift, batch normalization allows the use of higher learning rates, accelerating the training convergence.
- **Regularization Effect**: The noise added by the varying means and variances of each mini-batch during training can help to regularize the model, somewhat reducing the need for Dropout.
- **Improved Gradient Flow**: It helps in stabilizing the learning process by normalizing the inputs to layers within the network, which can lead to improved gradient flow through the network and reduce the impact of vanishing or exploding gradients.

103 Regularization

103.1 Weight Decay (L2 Regularization)

- **Concept**: Weight decay, often associated with L2 regularization, penalizes large weights in the model's cost function. The idea is that simpler models with smaller weights are less likely to overfit.
- **Implementation**: The L2 penalty term is added to the loss function $\mathcal{L}$, resulting in a modified loss function $\mathcal{L}_{reg}$ which includes the sum of squares of the weights $\mathbf{w}$, scaled by a regularization parameter λ.
- **Math**: The regularized loss function can be written as:

$$\mathcal{L}_{reg} = \mathcal{L} + \frac{\lambda}{2}\|\mathbf{w}\|^2$$

or

$$\mathcal{L}_{reg} = \mathcal{L} + W^T W$$

This additional term penalizes large weights by adding a cost that increases quadratically with the weight magnitude.

- **@ Optimizer**: In practice, weight decay can often be directly included in the optimization algorithm. For example, in stochastic gradient descent (SGD), the weight update rule includes a term that subtracts a portion of the weight value itself, effectively shrinking the weights during each update.

103.2 Dropout

- **Concept: 'drop out' some edges** - Dropout is a regularization technique that involves randomly "dropping out" (i.e., setting to zero) a number of output features of the layer during training.
- **Implementation**: During training, each neuron (or for convolutional layers, each channel) has a probability p of being temporarily "dropped out" and not contributing to the forward pass or the backpropagation process.
- **Effect: 'spreads out' learning around the network**. Dropout forces the network to be robust as it cannot rely on any single neuron and must learn redundant representations to perform well. It effectively creates a different "thinned" network on each forward pass.
- **Prediction-Time Correction**: At test time, the activations are scaled by the probability p, to account for the fact that all neurons are now present in the network. Alternatively, the network can be run multiple times with dropout still enabled, to generate an "ensemble" of predictions, which can be averaged to get a final prediction. This technique is called Monte Carlo Dropout.

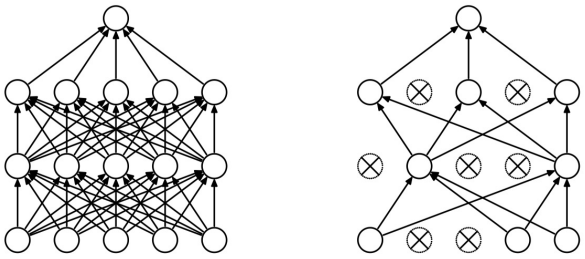


Figure 78: Drop Out

In Summary

Both weight decay and dropout are commonly used in neural networks to improve generalization. Weight decay does this by penalizing the magnitudes of the weights, encouraging smaller weights, while dropout does this by reducing a network's reliance on any single neuron or feature, promoting a more distributed and robust representation. The use of these techniques can be complementary and they are often used together in training deep neural networks to achieve better performance on unseen data.

ML Lecture Notes: Wk 11 — Neural Networks II: Electric Boogaloo

104 Overview

Core concepts of unsupervised learning

- Core NN concept: ***composability***
- More sophisticated things possible through NNs
  - Images
  - Irregular inputs - eg text

105 Recipe for a NN

105.1 Design of an NN

- **Inputs**
  - The number of neurons in the input layer should match the number of features in the input data.
  - Example: For an image of 28x28 pixels, the input layer would have 784 neurons (28*28).
- **Layers** (Input, Hidden, Output)
- **Non-linearities**
  - ReLU (Rectified Linear Unit):
$$f(x) = \max(0, x)$$
  - Sigmoid:
$$f(x) = \frac{1}{1 + e^{-x}}$$
  - Tanh:
$$f(x) = \tanh(x)$$
  - Softmax: Often used in the output layer for classification tasks,
$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$
- **Connections**
  - Fully Connected: Each neuron is connected to every neuron in the next layer.
  - Convolutional: Used in CNNs, where local receptive fields are connected.
  - Recurrent: Used in RNNs, where connections form loops for temporal data.
- **Outputs**
  - Regression: Single neuron with a linear activation function.
  - Classification: Multiple neurons with a softmax activation function (for multi-class classification).

105.2 Design a loss function

- Regression → MSE

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

- Classification → LogLoss (Cross-Entropy Loss)

$$\text{LogLoss} = -\frac{1}{n}\sum_{i=1}^{n}\left[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\right]$$

105.3 Choose an Optimizer

Updates the network's weights to minimize the loss function.
e.g. BFGS, SGD, Adam (in general Adam is best).

Stochastic Gradient Descent (SGD):

$$w = w - \eta \nabla L(w)$$

where η is the learning rate.

Adam (Adaptive Moment Estimation): Combines the advantages of two other extensions of SGD, AdaGrad and RMSProp. The update rule involves estimates of first and second moments of the gradients:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$
$$w_{t+1} = w_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

> Try Tensorflow Playground: https://playground.tensorflow.org/

106 Vanishing / exploding gradients

Addresses issues of gradient-based optimization (i.e. backpropagation)

Chain rule in Backpropogation Suppose L layers:

$$\frac{\partial \mathcal{L}}{\partial z_l} = \frac{\partial \mathcal{L}}{\partial z_L} \cdot \frac{\partial z_L}{\partial z_{L-1}} \cdots \frac{\partial z_{(l+2)}}{\partial z_{(l+1)}} \cdot \frac{\partial z_{l+1}}{\partial z_l}$$

Where:

- z_l represents the l-th layer ???
- z_L represents the pre-activation inputs to the neurons in the Lth layer ????

... ChatGPT preferred this notation:

$$\frac{\partial W^{(l)}}{\partial \mathcal{L}} = \frac{\partial a^{(L)}}{\partial \mathcal{L}} \cdot \frac{\partial z^{(L)}}{\partial a^{(L)}} \cdot \frac{\partial a^{(L-1)}}{\partial z^{(L)}} \cdot \cdots \cdot \frac{\partial z^{(l+1)}}{\partial a^{(l+1)}} \cdot \frac{\partial W^{(l)}}{\partial z^{(l+1)}}$$

Where:

- $a^(k)$ represents the activation in the kth layer
- $z^(k)$ represents the pre-activation inputs to the neurons in the kth layer
- Each term in the product is a partial derivative, representing how changes in one layer affect the next.

106.1 Assumption of Similar Relationships

Suppose that the derivatives $\frac{\partial Z_{l+1}}{\partial Z_l}$ are approximately constant for each layer k and can be represented by some constant J

$$\frac{\partial Z_{l+1}}{\partial Z_l} \approx J$$

This makes some sense: we're assuming layers have similar relationships to each other throughout.

Then...

106.2 Gradient Expression Simplification

Then, the gradient of the loss function L with respect to the weights in layer l can be approximated as:

ChatGPT:

$$\frac{\partial L}{\partial W^{(l)}} \approx J^{L-l} \cdot \text{other terms}$$

$$\frac{\partial \mathcal{L}}{\partial Z_l} \approx J^{L-l} \frac{\partial \mathcal{L}}{\partial Z_l}$$

106.3 Limits & Their Implications

If $J \in 0, 1$: as the number of layers L increases, J^{L-l} approaches 0. This results in vanishing gradients

$$\lim_{L \to \infty} J^{L-l} = 0$$

If $J \in (1, \infty)$, as the number of layers L increases, J^{L-l} grows exponentially. This results in exploding gradients:

$$\lim_{L \to \infty} J^{L-l} = \infty \text{ for } J \in (1, \infty)$$

For a matrix, a similar property holds based on the eigenvalues.
If $\lambda > 1$ the gradient diverges, if $\lambda < 1$ the gradient converges to zero.

106.4 Gradient clipping for exploding gradients

When gradients become too large, they can destabilize the training process by causing extreme updates to the model's weights, leading to divergence.

Gradient clipping modifies the gradient before it is used to update the weights. Specifically, it limits the size of the gradient. If the gradient is too large (above a certain threshold or constant c), it is scaled down to ensure it doesn't exceed this threshold.

Mathematically, if g is the computed gradient and $\|g\|$ is its norm, gradient clipping is defined as:

$$g' = \min\left(1, \frac{c}{\|g\|}\right) \cdot g$$

where:

- g is the original gradient,
- g' is the clipped gradient,
- c is a constant threshold for the maximum allowed gradient norm.

Direction Preservation

Clipping ensures that the gradient never exceeds the constant c, while preserving its direction. Thus, the optimization process continues to move in the correct direction, but in a controlled manner.

Interpretation as Adaptive Learning Rate

You can think of gradient clipping as adaptive learning rate scaling. By limiting the gradient's size, it effectively reduces the learning rate dynamically when gradients get too large. This prevents erratic jumps in parameter space, keeping training stable.

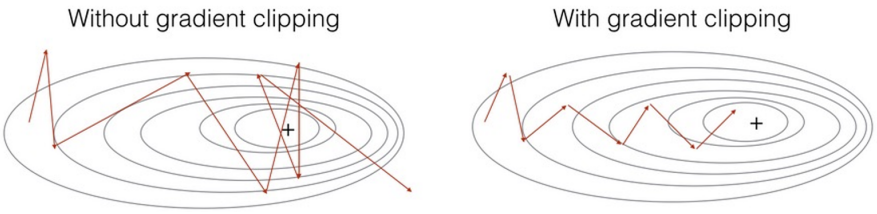


Figure 79: Gradient Clipping

106.5 Vanishing Gradients

The vanishing gradient problem occurs when gradients become very small as they are propagated backward through a deep neural network (particularly severe in deep networks where the gradients diminish exponentially as they are passed back through many layers).

When this happens, the updates to the network's weights become negligible, and the network stops learning.

If you have a v big network, you are likely to have a large input at some point -> gradient goes to zero -> stop being able to learn what the right feature represrntations are for the bottom part of the network

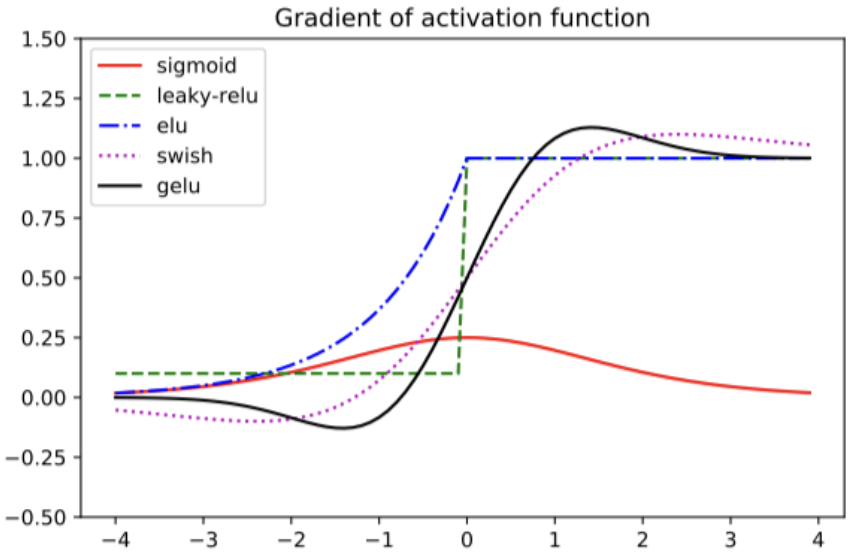


Figure 80: Shows how activation functions input changes as a function of the size of the input

Different activation functions exhibit varying behaviors with respect to the vanishing gradient problem.

1. Sigmoid Activation Function The sigmoid function is defined as:

$$z = \sigma(\beta x) = \frac{1}{1 + e^{-\beta x}}$$

The gradient of the loss function with respect to the input x is:

$$\frac{\partial \mathcal{L}}{\partial x} = z(1 - z)\beta x$$

This gradient approaches zero when z is near 0 or 1. Since the sigmoid function outputs values between 0 and 1, for very large or small inputs x, the gradient becomes extremely small, leading to the vanishing gradient problem.

Intuition: The gradient will disappear if the inputs get too large, and in deep networks, this can happen every time the sigmoid function is applied.

2. ReLU (Rectified Linear Unit) Activation Function The ReLU function is defined as:

$$z = \text{ReLU}(\beta x) = \max(0, \beta x)$$

The gradient of the loss with respect to x is:

$$\frac{\partial \mathcal{L}}{\partial x} = \mathbb{I}(z > 0) \cdot \beta x$$

For negative inputs, the ReLU gradient is zero, which means the gradient disappears for very negative values. However, for positive inputs, the gradient does not shrink as x increases, which helps mitigate the vanishing gradient problem.

Intuition: ReLU does not suffer from the vanishing gradient problem for positive inputs, although the gradient can become zero for negative inputs.

3. Leaky ReLU Activation Function Leaky ReLU is a variation of ReLU and is defined as:

$$z = \text{Leaky ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases}$$

where α is a small positive constant (e.g., 0.01). The gradient for Leaky ReLU is:

$$\frac{\partial \mathcal{L}}{\partial x} = \mathbb{I}(z > 0) \cdot x + \alpha \mathbb{I}(z \leq 0) \cdot x$$

Unlike ReLU, Leaky ReLU ensures that the gradient is never zero, even for negative inputs, thereby preventing the vanishing gradient problem.

Intuition: Leaky ReLU fixes the issue of zero gradients for negative inputs, ensuring that gradients never fully vanish.

Gradient Behavior as Input Size Increases

As input values become large:

- **Sigmoid:** The gradient vanishes as the output approaches its saturation points (0 or 1).
- **ReLU:** The gradient is non-zero for large positive inputs, but vanishes for negative inputs.
- **Leaky ReLU:** The gradient never vanishes, even for negative inputs, ensuring continuous learning.

Impact on Deep Networks

In very deep networks, the vanishing gradient problem is almost inevitable if using saturating activation functions like the **sigmoid**, as each application of the sigmoid diminishes the gradient. In contrast, **ReLU** does not shrink gradients for positive inputs, thus reducing the vanishing gradient problem. However, the issue of zero gradients for negative inputs can be addressed by using **Leaky ReLU**, which introduces a small gradient for negative inputs and prevents the complete vanishing of gradients.

Practical Considerations

- **ReLU** is commonly used due to its simplicity and effectiveness at preventing vanishing gradients.
- **Sigmoid** may still be used when smoothness is important, though it risks the vanishing gradient problem.
- **Leaky ReLU** is useful if you want to ensure that gradients are non-zero even for negative inputs, patching up ReLU's issue of zero gradients.

In practice, combining different non-linear activation functions can improve the performance and stability of deep networks.

106.6 Batch Normalization

Avoids vanishing gradient problem because it means you are rarely getting activations which are really big / small.

Maintains the same amount of variability at each layer; stops activation layer collapsing to a series of 0s.

Conducted both within each batch, and within each layer.

Batch Normalization is a technique used to normalize the activations of a neural network layer to ensure that they have a zero mean and unit variance. This helps in stabilizing and accelerating the training process by reducing internal covariate shift, which occurs when the distribution of activations changes during training.

106.6.1 The Idea

The main idea behind batch normalization is to ensure that the distribution of activations within a layer has zero mean and unit variance, making the training more stable and allowing for faster convergence.

106.6.2 Computation

Given an input x and an output y, batch normalization is computed as follows:

$$y = \frac{x - \bar{x}_{\text{batch}}}{\sigma^2_{\text{batch}}}\gamma + \beta$$

where:

- $\bar{x}_{\text{batch}}$ is the batch-specific mean,
- σ^2_{batch} is the batch-specific variation of inputs,
- γ and β are learnable parameters that allow the model to scale and shift the normalized output.

106.6.3 Learnable Parameters

Batch normalization introduces two learnable parameters, γ and β, which allow the model to recover the original distribution of activations if needed:

- γ: Scales the normalized activation.
- β: Shifts the normalized activation.

106.6.4 Handling Test Set

During training, μ_{batch} and σ^2_{batch} are computed from the current mini-batch. However, during testing (or inference), the batch-specific mean and variance are replaced by an exponential weighted moving average of these statistics computed during training.

$$\bar{x}_{\text{test}} = \text{moving average of } \bar{x}_{\text{batch}}, \quad \sigma^2_{\text{test}} = \text{moving average of } \sigma^2_{\text{batch}}$$

These moving averages are updated throughout training and used for the test set to ensure consistent normalization.

106.6.5 Key Benefits

Batch normalization offers several benefits:

- It reduces internal covariate shift, stabilizing the training process.
- It allows for higher learning rates, speeding up convergence.
- It acts as a regularizer, reducing the need for other forms of regularization (such as dropout).

106.7 Regularization

106.7.1 Weight Decay (L2 Regularization)

> Take all weights, put them in single vector, square them, add them together, apply some penalty... = ridge-style regularisation

Weight decay adds a penalty on the size of the weights, discouraging large weights and encouraging simpler models. This is similar to the ridge regression penalty (L2 regularization), and it can be expressed as:

$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda \|w\|^2$$

or

$$\mathcal{L}_{\text{reg}} = \mathcal{L} + w^T w$$

where:

- $\mathcal{L}$ is the original loss function,
- $\|w\|^2 = w^T w$ is the L2 norm of the weights,
- λ is a regularization coefficient that controls the strength of the penalty.

This penalty encourages the optimizer to find weight values that are small, thereby simplifying the model and improving generalization.

Implementation: Weight decay is typically an option included ***in the optimizer*** during the training process.

106.7.2 Dropout

More robust models by inducing the building in of redundancy in the learning.

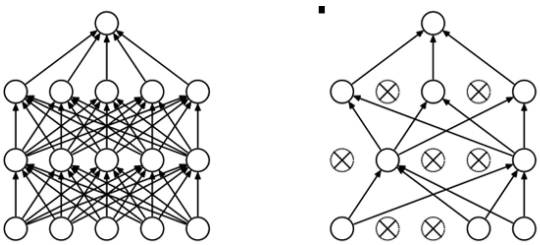


Figure 81: Dropout

Dropout is a regularization technique where, during training, a random subset of edges (or neurons) in the network is "dropped out," or set to zero. This helps in spreading out learning across the network, preventing any particular neuron from becoming overly reliant on specific features.

Training with Dropout: During each training iteration, some neurons are randomly "dropped out," meaning their outputs are set to zero. This forces the network to rely on different subsets of neurons, thereby inducing robustness and redundancy in the learning process.

At Prediction Time: At prediction time, the dropout effect is no longer applied directly. Instead, either the dropped-out units are averaged, or dropout is applied repeatedly to create an ensemble of models, which can give an estimate of uncertainty in the predictions.

106.7.3 Ensemble Method

Dropout introduces an implicit ensemble method by combining multiple subnetworks (created by dropping out different neurons during training). This allows the model to make more robust predictions and, in some cases, provides a measure of uncertainty by averaging the predictions from various dropout masks.

106.7.4 Intuition

Regularization via weight decay or dropout helps prevent overfitting and builds redundancy into the learning process, leading to a more robust model.

- Weight decay encourages small weights and discourages overly complex models.
- Dropout spreads learning across the network, ensuring that no single neuron dominates the prediction process.

107 CNNs

Convolutional neural networks provide a great way to model structured data like images.

They are based around the idea of moving a little window around the input.

107.1 Why is Image Data Hard?

1. High Dimensionality

Images consist of a large number of pixels, each of which represents a feature. Even a small image of size 256×256 pixels in RGB format results in:

$$256 \times 256 \times 3 = 196,608 \text{ features.}$$

+ they are *continuous features*!

Handling such a high number of features requires significant computational resources and can lead to challenges like overfitting if not properly managed.

2. Structured Data

Unlike many tabular datasets, images have a natural *localstructure*. The value of each pixel is often correlated with the values of its neighboring pixels. For example, the color of a pixel is likely similar to the colors of nearby pixels. This structured relationship is crucial for understanding images.

Comparison:

- for other types of data (e.g., in ridge regression or random forests), the order of features is often irrelevant, and features are treated independently (cf: RandForest, we are *randomly* sampling a subset of features at each tree).
- In images, however, the spatial relationships between pixels matter significantly.

3. Translation Invariance

Pixel locations in an image do not carry inherent meaning. For instance, a face can appear on the left-hand side or the right-hand side of an image, but it remains a face regardless of its position. Thus, we don't care about the exact location of objects in the image; rather, we care about what the image represents.

Translation invariance: Our models need to understand that objects can appear anywhere in the image, and the model should be invariant to such translations. A face on the left side of an image should be recognized just as easily if it appears on the right side.

We care more about the object or pattern in the image, not its position. This creates a challengen for models.

4. Continuous Features

The features in images (pixel intensities) are continuous values, unlike many tabular datasets that have categorical or discrete features. This continuity makes modeling images different from other types of data and

107.2 The Big Idea (in CNNs)

What if we cut the image up into little pieces?
Convolutional Neural Networks (CNNs) use a mathematical operation called *convolution* to process image data. The main idea is to extract meaningful features by applying a filter or kernel over small, localized regions of the image.

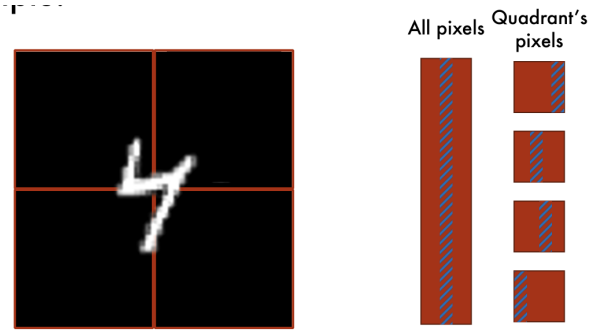


Figure 83: Enter Caption

Cutting the Image into Pieces

Instead of looking at the entire image at once, we divide the image into small pieces or regions, typically using a sliding window approach. This allows the model to capture local patterns and features, such as edges or textures, in the image.

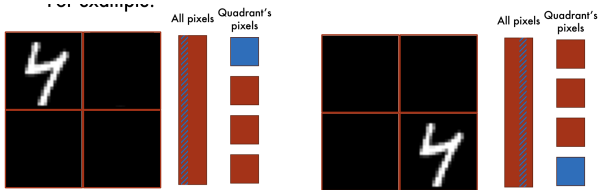


Figure 82: Convolution

However, if the object of interest doesn't neatly fit into one of these regions, it can be difficult for the model to capture it.

107.2.1 Convolutions

Solution: We solve this problem by using *convolutions*, where we move a filter across the image to extract features from different regions.

The Convolution Operator

The idea of a *sliding window* across an image is mathematically described using the convolution operator. A formal definition of convolution between two functions f and g is given as:

$$(f * g)(z) = \int_{\mathbb{R}} f(u)g(z-u)du$$

In the context of CNNs, we apply this operation to an image by using a filter (also known as a kernel) and an input (the image).

In One Dimension In one dimension, the convolution operation between a filter w and an input x is expressed as:

$$(w * x)_i = w_1 x_{i-1} + w_2 x_i + w_3 x_{i+1} + \cdots + w_k x_{i+k-1}$$

Here:

- w is the filter or kernel, which is a set of weights learned by the network,
- x is the input (the image data),

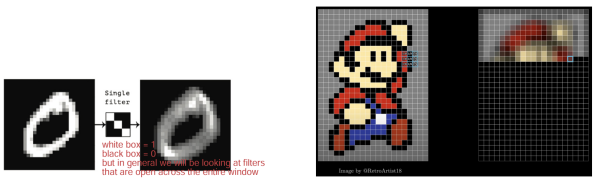


Figure 86: Enter Caption

- The result of the convolution is a *locally weighted sum* of the input, where each element in the input is multiplied by the corresponding filter weight and summed up.

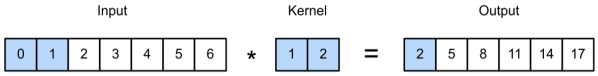


Figure 84: Enter Caption

Convolution in Two Dimensions For image data, which is typically two-dimensional, the convolution operation can be extended to 2D. The operation between a 2D filter W and a 2D input X is:

$$(W * X)_{i,j} = \sum_m \sum_n W_{m,n} X_{i-m,j-n}$$

In slides:

$$[W * X](i,j) = \sum_{u=0}^{H-1} \sum_{v=0}^{W-1} w_{u,v} x_{i+u,j+v}$$

Figure 85: Enter Caption

This means that the filter W is slid over the input X, and for each position (i, j), the elements of the filter are multiplied by the corresponding elements of the input, and the results are summed.

107.2.2 Key Idea

The convolution operation allows the model to focus on *local patterns* in the image, such as edges, corners, and textures, by applying a weighted sum over localized regions of the image. The model learns the filter weights during training, allowing it to detect meaningful patterns that contribute to the final classification or recognition task.

107.3 Connections to Matrix Multiplication

Convolution operations in CNNs can be interpreted as a specific form of *sparse matrix multiplication*.

Consider the convolution operation as a form of matrix multiplication, where the convolution can be represented as:

$$y = Cx$$

For example, the matrix C corresponding to the convolution might look like:

$$C = \begin{bmatrix} w_1 & w_2 & 0 & 0 \\ 0 & w_1 & w_2 & 0 \\ 0 & 0 & w_1 & w_2 \\ 0 & 0 & 0 & 0 \\ w_3 & w_4 & 0 & 0 \\ 0 & w_3 & w_4 & 0 \\ w_1 & 0 & w_2 & 0 \\ 0 & 0 & w_3 & w_4 \end{bmatrix}$$

Here:

- The matrix C is a sparse matrix, meaning it contains a lot of zeros.
- The non-zero entries correspond to the weights of the filter $w = [w_1, w_2, w_3, w_4]$, which are applied to specific parts of the input.
- $x = [x_1, x_2, x_3, x_4, \dots]$ is the input vector, which corresponds to the image or feature map.

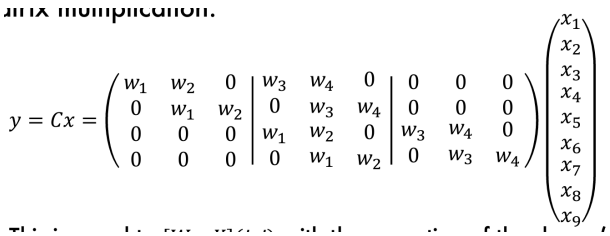


Figure 87: Enter Caption

Sparse Matrix Multiplication

The result of the matrix multiplication Cx is similar to performing a convolution. The convolution operation can be expressed as $[W * X](i, j)$, with the exception that we are dealing with a matrix multiplication rather than a filter being applied directly to the image.

Takeaway: A convolution is essentially a form of *sparse matrix multiplication*. The sparsity arises because we only focus on the local region of the input, corresponding to the sliding window used in convolutions. **Many elements of the matrix are zero, meaning they correspond to regions of the input that are ignored. This implies that we only pay attention to features within the window defined by the filter and ignore everything outside the window.**

107.4 Modifications to convolutions

Modifications to Convolutions

107.4.1 Padding

Padding allows convolutions to handle edge regions, preventing shrinking of the input.

Padding involves adding extra space (usually filled with zeros) around the edges of the input image. This modification allows the convolution operation to cover the border regions of the image, ensuring that features near the edges are not ignored.

Types of Padding:

- **Valid Convolution:** In valid convolution, no padding is added to the input. If the convolution window extends past the edge of the input, those regions are ignored. This results in the output shrinking as more convolutions are applied.
- **Same Convolution:** In same convolution, padding is added so that the output size remains the same as the input size. Typically, the input is padded with zeros, ensuring that the convolution window can still be applied at the edges.

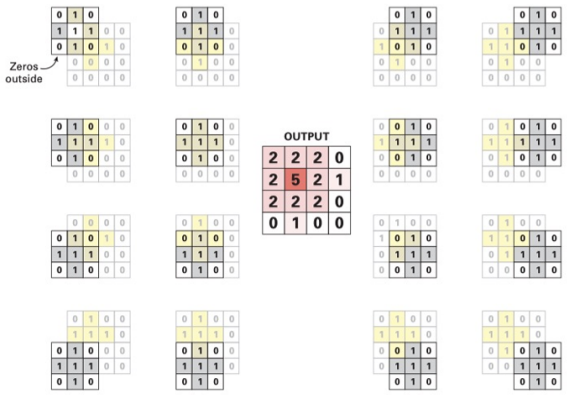


Figure 88: Enter Caption

Padding ensures that the window can still operate even at the boundaries of the image. For example, zero-padding adds zeros around the input, effectively making the input larger while keeping the original image centered.

107.4.2 Strides

Strides control the step size of the convolution window and allow for downsampling by skipping pixels.

Adjacent outputs are v similar, so skip over some of them!

The *stride* is the step size of the sliding convolution window as it moves across the image. By default, the window moves one pixel at a time (stride of 1), but increasing the stride allows the convolution to skip some pixels, reducing the computational cost and shrinking the output.

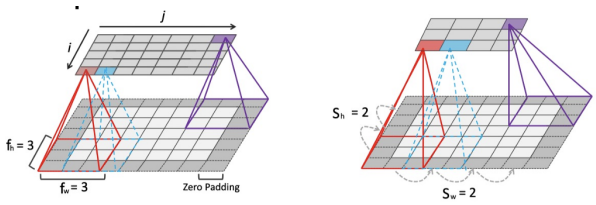


Figure 89: Shows padding, and stride =1, vs stride = 2

How Strides Work:

- **Stride of 1:** The window moves one pixel at a time, ensuring that adjacent windows overlap significantly, usually by about 90%.

- **Stride of 2:** The window skips one pixel between adjacent outputs. This effectively halves the spatial dimensions of the output compared to stride 1.
- **Stride of 3:** The window skips two pixels between adjacent outputs, further reducing the size of the output.

Intuition: When the stride is increased, adjacent windows become less similar because some of the pixels are skipped. This can be useful for downsampling the image and reducing the computational load. However, it also reduces the level of detail captured in the output.

107.5 Pooling

Pooling is a downsampling technique commonly used in Convolutional Neural Networks (CNNs) to reduce the spatial dimensions of feature maps while retaining important information.

Pooling helps make the model more robust to small changes in the input, such as translations, rotations, or noise.

Types of Pooling

Pooling typically operates over small regions (e.g., 2×2 pixels) in the input, and there are two common types of pooling:

- **Max Pooling:** In max pooling, the maximum value within each region is selected as the output.
- **Average Pooling:** In average pooling, the average of all values within the region is calculated as the output.

For example, in a 2×2 region, max pooling would select the largest value, while average pooling would compute the mean of the four pixel values.

Effect of Pooling on Backpropagation

When using max pooling, only the maximum value selected during pooling participates in backpropagation. This means that the gradient is only propagated through the location of the maximum value, while other values in the region do not receive any updates.

Robustness to Small Changes

Pooling adds robustness to the model by making it less sensitive to small translations, rotations, or noise in the input. Since only the maximum (or average) value from each region is used, slight variations in the input will not significantly change the pooled output. This helps reduce overfitting and makes the model more generalized.

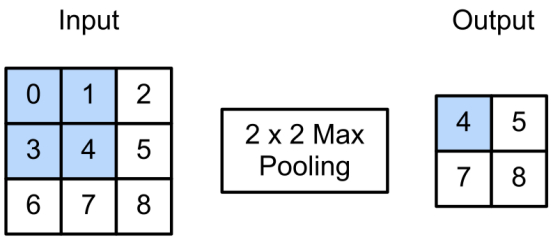


Figure 90: Enter Caption

107.6 Putting it all together - CNN architectures

Convolutional Neural Networks (CNNs) are designed to recognize patterns and structures in data by using architecture of layers of convolutions and pooling, building up a hierarchy of features.

CNN Architecture Overview

1. **Convolutions:**
    - CNNs start with convolutional layers, where small filters (kernels) are applied to the input data (e.g., image or other structured data). These filters detect local patterns like edges or textures in images.
    - Convolutions act as a way to extract features by scanning over small regions of the input (locality), enabling the network to understand how different parts of the data relate to one another.
    - Convolutions can also be applied to serial data (e.g., time-series, sequential data) when the specific position of the structure is not as important as the presence of the pattern itself.
2. **Pooling Layers:**
    - Pooling layers follow convolutions to downsample the feature maps, reducing their spatial dimensions.
    - Pooling deliberately loses information that the network doesn't need, such as small irrelevant details, while retaining important features.
    - This reduction helps make the model more efficient and less prone to overfitting by focusing on higher-level patterns rather than specific local details.
3. **Alternating Convolutions and Pooling:**
    - A typical CNN architecture alternates between convolutional and pooling layers. The idea is that as you go deeper into the network, the convolutions learn more abstract and complex features from the input data, while the pooling layers gradually reduce the data size.
    - Early layers detect simple, low-level patterns (edges, textures), and later layers combine these to form higher-level patterns (shapes, objects).

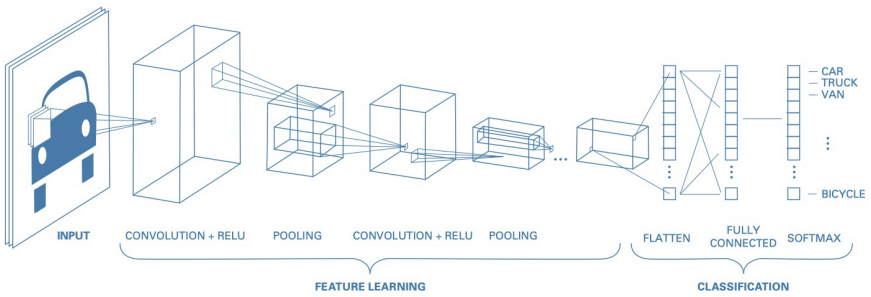


Figure 91: CNN Hierarchy Architecture

4. **Fully Connected Layers:**

- After the series of convolution and pooling layers, a CNN often ends with one or more fully connected layers (also called dense layers). These layers take the high-level features learned by the previous layers and map them to the final output, like class labels in a classification task.
- These dense layers allow the network to generalize from the local features detected earlier to make overall decisions (e.g., "Is this a cat or a dog?").

Convolutions for Different Data Types

CNN's hierarchical, feature-based approach allows CNNs to handle tasks that involve data with clear local patterns or structures, which is why they are so successful in fields like image recognition, video processing, and other domains where data can be processed similarly to images. But they can also be applied elswhere.

- **Serial Data:**
  - CNNs are not just for images. They can be applied to sequential data, like time series, text, or genomic data. In such cases, the exact position of a pattern might not be important, but the structure or presence of a pattern is still crucial. For instance, CNNs can capture local relationships in financial data or patterns in natural language sentences.
  - CNNs work well for data where **local structure** matters (e.g., in text, adjacent words might form a phrase), but they aren't ideal for tasks where **global sequence order** is critical, such as long-term dependencies in text or audio. Other models like RNNs or Transformers are used for such tasks.
- **Doesn't Work Well for Some Audio/Text:**
  - In some cases, the hierarchical nature of CNNs may not be effective for audio or text where dependencies exist over long distances. For instance, a word early in a sentence may relate to another word much later. CNNs, which emphasize local patterns, may miss these relationships. Other architectures like **RNNs**, **LSTMs**, or **Transformers** handle these types of data better because they are designed to capture longer dependencies and contextual meaning.

Hierarchical Feature Learning

- **Low-Level Features to High-Level Concepts:**
  - CNNs build up an understanding of the data through a **hierarchy of features**. Early layers learn basic structures (edges, lines), while deeper layers combine those to form more complex patterns (shapes, textures). Eventually, at the higher levels, the network can identify entire objects or concepts.
  - For example, in image recognition, the lower layers might detect edges or corners, while higher layers recognize complex structures like eyes or faces.

Local Structure and Abstraction

- **Local Structure:**
  - CNNs work well when the data has a clear, well-defined local structure that can be built up into larger patterns. This is why CNNs excel with images: small parts of an image (like edges or textures) can be combined to form larger patterns (like shapes and objects).

  - Other data types with similar local structures, like audio spectrograms or 2D genomic data, can also be processed effectively by CNNs.

- **Deliberate Information Loss:**

  - Pooling and other techniques reduce irrelevant information as you go deeper into the network. This helps the model generalize, moving from recognizing **specific details** (like the exact shape of a curve) to recognizing more **generic patterns** (like the presence of a circle), eventually outputting a classification or decision based on these generalized features.

Moving from Specific to Generic Labels

In essence, CNNs work by moving from very **specific details** (individual pixels, local features) to more **generic labels** or decisions. By focusing on what is important (through pooling and downsampling), they simplify complex data into a form that allows for robust and accurate predictions.

108 Recurrent Neural Networks (RNNs)

Big Idea: what if we maintain state inside the model? Allows varying input/output shapes!

108.1 Overview

- RNNs are designed to maintain *state* inside the model, allowing it to process sequences of data.
- At each iteration, t, we receive features X_t and a label y_t.
- The features could represent sequences such as characters (one-hot encoded), or an actual time series.
- In the model, we maintain *state* information, denoted as h_t, which evolves over time to capture sequential dependencies.

Initialize:

- Initially, the state is set to $h_0 = 0$.

Predict:

- At each time step, we make predictions using the current state h_t and features X_t.
- The model predicts the output $\hat{y}_t$ and updates the state h_{t+1} based on a function f:

$$(\hat{y}_{t+1}, h_{t+1}) = f(X_t, h_t)$$

Train:

- The model is trained by backpropagating based on the error between the predicted $\hat{y}_t$ and the true label y_t.
- During training, the function f is updated to improve both the state transitions and the immediate predictions.

108.2 Why are RNNs Useful?

- **RNNs can handle variable input sizes (= useful for text!):**
  - Unlike traditional neural networks, RNNs do not require a fixed input size.
  - Rather, they process inputs sequentially, one at a time, and maintain a hidden state, which allows them to retain relevant information from prior inputs.
- **Implicit dependence on past information:**
  - The hidden state enables RNNs to implicitly depend on all past inputs, which makes them suitable for sequential tasks such as time series, language modeling, or any problem involving temporal dependencies.
  - With sufficient state information, RNNs can theoretically capture long-term dependencies across the entire input sequence.
- **Theoretical expressiveness:**
  - In theory, with enough hidden state, RNNs are as powerful as Turing machines.
  - This means they can represent any computable function of the input sequence, making them highly expressive models.
- **Practical limitations:**
  - In practice, the limited capacity of the hidden state and the difficulty in training (e.g., vanishing and exploding gradient problems) mean that the theoretical potential of RNNs is often not fully realized.
  - As a result, training RNNs effectively over long sequences can be challenging, leading to the development of more advanced architectures like LSTMs and GRUs.

109 Attention!

= Extremely flexible representations of data

- **The building block of modern deep learning:**
  - At the core of modern deep learning models, such as transformers, is the mechanism of *attention*.
  - Attention is fundamentally a *weighted average* of values based on the relevance of different pieces of data.

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com
Noam Shazeer* Google Brain noam@google.com
Niki Parmar* Google Research nikip@google.com
Jakob Uszkoreit* Google Research usz@google.com
Llion Jones* Google Research llion@google.com
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu
Łukasz Kaiser* Google Brain lukaszkaiser@google.com
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.0 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature.

Figure 92:

109.1 Purpose

In neural networks, particularly transformers, the attention mechanism allows the model to focus on relevant parts of the input data when making predictions. The scaled dot-product attention is a specific attention mechanism commonly used in transformers.
Where

- **Query (Q)**: A set of features representing what you're looking for in the data.
- **Keys (K)**: Features in the data that are used to determine how relevant each piece of data is to the query.
- **Values (V)**: The actual data (e.g., labels, vectors) associated with the keys that we want to focus on.

The goal of scaled dot-product attention is to compute a weighted average of the values (V), where the weights are determined by how similar the query (Q) is to the keys (K). The similarity is measured by the dot product of the query and the keys.

Example: translating a sentence.

The query might represent the current word you're translating, and the keys could represent all the words in the sentence.

The attention mechanism helps the model "focus" on relevant parts of the sentence (the values) based on how related the query word is to the other words (keys).

The model then produces a translation that incorporates these important words.

109.2 Attention Definition

- Suppose we have:

  - A **query** (q), which represents a set of features we are interested in.
  - A collection of data, where each piece of data has:
    * A **key** (k), which is a set of features.
    * A **value** (v), which could be a label or the data associated with the key.
  - A measure of similarity between the **query** and the **keys**, often represented as a function $\phi(\cdot)$.

- **Attention formula:**

$$\text{Attn}(q, k_1, v_1, \ldots, k_n, v_n) = \sum_{i=1}^{n} \frac{\phi(q, k_i)}{\sum_{j=1}^{n} \phi(q, k_j)} v_i$$

1. Here, $\phi(q, k_i)$ represents the similarity measure (or kernel function) between the query q and key k_i.
2. The weights $\frac{\phi(q,k_i)}{\sum_{j=1}^{n} \phi(q,k_j)}$ are normalized to ensure they sum to 1.
3. The output is a weighted sum of the values v_i based on the similarity of their corresponding keys to the query.

- **Interpretation:**
  - Attention can be thought of as a *soft dictionary lookup*, where the query is used to retrieve relevant values from the data based on their keys.
  - It is also often described as a *kernel-weighted average*, where the kernel function $\phi(\cdot)$ determines how similar the query and the keys are.
  - The function $\phi(\cdot)$ can take various forms, but it is generally a kernel function such as dot product or Gaussian similarity.

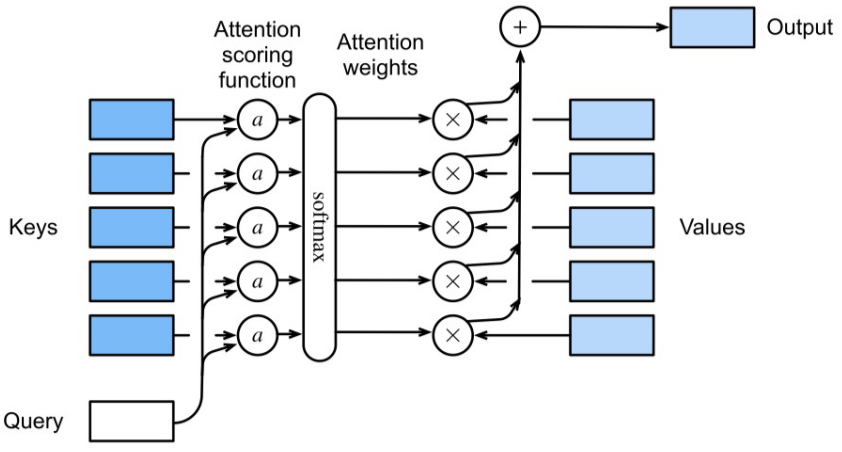


Figure 93: Enter Caption

109.3 Scaled Dot-Product Attention

- Suppose we have:
  - A **query** (q), representing a set of features.
  - A collection of data, each with:

    - **Keys** (k), another set of features.
    - **Values** (v), which could be labels or associated data.

- **Assumptions:**
    - The dimensions of the queries and keys are the same, denoted by d.
    - Both the queries and keys are assumed to follow an approximately standard normal distribution.
    - The similarity measure between the query and keys is the dot product.
- **Notation:**
    - Let the query matrix $Q \in \mathbb{R}^{m \times d}$, the key matrix $K \in \mathbb{R}^{n \times d}$, and the value matrix $V \in \mathbb{R}^{n \times p}$, where m is the number of queries, n is the number of keys (and values), and p is the dimension of the values.
- **Scaled Dot-Product Attention:**

$$\text{Attn}(q, k_1, v_1, \ldots, k_n, v_n) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)V$$

    - The dot product between the query matrix Q and the transpose of the key matrix $K^\top$ is computed, and then divided by $\sqrt{d}$ to scale the dot product.
    - The softmax function is applied to the scaled dot products to produce the attention weights.
    - The output is a weighted sum of the values V, based on these attention weights.

109.3.1 Softmax Function

For a matrix $X \in \mathbb{R}^{m \times n}$, the softmax function is applied row-wise, such that each element x_{ij} is transformed as follows:

$$\text{softmax}(X)_{ij} = \frac{\exp(x_{ij})}{\sum_{k=1}^{n} \exp(x_{ik})}$$

This ensures that the elements in each row of the matrix sum to 1, making them suitable for use as attention weights.
For $X \in \mathbb{R}^{m \times n}$:

$$\text{softmax}(X) = \begin{bmatrix} \frac{\exp(x_{11})}{\sum_{k=1}^{n} \exp(x_{1k})} & \cdots & \frac{\exp(x_{1n})}{\sum_{k=1}^{n} \exp(x_{1k})} \\ \vdots & \ddots & \vdots \\ \frac{\exp(x_{m1})}{\sum_{k=1}^{n} \exp(x_{mk})} & \cdots & \frac{\exp(x_{mn})}{\sum_{k=1}^{n} \exp(x_{mk})} \end{bmatrix}$$

Steps:
Dot-Product Similarity: The similarity between a query (Q) and keys (K) is measured by their dot product. If $Q \in \mathbb{R}^{m \times d}$ (queries) and $K \in \mathbb{R}^{n \times d}$ (keys), the dot product is calculated as:

$$QK^\top$$

Each element is exponentiated and normalized by the sum of the exponentiated values in its row.

This gives an $m \times n$ matrix, where each element represents the similarity between a query and a key.
Scaling by $\sqrt{d}$: Since the magnitude of the dot product can grow large with increasing dimensions d (the number of features), the dot products are scaled by dividing them by $\sqrt{d}$ to stabilize the values. This scaling helps prevent large gradients during training:

$$\frac{QK^\top}{\sqrt{d}}$$

Softmax: The softmax function is then applied to the scaled dot product. This ensures that the result is a set of attention weights that sum to 1, making it a proper probability distribution. The more similar the query is to a particular key, the higher the attention weight for that key:

$$\text{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)$$

Weighted Sum of Values: Finally, the attention weights are used to compute a weighted sum of the values ($V \in \mathbb{R}^{n \times p}$). This gives the final attention output, which is a matrix of size $m \times p$, where each row corresponds to the query's weighted attention on the values:

$$\text{Attn}(q, k_1, v_1, \ldots, k_n, v_n) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)V$$

Why Use Scaled Dot-Product Attention?

- **Efficiency**: The dot-product similarity can be computed efficiently, even for large datasets.
- **Handling Long Sequences**: It allows models to attend to important elements in sequences of varying lengths.
- **Preventing Exploding Gradients**: The scaling by $\sqrt{d}$ prevents the values from becoming too large as the dimensionality increases.

Use of Softmax:

The softmax function transforms the dot-product results into probabilities (or attention weights). It ensures that:

- The most similar keys (to the query) get higher weights.
- All the weights across the keys sum up to 1, so it's a valid weighted average.

109.4 Self-Attention:

Allows each position in the input to attend to every other position, enabling the model to capture relationships between elements within the input itself, rather than relying solely on external context.

Given an input matrix $X \in \mathbb{R}^{n \times d}$ (where n is the number of elements in the input sequence and d is the dimension of each element), the self-attention mechanism computes attention as follows:
Given an input matrix $X \in \mathbb{R}^{n \times d}$ (where n is the number of elements in the input sequence and d is the dimension of each element), the self-attention mechanism computes attention as follows:

1. **Query, Key, and Value Projections:**
   - From the input X, we compute three matrices:
     - **Query matrix** Q (represents what we're looking for):
       $$Q = XW_Q$$
       where $W_Q \in \mathbb{R}^{d \times d_Q}$ is a weight matrix that projects the input to the query space.
     - **Key matrix** K (represents the features in the data):
       $$K = XW_K$$
       where $W_K \in \mathbb{R}^{d \times d_K}$ projects the input to the key space.
     - **Value matrix** V (the actual data):
       $$V = XW_V$$
       where $W_V \in \mathbb{R}^{d \times d_V}$ projects the input to the value space.
2. **Dot-Product Similarity:**
   - We compute the similarity between the queries and keys using a dot product. The resulting matrix represents how much each element of the sequence should attend to every other element:
     $$QK^\top$$
     This gives an $n \times n$ matrix, where each element indicates the similarity between a query and a key.
3. **Scaling:**
   - The dot-product result is scaled by $\frac{1}{\sqrt{d}}$ (where d is the dimension of the keys) to prevent excessively large values that can make optimization difficult:
     $$\frac{QK^\top}{\sqrt{d}}$$
     This scaling ensures that the dot products remain within a stable range, particularly when d is large.
4. **Softmax:**
   - A softmax function is applied to the scaled dot product to convert it into a probability distribution, where higher similarity values result in larger attention weights:
     $$\text{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)$$
     This matrix represents how much attention each element in the sequence should give to every other element.

5. **Weighted Sum of Values:**

   - Finally, the attention weights are used to compute a weighted sum of the values:

$$\begin{aligned}\text{SelfAttn}(X) &= \text{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)V \\ &= \text{softmax}\left(\frac{xW_qW_k^\top X^\top}{\sqrt{d}}\right)XW_v\end{aligned}$$

   This yields the final attention output, where each position in the sequence is represented as a weighted combination of the values.

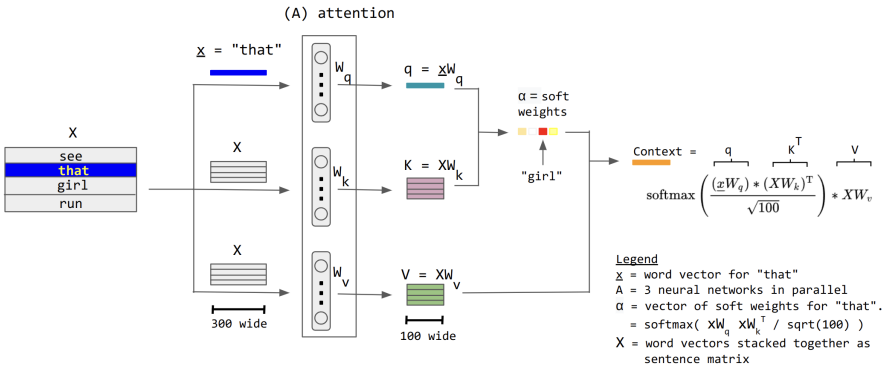


Figure 94: Enter Caption

Intuition

Self-attention allows the model to "focus" on different parts of the input sequence by assigning different attention weights.

For example, in a sentence, a word can attend to other relevant words, allowing the model to capture dependencies between different parts of the sentence.

Why Self-Attention?

- **Global Context:** Self-attention allows each element of the input to consider the entire sequence when making predictions, rather than just local context.
- **Flexible Representation:** It works well with sequences of variable length and is able to handle long-range dependencies effectively.
- **Parallelization:** Unlike recurrent networks, self-attention allows for efficient parallel computation over the entire sequence.

109.5 Why does this make sense?

Plugging in this measure of attention essentially lets us simultansouly train:

1. Feature representation of inputs (Q)
2. Weights (K)

3. Feature representation of “labels” (V)

= A kernel regression where you learn the representations, the kernel weighting and the representation of the ”label”.

Extremely over-parameterized!

ML Lecture Notes: Wk 12 — Unsupervised Learning

110 Overview

Core concepts of unsupervised learning

- PCA
- Kernel PCA
- Autoencoders

111 PCA: Dimension Reduction

$$\hat{\theta} = \arg\min_{\theta} \mathcal{L}(\theta)$$

1. **Point Estimate ($\hat{\theta}$):** (a matrix!)

   This represents the parameters or the latent variables that we're trying to estimate. In the context of dimension reduction, θ could be the low-dimensional representation of your data.

2. **Loss Function ($\mathcal{L}(\theta)$):**

   The loss function measures the error or the difference between the original high-dimensional data and the reconstructed data from the lower-dimensional representation. The goal is to minimize this loss to ensure that the low-dimensional representation captures as much information as possible from the original data.

3. **Optimization Objective ($\min_\theta$):**

   The objective is to find the point estimate (θ) that minimizes the loss function $\mathcal{L}(\theta)$. This process is typically done through optimization techniques, such as gradient descent, where you iteratively adjust θ to reduce the loss.

4. **A Matrix:**

   In the context of dimension reduction techniques like Principal Component Analysis (PCA), the data is often represented as a matrix, where rows correspond to samples and columns correspond to features. The goal is to find a low-dimensional representation (also in matrix form) that approximates the original data matrix.

5. **Optimization Expression ($\arg\min_\theta \mathcal{L}(\theta)$):**

   The term "argmin" denotes the value of θ that minimizes the loss function $\mathcal{L}(\theta)$. Essentially, it is the solution to the optimization problem.

111.1 Example in PCA

- the θ might represent the principal components (the directions in which the data varies the most).
- The loss function $\mathcal{L}(\theta)$ could be the sum of the squared differences between the original data and its projection onto the principal components.
- The optimization problem is to find the principal components that minimize this loss, effectively reducing the dimensionality of the data while preserving as much variance as possible.

111.2 PCA = supervised learning in a trenchcoat!

While the approach is technically unsupervised (since there's no labelled output), the method resembles supervised learning. The optimization problem is formulated similarly to supervised learning tasks, where the objective is to minimize a loss function, but instead of predicting labels, the goal is to reconstruct the input data.

- **Central Learning Objective:**

$$\hat{\theta} = \arg\min_{\theta} \mathcal{L}(\theta)$$

...becomes...

- **Optimization:**

$$\hat{\theta} = \arg\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} \|x_i - \text{decode}(\text{encode}(x_i, \theta), \theta)\|^2$$

Here, the process involves

  - encoding the original data x_i into a lower-dimensional space using the function encode,
  - and then decoding it back to the original space using decode.
  - The loss function measures the reconstruction error, and minimizing this error helps in finding the best low-dimensional representation.

= effectively an "auto-supervised" learning set up.

111.3 Low-Dimensional Representation

111.3.1 PCA Goal:

represent each vector $x_i \in \mathbb{R}^D$ using a low-dimensional representation.

Formally: the original vector x_i can be approximated as:

$$x_i \approx \sum_{l=1}^{L} z_{il} w_l$$

where:

- x_i is the original dimension vector
- $z_i \in \mathbb{R}^L$ is the **latent representation** (the coordinates of x_i in the reduced k-dimensional space).
- $w_l \in \mathbb{R}^D$ are the **weights over each dimension** (the principal components, which are the directions in the original space that capture the most variance).

111.3.2 Error Measurement:

The error of this approximation is measured by the following loss function:

$$\begin{aligned}\mathcal{L}(W, Z) &= \frac{1}{n} \left\| X - W Z^\top \right\|^2 \\ &= \frac{1}{n} \sum_{i=1}^{n} \|x_i - W z_i\| 2\end{aligned}$$

Where,

- X represents the original data matrix,
- W is the matrix of principal components, and
- Z is the matrix of latent representations.

111.3.3 Intuition:

We want to capture as much variance in X as possible with each dimension of our embedding (reduced representation).*

This involves finding a new set of axes (principal components) that best represent the directions of maximum variance in the data.

The principal components can be thought of as the new basis vectors in this reduced space, and the coordinates (embeddings) of the data in this new space are the projections onto these basis vectors.

> *An embedding in the context of machine learning, particularly in dimensionality reduction, refers to a mapping of high-dimensional data into a lower-dimensional space.
>
> Embeddings are used across various domains, such as natural language processing (word embeddings like Word2Vec), graph embeddings, and more, to reduce the dimensionality while preserving meaningful relationships between the data points.

SKIPPED SLIDES 11 - 15, FURTHER PCA DETAIL

112 Embeddings

112.1 Stochastic Neighbor Embeddings (SNE):

- **Idea:**

  The goal is to visualize high-dimensional data in a low-dimensional space. The challenge is to preserve some of the structure that exists in the high-dimensional space when mapping it to a lower dimension.

- **Approach:**

  1. **Convert High-Dimensional Distances into Conditional Probabilities:**
     The high-dimensional distance between points x_i and x_j is converted into a conditional probability:
     $$p_{j|i} = \frac{\exp\left(-\frac{1}{2\sigma_i^2}\|x_i - x_j\|^2\right)}{\sum_{k\neq i}\exp\left(-\frac{1}{2\sigma_i^2}\|x_i - x_k\|^2\right)}$$
     where $p_{j|i}$ represents the probability that point x_j would be selected as a neighbor of point x_i, given the distances in the high-dimensional space.
  2. **Construct a Low-Dimensional Approximation Z:**
     Similarly, in the low-dimensional space, we calculate:
     $$q_{j|i} = \frac{\exp\left(-\frac{1}{2\sigma_i^2}\|z_i - z_j\|^2\right)}{\sum_{k\neq i}\exp\left(-\frac{1}{2\sigma_i^2}\|z_i - z_k\|^2\right)}$$
     where $q_{j|i}$ represents the analogous probability in the low-dimensional space.
  3. **Optimization:**
     The goal is to find the low-dimensional representations Z that minimize the difference between the high-dimensional probabilities P and the low-dimensional probabilities Q. This is done by minimizing the Kullback-Leibler divergence (a measure of how one probability distribution diverges from a second, expected probability distribution):
     $$\begin{aligned}\mathcal{L} &= \sum_i D_{KL}(P_i\|Q_i) \\ &= \sum_i \sum_j p_{j|i}\log\frac{p_{j|i}}{q_{j|i}}\end{aligned}$$

112.2 t-distributed Stochastic Neighbor Embeddings (t-SNE):

Issues with Basic SNE:

- **Pulling Distant Points Together:**

  The basic SNE algorithm tends to pull distant points closer together rather than pushing nearby points apart. This behavior arises partly due to the use of the normal (Gaussian) distribution for measuring distances.

- **Normal Distribution:**

  The normal distribution has a light tail, which means that it doesn't strongly penalize distant points being placed close together in the low-dimensional space.

Idea Behind t-SNE:
To address the issues with basic SNE, t-SNE introduces the idea of using a *heavier-tailed distribution*, specifically the Student's t-distribution. This distribution allows distant points to exert less influence, making it easier to separate clusters in the low-dimensional space.
t-SNE Probability Calculation:
The probability in the low-dimensional space is calculated using the Student's t-distribution:

$$q_{j|i} = \frac{\left(1 + \|z_i - z_j\|^2\right)^{-1}}{\sum_{k \neq i} \left(1 + \|z_i - z_k\|^2\right)^{-1}}$$

This change helps to better separate clusters and spread out points that are distant in the high-dimensional space.
KL Divergence Objective:
Despite the change in distribution, t-SNE uses the same Kullback-Leibler (KL) divergence objective function as SNE:

$$\begin{aligned}\mathcal{L} &= \sum_i D_{KL}(P_i \| Q_i) \\ &= \sum_i \sum_j p_{j|i} \log \frac{p_{j|i}}{q_{j|i}}\end{aligned}$$

The optimization seeks to minimize this divergence, ensuring that the low-dimensional representation captures the structure of the data as effectively as possible.

112.3 t-SNE to UMAP (Uniform Manifold Approximation and Projection):

112.3.1 Limitations of t-SNE = Slow Computation

- t-SNE is computationally intensive, with a complexity of $O(N^2)$ or more. This makes it impractical for very large datasets.
- **High-Dimensional Gravitational Problem:** The t-SNE algorithm can be thought of as solving a high-dimensional gravitational problem, where every point exerts an attractive or repulsive force on every other point. This complexity further contributes to its inefficiency on large datasets.

To address these limitations, UMAP (Uniform Manifold Approximation and Projection) is introduced as a more efficient alternative to t-SNE.

112.3.2 UMAP

- **k-Nearest Neighbor Graph:**

  UMAP begins by constructing a k-nearest neighbor graph. This graph represents the local relationships between points in the high-dimensional space.

- **Preserving Distances:**

  The algorithm then seeks to find a low-dimensional representation that preserves the distances over this graph as much as possible. This approach allows UMAP to maintain both global and local structure in the data, and it does so with much greater efficiency compared to t-SNE.

112.4 Visual Intuition

UMAP starts with a k-nearest neighbor graph to capture local relationships and then seeks to maintain these relationships in the low-dimensional space, ensuring that the manifold's geometry is preserved.

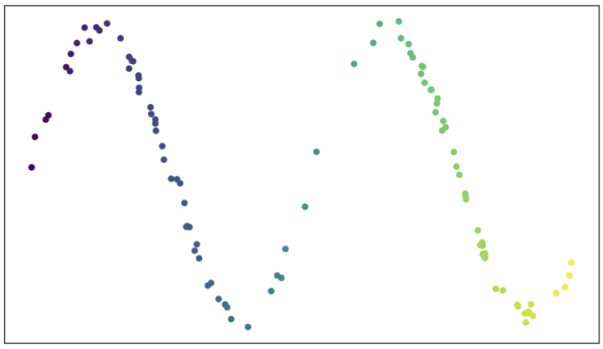


Figure 96: shows the original high-dimensional data projected onto a 2D plane, where different colors represent different clusters or structures within the data.

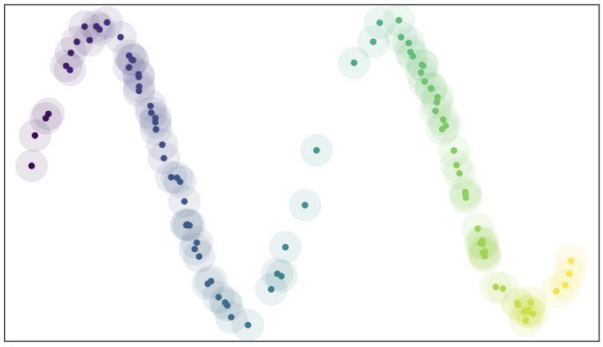


Figure 97: show how UMAP begins by constructing a k-nearest neighbor graph, where each point is connected to its closest neighbors. The shaded circles represent local neighborhoods.

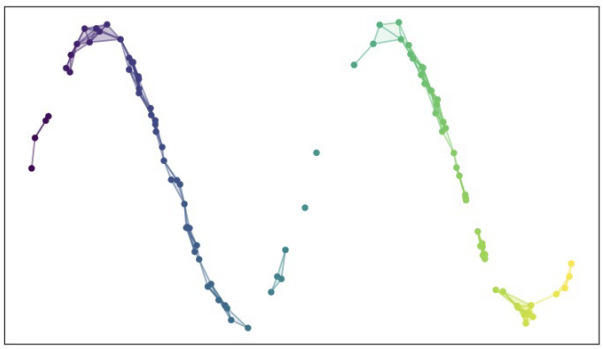


Figure 98: The connected lines between points represent how UMAP preserves the structure of the data as it reduces its dimensionality, ensuring that points that are close in high-dimensional space remain close in the lower-dimensional projection.

112.4.1 Locally Varying Geometry

data can exhibit different local structures in high-dimensional space. When using dimensionality reduction techniques like UMAP, it's crucial to maintain these local structures in the lower-

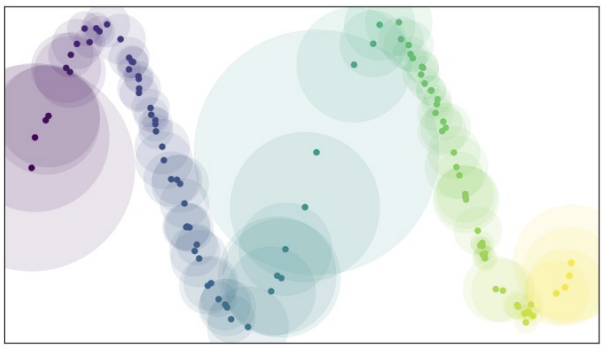


Figure 99: the shaded regions (possibly varying in intensity) show the influence or neighborhood of each point. The varying size and spread of these shaded areas suggest that UMAP can adapt to different local densities in the data, preserving the structure of both dense and sparse regions. (?)

dimensional projection.
UMAP handles varying local geometries effectively, preserving the structure of both dense and sparse regions.

tSNE / UMAP Demo https://pair-code.github.io/understanding-umap/

113 Auto-encoders

The basic idea behind autoencoders is to **predict the input features using the input features themselves**. This is achieved by compressing the input into a lower-dimensional representation (encoding) and then reconstructing it back to the original dimensions (decoding).

Simplest Version: The simplest form of an autoencoder consists of just one hidden layer between the input and output layers. This hidden layer acts as a "bottleneck," forcing the model to learn a compact, lower-dimensional representation of the input data.

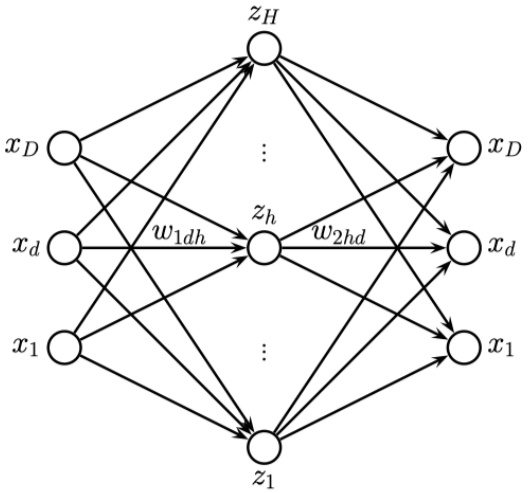


Figure 100: An Autoencoder

"Bottleneck" Autoencoder:

- **Hidden Nodes Less Than D:**

  If the number of hidden nodes (i.e., the dimensionality of the bottleneck layer) is less than the input dimension D, the autoencoder will learn a low-rank representation of the data. This compressed representation captures the most important features of the input.

- **Hidden Nodes More Than D:**

  If the number of hidden nodes is greater than the input dimension D, the autoencoder has the capacity to learn trivial representations (like simply copying the input). To prevent this, additional restrictions or regularization techniques must be applied to ensure meaningful learning.

113.1 Equivalence to PCA:

An autoencoder can be equivalent to Principal Component Analysis (PCA) under the following conditions:

- The autoencoder has only one hidden layer.
- There are no nonlinear activation functions (i.e., the network is linear).
- The loss function used is the mean-squared error.

In this scenario, the autoencoder learns to project the data onto a linear subspace that captures the maximum variance, just as PCA does.

113.2 Using Neural Networks for Autoencoders:

- **Central Learning Objective:**

$$\hat{\theta} = \arg\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} \|x_i - \text{decode}(\text{encode}(x_i, \theta), \theta)\|^2$$

  The primary goal when using neural networks for autoencoders is to learn an efficient representation (or encoding) of the input data, such that this representation can be used to accurately reconstruct the original data.

- **Encoder Network:**

  The encoder network is the part of the autoencoder that compresses the input data x_i into a lower-dimensional latent representation z_i. This network consists of several layers of neurons that progressively reduce the dimensionality of the input.

- **Decoder Network:**

  The decoder network is responsible for reconstructing the original data from the compressed latent representation. It essentially reverses the encoding process, expanding the low-dimensional representation back to the original data dimensions.

- **Backpropagation:**

  The error (or loss) between the original input and the reconstructed output is computed, and this error is backpropagated through both the encoder and decoder networks. By adjusting the network parameters during training, the model learns to minimize this reconstruction error.

- **Optimization Objective:**

  The optimization objective can be expressed as:

$$\hat{\theta} = \arg\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} \|x_i - \text{decode}(\text{encode}(x_i, \theta), \theta)\|^2$$

  where:

  - x_i is the original input data.
  - $\text{encode}(x_i, \theta)$ is the function representing the encoder network.
  - $\text{decode}(\text{encode}(x_i, \theta), \theta)$ is the function representing the decoder network that reconstructs the input.
  - θ represents the parameters of both the encoder and decoder networks.

  The goal is to find the parameters θ that minimize the average reconstruction error over all training examples.

113.3 Other Types of Autoencoders:

- **Denoising Autoencoders:**
  - The key idea behind denoising autoencoders is to improve the robustness of the learned representations by introducing noise into the input data.
  - During training, random noise is added to the input data, and the network is trained to predict the original, un-noised version of the input.
  - The objective is to learn a representation that is resilient to small perturbations in the input, which can improve the model's generalization ability.
- **Sparse Autoencoders:**
  - Sparse autoencoders introduce a sparsity constraint on the activations of the hidden layers. This means that only a small number of neurons are active (i.e., have non-zero outputs) at any given time.
  - The sparsity constraint is typically enforced using a regularization term, such as the L1 penalty, similar to Lasso regression.
  - This sparsity encourages the network to learn more efficient and interpretable representations of the data, as only the most relevant features are activated.
- **Variational Autoencoders (VAEs):**
  - Variational autoencoders introduce a probabilistic approach to the bottleneck layer, where the latent representation is modeled as a distribution rather than a single point.
  - Instead of encoding the input into a fixed vector, the encoder produces parameters (mean and variance) of a Gaussian distribution from which the latent vector is sampled.
  - The decoder then reconstructs the input from this sampled latent vector.
  - This probabilistic nature allows VAEs to generate new data samples by sampling from the latent space, making them useful for tasks such as data generation and anomaly detection.