Machine Learning

February 5, 2023

for whatever reason, we just ignore that pdf's are density functions.

Terms/data

Options on how to encode categorical data

- number them $0..C$
- one-hot encoding - have C inputs, where exactly 1 is 1, all others 0
- encode in binary

Considerations:

- encoding classes as numbers/in binary gives a structure to the classes that may not be there, and thus may confuse the model.
- one-hot is ideal in this sense, but it increase the size of the model a lot if there are many classes.
- if the model is sufficiently powerful, even with decimal/binary, it can learn past this issue.

Important functions

Softmax: $\mathrm{softmax}(\boldsymbol{x}) = \dfrac{\exp(\boldsymbol{x})}{\|\exp(\boldsymbol{x})\|_1}$

Sigmoid: $\mathrm{sigmoid}(t) := \dfrac{1}{1+e^{-t}}$ maps $\mathbb{R}$ to $[0,1]$

Alternatives: $\mathrm{sigmoid}'(t) := \dfrac{2}{1+e^{-t}} - 1$ maps $\mathbb{R}$ to $[-1,1]$

1 Linear Regression

input: a vector $\boldsymbol{x} \in \mathbb{R}^D$, where D is the data dimension

output: $y \in \mathbb{R}$

model params: $\boldsymbol{w} \in \mathbb{R}^D$

training data: N observations, $\langle(\boldsymbol{x}_i, y_i)\rangle_{i=1}^N$

linear model: $y = w_0 + x_1 w_1 + \cdots + x_d w_d + \varepsilon$ - note w_0 is called the bias/intercept, and ε the noise/uncertainty

easier model: introduce $x_0 = 1$ for all data points, so that $y = \boldsymbol{w} \cdot \boldsymbol{x} + \varepsilon$ - now the data is $D + 1$ dimensional.

prediction: $\widehat{y}(x) = \boldsymbol{w} \cdot \boldsymbol{x}$ - no error term here

loss function: varies greatly, generally called $\mathcal{L}$

- e.g. absolute loss: $|\widehat{y} - y|$, or squared loss: $(\widehat{y} - y)^2$

residual: the difference between prediction and actual data

In matrix notation:

The data is represented as a $N{\times}(D{+}1)$ matrix $\boldsymbol{X}$ (where N is the no. of datapoints), with each row being one of the $\boldsymbol{x}$ vectors, including $\boldsymbol{x}_0$ at the start.

The known output is $\boldsymbol{y}$, a $N \times 1$ matrix, and similarly $\hat{\boldsymbol{y}}$ is the predicted output.

$\boldsymbol{w}$ is the same as above.

Then $\hat{\boldsymbol{y}} = \boldsymbol{X}\boldsymbol{w}$.

General process to solve for parameters

Given a loss function, how to solve for the params:

- find the partial derivatives wrt to each param
- solve the system of equations when you set each partial derivative to 0
- this can turn out to be an expression in the covariance, variance and means of x and y

Least squares regression

Loss function:

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{2N}\sum_{i=1}^{N}(\boldsymbol{x}_i^T\boldsymbol{w}-y_i)^2 = \frac{1}{2N}(\boldsymbol{X}\boldsymbol{w}-\boldsymbol{y})^T(\boldsymbol{X}\boldsymbol{w}-\boldsymbol{y}) = \frac{1}{2N}(\boldsymbol{w}^T(\boldsymbol{X}^T\boldsymbol{X})\boldsymbol{w}-2\cdot\boldsymbol{y}^T\boldsymbol{X}\boldsymbol{w}+\boldsymbol{y}^T\boldsymbol{y})$$

(note the 2 there is to simplify the derivative - has no effect on the solutions)

Which we can then differentiate, using techniques from CM:

(note $\boldsymbol{X}^T\boldsymbol{X}$ is symmetric)

$$\frac{d\mathcal{L}}{d\boldsymbol{w}}(\boldsymbol{w}) = \frac{1}{2N}\left(2\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{w} - 2\left(\boldsymbol{y}^T\boldsymbol{X}\right)^T\right) = \boldsymbol{X}^T\boldsymbol{X}\boldsymbol{w} - \boldsymbol{X}^T\boldsymbol{y}$$

And solve by setting $= \mathbf{0}$: (assuming the inverse exists)

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

Thus our predictions are

$$\widehat{\boldsymbol{y}} = \boldsymbol{X}\boldsymbol{w} = \boldsymbol{X}(\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$$

note that $\boldsymbol{X}(\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T$ is often called the *hat matrix.*

Computational complexity: $O(D^2N)$ is a good bound, assuming $D < N$ (nb this is because the cost $O(D^3)$ of inverting $\boldsymbol{X}^T\boldsymbol{X}$ is(O) less than the $O(D^2N)$ for doing the matrix multiplications). (also, swap numbers if $N < D$)

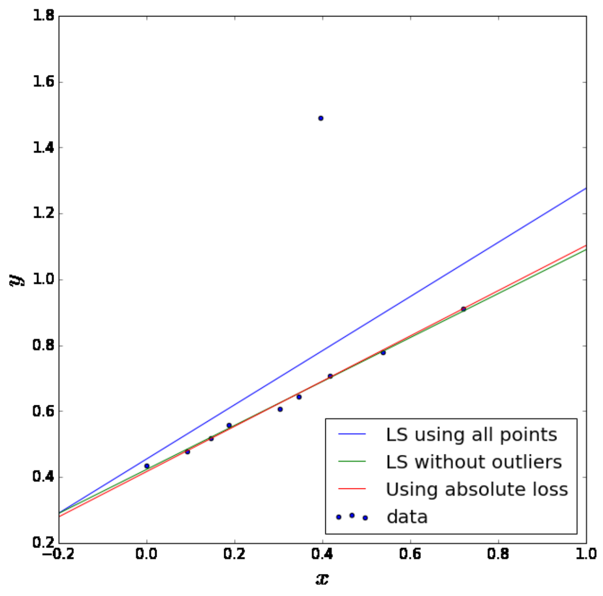
When is $\boldsymbol{X}^T\boldsymbol{X}$ invertible?

$\mathrm{rank}(\boldsymbol{X}^T\boldsymbol{X}) = \mathrm{rank}(\boldsymbol{X}) \le \min\{D+1, N\}$, so it is only invertible if $\mathrm{rank}(\boldsymbol{X}) = D+1$

Note that if we use *one-hot* encoding, we are introducing dependencies in the columns of X (as the sum of those columns is always $= 1$) and thus reducing the rank, making $\boldsymbol{X}^T\boldsymbol{X}$ non-invertible. We can solve this by dropping 1 class (e.g. sunday out of the days of the week) from our data, since we can recreate it easily.

Issues:

outliers have a large effect when using least square loss - to fix, either exclude outliers (somehow), or use absolute loss (not differentiable, so can't do the process above):



1.1 Perceptrons

A perceptron is a linear model as desc. above composed with the sign operation.

2 Maximum likelihood

$\mathrm{Cov}(X, Y) = \mathbb{E}[XY] - \mathbb{E}[X]\mathbb{E}[Y]$
$\mathrm{corr}(X, Y) = \frac{\mathrm{Cov}(X,Y)}{\sqrt{\mathrm{var}(X)\mathrm{var}(Y)}}$

The covariance of a vector $\boldsymbol{x}$ is the matrix $\mathrm{Cov}(\boldsymbol{x})$, which has i, j'th entry $\begin{cases} \mathrm{var}(x_i) & i = j \\ \mathrm{Cov}(x_i, x_j) & i \neq j \end{cases}$

Gaussian & multivariate

Laplace distribution:

$$\mathrm{Lap}(x|\mu, b) = \frac{1}{2b}\exp(-\frac{|x-\mu|}{b})$$

Mean: μ, variance: $2b^2$

2.1 Maximum likelihood principle

Suppose we have a set of data, and have chosen a model with parameters θ.

We make the universal assumption that our data points (observations) are independent. Let $p = p(\mathcal{D}|\theta)$ be the probability of observing all the data given θ under this model , and $p_i = p(\mathcal{D}_i|\theta)$ for observing the ith datapoint. THen by independence, $p = \prod_i p_i$, and so we introduce the negative log likelihood,

$NLL = -\log p = -\sum \log p_i$

which is much easier to work with.

The aim is to choose the parameters θ such that p is maximised, which is equivalent to minimising the NLL.

Then, once we've found θ^*, we just run new data through the model.

2.1.1 MLE for Linear Regression

If we have the same setup as LR, and model the noise as $\mathcal{N}(0, \sigma^2)$, then $y \sim \boldsymbol{w}^T\boldsymbol{x} + \mathcal{N}(0, \sigma^2) \sim \mathcal{N}(\boldsymbol{w}^T\boldsymbol{x}, \sigma^2)$, so $p_i = p(y_i|\boldsymbol{x}_i, \boldsymbol{w}, \sigma)$, and we use the pdf of the Gaussian

and thus the probability that we observe the labelled data $(\boldsymbol{x}_1, y_1), ..., (\boldsymbol{x}_N, y_N)$ given distribution p and parameters θ is

$$p(x_1, ..., x_N|\theta) = \prod_{i=1}^{N} p(x_i|\theta)$$

Then the NLL comes out to be the same as $\mathcal{L}$ above, except a constant and a multiplicative factor, so we know our estimates above are MLEs.

2.2 MLE for Laplace LR

We apply the MLE process to LR, but with the absolute loss, which turns out to be 2.1.1 again, but with the Laplace distribution instead of the Gaussian. There is no closed-form solution for the parameter μ.

3 Basis expansion

with polynomials

A d-dimension basis expansion is the map $\phi_d : \mathbb{R} \to \mathbb{R}^{d+1}$ given by $\phi_d(x) = \begin{bmatrix} 1 & x & ... & x^d \end{bmatrix}^T$.

We extend this to $\phi_d : \mathbb{R}^D \to \mathbb{R}^{\cdots}$ where $\phi_d(\boldsymbol{x})$ is a list of the terms in $(\boldsymbol{x}_1 + \cdots + x_D)^d$ (so a long vector - note we ignore the multiplicative factors)

If we "preprocess" our data by converting $\langle(\boldsymbol{x}_i, y_i)\rangle_{i=1}^N$ to $\langle(\phi_d(\boldsymbol{x}_i), y_i)\rangle_{i=1}^N$, then with a linear model such as LR, we can actually model polynomial relationships.

This is powerful, but it can lead to overfitting, because a $d = N - 1$ expansion will ensure there is a poly that perfectly fits our data, but is most likely useless. Also, it geenrates a lot more parameters, which can lead to very slow learning.

complexity: e.g. comparing degree 2 basis expansion to a kernel like $\kappa(\boldsymbol{x}, \boldsymbol{y}) = (1 + x^\perp y)^2$:

- basis expansion: D^2 time
- kernel: $\approx D$ time, as scalar mult is basically free.

with kernels

A kernel is a function $\kappa(x, x') \in \mathbb{R}$ that maps two data points to a real number. it is typically symmetric and non-negative, so we can consider it a "distance" or measure of similarity between the two points.

A radial basis function kernel with parameter γ is $\kappa(\boldsymbol{x}, \boldsymbol{x}') = \exp(-\gamma\|\boldsymbol{x} - \boldsymbol{x}'\|^2)$

(separately) a radial basis function is a a function $\phi(\boldsymbol{x}) = \exp(-\gamma\|\boldsymbol{x} - \mu\|^2)$ for some centre μ. This is a completely valid expansion in of itself

Then the basis expansion wrt to κ is $\phi_\kappa(\boldsymbol{x}) = \begin{bmatrix} 1 & \kappa(\boldsymbol{\mu_1}, \boldsymbol{x}) & \dots & \kappa(\boldsymbol{\mu_M}, \boldsymbol{x}) \end{bmatrix}^T$ for M centres $\boldsymbol{\mu}_1, \dots, \boldsymbol{\mu}_M$, which is an approximation, as the true expansion is infinite.

Choosing centres and γ

Centres: can choose the data points themselves, not really on the course

γ:

small then we probably get underfitting, as all the points have roughly $\kappa(\boldsymbol{\mu}, \boldsymbol{x}) \approx 1$ and so get similar feature values, so all data is very similar and thus we get underfitting

large, $\kappa(\boldsymbol{\mu}, \boldsymbol{x}) \approx 0$ for most points, unless they are v. close to $\boldsymbol{\mu}$, and this likely leads to overfitting: for existing points, they are differentiated by the kernel, but new points. , as the **kernel measures closeness???**

4 Learning curves and over/underfitting

A model is unbiased if $\mathbb{E}[\hat{y} - y] = 0$ for any new data points - the expected difference between actual and predicted is 0. It is otherwise biased.

A model is **overfitting = high variance** when it corresponds too closely to previously seen data, and thus doesn't make future predictions so well - precisely, when we fit it to different data set, we get wildly different models.

A model is **underfitting = high bias** when it does not capture the actual structure of the data, so predictions will be inherently wrong - i.e. $\mathbb{E}[\hat{y} - y] \neq 0$

Seeing this on a learning curve:

- two curves: one <u>training error</u>, one <u>validation error</u>, on a error against # training samples graph
- overfitting:

  - training error continues to decrease - i.e. matching training data even better, but validation error may increase
- underfitting:
  - neither error decreases fully - they may still decrease and approach each other, but the floor will be much higher
- good???
  - both converge, and at a low floor reflecting the natural noise in the data.

5 Regularisation & Model Selection

The idea: to avoid overfitting, we penalise weights being large, so the model only selects weights if they're actually helping the prediction.

5.1 Ridge regression

$$\mathcal{L}_{ridge}(\boldsymbol{w}) = (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^T(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) + \lambda \sum_{i=1}^{D} w_i^2 = ||\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}||_2^2 + \lambda ||\boldsymbol{w}||_2^2$$

Where in the norm of $\boldsymbol{w}$ we ignore $\boldsymbol{w}_0$. k

Standardisation, as desc. below, is important for ridge regression, as it ensures the weights are treated equally.

The optimal weights are $\boldsymbol{w}_{ridge} = (\boldsymbol{X}^T\boldsymbol{X}+\lambda\boldsymbol{I}_D)^{-1}\boldsymbol{X}^T y$, via the standard differentiation method.

Same runtime as Least squares, $O(D^2N)$

RR as an LP

We can also consider it as the following LP

$$\begin{aligned} \text{minimise: } & ||\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}||^2 \\ \text{st} & ||\boldsymbol{w}||^2 \leq R \end{aligned}$$

Where R is chosen somehow.

5.2 Lasso

$$\mathcal{L}_{ridge}(\boldsymbol{w}) = (\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})^T(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}) + \lambda \sum_{i=1}^{D} w_i = \|\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}\|_2^2 + \lambda\|\boldsymbol{w}\|$$

Where in the 1-norm of $\boldsymbol{w}$ we ignore $\boldsymbol{w}_0$.

Standardisation, as desc. below, is important for ridge regression, as it ensures the weights are treated equally. There is no closed form, so considering the LP again makes sense:

$$\begin{aligned} \text{minimise: } & \|\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}\|^2 \\ \text{st} & \|\boldsymbol{w}\|_1 \leq R \end{aligned}$$

Where R is chosen somehow.

$\|\boldsymbol{w}\|_1$ is not differentiable, so contour curves of it have corners.

5.3 Standardisation

For each feature (separately), we can standardise the training data to have mean 0 and variance 1. Obviously, we also apply the same transformation to the testing data.

Note we can also standardise the output variables.

MORE!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

6 Model selection

A **hyperparameter** is a parameter of the model that is not trained.

The validation set is a subset of the training set that is used separately for assessing hyperparameter performance. To test some hyperparameters, we train the model on the training set less the validation set - the new training set, and then run the model on the validation set to get an accuracy figure. we then chooose the best hyperparameters according to whichever did best on the validation set.

K-fold cross validation is where we divide the training set into K parts = folds, and use $K - 1$ of them as the training set, and the last as the validation set. We repeat this K times, using each fold as the validation set once, and average over all. Thus, we arrive at an accuracy/error for the given hyperparameters.

LOOCV is N-fold CV, where N is the number of data points.

feature selection: when we have lots of features, it can be worth working out which are useful before training (even with lasso/ridge), to make our model smaller. Two approaches:

forward search:

we build up a set F of selected features, initially $\emptyset$.

While it is not full, for each currently unselected feature i , train & test the model on features $F \cup \{i\}$. Set $F := F \cup \{i\}$ for whichever i gave the best results.

Out of all the F's generate, return whichever gave the best results.

filter feature selection:

calculate the mutual informsation(below) between each feature and the output, and choose the best according to this.

$I(X,Y) = \sum_x \sum_y p(X=x, Y=y) \log \frac{p(X=x,Y=y)}{p(X=x)p(Y=y)}$

7 Bayesian approach

We consider everything as probability, and use Bayes' rule lots.

$$p(A \mid B) = \frac{p(B \mid A) \cdot p(A)}{p(B)}$$

$p(\boldsymbol{w})$ is the **prior**, and reflects our belief/assumptions about the weights.

$p(\mathcal{D} \mid \boldsymbol{w})$ is the **likelihood** of $\mathcal{D}$ given $\boldsymbol{w}$.

$p(\mathcal{D}) = \int_{\boldsymbol{w}} p(\mathcal{D} \mid \boldsymbol{w}) p(\boldsymbol{w})\, d\boldsymbol{w}$ is the **marginal likelihood**

The **posterior** is $p(\boldsymbol{w} \mid \mathcal{D})$, and by Bayes rule: $p(\boldsymbol{w} \mid \mathcal{D}) = \frac{p(\mathcal{D}|\boldsymbol{w})p(\boldsymbol{w})}{p(\mathcal{D})}$. It is the updated belief about the parameters given the data observed.

The **maximum a posteriori/MAP** estimate of $\boldsymbol{w}$ is $\boldsymbol{w}_{MAP}$ that maximises $p(\boldsymbol{w} \mid \mathcal{D})$, equivalently maximising $p(\mathcal{D} \mid \boldsymbol{w})p(\boldsymbol{w})$ / $\log p(\mathcal{D}|\boldsymbol{w}) + \log p(\boldsymbol{w})$, as $p(\mathcal{D})$ is just a difficult constant.

Applying the Bayesian approach to choose new weights:

- given pre-existing data $\mathcal{D}$, and a new input point $\boldsymbol{x}_{new}$, we want to predict y_{new} - precisely, calculate/approximate $p(y_{new}|x_{new}, \mathcal{D})$ for different values of y_{new}.
- we have $p(y_{new} \mid x_{new}, \mathcal{D}) = \int_{\boldsymbol{w}} p(y_{new} \mid \boldsymbol{w}, \boldsymbol{x}_{new}) p(\boldsymbol{w} \mid D)\, d\boldsymbol{w}$ $(\dagger)$ (by law of total prob)

1. we could just calculate $(\dagger)$ which gives us the entire distribution of y_{new}, so we can calculate the median/mean/etc. but this is a very complicated integral
   (a) we can simplify a little, as$(\dagger) = \int_{\boldsymbol{w}} p(y \mid \boldsymbol{w}, \boldsymbol{x}_{new}) \frac{p(\mathcal{D}|\boldsymbol{w})p(\boldsymbol{w})}{p(\mathcal{D})}\, d\boldsymbol{w}$, so we can ignore the $p(\mathcal{D})$ term. This is still complicated, as integrating over all $\boldsymbol{w}$ may be very large.
2. Even simpler: we just choose the most likely parameters $\boldsymbol{w}_{MAP}$ given the data at hand, and use just those for prediction - i.e. $p(y_{new} \mid x_{new}, \mathcal{D}) = p(y_{new} \mid \boldsymbol{w}_{MAP}, \boldsymbol{x}_{new})$
   (there should be some mathsy justification for this)

Thus, if we choose 2, then we have a very simple learning process, which is just calculating $\boldsymbol{w}_{MAP}$

If we apply approach 2 to LR, we get Ridge Regression.

We can also apply approach 1 to LR (as it's not too complicated), and the closed form solution is in the notes.

8 Optimisation: Convexity & LPs

Basics:

- objective function
- feasible set
- optimal value
- global optima
- local optima

$$\begin{aligned} \text{minimise: } & f(\boldsymbol{x}) & \\ \text{st} g_i(\boldsymbol{x}) &\leq 0 & i \leq m \\ h_j(\boldsymbol{x}) &= 0 & j \leq n \end{aligned}$$

Convexity

$C \subseteq \mathbb{R}^D$ is **convex** if $\forall \boldsymbol{x}, \boldsymbol{y} \in C\ \lambda \in [0,1]\ \lambda \boldsymbol{x} + (1-\lambda)\boldsymbol{y} \in C$. E.G: $\mathbb{R}^D$, intersections, polyhedra, set of positive semi-definite matrices

$f : C \rightarrow \mathbb{R}$, where C is convex, is a **convex function** if $\forall \boldsymbol{x}, \boldsymbol{y} \in C\ \lambda \in [0,1]\ f(\lambda \boldsymbol{x} + (1-\lambda)\boldsymbol{y}) \leq \lambda f(\boldsymbol{x}) + (1-\lambda) f(\boldsymbol{y})$.

Convex optimisation is of the form:

$$\begin{aligned} \text{minimise: } & f(\boldsymbol{x}) & \\ \text{st} g_i(\boldsymbol{x}) &\geq 0 & i \leq m \\ h_j(\boldsymbol{x}) &= 0 & j \leq n \end{aligned}$$

Where f, g_i are convex, and h_j are affine (note equiv to min f st $g_i, h_j, -h_j \leq 0$, which is still convex as $-h_j$ is still affine)

In convex optimisation, all <u>local optima are global optima.</u>

Problems as LPs

Note LPs are convex optimisation problems.

Absolute Loss LR can be expressed as an LP, with the same optima.

Lagrangian & duals

The Lagrange form of this is

$$\Lambda(\boldsymbol{x}; \boldsymbol{\alpha}, \boldsymbol{\mu}) := f(\boldsymbol{x}) - \sum_{i=1}^{m} \alpha_i g_i(\boldsymbol{x}) - \sum_{j=1}^{l} \mu_j h_j(\boldsymbol{x})$$

KKT conditions for a critical point $(\boldsymbol{x}^*, \boldsymbol{\alpha}^*, \boldsymbol{\mu}^*)$ to be a minimum:

1. primal feasibility: $g_i(\boldsymbol{x^*}) \geq \boldsymbol{0}, h_j(\boldsymbol{z}^*) = 0$ for $i = 1 \leq m, j \leq n$
2. dual feasibility: $\alpha_i^* \geq 0$ for $i \leq m$
3. complementary slackness: $\alpha_i^* g_i(\boldsymbol{x^*}) = 0$ for $i \leq m$.

If the problem is convex optimisation, then the KKT conditions are sufficient and necessary for a critical point of the Lagrangian to also be a global minimum of the original problem.

9 Gradient descent

remember gradients, hessians

Aim: optimise a function $f(\boldsymbol{w})$ wrt to the argument $\boldsymbol{w}$, on an unconstrained set.

Start: some sensible-enough $\boldsymbol{w}_0$

iterative step: $\boldsymbol{w}_{t+1} := \boldsymbol{w}_t - \eta_t \boldsymbol{g}_t = \boldsymbol{w}_t - \eta_t \nabla f(\boldsymbol{w}_t)$, where $\boldsymbol{g}_t$ is the **gradient** $\nabla f(\boldsymbol{w}_t)$, η_t is the **learning rate** at time t.

Choice of η_t: varies based on method - often a hyperparameter.

If f is convex, then the iterative process will converge to a global minimum (assuming η_t doesn't converge to 0 first). If not, we may find a local minima or a saddle point

Subgradients

Aim: gradients, but for convex non-differentiable functions.

$\boldsymbol{g}$ is a **sub-gradient** of f at $\boldsymbol{w}_0$ if $f(\boldsymbol{w}) \geq f(\boldsymbol{w}_0) + \boldsymbol{g}^\top(\boldsymbol{w} - \boldsymbol{w}_0)$

For example, $\operatorname{sign}(\boldsymbol{w})$ ($\operatorname{sign}(0)$ can be anything in $[-1, 1]$) is a subgradient for $\|\boldsymbol{w}\|_1$ at $\boldsymbol{0}$

Newton's Method

uses the quadratic approximation from Taylor's theorem near $\boldsymbol{w}_t$: $f_q(\boldsymbol{w}) = f(\boldsymbol{w}_t) + \boldsymbol{g}_t^\top(\boldsymbol{w} - \boldsymbol{w}_t) + {}^1\!/_2(\boldsymbol{w} - \boldsymbol{w}_t)^\top \boldsymbol{H}_t(\boldsymbol{w} - \boldsymbol{w}_t)$

We step to the stationary point of the quadratic approximation, so $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \boldsymbol{H}_t^{-1}\boldsymbol{g_t}$ is the **iterative step**.

Note this is very expensive per step, but also much faster than gradient descent.

Note it also seeks stationary points in general, rather than minima, so if we have a non-convex function, we may be going the wrong direction.

SGD/Minibatching

$\boldsymbol{g_i} := \nabla_{\boldsymbol{w}}\ell(\boldsymbol{w}; \boldsymbol{x}_i, y_i)$ is the loss wrt data point i out of N total.

Now consider $\boldsymbol{g}_I$, where $I \sim \mathrm{Unif}\{1..N\}$

$\mathbb{E}[\boldsymbol{g}_I] = \frac{1}{N}\sum_{i=1}^{N}\boldsymbol{g}_i$, which is the entire gradient.

Thus a method where at each step we only calculate the gradient wrt 1 random point and use that for gradient descent is probabilistically correct, and much faster.

Note the regularisation term must still be included, but may need scaling, depending on whether the original objective summed or averaged the $\boldsymbol{g}_i$'s, for $i \in 1..N$

Improve by "picking multiple I" - this is minibatching.

This also works well when we can't load all the data at once - i.e. it works **online**

Constraints with Gradient Descent

Gradient descent can step outside the feasible/constraint set.

Can return to it after each step using projection operator back into the constrained set (can be very expensive)

10 Generative Models for Classification (incl NBC)

The problem:

- input data $\boldsymbol{x}$ or $\boldsymbol{X}$ (N lots of $\boldsymbol{x}$)
- output classes $\{1...C\}$
- (parameters $\boldsymbol{\theta}$)

Generative plan: we consider $p(\boldsymbol{x}, y \mid \theta)$,not $p(y \mid \boldsymbol{x}, \theta)$

How to predict:

- given a point $\boldsymbol{x}_{new}$,for each possible class $c \in \{1..C\}$, calculate $p(y = c \mid \boldsymbol{x}_{new}, \theta)$

- choose the class with the highest - i.e. $\hat{y} := \mathrm{argmax}_{c\in\{1..C\}}\, p(y = c \mid \boldsymbol{x}_{new}, \theta)$
- practically, we use the $\propto$ version of Bayes rule: $p(y = c \mid \boldsymbol{x}_{new}, \theta) \propto p(y = c \mid \theta) \cdot p(\boldsymbol{x}_{new} \mid y = c, \theta)$
- Note the cross entropy loss is the NLL of $\prod_{c\in C}(\text{probability of c})^{\text{number of occurences of c}}$

Fitting data:

- we will split the parameters $\boldsymbol{\theta}$ into $\boldsymbol{\pi}$and $\boldsymbol{\theta}$, $\boldsymbol{\pi}$ for y and $\boldsymbol{\theta}$ for x.
- $p(\boldsymbol{x}, y \mid \boldsymbol{\theta}, \boldsymbol{\pi}) = p(y \mid \boldsymbol{\pi}) \cdot p(\boldsymbol{x} \mid y, \boldsymbol{\theta})$ (def of cond prob)
- $\pi_c := p(y = c \mid \boldsymbol{\pi})$, and note $\sum_c \pi_c = 1$.
- the **likelihood** of observing *iid* data $\mathcal{D} = \langle \boldsymbol{x}_i, y_i\rangle_{i=1}^N$ is $p(\mathcal{D} \mid \boldsymbol{\theta}, \boldsymbol{\pi}) = \prod_{i=1}^N \left(\left(\prod_{i=1}^C \pi_c^{\mathbf{1}(y_i=c)}\right) \cdot p(\boldsymbol{x}_i \mid y_i, \boldsymbol{\theta})\right)$
- the **log likelihood** of "" is $\log p(\mathcal{D} \mid \boldsymbol{\theta}, \boldsymbol{\pi}) = \sum_{c=1}^C N_c \log(\pi_c) + \sum_{i=1}^N \log p(\boldsymbol{x}_i \mid y_i, \boldsymbol{\theta})$, where N_c is the no. of datapoints with $y_i = c$.
- Thus we can just work out π_c without caring about $\boldsymbol{x}$'s, so $\pi_c = \dfrac{N_c}{N}$

Naive Bayes:

we assume that the features in $\boldsymbol{x}_i$ are conditionally independent, given $y_i = c$, to avoid $\boldsymbol{\theta}$ being too large. For each c, we consider $\boldsymbol{\theta}_c$, and so:

$$p(\boldsymbol{x}|y = c, \boldsymbol{\theta}_c) = \prod_{j=1}^D p(x_j|y = c, \boldsymbol{\theta}_{jc})$$

So for the whole dataset,

$$\log p(\mathcal{D}|\pi, \boldsymbol{\theta}) = \sum_{c=1}^C N_c \log(\pi_c) + \sum_{c=1}^C \sum_{j=1}^D \sum_{i=1}^N \mathbf{1}_{y_i=c} \log p(\boldsymbol{x}_{i,j} \mid \boldsymbol{\theta}_{j,c})$$

The type & choice of $\boldsymbol{\theta}_{jc}$ depends on what feature $\boldsymbol{x}_j$ is - e.g. Gaussian $(\mu_{jc}, \sigma_{jc}^2)$ for real x_j, multinoulli/bernoulli for categorical x_j.

To fit these, consider each feature separately, and find the empirical means/variances/marginal dists, and use those for the parameters.

Gaussian discriminant analysis

Everything above except the NBC section applies.

We assume all features are real, and model them as multivariate Gaussian for each class c. Thus we have a mean $\boldsymbol{\mu}_c$ and a covariance matrice Σ_c for each class c, and $p(\boldsymbol{x} \mid y = c, \boldsymbol{\theta}_c = (\boldsymbol{\mu}_c, \Sigma_c)) = \mathcal{N}(\boldsymbol{x} \mid \boldsymbol{\mu}_c, \Sigma_c)$

The MLE's for these are $\hat{\boldsymbol{\mu}}_c := {}^1/_{N_c} \sum_{i:y_i=c} \boldsymbol{x}_i$, $\hat{\boldsymbol{\Sigma}}_c := \frac{1}{N_c} \sum_{i:y_i=c} (\boldsymbol{x}_i - \hat{\boldsymbol{\mu}}_c)(\boldsymbol{x}_i - \hat{\boldsymbol{\mu}}_c)^\top$

The **decision boundaries** for a model are the line of points $\boldsymbol{x}$ with $p(y = c \mid \boldsymbol{x}, \boldsymbol{\theta}) = p(y = c' \mid \boldsymbol{x}, \boldsymbol{\theta})$ for two different classes c, c'.

For Gaussian Discriminant Analysis, these are (piecewise) quadratic curves.

Linear Discriminant Analysis: we share Σ between the classes (but keep separate weights), and then the decision boundaries are linear, and $p(y = c \mid \boldsymbol{x}, \boldsymbol{\theta}) = \text{softmax}(...)$ (see notes for details)

11 Logistic Regression

A discriminative model for binary classification (either class 0 or class 1), so we consider $p(y \mid \boldsymbol{w}, \boldsymbol{x})$, and model $y \sim \text{Bernoulli}(\text{sigmoid}(\boldsymbol{w} \cdot \boldsymbol{x}))$. (note $\boldsymbol{x}$ contains a constant column)

Prediction: $p(y_{new} = 1 \mid \boldsymbol{x}_{new}, \boldsymbol{w}) = \text{sigmoid}(\boldsymbol{w} \cdot \boldsymbol{x}_{new})$, and then to decide on a class, we pick whichever of $0, 1$ has probability $\geq 1/2$ - i.e. $\hat{y}_{new} = \mathbf{1}(\boldsymbol{w} \cdot \boldsymbol{x}_{new} \geq 0)$

Likelihood: $p(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{w}) = \prod_{i=1}^{N} \text{sigmoid}(\boldsymbol{w} \cdot \boldsymbol{x}_i)^{y_i} \cdot (1 - \text{sigmoid}(\boldsymbol{w} \cdot \boldsymbol{x}_i))^{1-y_i} = \prod_{i=1}^{N} \mu_i^{y_i} \cdot (1 - \mu_i)^{1-y_i}$, where $\mu_i = p\,\text{sigmoid}(\boldsymbol{w} \cdot \boldsymbol{x}_i)$

$NLL(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{w}) = -\sum_{i=1}^{N} (y_i \log \mu_i + (1 - y_i) \log(1 - \mu_i))$, which is just the cross-entropy function !!!! note minus sign!!!.

To calculate the weights $\boldsymbol{w}$ to minimise the NLL, we can use Newton's method/others, as the NLL is convex. This is then **iteratively reweighted least squares**, as it is an iterative procedure where we update the weights using least-squares on each step.

We can extend this to multi-class classifiers, and then $p(y \mid \boldsymbol{x}, \boldsymbol{W}) = \text{softmax}\left(\begin{bmatrix} \boldsymbol{w}_1 \cdot \boldsymbol{x} & \dots & \boldsymbol{w}_C \cdot \boldsymbol{x} \end{bmatrix}^\top\right)$, where $\boldsymbol{W}$ is now a $D \times C$ matrix.

12 SVMs

Aim: given a set of data $\mathcal{D} = \langle \boldsymbol{x}_i, y_i \rangle_{i=1}^{N}$ with positive $+1$ or negative -1 labels y_i, identify a linear separator that separates the positive from the negatives.

If such a line exists, this is the **linearly separable** setting.

We specify the linear separator, a hyperplane, by $\boldsymbol{w} \cdot \boldsymbol{x} + w_0 = 0$, for $w_0 \in \mathbb{R}, \boldsymbol{w} \in \mathbb{R}^D$, so the positive side has $\boldsymbol{w} \cdot \boldsymbol{x} + w_0 > 0$, and the negative $\boldsymbol{w} \cdot \boldsymbol{x} + w_0 < 0$

the **margin** of a point x is the distance from x to the separating hyperplane, $\text{margin}(\boldsymbol{x}) = \dfrac{|\boldsymbol{w} \cdot \boldsymbol{x} + w_0|}{||\boldsymbol{w}||_2}$

Linearly separable case:

we assume the data is separable, and so want to find $\boldsymbol{w}, w_0$ st $\forall i \; y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + w_0) \geq 1$.

- such weights exist, as we know the data is separable
- and we use ≥ 1 rather than > 0 as it's easier, and since the dataset is finite, we can always scale the weights to make the LHS arbitrarily large.

We want the maximum margin separator, so the optimisation problem is:

$$\begin{aligned} &\text{maximise } \min\{\text{margin}(\boldsymbol{x}) : \boldsymbol{x} \in \mathcal{D}\} && \\ &\qquad \text{st } y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + w_0) \geq 1 && \text{for } i = 1...N \end{aligned}$$

Since $\forall i \; y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + w_0) \geq 1$, and the margin for a datapoint is $\dfrac{y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + w_0)}{\|\boldsymbol{w}\|_2}$, clearly for any $\boldsymbol{x}$, $\text{margin}(\boldsymbol{x}) \geq \dfrac{1}{\|\boldsymbol{w}\|_2}$.

Thus, we can simplify the problem to the following quadratic program:

$$\begin{aligned} &\text{minimise } \frac{1}{2}\|\boldsymbol{w}\|_2^2 && \text{wrt } \boldsymbol{w}, w_0 && (1) \\ &\qquad \text{st } y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + w_0) \geq 1 && \text{for } i = 1...N && (2) \end{aligned}$$

An equivalent formulation is (see sheet3q3) (note, is still quadratic):

$$\begin{aligned} &\text{maximise } \alpha && \text{wrt } \boldsymbol{w}, w_0, \alpha && (3) \\ &\qquad \text{st } y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + w_0) \geq \alpha && \text{for } i = 1...N && (4) \\ &\qquad \|\boldsymbol{w}\|_2^2 \leq 1 && && (5) \end{aligned}$$

Non (linearly) separable case:

Since there is no linear separator, the best we can do is a separator that makes the fewest misclassifications. Thus:

$$\begin{aligned} &\text{minimise } \frac{1}{2}\|\boldsymbol{w}\|_2^2 + C\sum_{i=1}^{N} \zeta_i && \text{wrt } \boldsymbol{w}, w_0 && (6) \\ &\qquad \text{st } y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + w_0) \geq 1 - \zeta_i && \text{for } i = 1...N && (7) \\ &\qquad \zeta_i \geq 0 && && (8) \end{aligned}$$

- A feasible solution to this quadratic progam always exists

- $\zeta_i \geq 0$ is so that the program can't compensate for mistakes by setting $-\zeta_i$ big on a point it has classified right
- note that if $\zeta_i \in [0,1]$ then the solution is being penalised for not separating a datapoint "by enough", as the point is on the correct side of the line, but not by very far (as if the dataset is truly separable, we could choose weights that separate by margin ≥ 1)
- clearly the optimal ζ_i is $\zeta_i := \max\{0, 1 - y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + w_0)\}$, which we call $\ell_{\text{hinge}}(\boldsymbol{w}, w_0; \boldsymbol{x}_i, y_i)$, the **hinge loss**

This is equivalent to minimising the objective function

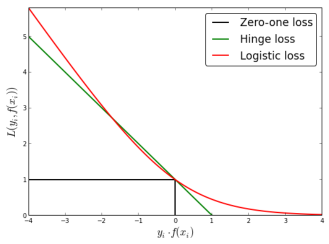
$$\mathcal{L}_{SVM}(\boldsymbol{w}, w_0 \mid \boldsymbol{X}, \boldsymbol{y}) := \frac{1}{2}\|w\|_2^2 + C\sum_{i=1}^{N} \ell_{\text{hinge}}(\boldsymbol{w}, w_0; \boldsymbol{x}_i, y_i)$$

This is convex but not diff, so subgradient descent can work.

If we want a smooth version of hinge loss, then we use **logistic loss**, which is the NLL of logistic regression on a single point:

$$\ell_{\text{logistic}}(y_i; \boldsymbol{w}, \boldsymbol{x}_i) - \log\left(1 + e^{-(2y_i - 1)(\boldsymbol{w} \cdot \boldsymbol{x}_i)}\right)$$

, which looks pretty similar:



Duals

If we write the Lagrangian of (6)-(8), and take its derivative wrt the primal variables then we get a stationary of the orig problem iff the derivatives $= 0$, and thus a minimum when the Lagrangian is minimised.

Simplified, we get:

$$\text{maximise} \sum_{i=1}^{N} \alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j (\boldsymbol{x}_i \cdot \boldsymbol{x}_j) \qquad \text{simplified Lagrangian} \tag{9}$$

$$\text{st} \sum_{i=1}^{N} \alpha_i y_i = 0 \qquad \text{from } \frac{\partial \Lambda}{\partial w_0} = 0 \tag{10}$$

$$0 \leq \alpha_i \leq C \text{ for } i = 1 \ldots m \qquad \text{from } \frac{\partial \Lambda}{\partial \zeta_i} = 0 \tag{11}$$

This has a simpler feasible space and fewer variables, but a more complicated objective (in N^2 terms), than the primal.

The complementary slackness conditions say that $\alpha_i \cdot (y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + w_0) - 1 + \zeta_i) = 0$, so either $\alpha_i = 0$ or $y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + w_0) = 1 - \zeta_i$, so $\boldsymbol{x}_i$ is a support vector, as it is one of the points with the least margin.

We have $\boldsymbol{w} = \sum_{i=1}^N \alpha_i y_i \boldsymbol{x}_i$ (from the derivatives of the Lagrangian), so clearly the optimal loss is made up of only the support vectors.

Multiclass classification

Either:

- **ovo:** train K choose 2 binary classifiers for each possible pair
- **ovr:** train K classifiers for each class k against all the rest

How to classify new points:

ovo:	choose the most commonly-output class from all the classifiers.
ovr:	hope only 1 classifier says yes, all others say "in the rest", otherwise compare $\boldsymbol{w} \cdot \boldsymbol{x}_{new}$'s.

Both a bit shit.

13 Kernel Methods

Note that since $\boldsymbol{w} = \sum_{i=1}^N \alpha_i y_i \boldsymbol{x}_i$, prediction is $\boldsymbol{w} \cdot \boldsymbol{x}_{new} = \boldsymbol{w} = \sum_{i=1}^N \alpha_i y_i (\boldsymbol{x}_i \cdot \boldsymbol{x}_{new})$, so all we actually need to be able to do is calculate inner products (as long as you solve using the dual)

we can generalise this to kernels κ as mentioned above, so then $\boldsymbol{w} \cdot \boldsymbol{x}_{new} = \boldsymbol{w} = \sum_{i=1}^N \alpha_i y_i \kappa(\boldsymbol{x}_i, \boldsymbol{x}_{new})$

Similar approaches work for other problems - e.g. for regularised linear regression, the optimal weights are always a linear combination of the input points, so we can build this into the optimisation problem and optimise for α_i instead, so we can then generalise the inner products.

Poly kernels

Recall degree d polynomial expansions from above. Used as-is (without introducing a kernel), the dual will be horrendous, as each vector has D^d entries.

Instead, we introduce a different expansion ϕ_d st $\phi_d(\boldsymbol{x}) \cdot \phi_d(\boldsymbol{x}') = (1 + \boldsymbol{x} \cdot \boldsymbol{x}')^d$ - e.g. for 2 $[1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, x_2^2, \sqrt{2}x_1 x_2]$ and then the inner produce is nice, but we still represent the same surfaces.

We can represent this instead as just the degree-d polynomial kernel $\kappa_d(\boldsymbol{x}, \boldsymbol{x}') := (1 + \boldsymbol{x} \cdot \boldsymbol{x}')^d$, which avoids having to actually expand the vectors

Mercer kernels

The **Gram matrix** of a kernel is

$$K := \begin{bmatrix} \kappa(\boldsymbol{x}_1, \boldsymbol{x}_1) & \cdots & \kappa(\boldsymbol{x}_1, \boldsymbol{x}_N) \\ \vdots & \ddots & \vdots \\ \kappa(\boldsymbol{x}_N, \boldsymbol{x}_1) & \cdots & \kappa(\boldsymbol{x}_1, \boldsymbol{x}_N) \end{bmatrix}$$

A kernel is a **Mercer kernel/ positive definite kernel** if the Gram matrix is always postive semi-definite.

If κ_1 and κ_2 are Mercer kernels, so are $\kappa_1 + \kappa_2, \alpha\kappa_1$ and $\kappa_1 \times \kappa_2$, and $\kappa(\boldsymbol{x}, \boldsymbol{x}') := \boldsymbol{x}^\top \boldsymbol{B}\boldsymbol{x}'$ is one, if B is semidefinite.

If we are solving the SVM dual with a Mercer kernel with Gram matrix K, we can write it as (where $\odot$ is elementwise multiplication)

$$\mathbf{1}^\top \boldsymbol{\alpha} - \frac{1}{2}(\boldsymbol{\alpha} \odot \boldsymbol{y})^\top \boldsymbol{K}(\boldsymbol{\alpha} \odot \boldsymbol{y})$$

This is concave, since K is positive-semidef, so has a unique max value, and a unique max point if K is pos-def.

Thus we choose Mercer kernels so that we can easily solve the SVM problem w/o worrying about local minima/runtime.

Examples:

- degree-d poly kernels as def above
- RBF kernels, which depend only on $\|\boldsymbol{x} - \boldsymbol{x}'\|_2$, e.g. the spherical Gaussian (above)
- for strings $\boldsymbol{x}, \boldsymbol{x}'$, let $\#_s(\boldsymbol{x})$ be no. of times the substring s occurs in $\boldsymbol{x}$. Then $\kappa(\boldsymbol{x}, \boldsymbol{x}') := \sum_{s \in \Sigma^*} w_s \cdot \#_s(\boldsymbol{x})\#_s(\boldsymbol{x}')$ is a Mercer kernel (and computable in $O(|\boldsymbol{x}| + |\boldsymbol{x}'|)$), that measures distance by common substrings (and allows weighting different substrings differently)

Measuring performance

Consider true & false positives/negatives

true positive rate $TPR = \dfrac{\#TP}{\#TP + \#FN}$

false positive rate $FPR = \dfrac{\#FP}{\#FP + \#TN}$

precision $Prec = \dfrac{\#TP}{\#TP + \#FP}$

We can plot the TPR against the FPR, giving us the **Receiver Operating Characteristic** curve when we vary some parameter - e.g. a threshold on $\boldsymbol{w} \cdot \boldsymbol{x}_{new}$, to see where we want to set the parameter for the given application.

The areas under the ROC curve is a measure of a classifier, as it shows which classifiers allow a good tradeoff.

The same can be done for the Precision against the TPR.

The **confusion matrix** is is a $C \times C$ matrix where N_{ij} is the # of elements in class j predicted to be in class i (so actual classes along the top, predicitions along the side).

A good confusion matrix is close to diagonal.

14 Neural Networks

a **unit/artificial neuron** with activation function f, bias $b$and weights $\boldsymbol{w}$ is a map $\boldsymbol{x} \mapsto f(b + \boldsymbol{w} \cdot \boldsymbol{x})$.

note logistic regression is an artificial neuron with the sigmoid function as the activation function.

Structure

We have $L \geq 1$ layers, where the first is the input, the last the output, and those in between the hidden layers.

Each layer l has n_l units, and each unit is connected to all on the previous layer.

$\boldsymbol{W}^l$ is the $n_l \times n_{l-1}$ matrix of weights for layer l, and $\boldsymbol{b}^l$ the $n_l \times 1$ vector of biases for layer l.

$\boldsymbol{z}^l := \boldsymbol{b}^l + \boldsymbol{W}^l \boldsymbol{a}^{l-1}$ is the preactivation vector for layer l, and $\boldsymbol{a}^l := f(\boldsymbol{z}^l)$ (mostly applied componentwise) is the activated vector for layers $l \geq 1$. These are the **forward equations**

Note $\boldsymbol{a}^0 := \boldsymbol{x}$, the input to the NN, and $\boldsymbol{y} := \boldsymbol{a}^L$ is the output of the NN.

$\boldsymbol{\theta}$ denotes all of the parameters in the model.

Loss & optimisation

We want to minimise an objective function of $\mathcal{L}(\boldsymbol{\theta}; \mathcal{D})$, which we separate into data points, so $\ell(\boldsymbol{x}_i, y_i; \boldsymbol{\theta})$ is the loss on a single datapoint, and is the difference between the model prediction $\hat{y}_i$ and the observed y_i.

We want the gradient wrt $\boldsymbol{\theta}$, which is:

$$\frac{\partial \mathcal{L}(\boldsymbol{\theta}; \mathcal{D})}{\partial \boldsymbol{\theta}} = \sum_{i=1}^{N} \frac{\partial \ell(\boldsymbol{x}_i, y_i; \boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$$

For a datapoint $(\boldsymbol{x}, y)$:

note ℓ " := " $\ell(\boldsymbol{x}, y; \boldsymbol{\theta})$
We want $\frac{\partial \ell}{\partial \boldsymbol{W}^i}$ and $\frac{\partial \ell}{\partial \boldsymbol{b}^i}$ for $i = 2..L$
We calculate the following derivatives:

$$\frac{\partial \ell}{\partial}\frac{\partial \ell}{\partial \boldsymbol{a}^L} \text{ depends on the loss function}$$

$$\frac{\partial \ell}{\partial \boldsymbol{z}^L} = \frac{\partial \ell}{\partial \boldsymbol{a}^L} \cdot \frac{\partial \boldsymbol{a}^L}{\partial \boldsymbol{z}^L} \tag{\textbf{BE1}}$$

$$\frac{\partial \ell}{\partial \boldsymbol{z}^i} = \frac{\partial \ell}{\partial \boldsymbol{z}^{i+1}} \boldsymbol{W}^{i+1} \frac{\partial \boldsymbol{a}^i}{\partial \boldsymbol{z}^i} \text{ for } i = 2...L-1 \tag{\textbf{BE2}}$$

$$\frac{\partial \ell}{\partial \boldsymbol{W}^i} = \left(\boldsymbol{a}^{i-1} \frac{\partial \ell}{\partial \boldsymbol{z}^i} \right)^\top \tag{\textbf{BE3}}$$

$$\frac{\partial \ell}{\partial \boldsymbol{b}^i} = \frac{\partial \ell}{\partial \boldsymbol{z}^i} \tag{\textbf{BE4}}$$

Considerations in training

Optimisation algorithm choice: ?????????????????????????

Runtime & space: The dominant term in backpropagation's runtime is the matrix multiplication.

We also need to store all the model parameters and preactivations and activations.

We generally do this in batches of data points, so all calculations above become tensors.

Convexity: Optimising the loss is not a convex problem, as units in the same layer are interchangable.

For categorical classification, we use cross-entropy loss, so $\ell(\boldsymbol{x}_i, y_i; \boldsymbol{\theta}) = -(y_i \log \hat{y}_i + (1 - y_i)\log(1 - \hat{y}_i))$, for a binary classification $y_i, \hat{y}_i \in [0, 1]$

Saturation: this is when the activation function for some layer is very flat, and thus gradient steps (in Newton's/whatever) are very small, and we make no progress.

Fixes:

- sometimes swapping the loss function to cross-entropy
- using ReLU, which has gradient 1 on half the plane, so it can't saturate that side
- setting the initial weights correctly (small positive biases for ReLU, weights from $N(0, 1/D)$ for Relu or sigmoid))
- note the randomness of initial weights is important, so that all the units aren't doing the same thing.

Vanishing & exploding gradients: e.g. $\sigma(t) \leq 0.25$, so if we have several layers, the gradients $\frac{\partial \ell}{\partial x_1}$ can rapidly vanish. Similarly, they can also explode, if they are all large.

Overfitting: We often have far too many parameters. We can use:

- standard regularisation like a $\ell_2$penalty
- **early stopping**: after each step, test the algorithm on a validation set, and when the performance starts to plateau on that, stop. This may not work because it is a non-convex problem.
- **adding data**: obvious, but difficult - might be able to rotate/scale/transform existing data?
- **dropout:** on each training step, a proportion (often 1/2) of the units in specificied layers are randomly dropped/ignored (their activation values assumed 0), and we train only with the ones left.
  - we use the full layer for testing
  - we must scale the weights down in the dropout layer, by the dropout proportion.
- smaller layers

15 CNNs

Input: an image, in the form of a 3d tensor of shape $m \times n \times c$, each element in $[0, 1]$, representing an image with c channels.

A **convolution filter** is a tensor of size $W \times H \times c, W, H < m, n$.

the **convolution operation** applies a filter to the image by taking the dot product of the filter and $W \times H$ areas of the image, at regular intervals in each dimension.

The **stride** controls the distance between (the centre) of areas convoluted (can be different for each dim)

The **padding** adds rows and cols of 0s to the edges, which allows every part of the original image to be included in a convolution

If we apply K kernels at once, the result is a $m' \times n' \times c'$ matrix, where $m' = \left\lfloor \frac{m + 2p - f}{s} + 1 \right\rfloor$, where s is the stride, f the size of the filter, and p the padding amount on each side. Same for n, n'.

For each layer,

- $\boldsymbol{A}^l$ is the activation tensor, defined by $g^l(\boldsymbol{Z}^l)$ for some (non-linear) activation function g,
- $\boldsymbol{W}^l$ is the convolution tensor that contains the the F_l filters on layer l

- $\boldsymbol{Z}^l := \boldsymbol{W}^l \star \boldsymbol{A}^{l-1} + \boldsymbol{b}^l$, where $\star$ represents convolution, taking into account stride & padding.

If we have no padding and stride$= 1$, this is explicitly:

$$z^{l+1}_{i',j',f'} := b^{l+1,f'} + \sum_{i=1}^{W_{f'}} \sum_{j=1}^{H_{f'}} \sum_{f=1}^{F_l} a^l_{i'+i-1,j'+j-1,f} w^{l+1,f'}_{i,j,f}$$

We can then calculate the standard backpropagation derivatives wrt $\boldsymbol{W}^l$ and $\boldsymbol{b}^l$, and use those as normal.

Pooling layers: a convolution layer will often output the same information in neighbouring pixels - e.g. an edge finder will find an edge at pixels all along it, so we add a layer that simplifies small areas.

Max-pool takes the max over a small area. let $\Omega(k,l)$ be the set of indices in the input layer which are considered at the point (k,l) in the output layer

the forward equation is $s^{l+1}_{k,l} := \max_{i,j\in\Omega(k,l)} a^l_{i,j}$

the backward equation is $\frac{\partial s^{l+1}_{k,l}}{\partial a^l_{i,j}} = \mathbf{1}\left((i,j) = \mathrm{argmax}_{o,p\in\Omega(k,l)} a^l_{o,p}\right)$

16 RNNs & N-grams

A **language** model is a model that takes as input a sequence of words $\boldsymbol{w} = (w_1, ..., w_n)$ and returns a probability $p(\boldsymbol{w})$, st $\sum_{\boldsymbol{w}\in\Sigma^*} p(\boldsymbol{w}) = 1$. We use them to answer the question "given our training, how probable is this new utterance $\boldsymbol{w}$?"

The problem: we want to map from input $\boldsymbol{x}$ to some utterance $\boldsymbol{w}$. It is often difficult to model $p(\boldsymbol{w} \mid \boldsymbol{x})$, where training data for the $\boldsymbol{x} \longrightarrow \boldsymbol{w}$ "conversion" is limited, and instead use Bayes' Rule to get $p(\boldsymbol{w}) \cdot p(\boldsymbol{x} \mid \boldsymbol{w})$.

$p(\boldsymbol{w}) = p(w_1) \cdot p(w_2 \mid w_1) \cdots p(w_N \mid w_1, .., w_{n-1})$ allows us to split a big joint probability up into smaller chunks.

Evaluation is either by <u>cross entropy</u>: $H(\boldsymbol{w}) = -\frac{1}{n}\log_2 p(\boldsymbol{w})$, or by <u>perplexity</u>, $2^{H(\boldsymbol{w})}$, which is very sensitive to the tokenisation of the data.

N-Grams

For a 2-Gram model, use the Markov chain assumption, st $p(\boldsymbol{w}) = p(w_1) \cdot p(w_2 \mid w_1) \cdots p(w_N \mid w_{N-1})$

For an N-Gram model, we assume $p(w_k \mid w_1, ..., w_{k-1}) \approx p(w_k \mid w_{k-N+1}, ..., w_{k-1})$. <u>note</u>: N is the length of the whole sequence we actually consider.

Model fitting is thus estimating these conditional probabilities of "$N-1$ words $\longrightarrow$ 1 successor".

note that we add a special starting symbol, say $< s >$, and assume there are N of those at the start of a sentence, and a $< eos >$ symbol for the end, which the model will generate when it thinks its done.

The traditional model fit for $p(w_k \mid w_{k-N+1}, ..., w_{k-1})$ is just $\frac{\text{Count}(w_{k-N+1}...w_k)}{\text{Count}(w_{k-N+1}...w_{k-1})}$

The NN method is to train a neural network - normally a pretty simple one with just 1 hidden layer, that takes as input $N - 1$ words, and outputs a probability distribution over the set of possible words.

E.g., a trigram network:

- input: $\boldsymbol{x} := [w_{-2}; w_{-1}]$ (i.e. the concatenation of 2 one-hot vectors)
- hidden: $\boldsymbol{h} := g(V\boldsymbol{x} + \boldsymbol{c})$
- output: $\hat{\boldsymbol{p}} := \text{softmax}(W\boldsymbol{h} + \boldsymbol{b})$
- use: convert training sentences into triples $[w_{n-2}, w_{n-1}, w_n]$, run the NN on input $[w_{n-2}, w_{n-1}]$; and given output $\hat{\boldsymbol{p}}_n$, compare $\hat{\boldsymbol{p}}_n$ to the actual w_n
  - (remember to add $< s >$'s at the start when converting to triples)

a word is often represented as a one-hot, so both it and the prob. dist. will be very large vectors.

objective: $\mathcal{F}(\boldsymbol{w}) - \frac{1}{|\boldsymbol{w}|} \sum_n \text{cost}(w_n, \hat{\boldsymbol{p}}_n)$, where the cost function is $\text{cost}(\boldsymbol{a}, \boldsymbol{b}) = \boldsymbol{a}^T \log \boldsymbol{b}$

Then we can backpropagate as normal.

Sampling: given a starting point of w_{-1}, w_0 (e.g. $< s >, < s >$), sample w_1 from the NN as above (MAP on $\hat{\boldsymbol{p}}_1$). Then repeat to get w_2, with input $w_0, w_1....$

RNNs

Model:

- input: a sequence $\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_n$ of input vectors
- hidden layer: $\boldsymbol{h}_n := g(V[\boldsymbol{x}_n; \boldsymbol{h}_{n-1}] + \boldsymbol{c})$ - note the weights are shared across time.
- output: $\hat{\boldsymbol{y}}_n := \text{softmax}(W\boldsymbol{h}_n + \boldsymbol{b})$

Thus each hidden layer depends on the last.

For a length N sequence (e.g. a sentence with N words), we can "unroll" the network into a DAG computation graph.

Backprop for derivative of total cost wrt a hidden layer value:

$$\frac{\partial \mathcal{F}}{\delta h_2} = \sum_{i=2\rightarrow\text{end of seq}} \frac{\partial \mathcal{F}}{\delta \text{cost}_i} \frac{\partial \text{cost}_i}{\delta \hat{\boldsymbol{p}}_i} \frac{\partial \hat{\boldsymbol{p}}_i}{\delta h_i}$$

(NB the sum

Backpropagation can be done as normal - called BPTT.

This is impractical for long sequences, so we do **Truncated BPTT:**

$$\frac{\partial \mathcal{F}}{\delta h_2} = \sum_{i=2\to\textbf{some limit}} \frac{\partial \mathcal{F}}{\delta \text{cost}_i} \frac{\partial \text{cost}_i}{\delta \hat{\boldsymbol{p}}_i} \frac{\partial \hat{\boldsymbol{p}}_i}{\delta h_i}$$

epochwise T-BPTT:

- an input sequence is split into length k subsequences - so the limit is nk for some n.
- for each layer weight to be learned, we only consider its effect on costs within that sequence
- this is somewhat to splitting training examples up into separate length k inputs

more general T-BPTT:

- when we want to update the weights based on the cost of $\hat{\boldsymbol{y}}_i$ and $\boldsymbol{y}_i$, we only generate derivatives via hidden layers for $i...i+k$
- All points in the input sequence are now treated equally
- more general, but likely slower...

17 PCA

dimensionality reduction technique

input: n vectors $\boldsymbol{x}_1, ..., \boldsymbol{x}_n \in \mathbb{R}^D$, assumed to be centred, by subtracting their mean $\boldsymbol{\mu}$ - i.e. $\sum_{i=1}^n \boldsymbol{x}_i = \boldsymbol{0}$

given a target no. of dimensions $k \ll D$, we aim to find a orthonormal set of k vectors $\boldsymbol{u}_1, ..., \boldsymbol{u}_k \in \mathbb{R}^D$ st $\forall i \in \{1..n\}$ we can approximate $\boldsymbol{x}_i$ by the vector $\tilde{\boldsymbol{x}_i} := z_{i1}\boldsymbol{u}_1 + \cdots + z_{ik}\boldsymbol{u}_k$ for scalars $z_{ij} \in \mathbb{R}$.

We aim to find the optimal set of $\boldsymbol{u}_i$'s, under the reconstruction error $\sum_{i=1}^n ||\boldsymbol{x}_i - \widetilde{\boldsymbol{x}_i}||$.

If we let $\boldsymbol{X}$ be a matrix with the $\boldsymbol{x}_i$ as columns, and $\tilde{\boldsymbol{X}}$ one with the $\tilde{\boldsymbol{x}}_i$ as cols, then the rec. error is just $||\boldsymbol{X} - \tilde{\boldsymbol{X}}||_F^2$, so by Eckhart-Young the optimal $\tilde{\boldsymbol{X}}$ is $\boldsymbol{X}_k$ from truncated SVD. Then $\boldsymbol{U}_k$ is the matrix with cols = left-singular vectors of k-truncated SVD.

$\boldsymbol{Z} := \boldsymbol{U}_k^\top \boldsymbol{X}$ is then the matric of coefficients z_{ij}.

PCA thus returns $\boldsymbol{X}_k$ and $\boldsymbol{Z}$.

Some uses:

- matching against a large dataset: simplify dataset with PCA, then for new point $\boldsymbol{x}$, the coords of $\boldsymbol{x} - \boldsymbol{\mu}$ wrt the spanning $\boldsymbol{u}_i$ are $\boldsymbol{U}_k^\top(\boldsymbol{x} - \boldsymbol{\mu})$, and then we choose the closest point to those coords.
- grouping based on some specific characteristics: use PCA to somehow "distil" each point into k principal components, then use inner products to compare?

SVD

Given a real $m \times n$ matrix A with rank r, a **singular value decomposition** of A is $A = U\Sigma V^\top$,where

- U is a $m \times r$ matrix st $U^\top U = I_r$. i.e. the columns of U are orthonormal and are called the **left singular vectors $\boldsymbol{u}_i$**
- V is a $n \times r$ matrix st $V^\top V = I_r$. i.e. the columns of U are orthonormal and are called the **right singular vectors $\boldsymbol{v}_i$**
- Σ is a $r \times r$ diagonal matrix with entries $\sigma_1 \geq \cdots \cdot \geq \sigma_r > 0$

$A = U\Sigma V^\top = \sum_{i=1}^r \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^\top$, and $\forall i = 1..r$, $A\boldsymbol{v}_i = \sigma_i \boldsymbol{u}_i$.

This is the reduced SVD.

We can expand to the **full SVD** by:

- adding $m - r$ extra columns of more left singular vectors (which span the left null space) to U to get $\widehat{U}$, $m \times m$
- adding $n - r$ extra columns of more right singular vectors (which span the right=standard null space) to V to get $\widehat{V}$, $n \times n$
- adding extra zeros to Σ to get a $m \times n$ matrix $\widehat{\Sigma}$

A full SVD represents A as a rotation, a scaling and then another rotation.

Every matrix has an SVD, the σ_i are uniquely determined as $\{\sqrt{\lambda} : \lambda \neq 0, \lambda$ is an eigenvalue of $A^\top A$ (equiv of $AA^\top$)$\}$

note the Frobenius norm of a matrix is $\|A\|_F := \sqrt{\operatorname{trace}(A^T A)} = \sum_i \sigma_i^2$ for the σ_i from SVD

given $A = \sum_{i=1}^r \sigma_i u_i v_i^\top$, define the **SVD truncation** $A_k := \sum_{i=1}^k \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^\top$ for $k \leq r$. A_k is a rank k matrix, and is the best rank k approximation to A under the Frob norm (**Eckhart-Young Theorem)**.

Thus, for any $k \leq r$ we know what the "best" approximation is, so if we want to choose a k, the relative error is: $\dfrac{\|A - A_k\|_F^2}{\|A\|_F^2} = \frac{\sigma_{k+1}^2 + \cdots + \sigma_r^2}{\sigma_1^2 + \cdots + \sigma_r^2}$, and we can use this to find the least k with acceptable error.

Calculation:

to find $\boldsymbol{v}_1$:

- randomly choose a unit vector $\boldsymbol{x}_0$
- $\boldsymbol{x}_k := (A^\top A)\boldsymbol{x}_{k-1}$, $\boldsymbol{y}_k := \boldsymbol{x}_k$ normalised, stop when $\|\boldsymbol{y}_k - \boldsymbol{y}_{k-1}\|$ is sufficiently small and return y_k
- this will converge to $\boldsymbol{v}_1$ as long as the first two eigenvalues of $A^\top A$ aren't the same.

To find $\boldsymbol{v}_{k+1}$: after applying $A^\perp A$, we project ortho to $\boldsymbol{v}_1$: i.e. $\boldsymbol{x}_k := \boldsymbol{x}_k - (\boldsymbol{x}_k^\perp \boldsymbol{v}_1)\boldsymbol{v}_1$

18 Clustering

18.1 Partition based clustering

input data: $\boldsymbol{x}_i \in \mathbb{R}^D$ for $i = 1..N$

goal: output a partition of $\{\boldsymbol{x}_1, ..., \boldsymbol{x}_N\}$

18.1.1 k-Means

we choose the partitions $C_1, ..., C_k$ to minimise the distance of each $\boldsymbol{x}_i$ to the mean $\boldsymbol{\mu}_j$ of the cluster $\boldsymbol{x}_i$ is in. Thus:

$$W(C_1, ..., C_k) = \sum_{j=1}^{k} \sum_{i \in C_j} \left\| \boldsymbol{x}_i - \left(\sum_{l \in C_j} \boldsymbol{x}_l \right) \right\|_2^2$$

Optimising this is NP-hard.

Lloyd's algorithm: simple, slow convergence, not neccesarily to the global optimum

- initialise $\boldsymbol{\mu}_1, ..., \boldsymbol{\mu}_k$ suitably (e.g. randomly chosen datapoints)
- repeat:
  - assign each point to the closest mean (normally squared Euclidian, others available)
  - update each mean to the mean of the points assigned to it
- until "convergence" - i.e. the partitions don't change any more.

Note initialisation really matters, so often we run it several times, and choose the best of the returned partitions

How to choose k:

- plot a graph of objective function against k
- it should have an "elbow" - as k increases towards the optimum, the objective should decrease, and past the optimum, the drop should be much slower (as we're making pointless distinctions)

Transforming data

can represent data as either points in $\mathbb{R}^D$, or as a $N \times N$ dissimilarity matrix, which is a symmetric matrix where the (i, j)th entry is a measure of dis/similarity between points i and j.

We can convert points in $\mathbb{R}^D$ to dissimilarity matrices by deciding on a distance function (which may differ for each feature, weight them differently, and put the whole thing through some non-decr function - e.g. tanh)

Converting a dissimilarity matrix M to $\mathbb{R}^D$: we want to solve $\operatorname{argmin}_{\tilde{\boldsymbol{x}}_1,...,\tilde{\boldsymbol{x}}_N} \sum_{i \neq j}(\|\tilde{\boldsymbol{x}}_i - \tilde{\boldsymbol{x}}_j\|_2^2 - M_{ij})^2$.

If M is pos-semidef:

- SVD $M = U\Sigma U^\top$ - note since M is square, symmetric and pos–semidef, $U = V$, and U, Σ are both $N \times N$
- let $\widetilde{\boldsymbol{X}} := U\sqrt{\Sigma}$ (exists as the diagonal is non-negative)
- $\widetilde{\boldsymbol{X}}\widetilde{\boldsymbol{X}}^\top = ... = M$, so the distances between the points defined by the cols in $\widetilde{\boldsymbol{X}}$ match M
- so return the columns of $\widetilde{\boldsymbol{X}}$.

(if M is not pos-semidef, we have to just solve the optimisation problem , which is not convex and has no closed form)

Hierarchical Clustering

want to represent relations between clusters, to show that some are more similar than others. Thus we produce a tree of clusters, with the coarsest (the whole dataset) at the top, and the finest (individual points) at the bottom.

1 option is to recursively divide (e.g. with 2-means) clusters to build up a tree.

The other is is to start with N (num. of points) clusters, and merge them to build the tree.

Linkage algorithm:

- initialise clusters as N singletons
- let $S = \{1..N\}$ be the set of clusters available for merging
- repeat, until $|S| = 2$:
  - let j, k be the indices to the 2 closest clusters
  - let $C_l := C_j \cup C_k$ be a new cluster (and a new index l)
  - add l to S, and remove j, k
  - update distances $d(C_i, C_j)$ for all i, j (using chosen linkage rule)

Common linkage rules for $d(C, C')$:

- **single**: $\min_{x \in C, x' \in C'} d(x, x')$
- **average**: $\frac{1}{|C| \times |C'|} \sum_{x \in C, x' \in C'} d(x, x')$
- **complete**: $\max_{x \in C, x' \in C'} d(x, x')$

Spectral clustering

when we have data in $\mathbb{R}^D$, but clusters may not be convex (which k-means enforces).

We create an undirected graph of the data, and partition that. We have a parameter k.

for each datapoint, we add a node.

for each node, we connect it to its k nearest neighbours. (some might have more degree $> k$)

we set a weight on each edge, roughly representing distance - commonly, $w_{ij} := \exp\left(-\|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2^2/\sigma\right)$, for some width parameter σ.

We now have a graph, so we can partition it:

- consider it as a symmetric $N \times N$ adjacency matrix W, where $W_{ij}$is the weight of the edge (i, j), and is 0 if i, j aren't connected. $W_{ii} = 0$ for all i, as no selfloops are allowed
- let D be the diagonal matrix $D_{ii} = \sum_j W_{ij}$, and $L := D - W$ the Laplacian of the graph
- Find the k eigenvectors $\boldsymbol{v}_1, ..., \boldsymbol{v}_k$ corresponding to the k smallest eigenvalues
- Construct $\boldsymbol{V}_k := [\boldsymbol{v}_2...., \boldsymbol{v}_k]$, the $N \times (k-1)$ embedding matrix
- Apply a clustering algorithm, e.g. k-means to the rows in $\boldsymbol{V}_k$, associating each row with one of the original points.

FINISH!!!!!!!!!