1. The costs of some classifiers:

We have a training set of \( N \) examples, with \( D \)-dimensional features and binary labels.

Assume the following computational complexities: matrix-matrix multiplication \( AB \) costs \( O(LMN) \) for \( L \times M \) and \( M \times N \) matrices \( A \) and \( B \). Inverting an \( N \times N \) matrix \( G \) and/or finding its determinant costs \( O(N^3) \).

a) What is the computational complexity of training a “Bayes classifier” that models the features of each class by a maximum likelihood Gaussian fit (a Gaussian matching the mean and covariances of the features)?

Answer:
Here is a terse summary of costs, with a sketch of where each cost comes from:

Fitting the class fractions, \( \pi^{(c)} \): \( O(N) \). Sweep through dataset once, maintaining counts for each class.

Fitting all of the class means, \( \{\mu^{(c)}\} \): \( O(ND) \). Sweep through dataset once, for each example add \( D \) feature values on to running total for each class.

Fitting the class covariances, \( \{\Sigma^{(c)}\} \): \( O(ND^2) \). Sweep through dataset once, for each example adding \( xx^\top \) to a running total for the relevant class. Get covariances from these totals and the means.

At that point we have the parameters of the model. However, to use the classifier we will need to evaluate expressions involving \( \Sigma^{(c)}^{-1} \) and |\( \Sigma^{(c)} \)| for each class \( c \). We can precompute these terms, or factorizations of \( \Sigma^{(c)} \) from which they can be quickly computed, at training time in \( O(D^3) \) for each class.

Thus the cost of training the binary classifier is summarized as \( O(ND^2 + D^3) \). It’s dominated by measuring the covariances, and factoring them could occasionally be significant.

b) What is the computational complexity of assigning class probabilities to a test feature vector?

Answer:
The log-posterior for each class is:

\[
\log P(y = c | x) = -\frac{1}{2} (x - \mu^{(c)})^\top (\Sigma^{(c)})^{-1} (x - \mu) - \frac{1}{2} \log |\Sigma^{(c)}| + \log \pi^{(c)} + \text{const.}
\]

Computing this expression is dominated by \( (\Sigma^{(c)})^{-1}(x - \mu^{(c)}) \) or an equivalent computation, which is \( O(D^2) \) if the inverse or a factorization of \( \Sigma \) has been precomputed. We compute this score for each class, and take a sigmoid (or softmax for \( >2 \) classes, see last week’s tutorial).

---

1. In fact, good implementations usually take a factorization of \( G \) rather than inverting it, which also costs \( O(N^3) \). Given that factorization, we can compute \( G^{-1}H \) in \( O(N^2 K) \), where \( H \) is \( N \times K \), and find the (log) determinant in \( O(N) \). I’m using “big-O notation”.

You don’t want to redo $O(D^3)$ matrix factorizations/inversions for each test example. These can be done once at training time.

c) In *linear discriminant analysis* we assume the classes just have different means, and share the same covariance matrix. Show that given its parameters $\theta$, the “log odds” for a binary classifier,

$$\log \frac{p(x \mid y=1, \theta)}{p(x \mid y=0, \theta)} = \log p(x \mid y=1, \theta) - \log p(x \mid y=0, \theta),$$

is a linear function of $x$ (as opposed to quadratic).

**Answer:**
The expression in the answer to b) is quadratic in $x$ with quadratic term $-\frac{1}{2} x^\top (\Sigma^{(c)})^{-1} x$.

As in last week’s tutorial, the posterior probability in a binary classifier takes the logistic sigmoid of the difference between two of these expression. The quadratic term cancels if the class covariances are equal, leaving only terms that are linear or constant with respect to $x$.

d) When the log odds are linear, the predictions have the same form as logistic regression: $P(y=1 \mid x, \theta) = \sigma(w^\top x + b)$, but the parameters $w(\theta)$ and $b(\theta)$ are fitted differently.

How do your answers to a) and b) change when the classes share one covariance? What can you say about the cost of the linear discriminant analysis method compared to logistic regression?

**Answer:**
At test time the cost is the same as logistic regression: $O(D)$ to take the inner product of the example with a weight vector.

At training time we still need to compute the shared covariance and to factorize it, which will cost $O(ND^2 + D^3)$.

(The shared covariance is estimated with the average of $(x - \mu^{(c)})(x - \mu^{(c)})^\top$, using the residual of each datapoint from its class mean.)

Finding the $w$ and $b$ parameters doesn’t have significant extra cost. Taking the difference between two terms in b) with $\Sigma^{(1)} = \Sigma^{(0)} = \Sigma$ gives a linear term $x^\top [\Sigma^{-1}(\mu^{(1)} - \mu^{(0)})]$. The term in square brackets are the weights $w$, which can be pre-computed in $O(D^2)$ given a factorization/inverse of $\Sigma$. Computing the constant terms for the bias is no more expensive.

It is difficult to compare the training cost to logistic regression. Evaluating the log-probabilities for all training items (and gradients) costs $O(ND)$ with logistic regression. And no $O(D^3)$ storage is required, so logistic regression scales more easily to enormous numbers of features. For really large datasets, logistic regression training might sweep through the dataset only a few times. For smaller datasets, the iterative optimization procedure can easily cost more in practice than the closed form computations for linear discriminant analysis.

**Discussion of Q1:**
You should try to get a rough idea of the scaling of algorithms that you learn. I’ve gone into more detail on the book-keeping of costs than I normally would here. However, I think you should be able to see that a Gaussian classifier spends $O(ND^2)$ time accumulating
covariances, and make similar statements for other models. It’s also useful to notice memory costs, here $O(D^2)$ for each covariance matrix, which can put a hard limit on the scale of problems that can be tackled.

Many machine learning methods are fitted with gradient-based optimizers. It’s common to only consider the cost per update, and assume that a reasonable number of sweeps through the dataset will be required. Different methods might require different numbers of updates, but that’s often ignored.

Computational cost is only one way to compare models. We might also be interested in comparing the accuracy of the models. Linear discriminant analysis makes predictions of the same form as logistic regression. For large datasets, logistic regression is usually more accurate as it explicitly optimizes classification performance using the same family of functions (it’s a “discriminative” rather than “generative” method). However, not setting the class covariances equal creates a non-linear classifier (quadratic discriminant analysis) so a Gaussian Bayes classifier can perform better than logistic regression when the best decision boundary is non-linear.

2. Linear autoencoder

We centre our data so it has zero mean and fit a linear autoencoder with no bias parameters. The autoencoder is a $D$-dimensional vector-valued function $f$, computed from $D$-dimensional inputs $x$, using an intermediate $K$-dimensional “hidden” vector $h$:

$$h = W^{(1)}x$$
$$f = W^{(2)}h.$$  

Assume we want to find a setting of the parameters that minimizes the square error $\|f - x\|^2 = (f - x)^\top (f - x)$, averaged (or summed) over training examples.

a) What are the sizes of the weight matrices $W^{(1)}$ and $W^{(2)}$? Why is it usually not possible to get zero error for $K < D$?

**Answer:**

$W^{(1)}$ is $K \times D$
$W^{(2)}$ is $D \times K$

$f = (W^{(2)}W^{(1)})x$.

The geometry of this transformation has previously been discussed in the notes. The final multiplication by $W^{(2)}$ brings points in $K$-dimensions up into $D$-dimensions, but the $\{f^{(n)}\}$ points will still all be in a $K$-dimensional linear subspace. Unless the $\{x^{(n)}\}$ points happen to lie exactly in a $K$-dimensional linear subspace, they can’t be exactly fitted.

An alternative explanation is that to get zero error, we need $W^{(2)}W^{(1)} = I$. However, this matrix product has rank at most $K$, and $I$ has rank $D$.

b) It’s common to transform a batch (or “mini-batch”) of data at one time. Given an $N \times D$ matrix of inputs $X$, we set:

$$H = XW^{(1)}\top$$
$$F = HW^{(2)}\top$$

The total square error $E = \sum_{nd}(F_{nd} - X_{nd})^2$, has derivatives with respect to the
neural network output

\[ \frac{\partial E}{\partial F_{nd}} = 2(F_{nd} - X_{nd}), \] which we write as \( \bar{F} = 2(F - X) \).

Using the backpropagation rule for matrix multiplication,

\[ C = AB^\top \Rightarrow \bar{A} = \bar{CB} \text{ and } \bar{B} = \bar{C}^\top A, \]

write down how to compute derivatives of the cost with respect to \( W^{(1)} \) and \( W^{(2)} \).

If time: you should be able to check numerically whether you are right.

\textbf{Answer:}
Applying the rule given in the question:

\[ E = \sum_{nd}(F_{nd} - X_{nd})^2 \]
\[ F = HW^{(2)}^\top \]
\[ H = XW^{(1)}_n \]
\[ \bar{W}^{(1)} = \bar{H}^\top X \]

The forwards computation goes from the bottom of the left column up to the top. Backpropagation then computes the second column from top to bottom. A demonstration of checking these expressions is available separately (probably not for discussion in tutorials, but you can ask questions online if not clear.)

c) The PCA solution sets \( W^{(1)} = V \) and \( W^{(2)} = V^\top \), where the columns of \( V \) contain eigenvectors of the covariance of the inputs. We only really need to fit one matrix to minimize square error.

Tying the weight matrices together: \( W^{(1)} = U \) and \( W^{(2)} = U^\top \), we can fit one matrix \( U \) by giving its gradients \( \bar{U} = \bar{W}^{(1)} + \bar{W}^{(2)}^\top \) to a gradient-based optimizer. Will we obtain the same solution as PCA?

\textbf{Answer:}
I mainly put this question in just to show you an example of “parameter tying”. Sharing parameters in different parts of a model is common in machine learning. It’s often easier to think about the parameters being separate at first, and then their derivatives just get added up.

We won’t usually fit \( U = V \). The cost function only cares about how well the output matches the input, so the optimizer could find \( U = QV \), where \( Q \) is any orthogonal matrix, fitting the function \( f = (V^\top Q^\top QV)x = (V^\top V)x \), which matches what a PCA projection would do and so has the same cost.

(Any other reasonable argument is fine. For example, explaining that the hidden units can be permuted, and so there are multiple solutions.)

3. Non-linear autoencoders

Some datapoints lie along the one-dimensional circumference of a semi-circle. You could create such a dataset, by drawing one of the features from a uniform distribution between \(-1\) and \(+1\), and setting the other feature based on that:

\[ x_1^{(n)} \sim \text{Uniform}[-1, 1] \]
\[ x_2^{(n)} = \sqrt{1 - (x_1^{(n)})^2}. \]
a) Explain why these points can’t be perfectly reconstructed when passed through the linear autoencoder in Q2 with $K=1$.

Answer:
The $\{f^{(n)}\}$ points must all lie in a $K$-dimensional linear subspace. With $K=1$, the outputs of the autoencoder all lie along a straight line in the 2-dimensional data space. The original data are on a line, but not a straight line, and so can’t be matched perfectly.

b) Explain whether the points could be perfectly reconstructed with $K=1$ by some non-linear decoder: $f = g(h)$. Where $g$ could be an arbitrary function, perhaps represented by multiple neural network layers. Assume the encoder is still linear: $h = W^{(1)} x$.

Answer:
We could set the weights so that $h = x_1$. Then set:

\[
\begin{align*}
g_1(h) &= h = x_1 \\
g_2(h) &= \sqrt{1-h^2} = x_2,
\end{align*}
\]

thus perfectly reconstructing any input.

c) Explain whether the points could be perfectly reconstructed with $K=1$ by some non-linear encoder: $h = g(x)$. Where $g$ could again be an arbitrary function, perhaps represented by multiple neural network layers. Assume the decoder is still linear: $f = W^{(2)} h$.

Answer:
No matter how we set the non-linearity $g$ here, the outputs lie along a straight line, and so cannot match the original data.

Discussion of Q2:
Autoencoders are often presented with tied weights, where the transpose of the “encoding” weights are used for “decoding”. That’s partly due to the link to PCA, and partly because of links to other models and initialization procedures. Such as Hinton and Salakhutdinov’s (2006) Science paper on deep learning.

However, it can make sense for encoding and decoding to look quite different. Non-linear functions can restore non-linear data from a small compact description. However, linear transformations may be flexible enough to find a good representation of high-dimensional vectors (they have $K \times D$ free parameters, which is a lot if $D$ is big!). I like linear transformations: they’re simple and topology preserving. Non-linear autoencoders can “tear” the input space, mapping nearby inputs into far off parts of the hidden space, and then reassemble them. Encoding with a linear projection stops that from happening.

If you run out of things to do (I think most of the class actually have plenty to do), you could try to implement and fit some of the models mentioned above. For example, can you fit a
dataset as well as PCA using the ideas in Q2? Or can you create and fit a dataset lying on a 
low-dimensional manifold as in Q3? There’s probably not time to discuss or debug code in 
your tutorial groups. However, you can post your code attempts to the forum (with three 
backtics ` on the lines above and below your code) and I can give feedback and/or help.