Machine Learning and Pattern Recognition
Self-Study sheet 8

Instructor: Iain Murray

1. **Pre-processing for Bayesian linear regression and Gaussian processes:**

   We have a dataset of inputs and outputs $\{\mathbf{x}_n, y_n\}_{n=1}^N$, describing N preparations of cells from some lab experiments. The output of interest, y_n, is the standard deviation of the radii of the cells in preparation n. The first input feature of each preparation indicates whether the cells were created in lab `A`, `B`, or `C`. That is, $x_{1,n} \in \{\texttt{A}, \texttt{B}, \texttt{C}\}$. The other features are real numbers describing experimental conditions such as temperature and concentrations of chemicals and nutrients.

   (a) Describe how you might represent the first input feature and the outputs when learning a regression model to predict the standard deviation of cell radii in future preparations from these labs. Explain your reasoning.

   (b) Compare using the lab identity as an input to your regression (as you've discussed above), with two baseline approaches: i) Ignore the lab feature, treat the data from all labs as if they came from one lab; ii) Split the dataset into three parts one for lab `A`, one for `B`, and one for `C`. Then train three separate regression models.

   Discuss both simple linear regression and Gaussian process regression. Is it possible for these models to learn to emulate either or both of the two baselines?

   (c) There's a debate in the lab about how to represent the other input features: log-temperature or temperature, and temperature in Fahrenheit, Celsius or Kelvin? Also whether to use log concentration or concentration as inputs to the regression. Discuss ways in which these issues could be resolved.

Answer:

(a) The outputs, being standard deviations of something, are positive numbers. However, the predictions from a Gaussian process regression model, $p(y_* \mid \mathbf{x}_*, \mathcal{D})$, are Gaussian distributions, and so predict positive and negative numbers. We could instead train and predict the logs of the cell-radii standard deviations. Then our Gaussian predictions would be on quantities that can be positive and negative. Often taking the log of positive quantities makes their distribution less skewed and more Gaussian.

(Do not confuse the fact that each y_n is a standard deviation of something with the other standard deviations in a Gaussian process model, the function standard deviation σ_f, or noise standard deviation σ_n. You need to have a firm grasp of the ingredients in the models we've covered, so you can give clear and accurate explanations.)

The lab identity is a categorical variable. A standard way to encode such variables (seen very early on in the course) is a 1-of-M, or 'one-hot' encoding (also called 1-of-K and I'm sure other names). The feature is replaced with $M = 3$ binary features. All but one of the features are set to zero, and one of them is set to one, indicating whether the lab was `A`, `B`, or `C`.

Replacing the lab identity with a single feature in $\{1, 2, 3\}$ would not be a good answer in this situation. For linear regression models, this representation gives a model that assumes that, for a given experimental condition, the mean output

from lab `B` is always between the mean outputs from lab `A` and lab `C`. We haven't been given any information that would suggest this assumption is sensible.

For more advanced (non-examinable) ways of dealing with multiple related settings you could search for 'multi-task learning'.

(b) For linear regression: Ignoring the lab feature gives a model with fewer parameters, which is less prone to overfitting if there's very little data. Training three separate models triples the number of parameters; if there's lots of data it could learn a situation where the three labs lead to *completely* different linear relationships, although that seems unlikely. Fitting one linear regression model with a one-hot-encoding of the lab adds only three extra parameters. It can emulate the first baseline by setting all of the weights for the lab features to zero. It can't emulate the second baseline: it can only model a different constant offset to the target function for each lab. Using the lab could underfit the data if the influence of the labs on cell radii depends on the experimental conditions.

For Gaussian process (GP) regression: ignoring the lab feature could still avoid overfitting when there's very little data. If we learn kernel function lengthscales associated with the lab identity features, we can potentially emulate either baseline. Very long lengthscales mean that the lab feature has negligible effect on kernel values, and the lab identity is ignored. Zero lengthscales would mean that the model thinks the functions for the different labs are independent (zero covariance in the Gaussian distributions we construct), so we could train three separate GPs instead. Intermediate lengthscales say that the labs could behave slightly differently, but that the functions for each lab are related, which we'd hope if the experiment is reasonably reproducible!

(c) In some situations, doubling a positive quantity may have a similar-sized effect on the outcome, regardless of the starting value. That is, it's the log of the quantity that is approximately linearly-related to the output. In these situations taking the log of input features can be necessary to get a good fit with simple linear regression, and can require less data to get a good fit with GPs or fewer basis functions for linear regression.

It's sometimes unclear whether to take logs of positive features however. I would normally do so for concentrations, but I'm less sure about temperature (in Kelvin, so it's always positive). Therefore, we have multiple possible models, one for each set of feature transformations we want to consider. These could be compared by performance on a validation performance, or by marginal likelihood. Alternatively we could create a model with all versions of the features, and hope (perhaps with regularization) we'll learn some small weights or large lengthscales so that we'll ignore some of them. (We could also treat the best choice of transformation as an unknown parameter and perform MCMC over them.)

(Warning: models with redundant features can be non-identifiable, and hard to interpret. Without careful regularization they're more prone to numerical problems and overfitting.)

If we don't take logs of the temperatures, the units (F, C or K) just set an offset and scaling of the numbers. That will have no effect if we standardize our inputs to have zero mean and unit variance before fitting. Even if we don't standardize our features, the regression models can learn the same mapping from features to outputs for any offset and scale of the inputs. However, the regularization constants will have different effects for the different scalings of the inputs, which is why we normally standardize our inputs.

---

2. **Gaussian processes with non-zero mean:**

In the lectures we assumed that the prior over any vector of function values was zero mean: $\mathbf{f} \sim \mathcal{N}(0, K)$. We focussed on the covariance or kernel function $k(\mathbf{x}_i, \mathbf{x}_j)$ that evaluates the K_{ij} elements of the covariance matrix (or 'Gram matrix').

If we know in advance that the distribution of outputs should be centered around some other mean $\mathbf{m}$, we *could* put that into the model. Instead, we usually subtract the known mean $\mathbf{m}$ from the $\mathbf{y}$ data, and just use the zero mean model.

Sometimes we don't really know the mean $\mathbf{m}$, but look at the data to estimate it. A fully Bayesian treatment puts a prior on $\mathbf{m}$ and, because it's an unknown, considers all possible values when making predictions. A flexible prior on the mean vector could be another Gaussian process(!). Our model for our noisy observations is now:

$$\begin{aligned} \mathbf{m} &\sim \mathcal{N}(0, K_m), \quad K_m \text{ from kernel function } k_m, \\ \mathbf{f} &\sim \mathcal{N}(\mathbf{m}, K_f), \quad K_f \text{ from kernel function } k_f, \\ \mathbf{y} &\sim \mathcal{N}(\mathbf{f}, \sigma_n^2 \mathbb{I}), \quad \text{noisy observations.} \end{aligned}$$

Show that — despite our efforts — the function values $\mathbf{f}$ still come from a function drawn from a zero-mean Gaussian process (if we marginalize out $\mathbf{m}$). Identify the covariance function of the zero-mean process for f.

Identify the mean's kernel function k_m for two restricted types of mean: 1) An unknown constant $m_i = b$, with $b \sim \mathcal{N}(0, \sigma_b^2)$. 2) An unknown linear trend: $m_i = m(\mathbf{x}_i) = \mathbf{w}^\top \mathbf{x}_i + b$, with Gaussian priors $\mathbf{w} \sim \mathcal{N}(0, \sigma_w^2 \mathbb{I})$, and $b \sim \mathcal{N}(0, \sigma_b^2)$.

Answer:
The function f is the sum of two Gaussian processes. Any vector of values $\mathbf{f}$ is the sum of two independent Gaussian vectors: $\mathbf{m}$ and a zero-mean draw from $\mathcal{N}(0, K_f)$. Thus any set of values are marginally drawn from $\mathbf{f} \sim \mathcal{N}(0, K_m + K_f)$, corresponding to a function drawn from a zero mean Gaussian process with covariance function $k(\mathbf{x}_i, \mathbf{x}_j) = k_m(\mathbf{x}_i, \mathbf{x}_j) + k_f(\mathbf{x}_i, \mathbf{x}_j)$.

What are the statistics of a vector $\mathbf{m}$, with identical elements $m_i = b$, with $b \sim \mathcal{N}(0, \sigma_b^2)$? The vector is Gaussian, and each element has mean zero and marginal variance σ_b^2. The other elements of the covariance matrix are given by:

$$\text{cov}[m_i, m_j] = \mathbb{E}[m_i m_j] - \mathbb{E}[m_i]\mathbb{E}[m_j] = \mathbb{E}[b^2] = \sigma_b^2.$$

So $k_m(\mathbf{x}_i, \mathbf{x}_j) = \sigma_b^2$, and the resulting matrix K_m has every element set to σ_b^2. What we've learned: to add a random offset to a process, we can just add a constant to every element of the covariance matrix.

The covariance function for a linear model was derived in the lecture slides, $k(\mathbf{x}_i, \mathbf{x}_j) = \sigma_w^2 \mathbf{x}_i^\top \mathbf{x}_j + \sigma_b^2$. If in addition to a random constant offset we want a random linear trend, we further add $\sigma_w^2 \mathbf{x}_i^\top \mathbf{x}_j$ to each element K_{ij}.

A little knowledge is a dangerous thing: If you actually want to implement Gaussian processes with a linear trend or constant offset, read §2.7, pp27–28, of Rasmussen and Williams, *Gaussian Processes for Machine Learning*. There can be numerical problems with an obvious implementation using the covariance matrices above.

A final comment: a linear trend is usually physically unrealistic. The model confidently predicts an extreme output if you go to an extreme input location. Under the model, the best way to measure the linear trend is to evaluate the function at extreme input locations. These properties could cause problems if using the model for planning or optimization. It may be better to model a large-scale trend by adding a stationary kernel (such as the squared exponential) with a long lengthscale, rather than the linear kernel.

The GPML code `http://www.gaussianprocess.org/gpml/` makes it easy to try out different combinations of kernels. See `http://learning.eng.cam.ac.uk/carl/mauna/` for an example.

---

3. **PCA and supervised learning:**

Principal Components Analysis (PCA) is an unsupervised learning technique. Given only an $N \times D$ matrix of input features X we learn a $D \times K$ matrix V that can project the data down into K dimensions. We could call the new input features $Z = XV$, where each row $\mathbf{z}_n$ is a K-dimensional feature vector.

PCA is often applied as a pre-processing step before supervised learning. For example we could apply logistic regression using the transformed input features Z instead of X. We would fit the weights $\mathbf{w}$ in the model $P(y = 1 \mid \mathbf{z}, \mathbf{w}) = \sigma(\mathbf{w}^\top \mathbf{z})$, where $\sigma(z) = 1/(1 + \exp(-z))$ is the logistic sigmoid.

(I'm not including bias parameters to keep any discussion of equations simpler.)

Alternatively, we could create a neural network with a linear hidden layer $\mathbf{z}_n = V^\top \mathbf{x}_n$, where V is a matrix of free parameters to learn along with the output weights during neural network training. A logistic output unit, $P(y = 1 \mid \mathbf{z}, \mathbf{w}) = \sigma(\mathbf{w}^\top \mathbf{z})$, makes the neural network again make the same predictions as logistic regression applied to the transformed input features $Z = XV$. We can choose to have K hidden units, such that V is a $D \times K$ matrix just like in the PCA approach above.

Compare the two approaches above, saying what (if anything) they could be useful for. Suggest a modification to the neural network approach that could make it more useful.

Answer:
Both approaches reduce the input features to K dimensions before making predictions. However, without modifying the neural network approach, it would never make sense to apply it. The PCA approach could be useful. (If you hadn't figured that out, stop reading and have another go before looking at the answer.)

PCA fits the projection matrix V without looking at the training labels. Therefore only the K parameters $\mathbf{w}$ are fitted to the input-output relationship instead of the D parameters weed need without PCA. The model with fewer parameters is more restricted in the relationships it can learn, which makes it less prone to overfitting when there isn't much data, but could make it perform less well for large datasets. However, for moderate-sized datasets finding eigenvectors or an SVD with standard linear algebra routines is easier.

The neural network approach, as described, fits a classification rule $P(y = 1 \mid \mathbf{x}, \mathbf{w}, V) = \sigma(\mathbf{w}^\top V^\top \mathbf{x})$. This model is simply logistic regression with a length-D weight vector $V\mathbf{w}$. There is no restriction in the weights we can set, in fact we have $K+1$ times more free parameters than necessary. Thus this model doesn't introduce any extra flexibility, or reduce overfitting. Introducing the hidden layer just wastes computation and makes the model harder to interpret.

If we introduce non-linearities in the hidden layer, such as $\mathbf{z}_n = \sigma(V^\top \mathbf{x}_n)$, then we will expand the family of classifiers $P(y = 1 \mid \mathbf{x}, \mathbf{w}, V)$ that we can represent. Having more parameters, we are still more prone to overfitting than logistic regression. However, if we have enough data to learn the extra parameters, we may get better classification performance. The motivation for the neural network is the opposite of the similar-looking PCA approach!

---

4. **PCA and autoencoders:**

It's possible to perform unsupervised dimensionality reduction with neural networks instead of implementing PCA by computing eigenvectors or a singular value decomposition (SVD). (The previous question talked about attempting *supervised* dimensionality reduction with neural networks, finding a hidden representation from which we can predict labels well.)

An autoencoder is a neural network with D inputs $\mathbf{x}$ and D outputs $\mathbf{y}$. The goal is to find parameters where the outputs are close to the inputs. The parameters are optimized to minimize the square difference between the two vectors $\sum_n (\mathbf{x}_n - \mathbf{y}_n)^\top (\mathbf{x}_n - \mathbf{y}_n)$.

(a) Compare the autoencoder approach to PCA for an auto-encoder network with K linear hidden units, $\mathbf{z} = V^\top \mathbf{x}$, and a linear output layer $\mathbf{y} = W\mathbf{z}$.

(b) Explain why a non-linear autoencoder, where non-linearities are applied to the layer(s) of hidden units, might work better than a linear autoencoder or PCA.

Answer:

(a) PCA finds a linear projection that can be reconstructed with the minimum possible square error for a linear projection. Optimizing the neural network will also perform this job. There's a lot of neural network software that will let us use stochastic gradient methods to fit large datasets.

In PCA the columns of V are orthogonal, and are ordered: the directions where the data varies most appear first. Without enforcing these constraints, the neural network won't find the same solution as its cost function doesn't define a unique optimum.

In the autoencoder the reconstruction is $\mathbf{y} = W\mathbf{z}$. In PCA we don't need a new matrix: $\mathbf{y} = V\mathbf{z}$. That tells us we can enforce $W = V$, and tie together these weights during training in the neural network approach.

(b) Consider a set of points lying along a semi-circle, which is a one-dimensional manifold. We could represent any of these points with an angle, a single number $z \in [0, \pi]$. However, PCA or a linear autoencoder would require $K = 2$ dimensions to perfectly reconstruct these points, because they don't lie on a straight line. A non-linear autoencoder could in principle contain a layer with only one hidden unit, from which a point could be reconstructed. In general, data may lie close to a K-dimensional manifold, but that manifold might not be a K-dimensional linear hyper-plane. Such data would requiring more than K principle components, but might only need K hidden units in a non-linear network.

(Fitting non-linear autoencoders can be a hard optimization challenge however.)
