SAMBA course: Mathematics and Algorithms of Machine Learning

James Hook

October 21, 2019

About this course

Machine Learning is a topic at the intersection of Mathematics and Computer Science, with a wide range of applications throughout industry and society. Machine Learning algorithms use data to 'learn' how to make predictions or decisions without needing to be explicitly programmed how to do these tasks. This course focuses on supervised learning, which is a type of prediction problem. There are already excellent software packages available that enable anyone to run powerful supervised learning algorithms for themselves. However if we want to be able to properly interpret the output of these algorithms, judge the suitability of different approaches or develop new techniques for particular problems then it is essential to develop a deep understanding of the underlying mathematics as well as the details of these algorithm's implementations. The aim of this course is to do just that! Along the way introducing students to two of the most powerful and widely used machine learning tools, namely deep neural networks and Gaussian processes. This course is designed for first year Mathematics PhD students and although it only assumes a little prior knowledge of Probability, Statistics and Analysis, it should be quite challenging because of the fast pace, breadth of topics covered and the Python programming coursework assignments.

There are a total of five coursework assignments. The first four are guided programming assignments that require you to write code from scratch for various Machine Learning models. The final coursework assignment is a group project to explore different machine learning techniques on data from real life problems and present your findings in a short talk for the group. Coursework assignments 1 and 2 will not be formally assessed and are intended to help students with less programming experiance get up to speed.

We recommend that you use your `https://notebooks.azure.com` account for Python programming on this course. Each lecture/section has a notebook associated with it. These notebooks are available at `https://notebooks.azure.com/jlh75/projects/samba-ml-course`. We encourage you to experiment with the code in these notebooks, trying different datasets and models for example. Can you find alternative approaches or problems that give better or more interesting results that those presented to you in the lecture? The notebook code will also be a very useful reference source during your coursework assignments, especially if you do not have much Python programming experience.

If you do not have previous experience using Python then you should make sure that you familiarize yourself a little before the course starts. E.g. check out `https://www.learnpython.org/`. Try to familiarize yourself with using the `numpy` package for mathematical operations and the `matplotlib.pyplot` package for plotting. You will need to know how to manipulate lists and arrays and how to write functions. The coursework exercises are designed to get progressively harder in terms of the programming so you don't need to be an absolute expert at the start but please try to make sure you understand the basics before the lectures begin! On Friday 27th September 2019 there is a SAMBA workshop on learning Python, R, Latex and MATLAB which you should make sure you attend. If you find that you are struggling with the programming side of the course then you can get extra help through the Research Software Engineering group `https://arc-lessons.github.io/courses/00_schedule.html` or contact `rse-support@bath.ac.uk`.

These notes layout the bare bones of the material to be covered in lectures and should be supplemented by your own notes as well as from reading some of the textbooks below. Depending on how much time is available and the groups preferences there will be additional lectures on further topics including decision trees, unsupervised learning and reinforcement learning.

References

[1] Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman. *The elements of statistical learning: data mining, inference, and prediction, 2nd Edition.* Springer series in statistics. Springer, 2009.

[2] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective.* The MIT Press, 2012.

[3] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning).* The MIT Press, 2005.

[4] David J. C. MacKay. *Information Theory, Inference & Learning Algorithms.* Cambridge University Press, New York, NY, USA, 2002.

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* The MIT Press, 2016.

[6] Alan Julian Izenman. *Modern Multivariate Statistical Techniques: Regression, Classification, and Manifold Learning.* Springer Publishing Company, Incorporated, 1 edition, 2008.

[7] Pankaj Mehta, Marin Bukov, Ching-Hao Wang, Alexand re G. R. Day, Clint Richardson, Charles K. Fisher, and David J. Schwab. A high-bias, low-variance introduction to Machine Learning for physicists. , 810:1–124, May 2019.

[8] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251 – 257, 1991.

[9] R. Adler. *The Geometry of Random Fields.* Society for Industrial and Applied Mathematics, 2010.

Contents

Week Beginning	Monday 10.15	Wednesday 9.15
30th September 2019	Introduction	Classification and KNN
7th October 2019	Regression and Linear Regression	Coursework 1 Lab
14th October 2019	Logistic Regression I	Logistic Regression II + Coursework 1 Model Answers
21st October 2019	Neural Networks I	Coursework 2 Lab
28th October 2019	Neural Networks II + Coursework 2 Model Answers	Neural Networks III
4th November 2019	Neural Networks IV	Coursework 3 Lab
11th October 2019	Neural Networks V	Gaussian Processes I (Coursework 3 Hand In)
18th October 2019	Gaussian Processes II + Coursework 3 Model Answers	Gaussian Processes III
25th October 2019	Gaussian Processes IV	Coursework 4 Lab
2nd December 2019	Gaussian Processes V	Further Topics (Coursework 4 Hand In)
9nd December	Group Presentations I + Coursework 4 Model Answers	Group Presentations II

Table 1: Provisional Timetable

1 Introduction to Supervised Learning

1.1 Classification

In a *binary classification problem* we are given a *training set* of n labeled data points

$$\left(\boldsymbol{x}_i \in \mathbb{R}^d, y_i \in \{0,1\}\right)_{i=1}^n.$$

The problem is to use this data to formulate a function $p : \mathbb{R}^d \mapsto \{0,1\}$ for predicting the labels of previously unseen points.

Note that in general we aren't told what form the function p should take or even told how to measure the accuracy of a prediction. We also don't know what a previously unseen point might look like. In order to approach solving a classification problem we need to make some assumptions to answer these questions. In practical situations we will hopefully be able to find answers to these questions by looking carefully at where the data comes from.

Define the *object-feature matrix* $X \in \mathbb{R}^{n\times d}$ by $x_{ij} = (\boldsymbol{x}_i)_j$ and the labels vector $\boldsymbol{y} \in \{0,1\}^n$.

1.1.1 K-nearest neighbours

K-nearest neighbours is a very simple and intuitive model for classification problems. The idea is to base the prediction for a new point on the labels of the nearby training points.

For some $k > 0 \in \mathbb{N}$ define $f : \mathbb{R}^d \mapsto [0,1]$ by

$$f(\boldsymbol{x}) = 1/k \sum_{i\in K(\boldsymbol{x})} y_i,$$

where $K(\boldsymbol{x}) = \{i_1, \dots, i_k\}$ are the indices of the k closest points to $\boldsymbol{x}$ in the training set.

Then define $p : \mathbb{R}^d \mapsto \{0,1\}$ by

$$p(\boldsymbol{x}) = \begin{cases} 1 & \text{if } f(\boldsymbol{x}) \geq 0.5 \\ 0 & \text{otherwise.} \end{cases}$$

Note that the model predicts that a point x is in class 1 if and only if half or more of its k-nearest neighbours in the training set are class 1.

Suppose we have a real life problem that we want to solve with k-nearest neighbours. How do we choose k? and how can we tell if our predictive function is any good?

1.1.2 Training, validation and test data

We will randomly split the training data points into three different subsets.

- Training data. This should be where we put most of the training data. This data will be used just like before.
- Validation data. This should be a small (say 10% or so) subset of the data. We will use this data to select a value for k.
- Test data. This should be a small (say 10% or so) subset of the data. This data is held back to provide an unbiased estimate of the final model's accuracy.

1.1.3 'Accuracy' of binary classifiers

There are a bewildering array of measures for evaluating the performance of a binary classifier. Each point in the test set can be classed as either true positive , false positive, true negative or false negative.

Let TP, FP, TN and FN denote the number of test points in each class. Then some popular measures are as follows.

- Accuracy=(TP+TN)/(TP+FP+TN+FN)
- Precision=TP/(TP+FP)
- Recall=TP/(TP+FN)
- F1 Score=2TP/(2TP+FP+FN)

1.1.4 See Example 1

In this example I use `sklearn.neighbors.KNeighborsClassifier` model to make predictions from a 2d classification problem. I show how to choose the optimal value for k using validation and cross validation. I also show how to write a class that neatly combines these functions together.

Experiment with altering the test problem as well as using different classification models from `sklearn`.

1.2 Regression

In a *regression* problem we are given a *training set* of n valued data points

$$\left(\boldsymbol{x}_i \in \mathbb{R}^d, y_i \in \mathbb{R}\right)_{i=1}^{n}.$$

The problem is to use this data to formulate a function $f : \mathbb{R}^d \mapsto \mathbb{R}$ for predicting the values of previously unseen points.

Define the *object-feature matrix* $X \in \mathbb{R}^{n\times d}$ by $x_{ij} = (\boldsymbol{x}_i)_j$ and the target vector $\boldsymbol{y} \in \mathbb{R}^n$.

1.2.1 Linear regression

We choose to measure the accuracy of a prediction function f on a dataset $\left(\boldsymbol{x}_i \in \mathbb{R}^d, y_i \in \mathbb{R}\right)_{i=1}^{n}$ by the *mean squared error*, which is defined by

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(f(\boldsymbol{x}_i) - y_i)^2.$$

Next we choose to restrict ourselves to functions of the following form. For $\boldsymbol{w} \in \mathbb{R}$ define $f : \mathbb{R}^d \mapsto \mathbb{R}$ by

$$f(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x},$$

we call $\boldsymbol{w}$ the *weights vector*.

We choose the weights vector to minimize the mean squared error on the training data

$$\boldsymbol{w} = \arg\min_{\boldsymbol{w}\in\mathbb{R}^d}\left(\sum_{i=1}^{n}\left(f(\boldsymbol{x}_i) - y_i\right)^2\right),$$

and we use the resulting $f : \mathbb{R}^d \mapsto \mathbb{R}$ as our predictive function.

1.2.2 Solution of least squares problem

Note that

$$\mathrm{MSE}(\boldsymbol{w}) = \sum_{i=1}^{n}\left(f(\boldsymbol{x}_i) - y_i\right)^2 = \|X\boldsymbol{w} - \boldsymbol{y}\|_2^2,$$

then

$$\frac{\partial \mathrm{MSE}(\boldsymbol{w})}{\partial \boldsymbol{w}} = 2X^\top(X\boldsymbol{w} - \boldsymbol{y}).$$

So (provided $X^\top X$ is inevitable) MSE is minimized by

$$\boldsymbol{w} = \left(X^\top X\right)^{-1}X^\top \boldsymbol{y}.$$

1.2.3 Tikinov regularization

In cases where d is very large we might want solutions that average over as many of the different features as possible, rather than basing their prediction on just a small subset of the features, which could be unreliable because of noise. One way to do this is Tikinov regularization, also called ℓ-2 regularization.

Choose $\lambda > 0 \in \mathbb{R}$, then set

$$\boldsymbol{w} = \arg\min_{\boldsymbol{w}\in\mathbb{R}^d} \left(\|X\boldsymbol{w} - \boldsymbol{y}\|_2^2 + \lambda^2\|\boldsymbol{w}\|_2^2\right).$$

Let

$$F(\boldsymbol{w}) = \|X\boldsymbol{w} - \boldsymbol{y}\|_2^2 + \lambda^2\|\boldsymbol{w}\|_2^2,$$

then

$$\frac{\partial F(\boldsymbol{w})}{\partial \boldsymbol{w}} = 2X^\top(X\boldsymbol{w} - \boldsymbol{y}) + 2\lambda^2\boldsymbol{w}.$$

So F is minimized by

$$\boldsymbol{w} = \left(X^\top X + \lambda^2 I\right)^{-1} X^\top \boldsymbol{y}.$$

1.2.4 Probabilistic interpretation

Least squares linear regression with Tikinov regularization allows a probabilistic interpretation as follows. Suppose that $\boldsymbol{y} = X\boldsymbol{w} + \boldsymbol{z}$, where the *noise terms* $(z_i)_{i=1}^n$ are i.i.d. $(0,1)$ Gaussians. Then the *negative log likelihood* of a weight vector is given by

$$NLL(\boldsymbol{w}) = -\log\rho(\boldsymbol{y}|\boldsymbol{w}, X) = n/2\log(2\pi) + \|X\boldsymbol{w} - \boldsymbol{y}\|_2^2/2.$$

Now suppose further that the the weight vector is also a random variable with $(w_j)_{j=1}^d$ i.i.d. $(0, 1/\lambda^2)$ Gaussians. We call $\rho(\boldsymbol{w})$ the *prior* and $\rho(\boldsymbol{w}|X,\boldsymbol{y})$ the *posterior*. From Bayes theorem we have that

$$\rho(\boldsymbol{w}|X,\boldsymbol{y}) = \frac{\rho(\boldsymbol{y}|\boldsymbol{w}, X)\rho(\boldsymbol{w})}{\rho(\boldsymbol{y}|X)}.$$

Therefore the *negative log posterior* of a weight vector is given by

$$NLPP(\boldsymbol{w}) = -\log\rho(\boldsymbol{w}|\boldsymbol{y}, X) = \|X\boldsymbol{w} - \boldsymbol{y}\|_2^2/2 + \lambda^2\|\boldsymbol{w}\|_2^2/2 + C,$$

where C is a constant that is independent of $\boldsymbol{w}$.

From this point of view it is not possible to completely determine $\boldsymbol{w}$ from the training data as there will always be some uncertainty in the posterior distribution. If we want to work with a single value of $\boldsymbol{w}$ then we can either take the maximum likelihood estimate (which corresponds to least squares regression without regularization) or the maximum *a posteri* estimate (which corresponds to regularized least squares regression). We will return to this Bayesian approach to machine learning in more detail in Section 3 of the course.

1.2.5 See Example 2

In this example I use write my own code to do linear regression and Tikinov regularized linear regression. I test the code on randomly generated example problems.

Experiment with altering the test problem. What is the worst number of features to number of data-points ratio that Tikinov regularized regression can handle? What if you set $y =$ some non-linear function of x plus noise. Is linear regression still able to do a good job? Experiment with different regression models from `sklearn`.

2 Neural Networks

2.1 Logistic regression

Logistic regression is a model that uses a differentiable loss function to fit a linear decision boundary for a binary classification problem. Logistic regression is important because a) it is an easily interpreted method that is very widely used and often gives good results in practical problems and b) it can be thought of as the most simple possible example of a neural network.

2.1.1 Linear decision boundary

For $\boldsymbol{w} \in \mathbb{R}^d$ and $b \in \mathbb{R}$ define $f : \mathbb{R}^d \mapsto \mathbb{R}$ by

$$f(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x} + b,$$

then define $p : \mathbb{R}^d \mapsto \{0, 1\}$, by

$$p(\boldsymbol{x}) = \begin{cases} 1 & \text{if } f(\boldsymbol{x}) \geq 0 \\ 0 & \text{otherwise.} \end{cases}$$

Now set

$$\boldsymbol{w}, b = \arg \max_{\boldsymbol{w} \in \mathbb{R}^d, b \in \mathbb{R}} \left(\sum_{i=1}^{n} \begin{cases} 1 & \text{if } p(\boldsymbol{x}_i) = y_i \\ 0 & \text{otherwise} \end{cases} \right).$$

This model chooses the optimal hyperplane for separating the two classes in the training data. However there is no remotely efficient method for computing $\boldsymbol{w}$ and b, especially in the case when d is large.

2.1.2 Logistic loss function

In general we can choose the parameters of a parametric ML model by minimizing the sum a loss function on the training set. In the case of a linear decision boundary this becomes

$$\boldsymbol{w}, b = \arg \min_{\boldsymbol{w} \in \mathbb{R}^d, b \in \mathbb{R}} \left(\sum_{i=1}^{n} L\big(f(\boldsymbol{x}_i), y_i\big) \right).$$

E.g. for the optimal linear decision boundary we use

$$L(f, y) = \begin{cases} 0 & \text{if } f \geq 0 \text{ and } y = 1 \text{ or } f < 0 \text{ and } y = 0 \\ 1 & \text{otherwise.} \end{cases}$$

For logistic regression we use the logistic loss function

$$L(f, y) = y \log(1 + e^{-f}) + (1 - y) \log(1 + e^{f}).$$

The logistic loss function has a probabilistic interpretation as follows. Suppose that the labels of the training data are random variables with

$$\mathbb{P}\big(y_i = 1 \mid \boldsymbol{w}, b, \boldsymbol{x}_i\big) = \sigma(\boldsymbol{w}^\top \boldsymbol{x} + b),$$

where $\sigma : \mathbb{R} \mapsto [0, 1]$ is the *logistic sigmoid* defined by

$$\sigma(z) = \frac{1}{1 + e^{-z}}.$$

Then

$$L\big(\boldsymbol{w}^\top \boldsymbol{x}_i + b, y_i\big) = -\log \mathbb{P}(y_i \mid \boldsymbol{w}, b, \boldsymbol{x}_i)$$

and

$$\sum_{i=1}^{n} L\big(\boldsymbol{w}^\top \boldsymbol{x}_i + b, y_i\big) = \sum_{i=1}^{n} -\log \mathbb{P}(y_i \mid \boldsymbol{w}, b, \boldsymbol{x}_i) = -\log \mathbb{P}(\boldsymbol{y} \mid \boldsymbol{w}, X, b) = NLL(\boldsymbol{w}, b | X, \boldsymbol{y}),$$

which we call the *negative log likelihood.*

2.1.3 Convexity of NLL function

A function $f : \mathbb{R}^d \mapsto \mathbb{R}$ is convex if

$$f(\lambda \boldsymbol{x} + (1-\lambda)\boldsymbol{y}) \leq \lambda f(\boldsymbol{x}) + (1-\lambda) f(\boldsymbol{y}),$$

for all $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^d$ and $\lambda \in (0,1)$.

Convex functions are easy to minimize because they either have a single minimum point or they have multiple minima that the
MSElves form a convex set.

Theorem 2.1. *For any $X \in \mathbb{R}^{n\times d}$ and $y \in \{0,1\}^n$ the function $NLL(\boldsymbol{w}, \boldsymbol{b}|X, \boldsymbol{y})$ is convex.*

2.1.4 Gradient descent

We seek

$$\boldsymbol{w}, b = \arg \min_{\boldsymbol{w}\in\mathbb{R}^d, b\in\mathbb{R}} NLL(\boldsymbol{w}, b|X, \boldsymbol{y}).$$

To use gradient descent we need the derivatives which are given by

$$\begin{aligned}
\frac{\partial NLL(\boldsymbol{w}, b|X, \boldsymbol{y})}{\partial \boldsymbol{w}} &= \sum_{i=1}^{n} \left(\frac{-y_i \boldsymbol{x}_i^\top e^{-(\boldsymbol{w}^\top \boldsymbol{x}_i + b)}}{1 + e^{-(\boldsymbol{w}^\top \boldsymbol{x}_i + b)}} + \frac{(1-y_i)\boldsymbol{x}_i^\top e^{\boldsymbol{w}^\top \boldsymbol{x}_i + b}}{1 + e^{\boldsymbol{w}^\top \boldsymbol{x}_i + b}} \right), \\
\frac{\partial NLL(\boldsymbol{w}, b|X, \boldsymbol{y})}{\partial b} &= \sum_{i=1}^{n} \left(\frac{-y_i e^{-(\boldsymbol{w}^\top \boldsymbol{x}_i + b)}}{1 + e^{-(\boldsymbol{w}^\top \boldsymbol{x}_i + b)}} + \frac{(1-y_i) e^{\boldsymbol{w}^\top \boldsymbol{x}_i + b}}{1 + e^{\boldsymbol{w}^\top \boldsymbol{x}_i + b}} \right).
\end{aligned}$$

Algorithm 1 Gradient Descent

```
1: Start with a function $F$ to optimise with respect to $\boldsymbol{\theta}$
2: Choose an initial point $\boldsymbol{\theta}_0$ and a step size $\eta$
3: for $k = 1, 2, \ldots$ until convergence do
4:     $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta \nabla F(\boldsymbol{\theta}_k)$
5: end for
```

2.1.5 Newton's method

We can also solve

$$\boldsymbol{w}, b = \arg \min_{\boldsymbol{w}\in\mathbb{R}^d, b\in\mathbb{R}} NLL(\boldsymbol{w}, b|X, \boldsymbol{y}),$$

using Newton's method. To do this it is convenient to incorporate the bias parameter into the weights vector by adding a column of 1s onto the object-feature matrix. This results in a $d+1$ dimensional problem with no bias term, where w_{d+1} plays the role of b in the original problem. Newton's method uses the Hessian matrix $H \in \mathbb{R}^{(d+1)\times(d+1)}$, which is defined by

$$h_{ij} = \frac{\partial^2 NLL(\boldsymbol{w}|X, \boldsymbol{y})}{\partial w_i \partial w_j}.$$

Algorithm 2 Newton's Method for Optimization

```
1: Start with a function $F$ to optimise with respect to $\boldsymbol{\theta}$
2: Choose an initial point $\boldsymbol{\theta}_0$
3: for $k = 1, 2, \ldots$ until convergence do
4:     $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - H(\boldsymbol{\theta}_k)^{-1} \nabla F(\boldsymbol{\theta}_k)$
5: end for
```

It is an exercise for the reader to derive an equation for the Hessian matrix and then implement Logistic regression with Newton's method (See Coursework 2, Q4).

2.1.6 Logistic regression

For $\boldsymbol{w} \in \mathbb{R}^d$ and $b \in \mathbb{R}$ define $a : \mathbb{R}^d \mapsto \mathbb{R}$ by

$$a(\boldsymbol{x}) = \sigma(\boldsymbol{w}^\top \boldsymbol{x} + b).$$

Now set

$$\boldsymbol{w}, b = \arg \min_{\boldsymbol{w} \in \mathbb{R}^d, b \in \mathbb{R}} NLL(\boldsymbol{w}, b | X, \boldsymbol{y}).$$

Then define $p : \mathbb{R}^d \mapsto \{0, 1\}$, by

$$p(\boldsymbol{x}) = \begin{cases} 1 & \text{if } a(\boldsymbol{x}) \geq 0.5 \\ 0 & \text{otherwise.} \end{cases}$$

Note that the predictions of logistic regression also allow a probabilistic interpretation by $a(\boldsymbol{x}) = \mathbb{P}(y = 1 \mid \boldsymbol{x}, \boldsymbol{w}, b)$ i.e. the probability that the new point is class one given our choice of $\boldsymbol{w}$ and b. Note that this is not the same as $\mathbb{P}(y = 1 \mid \boldsymbol{x}, X, \boldsymbol{y})$, i.e. the probability that the new point is class one given the training data - for that we would need to use Bayesian Logistic Regression. See e.g. [4].

2.1.7 See Example 3

In this example I use the `sklearn.linear_model.LogisticRegression` model to make predictions for some 2d classification problems.

Can you engineer a new feature (i.e. add a new column to the matrix X such that x_{i3} = some function of x_{i1} and x_{12}) which helps logistic regression solve the second example problem?. Experiment with altering the test problems and experiment with different classification models from `sklearn`.

2.2 Neural networks

Biological neural networks (like your brain!) consist of billions of individual neurons connected in a vast network. Roughly speaking each neuron receives input from the neighbouring neurons it is connected to and 'fires' (produces an electrical signal which is passed down to further neighbours) when the sum of the signals it receives passes a certain threshold. In this way we can think of each individual neuron as a logistic regression model that decides whether or not to fire based on its inputs. Taking this idea further we can think of the brain as a huge network of logistic regression models feeding into each other.
This is the idea behind artificial neural networks. We arrange neurons into distinct layers. Each layer takes inputs from the layer below and passes output to the layer above. Data is fed into the bottom of the network and the output or prediction comes out of the top.

2.2.1 Fully connected logistic feed forwards neural network

A fully connected logistic feed forwards neural network with d-dimensional input consists of

- Hidden layer sizes: $h_1, h_2 \ldots, h_{L-1} \in \mathbb{N}$,
- Weights

$$\begin{aligned} &W^{(0)} \in \mathbb{R}^{h_1 \times d} \\ &W^{(1)} \in \mathbb{R}^{h_2 \times h_1} \\ &\ldots \\ &W^{(L-2)} \in \mathbb{R}^{h_{L-1} \times h_{L-2}} \\ &W^{(L-1)} \in \mathbb{R}^{1 \times h_{L-1}}, \end{aligned}$$

- Biases: $b^{(0)} \in \mathbb{R}^{h_1}$, $b^{(1)} \in \mathbb{R}^{h_2}$, $\ldots b^{(L-2)} \in \mathbb{R}^{h_{L-1}}, b^{(L-1)} \in \mathbb{R}$.

Define the *activations*

$$\begin{aligned}
&a_0 : \mathbb{R}^d \mapsto \mathbb{R}^d \\
&a_1 : \mathbb{R}^d \mapsto \mathbb{R}^{h_1} \\
&\dots \\
&a_{L-1} : \mathbb{R}^d \mapsto \mathbb{R}^{h_L} \\
&a_L : \mathbb{R}^d \mapsto \mathbb{R},
\end{aligned}$$

by

$$a_0(\boldsymbol{x}) = \boldsymbol{x}$$

and

$$a_k(\boldsymbol{x}) = \sigma\big(W^{(k-1)} a_{k-1}(\boldsymbol{x}) + b^{(k-1)}\big),$$

for $k = 1, \dots, L$.
Also define

$$z_k(\boldsymbol{x}) = W^{(k-1)} a_{k-1}(\boldsymbol{x}) + b^{(k-1)},$$

for $k = 1, \dots, L$.
We set

$$W, b = \arg\min_{W,b} NLL(W, b|X, \boldsymbol{y}),$$

where

$$NLL(W, b|X, \boldsymbol{y}) = \sum_{i=1}^{n} L\big(z_L(\boldsymbol{x}_i), y_i\big),$$

where L is the logistic loss function.

Then define $p : \mathbb{R}^d \mapsto \{0, 1\}$ by

$$p(\boldsymbol{x}) = \begin{cases} 1 & \text{if } a_L(\boldsymbol{x}) \geq 0.5, \text{ equivalently if } z_L(\boldsymbol{x}) \geq 0 \\ 0 & \text{otherwise.} \end{cases}$$

Just like Logistic Regression we can interpret the activation $a_L(\boldsymbol{x})$ as the probbaility that $\boldsymbol{x}$ is class 1 given our choice of weights and biases.

2.2.2 Back propagation

Backpropogation is the name given to the process of computing the derivative of the loss function of a neural network on a training point. Despite its fancy name this is nothing more than repeated application of the chain rule.

We have

$$L(z_L, y) = y \log(1 + e^{-z_L}) + (1 - y) \log(1 + e^{z_L}),$$

so that

$$\frac{\partial L(z_L, y)}{\partial z_L} = \frac{-y e^{-z_L}}{1 + e^{-z_L}} + \frac{(1 - y) e^{z_L}}{1 + e^{z_L}},$$

and

$$\begin{aligned}
\frac{\partial L(z_L, y)}{\partial \boldsymbol{z}_j} &= \frac{\partial L(z_L, y)}{\partial z_{j+1}} \frac{\partial z_{j+1}}{\delta z_j} \\
&= \frac{\partial L(z_L, y)}{\partial z_{j+1}} \frac{\partial z_{j+1}}{\delta a_j} \frac{\partial a_j}{\delta z_j} \\
&= \frac{\partial L(z_L, y)}{\partial z_{j+1}} W^{(j)} \circ \big(a_j \circ (\underline{1} - a_j)\big)
\end{aligned}$$

for $j = 1, \dots, L - 1$.

Therefore

$$\frac{\partial L(z_L, y)}{\partial W_{ik}^{(j)}} = \left(a_j \frac{\partial L(z_L, y)}{\partial z_{j+1}} \right)_{ik}$$

for $i = 1, \dots, h_{j+1}$, $k = 1, \dots, h_j$ and

$$\frac{\partial L(z_L, y)}{\partial b^{(j)}} = \frac{\partial L(z_L, y)}{\partial z_{j+1}}.$$

2.2.3 Stochastic gradient descent

In applications when we have a huge number of training data points it can be very expensive to compute the gradient exactly as we need to compute the gradient of the loss of each data point separately then add them together

$$\frac{\delta NLL(W|X, \boldsymbol{y})}{\delta W} = \sum_{i=1}^{n} \frac{\partial L\big(z_L(\boldsymbol{x}_i), y_i\big)}{\partial W}.$$

Instead let t be a uniformly distributed on $\{1, 2, \dots, n\}$ then

$$\frac{\partial L\big(z_L(\boldsymbol{x}_t), y_t\big)}{\partial W}$$

is a random variable with

$$\mathbb{E}\left[\frac{\partial L\big(z_L(\boldsymbol{x}_t), y_t\big)}{\partial W} \right] = \frac{\delta NLL(W|X, \boldsymbol{y})}{\delta W}.$$

We can use this sampled gradient in place of the true gradient and provided the step size is chosen small enough the algorithm will still converge.

Algorithm 3 Stochastic Gradient Descent

```
1: Start with a function $F = \sum_{i=1}^{n} F_i$ to optimize with respect to $\boldsymbol{\theta}$
2: Choose an initial point $\boldsymbol{\theta}_0$ and a step size $\eta$
3: for $k = 1, 2, \dots$ until convergence do
4:     draw $t(k)$ uniformly at random from $\{1, 2, \dots, n\}$
5:     $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta \nabla F_i(\boldsymbol{\theta}_k)$
6: end for
```

A common strategy is to randomly select a small subset or batch of training points rather than a single point to sample the gradient. This is called *batch learning*. Provided the sampled gradient has expectation equal to the true gradient then SGD will converge for small enough step size.

2.2.4 See Example 4

In this example I write code for neural networks (without any stopping condition) and test them on some 2d classification problems. I also make a 'movie' showing how the network evolves whilst it is trained. I may or may not make this example hidden whilst you are doing coursework 3.

Have a play with the interactive demo on `https://playground.tensorflow.org`.

2.3 Neural networks for regression and multi-class classification

The neural network we saw in the previous section can be adapted in a number of ways. Before we look at any of these adaptations in detail we first look at a more general formulation of a neural network.

2.3.1 More general fully connected feed forwards neural network

A fully connected feed forwards neural network with d dimensional input and m dimensional output consists of

- Hidden layer sizes: $h_1, h_2 \dots, h_{L-1} \in \mathbb{N}$,
- Weights

$$\begin{aligned}
&W^{(0)} \in \mathbb{R}^{h_1 \times d} \\
&W^{(1)} \in \mathbb{R}^{h_2 \times h_1} \\
&\dots \\
&W^{(L-2)} \in \mathbb{R}^{h_{L-1} \times h_{L-2}} \\
&W^{(L-1)} \in \mathbb{R}^{1 \times h_{L-1}},
\end{aligned}$$

- Biases: $b^{(0)} \in \mathbb{R}^{h_1}$, $b^{(1)} \in \mathbb{R}^{h_2}$, $\dots b^{(L-2)} \in \mathbb{R}^{h_{L-1}}, b^{(L-1)} \in \mathbb{R}$,
- Activation functions: $f^{(1)}, \dots, f^{(L)} : \mathbb{R} \mapsto \mathbb{R}$,
- Loss function: $L : \mathbb{R}^m \times \text{domain}(y) \mapsto \mathbb{R}$.

Define the *activations*

$$\begin{aligned}
&a_0 : \mathbb{R}^d \mapsto \mathbb{R}^d \\
&a_1 : \mathbb{R}^d \mapsto \mathbb{R}^{h_1} \\
&\dots \\
&a_{L-1} : \mathbb{R}^d \mapsto \mathbb{R}^{h_L} \\
&a_L : \mathbb{R}^d \mapsto \mathbb{R},
\end{aligned}$$

by

$$a_0(\boldsymbol{x}) = \boldsymbol{x}$$

and

$$a_k(\boldsymbol{x}) = f^{(k)}\big(W^{(k-1)} a_{k-1}(\boldsymbol{x}) + b^{(k-1)}\big),$$

for $k = 1, \dots, L$ and also define

$$z_k = W^{(k-1)} a_{k-1}(\boldsymbol{x}) + b^{(k-1)},$$

for $k = 1, \dots, L$.

We set

$$W, b = \arg\min_{W,b} NLL(W, b | X, \boldsymbol{y}),$$

where

$$NLL(W, b | X, \boldsymbol{y}) = \sum_{i=1}^{n} L\big(z_L(\boldsymbol{x}_i), y_i\big).$$

2.3.2 Back propagation

The derivative of the loss w.r.t. z_L will depend on our choice of L but the rest of the back propagation formula is easy to work out in general

$$\begin{aligned}
\frac{\partial L(z_L, y)}{\partial \boldsymbol{z}_j} &= \frac{\partial L(z_L, y)}{\partial z_{j+1}} \frac{\partial z_{j+1}}{\delta z_j} \\
&= \frac{\partial L(z_L, y)}{\partial z_{j+1}} \frac{\partial z_{j+1}}{\delta a_j} \frac{\partial a_j}{\delta z_j} \\
&= \frac{\partial L(z_L, y)}{\partial z_{j+1}} W^{(j)} \circ \frac{df^{(j)}(z_j)}{dz_j}
\end{aligned}$$

for $j = 1, \dots, L-1$ and like before

$$\frac{\partial L(z_L, y)}{\partial W^{(j)}_{ik}} = \left(a_j \frac{\partial L(z_L, y)}{\partial z_{j+1}} \right)_{ik}$$

for $i = 1, \dots, h_{j+1}$, $k = 1, \dots, h_j$ and

$$\frac{\partial L(z_L, y)}{\partial b^{(j)}} = \frac{\partial L(z_L, y)}{\partial z_{j+1}}.$$

2.3.3 Neural networks for regression

To solve a regression problem with a neural network we use the squared error loss function

$$L(z_L, y) = \frac{1}{2}(z_L - y)^2.$$

2.3.4 Neural networks for multi-class classification

In a *multiclass classification problem* with $k \in \mathbb{N}$ classes we are given a *training set* of n labeled data points

$$\big(\boldsymbol{x}_i \in \mathbb{R}^d, y_i \in \{0, 1, \dots, k-1\}\big)_{i=1}^n.$$

The problem is to use this data to formulate a function $p : \mathbb{R}^d \mapsto \{0, 1, \dots, k-1\}$ for predicting the labels of previously unseen points.

To solve a multiclass classification problem with a neural network we use the softmax output $s = \text{softmax}(z_L)$, defined by

$$\text{softmax}(x)_i = \frac{e^{x_i}}{\sum_{j=1}^m e^{x_j}}.$$

Note that $\text{softmax}(z_L)_i$ can be interpreted as $\mathbb{P}(y = i \mid z_L)$. We then use the loss function defined by

$$L(z_L, y) = -\log \mathbb{P}(y \mid z_L) = -\log \big(\text{softmax}(z_L)_y\big).$$

2.3.5 Alternative activation functions

In Section 2.2.1 we used $f^{(1)} = f^{(2)} = \cdots = f^{(L)} = \sigma$, the logistic sigmoid. Some other choices which are popular are

$$f(x) = \tanh(x)$$

and

$$f(x) = \max(0, x),$$

which is called the rectified linear activation function (relu).

2.3.6 See Example 5

In this example I use the regression and classification models in `sklearn.neural_network` to make predictions in regression and multi-class classification problems.

Experiment by altering these example problems and by trying different options in the neural network models. Can you generate a really complicated multi-class classification problem? How do the decision boundaries vary for the different activation functions?

2.4 Universal Approximation Theorem

Neural networks can learn more complex patterns than logistic regression. But is there a limit to what functions they can be used to accurately approximate?

2.4.1 Universal Approximation Theorem

Theorem 2.2 (Horink 1991 [8])**.** *Let $\varphi : \mathbb{R} \mapsto \mathbb{R}$ be non-constant, bounded and continuous. Let I_d denote the unit hypercube*

$$I_d = \{x \in \mathbb{R}^d \ : \ 0 \leq x_i \leq 1, \ \textit{for all } i = 1, \ldots, d\}.$$

Then for all $f \in C^0(I_d)$ and $\epsilon > 0$ there exists $N \in \mathbb{N}$, $v_1, \ldots, v_N \in \mathbb{R}$, $w_1, \ldots, w_N \in \mathbb{R}^d$ and $b_1, \ldots, b_N \in \mathbb{R}$ such that

$$\sup_{\boldsymbol{x} \in I_d} \left| f(\boldsymbol{x}) - \sum_{i=1}^{N} v_i \varphi\big(w_i^\top \boldsymbol{x} + b_i\big) \right| < \epsilon.$$

2.5 Overfitting

The Universal Approximation Theorem shows that we can fit any pattern arbitrarily well using a big enough neural network. However in practise we are limited by the quality and quantity of our training data and when these are in short supply we must be very careful.
Overfitting is when we do something that improves the model's accuracy score on the training data but at the expense of its accuracy score on previously unseen test data. Rather than learning something about the pattern in the data that will generalize to new unseen examples we learn something particular to the training data which is usually noise. This is especially pertinent when we do not have a lot of data or when the data is very noisy or both.
The general principal that guides us is that simpler predictive functions tend to be less prone to overfitting because they have less capacity to learn complex patterns and are more likely to only learn the dominant (and therefore hopefully generalizeable) patterns in the data. We therefore need to strike a careful balance between using too simple a model and missing important non-linear patterns and using too complex a model and overfitting. The number of layers and the number of neurons in each layer can be adjusted along with the other hyperparameters of the neural network. These different setups can then be ranked according to their validation accuracy to find the best choice. This process is known as *model selection*.

2.5.1 Weight decay

We set up a neural network just as before only now we choose the weights to minimize the function F given by

$$F(W) = NLL(W|X, \boldsymbol{y}) + \alpha \sum_{i=1}^{L} \|W^{(i)}\|_F^2/2,$$

where $NLL(W|X, \boldsymbol{y})$ is the negative log likelihood, $\alpha > 0$ is the *weight decay parameter* and $\|\cdot\|_F$ is the Frobeius norm of a matrix defined by

$$\|M\|_F = \sqrt{\sum_{ij} m_{ij}^2}.$$

Minizing F instead of NLL only requires a small change to the equations for the gradient as follows

$$\frac{\partial F(W)}{\partial W^{(i)}} = \frac{\partial NLL(W|X, \boldsymbol{y})}{\partial W^{(i)}} + \alpha W^{(i)}.$$

2.5.2 Early stopping

Another method to prevent overfitting which is very widely used is early stopping. Here we apply Stochastic Gradient Descent to minimize our object function but also monitor the validation accuracy as we go. Rather than stopping SGD when it has converged we stop when the validation loss (or accuracy) stops decreasing.

2.5.3 See Example 6

In this example I show a problem that can be vulnerable to overfitting. I show how to use cross validation to select the optimal value for the regularization penalty in `sklearn.neural_network.MLPClassifier`. I also test early stopping and what happens when we instead choose a simpler model.

Algorithm 4 Stochastic Gradient Descent with Early Stopping

```
1: Start with a function $F = \sum_{i=1}^n F_i$ to optimize with respect to $\boldsymbol{\theta}$
2: Choose an initial point $\boldsymbol{\theta}_0$ and a step size $\eta$
3: for $k = 1, 2, \ldots$ until loss on validation data stops decreasing do
4:     draw $t(k)$ uniformly at random from $\{1, 2, \ldots, n\}$
5:     $\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \eta \nabla F_i(\boldsymbol{\theta}_k)$
6: end for
```

Try testing some other `sklearn` classification models on this problem. Think about how you avoid overfitting using these different models? Can you generate a higher dimensional version of this problem? Is overfitting still a problem?

2.6 Covnets for image analysis

Neural networks of various kinds are applied to a wide variety of problems in applications but there are two particular examples where they really outperform all of the competition. Namely language processing and image analysis. In this section we look at the later problem.

2.6.1 Image classification

We can represent a pixilated greyscale image as a matrix $\boldsymbol{x} \in \mathbb{R}^{n \times n}$, where x_{ij} is the blackness of pixel (i, j). Or similarly a pixilated RGB image as a tensor $\boldsymbol{x} \in \mathbb{R}^{n \times n \times 3}$.

In a *multiclass color image classification problem* with $k \in \mathbb{N}$ classes, we are given a *training set* of n labeled images

$$\left(\boldsymbol{x}_i \in \mathbb{R}^{n \times n \times 3}, y_i \in \{0, 1, \ldots, k-1\}\right)_{i=1}^n.$$

The problem is to use this data to formulate a function $p : \mathbb{R}^{n \times n \times 3} \mapsto \{0, 1, \ldots, k-1\}$ for predicting the labels of previously unseen images.

If we simply converted the images into vectors in $\mathbb{R}^{d^2}$ then we could apply any of the previous classification algorithms. However we will see that neural networks which work with the 2d image structure are far more effective.

2.6.2 Convolutional neural network

An image convolution with input size $n \times n$ and filter size $m \times m$ is a linear map $C_w : \mathbb{R}^{n \times n} \to \mathbb{R}^{(n-m+1) \times (n-m+1)}$, defined by

$$C_w(\boldsymbol{x})_{ij} = \sum_{t_2=1}^{m} \sum_{t_1=1}^{m} w_{t_1, t_2} x_{i+t_1, j+t_2},$$

where $w \in \mathbb{R}^{m \times m}$ are the filter weights.

A tensor convolution with input size $n \times n \times k$ and filter size $m \times m$ is a linear map $C_w : \mathbb{R}^{n \times n \times k} \to \mathbb{R}^{(n-m+1) \times (n-m+1)}$, defined by

$$C_w(\boldsymbol{x})_{ij} = \sum_{t_3=1}^{k} \sum_{t_2=1}^{m} \sum_{t_1=1}^{m} w_{t_1, t_2, t_3} x_{i+t_1, j+t_2, t_3},$$

where $w \in \mathbb{R}^{m \times m \times k}$ are the filter weights.

A convolutional neural network with $n_0 \times n_0 \times 3$ dimensional input and 1 dimensional output consists of

- Hidden layer sizes: $h_1, h_2 \ldots, h_{L-1} \in \mathbb{N}$,
- Image dimensions: $n_0 > n_1 > h_2 > \cdots > h_{L-1} > 1 \in \mathbb{N}$,
- Filter sizes: $m_i = n_i - n_{i+1} + 1$ for $i = 0, \ldots, L-1$,

- Weights:

$$\begin{aligned}
&W^{(0)} \in \mathbb{R}^{h_1 \times m_0 \times m_0 \times 3} \\
&W^{(1)} \in \mathbb{R}^{h_2 \times m_1 \times m_2 \times h_0} \\
&\dots \\
&W^{(L-2)} \in \mathbb{R}^{h_{L-1} \times m_{L-2} \times m_{L-2} \times h_{L-2}} \\
&W^{(L-1)} \in \mathbb{R}^{1 \times m_{L-1} \times m_{L-1} \times h_{L-1}},
\end{aligned}$$

- Biases: $b^{(0)} \in \mathbb{R}^{h_1}$, $b^{(1)} \in \mathbb{R}^{h_2}$, $\dots b^{(L-2)} \in \mathbb{R}^{h_{L-1}}, b^{(L-1)} \in \mathbb{R}$,
- Activation function: $f : \mathbb{R} \mapsto \mathbb{R}$.

Define the *activations*

$$\begin{aligned}
&a_0 : \mathbb{R}^{n_0 \times n_0 \times 3} \mapsto \mathbb{R}^{n_0 \times n_0 \times 3} \\
&a_1 : \mathbb{R}^{n_0 \times n_0 \times 3} \mapsto \mathbb{R}^{n_1 \times n_1 \times h_1} \\
&\dots \\
&a_{L-1} : \mathbb{R}^{n_0 \times n_0 \times 3} \mapsto \mathbb{R}^{n_{L-1} \times n_{L-1} \times h_{L-1}} \\
&a_L : \mathbb{R}^{n_0 \times n_0 \times 3} \mapsto \mathbb{R}
\end{aligned}$$

by

$$a_0(\boldsymbol{x}) = \boldsymbol{x}$$

and

$$\big(a_k(\boldsymbol{x})\big)_i = f\left(C_{W_i^{(k-1)}}\big(a_{k-1}(\boldsymbol{x})\big) + b_i^{(k-1)}\right),$$

for $i = 1, \dots, h_k$ and $k = 1, \dots, L$ and where

$$a_k(\boldsymbol{x})_i \equiv a_k(\boldsymbol{x})[:,:,i], \quad W_i^{(k-1)} \equiv W^{(k-1)}[i,:,:,:].$$

Also define $z_k \in \mathbb{R}^{n_k \times n_k \times h_k}$ by

$$(z_k)_i = C_{W_i^{(k-1)}}\big(a_{k-1}(\boldsymbol{x})\big) + b_i^{(k-1)},$$

for $k = 1, \dots, L$.

Then as before we choose

$$W, b = \arg\min_{W,b} NLL(W, b | X, \boldsymbol{y}),$$

where

$$NLL(W, b | X, \boldsymbol{y}) = \sum_{i=1}^{n} L\big(z_L(\boldsymbol{x}_i), y_i\big),$$

which we compute by stochastic gradient decent.

2.6.3 Covnets in practice

In practise the best performing image classifying nets usually start by applying a series of convolutions and then switch to a fully connected structure. They also combine the convolution layers with a number of different layer types including dropout layers and max-pooling layers.

For problems with a lot of training data the best performing networks tend to be huge with tens of layers and millions of weight parameters. When dealing with problems with a smaller training set one popular technique is transfer learning. Rather than initializing a network with zero or random weights and biases, we start with a network pre-trained on a different image analysis problem (with the same image size) then run SGD with early stopping on the new dataset.

2.6.4 See Example 7

In this example I show how to apply covnets to an image classification problem.

Try altering the number and type of different layers in the network. Can you achieve a higher test accuracy?

3 Gaussian Processes

Everyone is familiar with the idea that assuming data is normally distributed is often a useful approximation. Gaussian processes are a way of extending this approach to learning functions. Gaussian processes are widely used in machine learning because they provide uncertainty estimates as well as point predictions for their target values. This makes them especially useful in situations where there is not a lot of training data and more generally whenever it is important to quantify the confidence of the model's prediction. Gaussian processes are an important example of a Bayesian non-parametric model.

3.1 Properties of multivariate normal distribution

Before we look at Gaussian processes we need to review some theory for multivariate normal distributions.

3.1.1 Multivariate normal distribution

The symmetric matrix $A \in \mathbb{R}^{d\times d}$ is positive-definite if

$$\boldsymbol{x}^\top A \boldsymbol{x} > 0,$$

for all $\boldsymbol{x} \in \mathbb{R}^d$ with $\boldsymbol{x} \neq \underline{0}$. Equivalently A is positive-definite if all of A's eigenvalues are positive.

Two ways to define a multivariate normal distribution:

1. A d dimensional multivariate normal distribution is any distribution such that any 1-dimensional linear combination of its components is itself a 1-dimensional normal distribution.
2. A non-degenerate* d dimensional multivariate normal distribution is defined by its mean $\mu \in \mathbb{R}^d$ and covariance matrix $\Sigma \in \mathbb{R}^{d\times d}$ (which must be symmetric positive-definite). We write $\boldsymbol{x} \sim \mathcal{N}(\mu, \Sigma)$. The p.d.f is given by
$$\rho(\boldsymbol{x}) = \frac{1}{\sqrt{(2\pi)^d \det(\Sigma)}} \exp\left(-\frac{1}{2}(\boldsymbol{x}-\mu)^\top \Sigma^{-1}(\boldsymbol{x}-\mu)\right).$$

$(*)$ Note that degenerate examples occur when Σ is positive-semi-definite but not positive definite, which means it has at least one eigenvalue equal to zero. In these cases the distribution does not admit a continuous density in $\mathbb{R}^d$. However it is always possible to restrict to a lower dimensional subspace where the distribution does admit a continuous density and without loosing anything. E.g. $\boldsymbol{x} = [x, 1]^\top$, where x is a standard 1-d $(0, 1)$ normal.

3.1.2 Affine transformation

Theorem 3.1. *Let $\boldsymbol{x} \sim \mathcal{N}(\mu, \Sigma)$ and let $\boldsymbol{y} = A\boldsymbol{x} + b$ then*

$$\boldsymbol{y} \sim \mathcal{N}(A\mu + b, A\Sigma A^\top).$$

3.1.3 Conditional distribution

Theorem 3.2. *Let $\boldsymbol{x} \sim \mathcal{N}(\mu, \Sigma)$ be partitioned into blocks as*

$$\begin{bmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}, \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix}\right),$$

then

$$(\boldsymbol{x}_2|\boldsymbol{x}_1) \sim \mathcal{N}\left(\mu_2 + \Sigma_{21}\Sigma_{11}^{-1}(\boldsymbol{x}_1 - \mu_1)\ ,\ \Sigma_{22} - \Sigma_{21}\Sigma_{11}^{-1}\Sigma_{12}\right).$$

3.2 Gaussian linear models

We can use the above results about multivariate normal distributions to compute Bayesian solutions to linear regression problems.

3.2.1 Gaussian linear models

Suppose that $(\boldsymbol{x}_i \in \mathbb{R}^d, y_i \in \mathbb{R})_{i=1}^n$ is the training data for a regression problem.

We model the relationship between the features and target by

$$y_i = \boldsymbol{w}^\top \boldsymbol{x}_i + z_i,$$

where the weights vector $\boldsymbol{w}$ is a d-dimensional multivariate normal with $\boldsymbol{w} \sim \mathcal{N}(\underline{0}, \alpha^2 I)$ and $(z_i)_{i=1}^n$ are the noise terms, which are i.i.d. 1-dimensional normals with $z_i \sim \mathcal{N}(0, \sigma^2)$.

Equivalently

$$\boldsymbol{y} = X\boldsymbol{w} + \boldsymbol{z},$$

where the noise vector $\boldsymbol{z}$ is a n-dimensional multivariate normal with $\boldsymbol{z} \sim \mathcal{N}(\underline{0}, \sigma^2 I)$.

Therefore by Theorem 3.1 we have

$$\boldsymbol{y} \sim \mathcal{N}\big(\underline{0}, \alpha^2 X X^\top + \sigma^2 I\big).$$

Now lets suppose that we have n training points and m test points where we want to make the predictions. As in the proof of Theorem 3.2 we partition everything into blocks so that the training data is in block 1 and the queary points in block 2

$$X = \begin{bmatrix} X_1 \\ X_2 \end{bmatrix}, \quad \boldsymbol{z} = \begin{bmatrix} \boldsymbol{z}_1 \\ \boldsymbol{z}_2 \end{bmatrix}, \quad \boldsymbol{y} = \begin{bmatrix} \boldsymbol{y}_1 \\ \boldsymbol{y}_2 \end{bmatrix},$$

and from the Theorem we have

$$(\boldsymbol{y}_2 | \boldsymbol{y}_1) \sim \mathcal{N}\left(X_2 X_1^\top (X_1 X_1^\top)^{-1} \boldsymbol{y}_1, \alpha^2 X_2 X_2^\top - \alpha^4 X_2 X_1^\top \left(\alpha^2 X_1 X_1^\top + \sigma^2 I\right)^{-1} X_1 X_2^\top\right).$$

3.2.2 Non-linear basis functions

Rather than working with the raw features $(\boldsymbol{x}_i)_{i=1}^n$ we use a non-linear feature map $\phi : \mathbb{R}^d \mapsto \mathbb{R}^m$ which allows the model more flexibility.

If we assume that

$$y_i = \boldsymbol{w}^\top \phi(\boldsymbol{x}_i) + z_i,$$

where $\boldsymbol{w}$ is a m-dimensional multivariate normal with $\boldsymbol{w} \sim \mathcal{N}(\underline{0}, \alpha^2 I)$. Then we obtain

$$\boldsymbol{y} \sim \mathcal{N}\left(\underline{0}, \alpha^2 K + \sigma^2 I\right),$$

where

$$K_{ij} := \kappa(\boldsymbol{x}_i, \boldsymbol{x}_j) = \phi(\boldsymbol{x}_i)^\top \phi(\boldsymbol{x}_j).$$

Note that the choice of feature map only shows up in the inner product in the function κ. We call κ the kernel. Different feature maps result in different kernel matrices which give different predictive models.

3.2.3 Polynomial Kernel

For $d = 1$, let $\phi(x) = [1, x, x^2, \ldots, x^m]^\top$. Then

$$\kappa(x, y) = \sum_{k=0}^{m} (xy)^k.$$

Alternately, for $c > 0$, let $\phi : \mathbb{R} \mapsto \mathbb{R}^{m+1}$, with

$$\phi(x)_k = \binom{m}{k} c^{m-k} x^k,$$

for $k = 0, \ldots, m$. Then

$$\kappa(x, y) = (xy + c)^m.$$

3.2.4 See Example 8

In this example I show how to apply Gaussian linear and polynomial regression models to some 2d regression problems.

Experiment by altering the problem data and kernel parameters.

3.3 Gaussian processes for regression

3.3.1 Gaussian processes

A stochastic process $\{f(\boldsymbol{x})\}_{x\in\mathbb{R}^d}$ is a Gaussian process if for every finite set of points $\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_n \in \mathbb{R}^d$, the $\mathbb{R}^n$ valued random variable

$$\big(f(\boldsymbol{x}_i)\big)_{i=1}^n$$

is a multivariate normal distribution.

A Gaussian process can be completely determined from its mean and covariance functions

$$M : \mathbb{R}^d \mapsto \mathbb{R}, \quad K : \mathbb{R}^2 \times \mathbb{R}^d \mapsto \mathbb{R},$$

by

$$\big(f(\boldsymbol{x}_i)\big)_{i=1}^n \sim \mathcal{N}(\Sigma, \mu),$$

with

$$\mu_i = M(\boldsymbol{x}_i),$$

and

$$\Sigma_{ij} = K(\boldsymbol{x}_i, \boldsymbol{x}_j).$$

Theorem 3.3. *Let $(\boldsymbol{x}_i \in \mathbb{R}^d, y_i \in \mathbb{R})_{i=1}^n$ be the training data for a regression problem. Let $\boldsymbol{w}$ be the $\mathbb{R}^d$ valued random variable with prior $\mathcal{N}(\underline{0}, \alpha^2 I)$, conditioned on $y_i = \boldsymbol{w}^\top \boldsymbol{x_i} + z_i$ for $i = 1, \dots, n$, where each z_i is i.i.d. $\mathcal{N}(0, \sigma^2)$. Then the stochastic process $\{f(\boldsymbol{x})\}_{x\in\mathbb{R}^d}$, defined by $f(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x}$, is a Gaussian process.*

3.3.2 RBF kernels

We have seen one way to derive a Gaussian process model for a regression problem by first postulating a linear model for the target variable and then working through to find the covariance and mean functions for the posterior at a set of test points. Another method is to start by choosing the covariance and mean functions for a prior Gaussian process directly and then updating the Gaussian process to account for the training data. The idea of an RBF kernel is that y values for points which are closer together should be more correlated.

The squared exponential RBF kernel $K : \mathbb{R}^d \times \mathbb{R}^d \mapsto \mathbb{R}$ is given by

$$K(\boldsymbol{x}, \boldsymbol{x}') = \alpha^2 \exp\left(\frac{-\|\boldsymbol{x} - \boldsymbol{x}'\|_2^2}{2\ell^2}\right),$$

where $\alpha > 0$ is the standard deviation and $\ell > 0$ is the length scale.

The squared exponential kernel is commonly combined with a constant mean function $M : \mathbb{R}^2 \mapsto \mathbb{R}$ with $M(x) = \mu \in \mathbb{R}$, for all $x \in \mathbb{R}^d$. However it is also possible to use non-constant mean functions.

3.3.3 Gaussian Process regression

Suppose that $(\boldsymbol{x}_i \in \mathbb{R}^d, y_i \in \mathbb{R})_{i=1}^n$ is the training data for a regression problem.

We model the target by

$$y_i = f(\boldsymbol{x}_i) + z_i,$$

where $\{f(\boldsymbol{x})\}_{x\in\mathbb{R}^d}$ is a Gaussian process with squared exponential kernel K with standard deviation $\alpha > 0$ and length scale $\ell > 0$ and constant mean function $\mu \in \mathbb{R}$ and where $(z_i)_{i=1}^n$ are the noise terms, which are i.i.d. 1-dimensional normals with $z_i \sim \mathcal{N}(0, \sigma^2)$.

Now lets suppose that along with the n training points we have m test points $(\boldsymbol{x})_{i=n+1}^{n+m}$ where we want to make the predictions. Then the prior distribution is given by

$$\boldsymbol{y} \sim \mathcal{N}(\mu, \Sigma),$$

where

$$\Sigma_{ij} = K(\boldsymbol{x}_i, \boldsymbol{x}_j) + \sigma^2 \delta_{ij}.$$

Partitioning into blocks as in Theorem 3.2 the posterior distribution at the query points given the data at the training points is given by

$$(\boldsymbol{y}_2|\boldsymbol{y}_1) \sim \mathcal{N}\big(\mu + \Sigma_{21}\Sigma_{11}^{-1}(\boldsymbol{y}_1 - \mu), \Sigma_{22} - \Sigma_{21}\Sigma_{11}^{-1}\Sigma_{12}\big).$$

3.3.4 See Example 9

In this example I show how to apply Gaussian process regression to some 1d regression data. I demonstrate how the choice of hyper-parameters affects the way that the model predicts and also what random samples from the model look like.

Experiment by altering the problem data and kernel parameters. If you generate data with different frequency oscillations (slow, medium, fast) then what length scale values work best? How much training data do you need to learn the pattern and how is this affected by the noise level?

3.4 Estimating hyper-parameters by maximizing marginal likelihood

In practical applications we will not be given suitable values for the various hyper-parameters in the GP model. Instead we will need to choose the hyper-parameters ourselves to best match the data and give in the most accurate predictions. In Section 1.2.3 we solved this problem for the case of choosing the regularization parameter for the least squares linear regression. In that case we tested a grid of values with cross-validation to find the optimal hyper-parameter value. However, searching a grid of values becomes highly inefficient as the number of hyper-parameter increases and GPs often have a separate length scale for each feature, which can result in a huge number of hyper-parameters. Instead people often use the following method which maximizes the likelihood of the hyper-parameter given the data using gradient descent or similar optimization algorithms.
Suppose that $D = (\boldsymbol{x}_i \in \mathbb{R}^d, y_i \in \mathbb{R})_{i=1}^n$ is the training data for a regression problem. Suppose further that θ is a vector containing all of the different hyper-parameters for a GP kernel function K_θ and mean function M_θ. The the likelihood of θ given D is defined by

$$L(\theta \mid D) = \rho(\boldsymbol{y} \mid \theta, X) = \frac{1}{\sqrt{(2\pi)^d \det(\Sigma_\theta)}} \exp\left(-\frac{1}{2}(\boldsymbol{y} - \mu_\theta)^\top \Sigma_\theta^{-1}(\boldsymbol{y} - \mu_\theta)\right),$$

where $(\mu_\theta)_i = M_\theta(\boldsymbol{x}_i)$ and $\big(\Sigma_\theta\big)_{ij} = K_\theta(\boldsymbol{x}_i, \boldsymbol{x}_j)$.
The maximum likelihood estimate for θ is then given by

$$\theta^* = \arg\max_\theta L(\theta \mid D).$$

3.4.1 Derivative of NLL

Rather than maximizing the likelihood we minimize the negative log likelihood. Taking the log will make things easier for the the optimization algorithm and minusing everything gives us a minimization problem which we mathematicians prefer for some reason.

The negative log likelihood is given by

$$NLL(\theta \mid D) = \frac{1}{2}\left(d\log(2\pi) + \log\big(\det(\Sigma_\theta)\big) + (\boldsymbol{y} - \mu_\theta)^\top \Sigma_\theta^{-1}(\boldsymbol{y} - \mu_\theta)\right),$$

so that

$$\frac{\partial NLL(\theta \mid D)}{\partial \theta} = \frac{1}{2}\left(\mathrm{tr}\big(\Sigma_\theta^{-1}\frac{\partial \Sigma_\theta}{\partial \theta}\big) - (\boldsymbol{y} - \mu_\theta)^\top \Sigma_\theta^{-1}\frac{\partial \Sigma_\theta}{\partial \theta}\Sigma_\theta^{-1}(\boldsymbol{y} - \mu_\theta) - 2\frac{\partial \mu_\theta}{\partial \theta}\Sigma_\theta^{-1}(\boldsymbol{y} - \mu_\theta)\right),$$

where we have used the following results

$$\frac{\partial \log\big(\det(A)\big)}{\partial \theta} = \mathrm{tr}\Big(A^{-1}\frac{\partial A}{\partial \theta}\Big),$$

and

$$\frac{\partial A^{-1}}{\partial \theta} = -A^{-1}\frac{\partial A}{\partial \theta}A^{-1}.$$

If we use the standard RBF model we with hyper-parameters $\theta = [\alpha, \ell, \sigma, \nu]$ and

$$\begin{aligned}
\big(\Sigma_\theta\big)_{ij} &= \alpha^2 \exp\left(\frac{-\|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2^2}{2\ell^2}\right) + \sigma^2\delta_{ij},\\
\big(\mu_\theta\big)_i &= \nu,
\end{aligned}$$

then we have

$$\begin{aligned}
\frac{\partial\big(\Sigma_\theta\big)_{ij}}{\partial \alpha} &= 2\alpha \exp\left(\frac{-\|\boldsymbol{x} - \boldsymbol{x}'\|_2^2}{2\ell^2}\right)\\
\frac{\partial\big(\Sigma_\theta\big)_{ij}}{\partial \ell} &= \alpha^2\frac{\|\boldsymbol{x} - \boldsymbol{x}'\|_2^2}{\ell^3} \exp\left(\frac{-\|\boldsymbol{x} - \boldsymbol{x}'\|_2^2}{2\ell^2}\right)\\
\frac{\partial\big(\Sigma_\theta\big)_{ij}}{\partial \sigma} &= 2\sigma\delta_{ij}\\
\frac{\partial\big(\mu_\theta\big)_i}{\partial \nu} &= 1.
\end{aligned}$$

3.4.2 Numerical stability

Computing determinant and inverses of matrices is best avoided as these operations are highly unstable and likely to result in numerical errors. When evaluating $NLL(\theta \mid D)$ and its derivative it is advisable to use the following tricks.

1. Instead of calculating $\log(\det(\Sigma_\theta))$ directly it is much better to take the QR decomposition $QR = \Sigma_\theta$ and then take the sum of the logs of the absolute values of the diagonal entries of R.
2. We cannot get around having to compute Σ_θ^{-1} but we can avoid problems with instability here by insisting that $\sigma \geq \epsilon$ for some small positive constant $\epsilon > 0$. This results in a constrained optimization problem, which is easy to solve using the package `scipy.optimize.minimize`. You can select ϵ to be a small multiple of the standard deviation of the data $\boldsymbol{y}$.
3. Since $NLL(\theta \mid D)$ can be non-convex it is a good idea to start the optimization multiple times from different random starting positions then choose the solution that achieves the lowest value for the objective.

3.4.3 See Example 10

In this example I show how to fit hyper-parameters using `sklearn.gaussian_process`. I also show that this package has some shortcomings. In particular that it cannot fit the maximum likelihood value for the constant mean function and therefore has to use either mean zero or the sample mean, both of which can result in bad predictions for certain problems.

Experiment with applying `sklearn.gaussian_process` to a 2d problem. You could start by modifying one of the 2d classification problems that we used earlier in the course. Look at what values the model fits for its hyper-parameters and try altering the problem to change these values: does it behave as you expected?

3.5 Properties of kernel functions

3.5.1 Radial basis functions

A kernel function $K(\boldsymbol{x}, \boldsymbol{x}')$ is *isotropic* if it only depends on $\boldsymbol{x} - \boldsymbol{x}'$. If it only depends on $|\boldsymbol{x} - \boldsymbol{x}'|$ the we call it a *Radial Basis Function (RBF).* Note that the squared exponential kernel is an RBF but that the kernel resulting from the linear model is neither an RBF nor isotropic.

3.5.2 Continuity and smoothness

Let $\{f(\boldsymbol{x})\}_{x\in\mathbb{R}^d}$ be a stochastic process. We say that f is *continuous in mean square* at $\boldsymbol{x}^*$ if

$$\lim_{k\to\infty} \mathbb{E}\left[\left(f(\boldsymbol{x}_k) - f(\boldsymbol{x}^*)\right)^2\right] = 0,$$

for any sequence of points $\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots \in \mathbb{R}^d$ that converge to $\boldsymbol{x}^*$.

Theorem 3.4 (Adler Chapter 3 [9])**.** *A GP $\{f(\boldsymbol{x})\}_{x\in\mathbb{R}^d}$, with covariance function $K : \mathbb{R}^d \times \mathbb{R}^d \mapsto \mathbb{R}$, is continuous in the mean square at $\boldsymbol{x}^*$ if and only if K is continuous at the point $(\boldsymbol{x}^*, \boldsymbol{x}^*)$.*

Corollary 3.5. *Any GP with the squared exponential kernel will be everywhere continuous in the mean square.*

The derivative of $\{f(\boldsymbol{x})\}_{x\in\mathbb{R}^d}$ in the ith direction at $\boldsymbol{x}^*$ is defined by

$$\frac{\partial f(\boldsymbol{x})}{\partial x_i}\Big|_{\boldsymbol{x}=\boldsymbol{x}^*} = \lim_{h\to 0} \frac{f(\boldsymbol{x}^* + h\boldsymbol{e}_i) - f(\boldsymbol{x}^*)}{h}.$$

We say that f is differentiable in the mean square at $\boldsymbol{x}^*$ if

$$\lim_{h\to 0} \mathbb{E}\left[\left(\frac{f(\boldsymbol{x}^* + h\boldsymbol{e}_i) - f(\boldsymbol{x}^*)}{h} - \frac{\partial f(\boldsymbol{x})}{\partial x_i}\Big|_{\boldsymbol{x}=\boldsymbol{x}^*}\right)^2\right] = 0.$$

Theorem 3.6 (Adler Chapter 3 [9])**.** *A GP $\{f(\boldsymbol{x})\}_{x\in\mathbb{R}^d}$, with covariance function $K : \mathbb{R}^d \times \mathbb{R}^d \mapsto \mathbb{R}$, is k-times differentiable in the mean square at $\boldsymbol{x}^*$ if and only if K is $2k$-times differentiable at the point $(\boldsymbol{x}^*, \boldsymbol{x}^*)$.*

Corollary 3.7. *Any GP with the squared exponential kernel will be everywhere infinitely differentiable in the mean square.*

3.5.3 Matérn Kernel

The Matérn Kernel is an alternative RBF kernel to the squared exponential. In the Matérn Kernel we are able to adjust the degree of smoothness in the resulting GP via a parameter ν. In general the Matérn Kernel is given by

$$K(r) = \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\frac{\sqrt{2\nu}r}{\ell}\right)^{\nu} K_\nu \left(\frac{\sqrt{2\nu}r}{\ell}\right),$$

where $r = \|\boldsymbol{x} - \boldsymbol{x}|$, ℓ is the length scale, ν is the smoothness parameter and K_ν is a modified Bessel function. It can be shown that K results in a GP which is everywhere $\lfloor(\nu)\rfloor$-times differentiable in the mean square Two important examples

$$K_{\nu=3/2}(r) = \left(1 + \frac{\sqrt{3}r}{\ell}\right) \exp\left(-\frac{\sqrt{3}r}{\ell}\right),$$

$$K_{\nu=5/2}(r) = \left(1 + \frac{\sqrt{5}r}{\ell} + \frac{5r^2}{3\ell^2}\right) \exp\left(-\frac{\sqrt{5}r}{\ell}\right).$$

3.5.4 See Example 11

In this example I compare samples from models with different kernels. I also test RBF and the two matern kernels on some non-smooth data.

RBF kernels will work with any distance function provided it is a metric. Experiment with using different norms on the 2d examples that you tried last week.
There is also no reason why we have to use $\mathbb{R}^d$ as the domain. What about taking a set of strings and using edit distance? Can you think of some other unusual metric spaces that could serve as the domain for a Gaussian process? Can you think of some practical application where one of these models would be useful?

3.6 Gaussian Processes for Classification

So far we have seen how Gaussian processes can be used in regression problems but what about classification? Since Gaussian processes have an unbounded continuous domain we cannot use them directly to solve a classification problem. However, recall that in Logistic regression we 'squished' an affine function $f(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x} + b$ through the logistic function onto the interval $[0, 1]$ and then interpreted the results as class probabilities. It turns out we can do the same thing here only replacing the affine function with a Gaussian process $\{f(\boldsymbol{x})\}_{x\in\mathbb{R}^d}$. The extra flexibility of the Gaussian process compared to the affine function allows this method to learn more complicated decision boundaries and the Bayesian nature of Gaussian processes means we can also estimate the uncertainty in all of our predictions.

3.6.1 Gaussian Process Classification

Let

$$\big(\boldsymbol{x}_i \in \mathbb{R}^d, y_i \in \{0,1\}\big)_{i=1}^n,$$

be the training data for a classification problem.

Suppose that the labels of the training data are random variables with

$$\mathbb{P}\big(y_i = 1 \mid \boldsymbol{x}_i\big) = \sigma\big(f(\boldsymbol{x}_i)\big), \tag{1}$$

where $\sigma : \mathbb{R} \mapsto [0, 1]$ is the *logistic sigmoid* defined by

$$\sigma(z) = \frac{1}{1+e^{-z}},$$

and where $\{f(\boldsymbol{x})\}_{x\in\mathbb{R}^d}$ is a Gaussian process with mean function $M : \mathbb{R}^d \mapsto \mathbb{R}$ and covariance function $K : \mathbb{R}^2 \times \mathbb{R}^d \mapsto \mathbb{R}$. We will assume thorought this section that M is identically equal to zero.

Now suppose that we have a new point $\boldsymbol{x}' \in \mathbb{R}^d$ where we want to predict the class. Using the model (1) we have

$$\mathbb{P}\big(y' = 1 \mid \boldsymbol{x}, X, \boldsymbol{y}\big) = \int \sigma\big(f(\boldsymbol{x}')\big)\rho(f|X, \boldsymbol{y})df. \tag{2}$$

So all that remains is to compute this integral!

3.6.2 Laplace Approximation

First define $\boldsymbol{f} \in \mathbb{R}^n$ by $f_i = f(\boldsymbol{x}_i)$ and let $\Sigma \in \mathbb{R}^{n\times n}$ be the prior covariance matrix for $\boldsymbol{f}$. We can replace f with $\boldsymbol{f}$ in (2) by

$$\mathbb{P}\big(y' = 1 \mid \boldsymbol{x}, X, \boldsymbol{y}\big) = \int \mathbb{E}[\sigma\big(f(\boldsymbol{x}')|\boldsymbol{f}, X)\big)]\rho(\boldsymbol{f}|X, \boldsymbol{y})d\boldsymbol{f}. \tag{3}$$

We will now approximate $\rho(\boldsymbol{f}|X, \boldsymbol{y})$ by a normal distribution. Note that this kind of approximation inside of an integral is known as the Laplace approximation, which you may have seen in previously in courses on asymptotic methods. The normal approximation is given by

$$\rho(\boldsymbol{f}|X, \boldsymbol{y}) \approx \mathcal{N}(\hat{\boldsymbol{f}}, A^{-1}),$$

where

$$\hat{\boldsymbol{f}} = \arg\max_{\boldsymbol{f}} \rho(\boldsymbol{f}|X,\boldsymbol{y}), \quad A = -\nabla\nabla \log \rho(\boldsymbol{f}|X,\boldsymbol{y})\big|_{\boldsymbol{f}=\hat{\boldsymbol{f}}}. \tag{4}$$

Consider

$$\rho(\boldsymbol{f}|X,\boldsymbol{y}) = \frac{\rho(\boldsymbol{y}|\boldsymbol{f})\rho(\boldsymbol{f}|X)}{\rho(\boldsymbol{y}|X)},$$

and define

$$\begin{aligned}\Psi(\boldsymbol{f}) &= \log \rho(\boldsymbol{y}|\boldsymbol{f}) + \log \rho(\boldsymbol{f}|X) \\ &= -\frac{1}{2}\boldsymbol{f}^\top \Sigma^{-1}\boldsymbol{f} - \frac{1}{2}\log\det\big(\Sigma\big) - \frac{n}{2}\log(2\pi) + \sum_{i=1}^{n} \begin{cases} -\log(1+\exp^{-f}) & \text{if } y_i = 1, \\ -\log(1+\exp^{f}) & \text{if } y_i = 0. \end{cases}\end{aligned}$$

Then we obtain $\hat{\boldsymbol{f}}$ by solving

$$\boldsymbol{f} = \Sigma\big(\nabla \log \rho(\boldsymbol{y}|\boldsymbol{f})\big), \tag{5}$$

and set $A = \big(\Sigma^{-1} + W^{-1}\big)^{-1}$, where

$$W = -\nabla\nabla \log \rho(\boldsymbol{y}|\boldsymbol{f}). \tag{6}$$

3.6.3 Making Predictions

Now define $\boldsymbol{b} \in \mathbb{R}^n$ by $b_i = K(\boldsymbol{x}', \boldsymbol{x}_i)$ and let $a = K(\boldsymbol{x}', \boldsymbol{x}')$. The Laplace approximation gives

$$\rho\big(f(\boldsymbol{x}')|X,y\big) \approx \mathcal{N}\left(\boldsymbol{b}\Sigma^{-1}\hat{\boldsymbol{f}}, \underbrace{a - \boldsymbol{b}\Sigma^{-1}\boldsymbol{b}^\top + \boldsymbol{b}A^{-1}\boldsymbol{b}^\top}\right), \tag{7}$$

where the underbraced covariance term simplifies to $a - \boldsymbol{b}W^{-1}\boldsymbol{b}^\top$. Finally we substitute (7) into

$$\mathbb{P}\big(y' = 1 \mid \boldsymbol{x}, X, \boldsymbol{y}\big) = \int_{-\infty}^{\infty} \sigma(z)\rho\big(f(\boldsymbol{x}') = z|X,y\big)dz, \tag{8}$$

and compute the class probability with 1d numerical integration.

3.6.4 See Example 12

In this example I use the `sklearn.gaussian_process.GaussianProcessClassifier` model to make predictions for various classification problems.

Experiment by applying `sklearn.gaussian_process.GaussianProcessClassifier` to some different problems. Compare the performance of `sklearn.gaussian_process.GaussianProcessClassifier` to `sklearn.neighbors.KNeighborsClassifier`. Can you design an example problem where K-nearest neighbours consistently beats Gaussian process classification?

4 Coursework Assignments

4.1 Coursework 1

1. Write a function that takes a matrix $X \in \mathbb{R}^{n \times 2}$ and a vector $\boldsymbol{y} \in \{0,1\}^n$ and plots the rows $\boldsymbol{x}_i$ of X with different color and/or marker depending on weather $y_i = 0$ or $y_i = 1$. [1]

2. (a) Write a function that randomly generates binary classification data in the form of a matrix $X \in \mathbb{R}^{n \times 2}$ and a vector $y \in \{0,1\}^n$, such that the classes are linearly separable. Plot the data using your previously written function.

   (b) Repeat but such that the classes are separable but not linearly separable.

   (c) Repeat but such that the classes are nearly but not quite separable.

   (d) Repeat but such that the classes are far from separable. [1]

3. Write a function that takes a model (with a predict method) and uses `matplotlib.pyplot.contour` to plot a contor plot of the model's prediction. [1]

4. Write a function that takes a model (with a fit method and a predict method) and a binary classification data set $X \in \mathbb{R}^{n \times d}$ and $\boldsymbol{y} \in \{0,1\}^n$, randomly splits the data into a training set and a testing set, trains the model on the training data and then tests its accuracy on the test set and returns the test accuracy. Using `sklearn.neighbors.KNeighborsClassifier`, for each of your data-sets train a k-nearest neighbour model, test its accuracy and plot its prediction. [1]

5. ($*$) Generate an example problem where the k-nearest neighbour method makes poor predictions despite the prediction problem appearing to be easy. [1]

Note that ($*$) questions are intended to be a little more challenging. Please do not discuss these questions with your fellow students or try to Google the answer until you have submitted your work.

4.2 Coursework 2

1. Write code for logistic regression. You should have a fit function/method, which takes an input $X \in \mathbb{R}^{n \times d}$ and $\boldsymbol{y} \in \{0,1\}^n$ and computes the optimal parameters $\boldsymbol{w} \in \mathbb{R}^d$ and $b \in \mathbb{R}$, and a predict function/method, which takes as input $X \in \mathbb{R}^{n \times d}$ and returns a vector of predictions $\boldsymbol{p} \in [0,1]^n$. [3]

2. Test your logistic regression model on all of the examples you used in Coursework 1 and comment on your results. [2]

3. ($*$) Generate an example problem where logistic regression has a much lower training accuracy score than the optimal linear decision boundary. [2]

4. Write code for logistic regression that fits the optimal parameters $\boldsymbol{w} \in \mathbb{R}^d$ and $b \in \mathbb{R}$ using Newton's method. Is this method faster than gradient descent? [3]

If you feel confident with Python write the code for 1 and 4 as classes with fit and predict methods. If you don't feel so confident yet then just write each one as two separate functions, one for fitting and one for predicting.

4.3 Coursework 3

1. Write code for neural networks. You should include fit and predict functions/methods. The fit function/method should split the training data into training and validation subsets then monitor the validation accuracy whilst training to determine when to stop. [6]

2. Test your neural network model on all of the examples you used in Coursework 1 and comment on your results. [2]

3. ($*$) In general is NLL(W,b) a convex function for neural networks? Justify your answer with an explicit example. [2]

4.4 Coursework 4

1. Generate four characteristically different 1d regression data-sets with x values in the interval $[0, 1]$. [2]

2. Write code for Gaussian Process regression using the squared exponential kernel. The fit function/method should optimize the various hyper-parameters, you can use `scipy.optimize.minimize` so long as you write code to compute the gradient yourself. The predict function/method should return the posterior mean and standard deviation. [6]

3. Split the data into test and training subsets. Train a model on each data-set, plot the predictions, with standard-deviation over the interval and compute the test accuracy. Comment on your results. [2]

4.5 Coursework 5

Form a group of 2-4 and enter one of the `Kaggle.com` data science contests as a team. Work together with prepossessing the data, applying machine learning algorithms and interpreting results. Prepare a 20 minute talk based on your findings, structure your talk around an IPython notebook. You may use any software packages you like provided you give a brief expiation of how they work in the talk. [10]