Machine Learning and Pattern Recognition - Week 6 - Kernels for GPs & Softmax for Classification

Antonio León Villares

October 2022

Contents

Based on the online notes here.

1 Kernels and Gaussian Processes

1.1 The Kernel Trick

- **What is the kernel trick?**
  - in many machine learning applications, we often have to compute **dot products**
  - these dot products are often computed in **high-dimensional feature space**
  - for example, if we use 10^6 RBFs, and we want to do Bayesian Linear Regression via Gaussian Processes, we showed last week that the kernel is:
$$k(\underline{x}^{(i)}, \underline{x}^{(j)}) = \sigma_w^2 \phi(\underline{x}^{(i)})^T \phi(\underline{x}^{(j)}) + \sigma_b^2$$
  - the **kernel trick** allows us to compute $\phi(\underline{x}^{(i)})^T \phi(\underline{x}^{(j)})$ **directly**, without explicitly mapping our input vectors to the high-dimensional space
- **When is a ML algorithm kernelised?**
  - when it can be expressed in terms of dot products
  - in these cases, we can then apply the kernel trick
- **Why is the kernel trick so useful?**
  - we can use it to find dot products in **infinite dimensional** feature space
  - for instance, if we placed an RBF at every point on $\mathbb{R}$, it can be shown that a dot product in this infinite dimensional space is given by the **gaussian kernel**:
$$k(\underline{x}^{(i)}, \underline{x}^{(j)}) \propto \exp(-\|\underline{x}^{(i)} - \underline{x}^{(j)}\|^2)$$
  - hence, the **kernel trick** allows us to derive a model which:
    1. Approximates arbitrarily **complicated** functions
    2. Is valid on any domain (contrast this with using **finitely** many RBFs: beyond their defined range, the functions all become 0, so we have no information outside the domain of the RBFs)

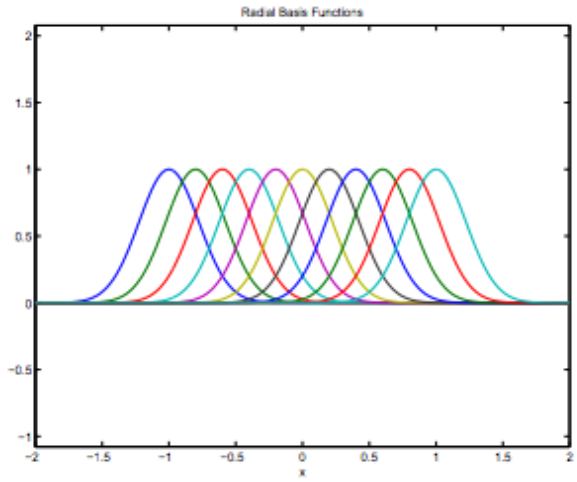


Figure 1: Beyond $[-2, 2]$, the RBFs all evaluate close to 0, so any linear combination will be close to 0, making this model useless outside of $[-2, 2]$.

- **For Gaussian processes, why are Mercer Kernels used?**
  1. They produce **positive definite** matrices, so they give us a **covariance matrix**
  2. It can be shown that **Mercer kernels** correspond to a **dot product** in a given feature space, which might be infinite, thus giving a GP arbitrary flexibility

1.2 Types of Kernels

See here for plots and explanations for the kernels.

1. **Squared-Exponential Kernel (with Parameters)**

$$K(\underline{x}^{(i)}, \underline{x}^{(j)}) = \sigma_f^2 \exp\left(-\frac{1}{2} \sum_{d \in D} \frac{\left(x_d^{(i)} - x_d^{(j)}\right)^2}{\ell_d^2}) \right)$$

   where:

   - σ_f is the **amplitude**
   - ℓ is the **lengthscale**

   (these parameters will be better explained in the next section)

2. **Periodic Kernel (with Parameters)**

$$K(\underline{x}^{(i)}, \underline{x}^{(j)}) = \sigma_f^2 \exp\left(-\frac{2}{\ell^2} \sin^2\left(\pi \frac{\|\underline{x}^{(i)} - \underline{x}^{(j)}\|}{p} \right) \right)$$

   where:

   - σ_f is the **amplitude**
   - ℓ is the **lengthscale**
   - p is the **period**

3. **Kernel Combinations**

$$K(\underline{x}^{(i)}, \underline{x}^{(j)}) = \alpha K_1(\underline{x}^{(i)}, \underline{x}^{(j)}) + \beta K_2(\underline{x}^{(i)}, \underline{x}^{(j)})$$

   If we combine Mercer kernels in this way, we obtain new Mercer kernels!

4. **Abstract Kernels**: these allow us to compare abstract objects, such as **strings** or **graphs**

1.3 Hyperparameters for the Gaussian Kernel

1.3.1 Effect of Hyperparameters

> *In this section we consider the Gaussian Kernel, defined by given parameters:*
>
> $$K(\underline{x}^{(i)}, \underline{x}^{(j)}) = \sigma_f^2 \exp\left(-\frac{1}{2} \sum_{d \in D} \frac{\left(x_d^{(i)} - x_d^{(j)}\right)^2}{\ell_d^2}) \right)$$
>
> *where σ_f is the **amplitude** and ℓ is the **lengthscale**.*

- **How does the amplitude affect the priors sampled from a GP?**

- σ_f affects how large the covariance becomes
- as such, the larger that σ_f, we expect our priors to reach larger values, and to also be "steeper"

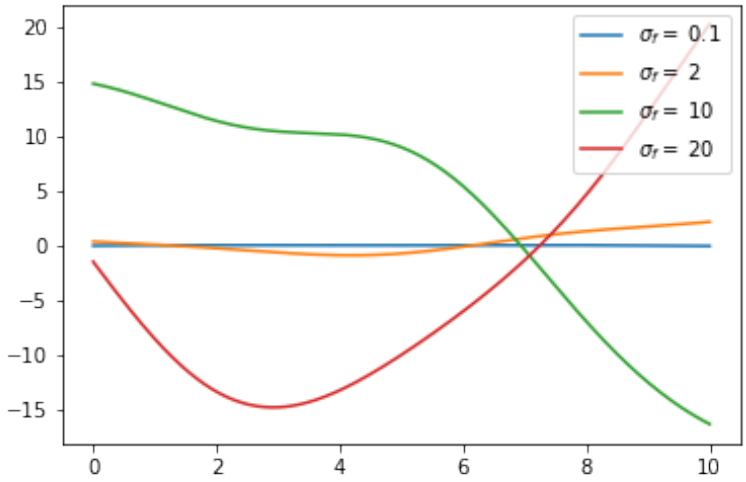


Figure 2: Priors sampled using a Gaussian kernel with $\sigma_f \in [0.1, 2, 10, 20]$ and $\ell = 3$.

- we should expect that around 68% of datapoints lie within $\pm\sigma_f$ of the mean (0 for the cases above)
- σ_f can also be called the **marginal variance** for a function value $\tilde{f}_i$:

$$\sigma_f^2 = Var[\tilde{f}_i] = K(\underline{x}^{(i)}, \underline{x}^{(i)})$$

- **How does the lengthscale affect the priors sampled from a GP?**
  - as the **lengthscale** gets smaller, we see that the term in the exponential gets more and more negative, meaning that the covariance will approach 0
  - this indicates that changes in features are less likely to be correlated, so we expect the function to change more sharply
  - on the other hand, larger ℓ_d encourages smooth functions
  - hence, ℓ_d will control how often "turning points" appear in the function, with a turning point appearing approximately with distances of ℓ between them:

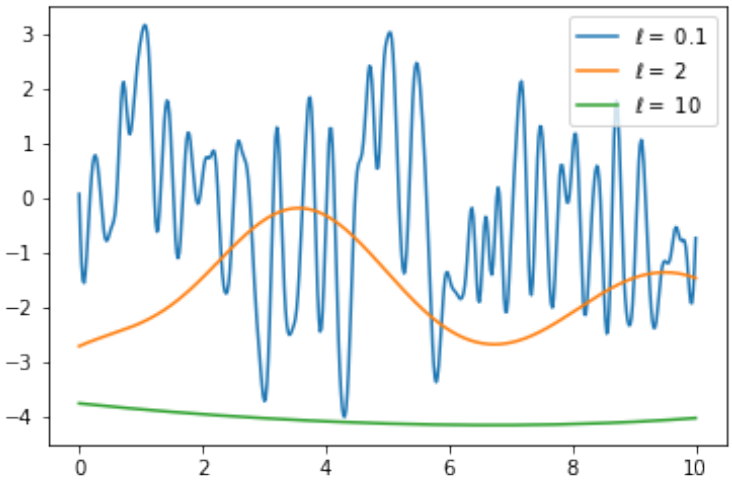


Figure 3: Priors sampled using a Gaussian kernel with $\ell \in [0.1, 2, 10]$ and $\sigma_f = 2$.

1.3.2 Learning Hyperparameters

- **What are the hyperparameters in Gaussian Processes?**
  - the **parameters** used to define the **kernel**, alongside the noise variance σ_y^2
  - for example, if we used a Gaussian Kernel, the hyperparameters would be:

$$\theta = \{\sigma_y^2, \sigma_f^2, \{\ell_d\}_{d\in[1,D]}\}$$

- **How can we learn the hyperparameters of a Gaussian?**

1. **Maximum Likelihood Optimisation**: the **marginal likelihood** of a GP is:

$$P(\underline{y} \mid X, \theta) = \mathcal{N}(\underline{y}; \underline{0}, K(X,X) + \sigma_y^2 \mathbb{I})$$

where $\underline{y}$ are our observations, and X is the data matrix. This is just the pdf of a multivariate Gaussian. Hence, this can be optimised by using **grid-search** on a **validation set**, or by exploiting the easy differentiability and using a **gradient-based optimiser** (this is best when there are a lot of parameters)

> *For reference, the log marginal likelihood is:*
>
> $$\log P(\underline{y} \mid X, \theta) = -\frac{1}{2}\underline{y}^T M^{-1}\underline{y} - \frac{1}{2}\log|M| - \frac{N}{2}\log 2\pi$$
>
> *where:*
>
> $$M = K(X,X) + \sigma_y^2\mathbb{I}$$
>
> *The hyperparameters θ are what define M.*

2. **Bayesian Approach**: alternatively, we can use **marginalisation** to write:

$$P(\underline{f}_* \mid \underline{y}, X) = \int P(\underline{f}_*, \theta \mid \underline{y}, X)d\theta = \int P(\underline{f}_* \mid \underline{y}, X, \theta)P(\theta \mid \underline{y}, X)d\theta$$

However, this can't be computed exactly (the term $P(\theta \mid \underline{y}, X)$ needs to be approximated)

- **How can we prevent overfitting of the parameters?**
  - the maximum likelihood optimisation approach can lead to hyperparameter **overfitting** (the Bayesian approach won't)
  - to prevent this, we can **regularise** the **log noise variance**:

    $$\log\ \sigma_y^2$$

    to avoid it from becoming too small

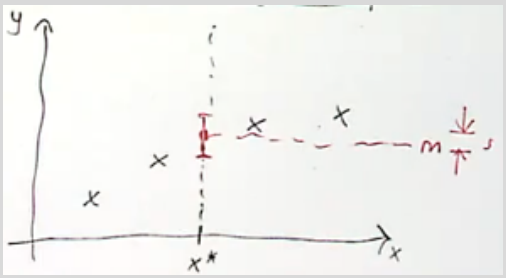
1.3.3 Importance of Good Hyperparameters

Recall, last week we showed that when predicting a new point we obtained the following parameters:

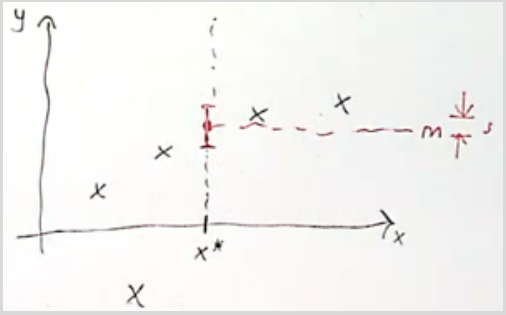
$$\mu_* = \underline{k}_*^T M^{-1} \underline{y}$$

$$\sigma_*^2 = k(\underline{x}_*, \underline{x}_*) - \underline{k}_*^T M^{-1} \underline{k}_*$$

So with GPs, we are always more confident when we make a prediction, since $M = K(X, X) + \sigma_y^2 \mathbb{I}$ is positive definite.
Now, say we have a bunch of well-defined observations, and we make a prediction:



However, if we then have a surprising observation:



our uncertainty about our prediction won't necessarily increase (this is what the formula says).
*Here is when **hyperparameter** fitting is important. Without the surprising data point, we might be confident that the data behaves relatively **smoothly**. If we didn't have hyperparameters, after seeing the surprising data point our model wouldn't change much. However, with hyperparamers, we can think that we would obtain a better fit by making ℓ_d (for example) smaller, since we should expect our original function to be a bit more "wriggly".*

1.3.4 Practical Tips for Fitting Hyperparameters

- *Always visualise the data, and look for weird artifacts which might not be captured by the model*
- *Consider encoding inputs/outputs to improve performance*
- *When possible, use domain knowledge to set* ***initial hyperparameters***
- *Otherwise:*
  - ***standardise*** *the* ***input****, and set* $\ell_d \approx 1$
  - ***standardise*** *the* ***targets*** *and set* $\sigma_f^2 \approx 1$
  - *set the noise* σ_y^2 *to a high level; this will make optimisation easier*

1.4 Limitations of Gaussian Processes

Gaussian processes provide an extremely flexible framework for modelling expensive functions (and even non-functions, like graphs). However, this power comes with certain flaws:

1. **Poor Performance with Large Datasets**: if N is the dataset size:
   - inverting/factoring the covariance matrix M is $\mathcal{O}(N^3)$
   - computing the **kernel** matrix is $\mathcal{O}(DN^2)$
   - the **kernel** matrix requires $\mathcal{O}(N^2)$ memory, which unfeasible in certain situations
2. **Not an Omnirepresentative Model**: for example, **monotonic** functions can't be represented by a GP, since given 2 observations, the probability of our prediction between the 2 observations violating the monotonic assumption is non-zero
3. **The Gaussian Assumption**: if we model processes which don't follow a Gaussian, we rely on using approximations

1.5 Exploring GPs via Code

- GPs for CO2 Prediction
- Visualising GPs with Different Kernels
- GPs in PyTorch
- GPs in Tensorflow

2 Reparametrisation and Convexity for Fitting Probabilistic Models

2.0.1 Reparametrisation

- **Why shouldn't we use constrained parameters in unconstrained optimisation?**

- consider **Bayesian Linear Regression**, with noise variance σ_y
- we showed that the negative log-likelihood was:

$$-\log(P(\underline{y} \mid X, \underline{w})) = \frac{N}{2}\log(2\pi\sigma_y^2) + \frac{1}{2\sigma_y^2}\sum_{n=1}^{N}(y^{(n)} - f(\underline{x}^{(n)}, \underline{w}))^2$$

- thus:

$$\sigma_y \to 0 \implies -\log(P(\underline{y} \mid X, \underline{w})) \to \infty$$

which means that our model is becoming more confident (since σ_y becomes infinitesimal), but the model is getting most things wrong (since the -ve log-likelihood is increasing, meaning $P(\underline{y} \mid X, \underline{w})$ is approaching 0

- this issue arises because σ_y is **constrained** (to be non-negative), whilst our optimisation technique is **unconstrained** (for example, with SGD:

$$\sigma_y \leftarrow \sigma_y - \eta\frac{\partial \mathcal{L}}{\partial \sigma_y}$$

means that σ_y can potentially take any set of values, even negative ones)

- **What is reparametrisation?**
  - a way of converting **constrained** variables into new, **unconstrained** variables
  - we can then use an **unconstrained optimiser** to optimise the **unconstrained** variable
  - if c is our **cost**, w is our **constrained** variable and v is w after being **reparametrised**, then by the chain rule:

$$\frac{\partial c}{\partial v} = \frac{\partial c}{\partial w} \times \frac{\partial w}{\partial v}$$

  - if the optimiser still pushes the unconstrained variable to an extrema (i.e $v \to \infty$, which could correspond to sending w towards one of its constraints), we can always use **regularisation** to prevent this
- **What types of reparametrisations can be used?**
  1. **To Positive**: use the **exponential**:

$$v = \exp(w)$$

  2. **Positive to Unconstrained**: use the **logarithm**:

$$v = \log(w)$$

  3. **Unconstrained to** $(0, 1)$: use the **logistic sigmoid**:

$$v = \sigma(w) = \frac{1}{1 + e^{-w}}$$

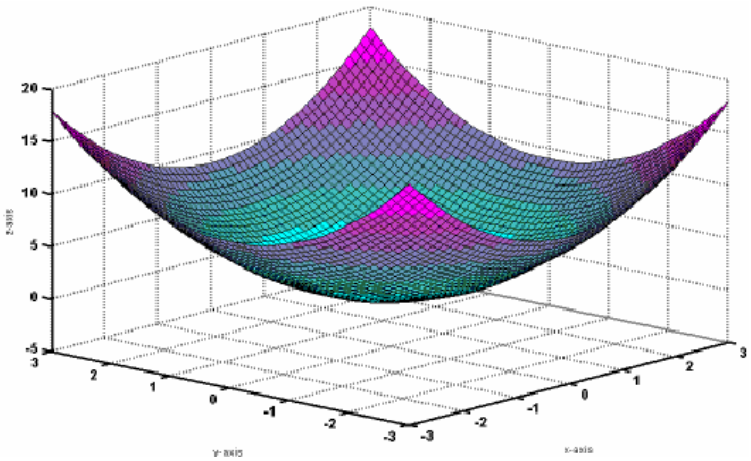
  4. $(0, 1)$ **to Unconstrained**: use the **logit**:

$$v = logit(w) = \log\left(\frac{w}{1 - w}\right)$$

- **How does reparemtrisation affect model fitting at boundaries?**
  - it is perfectly feasible to fit a GP using $\sigma_y = 0$ (the covariance matrix just goes from $K(X, X) + \sigma_y\mathbb{I}$ to $K(X, X)$
  - however, if we reparametrise using log, we can no longer fit points to it (since $\log(0) \to -\infty$)

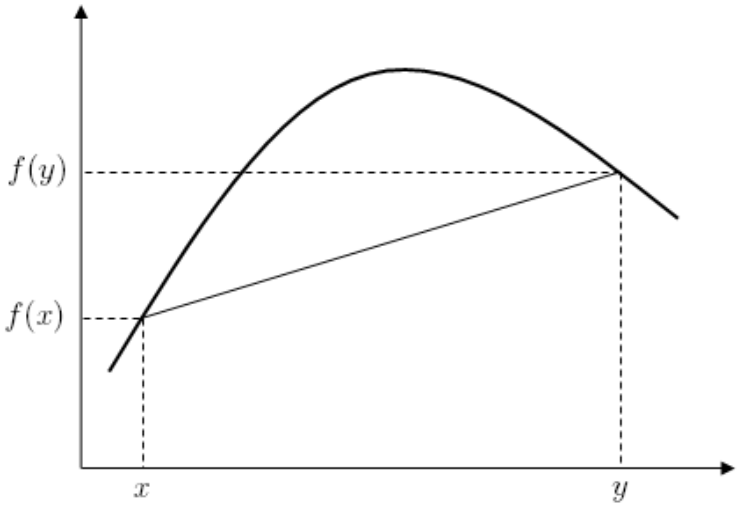
2.0.2 Convexity

- **What is a convex function?**
  - a **convex** function is one such that any line between two points in its surface lies entirely **above** the surface



  - as a reminder, use con**V**ex, and think that a parabola (which is convex) is shaped like a U/V

- **What is a concave function?**
  - a **concave** function is one such that any line between two points in its surface lies entirely **below** the surface



  - as a reminder, use con**V**ex, and think that a parabola (which is convex) is shaped like a U/V

- **What is a strictly convex function?**
  - let f be a function, and $\lambda \in [0, 1]$
  - f is **convex** if $\forall x, y, \lambda$:

$$f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y)$$

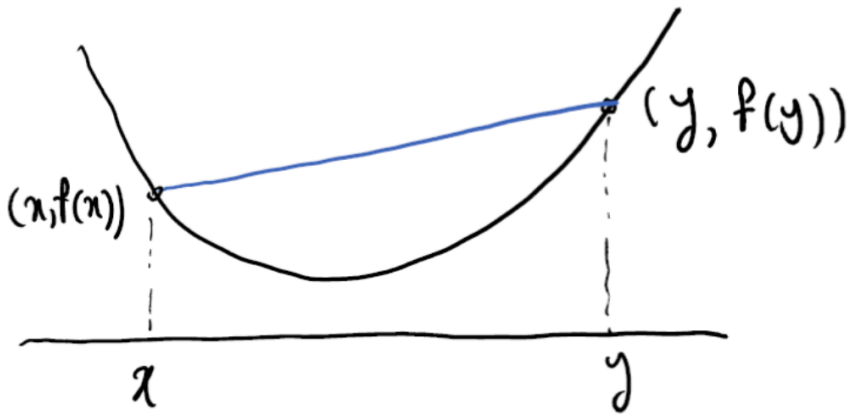


Figure 4: We can consider a point in $\mathbb{R}$ between x, y, given by $\lambda x + (1-\lambda)y$. The value at $\lambda x + (1-\lambda)y$ of the line between $f(x)$ and $f(y)$ will be $\lambda f(x) + (1-\lambda)f(y)$. Hence, this requirement is just stating how we defined convexity in the first place.

  - f is **strictly convex** of $\forall x, y$ and $\lambda \in (0,1)$ we have:

$$f(\lambda x + (1-\lambda)y) < \lambda f(x) + (1-\lambda)f(y)$$

  - notice, this formulae tells us that the sum of 2 convex functions will again be convex (if both functions satisfy the inequality, so will their sum)

- **How many extrema does a strictly convex function have?**
  - **strictly convex** functions have a **unique** extremum
- **Why are strictly convex loss functions desirable?**
  - strictly convex functions have a **unique** minium
  - there are many algorithms which ensure convergence to this local minimum for **convex** loss functions

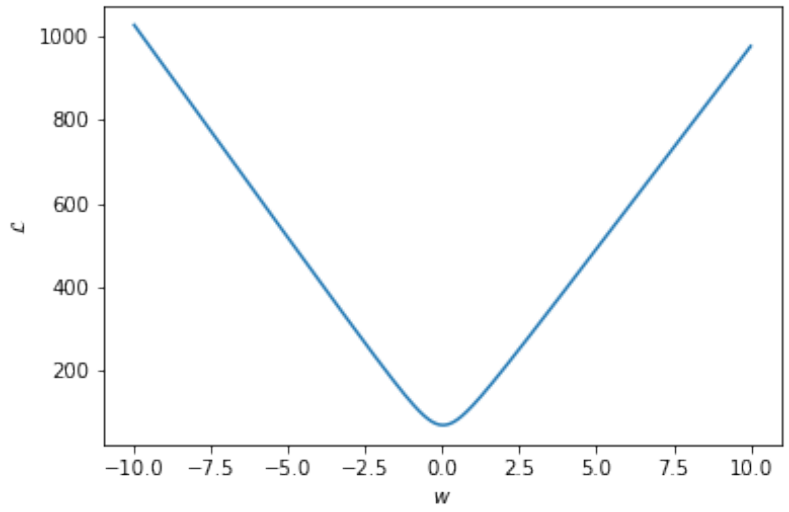


Figure 5: The logistic regression loss given by $\sum_{n=1}^{N} \log\left(\sigma((2y^{(n)}-1)\underline{w}^T\underline{x}^{(n)})\right)$ is **convex**. This uses linearly spread x, with the target being a Bernoulli trial.

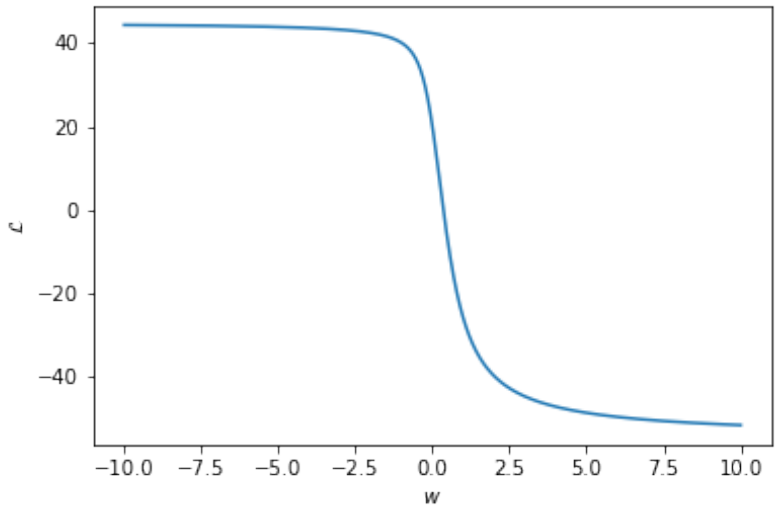


Figure 6: The square loss given by $\sum_{n=1}^{N}(y^{(n)}-\sigma(\underline{w}^T\underline{x}^{(n)}))^2$ is **not** convex. This uses linearly spread x, with the target being a Bernoulli trial.

- **Are typical ML problems convex optimisation problems?**
  - in general, most problems won't be **convex** (for example, optimising the location of basis functions)
  - we can still try reducing loss using **gradient methods**, which will result in parameters which are **reasonable**

- however, we won't have any guarantee of **optimality**: there might be a better optimiser which finds better parameters

- **Why use log for optimisation problems?**

  1. **Mathematical Convenience**: it converts products of probabilities into sums, which is easier to differentiate
  2. **Numerical Underflow**: representing very small numbers (i.e 0.5^{1000}) is easier using logs (i.e $1000\log(0.5)$) (these small numbers arise when working with probabilities)
  3. **Convexity**: a non-convex loss function might become convex with a log transformation (i.e finding the mean of the negative log of a Gaussian PDF via gradient methods can be done, since the ideal cost function will be convex; however, the Gaussian PDF is not convex)

3 Softmax for Classification

- **What is softmax regression?**
  - a method for **multiclass classification**
  - generalises **logistic regression**
- **What does the softmax model output?**
  - if we have K classes, we can represent the target $\underline{y}$ for a given input $\underline{x}$ as a **one-hot** vector
  - if $\underline{y}$ represents class c, then
$$y_k = \delta_{kc}$$
  - **softmax** outputs a vector $\hat{\underline{y}}$, where y_k represents the probability it assigns to class k
- **How does softmax differ from a one-vs-rest classifier?**
  - previously, we saw multiclass classification in terms of training K classifiers, which predicted whether a given input was class k or not with a probability
  - **softmax** computes the probabilities **all at once**
- **How does softmax compute the probability vector?**
  - the **softmax** model has a **weight matrix** W
  - the probability of assigning class k for an input $\underline{x}$ will be:
$$P(y_k = 1 \mid \underline{w}, W) = f_k(\underline{x}; W) = \frac{\exp((\underline{w}^{(k)})^T \underline{x})}{\sum_{j=1}^{K} \exp((\underline{w}^{(j)})^T \underline{x})}$$
  - if $W \in \mathbb{R}^{K \times D}$, the $\underline{w}^{(j)}$ can be thought of as the **row vectors** of W, and we can write:
$$\underline{f}(\underline{x}; W) = softmax(W\underline{x})$$
    to represent the vector output
- **Intuitively, how are the weight vectors adjusted for class classification?**
  - if $\underline{w}^{(k)}$ is the weight vector for class k, then $(\underline{w}^{(k)})^T \underline{x}$ will be **largest** when $\underline{w}^{(k)}$ and $\underline{x}$ are **parallel**
  - another factor which influences the value of $(\underline{w}^{(k)})^T \underline{x}$ is the **magnitude** of $\underline{x}$ and $(\underline{w}^{(k)})^T \underline{x}$
  - if a model just learns large weights for any input $\underline{x}$, then $\exp((\underline{w}^{(k)})^T \underline{x})$ will be large ... but so will all the other $\exp((\underline{w}^{(k)})^T \underline{x})$

- hence, the probability of class k will be small if some other weight $\underline{w}^{(j)}$ is larger
- thus, a model will learn weights, such that for a given class, most elements of the class are as parallel as possible to $\underline{w}^{(k)}$

- **When is an estimator consistent?**
  - when, given **infinite data**, the **estimator** will produce the **true paramaters** which generated the data

- **What loss functions can be used for softmax parameter optimisation?**
  - both **least squares** and **maximum likelihood estimatation** are **consistent**, and will fit optimal parameters which "explain" the observed data
  - however, MLE is slightly better since:
    * it has a **faster asymptotic convergence**
    * it **heavily penalises** confident but incorrect predictions

- **What is the optimisation procedure for softmax?**
  - the **log-likelihood** for a **single** observation $\underline{x}$ with label c is:

$$\begin{aligned}\log(P(y_c = 1 \mid \underline{x}, W) &= \log(f_c(\underline{x}; W)) \\ &= \log\left(\frac{\exp((\underline{w}^{(c)})^T \underline{x})}{\sum_{j=1}^{K} \exp((\underline{w}^{(j)})^T \underline{x})}\right) \\ &= \log\left(\exp((\underline{w}^{(c)})^T \underline{x})\right) - \log\left(\sum_{j=1}^{K} \exp((\underline{w}^{(j)})^T \underline{x})\right) \\ &= (\underline{w}^{(c)})^T \underline{x} - \log\left(\sum_{j=1}^{K} \exp((\underline{w}^{(j)})^T \underline{x})\right)\end{aligned}$$

  - hence, we can compute the **gradient** with respect to some weight $\underline{w}^{(k)}$:

$$\nabla_{\underline{w}^{(k)}} \log(f_c(x; W)) = \delta_{kc}\underline{x} - \frac{1}{\sum_{j=1}^{K} \exp((\underline{w}^{(k)})^T \underline{x})} \exp((\underline{w}^{(k)})^T \underline{x})\underline{x} = (y_k - f_k(\underline{x}; W))\underline{x}$$

    by using $y_k = \delta_{kc}$, since y_k is a binary target.
  - then, we can apply **stochastic gradient ascent** (since we haven't negated the log-likelihood) for each weight vector:

$$\underline{w}_k \leftarrow \underline{w}_k + \eta \nabla_{\underline{w}^{(k)}} \log(f_c(x; W))$$

> *Notice what the weight is telling us: $\underline{w}^{(k)}$ will be pushed to be more parallel to $\underline{x}$; the degree of the push will be based on the disparity between the prediction f_k and the actual label y_k. If f_k is close to y_k, the gradient will be small, indicating that $\underline{w}^{(k)}$ is already a good weight; otherwise, the gradient will be much larger.*

– alternatively, we can apply **batch gradient descent** where we use as gradient:

$$\nabla_{\underline{w}^{(k)}} \sum_{n=1}^{B} \log(f_{c(n)}^{(n)} = \sum_{n=1}^{B} (y_k^{(n)} - f_k(\underline{x}^{(n)}))\underline{x}^{(n)}$$

- **In what sense are weights redundant in softmax regression?**
  - our **softmax** model is defined by:

$$P(y_k = 1 \mid \underline{x}, W) = \frac{\exp((\underline{w}^{(k)})^T \underline{x}}{\sum_{j=1}^{K} \exp((\underline{w}^{(j)})^T \underline{x}}$$

  - we can divide the top and bottom through by $\exp((\underline{w}^{(K)})^T \underline{x}$ to obtain:

$$P(y_k = 1 \mid \underline{x}, W) = \frac{\exp((\underline{w}^{(k)} - \underline{w}^{(K)})^T \underline{x}}{\sum_{j=1}^{K} \exp((\underline{w}^{(j)} - \underline{w}^{(K)})^T \underline{x}}$$

  - thus, define a new model $\tilde{W}$ by:

$$\tilde{\underline{w}}^{(k)} = \underline{w}^{(k)} - \underline{w}^{(K)}$$

  - $\tilde{W}$ will yield the exact same predictions, and:

$$\tilde{\underline{w}}^{(K)} = \underline{0}$$

    will be "redundant"

4 Robust Logistic Regression via Probabilistic Modelling

4.1 Classification and Corrupted Data

- **How can softmax/logistic regression be affected by extreme outliers?**
  - a given sample might be **corrupted**, with some features having extreme values
  - **logistic/softmax** regression will then learn weight vectors for the class which are small for those features
  - however, this might not be the best strategy, if the non-corrupted samples don't behave in this way
- **How can we deal with corrupted inputs when training a classification model?**
  - **Magnitude Limitation** or **Unit Length**: enforce that feature vectors have a bounded magnitude
  - **Binary Features**: effect of corruption would just involve flipping ones and zeroes
  - **Outlier Detection**: discard faulty input vectors
  - **Optimisation Limitation**: limit how much weights can be updated by a given sample, thus reducing the effect of outliers
  - **Probabilistic Modelling**: assume the corruption is produced by noise, and incorporate this into the model

4.2 Dealing with Corruption Probabilistically

- **How can we model input corruption?**
  - the **labels** can be modified to include a **model** of label corruption
  - this will provide us with a new **loss**, which can be optimised as before
- **What types of corruption models can we use?**
  - let m be a binary indicator, with $m = 0$ indicating corruption, and $m = 1$ indicating no corruption
  - we can model m using a **Bernoulli distribution**:

$$P(m \mid \varepsilon) = Bernoulli(m; 1-\varepsilon) = \begin{cases} 1-\varepsilon, & m = 1 \\ \varepsilon, & m = 0 \end{cases}$$

  - then we can modify our logsitic regression in 2 ways (but not limited to these 2 ways):
    1. **Uniform Corruption**: we can pick the label for an input $\underline{x}$ uniformly randomly when data is corrupted:

$$P(y = 1 \mid \underline{x}, \underline{w}, m) = \begin{cases} \sigma(\underline{w}^T \underline{x}), & m = 1 \\ \frac{1}{2}, & m = 0 \end{cases}$$

    2. **Flip Corruption**: alternatively, we can flip the label:

$$P(y = 1 \mid \underline{x}, \underline{w}, m) = \begin{cases} \sigma(\underline{w}^T \underline{x}), & m = 1 \\ 0, & m = 0 \end{cases}$$

  - we shall proceed using the **uniform corruption model**
- **How can we incorporate the indicator variable m when computing the probability assigned by our classification model?**
  - if we observed m directly, we could just remove corrupted samples
  - since we can't observe it, if we know the probability of corruption ε, we can **marginalise** over m, by using the sum and product rules:

$$\begin{aligned} P(y = 1 \mid \underline{x}, \underline{w}, \varepsilon) &= \sum_{m \in \{0,1\}} P(y = 1, m \mid \underline{x}, \underline{w}, \varepsilon) \\ &= \sum_{m \in \{0,1\}} P(y = 1 \mid \underline{x}, \underline{w}, \varepsilon, m) P(m \mid \underline{x}, \underline{w}, \varepsilon) \end{aligned}$$

  - since corruption happens independently of the observed samples:

$$P(m \mid \underline{x}, \underline{w}, \varepsilon) = P(m \mid \varepsilon)$$

  - moreover, once we know m, knowing ε is irrelevant, so:

$$P(y = 1 \mid \underline{x}, \underline{w}, \varepsilon, m) = P(y = 1 \mid \underline{x}, \underline{w}, m)$$

  - thus, our model becomes:

$$P(y = 1 \mid \underline{x}, \underline{w}, \varepsilon) = \sum_{m \in \{0,1\}} P(y = 1 \mid \underline{x}, \underline{w}, m) P(m \mid \varepsilon)$$

- this is easily computed:

$$P(y = 1 \mid \underline{x}, \underline{w}, \varepsilon) = (1 - \varepsilon)\sigma(\underline{w}^T \underline{x}) + \frac{\varepsilon}{2}$$

- **What is the gradient for the robust logistic regression model?**
  - recall, with standard logistic regression we used the properties of the sigmoid to define:

$$\sigma_n = \sigma(z^{(n)} \underline{w}^T \underline{x}^{(n)})$$

    where $z^{(n)} = (2y^{(n)} - 1)$ is a label in $\{-1, +1\}$
  - we derived that:

$$\nabla_{\underline{w}} \log(\sigma_n) = (1 - \sigma_n) z^{(n)} \underline{x}^{(n)}$$

    and:

$$\frac{d}{dx}\sigma(x) = \sigma(x)(1 - \sigma(x))$$

  - then:

$$\begin{aligned}
\nabla_{\underline{w}} \log(P(z^{(n)} \mid \underline{x}^{(n)}, \underline{w})) &= \nabla_w \log\left((1 - \varepsilon)\sigma_n + \frac{\varepsilon}{2}\right) \\
&= \frac{1}{(1 - \varepsilon)\sigma_n + \frac{\varepsilon}{2}} (1 - \varepsilon)\sigma_n (1 - \sigma_n) z^{(n)} \underline{x}^{(n)} \\
&= \frac{1}{1 + \frac{\varepsilon}{2(1-\varepsilon)\sigma_n}} \nabla_{\underline{w}} \log(\sigma_n)
\end{aligned}$$

  - however, the loss will no longer be **convex**, so we don't have guarantees of optimal weights

> *Notice, if we set $\varepsilon = 0$, we recover our standard logistic regression model. After looking at this, it seems a bit weird to set $\varepsilon = 0$: it would be like assuming that all data is perfect!*

- **How will the robust model compare with the standard logistic regression model?**
  - as $\varepsilon \to 0$, $\frac{\varepsilon}{1-\varepsilon} \to 0$, and robust logistic regression behaves like normal logistic regression
  - as $\varepsilon \to 1$ $\frac{\varepsilon}{1-\varepsilon} \to \infty$, so the gradient becomes negligible
  - thus, if we are highly uncertain, the robust model will be very cautious about updating weights
  - if the probability σ_n is much smaller than ε, we again get negligible gradients, meaning that our robust model will be discouraged to greatly change the weights
  - hence, if we think that outliers are very likely, or our model detects a very unlikely input, they will be discarded by the model
- **How can we set ε?**
  1. **Domain Knowledge**: if we know the corruption rate, we can set ε manually
  2. **Grid Search**: try a grid of settings to see which setting best models data
  3. **Gradient Optimisation**: whilst $\varepsilon \in [0, 1]$, we can apply a **logit** transform so that $logit(\varepsilon) \in \mathbb{R}^+$. Then, we can apply the standard gradient optimisation techniques via:

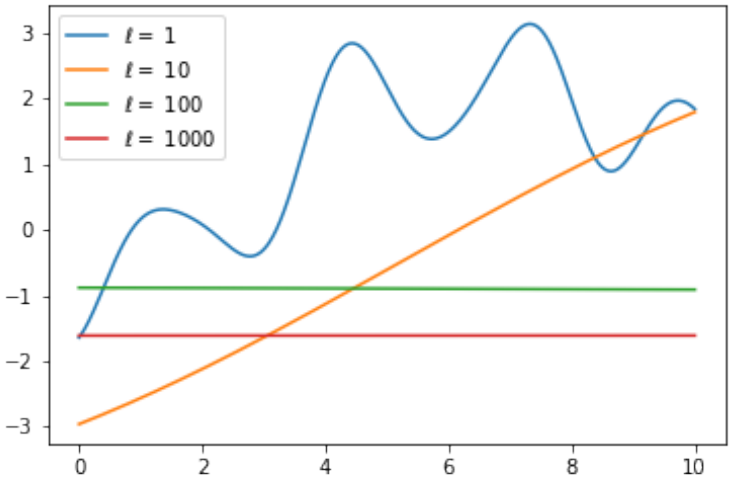
$$a = logit(\varepsilon) \implies \frac{\partial \mathcal{L}}{\partial a} = \frac{\partial \mathcal{L}}{\partial \varepsilon} \frac{\partial \varepsilon}{\partial a}$$

     whilst we update the weights of our model with $\nabla_{\underline{w}} \mathcal{L}$.

5 Question

5.1 Notes Questions

1. **How do the priors and posteriors look when sample from a GP with a kernel using a lengthscale ℓ which is very small?**
   - as $\ell \to 0$, the covarianc ebetween close samples will still be small
   - hence, we expect that the turning points appear more and more often, so prior samples will osciallate wildly
   - the posterior will look similar to this, and osciallte wildly, except possibly at the testing location, where the posterior will try to approximate these values
2. **How do prior samples look as $\ell_d \to \infty, \forall d \in [1, D]$?**
   - as ℓ_d grows, the term in the exponential approaches 0, so the covariance between any 2 points will be approximately σ_f^2, independent of distance between 2 points
   - hence, we expect a prior which looks like horizontal hyperplane



3. **How do prior samples look as $\ell_d \to \infty$ for a specific $d \in [1, D]$?**
   - elements along dimension d will have the same covariance, so we expect that the function along this dimension remains constant