CS419M

Lecture 11: Tutorial on Loss, Classification & Regression

17 Feb 2023

Lecturer: Abir De *Scribe: Groups 21 & 22*

This tutorial covers the problem based on loss functions, Regression and Classification.

Question 1

Assume that we are given a set of features $\{(x_i, y_i) \,|\, i \in \{1, 2, ..., N\}\}$ with $x_i \in R^d$, $y \in \{-1, +1\}$. We wish to train a function $h : R^d \to R$, so that $\text{Sign}(h(x)) = y$. To that aim, we seek to solve the following:

$$\underset{h\in}{\text{minimize}} \sum_{i=1}^{N} [\text{Sign}(h(x_i)) \neq y_i]$$

Moreover, H is the set of all functions that map from R^d to R. This problem is hard to solve in general. That is why, we resort to several approximations. In the following, mark and explain which ones are good approximator of $I[\text{Sign}(h(x_i)) \neq y_i]$ in the above equation.

$$(i) \quad \max\{0, 1 - y_i \cdot h(x_i)\} \quad \text{(Yes/No)}$$
$$(ii) \quad \min\{0, 1 - y_i \cdot h(x_i)\} \quad \text{(Yes/No)}$$
$$(iii) \quad \frac{\exp(-y_i \cdot h(x_i))}{1 + \exp(-y_i \cdot h(_i))} \quad \text{(Yes/No)}$$
$$(iv) \quad \frac{1}{1 + \exp(-y_i \cdot h(x_i))} \quad \text{(Yes/No)}$$

Solution:

$$\textbf{case} : \textbf{1} \quad if \quad y_i \quad and \quad h(x_i) \quad have \quad opposite \quad signs, \quad then$$
$$1 - y_i.h(x_i) \in [1, \infty)$$
$$\textbf{case} : \textbf{2} \quad if \quad y_i \quad and \quad h(x_i) \quad have \quad same \quad signs, \quad then$$
$$1 - y_i.h(x_i) \in (-\infty, 1]$$

The original loss function can take only discrete values 0 and 1. The good approximator of I[Sign $h(x_i) \neq y_i$] will be the one which penalizes based on how far the point is.

y_i	$h(x_i)$	$max\{0, 1 - y_i.h(x_i)\}$	$min\{0, 1 - y_i.h(x_i)\}$	original
H	H	L	H	L
H	L	H	L	H
L	H	H	L	H
L	L	L	H	L

Hence, (i) is a good approximator.

y_i	$h(x_i)$	(iii)	(iv)	original
H	H	L	H	L
H	L	H	L	H
L	H	H	L	H
L	L	L	H	L

(iii) ia also a good approximator.

Question 2

Suppose we restrict $h(x) = w^T x + b$, *i.e.*, $h(x)$ is a linear function. Then write the approximation of the optimization problem defined in the above question in terms of any (correct) one approximation in the previous question. Specifically, fill up the gaps

$$\text{minimize} \sum_{i=1}^{N} ??$$

Solution:

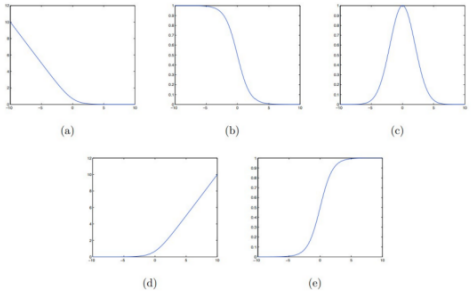
As specified in Q.1 we can conclude that,

$$\min_{w,b} \sum_{i=0}^{n} max\{0, 1 - y_i(w^\top x + b)\}$$

is the approximation of optimization problem.

Question 3

Suppose $h(x) = \text{sign}(f(x))$ where $h(x) : R^d \rightarrow \{+1, -1\}$. We now cosider a loss function defined as $\sum_{(x_i,y_i)} \ell(y_i f(x_i))$. *i.e.*, ℓ is a function of $yf(x)$. Given below are some graphs with x axis as $yf(x)$ and y axis as the loss ℓ value. Identify the graphs that are adept



Solution:

y	$f(x)$	$loss$	$y.f(x)$
H	H	L	H
H	L	H	L
L	H	H	L
L	L	L	H

Only graphs (a) and (b) satisfy above condition.

Question 4

Consider a Binary classification problem where the dataset D_{Train} is imbalanced. We have 90% examples that belong to class $+1$ and the remaining examples with class -1.

- What is your guess for the best $h \in$ All constant model?
- Compute $Error(h^*) - Error(\hat{h})$ for your guess. Assume that the test set is well-balanced.

Solution:

$h\hat{}(x_i)$ is a model which maximizes accuracy over D_{train} and $h^*(x_i)$ is a model which maximizes accuracy on D_{test}

part (a):

h is a constant model $\implies h(x_i) = C$
90% of the test data contains examples that belong to class +1. Hence, best guess for h would be +1.

part (b):

Test set is well balanced.
$h^*(x_i)$ = +1/-1 with the accuracy of 50%.
$h\hat{}(x_i)$ = +1 with 50% accuracy over test set.

$\therefore Error(h^*)$ - Error($h\hat{}$) = 0.

Question 5

Now, let us consider a weighted loss function given by:

$$\{w^*, b^*\} = \arg\min_{w,b} \sum_{i=1}^{M} r_i \max\left(0, \left(\frac{1}{2} - f(x_i)\right) y_i\right)$$

where $r_i > 0$ are weights associated with loss of each example. Can you propose a weighting scheme for r_i and justify your choice?

Repeat the exercise for the case when test set is also imbalanced with 60% test set examples that belong to class $+1$

Solution:

We have 90% examples in +1 and 10% in -1. So we need to choose r_i such that the training set gets balanced. The technique of weighting is utilized to address the bias in the training dataset. When the dataset is highly imbalanced, minimizing the unweighted loss function leads to a bias towards the majority class in the training set. This may be inappropriate if the test set is nearly balanced. In such a scenario, it is beneficial to assign greater weight to the loss from the minority class. One possible scheme could be choosing

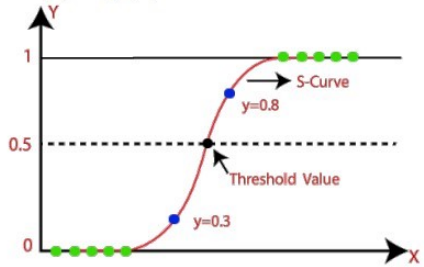
$$r_i = 9 \quad \text{if} \quad y = -1$$
$$r_i = 1 \quad \text{if} \quad y = +1$$

Here r_i values are such that a misclassification of -1 will be penalized 9 times more than a misclassification of +1. Another way to look at it is as if we are replacing the actual dataset with a new dataset where number of -1 is 9 times more than the original dataset to balance the test set. In case of 60% +1 and 40% -1 test set we need to take value of r_i less than 9

$$1 \leq r_i < 9 \quad \text{if} \quad y = -1$$
$$r_i = 1 \quad \text{if} \quad y = +1$$

Question 6

Recall that Logistic Regression model is given by: $h(x) = \frac{1}{1+e^{-w^T x}}$ where the labels are binary $\mathcal{Y} = \{0, 1\}$



And the loss that we minimize is called *cross-entropy* loss

$$\sum_{(x_j, y_j) \in D_{Train}} -\{y_i \log h(x_i) + (1 - y_i) \log(1 - h(x_i))\}$$

Finally the decision rule is given by $h(x_i) > 0.5$

- Argue that cross entropy loss is a valid loss function.
- What is $||w||$ when training loss is 0. Assume that all features have unit norm $||x|| = 1$
- Is it wrong, if we take $h(x) = \frac{1}{1+e^{-w^T x}}$. Can you tell verbatim, what interpretations change now?

Solution:

When y_i = +1 we are penalizing $-\log h(x_i)$
When y_i = 0 we are penalizing $-\log(1 - h(x_i))$ where $0 < h(x_i) \leq 1$

Part 1

y	h(x)	loss
H	H	L
H	L	H
L	H	H
L	L	L

H: High
L: Low

Part 2

For the training loss to be zero at y_i = +1, $h(x_i)$ has to be 1 and at y_i = 0, $h(x_i)$ has to be 0. Moreover, each term in the loss function is non-negative. Therefore they have to 0. Which gives $\|w\| = \infty$

Part 3

This can be obtained by a simple rotation about the y-axis, i.e. by putting x as -x, $h(x_i) \rightarrow 1 - h(x_i)$. The new loss function will be

$$\sum_{(x_i, y_i) \in \mathcal{D}_{\text{Train}}} -\{y_i \log(1 - h(x_i)) + (1 - y_i) \log h(x_i)\}$$

Question 7

Now given D_{Test}, the instructor allows you to change the model by modifying the decision rule as $h(x_i) > \tau$ where $\tau \in [0, 1]$. You are free to cheat by inspecting the test set and choosing a τ of your choice. However, you cannot change $\hat{w}, \hat{b}$. Let us evaluate the choices made by the following students:

- Naive student 1: Choose $\tau = 0$
- Naive Student 2: choose $\tau = 1$
- Millennial: choose $\tau = 0.5$
- What would the class choose? Can you pose it as an optimization problem by proposing a loss function and picking τ^* by means of minimizing it?

Solution:

If τ is chosen to 0 or 1 then all the points will be classified as the same class, i.e. for τ = 0, the model will always classify any point as +1 and for τ = 1 it will classify any point as 0.

For τ = 0.5, it is a good choice when we don't know anything about the test set, it will give the same value as $\hat{h}$. But if we have additional data about the test set then we can do better.
Optimized value of τ can be determined as

$$\tau^* = \text{argmin}_{\tau \in [0,1]} \quad \Sigma_{D_{test}}[\text{sign}(h(x_i) - \tau) \neq y_i]$$

Question 8

A function $f(x)$ is said to be linear in x if it satisfies the following two properties
(a) $f(x+y) = f(x) + f(y)$
(b) $f(\alpha x) = \alpha f(x)$
Are the following equations linear. If yes, then with respect to what parameters?

(a) $f(x) = w_1 * x_1 + w_2 * x_2$
(b) $f(x) = w_1 * x_1^2 + w_2 * x_2^3$
(c) $f(x) = w_1 * \ln x_1 + w_2 * e^{x_2}$
(d) $f(x) = x_1 * \ln w_1 + x_2 * e^{w_2}$
(e) $f(x) = w^T x \quad w, x \in \mathbb{R}^d$
(f) $f(x) = w^T x + b \quad w, x \in \mathbb{R}^d \quad b \in \mathbb{R}$

Solution:

Take

$$w = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \quad \text{and} \quad x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

1.
$$f(x+y) = w^T(x+y) = w^T x + w^T y = f(x) + f(y)$$
$$f(\alpha x) = w^T(\alpha x) = \alpha w^T x = \alpha f(x)$$
$$f(w_1 + w_2) = (w_1 + w_2)^T x = (w_1^T x + w_2^T x) = f(w_1) + f(w_2)$$
$$f(\alpha w) = (\alpha w)^T x = \alpha w^T x = \alpha f(w)$$
So linear in both w and x.

2.
$$f(x+y) = w_1 * (x_1 + y_1)^2 + w_2 * (x_2 + y_2)^3 \neq f(x) + f(y)$$
$$f(\alpha x) = w_1 * \alpha^2 x_1^2 + w_2 * \alpha^3 x_2^3 \neq \alpha f(x)$$
So linear in w but not in x.

3.
$$f(x+y) = w_1 * \ln(x_1 + y_1) + w_2 * e^{(x_2+y_2)} \neq f(x) + f(y)$$
$$f(\alpha x) = w_1 * \ln(\alpha x) + w_2 * e^{\alpha x} \neq \alpha f(x)$$
linear in w not in x.

4. Simply changing x and w in part (3), linear to x but in w.

5. Similar to (1), but more case, linear in both x and w.

6.
$$f(x+y) = w^T(x+y) + b = w^T x + w^T y + b = (w^T x + b) + (w^T y + b) - b \neq f(x) + f(y)$$
$$f(\alpha x) = w^T(\alpha x) + b = \alpha w^T x + b \neq \alpha f(x)$$
$$f(w_1 + w_2) = (w_1 + w_2)^T x + b = (w_1^T x + w_2^T x) + b \neq f(w_1) + f(w_2)$$
$$f(\alpha w) = (\alpha w)^T x + b = \alpha w^T x + b \neq \alpha f(w)$$
If we take w$'$ = [b w_1 w_2 ...]^T and x$'$ = [1 x_1 x_2 ...] then we can write f(x) as $f(x) = w'^T x'$.
Which is linear in both w' and x'.

Question 9

L-2 Loss in case of linear regression was defined as follows

$\mathcal{L}_2(w) = \sum_{i=1}^{N}(y_i - wx_i - b)^2$

$x_i \in \mathrm{R}$, $w \in \mathrm{R}$, $b \in \mathrm{R}$

The interesting thing about linear regression is there exist a closed form solution. This means that the solution can be calculated by minimizing the above function.

Take a gradient of the loss function stated above and prove that the solutions for 1-dimensional case are

$\widehat{w} = \sum_{i=1}^{N} \frac{(x_i - \overline{x})(y_i - \overline{y})}{(x_i - \overline{x})^2}$

$\widehat{b} = \overline{y} - \widehat{w}\overline{x}$

Solution:

$$L_2(w,b) = \sum_{i=1}^{N}(y_i - wx_i - b)^2$$

$$\frac{\partial L_2(w,b)}{\partial w} = 0 \; at \; optimal \; \hat{w} \; and \hat{b}$$

$$\sum_{i=1}^{N} 2(y_i - wx_i - b)x_i = 0$$

$$\sum[(y_i - \bar{y}) - w(x_i - \bar{x})]\,(x_i - \bar{x}) = 0$$

$$\hat{w} = \frac{\sum(y_i - \bar{y})(x_i - \bar{x})]}{\sum(x_i - \bar{x})^2}$$

$$\frac{\partial L_2(w,b)}{\partial b} = 0 \; at \; optimal \; w \; and \; b$$

$$\frac{\sum_{i=1}^{N} y_i}{N} - \hat{w}\frac{\sum_{i=1}^{N} x_i}{N} - \frac{\sum_{i=1}^{N} \hat{b}}{N} = 0$$

$$\hat{b} = \bar{y} - \hat{w}\bar{x}$$

Question 10

L-2 Loss in case of linear regression was defined as follows

$\mathcal{L}_2(w) = \sum_{i=1}^{N}(y_i - w^T x_i)^2$

This loss can be neatly written with the help of design matrix X and label vector Y

Prove that : $\mathcal{L}_2(w) = ||Xw - Y||^2$

Now we can take the gradient of the loss function stated above and prove that the solutions for general case. However while taking the gradient a little bit of matrix calculus will be used. We can then finally show that taking the gradient of $\mathcal{L}_2(w)$ and putting it to zero leads us to the normal equations

Derive $X^T Xw = X^T Y$

Solution:

$$L_2(w) = \sum_{i=1}^{N}(y_i - w^T x_i)^2$$

$$writing \; this \; in \; matrix \; form :$$

$$L_2(w) = \| \begin{bmatrix} (y_1 - w^T x_1) \\ \vdots \\ (y_n - w^T x_n) \end{bmatrix} \|^2$$

$$X = \begin{bmatrix} x_1^T \\ \vdots \\ x_n^T \end{bmatrix}$$

Hence this can be written as:

$$\|Xw - Y\|^2$$

Remember that:

$$\frac{\partial \|X\|^2}{\partial x} = 2X$$

$$\frac{\partial Ax}{\partial x} = A^T$$

So we get:

$$\frac{\partial \|Xw - Y\|^2}{\partial x} = 2X^T(Xw - Y)$$

$$X^T(Xw - Y) = 0$$

$$X^T Xw - X^T y = 0$$

$$X^T Xw = X^T y$$

$$w = (X^T X)^{-1} XY$$

Question 11

Design Matrix $X \in \mathbb{R}^{nXd}$ is a matrix where all samples of the dataset are stacked one below the other. More specifically

$$X = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & x_3^{(1)} & . & . & x_d^{(1)} \\ x_1^{(2)} & x_2^{(2)} & x_3^{(2)} & . & . & x_d^{(2)} \\ x_1^{(1)} & x_2^{(1)} & x_3^{(1)} & . & . & x_d^{(3)} \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ x_1^{(n)} & x_2^{(n)} & x_3^{(n)} & . & . & x_d^{(n)} \end{bmatrix}$$

Here $x_k^{(i)}$ is the k^{th} feature of i^{th} datapoint vector
Recall that the closed form solution of L-2 regression is $(X^TX)^{-1}X^TY$
Prove that the inverse of X^TX exist.

Solution:

X^TX is invertible if X has a full column rank

If a matrix A has a full column rank then

$Ax = b$ has only one solution i.e. x = 0

so here we haveA^TA

$$A^TAx = 0$$
$$\implies x^TA^TAx = 0$$
$$\implies \|Ax\| = 0$$
$$\implies Ax = 0$$
$$\implies x = 0$$

Hence we can say X^TX has a full column rank and is invertible.

Question 12

Although $(X^TX)^{-1}$ does not always exist. $(X^TX + \lambda I)^{-1}$ however does exist. To prove this we will need to understand the definition of positive definite matrices

Given a $n \times n$ matrix M The condition for positive definiteness is

M positive-definite $\iff$ $\mathbf{v}^\mathsf{T}M\mathbf{v} > 0$ for all $\mathbf{v} \in \mathbb{R}^n \setminus \{\mathbf{0}\}$

A positive definite matrix has a non zero determinant. Therefore its inverse always exists.

Can you prove that $(X^TX + \lambda I)$ is positive definite

Solution:

we have: $(X^TX + \lambda I)$

$$V^T(X^TX + \lambda I)V$$
$$\implies V^TX^TXV + \lambda V^TV$$
$$\implies \|XV\| + \lambda\|V\| \geq 0$$

Hence $(X^TX + \lambda I)$ is positive definite

Question 13

The Linear regression problem can be modelled in a probabilistic way under the assumptions

$$Y_i = w^Tx_i + \epsilon_i,$$

$\epsilon_i \sim$ N(0,,σ^2)

$Y_i \sim$ N($\mathrm{w}^Tx_i, , \sigma^2$)

Prove that the maximising the Likelihood of Data

$\mathcal{D} = \{x^{(i)}, y^{(i)}\}_{i=1}^n$

is equivalent to minimizing the l2-loss that we proposed earlier for the standard regression problem

Solution:

$$Y_i = w^T x_i + \varepsilon$$

Likelihood of data is given by:

$$P_i = \frac{1}{2} e^{-\frac{(y_i - w^T x_i)^2}{2\sigma^2}}$$

$$\Pi P = \frac{1}{2} e^{-\frac{\sum (y_i - w^T x_i)^2}{2\sigma^2}}$$

we can see that maximizing this would be equivalent to:

$$\min \sum (y_i - w^T x_i)^2$$

References

Lecture 12: Convexity, Dual Formulation and Similarity Measure

March 1, 2023

Lecturer: Abir De *Scribe: Group 23 & 24*

1 Introduction

Till now, we have seen that original problem of SVM is given by

$$\min_{\boldsymbol{w},\, b} \lambda ||\boldsymbol{w}||^2 + c \sum_{i \in \mathcal{D}} (1 - (y_i(\boldsymbol{w}^T \boldsymbol{x_i} + b))$$

This is a convex problem since its double derivative is a matrix with positive eigenvalues.

$$\frac{\partial \lambda ||\boldsymbol{w}||^2}{\partial \boldsymbol{w}} = 2\lambda \boldsymbol{w}$$

$$\frac{\partial^2 \lambda ||\boldsymbol{w}||^2}{\partial \boldsymbol{w}^2} = 2\lambda I_{dxd}$$

Here $||\boldsymbol{w}||^2$ is norm of the vector $\boldsymbol{w}$, so its derivative is a vector and the double derivative is a square matrix of dimension d. Since λ is positive, the double derivative will be positive and hence it becomes a convex optimization problem.

2 Convexity of Dual Formulation problem

The optimization problem for the dual formulation of SVM is given by

$$\max_{\boldsymbol{\alpha}} \sum_{i \in \mathcal{D}} \alpha_i - \frac{\sum_{i,j} \alpha_i \alpha_j y_i y_j {x_i}^T x_j}{2\lambda}$$

where,

$$\sum_{i \in \mathcal{D}} \alpha_i y_i = 0$$

$$0 \leq \alpha_i \leq C$$

Consider

$$G(\boldsymbol{\alpha}) = \sum_{i \in \mathcal{D}} \alpha_i - \frac{\sum_{i,j} \alpha_i \alpha_j y_i y_j {x_i}^T x_j}{2\lambda}$$

Like the original problem of SVM, here also the double derivative of $G(\boldsymbol{\alpha})$ will be a matrix whose ij^{th} entry is given by

$$[\frac{\partial^2 G(\boldsymbol{\alpha})}{\partial \boldsymbol{\alpha}^2}]_{(i,j)} = -\frac{y_i y_j {x_i}^T x_j}{2\lambda}$$

Since each element contains $y_i y_j(\boldsymbol{x}_i \cdot \boldsymbol{x}_j)$, interchanging i and j gives same element, means it is a symmetric matrix. Also all entries are real numbers so it is a real symmetric matrix.
Further, we can represent $G(\alpha)$ as

$$G(\boldsymbol{\alpha}) = \sum_{i \in \mathcal{D}} \alpha_i + \boldsymbol{\alpha}^T \frac{\partial^2 G(\boldsymbol{\alpha})}{\partial \boldsymbol{\alpha}^2} \boldsymbol{\alpha}$$

The matrix $\frac{\partial^2 G(\boldsymbol{\alpha})}{\partial \boldsymbol{\alpha}^2}$ has negative eigenvalues so the optimization for the dual formulation of SVM becomes a concave optimization problem. Hence we focus on maximizing $G(\boldsymbol{\alpha})$ here. While the original problem of SVM was a convex optimization problem, hence we minimized the function there.

3 Problem with Dual

While using dual, we have n variables $(\alpha_1, \alpha_2, \alpha_3, ...)$.
By comparison, the primal problem had d variables where $d \leq n$.
Thus with n parameters of dimension d, to store we need memory of order $O(||d^2||)$.
i.e.

$$G(\vec{\alpha}) = \sum_{i \in \mathcal{D}} \alpha_i + \vec{\alpha}^T \frac{\partial^2 G(\vec{\alpha})}{\partial \alpha^2} \vec{\alpha}$$

Since double derivative has terms consisting of $y_i y_j(\boldsymbol{x}_i \cdot \boldsymbol{x}_j)$, the number of computations required will be of order d^2. And we can neither diagonalize because for that too, we will require to store the matrix first in the memory.
The solution to this problem is that we choose a random variable $\boldsymbol{w}$ from some random distribution, perform mixing on it with $\boldsymbol{\alpha}$, and build a new function $\hat{G(\boldsymbol{\alpha})}$ such that the expectation of $\hat{G(\boldsymbol{\alpha})}$ over $\boldsymbol{w}$ equals to G($\boldsymbol{\alpha}$).

$$\hat{G(\boldsymbol{\alpha})} = g(\boldsymbol{w}, \boldsymbol{\alpha})$$
$$\mathbb{E}_{\boldsymbol{w}}[\hat{G(\boldsymbol{\alpha})}] = G(\boldsymbol{\alpha})$$

4 Similarity measure

The w can be represented in terms of $\boldsymbol{\alpha}$ as

$$w = \frac{\sum_i \alpha_i y_i x_i}{2\lambda}$$

Also, remember that

$$\sum_i \alpha_i y_i = 0$$

For example, take

$$\begin{aligned} \hat{y} &= sign(\boldsymbol{w}^T \boldsymbol{x} + b) \\ &= sign(\frac{\sum_{i,j} \alpha_i y_i {x_i}^T x_j}{2\lambda} + b) \end{aligned}$$

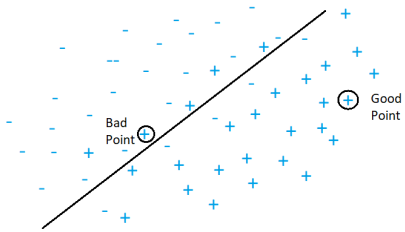
Here we are checking the similarity of x by doing ${x_i}^T x_j$ and then weighting similarity by y_i first and then again weighting it by α_i since $\sum_i \alpha_i y_i = 0$. So we are taking the weighted average of similarities.

But for correctly classified points, α_i = 0. So we are only observing misclassified points and points lying on the hyperplane. It sounds counter-intuitive since we are observing only misclassified points and points lying on hyperplane instead of correctly classified points.

Another strategy can be that we check the neighborhood and take the average of the labels. In this strategy, we will get correct result for good points but the wrong result for bad points.

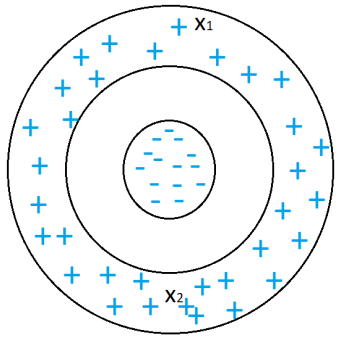
We want a unified strategy for all points. So will use the similarity measure strategy. Even if we are looking at misclassified/ boundary points only, it will take us max to the boundary only but it is fine since it also works for bad points.

So even though we are more confident in neighborhood checking method for good points, we use similarity measure method to get a uniform strategy for bad points as well.



similarity measure for a special case

Consider the dataset shown in the figure below.



As shown in the figure, the region containing '-' points is completely surrounded by the region containg '+' points. Consider two points x_1 and x_2 in the outer region. Now we have 2 choices for similarity measure.

$$(i) : sim(x_1, x_2) \propto -||x_1 - x_2||$$

or

$$(ii) : sim(x_1, x_2) \propto -|\ ||x_1|| - ||x_2||\ |$$

In this case, the distance from the origin is a more convenient measure of similarity; hence the second choice is better here. While in the first choice, it is directly taking the distance between two points. So if you consider points x_1 and x_2 of the diagram, although they are similar, the choice (i) of similarity will fail there. But as far as the purpose of SVM is concerned, choice (i) can also work as our similarity measure since we will be observing the nearest points only.

CS419

Lecture 13: Similarity Functions & SVM Continued

3^{rd} March 2023

Lecturer: Abir De *Scribe: Priyansh, Aadi, Shantanu, Vishruth*

1 Similarity Functions

Similarity learning is an area of supervised machine learning in artificial intelligence. It is closely related to regression and classification, but the goal is to learn a similarity function that measures two objects' similarity. It has applications in ranking, recommendation systems, visual identity tracking, face verification, and speaker verification.

The computation of these similarity functions becomes too expensive when we have an input feature with large dimensions (like 1 million). To address this issue, we introduce new approximation problems. Once we solve this problem, the computation becomes much faster and less resource-intensive in terms of storage and time.

These functions & methods are used extensively by search engines (like Google) to produce search results from a large basket of webpages.

1.1 Dot Product Similarity

$$\boldsymbol{Sim(X_i, X_j) = X_i^T X_j}$$

The approximation problem is to find a function *h*:

$$\boldsymbol{Sim}(X_i, X_j) = X_i^T.X_j \simeq h(X_i^T).h(X_j)$$
$$\ni dim(h(X_k)) << dim(X_k)$$

More formally, this can be restated as:

$$\mathop{\mathbb{E}}_{h \in H}[h(X_i^T).h(X_j)] = X_i^T.X_j$$

1.2 Cosine Similarity

$$\boldsymbol{Sim(X_i, X_j) = \frac{X_i^T.X_j}{||X_i||.||X_j||}}$$

The approximation problem is to find X_1 & Y_1:

$$\frac{X^T.Y}{||X||.||Y||} \simeq \frac{X_1^T.Y_1}{||X_1||.||Y_1||}$$
$$\ni dim(X_1)) << dim(X) \quad \& \quad dim(Y_1)) << dim(Y)$$

More formally, this can be restated as:

$$\mathop{\mathbb{E}}_{X_1 \in D(x), Y_1 \in D(y)} \left[\frac{X_1^T.Y_1}{||X_1||.||Y_1||} \right] = \frac{X^T.Y}{||X||.||Y||}$$

One such example of X_1 & Y_1 is:

$$X_1 = \begin{bmatrix} \sqrt{1-||X||^2} \\ X \\ 0 \end{bmatrix}, \quad Y_1 = \begin{bmatrix} 0 \\ Y \\ \sqrt{1-||Y||^2} \end{bmatrix}$$

But this approximation does not solve the problem of computation time and storage. We can take X_1 as the following and Y_1 in a similar manner:

$$X_1 = W.X$$
where X_1, W & X are matrices of dimension $d \times 1$, $d \times n$, $n \times 1$ respectively $\ni d \ll n$
& values of W are sampled from the Normal Distribution

Upon substitution of X_1 & Y_1 in the approximation problem, we get:

$$\frac{X_1^T.Y_1}{||X_1||.||Y_1||} = \frac{X^T.W^T.W.Y}{||X||.||Y||} \approx \frac{X^T.Y}{||X||.||Y||}$$

as $W^T.W \approx I$ *(Identity Matrix)*

Note: Formulating efficient approximations for Cosine Similarity is easier than Dot Product Similarity but is more time intensive (primarily because of the calculations of norms)

1.3 Distance Similarity

Distance similarity is a measure of similarity between two objects based on their distance in some metric space. The distance function defines the distance between two objects in the metric space, and the similarity function is typically defined as a decreasing function of the distance. For example,

$$\boldsymbol{Sim(X_i, X_j) = k.e^{\frac{||x_i - x_j||^2}{\Delta}}}$$

Many distance functions can be used to compute distance similarity; the choice of distance function can significantly impact the results of a clustering algorithm. Different distance functions may be more appropriate for different data types or applications.

1.4 Generalising Similarity

Generalising similarity functions aims to capture the similarity between two objects by mapping them into a higher dimensional space, where the similarity can be more easily defined. This is done by applying a function ϕ to each object that maps it into a feature space. The similarity between two objects x_i and x_j can then be defined as the dot product of their feature space mappings, $\phi(x_i)^\top \phi(x_j)$. However, computing this dot product in the feature space can be computationally expensive or even impossible if the dimension of the feature space is too high.

To address this issue, generalising similarity functions introduce a new function ψ, which maps each object into a new space that is typically of lower dimensionality than the feature space. This allows us to compute the similarity between two objects using the dot product of their lower-dimensional mappings, $\psi(x_i)^\top \psi(x_j)$. The function ψ is typically chosen to have certain desirable properties, such as being finite-dimensional, and it should be chosen such that it preserves the similarity between objects in the original feature space. In other words, if two objects are similar in the original space, their mappings should also be similar in the new space.

One way to ensure that the function ψ preserves similarity is to choose it such that it is a valid kernel function. A kernel function is a function that satisfies the properties of positive definiteness and symmetry and can therefore be used to define a valid similarity measure between objects. In particular, if ψ is a valid kernel function, then the dot product of any pair of mappings $\psi(x_i)^\top \psi(x_j)$ can be interpreted as the similarity between the corresponding objects x_i and x_j. Therefore,

$$Sim(x_i, x_j) = \phi(x_i^T).\phi(x_j) \approx \psi(x_i^T).\psi(x_j)$$
$$\ni Sim(x_i, x_i) \geq 0 \quad \& \quad Sim(x_i, x_j) = Sim(x_j, x_i) \quad \& \quad dim(\psi) << dim(\phi)$$

More formally, this can be restated as:

$$\mathop{\mathbb{E}}_{\psi \in D(\psi)} [\psi(x_i^T).\psi(x_j)] = \phi(x_i^T).\phi(x_j)$$

1.5 Random Fourier Kernel

For a general similarity function, if we represent it as $\boldsymbol{Sim}(x_i, x_j) = \phi(x_i)^T \phi(x_j)$ for some characteristic vector ϕ, then it could be an infinite vector. Therefore we want to approximate this function as $\psi(x_i)^T \psi(x_j)$ for some finite vector ψ. The way to do it is to use the inverse Fourier transform. We take the special case $\boldsymbol{Sim}(x_i, x_j) = F(x_i - x_j)$. Then,

$$F(x_i - x_j) = \int_{-\infty}^{\infty} g(w) e^{-\gamma w (x_i - x_j)} dw$$

where $\gamma^2 = -1$. We multiply and divide by some probability distribution p to get

$$\begin{aligned} F(x_i - x_j) &= \int_{-\infty}^{\infty} \frac{g(w)}{p(w)} e^{-\gamma w(x_i - x_j)} p(w) dw \\ &= \int_{-\infty}^{\infty} \left(\sqrt{\frac{g(w)}{p(w)}} e^{-\gamma w x_i} \right) \cdot \left(\sqrt{\frac{g(w)}{p(w)}} e^{\gamma w x_j} \right) p(w) dw \\ &= \mathop{\mathbb{E}}_{w \sim p(\cdot)} \left(\sqrt{\frac{g(w)}{p(w)}} e^{-\gamma w x_i} \right) \cdot \left(\sqrt{\frac{g(w)}{p(w)}} e^{\gamma w x_j} \right) \end{aligned}$$

Thus we have reduced F to an expectation and can use the Monte Carlo approximation to get:

$$F(x_i - x_j) \approx \frac{1}{|D|} \sum_{k \in D} \left(\sqrt{\frac{g(w_k)}{p(w_k)}} e^{-\gamma w_k x_i} \right) \cdot \left(\sqrt{\frac{g(w_k)}{p(w_k)}} e^{\gamma w_k x_j} \right)$$

for some large dataset D. Thu,s it is approximated by $\psi(x_i)^T \psi(x_j)$ for the finite vector

$$\psi(x) = \left[\sqrt{\frac{g(w_k)}{p(w_k)}} e^{\gamma w_k x_i} \right]_{k=1}^{|D|}$$

2 SVM Continued

A general SVM may have a loss function of the form:

$$\min_w \ f(y_1 w^T x_1, y_2 w^T x_2, \dots) + \lambda ||w||^2$$

We can write $x = u + v$ where u is perpendicular to span of $\{x_i\}$, and v is in span of $\{x_i\}$. In that case, $w^T x_i = v^T x_i$ because $u^T x_i = 0$, and $||w||^2 = ||u||^2 + ||v||^2 \geq ||v||^2$. Thus, replacing w with v will reduce the loss function. Therefore, the optimal w has the form $w* = \sum_{i \in D} \alpha_i y_i x_i$, even in this general case.

In case of a feature function $\phi(x)$, we have $w* = \sum_{i \in D} \alpha_i y_i \phi(x_i)$. The dimension of ϕ might be infinite. However, for classifying a new point, we need to find *sign* of $w^T \phi(x)$ (assume no bias).

$$\implies y = \textit{Sign} \left(\sum_{i \in D} \alpha_i y_i \phi^T(x_i) \phi(x) \right)$$

Here $\phi^T(x_i)\phi(x)$ has dimension 1, so we need to approximate this as $\psi^T(x_i)\psi(x)$ for some finite vector ψ. As discussed above, we can generally do this for a distance-based similarity function.

Lecture 14: Kernel Methods

10/03/2023

Lecturer: Abir De *Scribe: Groups 3 & 4*

1 Introduction to Kernel Methods

Most of the time real world data has a non-linear decision boundary, which is difficult to learn using simple models as they fail to capture the non-linear relationships between input features and the classes. Kernel methods provide a solution to this problem by mapping the input features to a higher dimensional space where a linear classifier can separate the data effectively. In this method, we use a kernel function which computes the similarity in the higher dimensional space without explicitly computing the feature vectors (less computational cost). Here's an example: (Assuming x,y are a 2D vectors)

$$\begin{aligned} K(x, y) &= (x^T y)^2 \\ &= (x_1 y_1 + x_2 y_2)^2 \\ &= x_1^2 y_1^2 + 2 x_1 x_2 y_1 y_2 + x_2^2 y_2^2 \\ &= \langle (x_1^2, \sqrt{2} x_1 x_2, x_2^2), (y_1^2, \sqrt{2} y_1 y_2, y_2^2) \rangle \end{aligned}$$

Here (x_1, x_2) in 2D space is mapped to $(x_1^2, \sqrt{2}x_1x_2, x_2^2)$ in 3D. Note : For a given K(x,y) the mapping to higher dimension is not unique. e.g. we can also find a mapping from $\mathbb{R}^2 \longrightarrow \mathbb{R}^4$ with the same function $(x^T y)^2$ i.e. $(x_1, x_2) \longrightarrow (x_1^2, x_2^2, x_1x_2, x_1x_2)$

2 Kernel functions

A function x x $x \longrightarrow \mathbb{R}$ is a kernel iff for any $x_1, x_2, ...x_m \in \mathbb{R}$ and any m $\in \mathbb{N}$

- K(x_i, x_j) = K(x_j, x_i) i.e. the kernel function must be symmetric
- the Gram matrix $\mathbb{G}$ given by $G_{ij} = K(x_i, x_j)$ where $x_1 ... x_n$ are the data points must be positive semi definite.

Any real symmetric matrix $\mathbb{G}$ can be written as UDU^T where D is a diagonal matrix with positive entries (as the eigen values are positive). Thus D can be written as $\sqrt{D}\sqrt{D}$ and $\mathbb{G}$ = U$\sqrt{D}\sqrt{D}U^T$ = (U$\sqrt{D}$)(U$\sqrt{D}$)^T which means that there always exists a mapping ϕ = U$\sqrt{D}$ to a higher dimension.

2.1 Examples

- K(x_i, x_j) = $x_i x_j$ (where x,y are real numbers)
  This is obviously symmetric.
  For positive semi definiteness if we assume a to be some column vector [$a_1 ... a_n$], then $a^T G a$ evaluates to:
  $$\sum a_i a_j K(x_i, x_j) = \sum_{i,j} a_i a_j x_i x_j = (\sum a_i x_i)^2 \geq 0$$
  The intermediate step is evident from the expansion.

  Similarly if x_i, x_j are d-dimensional vectors i.e. $x \in \mathbb{R}^d$ and the inner product is defined as K(x_i, x_j) = $\langle x_i x_j \rangle$, $\sum a_i a_j \langle x_i x_j \rangle = ||\sum a_i x_i||^2 \geq 0$. A similar argument can be given if we apply a transformation $x \longrightarrow \phi(x)$ and define the inner product as $\langle \phi(x_i)\phi(x_j) \rangle$.

- Polynomial Kernel : $(x^T y)^d$ (x,y $\in \mathbb{R}^n$)
  From this function the mapping can be obtained using multinomial expansion as shown:
  $$(x^T y)^d = (\sum_{i=1}^{n} x_i y_i)^d = \sum_{k_1,2,...k_n} \binom{d}{k_1 k_2 ... k_n} x_1^{k_1} x_2^{k_2} .. x_n^{k_n} y_1^{k_1} y_2^{k_2} .. y_n^{k_n}$$
  The above summation can be represented as an inner product $\langle \phi(x), \phi(y) \rangle$

If K_1, K_2 are two kernel functions such that:

$$K_1 = \phi_1^T \phi_1, K_2 = \phi_2^T \phi_2$$

then K = $K_1 + K_2$ is also a kernel function where the equivalent transformation to the higher dimension, ϕ is given by:

$$\begin{pmatrix} \phi_1 \\ \phi_2 \end{pmatrix}$$

3 Applications of Kernel methods

3.1 Support Vector Regression

Consider the loss function:

$$||w||^2 + \sum_{i=1}^{n} (y_i - w^T x_i)^2$$

Upon applying the mapping $x \longrightarrow \phi(x)$, computing $w^T \phi(x)$ is not always computationally feasible as ϕ can be of very large dimensions. In the previous lecture we saw that for any loss function of the form:

$$\ell((w^T x_i), y_i) + \lambda ||w||^2$$

the optimal ω ($\omega*$) can be written as $\sum_1^N \alpha_i x_i y_i$ for any α_i not necessarily the Lagrangian multiplier which was the case for SVM. Upon applying the transformation the loss function becomes:

$$||w||^2 + \sum_{i=1}^{n} (y_i - w^T \phi(x_i))^2$$

and the optimal value of w = $\sum_1^N \alpha_i \phi(x_i)$ (Assume y_i is included in α_i)
Substituting the optimal value of w and using the fact that K(x_i, x_j) = $\langle \phi(x_i), \phi(x_j) \rangle = \phi(x_i)^T \phi(x_j)$, the problem translates to finding optimal α_i such that the following loss function is minimised:

$$\sum \alpha_i \alpha_j K(x_i, x_j) + \sum_{i=1}^{N} (y_i - \sum_{j=1}^{N} \alpha_j K(x_i, x_j))^2$$

Thus after getting optimal α_i, the optimal value of w can also be obtained. Now for testing a new vector x_{test} instead of computing $w^T x_{test}$ we can simply compute $\sum \alpha_i K(x_i, x_{test})$. This method can be used for any loss function which includes x_i^T, x_j as it can be replaced with $K(x_i, x_j)$ and non-linear data can be fitted with linear classifier using kernel functions.

3.2 Principal Component Analysis

In this method of dimensionality reduction we project find the eigenvectors and thus the eigenvalues of the covariance matrix. Then, we sort the eigenvalues and for reduction to k-dimensions we pick the top k vectors each of which acts as a feature. Next, we project the entire dataset onto these k eigenvectors such that the variance is maximum i.e. minimum information is lost. However, in some cases, the data may not be well-separated in the original feature space, and linear methods like PCA may not be effective. The kernel trick in PCA allows us to perform a nonlinear transformation of the data into a higher-dimensional space, where the data may be more easily separated. We can then apply PCA to this transformed data to obtain a set of principal components that capture the most variation in the transformed data.

4 Inner Product Space

4.1 Definition

An inner product space is a vector space $\mathcal{V}$ over field $\mathbb{R}$ together with an inner product, that is a map $\langle \cdot, \cdot \rangle : \mathcal{V} \times \mathcal{V} \to \mathbb{R}$

$$\langle X, Y \rangle = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n.$$

that satisfies the following properties for all vectors $X, Y, Z \in \mathcal{V}$ and all scalars $a, b \in \mathbb{R}$

- $\langle X, X \rangle \geq 0$. $\langle X, X \rangle = 0$ iff $X = 0$.
- $\langle Y, X \rangle = \langle X, Y \rangle$ (symmetry)
- $\langle aX + bY, Z \rangle = a\langle X, Z \rangle + b\langle Y, Z \rangle$ (linearity)

4.2 Inner Product Space of functions

Consider a function $f : \mathcal{X} \to \mathcal{V}$ from input space $\mathcal{X}$ to vector space $\mathcal{V}$.

$$y = f(x) = \sum_{\alpha \in \mathrm{A}} \alpha \phi_\alpha(x)$$

where $f(x)$ is a non-linear function. where $\phi_\alpha : \mathcal{X} \to \mathcal{V}$ belongs to the class of feature maps. If the indexing set A is finite, we can write

$$y = f(x) = w^T \phi(x)$$

for appropriate w and ϕ. But if the indexing set A is infinite, we need to find another way to compute f.
Suppose there exists a Taylor Expansion of f. Let us represent y in terms of the kernel function

$$y = f(x) = \sum_{i=1}^{N} \frac{\alpha_i y_i K(x_i, x)}{N}$$

where $K(\cdot, \cdot)$ is the kernel(similarity function) and sum is over all training examples.
Consider the space of functions spanned by the set $\{K(x_i, \cdot) \mid i \in [n]\}$. Observe f lies in this vector space. We define an inner product on this vector space.

$$\langle K(x_i, \cdot),\, K(x_j, \cdot) \rangle = K(x_i, x_j)$$

Note that we are only defining the inner product on the spanning set and the definition on any vector which is a linear combination of the spanning set follows from linearity of inner product. Further, observe that symmetry of inner product follows from symmetry of the kernel function. In order to ensure that the induced norm is positive for non-zero vectors, we need further restrictions on the kernel. Consider $g \neq 0$, then $\langle g,\, g \rangle > 0$

$$\begin{aligned}
\langle g,\, g \rangle &= \langle \sum_i a_i K(x_i, \cdot),\, \sum_j a_j K(x_j, \cdot) \rangle \\
&= \sum_{(i,j)} a_i a_j K(x_i, x_j) \\
&= a^T K a > 0
\end{aligned}$$

where g belongs to the space of functions described above, K is a matrix defined as $K_{(i,j)} = K(x_i, x_j)$. In other words, K is positive definite.

Now,

$$\begin{aligned}
\langle f,\, K(x, \cdot) \rangle &= \langle \sum_{i=1}^{N} \frac{\alpha_i y_i K(x_i, \cdot)}{N},\, K(x, \cdot) \rangle \\
&= \sum_{i=1}^{N} \frac{\alpha_i y_i \langle K(x_i, \cdot),\, K(x, \cdot) \rangle}{N} \\
&= \sum_{i=1}^{N} \frac{\alpha_i y_i K(x_i, x)}{N} = f(x)
\end{aligned}$$

A key point to note is that we don't need to explicitly compute the feature map as long as we have the kernel. This solves the issue with infinite dimensional feature space.

CS419-Intro to ML

Lecture 15: Kernel Methods

15 March 2023

Lecturer: Abir De *Scribe: Group 5,6*

In the previous lecture, we have had an introduction on the basics of Kernel and its properties. Then we went on to see some populars Kernels in the tutorial session also. In this lecture we go on to build up on Kernels and solve SVM problems through Kernel Methods.

1 Introduction

Let us take a short recap of what we studied in the last lecture and then continue on to the todays content for a better flow.
Before we understand Kernels, we need to understand the inner product and its propertise in a precise manner.

1.1 Inner Product Space

An inner product space over reals is a vector space $\mathcal{V}$ and an inner product, which is a mapping

$$\langle \cdot, \cdot \rangle : \mathcal{V} \times \mathcal{V} \to \mathcal{R}$$

Following are the properties of inner product $\forall x, y, z \in \mathcal{V}$ and $a, b \in \mathcal{R}$:-

- Symmetry: $\langle x, y \rangle = \langle y, x \rangle$
- Linearity: $\langle ax + by, z \rangle = a\langle x, z \rangle + b\langle y, z \rangle$
- Positive-definiteness: $\langle x, x \rangle \geq 0$ and $\langle x, x \rangle = 0 \iff x = 0$

1.2 Kernel

The kernel by definition avoids the explicit mapping that is needed to get linear learning algorithms to learn a nonlinear function or decision boundary.

For all x and x' in the input space Φ certain functions $K(x, x')$ can be expressed as an inner product in another space Ψ. The function

$$K : \Phi \times \Phi \to \mathbb{R}$$

$\forall x, x' \in \mathcal{X}$,

$$K(x, x') = \langle \phi(x), \phi(x') \rangle$$

The Gram matrix is then defined as $G_{i,j} = K(x_i, x_j)$.
The two necessary condition for a kernel are:

- Symmetric: $K(x_i, x_j) = K(x_j, x_i)$
- Positive Semi Definite : $G_{i,j} >= 0$

From the above understanding we can confirm that for a set of N data points there exist N Kernel Mappings.

2 SVM Objective Function

For the SVM mechanism we had already learnt the following objective function formulation:

$$\min_{w} \; l(\{w^T \phi(x_i)\}_{i \in D}, \{y_i\}_{i \in D}) + \lambda R(||w||) \tag{1}$$

where $l : \mathbb{R}^{|D|} \to \mathbb{R}$ is an arbitrary function and $R : \mathbb{R}_+ \to \mathbb{R}$ is a monotonically non-decreasing Regularization function.

We found out the optimal ω as:

$$w* = \sum_{i=1}^{|D|} \alpha_i \phi(x_i)$$

Form of f is

$$\begin{aligned} f(x) &= w^{*T} \phi(x_i) \\ &= \sum_{i=1}^{|D|} \alpha_i \phi^T(x_i) \phi(x) \end{aligned}$$

Here $\phi^T(x_i)\phi(x)$ is like a similarity measure If $\phi(\cdot)$ is ∞-dimensional, we can write it as

$$f(x) = \sum_{i=1}^{|D|} \alpha_i \sum_{j=0}^{\infty} \phi(x_i)[j]\phi(x)[j]$$

Thus, if $\phi(\cdot)$ is ∞-dimensional, it is very high computational task to compute $w^{*T}\phi(x_i)$ as w as it has the same dimension as ϕ. So, we try to represent the objective function in functional form or through the kernel formulation so that we would not have to do such computations.

3 Kernel Formualation

Now as in the last topic, we got stuck in the case when ω and $\phi(x_i)$ are of infinite dimension, it gets impossible to find these vectors.
Hence here we try to formulate the objective function with the use of Kernels that we studied earlier.

Writing $w = \sum_{j=1}^{|D|} \alpha_j \phi(x_j)$,
we have that for all i

$$\langle w, \phi(x_i) \rangle = \langle \sum_{j=1}^{|D|} \alpha_j \phi(x_j), \phi(x_i \rangle = \sum_{j=1}^{|D|} \alpha_j \langle \phi(x_j), \phi(x_i) \rangle.$$

Similarly,

$$||w||^2 = \langle \sum_{j=1}^{|D|} \alpha_j \phi(x_j), \sum_{j=1}^{|D|} \alpha_j \phi(x_j) \rangle = \sum_{i,j=1}^{|D|} \alpha_i \alpha_j \langle \phi(x_i), \phi(x_j) \rangle.$$

Let $K(x, x') = \langle \phi(x), \phi(x') \rangle$ be a function that implements the kernel function with respect to the feature space. Hence, instead of solving Equation 1, we can solve the equivalent problem

$$\min_{\alpha \in \mathbb{R}^{|D|}} l(\{\sum_{j=1}^{|D|} \alpha_j K(x_j, x_i)\}_{i \in D}, \{y_i\}_{i \in D}) + \lambda R(\sqrt{\sum_{i,j=1}^{|D|} \alpha_i \alpha_j K(x_j, x_i)}) \tag{2}$$

4 Probability Gaussian Process

Now that we have gained enough knowledge on Kernels, its properties and also its formulation. Here we take our discussion further on another application of kernels in the context of Gaussian Processes and how to deal with smaller training sets to still give fair results.

Now we already know the objective function as

$$w^{\text{regression}} \rightarrow \min \left[\sum_{i \in D} (y_i - w^T x_i)^2 \right]$$

The solution to the above problem is:

$$w^* = (\sum_{i \in D} x_i x_i^T)^{-1} \cdot (\sum_{i \in D} x_i y_i)$$

The predictions are made using function $f : \mathbb{R}^d \rightarrow \mathbb{R}$, $f(x_i) = w^T \cdot x_i$

An different approach to this could be to design a distribution on the function we are trying to predict such that every point in the training data must have exactly the same output in the hypothesis as the training label. More precisely, we would like to design a non linear estimator f to model the training data with the additional restriction that $\forall x_i \in D \; f(x_i) = y_i$; for the other points

$x \notin D$, $f(x)$ is a random variable with an associated probability distribution, while having certain guarantees on accuracy on test set and assuming train and test set are from same distribution.

According to the above hypothesis, the function will look something as in Figure 1.

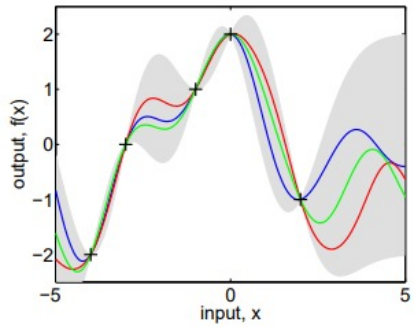


Figure 1: Graphical Representation

Here in this figure you can see that the points marked as + are the points in our dataset, for which the output is exactly one value while it is a distribution (as given by the shaded area) for all the other points (here assumed to be normal distribution).

Whenever we add an extra point to the dataset the mean line changes and passes through that point and the variance at that point becomes 0.

4.1 Gaussian Process

Gaussian processes are a method for non parametric estimation to provide confidence on the seen data and some kind of distribution on unseen data. For any subset of the training data, we must have that the joint prior distribution of this subset is normally distributed for some mean and covariance matrix.

For any subset $\{x_1 ... x_m\}$ of the training data, the prior distribution follows:

$$\begin{bmatrix} f(x_1) \\ \vdots \\ f(x_m) \end{bmatrix} \sim \mathcal{N}(\vec{\mu}(x_1, \ldots, x_m), \Sigma(x_1, \ldots, x_m))$$

where $\vec{\mu}$ and Σ are deterministic functions.

As discussed earlier, On introducing a new data point into any subset of the training data, we expect the resulting conditional distribution to also follow the normal distribution.

For the data point x^*

$$f(x^*)|(f(x_1),\dots f(x_m),x^*) \sim \mathcal{N}(\vec{\mu}(x_1,\dots,x_m,x^*),\Sigma(x_1,\dots,x_m,x^*))$$

4.2 Conditional Rule for Multi-variate Gaussian

Intuitively, if we start with a Gaussian distribution and update our knowledge given the observed value of one of its components(that is, find conditional probability distribution), then the resulting distribution is still Gaussian! Mathematically,
Let $[x\ y]$ jointly form multi variate Gaussian random variable,

$$\begin{bmatrix} x \\ y \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mu_x \\ \mu_y \end{bmatrix}, \begin{bmatrix} \Sigma_{xx} & \Sigma_{xy} \\ \Sigma_{yx} & \Sigma_{yy} \end{bmatrix}\right)$$

Here Σ_{ab} represents covariance matrix between random vectors a and b and $f(\cdot)$ represents the PDF.

$$f(x|y) = \frac{f(x,y)}{f(y)}$$

Now, we will substitute $f(x,y)$ with the expression for multi-variate Gaussian distribution($\mathcal{N}(\mu,\Sigma)$), and $f(y)$ with $\mathcal{N}(\mu_y,\Sigma_{yy})$. Simplifying the equations, we get

$$f(x|y) = \mathcal{N}(\Sigma_{xy}\Sigma_{yy}^{-1}y, \Sigma_{xx} - \Sigma_{xy}\Sigma_{yy}^{-1}\Sigma_{yx})$$

(μ is assumed to be zero for simplicity)

4.3 Updating Parameters on New data

Suppose we have a training set {y,x} and a test set and we try to fit a model through it. Under what conditions will the model go through all the points? Fitting a higher order model would ensure high accuracy on the training set, but test set accuracy must be satisfied.

Given P(y|x) $\Rightarrow$ As long as we observe y, it's variance should go to zero.

$$P(y|x) = \mathcal{N}(\mu,\Sigma)$$

but if (x,y) $\in \{(x_i,y_i)\}_{i=1}^N$ then $\sigma = 0$ There can be error on points we don't observe, but there mustn't be error on already observed points.

To achieve this $\Rightarrow \min_w \Sigma(y_i - w^T x_i)^2 \to 0$ where $x_i \to \phi(x_i)$ which dimension may tend to ∞.

$\phi(x_i)$ can't be finite as then it won't work for arbitrary number of training set. w and $\phi(x)$ is not computable but $f(x_i) = \Sigma\alpha_i k(x,x_i)y_i$.

$$\min_\alpha \Sigma(y_i - \alpha_i k(x,x_i)y_i$$

$$\min_\alpha \sum_j \left[\sum_i y_i - \alpha_i k(x,x_i)y_i)\right]^2$$

If we observe new point y_i, we have to adjust α_i such that to keep variance = 0.

CS 419

Lecture 16: Gaussian Processes

17 March 2023

Lecturer: Abir De *Scribe: Aditya, Jay, Soham & Sooraj*

Till now we had studied about kernels and their properties. In the last lecture, we learned about Gaussian Processes(GP).

1 Recap of the last lecture

Let's begin with a brief introduction to Gaussian processes.

Gaussian Processes are a class of probabilistic models that can be used for supervised machine-learning tasks such as regression and classification. GPs are a non-parametric approach, meaning they do not assume any particular functional form for the relationship between the inputs and outputs.

In a GP, a function is modeled as a probability distribution over functions. This distribution is defined by a mean function and a covariance function. The mean function specifies the expected value of the function at each input point, while the covariance function specifies how much the function values at different input points are correlated.

During training, the GP is fitted to the training data by adjusting the hyper-parameters to maximize the likelihood of the observed data. Once the GP is trained, it can be used to make predictions for new inputs by computing the posterior distribution over functions given the observed data.

2 Gaussian Processes and Multivariate Gaussian Distribution

A multivariate Gaussian distribution is a probability distribution that describes the joint probability distribution of a set of random variables that are normally distributed. It is an extrapolation of the univariate Gaussian distribution to higher dimensions, where the mean vector and covariance matrix fully specify the distribution.

The multivariate Gaussian distribution is widely used in statistics, machine learning, and many other fields, due to its flexibility and tractability. It is used in applications such as clustering, classification, regression and data analysis among others.

2.1 2-dimensional Gaussian Distribution

$$y \sim \mathcal{N}(\mu, \Sigma) \tag{1}$$

where y and μ are two-dimensional vectors and Σ is 2x2 matrix given by:

$$\Sigma = \begin{bmatrix} \Sigma_{AA} & \Sigma_{AB} \\ \Sigma_{BA} & \Sigma_{BB} \end{bmatrix} \tag{2}$$

where Σ_{aa} represents the variance of a and Σ_{ab} is a co-variance of a and b.

$$f(x) = \frac{1}{\sqrt{2\pi|\Sigma|^2}} e^{-\frac{(x-\mu)^T \Sigma^{-1}(x-\mu)}{2}} \tag{3}$$

2.2 Conditional Gaussian Distribution

We have two random variables X and Y, and we know that Y is dependent on X, then the conditional distribution of Y given X can be represented as a Gaussian distribution.

$$Y_A|Y_B \sim \mathcal{N}(\mu'_A, \Sigma_{A|B}) \tag{4}$$

Now, let's see the effect of conditioning on the mean of a distribution:
If we observe one variable,
let's say Y_B = Y_0, then Y_B is a distribution with mean Y_0 and variance 0.
Start with the intuition:

$$\mu'_A = \mu_A + \Sigma_{AB}\Sigma_{BB}^{-1}(Y_0 - \mu_B) \tag{5}$$

Verification of our intuition:

- If $Y_B = \mu_B$, then the mean of Y_A won't change which can be seen in the equation. $Y_0 = \mu_B$ then $\mu'_A = \mu_A + 0$.
- Substituting A with B yields, $\mu'_B = \mu_B + \Sigma_{BB}\Sigma_{BB}^{-1}(Y_0 - \mu_B) = Y_0$
- If co-variance of Y_A and Y_B is zero, then after observation mean of Y_A doesn't change. $\Sigma_{AB} = 0$, so $\mu'_A = \mu_A$
- If Σ_{BB} is very large, it means that there is a large variation in Y_B. Hence, we shouldn't rely on the observed value of Y_B and thus, the mean remains unaffected.
- Now $\Sigma_{BB} \to 0$ which means that Y_B is a constant then $(Y_B - \mu_B) \to 0$

So, we have to calculate the limit for μ'_A,

$$\lim_{\Sigma_{BB}\to 0}(\mu'_A) = \mu_A + \lim_{\Sigma_{BB}\to 0}[\Sigma_{AB}\Sigma_{BB}^{-1}(Y_0 - \mu_B)] \tag{6}$$

Calculating the limit:

$$\mathbf{L} = \lim_{\Sigma_{BB} \to 0} [\Sigma_{AB}\Sigma_{BB}^{-1}(Y_0 - \mu_B)] \tag{7}$$

where, Σ_{AB} = $\mathbf{E}[(Y_A - \mu_A)(Y_B - \mu_B)]$, Σ_{BB} = $\mathbf{E}[(Y_B - \mu_B)^2]$.

Since variance is tending towards zero, $(Y_B - \mu_B)$ can be assumed as constant.

$$\mathbf{L} = \lim_{\Sigma_{BB} \to 0} \frac{(Y_B - \mu_B)^2 \mathbf{E}[Y_A - \mu_A]}{(Y_B - \mu_B)^2} \tag{8}$$

$$= \lim_{\Sigma_{BB} \to 0} \mathbf{E}[Y_A - \mu_A] = 0 \tag{9}$$

so,

$$\lim_{\Sigma_{BB} \to 0} (\mu'_A) = \mu_A + 0 = \mu_A \tag{10}$$

Now that we have observed the effect of conditioning on the mean of a distribution, we proceed with the variance of $Y_A|Y_B$. Intuitively, one can guess the variance to be the following:

$$\Sigma_{A|B} = \Sigma_{AA} - \Sigma_{AB}\Sigma_{BB}^{-1}\Sigma_{BA} \tag{11}$$

Verification of our intuition:

- If $A = B$, then the value of variance will be:

$$\Sigma_{B|B} = \Sigma_{BB} - \Sigma_{BB}\Sigma_{BB}^{-1}\Sigma_{BB} \tag{12}$$

$$= \Sigma_{BB} - \Sigma_{BB} I = 0 \tag{13}$$

  Hence, we obtain $\Sigma_{B|B} = 0$.

- When $\Sigma_{BB} \to \infty$:

$$\Sigma_{A|B} = \lim_{\Sigma_{BB} \to \infty} (\Sigma_{AA} - \Sigma_{AB}\Sigma_{BB}^{-1}\Sigma_{BA}) \tag{14}$$

Now, we need to calculate the limit for $\Sigma_{A|B}$,
Calculating the limit:

$$\mathbf{L} = \lim_{\Sigma_{BB} \to \infty} (\Sigma_{AA} - \Sigma_{AB}\Sigma_{BB}^{-1}\Sigma_{BA}) \tag{15}$$

where, Σ_{AB} = $\mathbf{E}[(Y_A - \mu_A)(Y_B - \mu_B)]$= Σ_{BA}, Σ_{BB} = $\mathbf{E}[(Y_B - \mu_B)^2]$.

$$\mathbf{L} = \lim_{(Y_B - \mu_B)^2 \to \infty} \left(\mathbf{E}[(Y_A - \mu_A)^2] - \frac{(\mathbf{E}[(Y_A - \mu_A)(Y_B - \mu_B)])^2}{\mathbf{E}[(Y_B - \mu_B)^2]} \right) \tag{16}$$

As $\mathbf{E}[(Y_B - \mu_B)^2]$ becomes arbitrarily large, we can conclude that:

$$\mathbf{L} = \lim_{(Y_B - \mu_B)^2 \to \infty} \left(\mathbf{E}[(Y_A - \mu_A)^2] - \frac{(\mathbf{E}[(Y_A - \mu_A)(Y_B - \mu_B)])^2}{\mathbf{E}[(Y_B - \mu_B)^2]} \right) = \mathbf{E}[(Y_A - \mu_A)^2] = \Sigma_{AA} \tag{17}$$

One more argument that can be made in the favour of the above result is that since Σ_{BB} is arbitrarily large Y_B is effectively a noise term and the distribution of $Y_A|Y_B$ is identical to that of Y_A. Thus, we have modeled the bi-variate Gaussian distribution in the following manner:

$$Y_A|Y_B \sim \mathcal{N}\left(\mu_A + \Sigma_{AB}\Sigma_{BB}^{-1}(Y_0 - \mu_B), \Sigma_{AA} - \Sigma_{AB}\Sigma_{BB}^{-1}\Sigma_{BA} \right) \tag{18}$$

Following is a plot of the joint, marginal and conditional distribution for a bivariate gaussian function which has been discussed above.

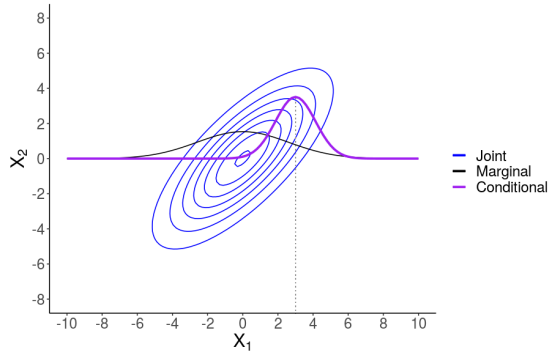


Figure 1: Joint, Marginal and Conditional Distribution for bivariate Gaussian distribution
source: https://fabiandablander.com/statistics/Two-Properties.html

We have used an extremely intuitive approach to modeling the bivariate Gaussian distribution, a more complete and rigorous derivation of the same can be found at `https://fabiandablander.com/statistics/Two-Properties.html`.

CS419M

Lecture 17: Interpolation and Regression revisited

24th March, 2023

Lecturer: Abir De *Scribe: Harshit, Samyak, Satush*

1 Introduction

This is a brief recap of the results of the previous lecture on interpolation.

Conditional Gaussian distribution: Consider the joint distribution between vectors x_1 and x_2:

$$\begin{bmatrix} y_A \\ y_B \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mu_A \\ \mu_B \end{bmatrix}, \begin{bmatrix} \boldsymbol{\Sigma}_{AA} & \boldsymbol{\Sigma}_{AB} \\ \boldsymbol{\Sigma}_{BA} & \boldsymbol{\Sigma}_{BB} \end{bmatrix} \right)$$

We are interested in the conditional distribution, which itself is Gaussian:

$$y_A | y_B \sim \mathcal{N}\left(\boldsymbol{\mu}_{A|B}, \boldsymbol{\Sigma}_{A|B}\right)$$

where

$$\boldsymbol{\mu}_{A|B} = \boldsymbol{\mu}_A + \boldsymbol{\Sigma}_{AB}\boldsymbol{\Sigma}_{BB}^{-1}(y_B - \boldsymbol{\mu}_B)$$
$$\boldsymbol{\Sigma}_{A|B} = \boldsymbol{\Sigma}_{AA} - \boldsymbol{\Sigma}_{AB}\boldsymbol{\Sigma}_{BB}^{-1}\boldsymbol{\Sigma}_{BA}$$

Product of Gaussian distributions: Consider the two distributions:

$$p_1(x) = \mathcal{N}(x; \mu_1, \boldsymbol{\Sigma}_1), \quad p_2(x) = \mathcal{N}(x; \mu_2, \boldsymbol{\Sigma}_2)$$

The product is an un-normalised Gaussian:

$$p_1(x)p_2(x) \propto \mathcal{N}(x; \mu, \boldsymbol{\Sigma})$$

where

$$\boldsymbol{\mu} = \boldsymbol{\Sigma}(\boldsymbol{\Sigma}_1^{-1}\boldsymbol{\mu}_1 + \boldsymbol{\Sigma}_2^{-1}\boldsymbol{\mu}_2)$$
$$\boldsymbol{\Sigma} = (\boldsymbol{\Sigma}_1^{-1} + \boldsymbol{\Sigma}_2^{-1})^{-1}$$

2 Linear Regression

Consider the supervised training data of n samples, each with an observation $\mathbf{x_i}$ and output y_i. The regression function $f(\mathbf{x})$ is linear if defined as

$$f(\mathbf{x}) = \mathbf{w}^T\phi(\mathbf{x})$$

and the target value has Gaussian noise so that

$$y(\mathbf{x}) = f(\mathbf{x}) + \epsilon$$

where

$$\epsilon \sim \mathcal{N}(0, \sigma^2)$$

For a given value of w, the likelihood of the outputs can be expressed as

$$p(y_1, \ldots, y_n | x_1, \ldots, x_n, \mathbf{w}) = \prod_{i=1}^{n} p(y_i | x_i, w) = \mathcal{N}(\mathbf{y}; \boldsymbol{\Phi}^T\mathbf{w}, \sigma^2 I)$$

where I is the $n \times n$ identity matrix and

$$\boldsymbol{\Phi} = \begin{bmatrix} \phi(\mathbf{x_1}) & \cdots & \phi(\mathbf{x_n}) \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}$$

The value of optimum $\mathbf{w}^*$ can then be found by maximizing this likelihood. This is equivalent to minimizing the least squares cost function

$$\min_{\mathbf{w}} \sum_{i=1}^{n} (y_i - \mathbf{w}^T\phi(\mathbf{x_i}))^2 = \min_{\mathbf{w}} \|\mathbf{y} - \boldsymbol{\Phi}^T\mathbf{w}\|^2$$

which gives us the following result (derived in a previous lecture):

$$\mathbf{w}^* = (\boldsymbol{\Phi}\boldsymbol{\Phi}^T)^{-1}\boldsymbol{\Phi}\mathbf{y}$$

3 Weight Vector Prior

Now consider a prior distribution over $\mathbf{w}$ given by the Gaussian:

$$\mathbf{w} \sim \mathcal{N}(0, \boldsymbol{\Sigma}_p)$$

The result now becomes:

$$\mathbf{w}^* = \left(\frac{\boldsymbol{\Phi}\boldsymbol{\Phi}^T}{\sigma^2} + \boldsymbol{\Sigma}_p^{-1} \right)^{-1} \frac{\boldsymbol{\Phi}\mathbf{y}}{\sigma^2}$$

We can confirm this intuitively by the following 2 checks:

- For the σ^2 outside the bracket: if σ^2 is large, that means y has a lot of noise so we should discard the data point. Clearly the weights become very small for such a data point.
- For the σ^2 inside the bracket: if we are discarding the data points as in the above point, the weights shouldn't depend on x, thus it is also divided by σ^2.

4 Interpolation and Regression

From the previous setting, lets say we have observed the points $\mathcal{D} = (\mathbf{x_i}, y_i)_{i=1}^N$.

Now let us try to find out the distribution of $y^*|\{\mathbf{x}^*, (\mathbf{x_i}, y_i)_{i=1}^N\}$, or $y^*|\{\mathbf{x}^*, \mathcal{D}\}$, where $(\mathbf{x}^*, y^*)$ is a new unobserved point.

FACT: The distribution will be a Gaussian, hence we just need to find the mean and variance. Since the distribution is a Gaussian, the mean and the mode will be the same. Now we know that the mode is the following (by maximum likelihood estimate done in the previous sections):

$$\hat{y}^* = \mathbf{w}^{*T}\phi(\mathbf{x}^*) = \phi(\mathbf{x}^*)^T\mathbf{w}^*$$

where $\mathbf{w}^*$ is as shown in Section 3. Hence:

$$\hat{y}^* = \phi(\mathbf{x}^*)^T \left(\frac{\mathbf{\Phi}\mathbf{\Phi}^T}{\sigma^2} + \mathbf{\Sigma}_p^{-1}\right)^{-1} \frac{\mathbf{\Phi}\mathbf{y}}{\sigma^2}$$

Thus we have found the mean of the distribution:

$$y^*|\{\mathbf{x}^*, \mathcal{D}\} \sim \mathcal{N}(\hat{y}^*, \mathbf{\Sigma} = ?)$$

Is this similar to the formula we derived in the last lecture? Lets see!

$$\boldsymbol{\mu}_{A|B} = \boldsymbol{\mu}_A + \mathbf{\Sigma}_{AB}\mathbf{\Sigma}_{BB}^{-1}(y_B - \boldsymbol{\mu}_B)$$

Here, A is equivalent to $\mathbf{x}^*$ and B is equivalent to the y_i's in $\mathcal{D}$. Now, μ_A and μ_B are both equal to 0 as we do not have any observation and both y_A and y_B are sampled from distributions with mean 0. Thus,

$$\boldsymbol{\mu}_{A|B} = \mathbf{\Sigma}_{AB}\mathbf{\Sigma}_{BB}^{-1}y_B$$

$$\overline{\mathbf{y}} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \mathbf{\Phi}^T\mathbf{w} + \epsilon$$

Hence,

$$\begin{aligned}\mathbb{E}[\mathbf{y}\mathbf{y}^T] &= \mathbf{\Phi}^T\mathbb{E}[\mathbf{w}\mathbf{w}^T]\mathbf{\Phi} + \mathbb{E}[\epsilon\epsilon^T] \\ &= \mathbf{\Phi}^T\mathbf{\Sigma}_p\mathbf{\Phi} + \sigma^2 I\end{aligned}.$$

Here, $\mathbb{E}[\mathbf{y}\mathbf{y}^T]$ denotes the covariance matrix and row x^* and column $1, 2, \ldots N$ of the covariance will be $\mathbf{\Sigma}_{AB}$. Therefore,

$$\begin{aligned}y_{A|B} &= \mathbf{\Sigma}_{AB}\mathbf{\Sigma}_{BB}\mathbf{y}_B \\ &= [\phi(x^*)^T\mathbf{\Sigma}_p\mathbf{\Phi}][\mathbf{\Phi}^T\mathbf{\Sigma}_p\mathbf{\Phi} + \sigma^2 I]^{-1}\overline{\mathbf{y}}\end{aligned}$$

We define $\mathbf{K}$ as $\mathbf{\Phi}^T\mathbf{\Sigma}_p\mathbf{\Phi}$

It is important to note that we simply can't take the inverse of $\mathbf{\Phi}$ as $\mathbf{\Phi}$ is not a square matrix.

Theorem 4.1. *Let* $A = \frac{\mathbf{\Phi}\mathbf{\Phi}^T}{\sigma^2} + \mathbf{\Sigma}_p^{-1}$*, then*

$$A\mathbf{\Sigma}_p\mathbf{\Phi} = \frac{1}{\sigma^2}\mathbf{\Phi}\left(\boldsymbol{K} + \sigma^2 I\right)$$

Proof.

$$\begin{aligned}\frac{1}{\sigma^2}\mathbf{\Phi}\left(\mathbf{K} + \sigma^2 I\right) &= \frac{1}{\sigma^2}\mathbf{\Phi}\left(\mathbf{\Phi}^T\mathbf{\Sigma}_p\mathbf{\Phi} + \sigma^2 I\right) \\ &= \frac{\mathbf{\Phi}\mathbf{\Phi}^T\mathbf{\Sigma}_p\mathbf{\Phi}}{\sigma^2} + \mathbf{\Sigma}_p^{-1}\mathbf{\Sigma}_p\mathbf{\Phi} \\ &= \left(\frac{\mathbf{\Phi}\mathbf{\Phi}^T}{\sigma^2} + \mathbf{\Sigma}_p^{-1}\right)\mathbf{\Sigma}_p\mathbf{\Phi}\end{aligned}$$

□

Therefore, we have

$$\begin{aligned}&\frac{1}{\sigma^2}\mathbf{\Phi}\left(\mathbf{K} + \sigma^2 I\right) = A\mathbf{\Sigma}_p\mathbf{\Phi} \\ &\implies \frac{1}{\sigma^2}A^{-1}\mathbf{\Phi} = \mathbf{\Sigma}_p\mathbf{\Phi}\left(\mathbf{K} + \sigma^2 I\right)^{-1}.\end{aligned}$$

Therefore,

$$\begin{aligned}\frac{1}{\sigma^2}\phi(x^*)^T A^{-1}\mathbf{\Phi}\overline{\mathbf{y}} &= \phi(x^*)^T\mathbf{\Sigma}_p\mathbf{\Phi}\left(\mathbf{K} + \sigma^2 I\right)^{-1}\overline{\mathbf{y}} \\ &= y_{A|B}\end{aligned}$$

This proves that in linear regression also we are interpolating. But how is this possible? The answer lies in a small assumption we have made.

The matrix A (from regression formula) is always invertible. If $\sigma^2 = 0$ then the formula has $\mathbf{K}^{-1}$ (for the interpolation- $y_{A|B}$ case). Dimension of $\mathbf{\Phi}$ is $d * N$, where d is the dimension of the feature set ϕ. Thus, K is a reduced rank matrix for $d < N$ and isn't invertible. Hence the analogy of both being the same fails. So, for the analogy to work, d should be greater than **any** N, that means it should be infinite.

This means we have assumed that K is invertible, or if $\sigma^2 = 0$, ϕ is of infinite dimension!

We will prove that the Variance is also the same in both cases in the next lecture.

CS419M

Lecture 18: Mean and Variance

02/04/23

Lecturer: Abir De *Scribe: Groups 11 & 12*

This is some warmup discussion before the first section.

1 Recap

Consider the following equation:

$$y = w^\top \phi(x) + \epsilon$$

This is our standard regression model, with $\phi(x)$ being the $d \times 1$ feature vector.

$\epsilon \sim \mathcal{N}(0, \sigma^2)$

is the noise in the model, modelled as a Gaussian with 0 mean and variance σ^2.

$w \sim \mathcal{N}(0, \sum_p)$

is the $d \times 1$ weight vector, drawn from a Gaussian distribution.

A Gaussian process is a collection of random variables which have a joint Gaussian distribution.

Given N observations $(x_1, y_1), (x_2, y_2), \cdots, (x_N, y_N)$, for a new observation $(x*, y*)$, we have:

$$y*/x*, D \sim \mathcal{N}(\mu, \Sigma)$$

$$D = \{(x_1, y_1), (x_2, y_2), ..., (x_N, y_N)\}$$

We want to find out μ and Σ in the above equation.

We have seen that

$$\mathbf{E}(y*/x*, D) = \Phi(x*)^\top \Sigma_p \Phi(\Phi^\top \Sigma_p \Phi + \sigma^2 \boldsymbol{I})^{-1} y \tag{1}$$

Here $\Phi^\top \Sigma_p \Phi$ may be invertible only if $d \to \infty$.

2 Analysing the mean further

Suppose $x* \in D$. Without loss of generality, let $x* = x_1$. If $\epsilon = 0$ (which implies $\sigma = 0$), we expect $y*$ to be exactly equal to y_1. If the noise was present even for an $x_i \in D$, the measured y can be different from y_i. Let us try to verify this.

Putting $x* = x_1$ and $\sigma = 0$ in (1), we have:

$$\mathbf{E}(y_1/x_1, D, \sigma = 0) = \Phi(x_1)^\top \Sigma_p \Phi(\Phi^\top \Sigma_p \Phi)^{-1} y \tag{2}$$

Now $\Phi(x_1)^\top \Sigma_p \Phi$ is the first row of $\Phi^\top \Sigma_p \Phi$.

If B is an invertible matrix and $B_{1,.}$ is its first row, then

$$(AB)_{1,.} = A_{1,.}B$$

We can write:

$$BB^{-1} = \boldsymbol{I} \implies B_{1,.}B = \boldsymbol{I}_{1,.} = [1, 0, ..., 0]_{1 \times n}$$

So if we take the matrix $\Phi^\top \Sigma_p \Phi$ as B above we obtain the same row vector as above.*(Note that we have assumed $\Phi^\top \Sigma_p \Phi$ to be invertible, which may not always be the case)*. Finally, multiplying with y which is a $n \times 1$ column vector, we obtain y_1 on the RHS of (2).

Now lets investigate what happens if $\sigma \neq 0$.

Again, taking $B = \Phi^\top \Sigma_p \Phi$ and B_1 as its first row, we have the RHS of (2) as

$$B_1(B + \sigma^2 I)^{-1} y = B_1(B + \sigma^2 BB^{-1})y = B_1(\boldsymbol{I} + \sigma^2 B^{-1})^{-1} B^{-1} y = y_1 - \sigma^2 B_1 1 B^{-2} y$$

Here $(\boldsymbol{I} + \sigma^2 B^{-1})^{-1}$ was expanded as $\boldsymbol{I} - \sigma^2 B^{-1}$ using Taylor's theorem, under the assumption that σ is small enough for the expension to be valid.

3 Variance

$$y = w^T.\phi(x) + \epsilon \sim \mathcal{N}(0, \sigma^2)$$

$$w \sim \mathcal{N}(0, \epsilon_P$$

What would the value of $var(y|D)$ be? Where D = $(x_i, y_i)_{i=1}^N$ P($y^*|x^*$, D)

$$var(y|D) = var(w^T \phi(x)) + \sigma^2$$

$$= \mathbb{E}(\phi(x^*)^T (w - \overline{w})(w - \overline{w})^T \phi(x^*)|D) + \sigma^2$$

Here w is kind of stochastic.

$$= \phi(x^*)^T \mathbb{E}((w - \overline{w})(w - \overline{w})^T |D) \phi(x^*) + \sigma^2$$

For now, we rather focus on P(w|D). The problem of finding the variance of y^* reduces to finding the covariance matrix of w.

If we know w, we can easily find the distribution of D.

$$P(w|D) = \frac{P(D|w).P(w)}{P(D)}$$

$$\implies P(w|D) \propto P(D|w).P(w)$$

Now,

$$P(D|w).P(w) = exp[-\frac{(\overrightarrow{y} - \phi^T w)^T (\overrightarrow{y} - \phi^T w)}{2\sigma^2}].exp[-w^T \epsilon_p^{-1} w/2]$$

$$z^{-1} = \phi\phi^T/\sigma^2 + \epsilon_P^{-1}$$

Confirming $\overline{w}$ is the same that we found earlier.

$$\overline{w} = \frac{z\sum_{i=1}^{N}\phi(x_i)(y_i)}{\sigma^2} = \frac{Z\phi.y}{\sigma^2}$$

$$\mathcal{E}(x^*, y^*|D) = \phi(x^*)^T[\phi\phi^T/\sigma^2 + \epsilon_P^{-1}]\frac{\phi.y}{\sigma^2}$$

Lecture 19:

31/03/2023

Lecturer: Abir De *Scribe: Group 21*

1 Recap

Consider the following process :

$$y = w^T\phi(x) + \epsilon \tag{1}$$

where $\epsilon \sim \mathcal{N}(0, \sigma^2)$, $\boldsymbol{w} \sim \mathcal{N}(0, \Sigma_p)$ is a d X 1 weight vector, and $\phi(x)$ is the d x 1 feature vector, then the probability of observing y^* given x^* and D, where D = $\{(x_i, y_i)|i = 1, 2, ..., n\}$ is given by:

$$P(y^*|x^*, D) \sim \mathcal{N}(\mu_{y^*}, \sigma^2_{y^*})$$

$$\mu_{y^*} = \phi(x^*)^T\left[\frac{\Phi\Phi^T}{\sigma^2} + \Sigma_p^{-1}\right]^{-1}\frac{\Phi y}{\sigma^2} \tag{2}$$

$$\mu_{y^*} = \phi(x^*)^T\Sigma_p\Phi(\Phi\Sigma_p\Phi + \sigma^2 I)^{-1}y \tag{3}$$

Note that these two expressions of μ_{y^*} will be equal only when both the inverses exist

$$\sigma^2_{y^*} = \phi(x^*)^T\left[\Sigma_p - \Sigma_p\Phi(\Phi^T\Sigma_p\Phi + \sigma^2 I)^{-1}\Phi^T\Sigma_p\right]\phi(x^*) + \sigma^2 \tag{4}$$

Recall ,

$$\mu_{A|B} = \mu_A + \Sigma_{AB}\Sigma_{BB}^{-1}(y_B - \mu_B)$$

Hence here $\mu_A = \mu_B = 0$, $\Sigma_{AB} = \phi(x^*)^T\Sigma_p\Phi$, $\Sigma_{BB} = \Phi^T\Sigma_p\Phi$

For σ = 0 we have shown that,

$$\mu_{y^*} = y_i \quad if \ \ (x^*, y^*) = (x_i, y_i) \tag{5}$$

2 Analyzing the Variance

Similar to calculation of μ_{y^*}, for $\sigma = 0\ and\ (x^*, y^*) = (x_i, y_i)$,

$$\sigma^2_{y^*} = \phi(x^*)^T\Sigma_p\phi(x^*) - \phi(x^*)^T\Sigma_p\Phi(\Phi^T\Sigma_p\Phi + \sigma^2 I)^{-1}\Phi^T\Sigma_p\phi(x^*)$$

As shown in previous lectures , $\phi(x^*)^T\Sigma_p\Phi(\Phi^T\Sigma_p\Phi + \sigma^2 I)^{-1}$ = [1 0 0..0] , when $(x^*, y^*) \in D$

$$\implies \sigma^2_{y^*} = \phi(x^*)^T\Sigma_p\phi(x^*) - [1\ 0\ 0..0]\Phi^T\Sigma_p\phi(x^*)$$

$$\implies \sigma^2_{y^*} = \phi(x^*)^T\Sigma_p\phi(x^*) - \phi(x^*)^T\Sigma_p\phi(x^*)$$

$$\implies \sigma^2_{y^*} = 0$$

Similarly we can also show that if (x^*, y^*) are such that,

$$\phi(x^*) = \Sigma\alpha_i\phi(x_i) \;\; , \;\; y^* = \Sigma\alpha_i y_i \;\; , \;\; then \;\; \sigma_{y^*} = 0 \tag{6}$$

3 Finding Variance of Gaussian process when the data belongs to the training data itself

We have to find $\sigma^2_{y^*} \; of \; P(y^*|x^*, D)$ for $x^* = x_i$
We know that,

$$\sigma^2_{y^*} = \phi(x^*)^T \left[\Sigma_p - \Sigma_p\Phi(\Phi^T\Sigma_p\Phi + \sigma^2 I)\Phi^T\Sigma_p\right]\phi(x^*) + \sigma_2$$

$$\Sigma_p - \Sigma_p\Phi(\Phi^T\Sigma_p\Phi + \sigma^2 I) = \Sigma_p - \Sigma_p\phi(I + \sigma^2(\Phi^T\Sigma_p\Phi)^{-1})^{-1}(\Phi^T\Sigma_p\Phi)^{-1}\Phi^T\Sigma_p$$

$$= \Sigma_p - \Sigma_p\phi(I - \sigma^2(\Phi^T\Sigma_p\Phi)^{-1})(\Phi^T\Sigma_p\Phi)^{-1}\Phi^T\Sigma_p$$

$$= \Sigma_p - \Sigma_p\Phi(\Phi^T\Sigma_p\Phi)^{-1}\Phi^T\Sigma_p + \sigma^2\Sigma_p\Phi(\Phi^T\Sigma_p\Phi)^{-2}\Phi^T\Sigma_p$$

Hence,

$$\sigma^2_{y^*} = \Phi(x^*)^T\Sigma_p\Phi(x^*) - \Phi(x^*)^T\Sigma_p\Phi(\Phi^T\Sigma_p\Phi)^{-1}\Phi^T\Sigma_p\Phi(x^*) + \sigma^2\Phi(x^*)^T\Sigma_p\Phi(\Phi^T\Sigma_p\Phi)^{-2}\Phi^T\Sigma_p\Phi(x^*) + \sigma^2$$

$$\Phi(x^*)^T\Sigma_p\Phi(x^*) - \Phi(x^*)^T\Sigma_p\Phi(\Phi^T\Sigma_p\Phi)^{-1}\Phi^T\Sigma_p\Phi(x^*) = 0 \qquad for \; x^* = x_i$$

Hence,
$\sigma^2_{y^*} = \sigma^2\Phi(x^*)^T\Sigma_p\Phi(\Phi^T\Sigma_p\Phi)^{-2}\Phi^T\Sigma_p\Phi(x^*) + \sigma^2$
= $2\sigma^2$

4 Writing the above variance in terms of kernel function

Let

$$\phi(x^*)^T\Sigma_p\Phi = K(x^*, x)$$
$$\Phi^T\Sigma_p\Phi = K(X, X) = K$$

Hence,

$$\sigma^2_{y^*} = K(x^*, x + \sigma^2 - K(x^*, x)[K + \sigma^2 I]^{-1}K(x, x^*)y$$

5 Some points to think about

If $\sigma \neq 0$ then for what instance the variance σ_{y^*} will be least?

Lets say we have $x_1, x_2, ..., x_{1million}$ unlabelled points. $x'_1, x'_2, ..., x'_{1000}$ are labelled as $y'_1, y'_2, ..., y'_{1000}$. Using these we want to pick some $x_{i's}$ for labelling , but which ones to pick, because picking all of them is not practically possible?

We will want those $x_{i's}$ which are dissimilar with the given $x_{j's}$. Hence they will have more variance. Therefore we find low variance $x_{i's}$ and just label them using nearest neighbour and discard them from getting picked up to label the hard way.

CS419M

Lecture 20: Introduction to Deep Learning

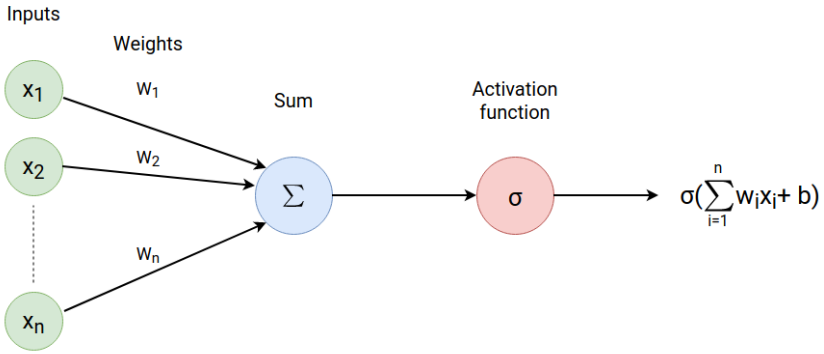
5 April 2023

Lecturer: Abir De *Scribe: Group 22*

1 Introduction

1.1 Capacity of the perceptron

What kind of function can a neural network represent ?
Let's start with the simplest neural network, the Perceptron. It is effectively a NN with a single hidden layer having 1 hidden unit with an activation function σ. Both the input layer and the weights are a $1 \times n$ vector.



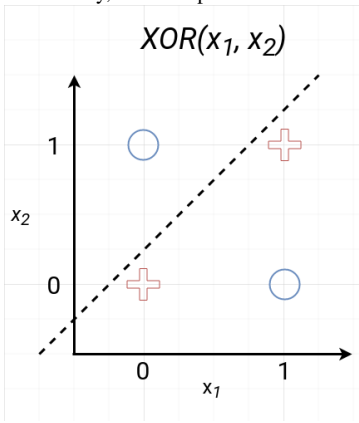
The perceptron can model a function of this form: $\sigma(\sum_i^n \mathbf{w}_i x_i + b) = \sigma(\mathbf{w}^\top \mathbf{x} + b)$. Often, we select the non-linear function σ to be one of the following:

$$\text{sigmoid} = \frac{1}{1+e^{-x}}$$

$$\tanh = \frac{e^{2x}-1}{e^{2x}+1}$$

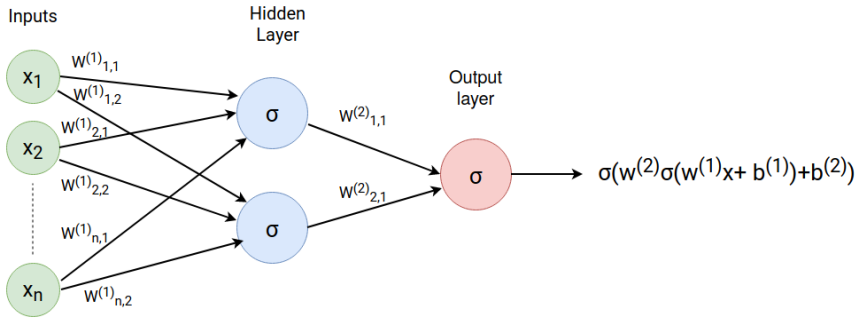
$$\text{ReLU} = \max(0, x)$$

Notably, the Perceptron is a linear classifier, and as such it famously can't model an XOR.



1.2 Capacity of multiple neurons

By allowing ourselves more than 1 neuron in the hidden layer, we can model a XOR and in fact, we get the simplest **universal approximator**.



2 Universal Approximation theory

The universal approximation theorem states that any continuous function $f : [0, 1]^n \to [0, 1]$ can be approximated arbitrarily well by a neural network with at least 1 hidden layer with a finite number of weights.

3 Deep Learning

3.1 Goal

We are trying to model:

$$y = w^T \phi(x)$$

but ϕ can be any non linearity. So how to model it to cover all types of functions?

Take a linear function: f(x) = $A^{m \times d}$ x

Now let's apply a non-linear function g such as relu, sigmoid or tanh to all the points and get a vector. We get:

$$g(Ax) = Relu(Ax)\ or\ tanh(Ax)\ or\ \sigma(Ax)$$

Next, let's apply another linear function h. This can be achieved by multiplying it by a matrix B. So we get:

$$h(g(Ax) = Bg(Ax)$$

As per the universal approximation theorem, If non linear function is fixed, then there exists an A and B which can model any arbitrary non linear function.
'm' will determine how close it is to the original function.

$$\|Bg(Ax) - w^T \phi(x)\| < \epsilon$$

3.2 Algorithm

Our goal is to minimize the following loss function:

$$\min_{A,B} \sum (y_i - Bg(Ax))^2$$

So we start with random A and B and perform gradient descent:

$$A_{t+1} \leftarrow A_t - r\Delta_A\, l(A, B)|_{A_t, B_t}$$
$$B_{t+1} \leftarrow B_t - r\Delta_B\, l(A, B)|_{A_t, B_t}$$

To deal with multiple minimas and to find the global minima, we repeat the process with another set of A and B multiple times. in the end, we compare losses from all sets and choose the minimum one.

for $t \in 1 \dots T$ **do**
  $loss \leftarrow 0$
  for $i \in D$ **do**
    $loss+ = (y_i - Bg(Ax))^2$
    $L \leftarrow loss$
    $A, B \leftarrow GradientDescent(L)$
  end for
end for

However, we run into a problem here. Since we are using for loop, it will take a tremendous amount of time to calculate the loss. Because we are calculating loss for individual points. This can be avoided if we tensorize the loss. and calculate the loss matrix and sum it as follows:

$$L = (Y - Bg(AX))^2$$
$$L = L.sum()$$

However we run into another problem as this we require a lot of memory. So to rectify this we tensorize in parts. This process is called batching and it allows for best utilization of memory

3.3 Batching

3.3.1 How to modify the algorithm

Divide the dataset into random batches. For doing so, we can use numpy function numpy.random .shuffle(). Dimension of X is taken as N x d, batch size = b. So the modified code would look like

for $t \in 1$ to T **do**
  $loss \leftarrow 0$
  $Index \leftarrow$ Permute(1..dim(Y))
  $X' \leftarrow$ X[Index]
  for $i \in 0$ to $\lfloor N/b \rfloor - 1$ **do**
    $X" \leftarrow X'$[ib:(i+1)b]
    $Y" \leftarrow Y'$[ib:(i+1)b]
    $L \leftarrow (Y" - B(g(AX")))^2$.sum()
    $A, B \leftarrow$ gradDes(L)
  end for
end for

But why we are randomly splitting the dataset to take one batch from it?
If we perform gradient descent on entire dataset it will be slow. Instead we are permuting the dataset, taking a batch from it and performing gradient descent on that batch. It is faster. To do so,

we need to keep in mind that the batch that we are selecting should represent the entire dataset. If we choose a batch without permuting the data, it might overfit one particular batch. On the other hand, if we are performing gradient descent on the entire dataset it might underfit.

4 References

[1] http://mitliagkas.github.io/ift6085-2020/ift-6085-lecture-10-notes.pdf

CS419M

Lecture 21: Deep Learning & Transformers

12 April 2023

Lecturer: Abir De *Scribe: Shubhhi*

1 Deep Learning in Natural Language Processing

In the context of NLP, deep learning algorithms are used to teach machines to understand language by analyzing large amounts of text data. This enables machines to identify patterns and relationships within the data, which can then be used to make predictions or generate new language.
S = We are students of IIT Bombay
Then for modelling this statement we find the relationship between the words.
P(S) = P('we') * P('are' | 'we') ..... * P('Bombay' | ...)
To create word relations, a machine learning algorithm is trained on a large set of text data. During training, the algorithm learns to predict the context in which each word appears, based on the words that appear around it. The resulting word vectors capture the meaning of each word in a high-dimensional space, based on its relationship to other words in the text.

2 Vectorization

Vectorization is jargon for a classic approach of converting input data from its raw format (i.e. text) into vectors of real numbers which is the format that ML models support. This is used in NLP.

$$W_{t+1} = F(W_t, h_t)$$

W_i are the word in vectorized form and h_t summarizes the previous state.

We have,

$$h_{t+1} = F(W_t, h_t) \quad W_{t+1} = G(h_{t+1})$$

We can attach positional encoding to each word

0	1	2	3	4	5	6	
W_1	W_2	W_3	W_4	W_5	W_6	W_7	...

This allows us to show sequential dependence using positional encoding. Word Embeddings or Word vectorization is a methodology in NLP to map words or phrases from vocabulary to a corresponding vector of real numbers which used to find word predictions, word similarities/semantics.

$$h_{w_i} = \sum^{j} A_\theta(w_i, w_j) V_\phi(w_j) \tag{1}$$

h_{w_i} is contextual vector
(w_i, w_j) is the dependence
V_ϕ is value of each word

3 Scope

All this is indeed the backbone of Google search and ChatGPT. Relevant discussion was held during lecture regarding the various limitations of a google search.

Lecture 22: Deep Learning Tutorial

13 april 2023

Lecturer: Abir De *Scribe: Abhishek, Tanmay, Sunandinee & Naman*

1 Introduction

In this tutorial, we will

1. Train a dummy neural network on a classification dataset and learn things like: using dataloaders, learning rates, checkpointing the best model, training followed by inferencing, early stopping
2. Experiment with different hyperparameters that may increase/decrease the accuracy score:
   (a) Effect of different batch sizes on model convergence
   (b) Effect of different learning rates on task loss/accuracy
   (c) Effect of different activation functions on task loss/accuracy

2 Train a dummy neural network on a classification dataset

We first installed transformers and imported all important libraries. Then we are loading mnist dataset using following code snippet.

```
# loading the mnist dataset, train and test are datasets containing tensors
mnist_train = datasets.MNIST('data', train = True, download = True, transform=
    transforms.ToTensor())
mnist_test = datasets.MNIST('data', train = False, download = True, transform=
    transforms.ToTensor())
```

The first line loads the training set of the MNIST dataset and stores it in a variable named mnist_train, and the second line loads the test set and stores it in a variable named mnist_test. Both mnist_train and mnist_test are PyTorch datasets containing tensors.

```
print(mnist_train)
print(mnist_test)
print("Image tensor shape {}".format(list(mnist_train)[0][0].shape))
print(type(list(mnist_train)[0][0]))
```

Here the third line prints out the shape of the first image tensor in the mnist_train dataset. It does this by converting the first sample in the dataset to a list and then accessing the first element of that

list, which contains the tensor representing the first image. The shape of the tensor is printed out using the shape attribute of the tensor. This line provides information about the size of the image tensor, which is (1, 28, 28) representing that each image is a grayscale image of size 28x28 pixels. The fourth line provides information about the data type of the tensor, which is a PyTorch tensor.

```
evens = list(range(0, len(mnist_train), 10))
odds = list(range(1, len(mnist_test), 10))
mnist_train = torch.utils.data.Subset(mnist_train, evens)
mnist_test = torch.utils.data.Subset(mnist_test, odds)
print("Final train and test sizes are {}, {}".format(len(mnist_train), len(
    mnist_test)))
```

Here, we select only a few samples for our training to have a lesser training time.
We are creating lists called evens and odds containing the indices of every 10th sample in the MNIST training set and test set respectively. We use torch.data.subset to select a subset of the mnist data
Line 3 creates a new subset of the MNIST training dataset called mnist_train, containing only the samples whose indices are listed in the evens list. Similiar working is of line 4.
Final train and test sizes are 6000, 1000

```
def set_seed(args):
    random.seed(args["seed"])
    np.random.seed(args["seed"])
    torch.manual_seed(args["seed"])
    torch.cuda.manual_seed_all(args["seed"])
```

Here we are defining a new function setseed. The function first sets the seed for the built-in "random" module using the value of "seed" key from the "args" dictionary. This ensures that any random number generation performed by the "random" module is reproducible, i.e., given the same seed, the module will generate the same sequence of random numbers.

Next, it sets the seed for the NumPy random number generator using the same seed value. This ensures that any random number generation performed by NumPy is also reproducible.

Then, it sets the seed for the PyTorch random number generator using the same seed value. This ensures that any random number generation performed by PyTorch is also reproducible.

Finally, it sets the seed for all CUDA devices using the same seed value.

```
args = {"seed": 42}
device = torch.device("cpu")
args["device"] = device
print(args["device"])

set_seed(args)
```

Here dictionary args is created with "seed" key of value 42. The device is set to the CPU. Then new key devices is added to args. Then finally setseed function is called.

```
class Net(nn.Module):

    def __init__(self, args):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(28*28, 80)
        self.fc2 = nn.Linear(80, 30)
        self.fc3 = nn.Linear(30, 10)

    def forward(self, x):
        x = x.view(-1, 28*28)
        x = args["activation"](self.fc1(x))
        x = args["activation"](self.fc2(x))
        x = self.fc3(x)
        return x
```

Here Net defines a simple fully connected neural network for classifying images of handwritten digits from the MNIST dataset. The init method defines the architecture of the network by creating three linear layers (fc1, fc2, and fc3) with different input and output sizes.

The forward method defines how input data is passed through the network during the forward pass. The input x is first flattened into a 1D tensor using the view method, then passed through the fc1 layer, followed by an activation function specified in the args dictionary (e.g. nn.ReLU() or nn.Tanh()). The output of fc1 is then passed through fc2 and another activation function, followed by the final fc3 layer to produce a 10-dimensional output tensor representing the logits for each class.

```
# training loop
def train(args, train_dataset, val_dataset, model):

    # Prepare train data
    train_sampler = RandomSampler(train_dataset)  # random sampling of
    training data

    train_dataloader = DataLoader(
        train_dataset, sampler=train_sampler, batch_size=args["
    train_batch_size"])
    train_batch_size = args["train_batch_size"]

    t_total = len(train_dataloader) * args["num_train_epochs"]
    optimizer = args["optimizer"](model.parameters(), lr=args["learning_rate"
    ], eps=args["adam_epsilon"])

    # explain what is learning rate warmup
    scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps=
    t_total // 10, num_training_steps=t_total)
    criterion = nn.CrossEntropyLoss()  # defining the loss function

    # Train!
    print("***** Running training *****")
    print("  Num examples = ", len(train_dataset))
    print("  Num Epochs = ", args["num_train_epochs"])
    print("  Instantaneous batch size per GPU = ", train_batch_size)

    global_step = 0
    train_losses, val_losses = [], []
```

```
    train_acc, val_acc = [], []
    tr_loss, logging_loss = 0.0, 0.0
    model.zero_grad()

    train_iterator = trange(int(args["num_train_epochs"]), desc="Epoch")

    best_f1_score = 0
    if not os.path.exists(args["output_dir"]):
        os.makedirs(args["output_dir"])

    patience = 3
    last_best_epoch = -1


    for epoch in train_iterator:
        epoch_iterator = tqdm(train_dataloader, desc="Iteration")

        for step, batch in enumerate(epoch_iterator):
            model.train()

            batch = tuple(t.to(args["device"]) for t in batch)  # bringing the examples on same device as the model
            input_, labels_ = batch
            outputs = model(input_)

            loss = criterion(outputs, labels_)

            loss.backward()

            # gradient clipping
            torch.nn.utils.clip_grad_norm_(
                model.parameters(), args["max_grad_norm"])

            tr_loss += loss.item()
            optimizer.step()
            scheduler.step()
            model.zero_grad()
            optimizer.zero_grad()
            global_step += 1

        print("Train loss: {}".format(tr_loss/global_step))
        train_losses.append(tr_loss/global_step)

        # get train accuracy
        print("Train accuracy stats: ")
        results = evaluate(args, train_dataset, model)
        print("Train accuracy: {}".format(results["acc"]))
        train_acc.append(results["acc"])

        # Recording validation f1 scores
        results = evaluate(args, val_dataset, model)
```



```
        print("Validation accuracy: {}".format(results["acc"]))
        print("Validation loss: {}".format(results["eval_loss"]))

        val_losses.append(results["eval_loss"])
        val_acc.append(results["acc"])

        if results.get('f1') > best_f1_score and args["save_steps"] > 0:
            best_f1_score = results.get('f1')
            model_to_save = model.module if hasattr(model, "module") else model
            torch.save(model_to_save.state_dict(), args["output_dir"] + "clssnn.pth")
            torch.save(args, os.path.join(args["output_dir"], "training_args.bin"))
            last_best_epoch = epoch
            print("Last best epoch is {}".format(last_best_epoch))
        elif epoch - last_best_epoch > patience:
            print("Early stopped at epoch {}".format(epoch))
            break

    return train_losses, train_acc, val_losses, val_acc

def evaluate(args, val_dataset, model):

    eval_sampler = SequentialSampler(val_dataset)
    eval_dataloader = DataLoader(
        val_dataset, sampler=eval_sampler, batch_size=args["eval_batch_size"])

    results = {}
    criterion = nn.CrossEntropyLoss()

    print("  Num examples = ", len(val_dataset))
    print("  Batch size = ", args["eval_batch_size"])
    eval_loss = 0.0
    nb_eval_steps = 0
    preds = None
    out_label_ids = None

    for batch in tqdm(eval_dataloader, desc="Evaluating"):
        model.eval()
        batch = tuple(t.to(args["device"]) for t in batch)

        with torch.no_grad():
            inputs, labels_ = batch

            outputs = model(inputs)  # forward pass
            logits = outputs

            loss = criterion(outputs, labels_)
            eval_loss += loss.mean().item()

```

```
        nb_eval_steps += 1

        if preds is None:
            preds = logits.detach().cpu().numpy()
            out_label_ids = labels_.detach().cpu().numpy()
        else:
            preds = np.append(preds, logits.detach().cpu().numpy(), axis=0)
            out_label_ids = np.append(
                out_label_ids, labels_.detach().cpu().numpy(), axis=0)

    eval_loss = eval_loss / nb_eval_steps
    preds = np.argmax(preds, axis=1)
    result = acc_and_f1(preds, out_label_ids)
    results.update(result)
    results["eval_loss"] = eval_loss

    return results


def simple_accuracy(preds, labels):
    return (preds == labels).mean()

def acc_and_f1(preds, labels):
    acc = simple_accuracy(preds, labels)
    f1 = f1_score(y_true=labels, y_pred=preds, average='weighted')
    precision = precision_score(
        y_true=labels, y_pred=preds, average='weighted')
    recall = recall_score(y_true=labels, y_pred=preds, average='weighted')

    return{
        "acc": acc,
        "f1": f1,
        "acc_and_f1": (acc + f1) / 2,
        "precision": precision,
        "recall": recall
    }
```

This code defines a function called train that is responsible for training a deep learning model. The function takes several arguments including args which is a dictionary of various hyperparameters, train_dataset which is the training dataset, val_dataset which is the validation dataset, and model which is the deep learning model being trained.

The first step in the train function is to prepare the training data. This involves creating a RandomSampler object to randomly sample the training data, creating a DataLoader object to load the data in batches, and setting the batch size. The total number of training steps (t_total) is calculated based on the number of epochs and the number of steps per epoch. The optimizer and learning rate scheduler are also created.

The train function then enters a loop that iterates over the specified number of epochs. Within each epoch, the function iterates over the batches of data in the train_dataloader. For each batch, the model is set to training mode (model.train()) and the input data and labels are loaded onto the

same device as the model using the to() method. The model is then run on the input data (outputs = model(input_)) and the loss is calculated using a cross-entropy loss function. The loss is then back-propagated through the model (loss.backward()) and the gradients are clipped to prevent them from becoming too large using torch.nn.utils.clip_grad_norm_(). The optimizer is then updated (optimizer.step()) and the gradients and loss are reset (model.zero_grad() and optimizer.zero_grad()).

After each epoch, the function calculates and prints the average training loss and accuracy (train_loss and results["acc"]), and the average validation loss and accuracy (results["eval_loss"] and results["acc"]). The function also saves the best model based on the validation accuracy and stops training early if the model does not improve for a specified number of epochs.

The evaluate function is called within the train function to evaluate the model on the validation dataset. This function is similar to the training loop, but it does not involve backpropagation or updating the model parameters. The function loads the data onto the device, runs the model on the input data, calculates the loss and accuracy, and returns the results.

Overall, the train function implements a standard training loop for a deep learning model with cross-entropy loss and gradient descent optimization. The function also includes early stopping and model checkpointing to prevent overfitting and improve model performance.

```
# defining training hyperparameters

args["train_batch_size"] = 60
args["eval_batch_size"] = 32
args["num_train_epochs"] = 5
args["optimizer"] = AdamW
args["learning_rate"] = 1.5e-3
args["adam_epsilon"] = 1e-8
args["output_dir"] = "./output/"
args["max_grad_norm"] = 1.0
args["save_steps"] = 1
args["activation"] = F.relu

model = Net(args)
model.to(args["device"])
```

In this section of the code, we are defining the hyperparameters for training our neural network model. These hyperparameters specify various settings for the training process, such as the batch size, number of epochs, and learning rate.

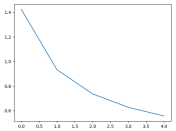
Finally, we create an instance of the Net class, which represents our neural network model. We pass in the args dictionary to configure the model's hyperparameters, and call model.to(args["device"]) to move the model to the specified device for computation.

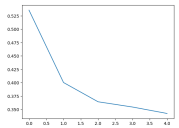
Now we'll call train function

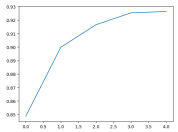
```
train_losses, train_acc, val_losses, val_acc = train(args, mnist_train,
    mnist_test, model)
```

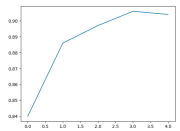
The train function is responsible for training the model on the training set, evaluating the model on the validation set, and returning the training and validation losses and accuracies at each epoch.

train_losses and val_losses are lists that contain the training and validation losses, respectively, for each epoch of training. The loss is a measure of how well the model is able to predict the


(a) train _losses


(b) val_losses


(c) train_acc


(d) val_acc

correct labels for the training and validation examples.

train_acc and val_acc are lists that contain the training and validation accuracies, respectively, for each epoch of training. The accuracy is a measure of how well the model is able to correctly classify the training and validation examples.

We then plotted all these four lists.

```
# Inference
def inference(model, sample):
    softmax = nn.Softmax(dim=-1)

    model.eval()

    with torch.no_grad():
        inputs, labels_ = sample

        logits = model(inputs)  # forward pass
        outputs = softmax(logits)
    print("Preds are {}".format(outputs))
    preds = outputs.detach().cpu().numpy()[0]
    logits = logits.detach().cpu().numpy()[0]
    print("Outputs are {}".format(preds))
    print("Logits are {}".format(logits))
    print("Predicted number is {}".format(np.argmax(preds)))
    print("Actual number is {}".format(labels_))


# load model
model = Net(args)
model.load_state_dict(torch.load("./output/clssnn.pth"))
model.to(args["device"])

sample = mnist_test[10]
inference(model, sample)
```

The code block above defines a function inference that takes a trained model and a sample as input and performs inference to predict the label of the input sample.

The inference function first initializes a softmax layer, sets the model to evaluation mode (using model.eval()) and then performs a forward pass through the model with the input sample to obtain the logits. The logits are then passed through the softmax layer to obtain the predicted probabilities for each class.

The function then prints the predicted probabilities and the actual label of the input sample, as

well as the predicted number (which is the class with the highest probability). The function also returns the predicted probabilities as a numpy array.

The code then loads the trained model using model.load_state_dict(torch.load("./output/clssnn.pth")) and moves the model to the appropriate device using model.to(args["device"]). Finally, the function is called with a sample from the test set to perform inference and print the results.

3 Experiment with different Training Batch Sizes

```
from collections import defaultdict
```

Here we are importing Defaultdict: a sub-class of the dictionary class that returns a dictionary-like object, and which like dictionaries is a container and is present in the module collections. The functionality of both dictionaries and defaultdict are almost the same except for the fact that defaultdict never raises a KeyError. Instead, for the keys that do not exist it provides a default value. This solves the issues that crop up when KeyError is raised.

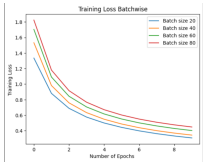
```
batch_sizes = [20,40,60,80]
train_loss_df = defaultdict()
train_acc_df = defaultdict()
val_loss_df = defaultdict()
val_acc_df = defaultdict()

for bs in batch_sizes:
    args["train_batch_size"] = bs
    model = Net(args)
    model.to(args["device"])
    train_losses, train_acc, val_losses, val_acc = train(args, mnist_train,
    mnist_test, model)
    train_loss_df[bs] = train_losses
    train_acc_df[bs] = train_acc
    val_loss_df[bs] = val_losses
    val_acc_df[bs] =  val_acc
```

Batch size is a parameter that defines the number of samples that will be propagated through the network. It gives the number of data points we train our model over, in every iteration before the model is updated. While it controls the accuracy of the estimate of the error gradient, there is a tension between batch size and the speed and stability of the learning process.

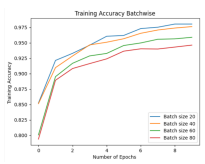
Line 1 of this code defines a list of batch sizes to be used for training - 20, 40, 60 and 80.
Then we define four defaultdict objects for storing the training loss, training accuracy, validation loss and validation accuracy values for each batch size. The batch size argument in the "args" dictionary holds the current batch size value. An object of the class "Net" is instantiated using the updated "args" dictionary.
Then the "train" function with the updated "args" dictionary is called, along with the training and testing datasets, and the model object as arguments. This function trains the model on the training dataset, evaluates its performance on the testing dataset, and returns the training and validation loss

and accuracy values which are stored in the respective defaultdict lists using the current batch size as the key.

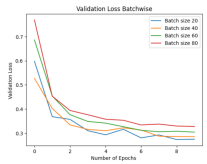
The resulting loss and accuracy values for each batch size can be used to compare their performance and select the best one.



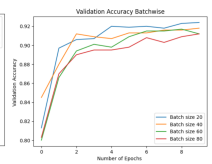
(a) Training Loss Batchwise



(b) Training Accuracy Batchwise



(c) Validation Loss Batchwise



(d) Validation Accuracy Batchwise

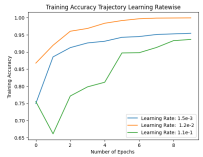
4 Experiment with different Learning Rates

```
learning_rates = [1.5e-3,1.2e-2,1.1e-1]
train_loss_df = defaultdict()
train_acc_df = defaultdict()
val_loss_df = defaultdict()
val_acc_df = defaultdict()

for bs in learning_rates:
    args["learning_rate"] = bs
    model = Net(args)
    model.to(args["device"])
    train_losses, train_acc, val_losses, val_acc = train(args, mnist_train,
    mnist_test, model)
    train_loss_df[bs] = train_losses
    train_acc_df[bs] = train_acc
    val_loss_df[bs] = val_losses
    val_acc_df[bs] =  val_acc
```

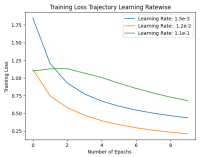
The learning rate is a hyper-parameter that determines the step size at which a machine learning model's parameters are updated during training. In other words, it controls the speed at which the model learns from the data. The learning rate determines how much the parameter is adjusted during each iteration of training.

The code then initializes empty dictionaries train_loss_df, train_acc_df, val_loss_df, and val_acc_df using the defaultdict function from the collections module. These dictionaries will be used to store the training and validation loss and accuracy for each learning rate.

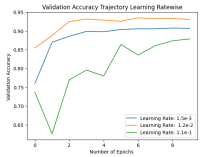
The code then loops over the learning rates using a for loop, setting the learning_rate parameter in the args dictionary to the current learning rate, and creates a new neural network model using the Net class defined elsewhere. The model is then moved to the device specified in the args dictionary which is the "Computer" in this case.



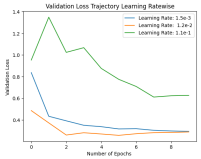
(a) Training accuracy



(b) Training loss



(c) Validation accuracy



(d) Validation loss

The train function is then called with the args dictionary, the MNIST training and test datasets, and the newly created model as arguments. The train function returns four lists: train_losses, train_acc, val_losses, and val_acc, which represent the training loss, training accuracy, validation loss, and validation accuracy over time during training.

Overall, this code is a simple example of how to train and evaluate a neural network model with different learning rates and record the training and validation performance for each learning rate.

Then we plot the Training accuracy and loss, and Validation accuracy and loss trajectory as a function of the learning rate:

5 Experiment with different Activation Functions

```
# defining training hyperparameters

args["train_batch_size"] = 60
args["eval_batch_size"] = 32
args["num_train_epochs"] = 5
args["optimizer"] = AdamW
args["learning_rate"] = 1.5e-3
args["adam_epsilon"] = 1e-8
args["output_dir"] = "./output/"
args["max_grad_norm"] = 1.0
args["save_steps"] = 1
```

In this section of the code, we are defining the hyperparameters for training our neural network model. These hyperparameters specify various settings for the training process, such as the batch size, number of epochs, and learning rate.

```
activationFunctions = [F.relu, F.tanh, F.sigmoid]
train_loss_df = defaultdict()
train_acc_df = defaultdict()
val_loss_df = defaultdict()
val_acc_df = defaultdict()

for bs in activationFunctions:
    args["activation"] = bs
    model = Net(args)
```

```
    model.to(args["device"])
    train_losses, train_acc, val_losses, val_acc = train(args, mnist_train,
    mnist_test, model)
    train_loss_df[bs] = train_losses
    train_acc_df[bs] = train_acc
    val_loss_df[bs] = val_losses
    val_acc_df[bs] =  val_acc
```

The code involves training the Net model with different activation functions. Three activation functions: Rectified Linear Unit (ReLU), hyperbolic tangent (Tanh), and sigmoid are used.

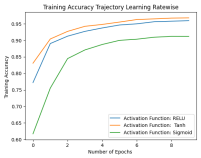
- activationFunctions: A list of activation functions to be experimented with
- train_loss_df, train_acc_df, val_loss_df, val_acc_df: Empty dictionaries to store the training loss, training accuracy, validation loss, and validation accuracy for each activation function

The code then enters a loop over the activationFunctions list. For each activation function bs in the list, the following steps are executed:

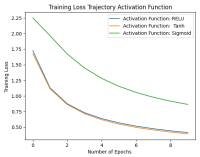
- The args dictionary is updated with the activation key set to bs
- A new neural network model is created using the updated args
- The model is moved to the device specified in args
- The train function is then called with the updated args, the training and testing datasets (mnist_train and mnist_test, respectively), and the created model. The train function performs training and evaluation on the model and returns four lists of values: train_losses, train_acc, val_losses, and val_acc
- These values are then added to their respective defaultdict objects using the bs activation function as the key

After the loop completes, the train_loss_df, train_acc_df, val_loss_df, and val_acc_df defaultdict objects contain lists of training and validation loss and accuracy values for each of the three activation functions.

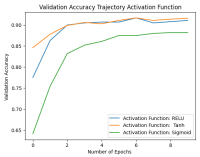
Then we plot the Training accuracy and loss, and Validation accuracy and loss trajectory as a function of the activation functions:



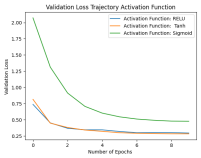
(a) Training accuracy



(b) Training loss



(c) Validation accuracy



(d) Validation loss