CS3244 Midterm Cheatsheet

Rishabh Anand, TG-06

Contents

Abstract

This document contains key facts, trivia, and knowledge you'll need to know to ace the midterms. This collection is in no way indicative of what will come out in the paper – it's a simple, comprehensive "cheatsheet" covering *everything* the topics within the scope of the midterms on Monday, 27 September, 4:00-6:00PM[1]. Do note that model implementation is not tested so please **do not** memorise the algorithms – understand them at a high level and apply. Likewise, do not memorise any Linear Algebra theorems or math equations. They are not the focus of this exam.

1 Introduction

There are 5 tribes in Machine Learning:

- **Symbolists** – Philosophy / Linguistics –
- **Analogisers** – Psychology
- **Evolutionaries** – Biology / Evolution
- **Bayesians** – Statistics
- **Connectionists** – Neuroscience

There are two main types of Machine Learning we are concerned with in CS3244:

- **Supervised Learning** – features and labels present. Mostly Classification or Regression
- **Unsupervised Learning** – only features present. Structural information needs to be captured

Your dataset has an implicit function that describes the relation between x and y. The main purpose of Machine Learning is to approximate that function using a trained model. However, there is probably no point using ML if,

- the task is fully deterministic
- there's a straightforward formula equating x and y
- the task demands different outputs every time, regardless of inputs (randomness)

2 k-NN

k-Nearest Neighbours is an Analog algorithm where the model is the data. There's no extra data preprocessing that needs to be done to start running inference (i.e., predicting) with this model. It follows the premise that data samples that are similar to one other *will* cluster together in n-dimensional space. This means samples belonging to a class will occupy more or less exist near one another, while samples belonging to another class will exist somewhere else (overlap of samples may or may not occur based on the dataset). You can find the k-NN algorithm in Algorithm 1.

[1]You can find the resources here: https://github.com/rish-16/CS3244-Tutorial-Material

Algorithm 1 Vanilla k-NN Algorithm

```
Require: dataset $(X, Y)$ with $m$ samples
Require: integer $k < m$
Require: new test instance $x_{\text{test}}$

Require: distances $D \leftarrow \{d(x_i, x_{\text{test}}) : x_i \in X\}$        ▷ $d(a, b)$ is a distance function
Require: sorted distances $S \leftarrow$ sorted$(D)$
Require: $k$ nearest points $S \leftarrow D[1 : k]$

  prediction $\leftarrow$ get_majority_label(S)
```

2.1 k-NN for Regression

k-NNs can be used for regression as well as classification. With a new observation x_{test}, calculate the mean of the k nearest neighbours and assign it as the prediction for this new observation.

$$y_{\text{test}} = \frac{1}{k}\sum_{i=1}^{k} x_i \qquad\qquad x_i \in S$$

where S is the sorted distances between all $x \in X$.

3 Decision Trees

Decision Trees have a systematic way of breaking down or *splitting* the dataset into granular sections based on certain properties. We discuss these properties below.

3.1 Definitions

3.1.1 Entropy

The amount of surprise, disorder, or uncertainty in something is called its Entropy. In Decision Trees, we seek to minimise Entropy.

3.1.2 Purity

Purity refers to the skew in distribution of points in the dataset or collection at each node. For example, a collection of 10 points with 9 class 1 points and 1 class 0 point has *higher* purity compared to a collection of 10 points with 6 class 1 points and 4 class 0 points.

3.1.3 Information Gain

Information Gain is the decrease in Entropy from a state of higher Entropy to a state of lower Entropy. When comparing two splits, how do we decide which one to take at a specific node? We look at which split yields the highest Information Gain.

3.2 Tree Building

We start off by deciding which of the features, and hence, which threshold for that feature, give us the highest Information Gain and Purity. The Decision Tree algorithm recursively creates nodes, moving from level to the next. At each node, it comes up with the best way of splitting the dataset based on some threshold (for example, all samples with feature $x_i < 0.75$ go left, otherwise right).

3.2.1 Calculating Information Gain

As mentioned above, Information Gain is the decrease in Entropy. That means we have to compute two Entropy values and compare them. Let's say we have C classes.

$$H(X) = -\sum_{i}^{C} X_i \log X_i$$

Suppose we have a few samples at a parent node with L of them going down the left branch and R going down the right branch. That means, if there's a complete dataset Y of size $L+R$ at the parent, let's say Y_L goes to the left and Y_R goes right. We calculate the Information Gain as follows.

$$\text{InfoGain(parentNode)} = H(Y) - (\frac{L}{L+R} \cdot H(Y_L) + \frac{R}{L+R} \cdot H(Y_R))$$

3.2.2 Using Information Gain

At each level, starting from the , here's what you do to create a Decision Tree

1. Look at all possible ways to split the data (i.e., look at all thresholds)
2. For each split, calculate Information Gain
3. Pick the split with the highest Information Gain relative to the Parent Node
4. Break the dataset according to that best split into `LeftDataset` and `RightDataset`
5. If there's no way to split the dataset, it's a leaf node
6. Repeat from 1. until there are only leaf nodes remaining

4 Linear Models

A line in 2-dimensional geometry is formulated with $y = mx + b$ where (x, y) is a point in $\mathbb{R}^2$, m is the gradient or slope, and b is the vertical intercept. In Machine Learning (and CS3244), we generalise a linear model to be of the following form:

$$\hat{y} = \theta_0 + \sum_{i=1}^{n} \theta_i \; x_i$$

where n is the number of features for every sample x, θ_0 is the bias (akin to b) and $\theta_{1 \to n}$ are the weights (akin to m). Together, the weights and bias are called the parameters of the model that are optimised during the course of training.

4.1 Linear Regression

4.1.1 Forward Pass

We use Linear Regression to predict real-values given some input x (a single feature or a vector of n features). For a specific sample $x \in \mathbb{R}^n$, our weights $w = \{\theta_{1 \to n}\}$ is also in $\mathbb{R}^n$. This way, we can simply perform a dot product $w^T \cdot x$ to get the weighted sum. Then we can add the bias $b = theta_0$ to get the prediction:

$$\hat{y} = w^T \cdot x + b$$

We call this the Forward Pass.

4.1.2 Measuring Error

To measure the error between the prediction $\hat{y}$ and ground truth y, we use a Loss/Cost Function. It gives us a numerical indication of how "off" our model is from the right answer. For Linear Regression, we use Mean Squared Error (MSE) as the loss function:

$$L = \mathbf{MSE}(y, \hat{y}) = \frac{1}{m} \sum_{i=0}^{m} (y - \hat{y})^2$$

We do this over all the points in the dataset at once to get the model's average error.

4.1.3 Backward Pass

We use Partial Differentiation to calculate the derivative/gradients of the Loss w.r.t. the weights w and b. It tells us by *how much* we must update our parameters.

$$w := w - \alpha \frac{\partial L}{\partial w}$$
$$b := b - \alpha \frac{\partial L}{\partial b}$$

Here, α is the Learning Rate or Step Size. It scales the gradient/derivative to an amount that helps us adjust the parameters decently.

This **Forward** $\rightarrow$ **Loss** $\rightarrow$ **Backward** process is repeated over N iterations to converge on a good set of parameters needed to approximate the relationship between x and y.

4.2 Logistic Regression

While Linear Regression is for regression, Logistic Regression is for classification – mostly problems with 2 classes (i.e., binary classification). Logistic Regression is similar to Linear Regression in that we compute $\hat{y} = w^T \cdot x + b$. However, for Logistic Regression, $\hat{y}$ is a real number and not a classification. To make the decision of classifying x as class 0 or 1, we use the Sigmoid Function:

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

It takes any number and squashes it into the range $[0, 1]$. If the final Sigmoid value is ≤ 0.5, we classify it as class 0, and it's ≥ 0.5, we classify it as class 1:

$$\hat{y} = \begin{cases} 0 & \sigma(z) < 0.5 \\ 1 & \sigma(z) \geq 0.5 \end{cases}$$

4.2.1 Measuring Error

Unlike Linear Regression, we use Cross Entropy Loss for Logistic Regression:

$$L = \text{CrossEntropyLoss}(y, \hat{y}) = -\sum_{i=1}^{m} y_i \log(\hat{y}_i)$$

However, when we are dealing with *only* 2 classes, we use a slightly modified version called Binary Cross Entropy (BCE):

$$L = \text{BCE}(y, \hat{y}) = -\sum_{i=1}^{m} y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)$$

For a more detailed explanation, please DM me on the CS3244 Slack at **[TA] Rishabh Anand**. I'm omitting it here to keep it short. It's not very important based on the module's scope at the moment.

4.3 Support Vector Machines

SVMs are another linear model that tries to draw a hyperplane between points from the two classes in question. A hyperplane is essentially the decision boundary that separates samples in one class from another. If we are dealing with points in $\mathbb{R}^n$, the hyperplane is *always* one dimension lower in $\mathbb{R}^{n-1}$. For example, if we're dealing with 2-dimensional data, the hyperplane is a line. If we're dealing with 3-dimension data, the hyperplane is a plane, and so on.

4.3.1 Forward Pass

The SVM equation is very similar to that of Linear Regression or Logistic Regression. I omit the bias b to keep it simple. Also, the classes are -1 and 1 for SVMs, instead of 0 and 1.

$$\hat{y} = w^T \cdot x$$

4.3.2 Measuring Error

The SVM introduces a new loss function called the Hinge Loss. This penalises the model only when it incorrectly classifies a point and gives no penalty when the model is correct.

$$\text{HingeLoss}(y, \hat{y}) = \begin{cases} 0 & y \cdot \hat{y} \geq 1 \\ 1 - y \cdot \hat{y} & \text{otherwise} \end{cases}$$

Say the prediction is positive (> 0) and the label is 1. The product of two positive numbers is positive, thereby giving us 0 error because the model classified correctly. Similarly, if the prediction is negative and the label is -1, the product of two negative numbers is also positive, also giving us 0 error. Now, if the prediction is positive and the label is negative (or vice versa), the product will always be negative, giving us an error of $1 - y \cdot \hat{y}$.
To get the average error, we can *only* consider the case when the sign of the prediction and label are different (denoted by the ψ):

$$L(y, \hat{y}) = -\frac{1}{m} \sum_{i=1}^{m} (1 - y \cdot \hat{y})_{\psi}$$

4.3.3 Backward Pass

Like Linear Regression, we use Partial Differentiation to get the gradients w.r.t. the weight w. Though, it's simpler because we can only consider the case when the model is wrong since the gradient of 0 is 0 (we don't need to care here). The gradient of Hinge Loss is as follows:

$$\frac{\partial L}{\partial w} = \begin{cases} 0 & y \cdot \hat{y} \geq 1 \\ -y \cdot x & \text{otherwise} \end{cases}$$

To update the weight, it's standard Gradient Descent:

$$w := w - \alpha \, \frac{\partial L}{\partial w}$$

5 Bias, Variance, and Noise

I omit this section because I've already given you a write up on Bias, Variance, Noise, and their various relationships. Again, please do reach out on Slack if you have doubts.

6 Regularisation and Validation

6.1 Regularisation

6.1.1 What is Regularisation

Regularisation is a technique used to minimise bias in a network. This means preventing the model from overfitting on a dataset or paying attention to less useful features by scaling down the parameters. To do so, we introduce a "penalty" to the loss function, thereby increasing it if the model does something it's not supposed to:

$$L = \frac{1}{m} \sum_{i=0}^{m} (y - \hat{y})^2 + \lambda \sum_{i=0}^{m} ||w||^2$$

λ is the regularisation parameter. We usually set this to a relatively number manually before training. When weights are large, they result in a large norm $|| \cdot ||^2$. When this large norm is multiplied by a large regularisation parameter, the loss value becomes very high, forcing the network to minimise the weights to counteract the increase in loss. Since the regularisation parameter is fixed, the model has only 1 degree of freedom – the weights. So, the model tries its best to bring it down, thereby reducing the chance of overfitting on the dataset because the weights for less useful features move closer to 0.

> Due to regularisation, there is a chance of your weights moving very close to 0. That's alright because we know the model is likely not overfitting on the data. If some weights zero out, it's not a cause for concern.

The main concern is the value of λ. If λ is too high, we penalise the model too much, causing weights to move to zero. This means our model becomes simpler and less complex which leads to underfitting or high bias. Consider this model:

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

If $\theta_{1\to n}$ goes too close to 0, it's like not having the weights at all and only retaining the bias θ_0. This is the situation we want to avoid when choosing λ.

6.1.2 Types of Regularisation

The two most popular types of Regularisation are *L1 Regularisation* (or Lasso Regularisation) and *L2 Regularisation* (or Ridge Regularisation).

In L1 Regularisation, we simply consider $\lambda \sum_{i=0}^{m} ||w||$ without the power. In L2 Regularisation, we consider $\lambda \sum_{i=0}^{m} ||w||^2$ with the norm raised to the power of 2 (i.e., the *L2 Norm*). L1 Regularisation produces a simple model whose predictions are easily explainable. It also retains a subset of features instead of zeroing them all out. L2 Regularisation prevents the weights from blowing up to large values by forcing them to move towards zero.

6.2 k-fold Cross Validation

6.2.1 How it's done

Given a dataset with m samples, we copy the dataset k times. For each copy, we take a new fold $i \in \{1, \ldots, k\}$ as the testing set and the other $k - 1$ folds collectively as the training dataset. For each model, we get the accuracy and discard the model. At the end, we have k training and testing accuracies. If they are all decently high with a low standard deviation from the mean accuracy, the model we are using is a good approach for the dataset.

6.2.2 Model Appropriateness, not Model Building

The purpose of k-fold Cross Validation is not to obtain a final model at the end, it's to check whether the algorithm we are using is appropriate for the prediction task. By algorithm, I mean Linear Regression, Logistic Regression, SVM, and others. Let's consider a dataset with a random 80-20 train-test split.
This split can either be very lucky (gives us the best model) or very unlucky (gives us the worst model). At the end we'll never know. However, let's say we use 10-fold Cross Validation. This means we have 10 sets of different 9/1 train-test folds that neither can be all lucky or all unlucky. Now, if the model performs decently on all 10 sets of folds, rest assured the model we're using is an appropriate

model for the problem. If there's poor performance on a majority of folds, we can move on to another algorithm and try again.
Again, the aim isn't to come up with a hypothesis. It's to see whether an algorithm is the best approach for a dataset.

7 Math and Common Derivatives

This section contains some popular derivatives and equations that might be of use during your time in CS3244. It's important to know how to take an equation and use Partial Differentiation to get the gradient w.r.t. the parameters.

7.1 Non-linear Activation Functions

7.1.1 Sigmoid / Logistic Function

Function:

$$\sigma(z) = \frac{1}{1+e^{-z}} \in [0,1] \quad , z \in [-\infty, \infty]$$

Derivative:

$$\sigma'(z) = \sigma(z) \cdot (1 - \sigma(z))$$

7.2 Loss Functions

7.2.1 Mean Squared Error

Function:

$$L = \mathbf{MSE}(y, \hat{y}) = \frac{1}{m}\sum_{i=0}^{m}(y_i - \hat{y}_i)^2$$

Derivative w.r.t w:

$$\frac{\partial L}{\partial w} = \frac{2}{m}\sum_{i=0}^{m}(y_i - \hat{y}_i)x_i$$

7.2.2 Cross Entropy / Log Loss

Function:

$$L = \text{CrossEntropyLoss}(y, \hat{y}) = -\sum_{i=1}^{m} y_i \log(\hat{y}_i)$$

Derivative:

$$\hat{y} = \sigma(w^T \cdot x)$$
$$\frac{\partial L}{\partial w} = -\frac{1}{m}\sum_{i=1}^{m} x_i \cdot y_i(1 - \hat{y}_i)$$

BCE Function:

$$L = \text{BCE}(y, \hat{y}) = -\sum_{i=1}^{m} y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)$$

BCE Derivative:

$$\hat{y} = \sigma(w^T \cdot x)$$
$$\frac{\partial L}{\partial w} = -\frac{1}{m}\sum_{i=1}^{m} x_i(y_i - \hat{y}_i)$$

7.2.3 Hinge Loss

Function:

$$L(y, \hat{y}) = \begin{cases} 0 & y \cdot \hat{y} \geq 1 \\ 1 - y \cdot \hat{y} & \text{otherwise} \end{cases}$$

Derivative:

$$\frac{\partial L}{\partial w} = \begin{cases} 0 & y \cdot \hat{y} \geq 1 \\ -y \cdot x & \text{otherwise} \end{cases}$$

7.3 Miscellaneous

7.3.1 Gradient Descent

$$\begin{aligned} z &= w^T \cdot x + b \\ \hat{y} &= \sigma(z) \\ L &= \dots \end{aligned}$$

by chain rule,

$$\begin{aligned} \frac{\partial L}{\partial w} &= \frac{\partial L}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial z} \times \frac{\partial z}{\partial w} \\ \frac{\partial L}{\partial b} &= \frac{\partial L}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial z} \times \frac{\partial z}{\partial b} \end{aligned}$$

update step:

$$\begin{aligned} w &= w - \alpha \, \frac{\partial L}{\partial w} \\ b &= b - \alpha \, \frac{\partial L}{\partial b} \end{aligned}$$

7.3.2 Analytical Linear Regression

You can use the analytical solution *only* for Linear Regression. You can simply plug in your values for features X and ground truths Y without performing Gradient Descent.

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

7.3.3 Bias Variance Decomposition

The data y is modeled by function $f(x)$ and stochastic noise which cannot be removed.

$$y = f(x) + \text{Stochastic Noise}$$

Furthermore, the function $f(x)$ is approximated by our ML model $h(x)$ and deterministic noise that can be minimised during training:

$$\begin{aligned} y &= f(x) \\ &= h(x) + \text{Deterministic Noise} \end{aligned}$$

If we had to expand the data y, it'd look like this:

$$y = h(x) + \text{Deterministic Noise} + \text{Stochastic Noise}$$

The total error is now made of deterministic noise (Bias), stochastic noise (irreducible error), and Variance.

$$y = \text{Irreducible Error} + \text{Variance} + \text{Bias}$$

More often than not, irreducible error has an expectation/mean of 0 and variance of σ^2

—————END—————