Midterm Exam Review Guide

Curated by: Rishabh Baral
Course: CMSC422 - An Introduction to Machine Learning
Exam Date: Oct 12, 2023, 11:00 AM

Disclaimer: This guide was made by me, a student. I am not a TA or a professor. If there are any errors in this document they are entirely my own. Please do not take this guide at its total face value unless a TA reviews it. If there are any errors, please contact me either by email at rbaral@terpmail.umd.edu, by phone at 214-701-0146, or via any channel of communication used for the course (Canvas, Piazza, and GroupMe).

Further Reference: For further information, please refer to the textbook **A Course in Machine Learning** by **Hal Daume.** The chapters associated with each topic will be in the topic heading. The textbook itself can be found online at A Course in Machine Learning**.**

Credit: All material is sourced from the Lecture Slides given by Prof. Teli and the textbook referenced above.

Topic 1: Introduction to Machine Learning

To delve into the intricacies of Decision Trees, which are undoubtedly an essential part of Machine Learning, it is imperative to first grasp the true essence of "Learning". For this class, "learning" refers to the process of acquiring expertise through experience. However, when we expand our horizons to Machine Learning, we need to modify this definition. According to the definition we will be using in this course, Machine Learning is the study of algorithms that help in performing specific tasks by identifying and utilizing patterns in data. This, in essence, is the crux of Machine Learning and has been the subject of extensive research over the years, with researchers striving to find the answer to one crucial question: "How can we generalize?"

To understand the importance of generalization, let us assume that we have been provided with a set of data that we need to classify based on certain criteria. Suppose that, in addition to the observations (features), we have also been provided with their corresponding classification (label). This scenario is an example of Supervised Learning, wherein we aim to see whether our model classifies the observations as expected. The existence of Supervised Learning implies the existence of "Unsupervised" learning as well, which is where the true essence of Machine Learning lies, especially in the context of Decision Trees. In Unsupervised Learning, we are given a set of data but are solely provided with the observations (features) and do not have access to any of the classifications (labels). The objective of Unsupervised Learning is to generate labels from the dataset based on observed patterns, which can often lead to the formulation of Binary (Yes/No) questions that aid in classifying the observations in the dataset.

Learning, in most cases, involves two distinct sets of examples: one for training and one for testing. However, merely memorizing the training examples is not sufficient for effective learning. We need to not only memorize the training examples but also be able to generalize those examples so that we can classify the testing examples. Lastly, we also need some form of "inductive bias", which is a hypothesis that can help in making the classification of test examples easier and is plausible based on the testing data provided.

If we delve further into the formalization of Machine Learning, we can define two functions: one that generates the classification for an unknown point, and another that defines a method for calculating our cumulative error over the entire training set. When given a data distribution (training set) and a classification function, our goal is to refine that function in a manner that minimizes our classification error for the dataset, also known as the "Loss".

Topic 2: Decision Trees (CIML, Chapter 1)

Decision Trees are one of the most basic classification methods in Machine Learning. To understand Decision Trees, we must answer 4 questions that will allow us to explore decision trees and understand the ideas and purpose behind them.

1. What is a Decision Tree?

A Decision tree is essentially a method of classification dependent on binary questions, typically those that can be answered with yes or no. It is called a “Decision Tree” because the answers to these questions, which lead to various decisions, can be presented as a tree with the results of answering “yes” to a query on one branch and the results of answering “no” to a query on the other. These trees can have varying depths depending on the structure of the dataset for which they are designed, but all decision trees end with the decision for each branch presented in the leaves at the bottom of the tree.

In the context of Decision Trees, the questions asked to classify are called **features** whilst the answers to these questions (yes/no) are called the **feature values.** When attempting to construct a tree, we must first determine the features and feature values available. Only then can we begin constructing the tree, usually by choosing the “best” (most informative) feature first.

2. How do we learn a decision tree from the data?

Now, this raises the question of how to determine the objective “best” feature to begin the tree. To do this, we need to understand two key concepts: Entropy and Information Gain.

Entropy is a fundamental concept in Machine Learning that plays a critical role in decision-making processes. In the context of Machine Learning, entropy can be defined as the measure of impurity or uncertainty in a dataset. In other words, entropy is a metric that quantifies the amount of information that is required to correctly classify a given set of data. The higher the entropy, the more uncertain or impure the dataset is. Conversely, a lower entropy indicates a more organized or pure dataset. The formula for calculating the entropy of a dataset is as follows: $\boldsymbol{H(X) = -\{\sum_{i=1}^{n}[P(x=i) * log_2 P(x=i)]\}}$

Information Gain is the second fundamental concept with respect to Decision Trees. Information Gain is defined as the decrease in entropy obtained from splitting a dataset using a particular feature. But prior to using Information Gain, we must understand and calculate a special form of entropy known as Conditional Entropy. Conditional Entropy is the Entropy of a label when conditioned on a particular feature value. It is calculated using the following formula:

$$H(Y|X) = -\{\sum_{j=1}^{v}[P(X = X_j) * \sum_{k=1}^{n}(P(Y = Y_i|X = X_j) * \log_2 P(Y = Y_i|X = X_j))]\}$$

Once the Conditional Entropy has been calculated, we need to calculate the information gain, which is computed as follows: $Inf_{Gain}(Y) = H(Y) - H(Y|X)$. So, essentially, the information gain is the difference between the entropy and the conditional entropy. This is consistent with the definition of information gain as change in entropy obtained from splitting a dataset using a particular feature. When building a decision tree, you must calculate the entropy and information gain for all feature split combinations to determine which feature to split on. Often, the features having a positive Information Gain (decrease in entropy) are usually chosen for splitting.

3. What is the Inductive Bias?

With respect to decision trees, the Inductive Bias is that we don't want our tree to get too large, which can happen if there are many features to split the dataset on.

As such, we are more likely to favor the **smallest** tree that fits our dataset and can fully classify it. This is a principle known as **"Occam's Razor",** which states that for machine learning algorithms, we prefer to follow the simplest hypothesis that fits the data. In fact, we tend to prefer simpler and shorter hypotheses as opposed to longer and more complicated ones because a simple hypothesis that fits the data well is less likely to be a statistical coincidence when compared to a longer or more complex hypothesis. However, even though a simple hypothesis is not a statistical anomaly, as with any model, we must evaluate its ability to minimize the error (loss).

4. How do we make sure the tree can generalize well?

A decision tree can classify every training example perfectly and still make errors on the testing data. This can be due to a multitude of reasons, but the most common are that the training data is only a sample of the entire distribution (lack of representative sample) or that the training data may be noisy (the data may not reflect the general trend of the dataset or may be affected by other factors). As such, every decision tree model can suffer from one of two issues. It either Overfits (too good at training set) or Underfits (Not good at either set) the data. If a tree model underfits the data, it is likely that the dataset was presented well but the model did not learn enough information necessary to make a good generalization from it. If a tree model overfits the data, it is likely that the model was poorly designed and spent too much time trying to fit accurately to every single training point, including many of the noisy ones which leads to a more complex model.

To alleviate the issues that arise from using a single tree, three of the most common methods are to use Bootstrapping (Randomly sampling the variance of Y, the prediction, at various points in the tree), the use of Random Forests (Multiple trees whose predictions are either voted or averaged to give the actual prediction from the tree), and the AdaBoost Algorithm (combining multiple weaker learning models to make a single, stronger learning model).

Topic 3: Classification with Nearest Neighbors (CIML, Chapter 3)

Up to this point, we have discussed the main principles of Machine Learning namely that we prefer generalization over memorization and then we have seen those principles applied to machine learning problems through the lens of Decision Trees.

Decision trees are great models for machine learning but are specialized to only one kind of problem: Classification. There are other types of problems in Machine Learning, and one of the most common types of problems outside of Classification is the question of Prediction. In this topic, we will discuss one of the main algorithms for Prediction, known as the Nearest Neighbors Algorithm.

Before we delve into the nitty-gritty of Nearest Neighbors, let’s first define what prediction tasks really are. If we consider Prediction tasks as functions mapping input (observations) to output (labels), we can obtain an almost geometric view of a dataset in which each feature of an input feature vector is seen as one dimension. In this view, the input (observation) can be thought of as a point in D-Dimensional space, where d is the number of features available for each observation.

Now that we have the foundation for the Nearest Neighbors Algorithm, let's try to get a deeper understanding of the algorithm itself. The biggest advantage to the vector model of data is that it allows for geometric operations on the data, which radically simplifies the process. One of the most basic, and most important, geometric operations in a vector space is distance calculation, which is typically done using Euclidean Distance, given by the formula $d(a, b) = \left[\sum_{d=1}^{D}(a_d - b_d)^2\right]^{\frac{1}{2}}$. Now that we have a way to measure distance, we can somewhat label the points in the dataset. How? This is because of the “inductive bias” that an unknown point should likely have the same label as the point or points to which it is most similar. In fact, it is this “inductive bias” that serves as the crux of the Nearest Neighbors Classifier. How it works is that the entire training set is stored, and then pointwise distances are computed from the unknown test point to every point in the training set.

There are two types of Nearest Neighbors Classifiers.

1. K-Nearest Neighbors Classification

The standard Nearest Neighbors algorithm is simple and astonishingly effective. However, it has a flaw. Since no quantity of nearest neighbors is specified, it only considers the single nearest neighbor when classifying an unknown data point. This is often problematic because the “nearest” neighbor is likely a “noisy” value that does not line up with the intuitive classification of the data point. For this reason, the K-Nearest Neighbors Classification, often abbreviated KNN, makes use of the technique seen in “random forests” where multiple data points (how many is specified by the value of k itself) are used to provide a set of labels for the unknown point. However, unlike Random Forests, where the classification is averaged, in KNN, a vote is taken to determine what the classification should be. The “winning” vote is the label that is the most common among each of the K nearest neighbors to the unknown. But this is exactly what makes KNN dangerous. Decision Trees often used a few features that were determined to be “useful”. However, in KNN, every feature is used equally. Therefore, if there are only a few relevant features but many irrelevant ones, KNN will likely do poorly (overfit) on the data set.

2. Epsilon Ball Nearest Neighbors Classification

The second kind of Nearest Neighbors classification is the Epsilon Ball Nearest Neighbors Classification. Epsilon Ball Nearest Neighbors functions identically to KNN, except that it tries to fix the one flaw that KNN has. It is essentially KNN in that the entire training set is stored and pointwise distances are computed, but instead of using a specific number of values, it uses all the values that lie at or below a certain threshold defined by ε. In doing so, ε-Ball Nearest Neighbors attempts to alleviate the noise problem that often plagues the accuracy of KNN.

Topic 4: Perceptron Neural Network

Before we can talk about the Perceptron Network, we need to understand what a Neural Network is and how it's structured. But before we do that, we need to get some background on Neural Networks.

Recall from Biology that our brain is made of many small units called neurons that send electrical signals to one another. The rate of firing tells us how activated a neuron really is. If one central neuron (just for the sake of this example) is connected to n other neurons, each firing at a different rate, the combination of the firing rates will tell our central neuron how strong the message is. It can then decide whether to fire (activate) and if it does fire, how strongly to do so. However, the truth is that this is just a simplified example of the complexities in our brain. Neural Networks are designed in this exact manner solely because it is easier to imitate the brain than to attempt to design a network that mirrors it exactly.

Just for simplicity's sake, let's think of the "neural network" as a single neuron that is connected to D other neurons, one for each of the D input features. In the biological discussion, it was mentioned that the central neuron decided whether to fire based on a specific activation. In the Neural Network model, that activation is calculated using the following formula: $a = \sum_{d=1}^{D} w_d * x_d$. If this activation, a, ends up positive, the neuron predicts that the example is a positive class example (labeled as positive) and if the activation ends up negative, then the neuron predicts that the example is a negative class example (labeled as negative). But, in practice, it is often preferred that the activation be non-zero. As such, a constant bias term is always added to the activation equation to move the threshold. The updated activation equation is as follows: $a = \left[\sum_{d=1}^{D} w_d * x_d\right] + b$.

The perceptron Network is a classic Neural Network and learning algorithm. The goal of the algorithm is to be able to find a linear decision boundary (split) for a given dataset that is both accurate and has the lowest error possible. There are two main characteristics of the perceptron algorithm that make it almost as simple and efficient as KNN.

1. Perceptron is Online

An online algorithm, like perceptron, is one that doesn't consider the entire dataset at once but rather considers the dataset one observation at a time. The perceptron processes that example, passing it through its training and testing functions before moving on to the next observation in the dataset.

For understanding purposes, I am attaching the code for the perceptron train and test algorithms.

```
PerceptronTrain(D,MaxIter):
$w_d$←0,for all d=1...D //initialize weights
b←0//initialize bias
for iter=1...MaxIter do
          for all (x,y)∈D do
                    a←$\sum_{d=1}^{D} w_d * x_d + b$ //compute activation for this example
                    if y*a ≤ 0 then
                              $w_d$ ←$w_d + y * x_d$,for all d=1...D //update weights
                              b←b+y //update bias
```

```
PerceptronTest($w_0, w_1, \ldots, w_d$, b, $\hat{x}$):
        $a \leftarrow \sum_{d=1}^{D} w_d * \hat{x}_d + b$     //Compute activation for the test example
        return SIGN(a)
```

2. Perceptron is Error Driven

The perceptron algorithm is error driven. This means that if a function (“guess” if you will) is working well on testing examples as it is running, the same “guess” is carried on until a mistake is made. However, it is this error-driven nature of the “vanilla” perceptron that makes it vulnerable to being a bad fit. Let's indulge in a scenario where, for example, the perceptron currently running is given a set of 10000 examples. If this perceptron runs through 9,999 of them and finds a set of parameters that work well, it will continue to use this set of parameters. Now, however, let's consider that the 10,000th example causes the perceptron to make an error. The error-driven nature of the perceptron means that it will update the parameters that had been working so well. In fact, this is one of the downsides of the perceptron being error driven since later points are given more importance than earlier points.

3. Alleviating the issues plaguing the Perceptron

Let's indulge in a scenario where, for example, the perceptron currently running is given a set of 10000 examples. If this perceptron runs through 9,999 of them and finds a set of parameters that work well, it will continue to use this set of parameters. Now, however, let's consider that the 10,000th example causes the perceptron to make an error. The error-driven nature of the perceptron means that it will update the parameters that had been working so well. In fact, this is one of the downsides of the perceptron being error driven since later points are given more importance than earlier points. In fact, this update ruined a set of parameters that had served well for ***99.99%*** of the data! To alleviate the effects of this issue, we would like for the perceptron algorithm to be reworked so that weight vectors that “survive longer” will have more “say” than those that “err quickly”. This is the principle behind the **voted perceptron**, whereby the perceptron recalls how long each weight vector survives and gives a weight based on that. In our example, we were using a “vanilla” perceptron. If we were using a voted perceptron, the initial weight vector would be given a weight of 9999 (it survived 9999 iterations) and the updated weight vector would be given a weight of 1 (it was correct only for the 10000th iteration). However, because datasets may be large and contain many observations, the voted perceptron can be much slower (up to 1/1000 the speed) than the “vanilla” perceptron. To alleviate the speed issue of the voted perceptron and the update issue of the “vanilla” perceptron, the solution is to use the averaged perceptron, which is built on a similar principle to the voted perceptron, but uses a single averaged vector as opposed to multiple full size vectors.

For reference purposes, I am including the prediction equations for the voted and averaged perceptron.

Voted Perceptron

$$\hat{y} = sign\left(\sum_{k=1}^{K} c_k * sign(w_k * \hat{x} + b_k)\right)$$

Averaged Perceptron

$$\hat{y} = sign(\sum_{k=1}^{K} c_k * (w_k * \hat{x} + b_k))$$

4. Convergence Analysis of the Perceptron Algorithm

The perceptron convergence algorithm states that if a perceptron algorithm is run on a **linearly separable** dataset, D, and the perceptron has a positive margin ($\gamma > 0$), then the algorithm will converge after at most $\frac{1}{\gamma^2}$ updates. Now, the proof of this algorithm is elementary since it involves some clever algebraic trickery.

To prove this, we will need to show two things:

1. The optimal weight vector and the kth weight vector get closer as k grows

First, let us suppose that the kth update occurs for example (x,y). We are trying to show that, by the kth update, w_k is slowly aligning with $w_{optimal}$. Because the example (x,y) caused an update, we know from the error-driven nature of perceptron that this must have been misclassified. In this case, we can write the misclassification as $y * (w_{k-1} \cdot x) < 0$. Now, after the update we get that $w_k = w_{k-1} + y * x$. Doing a little algebra, we get:

$$w_{optimal} \cdot w_k = w_{optimal} \cdot (w_{k-1} + y * x)$$

Then, with some vector algebra, we can rewrite this as:

$$w_{optimal} \cdot w_{k-1} + y * (w_{optimal} \cdot x)$$

Which, because of the margin of $w_{optimal}$ being γ, we can safely say is:

$$w_{optimal} \cdot w_{k-1} + y * \left(w_{optimal} \cdot x\right) > w_{optimal} \cdot w_{k-1} + \gamma$$

Thus, we can safely say that for each update that the perceptron makes, the projection of w_k onto $w_{optimal}$ increases by at least γ, meaning that

$$w_{optimal} \cdot w_k \geq k * \gamma$$

2. The norm of the weight vector $||w_k||$ does not grow by much for each update of k.

Now that we have an equation relating the kth weight vector and the margin of the perceptron, we need to show that the relation is not just a mere coincidence. Let’s show this by computing the norm of w_k, which can be done using the following formula: $||w_k||^2$. If we simplify this by plugging in the known definition of w_k, we get that $\|w_k\|^2 = \|w_{k-1} + y * x\|^2$. Now, if we apply the quadratic rule to this, we can decompose this into:

$$\|w_{k-1} + y * x\|^2 = \|w_{k-1}\|^2 + y^2 * \|x\|^2 + 2 * y * w_{k-1} * x$$

Lastly, making some basic assumptions and setting the activation such that a<0, we can get that: $\|w_{k-1}\|^2 + y^2 * \|x\|^2 + 2 * y * w_{k-1} * x < \|w_{k-1}\|^2 + 1$

This gives us that the norm of w_k increases by at most 1 in each update cycle. Finally, we can prove that $\|w_k\|^2 \leq k$.

Now, we can use the result from the first part ($w_{optimal} \cdot w_k \geq k * \gamma$) and the result from the second part ($\sqrt{k} \geq \|w_k\|$) to get one combined equation of the form: $\sqrt{k} \geq \|w_k\| > w_{optimal} \cdot w_k \geq k * \gamma$. If we ignore the middle two terms, we see that the two extreme terms (leftmost and rightmost) will define the number of updates that can be made before no more are possible. As such, we can see that we get $\sqrt{k} \geq k * \gamma$. Doing some algebra, we can see that we get $\frac{1}{\sqrt{k}} \geq \gamma$ which can be rewritten as $k^{-\frac{1}{2}} \geq \gamma$, which can be reworked to become the all famous: $\boldsymbol{k \leq \frac{1}{\gamma^2}}$**.**

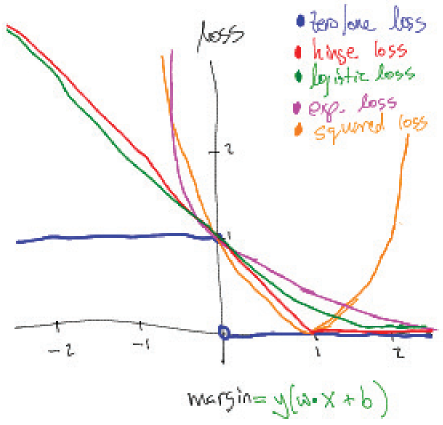
Topic 5: Linear Classifiers and Loss Functions (CIML, Chapter 7)

1. Linear Classifiers and Loss Functions

Recall that the perceptron algorithm was designed to serve one goal: it would find a **separating hyperplane** for some training dataset. To make understanding the linear classifier and perceptron easier, we will ignore overfitting for now. However, not all datasets are linearly separable meaning that the perceptron model may not be the best algorithm to use. In such a case, we know that there is not a perfect hyperplane, but we would like to find the hyperplane that makes the *fewest* errors. We can phrase this problem in the lens of optimization using the following equation: $\min_{w,b} \sum_n 1[y_n * (w \cdot x_n + b) > 0]$. This is called an objective function. In fact, the objective function is the thing that you are trying to minimize which is simply the error rate (or 0/1 loss) of the linear classifier that is parameterized by w and b. In the equation for optimization, the function 1[.] is the **indicator function**. In practice, this function returns one when true and zero otherwise. Because it is unbelievably difficult to find a hyperplane for data that is not linearly separable, we can attempt to make an estimation for the hyperplane by introducing what is known as a **regularizer** for the parameters (we will discuss this later). The introduction of the regularizer leads to the following modified objective function: $\min_{w,b} \sum_n 1[y_n * (w \cdot x_n + b) > 0] + \lambda * R(w, b)$.

In fact, optimizing the 0/1 loss is so hard that there is no objective function that can give us a good enough estimate. As such, we use surrogate loss functions to estimate the loss and provide an **upper bound** on the 0/1 loss. The reason that the 0/1 loss is so difficult to optimize is that it is a step function, meaning that it is nonconvex. Using surrogate loss functions, especially ones that are convex, we can strive to minimize loss with respect to linear models.

I am including the following information for reference:



Zero/One: $l_{\frac{0}{1}}(y, \hat{y}) = 1[y * \hat{y} \leq 0]$

Hinge: $l_{hin}(y, \hat{y}) = \max\{0, 1 - y * \hat{y}\}$

Logistic: $l_{log}(y, \hat{y}) = \frac{1}{\log 2}(\log(1 + e^{-y*\hat{y}}))$

(In logistic loss, the $\frac{1}{\log(2)}$ term ensures that $l_{log}(y, 0) = 1$.)

Exponential: $l_{exp}(y, \hat{y}) = e^{-y*\hat{y}}$

Squared Loss: $l_{sqr}(y, \hat{y}) = (y - \hat{y})^2$

2. Gradient Descent

Let's try to envision the following scenario. You are out for a hike in the mountains with your friends. You decide that you will hike blindfolded and see how quickly you can hike down the mountain. If you try to get down the mountain by feeling around and moving in what seems to be the most “downward” direction, you will eventually get to the base of the mountain. In exactly this manner, gradient based optimization moves “up” or “down” a function to get to the optimal point. At each step, it measures the gradient of the function it is trying to optimize. This measurement occurs at the current location, let's call it **x**. And the measurement made, which is the gradient at x, can be represented by **g**. The algorithm then takes a step in the direction of the gradient determined by the step size given by eta (η). If the step equation (equation for update) is given by $x \leftarrow x + \eta * g$, then the process is called **gradient ascent** and if the step equation (equation for update) is given by $x \leftarrow x - \eta * g$, then the process is called **gradient descent.** For any function or algorithm relating to objective functions, there is a regularized equation and gradient descent is no different. The regularized equation for gradient descent is as follows:

$$L(w, b) = \sum_n e^{-y_n*(w \cdot x_n + b)} + \frac{\lambda}{2}\|w\|^2$$

This looks almost like any other loss function, except that the usual λ in the regularizer is replaced with $\frac{\lambda}{2}$, but this replacement has no mathematical bearing and serves only to make calculations easier to perform with respect to gradients.

3. Subgradients

Gradient Descent is usually applied to convex functions. However, there are two problems. Firstly, Gradient Descent can become increasingly difficult for functions as they become more complex, leading to the algorithm not being the best. Secondly, and this is arguably the more exigent issue, some of the objective loss functions (Surrogate functions used to approximate the 0/1 loss) are not differentiable everywhere. For example, the Hinge Loss function isn't differentiable at x=1, the L1 Norm isn't differentiable at x=0, etc. The solution to this issue is a practice called Subgradient Optimization. Essentially, we ignore the discontinuity (point at which the function isn't differentiable) and apply gradient descent anyways. How we do this is that instead of taking full steps, we differentiate by parts to perform gradient descent on the smaller portions of the function.

Just for demonstrative purposes, let's look at how the Subrgadient Method would work for something like Hinge Loss.

Recall that Hinge Loss for a particular example is given by $\max\{0, 1 - y_n(w \cdot x_n + b)$. For us to be able to calculate the gradient descent, we need to split the Hinge Loss at its discontinuity (x=1). The equation now looks like this:

$0 \; if \; y_n(w \cdot x_n + b) > 1;$

$-y_n(w \cdot x_n + b) \; otherwise$

Now, let's take the partial derivative of this piecewise function with respect to the weight vector w, given by δ_w. After a few simplifications, we get the subgradient equation for the piecewise hinge loss function to be:

$0 \; if \; y_n(w \cdot x_n + b) > 1;$

$-y_n * x_n \; otherwise$

Topic 6: Naïve Bayes Classifier (CIML, Chapter 9)

The Naïve Bayes Classifier is one of the central models in a subset of machine learning known as the "probabilistic view" of Machine Learning. However, before we can discuss the Bayes Classifier, let us review what the Bayes Rule is first. The Bayes Rule can be summed up simply as "When we have some evidence and two prior probabilities: p(one event) and p(second event happening given that the first has happened), what is the chance that if the second event did happen that the first also happened?" Although this may seem unbelievably complicated, seeing the formula makes it all fall into place. The formula for Bayes Rule is: $P(A|B) = \frac{P(B|A)*P(A)}{P(B)}$. Now the power of Bayes Rule is that Bayes Rule can be applied to any distribution. For a joint distribution, for example, the probability of any logical expression is given by $P(E) = \sum_{rows\ matching\ E} P(Row)$. Now that we have some rudimentary understanding of Bayes Rule, we can try to tie this into Machine Learning. Recall from the first topic on Machine Learning that Machine Learning can be thought of as a Function Approximation. Also recall that Binary Classification when thought of as Function Approximation is essentially trying to find a function that can approximate the data and minimize the error. This is where the Bayes Classifier, also called the Bayes Optimal Classifier, can help. If we know the data generating distribution, D, we can define the Bayes Optimal Classifier as: $f_{BO}(\hat{x}) = \underset{\hat{y} \in y}{\operatorname{argmax}} D(\hat{x}, \hat{y})$. If this equation seems odd or abstract, it's not as complicated as it seems. In fact, all that the previous equation is saying is that "Out of all the possible classifiers, the Bayes Optimal Classifier achieves the **smallest** zero/one loss." Just as with any approximation, there is some error associated with the Bayes Optimal Classifier. This error is known as the Bayes Error and due to the nature of the Bayes Optimal Classifier, it is the minimum error for the 0/1 loss function. Now this sounds easy if we were given access to D, the data generating distribution. However, as much as we may try, we don't have access to D, meaning that we must attempt to estimate the Bayes Classifier instead. In doing so, we need to perform some form of "training" in which we estimate D from a finite training set operating under the assumption that D can only be a member of a particular family of probability distributions (Bernoulli, Gaussian, etc.). In addition, we need to perform some "learning" in which we estimate the parameters of D (such as estimating mean and covariance if D is Gaussian). One of the main assumptions about our training set that makes training simpler is the idea that the examples are Independently and Identically Distributed. The last concern is that we can never get the exact parameters needed to solve the equation perfectly. As such, we must settle for the set of parameters that maximize the probability of the data. Now you must be curious "Well what does that even mean?" In this context, maximizing the probability based on the set of parameters means that we try to find what is known as the "maximum likelihood estimation" for the parameters whereby we try to find the set of parameters that fits the data best and is most likely to occur.

One of the central assumptions of Naïve Bayes is the assumption that each data point and associated label are independent. Essentially, the central idea of Naïve Bayes can be given by the equation $P(X|Y) = \prod_{i=1}^{d} P(X_i|Y)$. Applying this logic, we can see that the Naïve Bayes Classifier is nothing more than Bayes Rule combined with the notion of conditional independence. As such, we can write the final prediction equation for Naïve Bayes Classifier as: $\hat{y} = \underset{y}{\operatorname{argmax}}\{P(y = y) * \prod_{i=1}^{d} P(X_i|Y)\}$.

Topic 7 : Logistic Regression

Let's remind ourselves that the task for the Naïve Bayes Classifier was to model the joint distribution of (x,y) pairs, which it did by giving us a probability, $p(y|x)$. So now, we seek to model that directly. If we begin with regression, we can get a general idea for how our model is going to look.

Let's assume that the relationship between x and y is linear, and is as given by $y = w \cdot x + b$. Essentially, we hope that $y = w \cdot x + b$ should hold for some pair of values (w,b). If we see real data, however, we notice that there are often deviations from $y = w \cdot x + b$. These deviations are what we would consider Noise. To really understand the probabilistic approach to Machine Learning, let's assume that the noise follows a distribution of some sort (such as a 0-mean Gaussian). This leads to the following three steps for all observations in the dataset, given by $n = 1 \ldots N$:

1. Compute $t_n = w \cdot x_n + b$
2. Choose the Noise function $e_n \sim Nor(0, \sigma^2)$
3. Get the label given by $y_n = t_n + e_n$

In the set of steps above, t_n refers to the target, a noiseless variable that cannot be observed directly and e_n refers to the noise for observation n. So in reality the y that we get is the sum of this noiseless target and some inherent noise. If we modify our Noise Function to be $Nor(w \cdot x_n + b, \sigma^2)$ and then take the log-likelihood of this function, we get that: $\log P(d) = -\frac{1}{2\sigma^2} \sum_n (w \cdot x_n + b - y_n)^2 + constant$. If we look at this model, we can see that it is exactly the Linear Regression Model.

Now, we can apply these same steps for binary classification if we choose a Bernoulli function instead of a Gaussian function.

1. Compute $t_n = \sigma(w \cdot x_n + b)$
2. Compute $z_n \sim Ber(t_n)$
3. Get the label given by $y_n = 2 * z_n - 1$

Now, if we try to get the log-likelihood of this model, we get that $\log P(d) = \sum_n \{[(y_n = 1) * (\log \sigma(w \cdot x_n + b))] + [(y_n = -1) * (\log \sigma(w \cdot x_n + b))]\}$, which reduces to (with some complex algebra), $-\sum_n l_{log}(y_n, w \cdot x_n + b\ )$. If you look at this, it is exactly the negative of (a scaled form of) the logistic loss. Because of this, the model we just derived is called the logistic regression model which is where logistic loss comes from.