CS M146:
Introduction to Machine Learning

Einar Balan

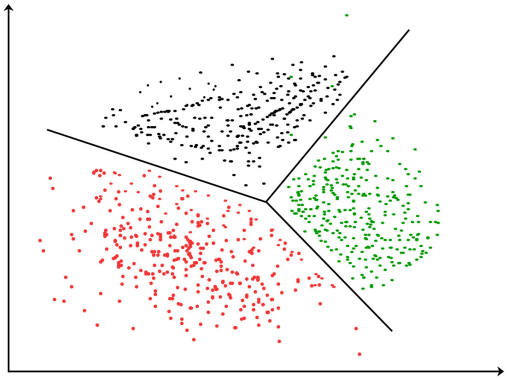
Contents

1 Introduction

Machine learning is the study of algorithms that improve performance when executing a task based on experience. For example, an algorithm that recognizes hand written digits. The task is the recognition, the performance is measured by the accuracy of recognition, and the experience is the database of human labeled images. Some more applications include reinforcement learning with playing games, language generation, and image generation (stable diffusion).

There are several major types of learning protocols

1. Supervised Learning
   - feed in **labeled** data in order to generalize to unencountered, unlabeled data
   - given target function $f : x \to y$ build a model g that mimics the behaviour of f in a generalized way
   - we build the model based on the training set and test it on the test set (80:20 split)
   - the test set should be labeled in order to measure accuracy of the model (but the model should never see these labels)
2. Unsupervised Learning
   - given **unlabeled** inputs, cluster them together based on traits in common



3. Reinforcement Learning
   - give sequences of states & actions w/ rewards
   - learn actions that maximize reward

1.1 Challenges in ML

- structured inference: classifcation may change based on context

- robustness ambiguities may arise that make classifcation difficult
- adversarial attack: adding a small amount of noise in a systematic way may change classification
- common sense: humans can reason through ambiguity based on context & common sense; this is more difficult for machines
- fairness & inclusion: models may treat different races differently and inadvertently excluse certain groups; stereotypes may also be reinforced

1.2 Defining a Supervised Learning Problem

Consider the Badges Game in which name badges at a conference were labeled with either a "+" or "-". Given a labeled training set, can we determine what general rules produced the labels?

Several important definitions:

- instance space - what features are we using to produce labels
- label space - what is our learning task i.e. what labels
- hypothesis space - what kind of model are we using
- loss function - how do we evaluate the performance of our model; what makes a good prediction

1.2.1 Instance Space

- consider $\vec{x} \in X$, where x is a feature vector in X our instance space, which is a vector space
- typically $\vec{x} \in \{0,1\}^n$ or $\mathbb{R}^n$
- each dimension of $\vec{x}$ represents a feature
- Examples of features: length of first name, does the name contain the letter X, how many vowels, is the nth letter a vowel, etc.
- Good features are **essential**; we cannot generalize without them

Example

X = [first-char-vowel, first-char-A, first-char-N]

Naoki Abe = [0, 0, 1]

1.2.2 Label Space

How should we classify based on $\vec{x}$? There are a couple options.

- Binary $y \in \{-1, 1\}$
- Multiclass $y \in \{1, 2, 3, ..., k\}$
- Regression $y \in \mathbb{R}$
- Structured Output $y \in \{1, 2, 3, ..., k\}^N$

Example Animal recognition $\vec{x}$: Image Bitmap y:

- Binary: Is it a lion?
- Multiclass: Is it a lion a cat or a dog?

- Multilabel: Is it a lion, mammal, cat, or dog?

1.2.3 Hypothesis Space

This is the set of all possible models. We need to find the best one for our use case. Consider an unknown boolean function. A potential hypothesis space could be every possible function.

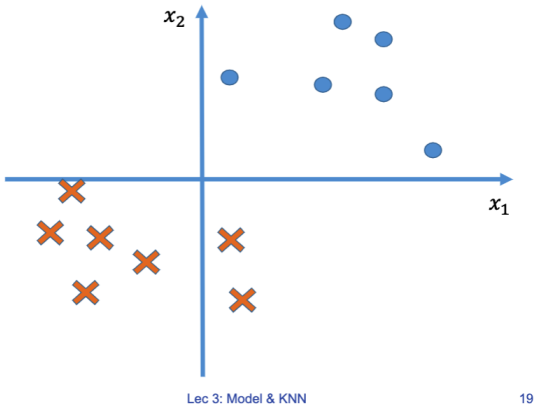
In general there are $|Y|^{|X|}$ possible functions from the instance space X to the label space Y. The hypothesis space is typically a subset of these. We can add rules to limit the hypothesis space, but we need to take care that these rules are not too restrictive, as it is possible that **no** simple rule can explain the data.

To **learn** is to remove remaining uncertainty and find the best function/model in our hypothsis space. In general it is a good idea to start the hypothesis space as restrictive, and get more general.

1.3 General Problem Flow

1. Develop a flexible hypothesis space i.e. Decision Tree, Neural Network, Nested Collections, etc
2. Develop algorithm for finding the best hypothesis
3. Hope that it generalizes beyond the data

Example



How to define hypothesis space?

- Option 1: Lines separating the two groups
- Option 2: Proximity to existing data represented by circles

Option 1 will likely generalize better. Option 2 will suffer due to **overfitting**. Underfitting, or not fitting the data well enough, is another concern.

How can we prevent overfitting? Some guidelines:

- use a simpler model e.g. linear
- add regularization
- add noise
- halt optimization earlier

How can we learn? Brute force or optimization with calculus.

Aside: Bias vs Variance

- bias: data is shifted a consistent amount
- variance: data is spread out around a point

2 | K-Nearest Neighbors

KNN is a type of supervised learning classifier which uses proximity to make classifications. It can be useful to determine a category for something i.e. spam or not spam, or type of flower. KNN is quite good at this.

2.1 Basic Algorithm

- Learning: just store training samples
- Prediction: Find k existing examples closest to input and group them; construct new labels based on k Neighbors
- Issues:
  - Need to define distance based on the domain i.e. Euclidean, Manhattan, L-p norm, Hamming (# diff bits), etc.
  - What is the best value of k?
  - How to aggregate the labels of the nearest neighbors (majority vote, weight by distance, etc)
  - How to store the data? (matrix, K-d tree)

Definition Inductive bias

The set of assumptions that the learner uses to predict outputs of unseen inputs

Ex) In KNN, the inductive bias is the the set of input data (the label of a point is similar to the label of nearby points)

2.2 Hyperparameters

A hyperparameter is a parameter that controls aspects of the model, as opposed to features in the data. For example, a hyperparameter could be k or the function used to measure distance. These are not specified by the algorithm and require empirical study to optimize. The best set of parameters is specific to the dataset.

2.2.1 Data Splits

It is advised to split your dataset into train, dev, and testing sets. The training set is used to train the model, the dev (AKA validation set) is used to find the best hyperparameters, and the test set is used to evaluate the final performance of the model.

How can we find the best hyperparameters? One way is the following:

- For each possible value of the parameter
  - train a model using the training set
  - evaluate the performance on the dev set
- choose the model with the best performance on the dev set
- (optionally) retrain the model on both the training set and the dev set with the best hyperparameters
- evaluate the final model on the test set

How to optimally split data between training and dev? If we have too little training data the model will not be robust and if we have too little dev data the dev dataset will not be representative of the entire dataset.

Solution: **N-fold cross validation**

- instead of a single training dev split, we split the data into N equal sized parts
- we train and test N different classifiers, and use a different part of the data as the dev set for each classifier
- report the average accuracy and standard deviation of the accuracy

2.3 Decision Boundary

The KNN algorithm is not explicitly building a function. So what is the implicit function that is being computed? How do we determine a decision boundary in the set of data points?

We use the "Voronoi Diagram." For every point X in a training set S, the **Voronoi Cell** is a polytope consisting of all points closer to x than any other points in S. The Voronoi diagram is the union of all these cells. Basically, its just a diagram indicating areas that are closest to each point. This is for $k = 1$. For $k > 1$, it will also partition the space but with a much more complex decision boundary.

2.4 Curse of Dimensionality

The more dimensions we have, the greater the proportion of points further away from the origin vs close to the origin. This essentially breaks KNN. Even if all features are relevant, in high dimensions, most points are equally far away from eachother, so it is difficult to classify them by proximity. How to deal with this?

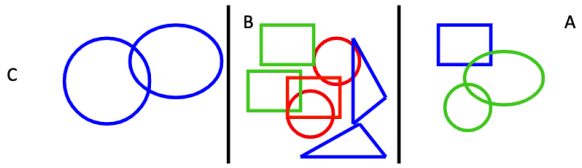
In practice, we apply dimensionality reduction. That is, we consolidate dimensions by removing irrelevant features and combining certain features.

2.5 Data Preprocessing

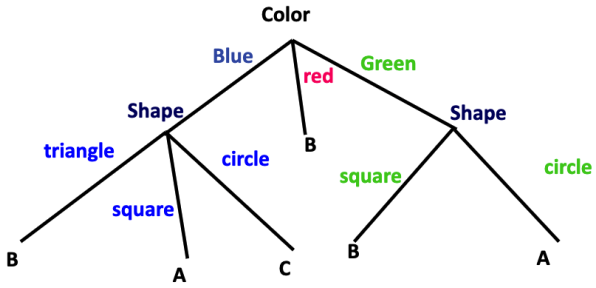
- normalize data to have zero mean and unit std in each dimensionality
- scale each feature accordingly

3 Decision Trees

Consider the sample dataset below:



What would be the label for a red triangle? We can construct a decision tree in order to determine the answer.



Based on this tree, we can see that a red triangle would be labeled B.

Many decisions can be modeled as tree structures in this way.

3.1 Representation

- decision trees are classifiers for instances represented as feature vectors
- nodes are tests for feature values
- we draw a branch for each value of a node's feature (for numerical values we use thresholds i.e. X > 100)
- leaves specify labels

We can use decision trees to express any boolean function. How do we learn the decision tree?

3.2 Algorithm

We recursively build the decision tree top down with funcion ID3(S, Attributes, Label) as follows:

- base case: if all examples are labeled the same, then return a single node with the label
- let A = the attribute that "best" classifies S
- For each possible value v of A
  - Add a new tree branch corresponding to A=v
  - Let S_v be the subset of examples in S w/ A = v
  - if S_v is empty, add leaf node with the common label in S
  - otherwise, add the subtree generated by ID3(S_v, Attributes - $\{A\}$, Label) at this branch

How do we determine the "best" attribute to split? The goal is to create the smallest decision tree possible (which can be accomplished w/ a greedy heuristic, sometimes). The problem itself is NP-hard, but it is good enough to be optimal most of the time.

3.3 Information Gain

Information gain is the heuristic described above. The idea is that gaining information will reduce uncertainty, which can be measured by entropy. The formal definition of entropy H for a set S is as follows

$$H[S] = -P_+ log_2(P_+) - P_- log_2(P_-)$$

where P_+ and P_- represent the proportin of positive and negative examples in S respectively.

More generally, if a random variable S has K different values, the entropy is given basicstyle

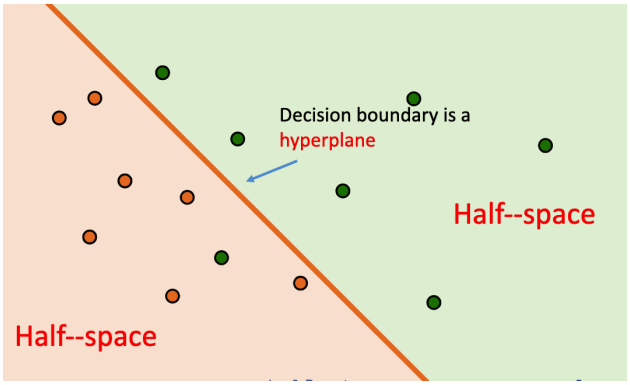
$$H[S] = -\sum_{v=1}^{K} P(S = a_v) log_2 P(S = a_v)$$

The information gain of an attribute is the expected reudction in entropy caused by partitioning on that attribute

$$Gain(S, A) = Entropy(S) - \sum_{v \in Values(a)} \frac{|S_v|}{|S|} Entropy(S_v)$$

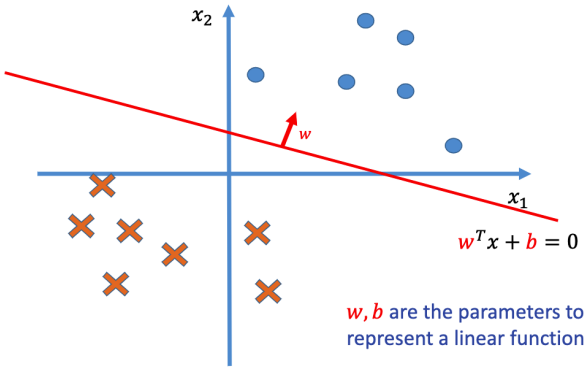
4 | Linear Models

In linear models, a hyperplane separates the data space in order to classify the data. Data points on either side of the hyperplane are classified differently. In two dimensions, a line is drawn to separate the data.



The hypothesis space consists of many potential linear models, some of which are better than others. In general, to be "better" means that the model separates the data better than other models.

We have two parameters for our linear model: w and b. w is a column vector representing how much each feature should be weighted and b is the bias.



We classify based on if $w^T x + b > 0$ or < 0.

Example If a line represented by $w^T x + b = 0$ passes through (0, 1) and (3, 0), what are w and b?

To solve we simply set up a system of equations over $w_1 x_1 + w_2 x_2 + b = 0$.

$$0x_1 + 1x_2 + b = 0$$
$$3x_1 + 0x_2 + b = 0$$

$$3x_1 - 1x_2 = 0$$
$$3x_1 = x_2$$

So one solution is $w^T = [1, 3]$ and $b = -3$. Note that there are infinitely many solutions.

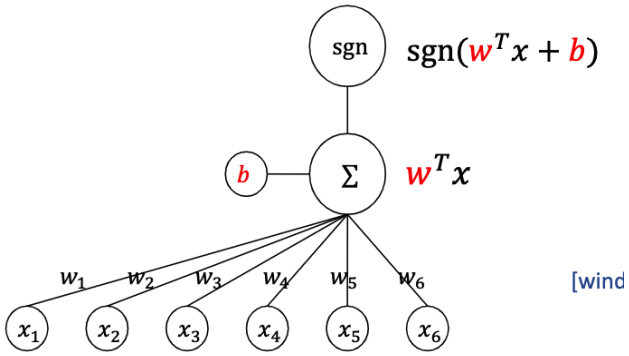
Definition Linear Models for Binary Classification

Given training set $D = \{(x, y), x \in \mathbb{R}^d, y \in \{-1, 1\}$, we can learn a hypothesis function $h \in H$.

$$H = \{h|h(x) = sgn(w^T x + b)\}$$

where y = h(x) and sgn(z) outputs 1 if z >= 0 and -1 otherwise.

This can be modeled with the following picture:



We can also add the bias term to the weight vector and a single 1 term at the end of x to reduce complexity of notation.

See slides for an example of converting a decision tree to a linear model.

4.1 Perceptrons

To learn is to find the best parameters. In this case, w and b. There are several algorithms to do so such as the perceptron, logistic regression, linear support vector machines, and more. Different methods will define the "best parameters" in a different way. First we will cover perceptrons.

The main idea behind the perceptron algorithm is to learn from mistakes. Concretely, it is the following:

- For (x, y) in D:
- $\hat{y} = sgn(w^T x)$
- if $\hat{y} \neq y, w \leftarrow w + yx$

Essentially, for a misprediction we are adjusting the decision boundary in order correct our mistake. After enough mistakes, we will converge (if possible).

We can expand on this idea by adding **epochs**, or more iterations of the algorithm being performed on our data. We continue these epochs until the hyperplane converges. This is one hyperparameter. We can also add η as a scaling factor to our adjustment term yx (i.e. $w \leftarrow w + \eta yx$). This is another hyperparameter.

It is important to not that a linear model **cannot** represent all functions. In fact, it can only separate linarly separable functions. As a consequence, functions like XOR cannot be learned. However, if a data is linearly separable, the perceptron algorithm will always converge (Convergence Theorem). The converge rate depends on the difficulty of the problem. If it is not, the algorithm will enter an infinite loop.

To quantify difficulty, lets dicuss margin.

Definition Margin

If a hyperplane can separate the data, the margin of a hyperplane for a dataset is the distance between the hyperplane and the data point nearest to it.

The margin of a dataset (γ) is the maximum margin possible for that dataset using any weight vector.

We can quantify the difficulty with $\frac{R}{\gamma}$ where $\forall i, ||x_i|| \leq R$ (x_i is a feature vector). The **Mistake Bound Theorem** states that the perceptron algorithm will make at most $(\frac{R}{\gamma})^2$ mistakes on the training sequence. We can see that γ and the number of mistakes are inversely proportional, meaning that a larger margin indicates an easier problem. Intuitively, this makes sense as the decision boundary does not need to be tuned as tightly if the margin is high, resulting in fewer mistakes being needed to tune the parameters. Note that the number of mistakes is not the same as the number of data points seen.

Perceptrons are good when the data is linearly separable. Unfortunately, in the real world this is very often not the case. We will see ways to deal with this.

4.2 Logistic Regression

As discussed, if the data is not linearly separable the perceptron algorithm will not converge. Another approach is to "model the probability" with logistic regression. Here the idea is to have discrete labels y, but to predict $P(y = 1|x)$ rather than the label itself. In other words, we want to build a model h(x) where

$$h(x) = \sigma(w^T x + b) \approx P(y = 1|x)$$

and

$$\sigma(x) = \frac{exp(x)}{1 + exp(x)} = \frac{1}{1 + exp(-x)}$$

Why Sigmoid?

Probability by definition is a number between 0 and 1. We use the sigmoid function because it has a domain over $\mathbb{R}$ and a range between 0 and 1. That is, it maps all real numbers to (what looks like) a probability. Intuitively, our decision boundary will be where $\sigma(x) = \frac{1}{2}$ because the value is equally likely to be classifed either way (Note that right now we are discussing binary classification).

Training Logistic Regression

We want to find the best w and b such that the predicted probabilities most closely align with their true labels.

Definition Maximum Likelihood Estimator

Let $X_1, ..., X_N$ be independent and identically distributed with PDF $p(x|\theta)$. The likelihood function $L(\theta)$ is defined as

$$L(\theta) = p(X_1, ..., X_N|\theta) = \prod_{i=1}^{N} p(X_i|\theta)$$

The maximum likelihood estimator $\hat{\theta}$ is the value of θ such that $L(\theta)$ is maximized.

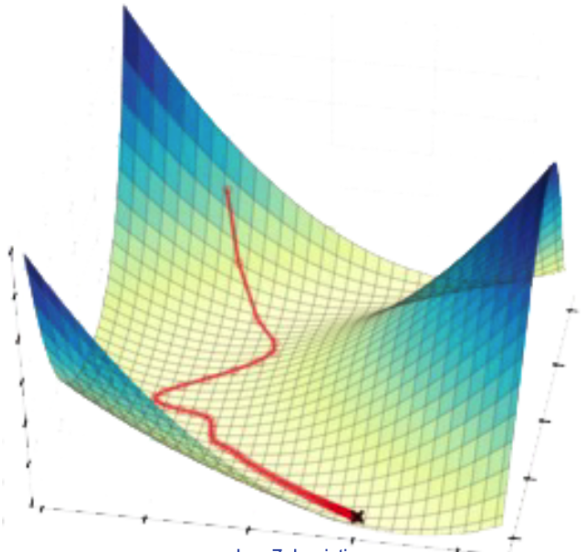
Concretely, the MLE is

$$\text{argmax}_{w,b} \sum_{i=1}^{m} log(\sigma(y_i(w^T x_i + b)))$$

(after some algebraic manipulation).

We can train our model using the above idea. Essentially, we will find w and b by maximizing P(S | w, b). There is no closed form solution for this optimazation. Instead, we must apply another strategy. We will look at gradient descent.

4.3 Gradient Descent



We can solve the logistic regression optimzation and determine the maximum likelihood estimator using gradient descent. The intuition behind gradient descent is that we are looking for the best direction to reach a location. In this case, the best direction is given by the gradient and the location is the global minimum of our function. Concretely, the steps are as follows:

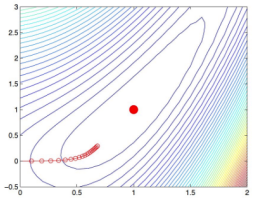
1. start at a random point
2. determine a descent direction
3. choose a step size
4. update
5. repeat until stopping criterion is satisfied

Algorithm 1 Gradient Descent (J)

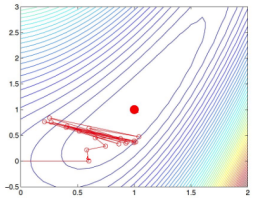
1: $t \leftarrow 0$
2: Initialize $\theta^{(0)}$
3: **repeat**
4: $\quad \boldsymbol{\theta}^{(t+1)} \leftarrow \boldsymbol{\theta}^{(t)} - \eta \nabla J(\boldsymbol{\theta}^{(t)})$
5: $\quad t \leftarrow t + 1$
6: **until** convergence
7: Return final value of $\boldsymbol{\theta}$

Choosing the right step size η is important. If it is too small it will take too long to reach a minimum, and if it is too large we will rubberband back and forth repeatedly.

small η is too slow?



large η is too unstable?



We can apply gradient descent in order to minimize our log likelihood function (AKA our loss function) in logistic regression. We define

$$L(w, b) = \arg\min \sum_{i=1}^{m} log(1 + exp(-y_i(w^T x_i + b)))$$

L is our loss function. Recall that it comes from our model $h(x) = \sigma(w^T x + b) \approx P(y = 1|x)$ where $\sigma = \frac{1}{1+exp(-z)}$.

We can take its gradient via the following series of steps.

$$\begin{aligned} \nabla L(w, b) &= \sum_{i=1}^{m} \nabla log(1 + exp(-y_i(w^T x_i + b))) \\ &= \sum_{i=1}^{m} \nabla log(\frac{1}{\sigma(y_i(w^T x_i + b)}) \end{aligned}$$

We take the gradient of the reciprocal sigmoid function without yet applying the chain rule.

$$\begin{aligned} \nabla log \frac{1}{\sigma(z)} = \nabla log(1 + exp(-z)) &= \frac{exp(-z)}{1 + exp(-z)} \\ = -\frac{1 + exp(-z) - 1}{1 + exp(-z)} = -1 + \frac{1}{1 + exp(-z)} \\ &= \sigma(z) - 1 \end{aligned}$$

Now to apply chain rule.

$$\nabla_w L(w,b) = \sum_{i=1}^{m} \nabla_w log \frac{1}{\sigma(y_i(w^T x_i + b))}$$
$$= \sum_{i=1}^{m} (\sigma(y_i(w^T x_i + b)) - 1) y_i x_i$$

$$\nabla_b L(w,b) = \sum_{i=1}^{m} (\sigma(y_i(w^T x_i + b)) - 1) y_i$$

This completes the derivation of our partial gradients. To apply them to our logistic regression, we use the following algorithm:

Gradient Descent

- initialize w
- For every epoch
  - compute $\nabla_w L(w,b)$ and $\nabla_b L(w,b)$
  - $w \leftarrow w - \eta \nabla_w L(w,b)$
  - $b \leftarrow b - \eta \nabla_b L(w,b)$
- return w and b

This algorithm can get quite expensive. Consider a large data set. We need to compute $\sigma(y_i(w^T x_i + b)) - 1$ for every data point (x_i, y_i). In addition, gradient descent typically requires many iterations to converge. This adds up to many operations that need to be performed. Can we do better?

Stochastic Gradient Descent

Rather than compute the gradient at all points in our training set to do a single update, we use a single point in order to perform an update. These changes accumulate to approximate the overall gradient. So, in effect, the only difference is how frequently we update our parameters.

Stochastic Gradient Descent

- initialize w
- For every epoch
  - sample a data point (x_i, y_i) from S
  - compute $\nabla_w L_i(w,b)$ and $\nabla_b L_i(w,b)$
  - $w \leftarrow w - \eta \nabla_w L_i(w,b)$
  - $b \leftarrow b - \eta \nabla_b L_i(w,b)$
- return w and b

Linear Regression

Linear regression seeks to approximate a real value output y based on input x. It is essentially drawing a line of best fit for the data. We use Least Mean Squares Regression (LMS) to compute it.

$$\text{argmin}_{w,b} \frac{1}{2} \sum_{i=1}^{m} (y_i - (w^T x_i + b))^2$$

As the name suggests, we are finding the parameters that minimize the squares of the difference of the labels and the predicted values. This can be done with stochastic gradient descent.

4.4 Evaluation Metrics

Sometimes accuracy and error rates can be misleading. Consider a data set that is heavily biased towards one of the labels i.e. the majority of the labels are classified negative. A lazy majority classifier could outperform a classifier that received a substantial amount of training! Obviously, this is not ideal and we should find a better evaluation metric that takes into account how balanced a dataset is.

A useful tool is the confusion matrix. Consider the following dataset.

True label	-	-	-	-	-	+	-	-	-	-	-	+	-	-	+	-	-
Predicted label	-	-	-	-	-	+	+	-	-	-	-	-	+	-	+	-	-

The confusion matrix is the following:

	True Label Positive	True Label Negative
Predicted Label Positive	2 (True Positive)	2 (False Positive)
Predicted Label Negative	1 (False Negative)	16 (True Negative)

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FN + FP} = \frac{18}{21} = .86$$
$$\text{Precision} = \frac{TP}{TP + FP} = \frac{2}{4} = .5$$
$$\text{Recall} = \frac{TP}{TP + FN} = \frac{2}{3} \approx .67$$

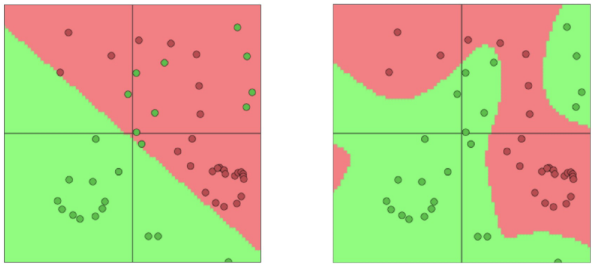
We can see that both the precision and the recall are much lower than the accuracy due the imbalanced dataset.

We can define a new metric, the F1 score, which is just the harmonic mean of the precision and recall.

$$\frac{1}{F_1} = (\frac{1}{P} + \frac{1}{R})/2 = \frac{2TP}{2TP + FP + FN}$$

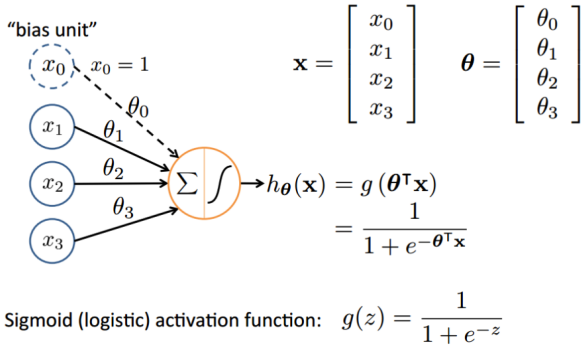
5 | Non-Linear Classifiers

What if is impossible to create a good linear separation of the data? In this case, we may want to look into models that allow for non-linear decision bounaries.



5.1 Neural Networks

Designed to mimic the brain, neural networks are capable of producing non-linear decision boundary models. Each neural network, is made up of many **nodes** connected by **links**. Each link as an associated weight and activation level. Every node has an input function (typically just a weighted sum of inputs), an activation function, and an output. The following diagram models a single node:



There are many possible activation functions including:

- sigmoid $\sigma(x) = \frac{1}{1+e^-x}$
- hyperbolic tangent $tanh(x) = \frac{e6x-e^-x}{e^x+e^-x}$
- step function
- ReLU, which returns 0 if x $\leq$ 0 or x if x $>$ 0

The steps for a "feed forward" neural network are the following (where x is input layer and $\theta^{(i)}$ is the weight matrix for layer i):

- $z^{(2)} = \theta^{(1)} x$
- $a^{(2)} = g(z^{(2)})$
- concatenate 1 to start of $a^{(2)}$ (to account for bias)
- and repeat for remaining layers

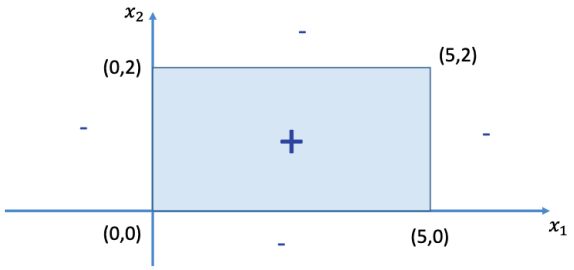
Example

let $\theta^{(1)} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & -1 & 1 & 0 \\ 2 & 0 & 1 & 1 \end{bmatrix}$, $\theta^{(2)} = \begin{bmatrix} 0 & 1 & 1 & 0 \end{bmatrix}$, step function as activation function, and $x = \begin{bmatrix} 1 \\ 0 \\ 2 \\ 1 \end{bmatrix}$

What is the output with x as input?

$$z^{(2)} = \theta^{(1)} x = \begin{bmatrix} 2 \\ 2 \\ 5 \end{bmatrix} \implies a^{(2)} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \implies z^{(3)} = 2 \implies \text{output} = 1.$$

Example



We want to classify all points within the rectangle as positive, and all others as negative. Show a feed forward neural network that accomplishes this.

First, we notice that the rectangle consists of four lines, which can be representedas the following set of inequalities

$$\begin{cases} x_1 > 0 \\ 5 - x_1 > 0 \\ x_2 > 0 \\ 2 - x_2 > 0 \end{cases}$$

From this we construct our first weight matrix (with first column as constant, second as x_1 coefficient, and third as x_2 coefficient)

$$\theta^{(1)} = \begin{bmatrix} 0 & 1 & 0 \\ 5 & -1 & 0 \\ 0 & 0 & 1 \\ 2 & 0 & -1 \end{bmatrix}$$

We want all values in $a^{(2)}$ to be 1 (indicating the point falls satisfies all the equations), so we the set the bias to be -3.5. Therefore,

$$\theta^{(2)} = \begin{bmatrix} -3.5 & 1 & 1 & 1 & 1 \end{bmatrix}$$

We can take this further and construct any arbitrary decision boundary by combining examples such as the one above in multiple layers.

5.2 Training Neural Networks

We can use stochastic gradient descent, except our loss function is changed to

$$J(\theta) = -\sum_{i}^{m} [y_i log h_\theta(x_i) + (1 - y_i) log(1 - h_\theta(x_i))]$$

How do we compute $\nabla J(\theta)$? We need to use an algorithm called back propogation.

Back Propogation

Backpropogation utilizes the chain rule to compute a gradient. It's important to understand back propogation in order debug neural networks when issues arise. For example, sometimes the gradient of a function might go to zero at postive or negative infinity (such as in the sigmoid function). Back propogation is more difficult in this case and the algorithm needs to be adjusted.

TODO: wtf was he talking about bro https://medium.com/@karpathy/yes-you-should-understand-backprop-e2f06eab496b

6 | Multiclass Classification

Multiclass classification occurs when we have k possible classes that our data can potentially fall into. In the basic case, each input should belong to exactly 1 of these classes (however this is not always the case, as seen in multilabel classification). There are two key ideas to solve multiclass classification: 1) we can train k binary classifiers and make the final decision based on the binary classifiers or 2) train a single classifier to consider all cases simultaneously.

6.1 Binary Decomposition

1. One Against All Strategy
   - decompose into k binary problems (i.e. is a classification satisifed or not)
   - learn k of the above models
   - in the ideal case, only the correct label will have a positive score from the classifier (however this is typically not the case)
   - how to decide which class to classify as if multiple have positive scores?
   - one way is to just pick the most positive (i.e. the max score class) "Winner takes all"
   - Analysis
     - not always possible to learn bc not all points are linearly separable from the others (i.e. if a class falls between two classes)
     - the output range of each classifer might not be the same leading to one classifier dominating the others despite data being closer to the decision boundary; this is because each of the classifiers are trained independently
     - easy to implement and works well in practice
2. One Against One Strategy
   - this also decomposes the problem in binary classification
   - while training we only consider two classes at a time, which are usually linearly separable; we need more classifers here (K choose 2)
   - for each class pair, positive examples will be of one of the labels while negative will be of the other label
   - this is more complex as each label gets k - 1 comparisons; the output of the binary classifer may not be coherent
   - we compare each one to one classifer and count the number of times each label wins; the majority is the overall winner

- sometimes there will be a tie; in this case we can look at the one to one classifer for the tied classes and return its output

Comparing these two:

1. One Against All
   - O(K) weight vectors to train and store
   - training set of the binary classifers may be unbalanced (most labels do not belong to a certain class)
   - less expressive and requires a strong assumption (data is linearly separable from all other classes)
2. One Against One
   - O(K^2) weight vectors to train and store
   - size of training set for a pair of labels could be small, leading to overfitting
   - need large space to store the model

Example

Consider we have a 10 class classification problem with 29 features, each class has 1000 examples.

1. How many parameters in total for linear models with one-vs-one? (10 choose 2) * (29 + 1)
2. How many parameters in total for linear models with one-against-all? (10) * (29 + 1)
3. How large is the training data for each one-vs-one classifier? 1000 per class -> 2000 for each classifier
4. How large is the training data for each one-against-all classifier? 1000 for positive class, 9000 for negative -> 10,000

Problems with Decomposition

- learning optimizes over local metrics rather than global; does not guarantee good global performance
- a poor decomposition leads to poor performance i.e. if the local problems are difficult or irrelevant

For these reasons, we might want to use **Multiclass Logistic Regression**.

6.2 Multiclass Logistic Regression

Recall that in binary logistic regression we model probability with seek to minimize the log likelihood

$$min_w(\frac{1}{2}w^T w + C\sum_i log(1 + e^{-y_i(w^T x_i)}))$$

Labels are generated with the following probability distribution:

$$P(y = 1|x, w) = \frac{1}{1 + e^{-w^T x}}$$

We will make the assumption that we can use the following in multiclass logistic regression (log-linear model)

$$P(y|x, w) = \frac{exp(w_y^T x)}{\sum_{y' \in \{1,2,\ldots,k\}} exp(w_{y'}^T x)}$$

This is also known as the softmax function. We can control the "peakedness" of the distribution using the temperature as shown below:

$$P(y|x,w) = \frac{s(y)/\sigma}{\sum_{y' \in \{1,2,\ldots,k\}} exp(s(y)/\sigma)}$$

We can train using maximum log likelihood estimation. So we end up trying to minimize the following:

$$min_w(\sum_i [log(\sum_{k \in \{1,2,\ldots,k\}} exp(w_k^T x_i)) - w_{y_i}^T x_i])$$

7 | Computational Learning Theory

We will discuss the difference between learning and memorizing and Probably Approximately Correct (PAC) Learning. For PAC we want to answer the question, how much training data do we need to get a good classifier (i.e. a classifier that has a high probability of having a low error on a test set). First we will discuss the monotone conjunction function class.

7.1 Learning Monotone Conjunctions

First off, a monotone conjunction is a class of functions in which the only "operation" being performed is logical and i.e. $f(x_1, x_2, x_3) = x_1 \wedge x_2 \wedge x_3$.

First off, we say a concept is learnable if you don't need to see all possible examples to make a good prediction. For example, you've probably never seen the arithmetic problem 1234 + 2332 before, but you can determine the answer is 3566. This makes arithmetic learnable. The important question is, how much training data do we need in order to train a good classifier (also known as our sample complexity)?

Definition PAC learnable

We say a function is PAC learnable if the number of examples we need to see is polynomial to the parameters defining the concept

Set up

- Instance Space: X, the set of examples
- Concept Space: C, the set of possible target functions where $f \in C$ is the hidden target function
- Hypothesis Space: H, the set of possible hypothesis mappings that the learning algorithm will explore
- Training Instances: S x {-1, 1}: positive and negative examples of the target concept drawn from distribution D (a subset of X)
- What we want: a hypothesis $h \in H$ such that h(x) = f(x)
- Assumption: training and test data are both drawn independently and identically distributed from X

Note that it's possible for the concept space and hypothesis space to not line up completely, meaning the target function might not always be found in the hypothesis space. For now, we will assume that C = H.

We can define a simple algorithm for learning monotone conjunctions.

Consider the following data formatted as <x, f(x)>:

<(1, 1, 1, 1, 1, 1, ..., 1, 1), 1>
<(1, 1, 1, 0, 0, 0, ..., 0, 0), 0>
<(1, 0, 1, 1, 1, 0, ..., 0, 1), 1>
<(1, 1, 1, 1, 1, 1, ..., 0, 1), 1>
<(1, 1, 1, 1, 1, 0, ..., 1, 1), 0>
<(1, 1, 1, 0, 0, 0, ..., 0, 1), 1>
<(1, 1, 1, 1, 1, 1, ..., 0, 1), 0>

How do we determine f? Well, we know that f is a monotone conjunction, so it only consists of and operations. As a result, a variable cannot be included in f if it is equal to 0 while the output is 1. We simply search all of our data to find these counter examples and remove all variables with this quality. For example, the function cannot include x_2 because the third row of data is a counterexample.

This algorithm is good, but it does not guarantee that we will find our exact target function f. There might be a counter example for a variable that we simply have not seen. It is useful to quantify how likely we are to see a counterexample when we have N examples in our training set.

Definition Error of hypothesis

Given a distribution D over examples, the error of a hypothesis h with respect to a target concept f is

$$err_D(h) = P_{x \sim D}[h(x) \neq f(x)]$$

7.2 PAC Learnability

Theorem

Suppose we are learning a monotone conjunctive concept with n dimensional boolean features using m training examples. if

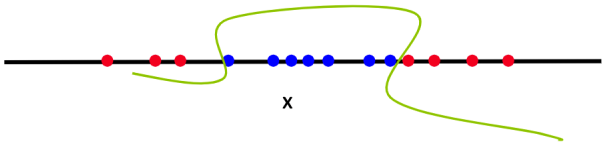
$$m > \frac{n}{\epsilon}(log(n) + log(\frac{1}{\delta}))$$

then with probability $> 1 - \delta$, the error of the learned hypothesis will be less than ϵ. The derivation for this is available in lecture 11 slides.

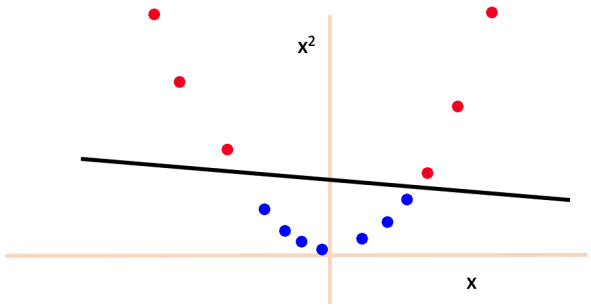
We say a concept class C is **PAC Learnable** by a learner L using hypothesis space H if the algorithm produces with probability (1 - δ) a hypothesis that has error at most ε where the m training samples is polynomial in $1/\varepsilon$, $1/\delta$, n and size(H) and is efficiently learnable in polynomial time. Essentially, if there is polynomial space complexity and polynomial time complexity, it is PAC Learnable.

8 | Kernel Methods

We saw in previous chapters that linear models such as perceptrons and logistic regression could not classify data that was not linearly separable with 100% training accuracy. However, it may be possible to create some mapping for non-linearly separable data that makes it linearly separable, and thus learnable for linear models. Consider the image below.



The data appears to not be separable linearly. However, if we introduce a new feature x^2, we can separate it as follows.



We can use this idea to modify our perceptron algorithm to allow it to converge when data is not linearly separable in 2 dimensions. Here, $\phi(x) = [x, x^2]^T$.

Modified Perceptron Algorithm

- Initialize $w \leftarrow 0 \in \mathbb{R}^{2n}$
- For (x, y) in D:
  - if $yw^T\phi(x) \leq 0$
    - $w \leftarrow w + y\phi(x)$
- return w

What if our mapping function is more complex i.e. mapping data to infinite dimensions? We can use the fact that the perceptron algorithm essentially just builds a linear combination throughout its run time to

create a more specialized version of the algorithm. We can consider $w = \sum_{i=1}^{m} \alpha_i y_i x_i$ where α_i is the number of mistakes the perceptron made on x_i.

Modified (Again) Perceptron Algorithm

- Initialize $a \leftarrow 0 \in \mathbb{R}^m$
- For (x_i, y_i) in D:
  - if $y_i \sum_j \alpha_j y_j \phi(x_i)^T \phi(x_j) \leq 0$
    - $\alpha_i \leftarrow \alpha_i + 1$
- return α

We can compute the dot product $\phi(x_i)^T \phi(x_j)$ efficiently even at high dimensions. In fact, it scales at $O(D)$ where D is the number of dimensions. The **Kernel Trick** allows us to save time/space by computing the value of our kernel function by performing operations in the original space (without a feature transformation!)

Definition Kernel function

A kernel function k satisifies

$$k(x_m, x_n) = k(x_n, x_m)$$
$$k(x_m, x_n) = \phi(x_m)^T \phi(x_n)$$

for some function ϕ.

For example, $(x_m^T x_n)^2$ is a kernel because it is the linear product of the following mapping

$$\phi : x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \rightarrow \phi(x) = \begin{pmatrix} x_1^2 \\ \sqrt{2} x_1 x_2 \\ x_2^2 \end{pmatrix}$$