CS M146:
Introduction to Machine Learning

Einar Balan

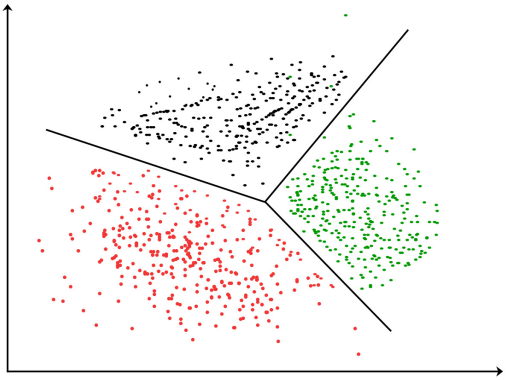
Contents

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

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

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

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]

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?

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.

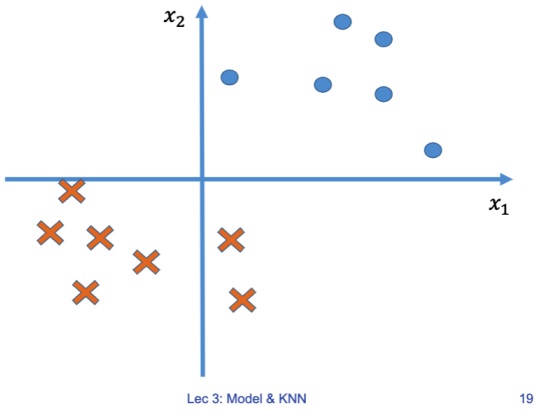
In general there are $|Y|^{|X|}$ possible functions from the instance space X to the label space Y. The hypothesis space is typically a subset of these. We can add rules to limit the hypothesis space, but we need to take care that these rules are not too restrictive, as it is possible that **no** simple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

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.

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)

Defintion 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)

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.

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

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.

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?

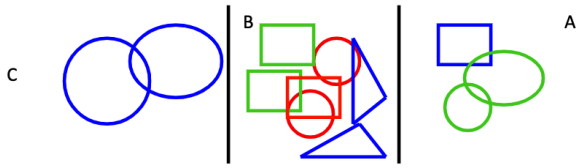
In practice, we apply dimensionality reduction. That is, we consolidate dimensions by removing irrelevant features and combining certain features.

Data Preprocessing

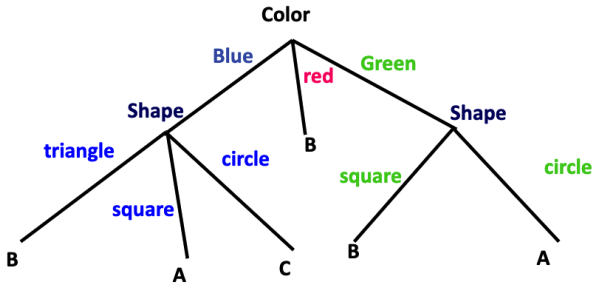
- normalize data to have zero mean unit std in each dimensionality
- scale each feature accordingly

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.

Decision Tree 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?

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.

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

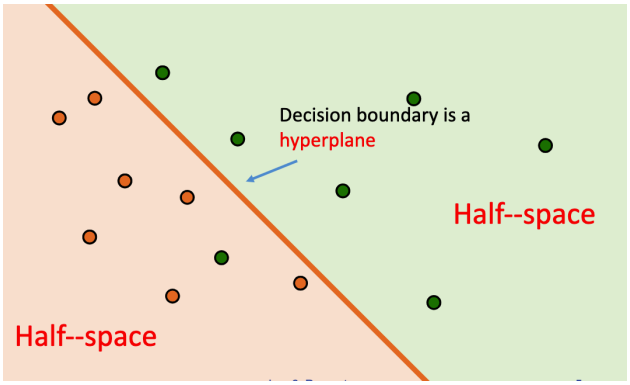
$$H[S] = -\sum_{v=1}^{K} P(S = a_v) log_2 P(S = a_v)$$

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

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

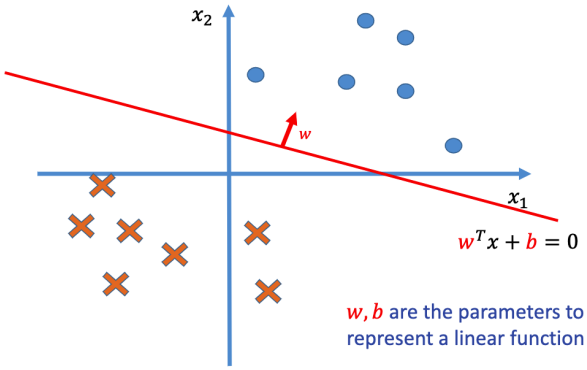
3 | Linear Models

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.

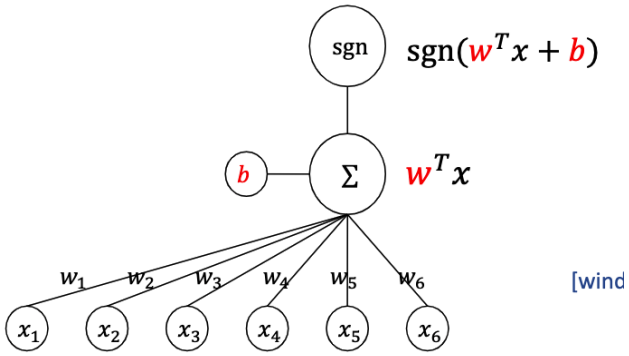
Defintion Linear Models for Binary Classification

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.

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.

Defintion 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. 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.