CS 188: Intro to Machine Learning notes

Lecture 1

- Machine Learning algorithms are algorithms that improve their performance P on a task T from an experience E.
- Applications: recognizing handwritten digits - a lot of variability as to what constitutes a "2" for example. Instead of thinking up all the different cases that could be a handwritten 2, teach a computer how to recognize a 2.
- Types of Learning:
    - **Supervised**: given training data with labels, predict labels for unseen instances.
    - **Unsupervised**: given unlabeled training data, find some intrinsic structure/pattern in the data.
    - **Reinforcement Learning**: Given an agent (algorithm) interacting in an environment (interacting with some set of states), learn a **policy** (state -> action mapping) that maximizes the agent's reward.

- Supervised Learning: Regression -
- Given a training set of pairs $(x_1, y_1), \ldots (x_n, y_n)$ learn a function *f* to predict y given x, where y is real valued.
- Classification: y is discrete/categorical. Learn a function f to predict categories.
- x is usually n-dimensional, each component of x refers to an attribute/feature.

- Unsupervised Learning:
    - Clustering data, grouping individuals by genetic similarity, independent component analysis (separate an audio from many speakers into their individual components)
- Reinforcement Learning:
    - Given a sequence of states and actions with (delayed) rewards, learn a policy that maximizes the agent's reward.

- Key Idea: Learning is about generalizing from the training data.
- What does this assume about the training & testing data? It assumes that they come from the same distribution/sample space.
- We care about the performance of the learning algorithm on test data (ability to generalize), so the training and testing data should be strongly related.

- Loss Functions:
    - $L(y, y_{pred})$ where $y_{pred} = f(x_1, x_2, \ldots x_n)$
    - Loss function for regression: $L(y, y_{pred}) = (y - y_{pred})^2$ (called the squared error)
    - Loss function for classification: $L(y, y_{pred}) = 0 \; if \; y = y_{pred} \; else \; 1$

- Cross-entropy loss: $-\sum_{i=1}^{N}\sum_{j=1}^{M} L_{ij} log(S_{ij})$ (*L* is the one-hot encoded labels, entropy is high if the labels are incorrect and low if they are not).

- Importantly, there exists some probability distribution over the features/labels space called the <u>data generating distribution</u>. The key is that this is almost always unknown. It is the distribution that indicates the likelihood of a given vector of *x*'s (features) mapping to a certain output (*y*).
- For example, in MNIST, there exists some probability distribution that indicates what set of inputs (pixels in this case) would indicate a 2, and a probability for those set of inputs to be a 3. There are essentially probabilities for any set of inputs that can exist, and these probabilities indicate what number they are.
- If we knew the distribution (ie, the probability distribution for malignant tumors), then machine learning would be very easy.

- A learning problem is defined by:
  - Loss function: to measure performance
  - samples from the data generating distribution: the experience we will provide our algorithm

- <u>Defining the Learning Problem:</u>
  - X = vectors of instances, Y = labels, f: X -> Y is an unknown function.
  - We have a set of hypothesis functions $H: \{h \mid h: X -> Y\}$ of infinite size.
  - Our inputs to the function are (feature, label) pairs and we want to output some *h* that "best approximates" *f*. *h* should be the function that does the best job at predicting *y* based on *new* instances of *X.*
  - Formally, h should have <u>low expected loss.</u>
- We want to find $h(x)$ such that $\Xi_{(x,y)}[L(y, h(x))] = \sum_{(x,y)} p(x, y)L(y, h(x))$ is small. This means that the weighted average loss is small. We weight our loss by multiplying by p(x, y), which is large if an (x, y) pair is likely to occur, and small if an (x, y) pair is unlikely to occur.
- Since we can't minimize this since we don't know the probability distribution, we instead approximate the expected loss with the <u>training loss:</u>
  - $\frac{1}{N}\sum_{i=1}^{N} L(y_i, h(x_i)$
  - Problem: we can make this training error very small, but this is on the training set, and then our hypothesis function only performs well on training data and not testing data. <u>We are most interested in the prediction function's performance on an unseen testing set set.</u>
- Say our hypothesis space is degree *m* polynomials, and we wish to find a polynomial *h* based on some training data that we are given. As the degree of our polynomial increases, our model gets more and more complex and fits the seen data better.

- But then, the risk of **overfitting** increases: our function performs badly on unseen data, and does not generalize well. It has low bias, but high variance. (This means that if we train several different functions from different samples from the data generating distribution, our function will be able to predict the testing data with great accuracy, but the several functions will differ greatly from each other).

- Fundamental difficulty of ML:
    - We have access to the training loss, but we really care about the test (expected loss).
- Key problems for Machine Learning:
    - **Representation:** how do we choose a hypothesis space of functions? (IE, deep neural nets, decision trees, regression)?
    - **Optimization:** how do we find the best hypothesis? Algorithms/CS/Math/Computational problem.
    - **Evaluation:** How can we gauge the accuracy of our hypothesis function? A statistical problem.
    - **Formulation:** How can we formulate a problem as an application of machine learning? The engineering problem.
    - Pipeline: Formulate -> Represent -> Optimize -> Evaluate
- Machine Learning in Practice:
    - While unsatisfied:
        - Understand/formulate the problem.
        - Select/preprocess data
        - Build + learn model
        - Evaluate the model/interpret the results.

Lecture 2: Probability & Statistics

- A sample space, denoted by Ω, denotes all possible outcomes of an event. Events are a subset of the sample space.
- Example: Ω = {HH, HT, TH, TT} when flipping two coins. Then $A \subseteq \Omega$ denotes a subset of possible events, ie, A = {HH}, A = {HH, TH, TT}.
- Let a random variable X be the number of heads that we observe. Then, $X \in \{0, 1, 2, 3, 4\}$. Namely, $P(X = 0) = 1/4, P(X = 1) = 1/2, ...$

- Three axioms of probability:
  - $P(A) \geq 0 \ \forall \ A \subseteq \Omega$
  - $P(\Omega) = 1$
  - $\forall \ A_1, A_2 \subseteq \Omega \mid A_1 \cap A_2 = 0, P(A_1 \cup A_2) = P(A_1) + P(A_2)$
  - In english, if there are two disjoint subsets of events, the probability of either one of them happening is the sum of the individual probabilities.

- Corollaries of 3 axioms:
  - $\forall \ A_1, A_2 \subseteq \Omega$, $P(A_1 \cup A_2) = P(A_1) + P(A_2) - P(A_1 \cap A_2)$
  - $P(A^C) = P(\Omega - A) = 1 - P(A)$ (if $P(A^C) = 0,$ then A will almost surely happen.

- Discrete Random Variables
  - If X is a random variable that takes on M possible values, ie, $X \in \{x_1, \ldots x_m\}$ then the **probability mass function** is given by $P_X(x_m) = P(X = x_m)$
  - Thinks of this as a table: one column is all m values for x, and the other column is the probabilities for the random variable x having those possible values. They should sum to 1.
  - Using the above axioms and corollaries, we can derive a few important probability rules for what happens when there are two events.
  - let X be an event with *m* possibilities ($X = \{x_1, x_2, \ldots x_m\}$) and Y be an event with *m* possibilities as well. Then the following grid can be formed:

	y_1	y_2	y_m
x_1	$p(x_1, y_1)$	$p(x_1, y_2)$	$p(x_1, y_m)$
x_2	$p(x_2, y_1)$	$p(x_2, y_2)$	$p(x_2, y_m)$
x_m	$p(x_m, y_1)$	$p(x_m, y_2)$	$p(x_3, y_m)$

    - $p(x_1, y_1)$ = probability of x1 and y1.
  - We can denote n_{ij} to be the number of times we see X having value x_i and Y having value y_j. Moreover, we can denote c_i to be the number of times we see X having value x_i (regardless of the value of Y), and we can denote c_j to be the number of times we see Y having value y_j (regardless of the value for X).
  - If we let N be the total number of trials (tending to infinity), then we can be

interested in the **joint probability** of two events happening at once, namely, $P(x_i, y_j)$, which is clearly n_{ij}/N.

- **Marginal Probability:** for example, the probability of $P(X = x_i)$, which is c_i. This can be thought of as a marginal probability because it is summing along the margin of x_i, for all possible *m* values that *y* could take on. So, we have $c_i = \sum_{j=1}^{M} n_{ij}$, so the marginal probability $c_i/N = \sum_{j=1}^{M} n_{ij}/N$.
- This can also be written as $P(x_i) = \sum_{j=1}^{M} P(x_i, y_j)$. This is known as the **sum rule.**
- **Conditional Probability:** the probability of an event given some other event has occurred. Ex: $P(Y = y_j \mid X = x_i) = n_{ij}/c_i = \frac{n_{ij}/N}{c_i/N} = \frac{P(X = x_i, Y = y_j)}{P(X = x_i)}$
- This leads us to the **product rule:** $P(X = x_i, Y = y_j) = P(Y = y_j \mid X = x_i) * P(X = x_i)$**,** or, equivalently, $P(X = x_i, Y = y_j) = P(X = x_i \mid Y = y_j) * P(Y = y_j)$
- This is commonly abbreviated to $P(x, y) = P(y \mid x) * P(x)$.
- But since we have $P(x, y) = P(y \mid x) * P(x)$ as well as $P(x, y) = P(x \mid y) * P(y)$, we know that $P(y \mid x) * P(x) = P(x \mid y) * P(y)$. From this we obtain:
    - **Bayes Theorem:** $P(x \mid y) = \frac{P(y \mid x) * P(x)}{P(y)}$. This is how a simple/naive spam classifier can be made: calculate the probability of a message being spam given some words based on the probability of some words existing in an email, given that the email is a spam message.
- This leads to the idea of <u>independence.</u> Two events X, Y are independent iff one of the events happening has no influence on the other event happening. That is, $P(y \mid x) = P(y)$ and $P(x \mid y) = P(x)$. From the product rule, we have $P(x, y) = P(x \mid y) P(y)$, and iff x and y are independent, then $P(x, y) = P(x) * P(y)$. When two events are independent, their joint distribution is equal to the product of their marginal distributions.

- **Common Distributions:**
    - **Normal/Gaussian** probability density function: $p(x, \sigma, \mu) = \frac{1}{\sqrt{2\pi\sigma^2}} exp(\frac{-1}{2\sigma^2}(x - \mu)^2)$. E[X] = μ and Var(x) = σ^2.
    - **Bernoulli:** $X \subset \{0, 1\}$, defines $P(X = 0)$, $P(X = 1)$. If $p = P(X = 1)$ then $1 - p = P(X = 0)$. Therefore, the distribution can be defined by a single parameter *p*, and probability mass function is defined by $P(x, p) = p \; if \; x = 1 \; else \; 1 - p$.
    - Mean: p, Variance: p(1-p)
    - **Poisson PDF**: use when a successful outcome is generally rare, and it doesn't make much sense to ask about the opposite of a successful outcome.
    - Poisson = binomial without the "didn't happen", but with the "less and less likely to happen"
    - The rarer the success of the event becomes, the more it resembles a binomial distribution.
    - Two parameters: x - the actual number of successes, lambda: the expected number of successes (mean):

- $P(x, \lambda) = e^{-\lambda}\lambda^x/x!$
- Mean = variance = lambda.
- **Uniform:** unlike normal, all probabilities are equal: p(x, a, b) = 1/b-a for a<=x<=b. Expectation: (a + b)/2, variance: (b-a)^2 /12
- **Binomial:** $P(x, n, p) = C(n, x)p^x(1-p)^{n-x}$. Here, we have $E[x] = np$ and $Var[x] = np(1-p)$. If n = 1, we get the bernoulli distribution.
- **Exponential:** $P(x, \lambda) = \lambda e^{-\lambda x}$. $E[x] = 1/\lambda$, $Var[x] = 1/\lambda^2$.

- Continuous random variables:
  - The random variable X is now real-valued and takes on an infinite number of states, rather than a discrete *M* states.
  - For $x \subset \Re$, $P(X \subset [x, x + dx]) = p(x)\,dx$
  - $P(x \subset [a, b]) = \int_a^b p(x)dx$
  - **CDF -** cumulative density function: $F(x) = P(X \leq x) = \int_{-\inf}^{x} p(x)dx$ implying that $p(x) = \frac{dF(x)}{dx}$.
- Computing Expectation & Variance:
  - Let x ~ p(x) take on *m* discrete values. Then the expectation of X is given by $E[X] = \sum_{i=1}^{M} x_i\, p(x_i)$. Intuitively, this calculates the weighted mean of the distribution, the value that we would "expect" having seen many values, each at different probabilities of occurring. Moreover, the variance of X is given by $V[x] = \sum_{i=1}^{m} (x_i - E[X])^2 p(x_i)$. Intuitively, this calculates the "spread" of the distribution by summing up the weighted average of how far each point is.

- Quick note on non-parametric vs parametric models (from wikipedia):
  - Nonparametric statistical models make no assumption about the probability distribution that the training data are being drawn from.
  - Parametric models have a **fixed** number of parameters, while nonparametric models have a **variable** number of parameters, which are determined by (and grow based on) the amount of training data.
  - Clearly, neural networks are nonparametric function estimators:
    - They make no assumption about the underlying probability distribution from which they are being drawn from
    - The number of parameters increases along with the amount of features that each training example has.
- Things to keep in mind:
  - A machine learning **model** is a mathematical function with a certain number of parameters that are learned from the data. The crux of machine learning is fitting

a model to our training data, through learning certain parameters.

- **Hyperparameters** refer to higher level “knobs” that one can “tune” regarding qualities of the model. They cannot be learned from the training data, and are usually set before the training even begins. However, they can be changed while learning (such as an automatic program to decay the learning rate).

Lecture 3: Statistics

- Probability vs Statistics Example:
  - Recall the Bernoulli Distribution: $x \sim Ber(p),\ x \in \{0, 1\}$ meaning X is a random variable which follows a Bernoulli Distribution.
  - $p = P(x = 1),\ 1 - p = P(x = 0)$
  - Params -> probability -> obtain samples/data
  - Samples/data -> statistics -> obtain parameters (for the underlying distribution that the data came from)
  - If theta parametrizes a probability distribution and we draw X from it, we say X ~ P(X; theta).
- Example:
  - Toss a coin 5 times, observe 3 heads. Find p where p = P(X = 1).
  - **Likelihood:** A function of the parameters of a statistical model. While this is not

completely correct, it can be thought of as giving the probability of the sample data for a fixed value of the parameter.

- Consider 5 random variables: $x_1, \ldots x_5$ = 1,0 1, 1, 0. The likelihood is: $L(\theta) = P(x_1, \ldots x_n; \theta)$, ie, the probability of the sample data given the parameters. The likelihood is the probability of observing a certain sample given a parameter vector theta.
- We assume that the samples are drawn from a Bernoulli distribution, ie, $x_i \sim Ber(p)$ and we want to find the parameter *p*.
- If we make the assumption that the five variables are independent, then we have:
- $L(\theta) = P(x_1; \theta) \ldots P(x_n; \theta) = \prod_{i=1}^{n} P(x_i; \theta)$. Since we assumed a Bernoulli distribution, we know that $P(x_i = 1) = p, P(x_i = 0) = 1 - p$. Thus, $L(\theta) = \theta^3(1 - \theta)^2$. $\theta_{max,\, hat} = 3/5$.
- **Estimators** - essentially a rule for estimating the given parameter. They are approximations of the true parameter(s) of the distribution that we obtain from our sample data. If we search for the theta that maximizes our likelihood function, then this is called the **maximum likelihood estimator.**
- Machine learning algorithms, such as algorithms which maximize the likelihood, have many "knobs" (aka hyperparameters) that are to be tuned using statistical inference.
- <u>Models:</u> Probabilistic/statistical models, meaning some probability distribution your data is sampled from, dependent on some parameters for the distribution.
- Data/observations: $D = \{x_1, \ldots x_n\}$ and $x \sim P(x_i; \theta)$ are data drawn (presumably independently) from the model.
- 2 steps to modelling data: specifying probability distributions, specifying parameters for those distributions
- Likelihood: $L(\theta) = P(x_i, \ldots x_n; \theta)$. A higher likelihood implies that we are more likely to see the data for the given value of the parameter.
- If we assume independence, one obtains $L(\theta) = \prod_{i=1}^{N} P(x_i; \theta)$.
- If we are looking for the maximum likelihood estimator, we pick the theta that maximizes the above function and obtain $argmax(L(\theta)) = \hat{\theta}$.
- Many times, an ML model with start with some probability distribution, state its assumptions, and then maximize the likelihood.
- Example:
    - Consider the Normal distribution: $P(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} exp(\frac{-1}{2\sigma^2}(x - \mu)^2)$
    - Assumption: n data points are drawn from the normal distribution randomly independently.
    - We don't know the parameters of the distribution, and therefore wish to find estimators for them: $\hat{\theta} = [\hat{\mu}, \hat{\sigma}^2]$
    - Therefore, $L(\theta) = P(x_1, \ldots x_n \,'\, \theta) = \prod_{i=1}^{N} P(x_i; \theta)$
    - We can define the log-likelihood to be $log(L(\theta)) = log \prod_{i=1}^{N} P(x_i; \theta) = \prod_{i=1}^{N} log\,(P(x_i; \theta)) = \sum_{i=1}^{N} log(p(x_i; \theta))$.
    - Now, to find the maximum likelihood estimators, we need to find sigma

and mu such that the likelihood is maximized. We need to take the partial derivatives $\frac{dl}{d\mu}$ and $\frac{dl}{d\sigma}$ and set them equal to zero. One obtains:

- $\frac{dl}{d\mu} = \sum_{i=1}^{N} \frac{x_i - \mu}{N} = 0$ then $\mu = \frac{\sum_{i=1}^{N} x_i}{N} = x_{bar}$
- $\frac{dl}{d\sigma^2} = 0$ then $\sigma^2 = \sum_{i=1}^{N} \frac{(x_i - \mu)^2}{N}$. These are what we would intuitively expect to obtain for estimates of the population mean and population standard deviations: they are exactly the sample mean and sample standard deviations.
- Maximum likelihood estimation is a method of estimating the parameters of a statistical model given some set of observations, through finding the parameters that maximize the likelihood of making the observation given those parameters.

Lecture 4: Decision Trees

- Sample dataset: cols denote features/attributes & labels/targets. Rows denote (instance, label) pairs (x_n, y_n) where x_n is a vector and y_n is discrete valued. Class labels denote whether the tennis game is played. partial dataset:
-

Outlook	Temp	Wind	Response
sunny	warm	low	Y
rainy	cold	high	N

- In decision trees, each internal node tests one feature. Each edge selects a value for the feature, and when classifying a new example we travel the edge that corresponds to the value of the feature. Each leaf node is a class prediction.
- Many decisions are tree structures, such as medical treatments and salary in a company.

- Key Idea: A tree partitions the feature space. It classifies a sample based on the value of its features. At each node, it makes a decision about which edge will be travelled based on the value of the corresponding feature indicated by the node.
- Decision tree learning problem:
    - Set of possible instances X, where each $x \in X$ is a feature vector.
    - Set of possible values Y where each y is discrete valued.
    - Unknown target function $f: X \rightarrow Y$
    - Model/hypothesis space: $H = \{h \mid h: X \rightarrow Y\}$
    - Each hypothesis is a decision tree. We have to learn the optimal h that maps instances to labels.
    - Learning a tree model:
    - 3 things to learn: 1) struct of tree (what features are at each node, 2) threshold values of features (the edges), and 3) the value of leaves (which classes are predicted at which leaf)
    - Algorithm: Example: choosing a restaurant to eat. N labels with m features, target is T or F denoting whether we will wait to eat at the restaurant.
    - How do we choose the best tree to use? From philosophy, one has Ockham's Razor: the simplest and most consistent explanation for something is the one that we prefer. In this case, we want the simplest (and therefore smallest) tree model.
    - Our problem is that we have to find the smallest decision tree that correctly classifies training examples. This is a problem that is in NP hard. It's computationally difficult to enumerate all trees to find the tree with the smallest size that still classifies well.
    - Instead, we use a greedy algorithm that is based on a heuristic to construct a "small enough" (but likely not THE smallest) decision tree that still classifies well.
    -
    - Algorithm:

```
DecisionTreeTrain(data, features){
      guess ← most frequent label in data
      if all labels in data = guess or features is empty:
            return LEAF(guess)
      f = best feature in features
      NO = all d in data such that f = NO
      YES = all d' in data such that f = YES
      left = DecisionTreeTrain(d, features - {f})
      right = DecisionTreeTrain(d', features - {f})
      return NODE(f, left, right)
}
```

    - This builds a subtree recursively:
        - base case: all labels in data are the same.

    - Otherwise, pick the “best” feature in f (using some heuristic). This f will split the data.
    - train left subtree on data that has f, on a feature set given by features - {f}
    - train right subtree on data that does not have f, on a feature set given by featuers - {f}
- How do we choose the best feature in features at each point?
- Possibilities: random, highest accurayc, **max-gain** - select feature w/largest information gain (greedy approach).
- Use the information we gain to figure out which attribute to split.
- Idea: less uncertainty implies more information gain. **Entropy** is a measure for uncertainty. If a random variable X has k different values, the entropy is given by:
    - $H[X] = -\sum_{i=1}^{K} P(X = a_i) log\, P(X = a_i)$. This is the entropy for a random variable with k possible values.
    - Example - weighted average if we go with no patrons:
    - H[X] =
    - $-(P(X = YES) log\, P(X = YES) + P(X = NO) log\, P(X = NO)) = -(\frac{0}{2} log \frac{0}{2} + \frac{2}{2} log \frac{2}{2}) = 0$
    - for some branch:
    - $-(P(X = yes) log P(X = yes) + P(X = no) log P(x = no)) = -(1 log 1 + 0) = 0$
    - for full branch:
        - entropy is 0.9
    - The weighted average of these entropies is called the **conditional entropy**.

      $H[Y|X = Patron] = \sum_{i=1}^{\#\ of\ values\ patron\ can\ take\ on} H[Y|X = a_i]]\, p(X = a_i) = \frac{1}{6}(0) + \frac{1}{3}(0) + \frac{1}{2}(.9)$

    - **Conditional entropy:**
        - Given 2 random variables X and Y, we define the conditional entropy as:
        - $H[Y|X] = \sum_{i=1}^{K} P(X = a_I)\, H[Y|X = a_i]$, essentially taking the probability of a specific value that X can take on, multiplying it with the entropy that would give, and summing across all possible values of those.
        - In the example, Y - vector of decisions (wait or not) and X = the attribute to be split.
        - Relation to information gain:
        - Gain = $H[Y] - H[Y|X]$
        - This is the expected reduction in entropy of the target variable, also called the **mutual information**.
- The **Gain** can be thought of as “what is the reduction for the uncertainty in Y, after getting information about X?”

- In the example, the entropy for patron was .45 so the gain was 1 - .45 = .55. For the type of restaurant, the entropy was 1, meaning that the gain was 1 - 1, so in the case of type, knowing that feature did not make us any more certain about what Y could be. Therefore, we generally want to greedily pick the feature that gives us the best gain at each step.
- Optimal tree depth: if the tree is too deep, we may overfit on the training set (essentially, our algorithm would memorize the training set and not generalize well). Likewise, if it is too shallow, we may underfit on the data (not learn enough).
- Max depth is a hyperparameter that should be tuned.
- **Overfitting:**
- **Reasons for overfitting:**
    - Noisy data: 2 instances have the same feature values but different class labels.
    - Some features/values are incorrect.
    - Some features are irrelevant to classification.
    - target variable is non deterministic in the features.
    - In general, we can't measure all of the features that we would like to make the most accurate prediction. SO the target is not uniquely determined by the input feature values.
    - training error is not gauranteed to be zero.
    - The learning precodeure can increase accuracy on the training set, but the test accuracy can decrease. This means that our training loss may get lower and lower, but at some point the model will start performing worse on the testing set. This indicates a classifier that is low bias and high variance.
- How to detect overfitting, or "how to get a realistic estimation for accuracy?"
- Training and test data split: train the model on training data, compute the acc on testing data.
- But instead of just one particular split of the data, we can actually choose K particular splits on the data.
- In principle, we can do this many times since the performance may be different for each split. (If we notice the performance is very different, this indicates a high variance model).
- K-fold cross-validation example:
- randomly partition the N points into K disjoint subsets (N/K).
- Choose one fold as a test set and train model on the others. Compute average statistics over K test performances.
- Leave-one-out CV: K = N. LEave one data point to test on, train on rest….
- Then take a summary of the statistics over the K trained models, and use these statistics for your predictions.
- Formally:
    - Split training data into K parts. Each part is D_k^{Train} $\forall\ k\ \in [1,\ K]$

- Train the model using the dataset $\{D^{Train} - D_k^{Train}\}$. Evaluate the model using the dataset D_k^{Train}. Take average of the K performance metrics. Intuitively, this makes sure that the model does not learn too much from one particular dataset, but learns from many different datasets. This will hopefully help it generalize better.
- Other ways to prevent overfitting: get more training data & remove irrelevant features.
- Decision tree pruning:
  - 1) prune while building tree (early stopping)
  - 2) prune after building tree (post pruning or something)
- We need an optimal tree depth in order to prevent overfitting. So we prune our large tree into a smaller one:
  - First, decide on a max depth for the tree. Then to make leaves that make label predictions there, simply take the majority vote.
  - IE, if max depth = 3 at one node, and at that node we've got 10 trues and 3 false, then instead of pruning the tree, we can take a majority vote and set the label to T since 10 > 3.

Lecture 5: Nearest Neighbor Classification/Hyperparameter Tuning

- General assumption of nearest neighbors: the label of an instance is similar to the label of nearby points.
- Multi-class classification: Classify data to one of the multiple categories. Instance: feature vectors $x \in R^D$, label $y \in C = \{1, 2, ...C\}$.
- Learn h: x->y. If binary, labels are -1, 1 or 0,1.
- Terminology: training data are a set of instance, label pairs D^{Train} that are used for learning. Test data: D^{test}, used for assessing how well h will do.
- Training step is to just store all the training data.
- Testing: $x(1) = x_{nn(x)}$ where $nn(x) \in \{1, 2, ... N\}$ is the index to one of the training instances. $argmin \| x - x_n \|^2 = argmin \sum_{d=1}^{D} (x_d - x_{n,d})^2$.
- Classification rule: $y = h(x) = y_{nn(x)}$. Basically, we find the index of the vector that's closest (by some distance metric) to our testing vector. Then we take that index and return the vector.
- Other distances besides Euclidean: L1-norm, LN norm.

- Other types of features:
    - if binary, just map features to 0 or 1. Otherwise, for categorical features with V values, use one hot encoding.
- K-nearest neighbors:
    - Generally, it is better to use K > 1 nearest neighbors in order to classify, as opposed to only the closest one. This can be defined as:
    - $nn_1(x) = argmin \|x - x_n\|^2$, $nn_2(x) = argmin\|x - x_n\|^2 \ \forall n \in N - \{nn_1(x)\}$, etc
    - To classify with the k nearest neighbors, use a voting algorithm. Each neighbor gets one vote and we take the label with the majority of votes. Mathematically, we have the indicator function:
    - $I(y_n == c)$ represents the voting.
    - $V_c = \sum_{n \in knn(x)} I(y_n == c) \ \forall \ c \in [C]$. Label w/majority: $y = h(x) = argmax\, v_c$.
    - When K increases, the decision boundary becomes more and more smooth.
- Summary of KNN: easy to understand, implement.
- Disadvantages: O(ND) to label a single data point. Need to carry around entire training set for classification (nonparametric). Choosing the right distance metric and K can be tough.
- Parameters of KNN are all of the training set. The parameters of decision trees are just certain thresholds for features. In this way, we can see the definition of nonparametric for kNN: the number of parameters grows with the increase in training size, while the parameters of decision trees are just certain thresholds for features.
- Other issues with KNN: if some features ranges differ by a lot, one feature can drown out the other one. For example, imagine if we had one feature whose values were in between 5-10 but another feature whose values were in between 1000-20000. We can't know for sure which feature is more important, but the way knn would look at it would cause the second feature to drown out the first.
- Due to this, we need to normalize data:
    - $x_{normed} = \frac{x - \hat{\mu}}{\hat{\sigma}}$. puts x with 0 mean and unit variance. To make all data between 0 and 1, we can use $x_{normed} = \frac{x - x_{min}}{x_{max} - x_{min}}$
- Differences from decision trees: KNN treats each feature as equally important, while decision trees don't, they prioritize feature that maximize the gain in the label Y that we want to predict.
-
- Hyperparameters to tune: K, distance metric.
- **Important dataset/CV stuff:**
    - Training data, validation data, testing data. Validation data is used to optimize hyperparameters. Training data is used for learning. Testing data is used at the very end to see how the final model performs on unseen data. These should not overlap.
    - Hyperparameter tuning with validation dataset:

- For each possible value of the hyperparameter (say max depth = 1,2,3,4,5):
    - train model using D_{train}.
    - evaluate the model on $D_{validation}$.
- Choose hyperparameters that gave the best performance on the validation set.
- Train a model on D_{train} with those hyperparameters
- Finally, use that model + hyperparameters to test on D test.

- We can also use **K-fold cross-validation** in order to find the optimal hyperparameters:
    - Split training data into K equal parts. Denote each part as $D_k^{train} \; \forall \; k \in \{1...K\}$.
    - For each possible value of the hyperparameters (say max-depth = 1,2,...):
        - for every value of k:
            - train model using $D^{Train} - D_k^{Train}$.
            - Evaluate performance of the model on D_k^{train}.
            - Save relevant data.
        - Average the k performance metrics. The performance of this hyperparameter is given by the average performance metric. Save it.
    - Now, with all these hyperparameters, find the one that corresponds to the best average performance.
    - Use that set of hyperparameters while training a model on all of D_{train}.
    - Finally, evaluate the model on D_{test}.

<u>Lecture 6: Perceptrons and Linear Classifiers</u>

- Key assumption: the data are linearly separable.
- The perceptron algorithm draws a line (in the case of binary classification), or several lines (in the case of multi-class classification) to separate data. Consider a line: $w_1x_1 + ... \; w_nx_n + b = 0$ with $n + 1$ parameters, in n-dimensional space. For example, if we take the tuple of parameters $(w_1, \; w_2, \; b) \; = (1, \; -1, \; 0)$ we get a line in R^2 that cross the origin.
- Perceptron learning (only binary classification case for now):
    - We have $x \in R^D$ and the label $y \in (-1, \; + \; 1)$. We have a set of hypothesis functions $H: \; \{h \mid h: \; X -> \; Y\}$ where $h(x) = sign(\sum_{d=1}^{D} w_d x_d + b)$. Our goal is to learn the function $h(x)$ which involves learning the optimal weights and bias.

- Linear Algebra overview: we have $w \in R^D$ and $x \in R^D$, and the dot product $w^t x$ + b gives us a scalar. To not have to carry around the extra bias term all the time, we can use the bias trick in order to absorb the bias into the vectors. Just append the bias to the end of the weights vector, and a 1 to the end of the feature vector x.
- Imagine the equation $w^T x + b = 0, b = 0$. Then the vectors x and w are perpendicular. In 2d, we have the line w and a corresponding line L that is perpendicular to W. On the left side denotes all values such that the dot prod is less than 0, and on the right denotes all values such that the dot prod is > 0 (try to draw a pic).
- <u>Perceptron prediction:</u>
    - Input: $x \in R^D, w \in R^D, b \in R$. The activation, $a = \sum_{d=1}^{D} w_d x_d$. Then, we have our prediction $\hat{y} = sign(a)$. We can think of the perceptron as "firing" if the sign is 1, and not firing if it is 0.
- <u>Perceptron learning:</u>
    - If we have 1 training example x_n, y_n then how can we change w such that $y_n = sign(w^T x_n)$ (ie, the prediction is correct?
    - 2 cases: let $a = w^T x_n$.
    - $y_n = sign(a)$. Do nothing.
    - $y_n != sign(a)$. Then update the weights: $w_{new} = w_{old} + y_n x_n$
- Intuition for why this works:
    - If $y_n a \leq 0$, this means that the signs differ. Then we know that $y_n(w^T x_n) \leq 0$. What happens if we do the update $w_{new} := w_{old} + y_n x_n$? We obtain the sign as $y_n([w + y_n x_n]^T x_n) = y_n w^T x_n + y_n{}^2 x_n{}^T x_n$. This expression has two terms: the first is <= 0 (since we misclassified), and the second term is clearly positive. Therefore, Our next prediction with the new weights will bring $y_n(w_{new}{}^T x_n)$ closer to being greater than zero. If we repeat this a lot of times, the signs will eventually match.
    - Note that the expression in the update $y_n x_n$ may be either positive or negative, but the second term in the activation function is certainly positive.
    - Perceptron learning algorithm:
        - Initialize weights.
        - For i in num_epochs/maxIter:
            - pick x_n. Let $a = w^T x_n$.
            - If $a_n y_n <= 0 : w := w + y_n x_n$.
    - Questions to consider: how do we initialize w? When do we consider convergence to be reached?
    - Design decisions: maxIter/num_epochs: a hyperparameter that denotes how many times we present the dataset to the learning algorithm. While

presenting our data to the algorithm, It's good to shuffle the data for each epoch.

- Properties of perceptron learning:
    - Online algorithm (looks at one instance at a time)
    - Does it necessarily converge?
        - Converges for sure only if the training data are 100% linearly separable. Otherwise, the algorithm does not necessarily converge and we must stop it after iterating a maximum number of times or reaching below a certain error threshold.
    - Extensions to perceptron algorithm: Voting (choosing the weights that perform best overall, instead of always updating the weights when an error is seen). Averaging
- In some respect, the magnitude of the weights for each feature denotes how important the particular feature is.
- Protip: If the original set of features is not linearly separable, you can try to add/remove/transform features so that they ARE linearly separable. But often, this is very difficult and requires methods such as PCA. This is why deep neural networks are very useful if we have massive datasets: we don't even need to worry about knowing what are features are, the deep neural network takes care of figuring out which features are important and which are not.