# CS 480 Course Notes

## Introduction to Machine Learning

**Michael Socha**

**University of Waterloo**
**Winter 2019**

# Contents

# 1 Course Overview

This is an applied introductory machine learning course covering the basics of machine learning algorithms and data analysis. Topics covered include:

- Regression analysis
- Probabilistic modeling
- Support vector machines
- Supervised vs unsupervised learning
- Reinforcement learning
- Neural networks

# 2 Introduction - What is Machine Learning?

Machine learning is the field of study of how computers can improve their performance at tasks (i.e. learn) without being explicitly programmed to do so. Machine learning can be useful for tasks for which it is difficult to write a step-by-step imperative program. Sample applications include optical character recognition, computer vision, and game playing.

## 2.1 Learning Frameworks

- **Supervised Learning:** Goal is to learn a function based on its input and output (e.g. determining if an email is spam based on a set of emails labeled as spam or not spam).
- **Unsupervised Learning:** Goal is to learn a function based on its input alone (e.g. organizing data into clusters).
- **Reinforcement Learning:** Goal is to learn a sequence of actions that maximize some notion of reward (e.g. learning how to control a vehicle to perform some maneuver).

## 2.2 Challenges

Some of the challenges facing machine learning today are:

- Dealing with large amounts of data (algorithm complexity and distributed computing become very relevant)
- Generating reproducible results
- Challenges concerning real-world adoption of work (e.g. human computer interaction, robustness, ethical concerns)

# 3 Decision Trees

Decision trees contain questions as nodes and answers as edges, and serve to guide to an answer based on a set of observations.

## 3.1 Predictions Using Decision Trees

Consider a data set of employees and their job satisfaction, where we use the data set to predict whether an employee is satisfied with their job. An employee can have many features, such as age, salary, seniority, working hours. It is infeasible to build all possible decision trees when there are many features due to the exponential growth of the tree. Instead, for predictive purposes, it makes sense to focus on those questions that are informative to the prediction. For example, if there is no major difference in job satisfaction based on an employee's age, then it is unnecessary to include age in a decision tree, since the answer is not informative for the prediction.

In general, supervised learning problems have a set of possible inputs $X$, an unknown target function $f : X \to Y$, and a set of function hypotheses $\{h|h : X \to Y\}$. Supervised learning algorithms accept training examples (a pair $(x, y)$ where $x \in X$, $y \in Y$) and output a function hypotheses that approximates $f$. When using decision trees for learning, each decision tree is a function hypothesis.

## 3.2 Encoding Functions in Decision Trees

Boolean functions can be fully expressed in decision trees, with one branch for true and another for false. Other function can be approximated as a boolean function. For example, instead of a node asking what an employee's salary is, it could ask whether the salary is above a certain amount.

## 3.3 Training and Testing

The key idea behind decision tree generation algorithms is to grow a tree until it correctly classifies all training examples. The rough procedure followed is:

1. If all training data has the same class, create a leaf node and return.
2. Else, create the best (i.e. most informative) node on which the split the data.
3. Split the training set over the above node.
4. Continue the procedure on each subset of training data generated.

### 3.3.1 Information Content

Criteria for finding the "best" split of data can be defined mathematically. If event $E$ occurs with probability $P(E)$, then when $E$ occurs, we receive $I(E) = \log_2 \frac{1}{P(E)}$ bits of information. This can be interpreted as that less likely events yield more information.

### 3.3.2 Entropy

An information source $S$ which emits results $s_1, s_2, ..., s_i$ with probabilities $p_1, p_2, ..., p_i$ produces information at $H(S) = \sum_i^k p_i I(s_i)$. $H(S)$ is known as the information entropy of S. Information entropy can vary between 0, which indicates no uncertainty (i.e. all members of $S$ in same class), and 1, which indicates high uncertainty (i.e. equal probability of all classes). The best split of data for classification is one that maximally decreases its entropy (a concept known as information gain).

## 3.4 Overfitting

A hypothesis $h_1 \in H$ is said to overfit training data if there is some alternative hypothesis $h_2 \in H$ that has a larger error over the training data but a smaller error over a larger set of inputs. Overfitting can occur due to errors in a data set or just due to coincidental irregularities (especially in a small dataset). Overfitting can be avoided by removing nodes with low information gain, either by stopping decision tree construction early or by pruning such nodes after the tree is constructed.

## 3.5 Inductive Bias

Inductive bias refers to the assumptions made about the target function to predict future outputs. Common inductive biases for decision trees are:

- Assumption that simplest hypothesis is the best (i.e. Occam's razor).
- Decision trees with information gain closer to the root are considered better.

These are examples of preference bias, which influence the ordering of the hypothesis space. This is distinct from restriction bias, which limits the hypothesis space.

## 3.6 Advantages and Disadvantages

Decision trees are good for:

- Ease of interpretation
- Speed of learning

Limitations of decision trees include:

- High sensitivity, with tree output changing significantly due to small changes in input.
- Not good for learning data sets without axis-orthogonal (i.e. constant in all but 1 dimension) decision boundaries.

# 4 Evaluation of Learning

## 4.1 Performance Formulation

Let $\hat{y}$ be an output generated by a function $f$ approximating some target function. Let $y$ be the corresponding output of the target function. A loss function $l(y, \hat{y})$ can be used to measure the accuracy of the approximation function $f$. Some common loss functions include:

- Squared Loss: $l(y, \hat{y}) = (y - \hat{y})^2$
- Absolute Loss: $l(y, \hat{y}) = |y - \hat{y}|$
- Zero/One Loss: $l(y, \hat{y}) = 1_{y \neq \hat{y}}$

We assume that the data coming from our target function comes from some probability distribution $D$, and that our training data is a random sample of $(x, y)$ pairs from $D$. A Bayes Optimal Classifier is a classifier that for any input $x$, returns the $y$ most likely to be generated by $D$.

Based on the available training data, the goal of supervised learning is to find a mapping $f$ from $x$ to $y$ such that generalization error $\sum_{(x,y)} D(x, y) l(y, f(x))$ is minimized. However, since $D$ is unknown, we instead estimate the error from the average error in our training or test data, which is $\frac{1}{N} \sum_{n=1}^{N} l(y_n, f(x_n))$.

## 4.2 Common Learning Challenges

Some common challenges in learning include:

- Inductive bias of the algorithm being distant from the concept actually being learned.
- The data itself having challenging characteristics, such as lots of noise, ambiguity, or missing information.

## 4.3 Training vs Validation vs Test Sets

Models are initially built based on a training dataset. Test sets (also known as holdout sets) are then used to estimate the generalization error. Validation sets are also used to measure the model's performance, but unlike test sets, validation sets can make changes to the model's parameters.

## 4.4 Bias-Variance Decomposition

Many machine learning algorithms are based on building a formal model based on the training data (e.g. a decision tree). Models have parameters, which are characteristics that can help

in classification (e.g. a node in a decision tree). Models may also have hyper-parameters, which in turn control other parameters in a model (e.g. max height of decision tree).

Generalization errors result from a combination of noise, variance, and bias. Bias concerns how well the type of model fits the data. Models with high bias pay little attention to training data and suffer from underfitting, while models with low bias may pay too much attention to training data and become overfitted. Bias and variance tend to be at odds with one another (high bias typically leads to low variance, and vice versa). For example, a decision tree that makes the same prediction for all input has high bias and low variance, while a decision tree trained to return a correct prediction for each point of training data will have low bias and likely high variance.

## 4.5 Cross Validation

Cross validation is a technique for measuring how well a model generalizes. The idea behind it is to break up a training data set into $K$ equally sized partitions, and use $K-1$ of the partitions as training data and the remaining partition for testing. This should be repeated $K$ times, so that all points of data are at some point used for testing. Higher values of $K$ lower the amount of variance of in the error estimation. To avoid training and testing data having a different probability distribution, the data should be shuffled before being split.

## 4.6 Bootstrapping

Bootstrapping is an alternative to cross validation where instead of dividing a training data set into partitions, a random sample of points (with possible duplicates) is used as training data. The remaining points are then used as testing data, with the goal being similar to that of cross validation.

## 4.7 Avoiding Overfitting

Overfitting often occurs when training data has many features, which allows for an approximation of the target function with many degrees of freedom. Overfitting can be avoided by only considering features with a strong correlation to the output. Cross-validation could be applied as follows:

1. Divide training data into $K$ groups at random.
2. Within each group, find a small set of features with strong correlation to the output. Running this step for each group individually instead of for the entire training set further helps avoid overfitting.
3. Build a classifier using the features and examples from the $K-1$ groups.
4. Use the classifier to predict the examples and in group $K$ and measure the error.

5. Repeat 2-4 to produce an overall cross-validation estimate of the error.

Bootstrapping can be applied in a similar way to avoid overfitting.

## 4.8 Performance Evaluation of Classifiers

Consider the following terminology for classification problems:

- True positive ($TP$) - Examples of class 1 predicted as class 1
- False positive ($FP$) - Examples of class 0 predicted as class 1 (Type 1 Error)
- True negative ($TN$) - Examples of class 0 predicted as class 0
- False negative ($FN$) - Examples of class 1 predicted as class 0 (Type 2 Error)

### 4.8.1 Accuracy and Error

The following formulas can be used to measure accuracy and error:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

$$ErrorRate = \frac{FP + FN}{TP + TN + FP + FN}$$

### 4.8.2 Precision and Recall

Precision and recall can be measured as follows:

$$P = \frac{TP}{TP + FP}$$

$$R = \frac{TP}{TP + FN}$$

Precision measures the ratio of positive predictions that were correct, while recall measures the ratio of total positive instances that were predicted. Similarly to how variance and bias are often at odds with one another, so are precision and recall.

### 4.8.3 F-Measure

An F-measure (also known as a F1 score) measures a model's accuracy by taking into account both precision and recall as follows:

$$F = \frac{2PR}{P + R}$$

To adjust the relative importance of precision vs recall, a weighted F-measure can be used, which is defined as follows:

$$F = \frac{(1+\beta^2)PR}{\beta^2 P + R}$$

In a standard F-measure, $\beta = 1$, $\beta < 1$ means that precision is valued over recall, while $\beta > 1$ means recall is valued over precision.

### 4.8.4 Sensitivity and Specificity

Sensitivity is the same measure as recall. Specificity is a measure of how well a classifier avoids false positives, and is measured as:

$$Specificity = \frac{TN}{TN + FP}$$

# 5 Instance-Based Learning

## 5.1 Parametric vs Non-Parametric Methods for Supervised Learning

Datasets can be represented as a set of points in a high-dimensional space; a data point with $n$ features $f_1, f_2, ...f_n$ can be represented with the feature vector $(f_1, f_2, ..., f_n)$ in n-dimensional space. Parametric methods of supervised learning attempt to model the data using these features, while non-parametric (also known as instance-based) methods do not.

### 5.1.1 Approximation

Parametric methods use parameters to create global approximations. Non-parametric methods instead create approximations based on local data.

### 5.1.2 Efficiency

Parametric methods do most of their computation beforehand, and the summarize their results in a set of parameters. Non-parametric methods tend to have a shorter training time but a longer query answering time.

## 5.2 K-Nearest Neighbors

K-nearest neighbors (KNN) is a common non-parametric method. The idea is to predict the value of a new point based on the values of the $K$ most similar (i.e. closest) points.

### 5.2.1 Implementation

A common implementation of KNN involves looping through all $N$ points in a training set and computing their distance to some point $x$. Then the $K$ nearest points are selected. This process can be sped up by storing the data points in a data structure that helps facilitate distance-based search (e.g. a k-d tree).

### 5.2.2 Distance Function

"Nearby" means of minimal distance, which is commonly defined by Euclidean distance. Other distance functions $d(x, x')$ can be used, though must meet the following conditions:

- $d(x, x') = d(x', x)$ (i.e. symmetric)
- $d(x, x) = 0$ (i.e. definite)

- $d(a, c) \leq d(a, b) + d(b, c)$ (i.e. triangle inequality holds)

### 5.2.3 Decision Boundaries

Decision boundaries define the borders of a single classification of input. These boundaries are formed of sections of straight lines that are equidistant to two points of different classes. A highly jagged line is an indicator of overfitting, while a simple line is an indicator of underfitting.

### 5.2.4 Selection of K

The selection of the value of $K$ is a bias-variance tradeoff. Low values of $K$ have high variance but low bias, while high values of $K$ have low variance but high bias. High-values of $K$ result in smoother decision boundaries, which can be a sign of underfitting, and vice versa.

$K$ can be selected experimentally by evaluating the performance for different values of $K$ through cross-validation or against a testing set. In theory, as the number of training examples approaches infinity, the error rate of a 1NN classifier is at worst twice that of the Bayes Optimal Classifier.

### 5.2.5 Pre-Processing

Some common forms of pre-processing for KNN include:

- Removing undesirable inputs. Common removal methods are:
  - Editing methods, which involve eliminating noisy points of data.
  - Condensation methods, which involve selecting a subset of data that produces the same or very similar classifications.
- Use custom weights for each feature (not all features may be equally relevant for the situation)

### 5.2.6 Distance-Weighted Nearest Neighbor

A common problem with KNN is that it can be sensitive to small changes in the training data. One way to mitigate with drawback is to compute a weight for each neighbor based on its distance (e.g. through a Gaussian distribution), and this weight determines how much of an influence that point's value has. This differs from standard KNN which weighs the values of the $K$ nearest neighbors equally and ignores all other values.

### 5.2.7 High Dimensionality

In uniformly distributed high-dimensional spaces, distances between points tend to be roughly equal, since there are so many features that changing a few features results in only a small change in distance. However, KNN can still be applied in practice for high-dimensional spaces, since data in high-dimensional spaces tends to be concentrated around certain hubs rather than uniformly distributed.

# 6 Perceptron

A perceptron is a binary classification algorithm that accepts a set of features associated with weights.

## 6.1 Formulation

Let $x = (x_1, x_2, ..., x_D)$ be a feature vector with $D$ features, let $w = (w_1, w_2, ..., w_D)$ be the corresponding weight vector, and let $b$ be some fixed bias/threshold term. Activation $a$ is defined as:

$$a = w^T x + b = \sum_{d=1}^{D} w_d x_d + b$$

If $a \geq 0$, then the perceptron classifier predicts a positive classification, and otherwise it predicts a negative classification.

## 6.2 Decision Boundary

A perceptron decision boundary is where the sign of the activation changes from negative to non-negative. It can be expressed as $a = w^T x + b = 0$, so $w^T x = -b$. Hence, the decision boundary is a hyperplane perpendicular to $w$. The bias term can shift the hyperplane, but does not change its alignment.

If the weight and bias terms are scaled together, then the decision boundary does not change. However, it is common to normalize the weight vector to have a magnitude of 1.

## 6.3 Learning Algorithm

The perceptron learning algorithm starts with any (e.g random) $w$ and $b$ parameters, and iterates over each point training point as follows:

1. Compute the activation $a$ for the point.
2. Compute the classification $y$ based on the activation $a$ (i.e. 1 if $a \geq 0$, otherwise 0).
3. If the perceptron correctly classifies the point, continue.
4. Otherwise, update the weights as $w_d = w_d + yx_d$, and update the bias as $b = b + y$. This guarantees that this point is correctly classified for this iteration.

The perceptron algorithm is guaranteed to converge on parameters (i.e. weights and bias) that correctly classify a set of points if such parameters exist.

The algorithm is considered to be:

- **Online**, since data becomes available in sequential order instead of all at once.
- **Error driven**, since it requires an incorrectly classified point to update its parameters.

### 6.3.1 Hyper-parameters

The only hyper-parameter of the perceptron algorithm is the maximum number of iterations of the above algorithm (i.e. the number of times each point is considered). A high number of iterations can lead to overfitting, while a low number of iterations can lead to underfitting. Since the order in which training points are encountered matters in the training algorithm, it is important to permute the order of examples between iterations.

### 6.3.2 Extensions

A potential problem with the perceptron algorithm described above is that later training points are weighted more than earlier ones. For instance, it is possible that after selecting weights that correctly classify thousands of training points, a single mislabeled point updates the weights in a way that drastically changes the decision boundary in order to correctly classify this new point, but in doing so, mislabels many training points that were previously correctly classified.

This problem can be mitigated by favoring weight vectors that "survive" a long time. This can be done by letting the activation $a$ use a sum of weight vectors, each of which in turn is weighted by its survival time (i.e. voted perceptron algorithm). Alternatively, a single weight vector and bias term can be tracked using the weighted average of their survival times (i.e. average perceptron algorithm).

## 6.4 Convergence

Data can only be classified by a perceptron if it is linearly separable. Otherwise, the perceptron algorithm will not converge, and the decision boundary will oscillate.

The speed of convergence of the perceptron algorithm depends on its margin, which is the distance from the separating hyperplane to the nearest data point. Large margins result in faster convergence, since there is more "wiggle room" to adjust the parameters defining the hyperplane.

### 6.4.1 Rosenblatt's Perceptron Convergence Theorem

Let the margin $\gamma$ of a linearly separable dataset $D$ be the largest attainable margin on that dataset. Rosenblatt's perceptron convergence theorem states that if $||x|| < 1$ for all $x \in D$, then the perceptron algorithm will converge after at most $\frac{1}{\gamma^2}$ updates. This is an upper bound; the number of updates also depends on the dataset and the initial parameters.

## 6.5 Limitations

Two key limitations of perceptron classifiers are:

- The generated solutions (i.e. separating hyperplanes) may not be unique.
- Non linearly-separable data cannot be modeled. In some cases, this limitation can be overcome by transforming feature vectors so they become linearly separable, or by combining multiple perceptrons, which can create a network that might be able to learn non-linear boundaries.

# 7 Linear Models

## 7.1 Formulation

A loss function concerning the distance $d$ between the value of points $x$ predicted by some model $w, b$ and their actual values $y$ can be minimized as follows:

$$min_{w,b} \sum_n d(y_n, w^T x_n + b) + \lambda R(w, b)$$

The first term measures the training error, while the second term is known as a regularizer, and exists to prevent overfitting. $\lambda$ is a constant hyper-parameter while $R$ is a function that imposes some penalty based on the complexity of parameters $w, b$.

## 7.2 Training Parameters

### 7.2.1 Gradient Descent Solution

The gradient of error can be used to determine the direction of steepest decline in error, and the training algorithm then adjusts parameters to move in that direction. The training algorithm maintains the state of some parameter $z$, finds its gradient $g$, and updates $z$ to be $z + \eta g$, where $\eta$ is the learning rate (i.e. the size of each step).

As an example, consider a loss function:

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

The partial derivative of $L$ over $b$ is $-\sum_n y_n \exp(-y_n(w^T x_n + b))$, so $b$ is adjusted in each step to $b + \eta \sum_n y_n \exp(-y_n(w^T x_n + b))$.

The partial derivative of $L$ over $w$ is $(-\sum_n y_n x_n \exp(-y_n(w^T x_n + b))) + \lambda w$, so $w$ is adjusted in each step to $w + \eta((-\sum_n y_n x_n \exp(-y_n(w^T x_n + b))) + \lambda w)$.

The learning rate $\eta$ impacts the size of each step. A high learning rate can result in oscillations around local minima, while a low learning rate results in training being slow. It is common to lower the learning rate during later steps. When parameters start to not change by much, then a local minimum has likely been approached, and the algorithm can stop.

Note that gradient descent does not guarantee convergence to a global minima; it can get stuck in a local minima that may not be optimal. However, approximations using linear functions have a single global minimum.

### 7.2.2 Closed Form Solution

The gradient descent solution iteratively approaches some minimum. On the other hand, a closed form solution provides a formula for the absolute minimum. A closed form solution is not always attainable, but this section focuses on an attainable case, which is where the loss function is measured by Euclidean distance.

Our model can start as linear; we assume that the target function can be modeled as $f_w(x) = w_0 + w_1x_1 + w_2x_2 + ... + w_Dx_D$. A linear model is often a poor approximation, so further input variables can be added based on $x_1, ..., x_D$, such as:

- Transformations of variables, such as $\log x_i$
- Interaction terms, such as $x_i \cdot x_{i+1}$
- Numeric encoding of qualitative variables

The goal of the closed-form solution is to minimize $\sum_{i=1}^{n}(y_i - w^T x_i)^2$ (i.e. minimize Euclidean distance of errors). It is convenient to rewrite the problem in matrix format where:

- $X$ is a $NxD$ training data matrix, where $X_{n,d}$ is the value of feature $d$ on example $n$.
- $w$ is a size $D$ column vector storing weights.
- $\hat{Y} = Xw$ are the predicted values.

Since we aim to minimize squared error, we aim to minimize $||\hat{Y} - Y||^2 + \lambda||w||^2$. Taking the derivative of this against $w$ and setting it to 0 results in $w = (X^TX + \lambda I_D)^{-1}X^TY$. This equation takes $\theta(D^3 + D^2N + DN)$ steps to compute, so this method can be computationally expensive for very high-dimensional problems.

## 7.3 Regularization

The Gauss-Markov theorem states that in a linear regression model, such as the one described above, the least-squares estimate (i.e. Euclidean distance) results in a model with the least bias. We nonetheless add a regularization term, which increases bias, to avoid overfitting. A few desirable features of a regularization function are:

- It is convex
- Small (even 0) weight vector components are preferred (note that this is a form of inductive bias).

### 7.3.1 P-norm

The p-norm of the weight vector can be used as a regularization function, defined as follows:

$$R(w) = (\sum_d |w_d|^p)^{\frac{1}{p}}$$

The p-2 norm is the same as the Euclidean norm used above.

### 7.3.2 Ridge Regression

Ridge regression uses the following regularization function:

$$R(w) = \sum_d w_d^2$$

Ridge regression tends to give a smooth solution with low weights.

### 7.3.3 Lasso Regression

Lasso regression uses the following regularization function:

$$R(w) = \sum_d |w_d|$$

Lasso regression leads to a less smooth and more computationally expensive solution than ridge regression. It is equivalent to a p-1 norm.

## 7.4 Application to Classification Problems

The perceptron algorithm covered above can find a hyperplane that splits a linearly separable dataset. However, the perceptron algorithm does not converge for datasets that are not linearly separable.

Due to this limitation, it can make sense to relax the problem to finding a hyperplane that splits data with the fewest possible classification errors. However, due to the discrete nature of 0/1 error, it is NP hard to minimize. Thus, 0/1 classification functions are commonly modeled with continuous approximations. One common class of approximations are convex surrogate loss functions, which extend between 0 and 1 and are convex, which is a property that makes them easy to minimize.

# 8 Probabilistic Methods

Data is often incomplete, indirect, or noisy. Considering such forms of uncertainty can help build better models.

## 8.1 Bayesian Learning

Bayes' theorem describes the probability of an event $H$ given evidence $e$.

$$P(H|e) = \frac{P(e|H)P(H)}{P(e)}$$

The terms of the above equation are known as follows:

- $P(H|e)$: Posterior probability
- $P(e|P)$: Likelihood
- $P(H)$: Prior probability
- $P(e)$: Normalizing constant

Bayesian learning consists of determining the posterior probability using Bayes' theorem.

As an example, consider a bag of candy with an unknown flavor ratio. The possible flavors are cherry and lime. The hypotheses $H$ could be:

- $h_1$: 100% cherry, 0% lime, $P(h_1) = 0.1$
- $h_2$: 75% cherry, 25% lime, $P(h_2) = 0.2$
- $h_3$: 50% cherry, 50% lime, $P(h_3) = 0.4$
- $h_4$: 25% cherry, 75% lime, $P(h_4) = 0.2$
- $h_5$: 0% cherry, 100% lime, $P(h_5) = 0.1$

Sample examples $E$ could be:

- $e_1$: 1st candy is cherry
- $e_2$: 2nd candy is lime
- $e_3$: 3rd candy is lime

Suppose of the first 10 candies, all are lime, and let us compute $P(h_4|e)$. $P(e|h_4) = 0.75^{10}$. $P(e) = \sum_i P(e|h_i)P(h_i) \approx 0.1114$. It follows that $P(h_4|e) = \frac{P(e|h_4)P(h_4)}{P(e)} = \frac{(0.75)^{10} \cdot 0.2}{0.1114} \approx 0.0987$. The probability of the other hypothesis can also be calculated in this manner.

Bayesian probability is:

- **Optimal**; given a prior probability, no prediction is correct more often than the Bayesian prediction.
- **Overfitting-free**; all hypothesis are weighted and considered, eliminating overfitting.

However, Bayesian learning can be intractable when the hypothesis space grows very large, often as a result of approximating a continuous hypothesis space with many discrete hypotheses.

## 8.2 Maximum a Posteriori

Maximum a Posteriori (MAP) makes predictions based on only the most probable hypothesis $h_{MAP}$ (i.e. $h_{MAP} = \arg\max_h P(h|e)$). This differs from Bayesian learning, which makes predictions for all hypotheses weighted by their probability. However, MAP and Bayesian predictions tend to converge as the amount of data increases, and overfitting can be mitigated by giving complex hypotheses a low prior probability. However, finding $h_{MAP}$ can be difficult or intractable.

## 8.3 Maximum Likelihood

Maximum Likelihood (ML) simplifies MAP by assuming uniform prior probabilities, and then makes a prediction based on the most probable hypothesis $h_{ML}$. ML tends to be less accurate than MAP and Bayesian predictions, and due to prior probabilities being uniform, is subject to overfitting. However, finding $h_{ML}$ is often easier than $h_{MAP}$, since finding $h_{ML}$ for $P(e|h)$ is equivalent to calculating it for $\sum_n \log P(e_n|h)$, which is often easier to optimize.

## 8.4 Determining Priors

Selecting a prior is important for Bayesian learning and MAP estimates. If the posterior distributions $P(H|x)$ are in the same family of functions as the prior probability distributions $P(H)$, then the prior and posterior are known as conjugate distributions, and have properties that make them easy to work with.

## 8.5 Bayesian Linear Regression

The linear regression problem discussed in the above section can be reformulated by assuming part of $y$ to be set from a probability distribution $\eta \sim N(0, \omega^2)$ as follows:

$$y = w^T b(x) + \eta$$

$b$ is a fixed set of basis functions (i.e. $b(x) = [b_1(x), b_2(x), ..., b_M(x)]$).

The goal of Bayesian linear regression is not to find the best model parameter $w$, but rather to find a model parameter's posterior distribution as follows:

$$P(w|y, X) = \frac{P(y|w, X)P(w|X)}{P(y|x)}$$

# 9 Linear Models for Classification

## 9.1 Probabilistic Framework for Classification

As covered in the above section, learning can be viewed as a problem of statistical inference. If we have access to the underlying probability distribution of a set of data, then we can form an optimal Bayesian classifier. In practice, we typically do not know the underlying probability distributions, so we have to estimate them from the available training data.

It is usually best to choose a family of parametric distributions (e.g. Gaussian or Binomial) and then determine which parameters describe the available training data the best. This is known as a density estimate, and we assume that each point of training data is independently selected from the same distribution.

## 9.2 Generative Learning

The idea behind generative learning is to separately model $P(x|y)$ and $P(y)$, and to use Bayes' theorem to estimate $P(y|x)$.

$P(x|y) = P(x_1, x_2, ..., x_m|y)$. Using what is known as the naive Bayes assumption, we assume all points $x_i$ are conditionally independent. Thus, $P(x|y) = P(x_1|y) \cdot P(x_2|y) \cdot ... \cdot P(x_m|y)$, greatly simplifying the probability calculation.

### 9.2.1 Training

Training involves finding parameters that maximize $P(y|x) \propto P(y)P(x|y) = \prod_{i=1}^{n}(P(y_i)\prod_{j=1}^{m} P(x_{i,j}|y_i))$. The derivative of the log of the above expression can be taken against output variables to find the probabilities of each output. For example, suppose $\theta_1$ is the likelihood that the output of a binary classification problem is 1. The above expression is maximized at $\theta_1 = \frac{1}{n}\sum_{i=1}^{n} y_i$, which is the number of outputs where $y = 1$ over the total number of outputs.

### 9.2.2 Decision Boundary

The decision boundary occurs where $\frac{P(y=1|x)}{P(y=0|x)} = 1$, or equivalently, the log-likelihood ratio $\log \frac{P(y=1|x)}{P(y=0|x)} = 0$.

### 9.2.3 Laplace Smoothing

Laplace smoothing, also known as additive smoothing, is a technique used to smooth categorical data. For instance, consider the likelihood ratio $P(x_j|y = 1) = \frac{\text{Num instances where } x_j=1, y=1}{\text{Num examples with } y=1}$, and suppose that there are few, or even no instances of $y = 1$. Adding a constant 1 to the numerator and 2 to the denominator can help prevent division by 0 and acts as a Bayesian

prior. Laplace smoothing introduces some bias, but it is hardly significant with a large number of training examples.

### 9.2.4 Multiclass Classification

The above examples assumed binary classification. For multiclass classification, there are two main options:

- Train a single classifier that can produce all classifications.
- Train a 1-vs-all binary classifier for all classes. This is more flexible, but often slower due to the number of separate classifiers that need to be trained, and introduces a class imbalance problem where the target class has relatively fewer points than the aggregation of all other classes.

## 9.3 Discriminative Learning

Discriminative learning attempts to model $p(y|x)$ directly. Logistic regression is a common example of discriminative learning.

Logistic regression is used to model binary outputs. Real target values are produced and are then mapped to a value between 0 and 1. Logistic regression assumes an exponential estimation parametric form of the distribution $P(Y|X)$ for the form $\sigma(w^T x) = \frac{1}{1+e^{-w^T x}}$. $\sigma(w^T x)$ is the probability that $y_i = 1$, and $1 - \sigma(w^T x)$ is the probability that $y_i = 0$. It follows that

$$P(Y = 1|X) = \frac{1}{1 + \exp\left(w_0 + \sum_{i=1}^{n} w_i X_i\right)}$$

$$P(Y = 0|X) = \frac{\exp\left(w_0 + \sum_{i=1}^{n} w_i X_i\right)}{1 + \exp\left(w_0 + \sum_{i=1}^{n} w_i X_i\right)}$$

### 9.3.1 Training

The goal of training is to find weights that maximize the likelihood $P(x_1, y_1, ..., x_n, y_n|w) = \prod_{i=1}^{n} = \sigma(w^T x_i)^{y_i}(1-\sigma(w^T x_i))^{(1-y_i)}$, which is equivalent to minimizing $-\sum_{i=1}^{n} y_i \log\left(\sigma(w^T x_i)\right) + (1 - y_i) \log\left(1 - \sigma(w^T x_i)\right)$

### 9.3.2 Decision Boundary

The decision boundary of logistic regression is the same as that of the naive Bayesian classifier (i.e. $\log \frac{P(y=1|x)}{P(y=0|x)} = 0$).

# 10 Support Vector Machines

## 10.1 Margins

A binary linear classifier can be defined as $h_{w,b}(x) = g(w^T x+b) = 1$ if $w^T x+b \geq 0$, otherwise $-1$. A large margin involves points begin far from the decision boundary. There are two main ways to measure margin.

### 10.1.1 Functional Margins

For a training example $(x_i, y_i)$, the functional margin is $\hat{\gamma}_i = y_i(w^T x_i + b)$. However, this is not a good measure of classification confidence, since $w$ and $b$ can be scaled to increase the margin without affecting the classification.

### 10.1.2 Geometric Margins

Geometric margins avoid the above problem with functional margins by normalizing the weight vector $w$. For a training example $(w_i, y_i)$, the geometric margin is $\gamma_i = y_i \frac{w^T x_i+b}{||w||}$.

### 10.1.3 Margin of Training Set

The margin of a training set is the smallest margin of its individual training examples. Minimizing the geometric margin for a binary classifier is equivalent to minimizing $||w||$.

## 10.2 Optimal Margin Classifier

The goal of an optimal margin classifier is to find the maximum geometric margin for a linearly separable data set. This can be denoted as solving $\max_{w,b} \frac{1}{2}||w||^2$ such that $y_i(w^T x_i + b) \geq 1$ for $i = 1, \ldots, n$. This is the problem that a support vector machine (SVM) attempts to solve.

## 10.3 Constrained Optimization

### 10.3.1 Lagrange Multipliers

The above equation can be treated as a constrained optimization problem. These types of problems can be solved by Lagrange multipliers.

Given a constrained optimization problem of the form $\min_w f(w)$ such that $h_w(w) = 0$ for $i = 1, \ldots, l$, the Lagrangian is defined as $L(w, \beta) = f(w) + \sum_{i=1}^{l} \beta_i h_i(w)$, where $\beta = \{\beta_1, \ldots, \beta_l\}$

are called Lagrangian numbers. To solve for $w$ and $\beta$ we set the partial derivatives of $L$ over $w$ and $\beta_i$ to 0.

Consider an example where we try to minimize $x+y$ such that $w^2+y^2=1$. The Lagrangian is $L(x,y,\beta)=x+y+\beta(x^2+y^2+1)$. Setting the gradient to 0 with respect to $x$, $y$, and $\beta$ produces the system of equations $1-2\beta x=0$, $1-2\beta y=0$, and $x^2+y^2-1=0$, which can be solved to yield $x=y=-\frac{1}{\sqrt{2}}$.

The generalized Lagrangian includes a $g_i(w)\leq 0$ constraint for $i=1,\ldots,k$. The generalized Lagrange is denoted as $L(w,a,b)=f(w)+\sum_{i=1}^{k}\alpha_i g_i(w)+\sum_{i=1}^{l}\beta_i h_i(w)$.

### 10.3.2 Primal and Dual Problem

Given the generalized Lagrangian above, we can define a primal $\theta_P(w)$ as $\max_{\alpha,\beta,a_i\geq 0} L(w,\alpha,\beta)$. If $f$ satisfies all constraints, then $\theta_P(w)=f(w)$, otherwise $\theta_P(w)$ is infinite. The "primal problem" consists of finding $p*=\min_w \theta_P(w)$.

In the "dual problem", we first minimize with respect to $w$, and then maximize with respect to $\alpha,\beta$. The dual is defined as $\theta_D(\alpha,\beta)=\min_w L(w,\alpha,\beta)$, and the "dual problem" consists of finding $d*=\max_{\alpha,\beta,\alpha_i\geq 0}\theta_D(\alpha,\beta)$.

The "max min" of a function is always less than or equal to the "min max", so $d*\leq p*$. Under conditions known as Karush-Kuhn-Tucker (KTT) conditions, $d*=p*$.

## 10.4 Training

The high-level goal of training is to find $w,b$ such that $\frac{1}{2}||w||^2$ is minimized and $y_i(w^Tx_i+b)\geq 1, i=1,2,\ldots,n$.

We can write the constraints for each training example $i$ as $g_i(w)=1-y_i(w^Tx_i+b)\leq 0$. The Lagrangian of the optimization problem is:

$$L(w,b,a)=\frac{1}{2}||w||^2+\sum_{i=1}^{n}\alpha_i(1-y_i(w^Tx_i+b))$$

$\frac{\partial L(w,b,a)}{\partial w}=w-\sum_{i=1}^{n}\alpha_i y_i x_i$, which is set to 0 at $w=\sum_{i=1}^{n}\alpha_i y_i x_i$. Also, $\frac{\partial L(w,b,a)}{\partial b}=\sum_{i=1}^{n}\alpha_i y_i=0$. Plugging these back into $L$, we end up with the dual problem for SVM being $\max_\alpha \sum_{i=1}^{n}\alpha_i-\frac{1}{2}\sum_{i,j=1}^{n}y_iy_j\alpha_i\alpha_j(x_i^Tx_j)$ such that $\alpha_i\geq 0$ for $i=1,\ldots,n$ and $\sum_{i=1}^{n}\alpha_i y_i$ for $i=1,\ldots,n$.

If both $f$ and $g_i$ are convex, and all $g_i$ can be satisfied for some $w$, then $d*=p*=L(w*,b*,\alpha*)$, where $w*$ is the optimal weight vector set (primal solution) and $\alpha*$ is the optimal support vector set (dual solution). In SVMs, both $f$ and $g_i$ are convex, and for linearly separable data, all $g_i$ can be satisfied simultaneously. $g_i(w)$ that equal 0 are on the decision boundary, and are known as support vectors. The output classifier of a query point

$x$ is computed as $h_{w,b}(\hat{x}) = sign(\sum_{i=1}^{n} \alpha_i y_i (x_i^T \hat{x}) + b)$. Since $a_i > 0$ for only support vectors, only support vectors are considered in this computation.

## 10.5 Handling Non-Linearly Separable Data

Two options to model data that cannot be classified using a linear boundary are:

- Relaxing the constraints by allowing some points to be incorrectly classified by the margin. In this case, $\min_{w,b} \frac{1}{2}||w||^2$ such that $y_i(w^T x_i + b) \geq 1, i = 1, 2, \ldots, n$ can be rewritten as $\min_w \frac{1}{2}||w||^2 + L_(w^T x_i + b)$, where $L$ is some loss function penalizing incorrect classifications or points close to the decision boundary. A common loss function is known as hinge loss, and is defined as $\max(1 - y_i(w^T x_i + b), 0)$.
- Allowing a non-linear decision boundary by finding a linear decision boundary of some mapping $\phi$ of features.

# 11 Kernel Methods

As mentioned in the section above, the dual form of SVM can be expressed as $\max_\alpha \sum_{i=1}^n \alpha_i - \frac{1}{2}\sum_{i,j=1}^n y_i y_j \alpha_i \alpha_j (x_i^T x_j)$ such that $\alpha_i \geq 0$ for $i = 1, \ldots, n$ and $\sum_{i=1}^n \alpha_i y_i$ for $i = 1, \ldots, n$. If applying a feature mapping $\phi$, the $x_i^T$ and $x_j$ terms become $\phi(x_i)^T$ and $\phi(x_j)$ respectively. It follows that classification of an input $x$ is given by $sign(\sum_{i=1}^n \alpha_i y_i (\phi(x_i)^T \phi(\hat{x})) + b)$, where $\phi(x_i)^T \phi(\hat{x})$ is known as the kernel.

## 11.1 Motivation

A linear model does not make use of $\phi$, which results in simple decision boundaries. Data that is not linearly separable is often separable in higher dimensions, introducing the need for more complex mappings $\phi$.

The motivation for adding kernels is similar to that of expanding the feature space to include more terms. For example, consider a quadratic feature space, where a set of features $x_1, x_2, \ldots, x_m$ is mapped to $x_1x_1, x_1x_2, \ldots x_1x_m, x_2x_1, x_2x_2, \ldots, x_2x_m, \ldots$. Problems with such a large feature space are increased computational cost and an increased chance of overfitting. However, this feature space can be represented by the kernel function $K(x, z) = (x^T z)^2$, in which case its calculation takes $O(m)$ steps instead of $O(m^2)$. Kernels can be widely applied to reduce computational complexity by rewriting algorithms in SVM dual form and then transforming the dot product term into a kernel function.

## 11.2 Common Kernels

- **Radial basis function (RBF)/Gaussian Kernel** $K(x, z) = \exp(-\frac{||x-z||^2}{2\sigma^2})$: High when points are close, decreases as points are further apart.
- **Linear Kernel** $K(x, z) = x \cdot z$
- **Polynomial Kernel** $K(x, z) = (1 + x \cdot z)^d$: Feature expansion has monomial terms of power $d$.

## 11.3 Kernel Properties

Let a kernel matrix $K$ for $n$ points $x_1, \ldots, x_n$ be defined as $K_{ij} = K(x_i, x_j)$. A kernel $K$ is considered to be a Mercer kernel if and only if for any finite set of $n$ points, the corresponding kernel matrix is symmetric and positive semi-definite. Symmetric means that $K(x_i, x_j) = K(x_j, x_i)$. Positive semi-definite means that for all functions $f$ that are square integrable (i.e. $\int f(x)^2 dx < \infty$) other than the 0 function, $\int \int f(x)K(x, z)f(z)dxdz > 0$.

This characterization is useful because it provides ways to construct kernels from other kernels. For example, adding, multiplying, or scaling two Mercer kernels always generate a Mercer kernel.

## 11.4 Kernel Perceptron Algorithm

By the perceptron representer theorem, the weight vector during a run of the perceptron algorithm can be described as a linear combination of the expanded training data (i.e. $w = \sum_{i=1}^{n} a_i \phi(x_i)$). We can alter the perceptron algorithm using this property to update $a$ to be $w \cdot \phi(x) + b$ and $w$ to be $w + y\phi(x)$.

# 12 Feature Construction and Selection

A high-level overview of a supervised learning procedure is:

1. Decide what the inputs and outputs of a problem are.
2. Decide how to model these inputs and outputs.
3. Choose a class of hypotheses.
4. Choose a cost function to help find the best hypothesis.
5. Chose an algorithm for searching the hypothesis space.

So far, this course has covered 3-5. This section focuses on points 1 and 2.

## 12.1 Feature Construction

Features can be modified for various reasons, including to increase predictor performance and to reduce time or memory requirements. Below are common techniques for constructing features.

### 12.1.1 Transformation

Common feature transformations include:

- **Centering** each feature to be around the origin.
- **Scaling** each feature to be of the same scale. For example, scaling can be done to make sure each feature has the same variance or the same maximum absolute value.
- **Logarithmically** transforming each feature to reduce the skewness of feature distributions.

Note that feature transformation runs the risk of discarding useful information. For example, scaling to make each feature have the same variance should not be done if the differing variances of the features are actually relevant to the problem.

### 12.1.2 Feature Expansions

Feature expansion involves combining multiple features into new features when first order interactions are not good enough. For example, given features $x_1$ and $x_2$, $x_1 \cdot x_2$ is a new feature (i.e. meta-feature) formed by an expansion of $x_1$ and $x_2$.

### 12.1.3 "Ad hoc" Features

Constructing ad hoc features involves applying domain knowledge to introduce custom features.

## 12.2 Feature Selection

Irrelevant features are features that are uncorrelated with a prediction task. Redundant features are features that are highly correlated with one another, so using multiple redundant features does not help with predictions much more than using a single such feature.

Different learning algorithms have differing levels of robustness to irrelevant or redundant features. For example, decision trees are robust to redundant features, since such features have low information gain, while KNN is not, since the set of redundant features will behave as one heavily weighted feature. When possible, these feature should not be selected in the first place. Below are common ways to avoid selecting such features.

### 12.2.1 Wrapper Methods

Wrapper methods involve building a model for feature subsets, and then selecting the best performing model. A "forward search" approach starts with no features and then adds the feature that best improves the model until a certain number of features are selected. A "backward" search approach starts with all features and removes the feature that improves the model the least until a certain number of features have been removed.

Computing all possible feature subsets would guarantee finding the optimal one. However, a problem with $m$ features has $2^m$ possible feature subsets, so finding all possible subsets is infeasible for large values of $m$. The forward and backward search approaches approximate this but with a time complexity of $O(m^2)$.

### 12.2.2 Filter Methods

Filter methods, also known as variable ranking, involve assigning each feature a score measuring how informative it is in predictions. This score is determined by some "scoring function" $S$. Features are then ranked by score, and a number of top features are selected.

### 12.2.3 Embedded Methods

Embedded methods involve modifying the cost function to constrain the choice of model. A common example of this is regularization, which can be used to penalize complex models and encourage a sparse feature set.

# 13 Ensemble Learning

Ensemble learning models combine multiple different machine learning models to form predictions. Such models can lead to improved performance over machine learning models that use a single algorithm, often without a large increase in training and test time due to ease of parallelization.

The machine learning models applied in ensemble learning can differ in either their algorithm or in their training data. This section focuses on techniques for forming different subsets of training data.

## 13.1 Baggings

Recall that in bootstrapping, $N$ training examples are drawn randomly from a dataset $D$ with replacement. Bagging can be described as bootstrap aggregation, where each subset of data formed by a round of bootstrapping is used to train its own model. For a classification problem, the overall prediction is the most common prediction of the models. For a regression problem, the overall prediction is the average prediction of the models.

Baggings tends to reduce variance, since even if individuals models overfit to their data, different models will overfit in different ways. However, baggings tends to increase bias. Therefore, bagging is most useful when used with models that are unstable and prone to overfitting (e.g. tall decision trees). Drawbacks of bagging include a loss of interpretability of results and an increase in computational complexity.

## 13.2 Random Forests

Random forest ensemble learning algorithms address that it is often computationally expensive to train parallel decision trees. In random forests, a tree's structure is fixed, and decision nodes are filled with randomly selected features. A common random forest algorithm is as follows:

1. Form $K$ bootstrapped subsets of a dataset $D$.
2. For each of the $K$ subsets, train a decision tree as follows:
   (a) Pick a random subset of $m$ features at each node. Smaller values of $m$ result in more randomized trees.
   (b) Select the feature with maximal information gain.
   (c) Recurse until the tree reached a predefined maximal depth.

The effects of a random forest algorithm are similar to those of bagging. Variance decreases, since different trees will overfit in different ways, and bias increases. A recurring pattern

in ensemble learning algorithms is to form lots of randomized hypothesis and average their results.

## 13.3 Boosting

Instead of averaging out random hypotheses, boosting operates by placing extra weight on training instances that are problematic for the current hypothesis. Boosting can be used to gradually adapt a weak learning model (i.e. one only loosely correlated with correct predictions) to eventually be a strong learning model.

AdaBoost (adaptive boosting algorithm) is a widely-used, simple, and efficient boosting algorithm. The AdaBoost algorithm for binary classification is defined as follows:

1. Given training data $(x_1, y_1), \ldots, (x_m, y_m)$, where $Y \in \{-1, +1\}$, initialize $D_1(i) = \frac{1}{m}$.
2. For $t = 1, \ldots, T$:
   (a) Using dataset $D_t$, train a "decision stump", which is some simple weak learner such as a single-node decision tree.
   (b) Obtain the weak hypothesis $h_t : X \to \{-1, +1\}$ with error probability $\epsilon_t$.
   (c) Let $\alpha_t = \frac{1}{2} \ln(\frac{1-\epsilon_t}{\epsilon})$. $\alpha_t$ is determines the rate at which the hypothesis will change.
   (d) If $h_t(x_i) = y_i$, then $D_{t+1}(i) = \frac{D_t(i)}{Z_t} e^{-\alpha_t}$ and otherwise $D_{t+1}(i) = \frac{D_t(i)}{Z_t} e^{\alpha_t}$. This can be expressed as $D_{t+1}(i) = \frac{D_t(i) \exp(-\alpha_t y_i h_t(x_i))}{Z_t}$. $Z_t$ is some normalization factor chosen so that $D_{t+1}$ is a probability distribution.
3. The final hypothesis is $H(x) = sign(\sum_{t=1}^{T} \alpha_t h_t(x))$.

### 13.3.1 Bagging vs Boosting

Bagging techniques tend to reduce variance and increase bias, and tend to only be effective with reasonably sophisticated models. On the other hand, boosting techniques reduce bias, and work well with very simple initial models. Boosting can lead to overfitting as the number of iterations increases, though in practice this is rarely a concern.

# 14 Active Learning

In supervised learning examples covered so far, models are trained using a set of pre-labeled training data. In active learning, we assume a learner has access to a pool of unlabeled data, and can request labeling for specific points. The motivation behind active learning is that labeling can be expensive, so resources can be saved by letting an algorithm pick which subset of data points should be labeled to make good enough decisions.

In general, if a distribution can be classified perfectly by some hypothesis $h$ and $\epsilon$ is the maximum tolerated error rate, then testing $\frac{1}{\epsilon}$ training instances yields a model with the desired error rate.

## 14.1 Query Scenarios

### 14.1.1 Query Synthesis

Query synthesis involves a learner having access to some unlabeled input space or unlabeled distribution of inputs. A learner uses this to create new points and queries some "oracle" to label them.

### 14.1.2 Selective Sampling

Selective sampling involves a learner observing inputs from a continuous data stream. For each point, the learner decides to either discard the point or to label it. Unlike query synthesis, where a learner creates its own training points, selective sampling ensures training points come from the true underlying data distribution.

### 14.1.3 Pool-based Active Learning

Pool-based active learning involves a learner having access to a pool of unlabeled examples. The learner iteratively chooses the best (i.e. most informative) example from this pool and gets it labeled.

## 14.2 Selection Strategies

### 14.2.1 Uncertainty Sampling

In uncertainty sampling, a learner selects the training point that it is most uncertain about. Definitions of the "most uncertain" point can include:

- Least confident point (i.e. most likely labeling has the lowest probability).

- Point with smallest margin (i.e. smallest difference in probabilities of top 2 labelings).
- Point with maximum entropy (i.e. has most similar probabilities across all labels).

Uncertainty sampling can be carried out for pool-based active learning by querying for a set of most uncertain points, labeling them, retraining the model, and repeating. Uncertainty sampling also be used for selective sampling by defining an uncertainty threshold, and only labeling points that meet this threshold.

### 14.2.2 Query-By-Committee (QBC)

A major drawback for uncertainty sampling is that only a single classifier is used. This classifier can be overly confident about some points without being corrected.

In a query-by-committee approach, a set of classifiers (typically a small set, even just 2 classifiers) only label points for which committee members do not agree on the label. Committees can be built in a variety of ways, including using different types of training models or a set of models trained in ensemble learning.

For QBC in pool-based active learning, the level of disagreement is measured for each training point. A selection of most disagreeable points is labeled, the committee models are retained, and the process repeats. QBC can be used for selective sampling by only querying points beyond some disagreement threshold. Disagreement is measured by some entropy calculation among the votes.

Handling outliers can be a problem in QBC, since these are expected to produce disagreements even if the models are in agreement for the vast majority of points. Ways to mitigate this problem include:

- Weighting disagreement by some measure of the similarity of a point to other points in its pool. This idea behind this is that informative training examples are ones representative of their input space.
- Including the total estimated error reduction in disagreement calculations.

# 15 Unsupervised Learning

Supervised learning problems consist of training examples with corresponding labels. In unsupervised learning problems, these labels are missing. Below are some methods that can be used to analyze unlabeled data.

## 15.1 K-Means Clustering

Clustering involves partitioning data into similar groups. K-means clustering is a common clustering model that represents each cluster by a center point, and other points are assigned to their nearest cluster. The algorithm is as follows:

1. Select a desired number of clusters $K$.
2. Assume some parametric distribution for each class.
3. Initialize the parameters of the distributions.
4. Until convergence, perform the following:
   (a) Assign points to the cluster to which they most likely belong based on the parametric distributions.
   (b) Re-estimate the parametric distributions of each class based on their points.

### 15.1.1 Convergence

K-means clustering is guaranteed to converge to a local optimum, and in practice does so fairly quickly. The optimum converged upon and the rate of convergence depends on the initialization of the distribution parameters, which can be done randomly or in a carefully selected way.

### 15.1.2 Selection of $K$

$K$ is often chosen experimentally by finding a solution for different values of $K$ and then picking the one that optimized some criterion.

Let $I_k$ be the set of indices of data points within cluster $C_k$, and let $n_k$ be the number of data points in cluster $C_k$. A common measure of the quality of a clustering solution is as follows:

- $\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$.
- $\bar{x}_k = \frac{1}{n_k} \sum_{i \in I_k} x_i$.
- $W(K) = \sum_{k=1}^{K} \sum_{i \in I_k} ||x_i - \bar{x}_k||^2$, which is a measure of the similarity of the grouping of clusters.

- $B(K) = \sum_{k=1}^{K} n_k ||\bar{x}_k - \bar{x}||^2$, which is a measure of how spread apart different clusters are.

The goal is to pick a $K$ such that $W(K)$ is small and $B(K)$ is large. This can be measured by the Calinski-Harabasz index, which is defined as $CH(K) = \frac{B(K)/(K-1)}{W(K)/(N-K)}$. Values of $K$ can be increased until some upper limit, and then the solution with the highest Calinski-Harabasz index is selected.

## 15.2 Hierarchical Clustering

Hierarchical clustering is a clustering algorithm in which higher-level clusters are formed by merging clusters at lower levels. Clusters can be built using either a bottom-up approach, where smaller clusters are repeatedly merged, or a top-down approach, where larger clusters are repeatedly split.

A hierarchical clustering solution can be visualized as a dendrogram, which is a type of tree in which each node represents a cluster. Leaves contain a single point, the root contains the entire dataset, and internal nodes point to the subgroups from which form them. Internal nodes are drawn at a height that reflects the dissimilarity between their children.

To measure dissimilarity between groups $G$ and $H$, some linkage criterion $d(G, H)$ is used. Common linkage functions include:

- Single linkage (i.e. distance to nearest neighbor).
- Complete linkage (i.e. distance to furthest neighbor). This is less sensitive to noise and outliers than single linkage.
- Average linkage (i.e. average distance between points in opposite groups).

## 15.3 Principal Component Analysis (PCA)

A data set can be simplified by removing redundant dimensions. PCA does this by finding a vector that points in the direction of maximum variance, which is done by finding a fixed-size subset of orthonormal basis vectors that can express the original data set with maximum accuracy. This is equivalent to the following:

1. For each dimension, subtract the mean dimension value from each point (i.e. points should be normalized at 0).
2. Calculate the covariance matrix.
3. Calculate the eigenvectors and eigenvalues of the covariance matrix.
4. To form a $k$-dimensional orthonormal basis, select $k$ eigenvectors with the largest eigenvalues.
5. Construct a new dataset by expressing each point using the new basis.

# 16 Semi-Supervised Learning

Semi-supervised learning problem have a combination of labeled and unlabeled training points, with most points typically being unlabeled. The idea behind semi-supervised learning is similar to the idea behind active learning; labels are often costly to obtain, and the marginal performance improvement of adding labels decreases as more labels are added.

For the examples below, $X_l$ represents labeled training points, $Y_l$ represents their corresponding labels, and $X_u$ represents unlabeled points.

## 16.1 Self-Training

Self-training methods follow the procedure below:

1. Select some classification method.
2. Train a classifier $f$ using $X_l$ and $Y_l$.
3. Use $f$ to classify points in $X_u$.
4. Pick the $k$ points $x \in X_u$ with the highest labeling confidence.
5. Add these $k$ points and their labels to $X_l$ and $Y_l$.
6. Repeat.

Self-training is simple and can be applied to a wide variety of learning algorithms. However, early mistakes are difficult to correct, which may occur if the initial labeled points are not from well-separated clusters.

## 16.2 Co-Training

Co-training involves training two separate classifiers on a disjoint set of features, and then the two classifiers take turns training one another. The co-training procedure is as follows:

1. Train two classifiers, $f^1$ on $X_l^1, Y_l$ and $f^2$ on $X_l^2$ and $Y_l$.
2. Classify $X_u$ using $f^1$ and $f^2$ separately.
3. Add the $k$ most confident points in $f^1$ to $f^2$'s labeled data.
4. Add the $k$ most confident points in $f^2$ to $f^1$'s labeled data.
5. Repeat.

Co-training is less sensitive to mistakes than self-training, and supports similar flexibility in the choice of learning algorithms. However, not all data sets have a conditionally independent split of features, and even when they do, classifiers that use the entire feature set may perform better then the co-training classifier.

## 16.3 Gaussian Mixture Models (GMMs)

The underlying assumption of mixture models is that a dataset was generated by some unknown set of mixture components. Gaussian mixture models (GMMs) assume that these mixture components are Gaussian distributions, with data generated by selecting some mixture component $y \sim p(y)$ and then generating $x$ based on the mixture component as $x \sim p(x|y = k) = N(x; u_k, \sigma_k^2)$, with the joint distribution being $p(x, y) = p(y)p(x|y)$. The probability density function $p(x) = \sum_y p(y)p(x|y) = \sum_{k=1}^{K} p(y = k)p(x|y = k)$. The goal is to estimate parameters $p(y = k)$ (often denoted as $\pi_k$), $u_k$, and $\sigma_k^2$.

For machine learning purposes, each of the $K$ classes can be treated as a mixture component. Unlabeled data describes how instances from all classes together are distributed. Determining the distributions of each class can be used to classify unlabeled points.

This problem boils down to finding the MLE for $p(D|\theta)$, where $\theta$ represents the parameters. $p(D|\theta)$ is as follows:

- With labeled data: $\log \prod_{i=1}^{l} p(y_i|\theta)p(x_i|y_i, \theta)$.
- With labeled and unlabeled data: $\log \prod_{i=1}^{l} p(y_i|\theta)p(x_i|y_i, \theta) \prod_{i=l+1}^{l+u} \sum_{k=1}^{K} p(y_i = k|\theta)p(x_i|y_i, \theta)$.

Since this model contains unobservable values, it is necessary to make a guess on how the data looks to run the MLE algorithm. We repeatedly "complete" the data by assigning labels to the unlabeled points based on the current distributions, and then compute the MLE based on this completed data. This process is guaranteed to make the MLE function converge to a locale minimum.

## 16.4 Other Methods

Some other semi-supervised learning techniques include:

- **Graph-based Methods**, which involve constructing graphs with edges between related points.
- **Semi-Supervised SVM**, which involves finding the labelling of $X_u$ that results in an SVM with the largest margin.

# 17 Neural Networks

Neural networks are a kind of framework for combining machine learning algorithms together. These networks consist of nodes called neurons that receive input from other neurons, manipulate it, and propagate it to further neurons.

## 17.1 Feed Forward Neural Networks

Feed forward neural networks consist of layer of neurons. Layer 0 is the input layer, and simply copies the input. The final layer provides the final output. Layers in between propagate values only to the next layer (i.e. there is no cross-connection between neurons in the same layer, and there are no backward connections). A fully connected network is one in which all neurons in layer $k$ provide input to all neurons in layer $k+1$.

$x_i$ denotes the vector of inputs entering unit (i.e. neuron) $i$, $w_i$ denotes the vector of weights entering unit $i$, and $o_i$ denotes the output of unit $i$. Generally speaking, a sigmoid activation is used to determine the output (i.e. $o_i = \sigma(w_i \cdot x_i)$, where $\sigma = \frac{1}{1+e^{-x}}$). These formulas can be used to compute the output of a feed forward neural network layer-by-layer.

### 17.1.1 Learning

The goal of learning in a neural network is to find a set of weights that minimize the error at the network's last layer (i.e. the output). An advantage of using the sigmoid function to compute neuron outputs is that it is differentiable, so approaches based on gradient descent can be applied. The process is as follows:

1. Select initial values for weights.
2. Let $o$ be the output of the neural network (let this be denoted by node $k$) for a selected training example (assuming a single output node). For the output unit, the adjustment is $\delta_k o(1-o)(y-o)$.
3. For each hidden (i.e. inner) unit $h$, the adjustment is $\delta_h = o_h(1-o_h)w_{k,h}\delta_k$.
4. Update each network weight by adding the adjustment multiplied by the input $x$ to the neuron and some learning rate $\alpha$.
5. Repeat steps 2-4 until the gradient descent process converges.

This gradient descent can be applied on a single example at a time, or through batch gradient descent, where all (or some subset, which is known as mini-batch gradient descent) the training examples are looped through to accumulate weight changes, and only then the weights are updated. Note that since this is a gradient descent algorithm, its convergence is to a local minimum.

Neural networks are often very sensitive to changes in learning rate, and the learning rate can be difficult to tune by hand. Learning rates often differ between layers, and it is often good to start with a higher learning rate and then decrease it later during training.

### 17.1.2 Depth of Network

A single sigmoid neuron has the same representation power as a perceptron. A neural network with a single hidden layer can represent every boolean function, and all bounded continuous functions can in turn be approximated using boolean functions to an arbitrary level of precision. A network with two hidden layers can approximate any function to an arbitrary level of precision. There is no theoretical reason for a neural network to have more than 2 layers, and 1 hidden layer is sufficient for most problems.

Note that deeper neural networks are often more difficult to train. This is because the gradient adjustment is multiplied by the weight in hidden layers of the network, and since the weight is typically between $-1$ and 1, repeated multiplications tend to decrease the magnitude of the adjustment.

### 17.1.3 Input Encoding

Discrete inputs are typically encoding using "1-hot" or "1-of-k" encoding. There is a one-to-one mapping between possible inputs and input bits. Only a single input bit can be set to 1, and the $k$th bit being set to 1 encodes the $k$th input.

Real-valued inputs typically need to be normalized (e.g. set mean to 0 and set variance to a desired value). Some discretized approximations include 1-to-n encoding, where the input is discretized into $n$ intervals, and thermometer encoding, which is similar to 1-to-n but if a variable falls in the $i$th interval, all bits from 1 to $i$ are set to 1 instead of just $i$. Thermometer encoding tends to perform better than 1-to-n encoding.

### 17.1.4 Output Encoding

Multiple output units can be used to output values for different classes. This allows for training a multi-class classifier instead of a bunch of 1-vs-all classifiers. Another use for multiple outputs units is to encode different ranges of the output, which can be useful when using neural networks for regression problems.

### 17.1.5 Overfitting and Underfitting

Networks with many hidden features are prone to overfitting, while networks without many hidden features are susceptible to underfitting. Overtraining is a neural network phenomenon related to overfitting, and describes how a network tends to have more and more active (i.e. high weight) nodes as training progresses.

Two common techniques to mitigate overfitting in neural networks are:

- **Regularization**, which can help keep weights low by incorporating some penalty term that prefers simple models.
- **Dropout**, which involves dropping nodes at random in each training iteration. These nodes are all combined later on, which can yield effects similar to that of ensemble learning.

## 17.2 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a class of neural networks designed to:

- Handle very high-dimensional inputs.
- Exploit 2D and 3D topologies.
- Be invariant to certain types of transformations such as translations and illuminations.

These properties make CNNs well tailored to analyzing visual data such as pictures. In such applications, each input and output is a 3D tensor (i.e. a 2D image plus a set of colors such as RGB). To achieve the above properties, CNNs apply the following ideas:

- **Local connectivity**. A hidden unit is connected to some local section of an image.
- **Pooling units**. Pooling involves combining adjacent layers exploiting local connectivity to find patterns across larger spatial areas. CNNs typically alternate between convolutional, pooling, and fully connected layers. This can make them much deeper than the 2-hidden-layer theoretical bound discussed earlier, though this deep learning approach seems to work well in practice (though theoretical understanding of why is still limited).

## 17.3 Recurrent Neural Networks

In datasets that are sequences, it is important to capture the ordering of the inputs. Recurrent neural networks (RNN) are useful for dealing with sequences of data because their outputs can be fed back to their inputs, allowing them to maintain an internal memory based on previous inputs.

Although an RNN can be unrolled over time to generate a feed forward neural net, RNNs can handle varying input lengths, and generate a longer feed forward neural net for longer input sequences. RNNs support arbitrary topologies (i.e. any node can feed into any other node).

### 17.3.1 Input Encoding

The most common way to model inputs in RNNs is to specify the states of a subset of nodes (i.e. the input nodes) for each step. For example, if modeling a sequence of words, the first step corresponds to the first word, the second step corresponds to the second word, and so forth.

### 17.3.2 Output Encoding

The output can be represented as the final activation of a subset of units, or by the activation of this subset of units over several steps (useful if treating the output as a sequence).

### 17.3.3 Learning

Training an RNN is equivalent to training its unrolled feed forward neural net. Note that the feed forward neural nets can have repeated weights. The training procedure is as follows:

1. Compute the activations of all nodes.
2. Compute error derivatives at each step of the unrolled RNN.
3. Add together the derivatives for each weight, and adjust the weights accordingly.

In general, RNNs are considered difficult to train. Setting initial states can be challenging, and training gradients often become very small or very small. Also, although RNNs can maintain internal memory, they struggle to hold onto it very long.

### 17.3.4 Long Short Term Memory

Long short term memory (LSTM) is a node architecture for RNNs that overcomes many of the above challenges. Instead of each node having standard input and output edges, each node has a:

- **Write gate**, which if activated, allows information to enter the node.
- **Keep gate**, which if activated, allows the incoming information to be stored in the node.
- **Read gate**, which if activated, allows information to be read the node.

The introduction of a keep gate allows for better modeling of long term memory dependencies. Each gate is controlled by a separate activation vector.

## 17.4 Generative Adversarial Networks

So far, we discussed how neural networks can be used for classification and regression problems. However, neural networks can also be used to generate data.

The goal of generative adversarial networks is to generate data similar to example input data. A generative network produces data, and passes it as input into a separate discriminative neural network which accepts the data if it resembles the examples, and otherwise rejects it. These two networks complete is a zero sum game; the discriminator tries to maximize its recognition of that actual examples come from some underlying distribution and minimize the change of accepting "fake" examples produced by the generative network, while the generative network tries to do the opposite.

# 18 Structured Prediction

Some machine learning problems have a very large number of correlated outputs. For example, labelling parts of speech results in a separate label for each word, and these labels are dependent on one another.

Structured prediction models are designed to predict such "structured" objects with many outputs. This section focuses on score-based structured prediction. For some scoring function $s$, let $x$ be an input, let $\hat{y}$ be the predicted output, and let $y$ be the actual output. $s(x, \hat{y})$ measures the compatibility between $x$ and $\hat{y}$ (i.e. "better" predictions should score higher), and $s(x, y)$ should produce the maximum value.

## 18.1 Structured Perceptron

In multi-class perceptron problems, we define a feature function $\theta(x, y)$, which takes in an input $x$, and output $y$, and outputs some describing how compatible $x$ is with $y$. For some vector $w$, we define $s(x, y) = w^T\theta(x, y)$ (i.e. $w$ determines how to weigh the compatibility features). The goal is to learn a $w$ such that the correct output maximizes the scoring function.

The multi-class perceptron algorithm runs as follows:

1. Initialize $w$.
2. For each example $x$:
   (a) Find a label $\hat{y}$ that maximizes the score.
   (b) If the prediction is correct (i.e. $\hat{y} = y$), continue.
   (c) If the prediction is incorrect, we update $w$ so that the prediction becomes correct (i.e. $w = w + \theta(x, \hat{y}) - \theta(x, y)$).

Structured perceptron is very similar to multi-class perception, but finding a label that maximizes the score can be intractable. The "optimal" label can be found by making approximations or by applying some problem-specific search algorithms.

## 18.2 Structured SVM

SVMs support complex loss functions. Structured SVM problems typically use a variant of Hamming loss $l(y, \hat{y})$, which is the fraction of output labels that are incorrectly predicted. Since $s(x, y)$ is supposed to be larger than any other $s(x, \hat{y})$, we add the constraint $s(x, y) - s(x, \hat{y}) \geq l(y, \hat{y}) - \epsilon$, where $\epsilon$ is some slack variable allowing for points to be on the wrong side of the decision margin.

However, we end up with a different slack variable for every possible output, which can make the SVM optimization problem intractable. Instead of stating that $s(x, y) - s(x, \hat{y}) \geq$

$l(y, \hat{y}) - \epsilon$ must apply to all $\hat{y}$, we can say it needs only apply to the most confusing imposter (i.e. the $\hat{y} \neq y$ with a maximum $s(x, \hat{y}) + l(y, \hat{y})$). The weights of the resulting optimization problem can be minimized using a gradient descent approach.

# 19 Reinforcement Learning

Reinforcement learning deals with learning what actions to take in an environment to maximize some notion of reward. The learner is not explicitly told what actions to take, and determines the quality of actions by trying them out to measure their reward.

The learner is provided with a set of states $S$, a set of actions $A$ to transition between these states, and a set of reinforcement signals $R$. For some constant $0 < \gamma < 1$, the goal of the learner is to choose actions such that $r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots$ is maximized. The $\gamma$ term is known as the discount rate, and represents how an award becomes devalued as time progresses.

## 19.1 Markov Decision Processes

Markov chains are used to model a sequence of states where the probability of the next state only depends on the current state (i.e. the next state depends on where you are, but not how you got there). Let $p_{ij}$ be the probability of a transition to state $s_j$ from state $s_i$. Let $U(s_i)$ be the expected discounted sum of future rewards starting at $s_i$.

$$U(s_i) = \gamma(p_{i1}U(s_1) + p_{i2}U(s_2) + \cdots + p_{in}U(s_n))$$

For a probability matrix $P$ and a reward vector $R$, this has a closed form solution of $U = (I - \gamma P)^{-1}R$.

While the above formula has a closed-form solution, it involves solving an $n$ by $n$ series of equations, which can be very computationally expensive for large values of $n$. As an alternative, a value iteration method can be used. For this technique, let $U^k(s_i)$ be the expected discounted sum of awards over the next $k$ time steps.

$$U^{k+1}(s_i) = r_i + \gamma(p_{i1}U^k(s_1) + p_{i2}U^k(s_2) + \cdots + p_{in}U^k(s_n))$$

In the iterative method, the above calculation is repeated until $|U^{k+1}(s) - U^k(s)|$ falls below some margin $\epsilon$. While the solution is an approximation, it tends to be much faster than the closed form solution for large values of $n$.

## 19.2 Passive Learning

In passive reinforcement learning, an agent is provided with a fixed decision policy to apply. The actions to take can be found using the Markov decision process techniques described above.

## 19.3 Active Learning

In an active learning approach, an agent needs to explore its environment to determine the decision policy to apply. The agent records the rewards of actions in a Q-table, which it can reference later to determine which move is the best.

# 20 Learning by Demonstration

Learning by demonstration, often referred to as imitation learning, apprenticeship learning, or programming by example, refers to teaching a computer new behavior by providing it with an example of this behavior. Demonstrations are used as training data, and a machine uses these demonstrations to come up with a good decision policy. Learning by demonstration is often used for sequential decisions where one decision influences the next, such as in driving a car.

## 20.1 Behavioral Cloning

The goal of behavioral cloning is to learn a policy $\pi$ such that an agent takes actions that minimize some notion of expected loss $E$. For an example with states $S$, actions $A$, and losses $L$, we collect all (state, action, loss) pairs to train a multiclass classifier. This reduces learning by demonstration to a standard supervised learning problem.

This simple approach runs into some challenges, including:

- Learning errors can cascade.
- Learner is only trained over familiar examples, and this may not generalize well.

Below are a few techniques that try to overcome this limitation.

### 20.1.1 Forward Training Algorithm

The forward training algorithm involves learning a separate policy for each step. This allows for each policy to be trained on a distribution of states actually encountered, but is slow and impractical for a large number of steps.

### 20.1.2 SMILE

SMILE stands for Stochastic Mixing Iterative Learning. This algorithm is similar to the forward training algorithm in that it has a separate policy for each step, but instead of only relying on an expert's policy, the training dataset is based on the trajectories induced by the policy of the previous step. Each trained policy is added with some geometric discount factor, so each step policy ends up being a combination of policies.

### 20.1.3 DAgger

Dagger stands for Dataset Aggregation, and trains a single policy using an aggregation of data generated by the expert and data generated by previous policies. This builds a set of

inputs the final policy is likely to encounter based on previous experience. The high-level algorithm is:

- Execute the current policy $\pi_i$.
- Observe states of execution during this policy, and query the expert for actions in those states.
- Add the newly obtained data to a dataset $D$.
- Train the next policy $\pi_{i+1}$ on dataset $D$.

## 20.2 Inverse Reinforcement Learning

Behavioral cloning is useful for learning the actions to take to perform a task, but is not good for learning the objective of the expert. In standard reinforcement learning, we are provided with a reward function, and we find an action policy to maximize expected reward. In inverse reinforcement learning, we are provided with expert actions, and use them to try to determine the reward function. After learning the reward function, a regular reinforcement learning algorithm can be applied to find a good policy.