COMP9417 Final Exam Notes

These notes cover the topics tested in the UNSW COMP9417 final exam for 2013. Using them, I earned a final exam mark of 91% (2nd in the cohort). I can't guarantee that they're correct, and they're certainly not complete. They cover no more than what is covered in the course. Use them at your own risk .

Daniel Goldbach, 23 June 2013.

Note that **instance-based learning is not covered in these notes!** Because it's really easy.

Algorithm Independent Learning

Algorithm independent learning looks at **computational learning theory**: the study of learning in general as a theoretical process, focusing on its limitations.

Terminology

- X is the space of instances, containing all possible instances.

- C is the space of all possible concepts. $c : X \to \{0, 1\}$ is a concept. $c(x) = 1$ iff an instance x is a positive example of a concept c.

- Instances are generated at random from X according to a distribution . does not change over time.

- The learner L considers a set of hypotheses H, and attempts to learn the hypothesis $h \in H$ that best approximates c -- that is, the hypothesis $h : X \to \{0, 1\}$ that maximises $\Pr_{x \in}[c(x) = h(x)]$ The performance of L is measured based on new instances randomly drawn from X according to .

- The **version space** $VS_{H,D}$ is the set of all hypotheses $h \in H$ that correctly classify the training examples D. Formally, $VS_{H,D} = \{h \in H \mid (\forall \langle x, c(x) \rangle \in D\ (h(x) = c(x))\}$.

Syllabus Dotpoints

Describe a basic theoretical framework for sample complexity of learning

Sample complexity

> The number of training examples required for the learner to learn (with high probability) a concept -- to converge (with high probability) to a successful hypothesis.
>
> Alternatively, the growth in the number of required training examples with problem size.

Define training error and true error

Training error

The training error error_{D} is the fraction of training examples misclassified by the learner.

True error

The true error error is the probability that L will misclassify a random instance picked from X according to -- that is, $\mathrm{error} \equiv \Pr_{x\in} [c(x) \neq h(x)]$.

Define the Probably Approximately Correct (PAC) learning framework

PAC learning

We want the learner to be able to get 100% accuracy on every set of instances. However, this is unrealistic for two main reasons:

1. The training set available to the learner cannot possibly be complete, so there will almast always be multiple hypotheses in the version space, with no way of determining which is correct.
2. There is always some probability of the training set being awful, and there being *no* way to form any meaningful hypotheses.

To accomodate for (1), we allow the learner to make mistakes, as long as those mistakes are small (the learner is *approximately* correct). Furthermore, to accomodate for (2), we recognise that we can only guarantee that the mistakes are small with some probability (the learner is *probably approximately* correct).

PAC-learnability

A concept class C is PAC-learnable by L if for any $c \in C$, L always outputs a hypothesis h so that there is a $(1 - \delta)$ chance that $\mathrm{error} (h) \leq \epsilon$, in an amount of time that is polynomial in proportion to $1/\epsilon, 1/\delta, |X|$ and the size of the encoding of c.

It is worth noting that there is no guarantee that an h exists that arbitrarily approximates c (for example, where c is the concept *face is female* given an input image of a human face), except where H is the power set of X.

Consistent and agnostic learners

A learner is **consistent** if, when possible, it outputs a hypothesis that perfectly fits the training data (i.e. a hypothesis in the version space).

To bound the sample complexity of a consistent learner, we need only bound the number of examples needed to ensure that all hypotheses in the version space are acceptable. Since our definition of PAC-learnability defined an *acceptable* hypothesis as one with error less than ϵ, we say that a version space is **ϵ-exhausted**.

ϵ-exhaustion

$VS_{H,D}$ is ϵ-exhausted iff $(\forall h \in VS_{H,D})\ \mathrm{error} (h) < \epsilon$.

It can be proved that if m instances are drawn randomly from X according to , then the probability that the version space is **not** ϵ-exhausted decreases exponentially and is less than $|H|e^{-\epsilon m}$. For this probability to be less than or equal to δ as we require for PAC-learnability, we rearrange to get

$$m \geq \frac{1}{\epsilon}(\ln|H| + \ln(\frac{1}{\delta}))$$

This is a bound that determines the minimum number of training examples for a consistent learner to PAC-learn a concept in C.

Agnostic learner

> A learner is **agnostic** if it does not expect there to be a hypothesis with 0 training error, and tries only to minimise the training error. An agnostic error may output a hypothesis h that is not in the version space $VS_{H,D}$.

Learnability extends to agnostic learners; just require that the chosen hypothesis have error no more than $\epsilon + \text{error}_D(h_{best})$, where h_{best} is the hypothesis that minimises $\text{error}_D(h)$. For an agnostic learner, we obtain a similar bound on m:

$$m \geq \frac{1}{2\epsilon^2}(\ln|H| + \ln(\frac{1}{\delta})).$$

A reminder that C is PAC-learnable iff m grows polynomially with $\frac{1}{\epsilon}, \frac{1}{\delta}$ and $|H|$.

The concept class of boolean literals e.g. $(a \wedge b) \vee \neg c$ is PAC-learnable. To see this, substitute in $|H| = 3^n$ (since each variable can either be included, included in negation, or excluded).

Define the Vapnik-Chervonenkis (VC) dimension framework

When the hypothesis space is infinitely large, we cannot bound the sample complexity using the formula above. In cases like these, we can use the VC dimension. The VC dimension also often provides a tighter bound on m.

Shattering

> A set of instances X is **shattered** by a hypothesis space H if every dichotomy of X (i.e. all $2^{|X|}$ subsets of X) are covered by a hypothesis in H.

Intuitively, a hypothesis space that shatters the instance space is more expressive than one that does not. We might also say that a hypothesis that shatters a larger subset of the instance space is more expressive than one that shatters a smaller subset of the instance space.

If a hypothesis shatters X, then H is capable of representing **every concept on X**. The VC dimension is simply a measure of the size of the largest finite subset of X shattered by H.

Vapnik-Chervonenkis dimension

> The VC dimension $VC(H)$ of H is the size of the largest finite subset of X shattered by H. $VC(H) = \infty$ if the size of the largest subset is not finite.

For instance, consider the problem where H is the power set of X and $|X|$ is finite. H is capable of expressing any dichotomy of X, so $VC(H) = |X|$.

Consider now X being points on the x-axis and H being the set of all intervals along the x-axis. Any set of two instances of X can be shattered. However, no set of three instances of X can be shattered, because you can't isolate the two endpoints without isolating the middle point. The VC dimension of this hypothesis space is thus 2. The VC dimension of linear decision surfaces in r dimensions is $r + 1$.

VC gives us another bound on m as follows:

$$m \geq \frac{1}{\epsilon}\left(4\lg\left(\frac{2}{\delta}\right) + 8VC(H)\lg\left(\frac{13}{\epsilon}\right)\right)$$

We can also obtain an equivalent *lower bound* on m; that is, any PAC-learner must receive more instances than this in order to learn:

$$m \geq \max\left[\frac{1}{\epsilon}\log\left(\frac{1}{\delta}\right), \frac{VC(C) - 1}{32\epsilon}\right]$$

Define the Mistake Bounds framework

Instead of considering sample complexity, consider instead the number of incorrect 'guesses' the learner must make while it learns. Each time the learner is presented with a training example, it must give its 'guess', then it receives the correct classification. We seek to bound the number of incorrect guesses it makes.

Optimal mistake bound

> Define $M_A(C)$ to be the maximum number of mistakes made by learning algorithm A in learning a concept in C, for all possible training sequences. The **optimal mistake bound** $Opt(C)$ is the minimum value of $M_A(C)$ over all A.
>
> $$VC(C) \leq Opt(C) \leq M_{Halving}(C) \leq \lg(|C|)$$

Halving algorithm

The halving algorithm is an extension of the candidate-elimination concept learning algorithm. At each stage, it predicts by taking the majority vote of all the hypotheses in its version space.

- If the majority vote is incorrect, we make a mistake and exclude at least half the hypothesis space.
- If the majority vote is correct, no mistake is made and we proceed.

We can bound its number of mistakes by $\lg(|C|)$.

Weighted majority

The weighted majority algorithm is a generalisation of the halving algorithm. It uses a *weighted vote* as opposed to a uniform vote across a set of predicting algorithms, and learns weights by altering them as it goes.

When an algorithm performs badly, its weight reduces. When an algorithm performs well, its weight increases.

$\mathrm{Opt}(C)$ for a weighted majority algorithm is in $O(\mathrm{Opt}(C))$, where this second $\mathrm{Opt}(C)$ corresponds to the 'best' predictor in its ensemble.

It can be easily implemented by starting each weight at 1, then multiplying the weight of each incorrect predictor by some constant β.

Apply basic analysis of learning in each of PAC, VC & Mistake Bounds frameworks

See above.

Outline the No Free Lunch Theorem

The No Free Lunch (NFL) theorem asserts that no learning algorithm is 'better' than another learning algorithm across all possible target functions. In fact, averaged across all possible target functions, the true error of all learning algorithm is 0.

To see why this is so, consider two hypotheses over n boolean variables. Say the first has a true error of 0.4 and the second has true error of 0.6 for some target function $c(x)$. However, there are 2^{2^n} possible concepts, and for every target function, there is exactly one 'inverse' target function; so there exists some $c^{-1}(x)$ where the true errors are 0.6 and 0.4 respectively.

This is a sort of 'conservation' principle: if we do well over some set of target functions, we must do equally badly on some other set. It is the *assumptions* we make about our domain that allow us to say that one algorithm is better than another.

This is a strange notion, but it is at the core of machine learning and you should understand it. Here's another explanation from a different angle.

Mary likes Twilight and Dracula. Kate likes Twilight and Dracula. Susan likes Twilight. Is Susan likely to like Dracula?

You may be inclined to say *yes* -- but why? The existence of a pattern does not make the presence of that pattern likely in the future. Your innate sense that Susan should like Dracula is a product of your *inductive bias*.

The NFL theoreom says that there's no such thing as "free learning". Even in a system with no noise, mathematically you **cannot** extrapolate from what you've seen. Each model has its own form of inductive bias: decision trees prefer short hypotheses, perceptrons only predict linear decision boundaries, etc. But no inductive bias is innately *better* than any other across all target concepts, and thus all models are equal when averaged across all target concepts.

Ugly Duckling theorem

The ugly duckling theorem asserts that any notion of similarity between two patterns is a result of bias. It is named this because it states that, all things being equal, an ugly duckling is just as similar as a swan as two swans are to each other.

Here, we think of the *similarity* of two instances as *the number of target functions in which both instances are positive*. Alternatively, the *number of concepts shared by both instances*.

To expand on the analogy: we might say swan 1 is more similar to swan 2 than swan 1 is to a duck, because both swans are white but the duck is grey; both swans are graceful but the duck is awkward; both swans have long necks but the duck has a short neck; etc. Each of these is a *concept* shared by the two swans and excluded by the duck. However, in the space of *all possible concepts*, there are just as many that contain both the duck and the swan as there are that contain the two swans. Of course, it is difficult to speak about this in English terms.

Describe the framework of bias-variance decomposition

Bias-variance decomposition refers to the decomposition of true error into two components:

- Error that is due to the training data used (variance)
- Error that is simply due to not learning the model (bias)

Say you overfit a model. The training error will be very small, but the true error will be large. Here, the true error is large because the variance is large.

Say you underfit a model. Both the training error and the true error will be large. Here, the true error is large because the bias is large.

To decide if bias or variance are the issue, consider choosing a new training set from a different distribution.

- If the model changes, your error is due to high variance.
- If the model barely changes at all, your error is due to high bias.

Association Rule Learning

Association rule learning (ARL) involves finding patterns of items that commonly occur together in transactions -- for example, learning that butter and bread are often bought together.

Terminology

- An **itemset** is a set of items.

- A **transaction** is a tuple of an itemset and a transaction ID. Itemsets are abstract entities whereas transactions are usually concrete instances.

- A **transaction database** is a set of transactions.

- The **cover** of an itemset X in a transaction database   is the set of transaction identifiers of the transactions whose itemsets are supersets of X:

  $$\mathrm{cover}(X, \quad) = \{\mathrm{tid} \mid (\mathrm{tid}, I) \in \quad, X \subseteq I\}$$

- The **support** of an itemset X is the number of transactions in the cover of X in .

- An itemset is **frequent** in   if it exceeds some minimum support threshold in .

Syllabus Dotpoints

Contrast supervised vs. unsupervised learning

In supervised learning, instances are labelled with their classes or "correct answers". The task is to learn to predict the "correct answer" for an unseen instance.

In unsupervised learning, instances are not labelled with classes. The task is to group instances into classes based on similarities and patterns between them.

Define association rules

Association rules are similar to classification rules, and are of the form

$$\mathrm{antecedent} \rightarrow \mathrm{consequent}$$

meaning *"the presence of antecedent implies the presence of the consequent in an itemset"*. The antecedent and consequent are both itemsets.

- The **support** of an association rule $X \rightarrow Y$in   is the number of transactions in which all the items in $X \cup Y$are found, as a fraction of the number of transactions.

- The **confidence** of an association rule $X \rightarrow Y$in   is the number of transactions in which all the items in $X \cup Y$are found, as a fraction of the number of transactions in which all the items of X are found.

  Equivalently, $\mathrm{confidence}(X \rightarrow Y) = P(Y \mid X)$.

- The **lift** of an association rule $X \rightarrow Y$indicates how much the support of the rule deviates from the expected support of the rule if there was no correlation between the presence of X and the presence of y.

$$\text{lift}(X \to Y) = \frac{\text{support}(X \cup Y)}{\text{support}(X) \times \text{support}(Y)}$$

Reproduce the basic association rule discovery algorithm

Most ARL algorithms involve the observation that if we wish to generate an association rule $X \to Y$, we can first identify a frequent itemset $X \cup Y$, then generate all rules in the power set of $X \cup Y$and pick only those whose confidence is above some minimum threshold. We thus seek an algorithm to find frequent itemsets.

Apriori algorithm

The Apriori algorithm revolves around the following observation:

Apriori's *anti-monotone* heuristic / Downward closure lemma:

> **If an itemset of size k is infrequent, a superset of size $k + 1$ will also be infrequent.**

We can thus generate frequent itemsets as follows:

```
1. $S \leftarrow$ a set of itemsets where each itemset contains one item
2. Do forever:
    1. $S \leftarrow S -$ all infrequent itemsets in $S$
    2. If $S$ contains too few items, end.
    3. $S' \leftarrow \{\}$
    4. For each itemset $I$ in $S$:
        1. For each item $c$ in the schema not already in $I$:
            1. Add $c$ to $S'$
    5. $S \leftarrow S'$
```

So we start off by only looking at itemsets of one item each, and remove all infrequent itemsets. Then we consider all pairs of frequent items, and remove all infrequent itemsets. Then we consider all triples, etc.

Itemsets to association rules

It is clear that $\text{confidence}(A, B \to C) \geq \text{confidence}(A \to B, C)$, since the support of both rules is the same (therefore the denominators are the same) and the numerator of the lefthand side is $\text{support}(C) \geq \text{support}(B, C)$ which is the numerator of the righthand side. In a sense, the antecedent $A \cup B$is **more general than** the antecedent A (or B).

The converse of this is that if $X \to Y$does not have minimum confidence, then the rule we get by moving items from X into Y will also not have minimum confidence. This is similar in nature to the downward closure lemma above.

We thus stumble upon an algorithm to generate rules with minimum confidence from frequent itemsets. We start with rules with *single-consequent* rules (rules with one item in the consequent), remove all that are below the minimum support, then move a term from the antecedent to the consequent, the remove those with too-small confidence, ...

```
1. rules $\longleftarrow \{\}$
2. For each itemset $I$:
    1. cur_rules $\longleftarrow$ all single-consequent rules
    2. next_rules $\longleftarrow \{\}$
    3. For each rule $r \in$ cur_rules:
        1. If $r$ has over the minimum confidence:
            1. add $r$ to rules
            2. For each item $i \in \mathrm{ante}(r)$ :
                1. $r' \longleftarrow r$
                2. Move $i$ from the antecedent of $r'$ to the consequent of $r'$
                3. Add $r'$ to next_rules
        2. cur_rules $\longleftarrow$ next_rules
```

Bayesian Learning

Bayesian learning is a field which revolves around the use of *Bayes' theorem* and related probability concepts in machine learning.

Syllabus Dotpoints

Reproduce basic definitions of useful probabilities

Bayes Theorem

For a hypothesis h and a dataset D,

$$P(h \mid D) = \frac{P(D \mid h)P(h)}{P(D)}$$

Prior

The probability $P(h)$; that is, the probability of the hypothesis *prior* to us being given the dataset we should be considering.

Note that if "prior probability of the dataset" is specified, this refers to $P(d)$.

Posterior

The posterior probability $P(h \mid D)$ -- what we *now know* to be the probability of h, given our knowledge of the dataset. Bayes' rule allows us to calculate the posterior probability.

***Maximum a priori* (MAP) hypothesis**

The hypothesis that maximises the posterior probability $P(h \mid D)$.

Maximum likelihood (ML) hypothesis

The hypothesis that maximises the probability $P(D \mid h)$. Note that this is the same as the MAP hypothesis if the priors are all equally likely.

Derive Bayes theorem and the formulae for MAP and ML hypotheses

Deriving Bayes' theorem

$$\begin{aligned} P(a \mid b) &= P(a \wedge b) \cdot P(b) \\ &= P(b \wedge a) \cdot P(a) \\ &= \frac{P(b \mid a)}{P(a)} \cdot P(a) \\ &= \frac{P(b \mid a)P(a)}{P(b)} \end{aligned}$$

Deriving the MAP and ML hypotheses

$$\begin{aligned} h_{MAP} &\equiv \underset{h \in H}{\arg\max}\ P(h \mid D) \\ &= \underset{h \in H}{\arg\max}\ \frac{P(h \mid D)}{P(D)} \\ &= \underset{h \in H}{\arg\max}\ P(D \mid h)P(h) \end{aligned}$$

Noting that $P(D)$ is constant when we're considering different hypotheses over a single dataset.

$$h_{ML} \equiv \underset{h \in H}{\arg\max}\ P(D \mid h)$$

Describe concept learning in Bayesian terms

Concept learning by a consistent learner is equivalent to finding a MAP hypothesis. A consistent learner is one who outputs a hypothesis with 0 classification error.

We can see this by deriving $P(h \mid D)$ as

$$P(h \mid D) = \begin{cases} \frac{1}{|VS_{H,D}|} & \text{h is consistent with D} \\ 0 & \text{otherwise} \end{cases}$$

(this follows from the assumption that all hypotheses are equally likely and that $P(D \mid h)$ is 1 iff D is consistent with h and 0 otherwise)

This leads to the surprising conclusion that concept learning outputs a MAP hypothesis, even though it never explicitly deals in probabilities.

Outline the derivation of the method of numerical prediction by minimising the sum of squared errors in terms of maximum likelihood

Consider

$$h_{ML} \equiv \underset{h \in h}{\arg\max}\ P(D \mid h)$$

First, we note that $P(D \mid h) = \prod_{i=1}^{m} P(d_i \mid h)$, where d_i is the ith instance in D. Since the error of any instance will follow a normal distribution, $P(d_i \mid h)$ can be written as a normal distribution with variance σ^2 and mean $\mu = f(x_i)$, where f is the target function.

Since maximising a value is the same as maximising its logarithm, we take the log of the right hand side. This transforms the product into a sum.

We then negate the right hand side, and now the problem is to minimise it.

Stripping away the multiplicative constant, we get

$$h_{ML} = \underset{h \in H}{\arg\min} \sum_{i=1}^{m} (d_i - h(x_i))^2$$

in other words, minimising the sum of differences squared is equivalent to finding the maximum-likelihood hypothesis.

Define the Minimum Description Length principle in Bayesian terms and outline the steps in its application, e.g., to decision tree learning

Consider

$$h_{MAP} \equiv \underset{h \in H}{\arg\max}\ P(h \mid D)$$

Take the base 2 logarithm of the right hand side and negate it, giving

$$h_{MAP} = \underset{h \in h}{\arg\max}\ [-\lg P(D \mid h) - \lg P(h)]$$

$\lg(P(x))$ is a statement about the number of bits in $P(x)$. We have the following statement from information theory:

> The optimal (shortest expected coding length) code for an event with probability p is −log2 p bits.

So if $L_C(x)$ is the description length of x under the optimal encoding C, then

$L_C(h) = -\lg P(h)$ is the length of h under the optimal code.

It seems that the MAP hypothesis h_{MAP} thus seeks to minimise the description length of the hypothesis and the dataset given the hypothesis. The minimum description length principle says that this value should be minimised.

If all examples are classified perfectly by h, the optimal encoding for $D \mid h$ is 0! However, as the hypothesis begins to fail on training examples, we must encode these as well. The MDL principle thus advocates **minimising the length of h and the length of misclassifications**. It might choose a shorter hypothesis even if training error increases. This tradeoff means MDL can be used to prevent overfitting!

MDL approaches to decision tree learning achieve similar performance to standard tree pruning methods.

Define the Bayes Optimal Classifier

We have asked the question *"what is the most likely hypothesis given the training data?"* when in fact the more appropriate question would be *"what should be the classification of this new instance given the training data?"* It turns out that merely applying the MAP hypothesis to our new instance isn't always the best strategy. For instance, if the hypothesis space contained three hypotheses with respective posterior probabilities $0.4, 0.3, 0.3$. On a new instance, the first hypothesis gives a positive classification and the second and third give negative classifications. The most likely classification here is $-$.

The Bayes optimal classifier considers every hypothesis in the hypothesis space when it makes its decisions. If potential classifications are $v_1 .. v_n$, the probability that the correct classification for the new instance is v_j is

$$P(v_j \mid D) = \sum_{h_i \in H} P(v_j \mid h_i) P(h_i \mid D)$$

The Bayes optimal classifier selects the v_j that maximises this expression.

Algorithms based on the Bayes optimal classifier have best average case predictive performance. Interestingly, because the formula above relies on the predictions of multiple hypotheses, the hypothesis corresponding to the predictions made by the Bayes optimal classifier does not have to be in H! Bayes optimal classification algorithm are generally very slow though.

Describe the Naive Bayes model and the role of Bayesian inference in it

Naive Bayes is a simple model for prediction. We have an instance, described by some tuple of attributes $\langle a_1, a_2, \ldots, a_n \rangle$, and we wish to find the most probable classification v_{MAP}.

$$v_{MAP} = \arg\max P(v_j \mid a_1, a_2, \ldots, a_n)$$

$$_{v_j \in V}$$

$$= \arg\max_{v_j \in V} P(a_1, a_2, \ldots, a_n \mid v_j)P(v_j)$$

Now, if x and y are two completely independent events, $P(x, y) = P(x)P(y)$. We will make the simplifying assumption that all the attribute values $a_1, \ldots, a_n$ are independent of each other. This is clearly incorrect, which is why the algorithm is termed 'naive'.

$$v_{NB} = \arg\max_{v_j \in V} P(v_j) \prod_{i=1}^{n} P(a_i \mid v_i)$$

We simply return this value of v_{NB}!

Reproduce the Naive Bayes classifier, e.g. for text classification

Naive Bayes example

Consider the training data:

Weather	Time of day	Day of week	Soccer?
Sunny	Day	Monday	Yes
Sunny	Day	Tuesday	Yes
Windy	Night	Monday	No
Windy	Day	Friday	No
Sunny	Day	Wednesday	Yes
Sunny	Night	Saturday	No

And the instance

$$x = \langle \text{Sunny}, \text{Day}, \text{Friday} \rangle$$

The set of possible target values $V = \{Yes, No\}$. 50% of the training examples are classified as Yes and 50% are classified as No, so $P(Yes) = P(No) = 0.5$.

Note that $P(\text{Friday} \mid Yes) = 0$; this would stuff up our algorithm, so we add 1 to all counts (a version of the **Laplace estimator**) to correct such errors.

$$\begin{aligned} P(Yes \mid \text{Sunny}, \text{Day}, \text{Friday}) &= P(Yes)P(\text{Sunny} \mid Yes)P(\text{Day} \mid Yes)P(\text{Friday} \mid Yes) \\ &= \frac{1}{2} \times \frac{4}{4} \times \frac{4}{4} \times \frac{1}{4} \\ &= \frac{1}{8} \end{aligned}$$

$$\begin{aligned} P(No \mid Sunny, Day, Friday) &= P(No)P(Sunny \mid No)P(Day \mid No)P(Friday \mid No) \\ &= \frac{1}{2} \times \frac{2}{4} \times \frac{2}{4} \times \frac{2}{4} \\ &= \frac{1}{16} \end{aligned}$$

Normalising these values, we get $P(Yes) = \frac{1/8}{1/8+1/16} = \frac{2}{3}$, $P(No) = \frac{1}{3}$ and thus our output is Yes.

Zero frequency problem

As above, probabilities of 0 can break the Naive Bayes estimator.

A simple version of the **Laplace estimator** adds 1 to every count.

More complex versions such as the **m-estimate** exist.

Outline the key elements of the Bayes Net formalism

The NBE assumes that all attributes are conditionally independent of each other. This is clearly problematic -- but at the same time, it is computationally impossible to assume that they are all conditionally dependent on each other.

A Bayesian belief network assumes that certain subsets of variables are conditionally dependent on each other, a tradeoff between NBE and the infeasible approach.

Each node in the Bayes net represents a variable. A variable is conditionally independent of all non-descendants, given its immediate parents.

$$P(y_1, y_2, \ldots, y_n) = \prod_{i=1}^{n} P(y_i \mid Parents(Y_i))$$

The process of learning Bayes nets is similar to the process of learning the weights of a neural net, in the sense that we can only observe inputs and outputs -- not the links between nodes.

Outline issues in learning Bayes Nets

Inference in Bayes nets is NP-hard. If we know the values of some nodes and we want to determine the probabilities of the values of another, we cannot do this in polynomial time.

Gradient descent can be used to learn Bayes nets.

The expectation-maximisation (EM) algorithm can also be used.

Describe the EM algorithm for learning with missing data

Expectation-maximisation is an algorithm for unsupervised learning. It's similar to k-means clustering and can be used for clustering.

EM is used in the following scenario: you have a bunch of sources of data called **generating distributions**, located in particular spots. Each source spits out datapoints around it, in a manner that follows a Gaussian distribution. Given the datapoints, the task is to identify the location of the sources. The set of all datapoints is called a **finite mixture**, because there are a finite number of generating distributions.

Formally:

- Instances in X generated by a mixture of k Gaussian distributions with unknown means $\langle \mu_1, \ldots, \mu_k \rangle$.
- Don't know which Gaussian generated which instance.
- Determine maximum likelihood estimates for $\langle \mu_1, \ldots, \mu_k \rangle$ i.e. find means that maximise the probability that the instances were generated by those means.

The algorithm works by repeatedly alternating between expectation and maximisation. First, a bunch of means are randomly generated.

1. **Expectation** Each point is associated to each mean with some strength.
2. **Maximisation** Each mean is moved to make it more likely that it produced the points to which it is more strongly associated with.

We terminate when the probability that the points were generated by the means converges.

EM is pretty much the same as k-means, but instead of each instance being associated to exactly one cluster, every instance is associated to every Gaussian with different degrees of strength.

A mostly-understandable explanation of EM can be found here.

Classification Rule Learning

Classification rules are a simple and intuitive model for classification.

Syllabus Dotpoints

Define a representation for rules

Classification rules are one of the simplest models for machine learning. A classification rule is of the form

$$\text{antecedent} \rightarrow \text{consequent}$$

The antecedent is also called a *pre-condition*, and the consequent is also called a *post-condition*.

The antecedent must be expressed as a logical conjunction of constraints: that is, it can only be of the form *if*

$$constraint1$$

AND

$$constraint2$$

AND

$$constraint3$$

AND .... Any condition requiring disjunction (ORs) can be expanded out into multiple rules with the same consequent.

They can be interpreted intuitively as *if-then-else* statements.

Many rules may 'fire' for the same instance. In this case, we can either give no conclusion, or return the class that occurs more frequently in the training set (this is based on the heuristic that the class with higher frequency in the training set has a higher prior probability and is thus more likely to be the correct classification).

Describe the decision table and 1R approaches

Decision tables

Decision tables are essentially compressed versions of the training data. The compression is achieved by only including a subset of the attributes/features. This compression allows for a simple form of generalisation.

A decision table is uniquely specified by a **schema** (set of attributes) and a **body** (multiset of instances, where each instance has a value for each attribute in the schema).

For example, consider the following training data:

Weather	Time of day	Day of week	Soccer?
Sunny	Day	Monday	Yes
Sunny	Day	Tuesday	Yes
Windy	Night	Monday	No
Windy	Day	Friday	No
Sunny	Day	Wednesday	Yes
Sunny	Night	Saturday	No

A rule with 100% accuracy on this data would be *if weather=sunny and time of day=day, then* +. A decision table representation of this rule is as follows:

Weather	Time of day	Soccer?

Sunny	Day	Yes
Sunny	Night	No
Windy	Day	No
Windy	Night	No

This table generalises further than our input data, because it will classify the following instance (which was not present in the training data) correctly:

Weather	Time of day	Day of week	Soccer?
Sunny	Night	Sunday	No

The following algorithm learns decision tables, by greedily picking the attribute that minimises cross-validation error each time:

```
remaining attributes = all attributes
schema = {}

DO
    a = find the best attribute in remaining attributes to add to the schema,
        by minimising CV error
    add a to the schema
WHILE CV error is decreasing
```

Note that we must use the cross-validation error, as the training set error will *always* decrease as we keep adding attributes.

1R

1R is a simple decision rule that stands for *1 Rule*. This is a misnomer, as 1R actually learns **multiple rules for one attribute**. It can be thought of as a one-level decision tree.

The algorithm to learn the rule is intuitive: we simply iterate through each attribute and calculate the error we get from making a rule based on this attribute. We select the attribute that minimises this error.

We can treat **missing values** as their own separate value.

Continuous values are discretised by choosing thresholds as usual. Beware not to choose too many thresholds or you'll over-fit!

Zero-R

Simply return the most frequent class that appears in the training set.

This provides a 'baseline' level to compare other learners against.

Outline overfitting avoidance in rule learning using pruning

Overfitting is avoided by using **cross-validation error** instead of training set error. CV error more closely approximates the **true error** (off-training-set error) of the learner).

Reproduce the basic sequential rule covering algorithm

Sequential covering is a simple algorithm for learning rules that does not involve decision trees.

It is a greedy algorithm that repeatedly learns one rule with 100% accuracy, then removes from the training set any instances covered by that rule. The rules are then outputted in order of their performance.

The complexity of sequential covering comes from the choice of algorithm for `learn_one_rule`. A simple method is to start off with the most general rule antecedent possible, then keep specialising it with a 'good' minimal specialisation (for some definition of 'good') while it covers more than one class. Because this search is highly greedy and relies heavily on a good heuristic for its minimal specialisation, there's no guarantee of it producing a good rule; a **beam search** is thus often employed, which is essentially a parallel search that keeps track of multiple antecedents at a time. To choose which is the best specialisation to a rule at any particular stage, we can use information gain, much like *ID3*. The *CN2* program uses a beam search with information gain.

Ensemble Learning

Ensemble learning is the notion of combining multiple learners into one improved 'meta-learner'.

Syllabus Dotpoints

Describe the framework of ensemble learning

Ensemble learning produces multiple different learners / 'experts' and allows them to "vote" on the classification.

It has the advantage of often producing better classifications than any of its constituent learners.

It has the disadvantage of being a very opaque model.

There are three schemes for ensemble learning: bagging, boosting and stacking.

Define the method of bagging

Bagging stands for **bootstrap aggregation**, and is the idea of training multiple learners then treating each learner's vote equally.

Each learner is trained on their own dataset. We generate these datasets by sampling n items **with replacement** from the set of available training data with n instances.

1. Create each new training set by sampling with replacement.
2. Train each model with its own dataset.
3. When classifying, give each of the models the same weight -- so essentially predict the average of the classifications.

Bagging reduces variance, thus reducing the total expected error.

Bagging is especially useful when the model are **unstable**, for example decision trees. However, once you bag, you lose the main benefit of trees: interpretability.

A model is **unstable** if, when trained on different datasets sampled from the same distribution, it is likely to produce a (substantially) different model, and **stable** otherwise.

Define the method of boosting

Boosting is similar to bagging, but weights models differently. Specifically, each model is weighted based on how well it performs. Models that perform badly are given low weights. New models are also encouraged to become experts in instances which old models performed badly on, thus producing a well-rounded combined learner. Boosting is particularly effective in boosting several weak learners into one strong learner.

The algorithm is straightforward and follows from its description:

1. Weight all instances equally.
2. While we want more models:
    1. Train a new model on the instances, with their corresponding weights.
    2. $e \leftarrow$ the error of our model.
    3. Weight this model by $1 - \frac{e}{1-e}$ (gives high weights to models with low e).
    4. For each instance:
        1. If the model got it right:
            1. Multiply its weight by $\frac{e}{1-e}$.

Boosting enlarges the model class. For example, if you boost a bunch of linear models, the final classifier isn't necessarily linear! The decision boundary will probably be a set of line segments.

Define the method of stacking

Stacking plugs in the predictions from a set of learners into a 'meta-learner'.

What we would normally do in cross validation is to train a bunch of models on a hold-in set, then compare their performances on a hold-out set and pick the best. What stacking does is, instead of picking the best, it combines all the models.

The 'low level' models are trained on data from a hold-in set. Once done, the 'high level' model is trained using the outputs of the low level models on the hold-out set.

Evaluating Hypotheses

Syllabus Dotpoints

Define sample error and true error

Sample error

> The sample error error_S of a model on a sample. For a classifier, this may be simply the fraction of instances in the sample that the model classifies incorrectly.
>
> $$\mathrm{error}_S \equiv \frac{1}{n}\sum_{x\in S}[f(x) \neq h(x)]$$

True error

> The true error error is the probability that L will misclassify a random instance picked from X according to .
>
> $$\mathrm{error} \quad \equiv \Pr_{x\in} \; [c(x) \neq h(x)]$$

Describe the problem of estimating hypothesis accuracy or error

If you choose the sample by randomly sampling from , $\mathrm{error}_S(h)$ is a random variable that is an **unbiased estimator** for $\mathrm{error} \quad (h)$.

What we wish to do is identify $\mathrm{error} \quad (h)$ (or at least, identify properties of it) given error_S (h).

Understand confidence intervals for observed hypothesis error

Confidence intervals allow determining a bound on the true error, with some confidence, given the sample error.

In particular, if S contains $n \geq 30$ instances, we can state that with $N\%$ probability, $\mathrm{error} \quad (h)$ lies in the interval

$$\mathrm{error}_S \pm z_N\sqrt{\frac{\mathrm{error}_S(h)(1-\mathrm{error}_S(h))}{n}}$$

where z_N is a value read from a table.

Understand learning algorithm comparisons using paired t-tests

A t-test is a measure of how likely it is that two samples were generated from the same distribution.

Practically, we use it to measure how likely it is that two sets of data are meaningfully different (their differences are *statistically significant*). If they are not statistically significant, they could have been generated from the same distribution, and their differences are just based on chance. If they are statistically different, it is highly unlikely that their differences are based on chance, and they were probably generated from different distributions.

Define and calculate typically-used evaluation measures, such as accuracy, precision, recall, etc.

Accuracy

The proportion of instances classified correctly by the learner, as a fraction of the total number of instances. The most obvious measure of how good a learner is.

Precision

The proportion of true positives as a fraction of all classified positives.

Think of this as a measure of *how precise*, or sharp, the returned values are. A 'blunt' learner would return false instances, and stuff up. A precise learner would only return positives, even if it misses out on some true negatives.

Recall / sensitivity

The proportion of true positives as a fraction of all positive instances.

Think of this as *how good the learner's recollection is* -- his memory. All we care about is whether it recalled all the positive instances, even if it gave us some negatives as well.

Specificity

The proportion of true negatives as a fraction of all negative instances. The negative version of recall.

Think of this as *how good the learner is at avoiding false alarms*. If it returns negative and has high specificity, you can be pretty sure it's negative.

F-measure

A combination of precision and recall. Both are weighted equally.

$$F = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

Define and calculate typically-used compound measures, such as lift charts, ROC curves, F-measure, AUC, etc.

Lift charts

Imagine that we want to send advertising fliers out to people. Each person has a susceptibility to advertising. We could send out 100 fliers to the top 100 most susceptible person, and 80 of them would respond; or we could send out 1000 fliers to the top 1000 most susceptible, and get 400 responses. 1000 fliers is obviously more expensive.

In this scenario, there's no 'best answer'; it depends on how much we're willing to spend. A **lift chart** plots these different solutions agains each other and allows for a visual comparison. The x-axis is the number of fliers we send (or equivalently, the size of the sample classified as positive) and on the y-axis is the number of responses (or equivalently, the proportion of positive instances classified as such as a fraction of all positive instances -- the recall of the learner).

A perfect learner would look like a steep vertical line (as the sample size increases to accomodate all positive samples), then a straight horizontal line.

ROC curves

The same as a lift chart, except the x-axis shows the *number of false positives* instead of the total sample size / number of true positives. The y-axis still shows the number of true positives.

A perfect learner would be characterised by the line $y = 100\%$, since they can get all true positives with no false positives.

Stands for **receiver operating characteristics**, a term from signal detection.

AUC

A real number that stands for **area under the ROC curve**. The AUC can be interpreted as the *probability that the classifier will rank a randomly drawn positive example higher than a randomly drawn negative example*.

We obtain the AUC as follows:

1. Sort all classified instances by their strength (probability of being in the positive class as decided by the classifier), keeping track of their actual classes.
2. For each positive instance, count the number of negative instances ranked strictly below it. If a negative instance has the same strength, count it as $\frac{1}{2}$.
3. Sum all these counts into a single value C and return $\frac{C}{P \times N}$, where P and N are the number of positive and negative instances, respectively.

Demonstration

Strength	Class	# -ve below
0.93	+	6
0.86	-	
0.82	+	5
0.64	+	5
0.55	-	
0.41	+	4
0.40	+	4
0.37	-	

0.25	-	
0.20	+	3
0.12	-	
0.03	-	
TOTAL		**27**

$\mathrm{P} = 6$ and $\mathrm{N} = 6$, so $\mathrm{AUC} = \frac{27}{36} = 0.75$.

Kernel Methods

Kernel methods encompass techniques whereby an attribute vector is mapped into higher dimensions.

Syllabus Dotpoints

Outline the key ideas in kernel methods

Consider the problem of separating instances with a linear decision boundary. In many cases, a linear decision boundary does not exist. One option would be to use a more complex decision boundary. Another option is to map the instances into a higher dimensional space in which a linear decision boundary is more successful. A simple way of doing this is to simply generate more features for each instance, where each new feature is a combination of existing features.

This often works. However, adding too many new features slows the algorithm down a lot, since many new weights must be learnt. Also, the curse of dimensionality applies and overfitting becomes more likely.

The 'kernel trick' in SVMs is an observation that allows you to map into this higher dimension implicitly. This means you don't actually need to generate new features -- the learner deals with the instances as if these new features were generated, without them actually existing. This means that the algorithm isn't slowed down at all.

The **kernel function** is the function that performs this mapping. Common kernel functions are the **polynomial kernel**, the **sigmoid kernel**, and the **radial basis function kernel**. Kernel functions are modular and can easily be swapped out and replaced with others.

Describe the method of support vector machines (SVMs)

Support vector machines use a hyperplane to separate instances. The hyperplane learnt is the **maximum margin hyperplane** -- the hyperplane that maximises the distance to its closest instance. Mathematically, this involves solving a **quadratic optimisation** problem.

The maximum margin hyperplane can be found easily as follows: take the convex hull of all the instances belonging to each class, the draw the line between the two closest points on each of the hulls, the take the hyperplane that passes through the middle of this line and is perpendicular to the line (the higher dimensional 'perpendicular bisector').

Although SVMs employ a linear model, kernel methods as described above allow SVMs to effectively operate in much higher dimensions, with little slowdown. They are highly effective in practice.

Learning and Logic

Terminology

- **Expressions** are composed of **constants** (capitalised, e.g. Bob), **variables** (lowercased, e.g. x) that take on the value of expressions, **predicates** (capitalised, e.g. $Greater$, Is_Green that return booleans, and **functions** (lowercased, e.g. age , $height$) that return arbitrary values.

- A **term** is any constant, any variable or any function applied to a term.

- A **literal** is a predicate or the negation of a predicate applied to a term. Literals are the only way to get a boolean value in an expression. Literals are **positive** or **negative** (the latter iff they are negated).

- A **clause** is a disjunction of literals.

- A **Horn clause** is a disjunction of literals with at most one positive literal.

- A **substitution** is a function that replaces variables with terms. For example, the substitution $\{x/Bob, y/Mary\}$ replaces x with Bob and y with $Mary$. Given a substitution θ and a literal L we write $L\theta$ to indicate the result of the substitution. e.g. $(Daughter(y, x))\theta = (Daughter(Mary, Bob))$.

- A **unifying substitution** for two literals is a substitution L that makes the two literals look the same. e.g. $\theta = \{x/Bob\}$ is a unifying substitution for $Daughter(Mary, x)$ and $Daughter(Mary, Bob)$.

Syllabus Dotpoints

Outline the key differences between propositional and first-order learning

Propositional learning allows learning under the framework of **propositional (variable-free) logic**. Propositional logic is essentially boolean logic: we only have the basic logical operators at our disposal. This allows us to describe (and hence, to learn) basic things, e.g. $is\_free\_time = weekend$. Note that here, $weekend$ is not a

$\vee\ (\mathrm{weekday} \wedge \neg \mathrm{during\_work\_hours})$

variable; rather, it is simply the name of an entry.

Propositional logic cannot describe more complex concepts, such as the notion of a *grandchild*. Even given an infinite amount of training data in the form x is a parent of y, a propositional learner will never learn the concept $\mathrm{Grandchild}(x, y) = \mathrm{Parent}(z, x) \wedge \mathrm{Parent}(y, z)$.

First-order learning rectifies this by learning first-order logic rules. The above rule is a valid rule in first-order logic.

Describe the problem of learning relations and some applications

See the above example about learning the relation of $\mathrm{Grandchild}$. Another example is learning about connectivity concepts in graphs, e.g. the concept $\mathrm{connected\_to}$.

Outline the problem of induction in terms of inverse deduction

Describe inverse resolution in propositional and first-order logic

The problem of machine learning in general can be described as follows:

> **Find a hypothesis h so that with background knowledge B and any instance x, we can precisely deduce the value of the target function applied to x.**

Formally,

$$(\forall \langle x, f(x) \rangle \in X)\ (B \wedge h \wedge x) \vdash f(x)$$

This operation of deduction can be inverted into **induction**: given training data and background knowledge, can we work out a hypothesis?

Note that unlike deduction, this is a much harder operation, and a non-deterministic one; there may be multiple hypotheses that all fit the training data perfectly.

Propositional resolution

Resolution operation

$$(P \vee L) \wedge (R \vee \neg L) \implies P \vee R$$

Intuitively, either L or $\neg L$ must be false; therefore either P or R must be true.

Inverse resolution in propositional logic is an operation that takes the resolvent and one initial clause, and produces the other initial clause.

Let C_1 be the initial clause given, R be the resolvent and C_2 being the clause to derive. C_2 is not unique! And thus inverse resolution is not deterministic. A common heuristic is to choose the shortest of the possible C_2's We derive by following these two rules.

- Any literals appearing in R and not appearing in C_1 must have appeared in C_2.
- Any literals appearing in C_1 and not in R must have been negated in C_2.

First order resolution

Resolution extends to first order logic. Again, we take two clauses and produce an output clause.

However, here, we rely on unifying substitutions. This is because our task is to identify a literal L_1 from C_1 and L_2 from C_2 so that there exists a substitution $L_1\theta = \neg L_2\theta$. First order logic is more complex than propositional logic and things can be hidden behind variables, so substitutions uncover these hidden things.

For instance,

$$C_1 = \text{White}(x) \vee \neg\text{Swan}(x)$$

$$C_2 = \text{Swan}(\text{Fred})$$

Choosing $\theta = \{x/\text{Fred}\}$, we see that $C_1\theta = \text{White}(\text{Fred}) \vee \neg\text{Swan}(\text{Fred})$ and $C_2\theta = \text{Swan}(\text{Fred})$, and hence we can conclude $\text{White}(\text{Fred})$.

In general,

1. Find a literal L_1 from C_1 and L_2 from C_2 so that there exists a substitution $L_1\theta = \neg L_2\theta$.

2. Form the resolvent by including all the literals from $C_1\theta$ and $C_2\theta$ except for $L_1\theta$ and $L_2\theta$.

This is reversed in exactly the same way as in propositional logic to give inverted resolution. Often, we require multiple steps.

Example

Consider the resolvent $R(B, x) \vee P(x, A)$ and the clause $C_1 = S(B, y) \vee R(z, x)$.

The term $R(B, x)$ appears in C_1 as $R(z, x)$ so we must apply the inverse substitution $\{z/B\}$. The term $S(B, y)$ appears in C_1 but not in the resolvent, so it must have been negated in C_2. The term $P(x, A)$ is also unique to the resolvent, so it must have come from C_2. We hence find $C_2 = \neg S(B, y) \vee P(x, A)$. Note that $C_2 = \neg S(B, y) \vee P(x, A) \vee R(B, x)$ is also a valid C_2, as is $C_2 = \neg S(q, y) \vee P(m, A) \vee R(B, j)$ and various others, with suitable substitutions. The minimal description length principle suggests that we take the shortest possible C_2.

Describe least general generalisation and θ-subsumption

θ-subsumption

A clause A **θ-subsumes** a clause B if $\exists A : A\theta \subseteq B\theta$. In english, θ-subsumption is essentially a statement that A is more general than B. Indeed, $\mathrm{more\_general\_than}$ is a special case about θ-subsumption. Also, if A θ-subsumes B, then since A and B are clauses, $A \vdash B$, indeed, $A \implies B$: A entails B and implies B. θ-subsumption is a special case of logical entailment.

What this tells us is that guiding a search by specialisation and generalisation (like *Find-S*) is more limited than guiding a search by inverse entailment operators. Unfortunately, purely relying on inverse entailment is intractable. *CIGOL* is a more-or-less successful implementation of a learner that uses inverse entailment.

LGG

Since θ-subsumption allows us to partially order clauses, we can talk about clauses in a *lattice* of generality. In this lattice, we can define the **least general generalisation** of two clauses to be the "lowest common ancestor" of the two.

Because we're in first-order logic land, we have to also cope with more complex terms. We have a few intuitive rules for doing this:

- The LGG of two different variables is the variable X.
- The LGG of two different constants, or two different functions, is the variable X.
- $\mathrm{LGG}(f(a_1, \ldots, a_n), f(b_1, \ldots, b_n)) = f(\mathrm{LGG}(a_1, b_1), \ldots, \mathrm{LGG}(a_n, b_n))$.

Because we're talking about generality, $\mathrm{LGG}(A, B) \vdash A \wedge B$. We can also talk about the ***relative* LGG** as being the clause such that, with background knowledge B in hand, we can deduce $A \wedge B$; formally, $B \wedge \mathrm{RLGG}(A, B) \vdash A \wedge B$.

Reproduce the basic FOIL algorithm and its use of information gain

FOIL is a rule learning algorithm that is a natural extension to sequential covering. However, instead of trying to learn propositional rules, it tries to learn first-order rules.

Like sequential covering, it is a **search for a set of disjunctions that altogether fully determine whether an instance is in the positive class**.

The basic FOIL algorithm is identical to sequential covering:

```
pos = all instances of positive class
neg = all instances of negative class

learned_rules = {}

while pos:
    new_rule = the most general possible positive rule i.e. '<> -> positive'

    while there are negative instances covered by this rule:
        generate candidate literals for the rule
```

```
          best_literal = the best candidate literal: the one that maximises Foil
_Gain(l, new_rule) forall l in literals

          add best_literal to the antecedent of new_rule
     add new_rule to learned_rules
```

The procedure to generate candidate specialisations spits out a bunch of literals. In particular, the following:

- For all predicates available, for all variables available, that predicate applied to that variable: $P_i(v_j)$.
- For all pairs of variables x_i, x_j already present in the rule, the relation $\mathrm{Equal}(x_i, x_j)$.
- All negations of the two above.

The `Foil_Gain` function is similar to information gain, and estimates the utility of adding in a literal based on how many positive and negative instances it covers.

One of FOIL's biggest failings is that is does not use function symbols at all. However, FOIL rules are more expressive than Horn clauses because they are able to express any clause.

Recommender Systems

Syllabus Dotpoints

Define the problem of recommender systems

We have a set of users S and items C, and wish to maximise the utility function u : S \times C \to \mathbb{R} that tells us the 'goodness' of an item to a user. We wish to predict the utility function, to allow us to choose items that maximise the utility for a user.

This is learning in the sense that it requires extrapolation.

Describe content-based, collaborative and hybrid recommender systems

Content-based

Content-based systems recommend based on *users' past choices*. They attempt to match content to users based on the content itself, and the user's tastes. Content often has to be tagged with meta-information.

For instance, a user might like Harry Potter, Alice in Wonderland and Lord of the Rings. All these films might be tagged with 'fantasy'. Based on this, the system might decide that the user likes fantasy films and recommend The Chronicles of Narnia.

Collaborative

Collaborative systems cluster users based on similar interests, and recommend content enjoyed by other users with similar tastes. *"Users who bought this item also bought..."*

For instance, if Mary likes Harry Potter, Alice in Wonderland and Lord of the Rings, and Bob likes Harry Potter and Alice in Wonderland, the system might decide that Mary and Bob have similar tastes, and recommend Lord of the Rings to Bob.

This approach has the **cold start** problem that new users cannot be recommended items. Hybrid approaches can solve this problem.

A **grey sheep** is a user who is too 'normal', and is not sufficiently individual to produce meaningful recommendations.

A **black sheep** is a user who is too individual, and not enough similar users can be found to produce meaningful recommendations.

Collaborative filtering

Collaborative filtering is a formalised collaborative recommender system. It predicts the unknown rating $r_{s,c}$ for user s and item c by looking at the k most similar users to s that bought c and averaging their ratings for it. This is similar in concept to k-NN.

The averaging can be weighted by the similar users' similarity, instead of taking the k closest:

$$r_{s, c} = \alpha \sum_{s' \in S} \text{sim}(s, s') \cdot r_{s', c}$$

Where α is a normalising factor.

Hybrid

Combine the two systems above.

This can solve the cold start problem.

Reproduce key similarity-based approaches to recommender systems

We have the training data:

	Casino Royale	Terminator	Mean Girls	Twilight
Jack	5	3.5	3	1
Tim	4	4	2	2.5
Mary	2	1.5	4	3

Sue	3	1	3	5

And test instances

	Casino Royale	Terminator	Mean Girls	Twilight
John	4.5	?	2	2
Jenny	1	3	5	?

Collaborative filtering example

Let's define a simple distance measure between two users

$$dist(u, w) = \sum_{m \in M} \left\vert r_{u,m} - r_{w,m}\right\vert$$

Where $r_{x,m}$ is user x's rating of movie m. We will only consider movies for which we have rating data for *both* u and w.

Reinforcement Learning

Reinforcement learning is a type of learning where feedback is given dynamically in response to actions chosen by the learner.

I couldn't find the syllabus points for this unit, so I cover the textbook content.

Terminology

- S is the set of all possible **states** the learner can inhabit, and A is the set of all possible **actions** the learner can take in those states.
- The aim is to learn a **policy** $\pi : S \to A$ that gives the learner an action to take based on the current state.
- The **cumulative value** $V^\pi(s_t)$ is the value of following the policy pi from a given starting point s_t. r_i is the reward given by $\pi(s_i)$, Delayed rewards are worth less than immediate rewards, so we have

$$V^\pi(s_t) = \sum_{i=0}^\infty \gamma^i r_{t+i}$$

- The **optimal policy** π^* is the policy that maximises cumulative value $V^\pi(s)$ for all states s: $\pi^* \equiv \underset{\pi}{\arg\max} \; V^\pi(s), \; (\forall s)$

We refer to $V^{\pi^*}(s)$, the value of following the optimal policy from a state s, simply as $V^*(s)$.

- An **absorbing state** is a state from which no actions can be taken. Goal states are often absorbing states.

Syllabus Dotpoints

Q learning

Q learning is a strategy for learning π^*.

The Q function

The Q function $Q(s, a)$ is the function that maximises discounted cumulative reward, starting from s and immediately applying a. Note that by the definition of π^*, $\pi^*(s)$ is the function that maximises $Q(s, a)$ across all actions a.

$$Q(s, a) = \underset{a}{\max} \pi^*(s, a)$$

The algorithm

Q learning is similar to dynamic programming, in that the optimal choice for each state is determined based on optimal choices in subsequent states. Future decisions also influence the current state, but their influence is exponentially decreasing over time. The algorithm itself is simple:

1. Initialise the table of $\hat Q(s, a)$ values to 0
2. Observe the state s
3. Do forever:
    1. Select an action a and execute it
    2. Receive immediate reward r
    3. Observe the new state s'
    4. Update the entry for $\hat Q(s, a)$ as follows: $$\hat Q(s, a) \leftarrow r + \gamma \underset{a'}{\max} \hat Q(s', a')$$
    5. $s \leftarrow s'$

In other words, the 'goodness' for action a in state s is the immediate reward that this action provides, plus a reduced version of the maximum possible 'goodness' that we can achieve from the new state.

If the system is a deterministic Markov decision process, it can be proved that $\hat Q$ will converge towards the true function of Q.

Exploration strategies

To ensure a balance between exploration and exploitation, Q learning can use a probabilistic strategy to choose the next state to apply. One method to assign such probabilities is to give higher probabilities to actions that provide higher reward:

$$P(a_i \mid s) = \frac{k^{\hat Q(s, a_i)}}{\sum_j k^{\hat Q(s, a_j)}}$$

The value of the constant k can be adjusted. Higher values assign more importance to actions with high $\hat Q$ values i.e. exploitation.

The value of k can be varied over time to control the degree of exploration vs. exploitation.

Non-deterministic Q-learning

In the non-deterministic case (both of state transitions and of rewards), an equivalent Q function exists:

$$Q(s, a) = E[r] + \gamma \sum_{s'} P(s' \mid s, a) \underset{a'}{\max} Q(s', a')$$

There is also a slightly different learning function $\hat Q$ for the non-deterministic case.

Exploration vs. exploitation

Exploration vs. exploitation refers to a common tradeoff in reinforcement learning. The learner may choose to *exploit* actions which are currently known to provide high reward, at the cost of potentially discovering actions that provide higher reward; however, the learner may also choose to *explore* new actions that could provide higher reward, at the cost of high rewards provided by currently known actions.

Unsupervised Learning

Unsupervised learning is learning where no feedback is given and 'answers' are withheld.

Syllabus Dotpoints

Describe the role of the EM algorithm in k-means

EM is effectively a probabilistic version of k-means. Instead of each instance being associated to the closest cluster in each iteration, each instance is associated to every cluster with a different degree of strength, corresponding to the likelihood that it was generated by that cluster. During the *maximisation* stage, each cluster is moved to the position that maximises the likelihood that it generated its instances, which is equivalent to the k-means step of moving clusters to the mean of its associated instances.

Describe conceptual clustering

Conceptual clustering clusters instances based on how well they fit into defined concepts. For example, an instance with the features \left<is_black, flies, is_small\right> fits better into the concept bird than \left<is_green, flies, is_large\right>. Each cluster is called a **category**.

COBWEB

COBWEB is a completely probabilistic conceptual clustering algorithm. This means that each concept holds only the probabilities that a value is set; for instance, bird might have a 95% chance of the attribute flies, a 40% chance of %is_black% and an 80% chance of is_small. These are called the *conditional likelihoods* P(x \mid bird). Probabilities are useful for concept classification since in real life scenarios, there is an exception to almost every rule.

COBWEB builds a taxonomy of concepts, and begins with a single concept. Each time it is presented with a new instance, it chooses the best of the following:

1. Create a new concept for this instance.
2. Add this instance to an existing concept.
3. Merge two existing concepts and add this instance to the result.
4. Split an existing concept and add this instance to the result.

It chooses between these by choosing the option that maximises the **category utility**. Category utility attempts to maximise the probability that two objects in the same category have values in common, and two objects in different categories have differing values.

COBWEB is a hill-climbing algorithm which strictly speaking does not backtrack; however, since the operations can reverse each other, this allows the algorithm to 'revert' bad decisions.

For more on concept learning and COBWEB, see here.

Extra Resources

Sample Paper

Sample paper can be found here.

Q1

A)

1. At least 90\%. 90%
2. At least 90\%. 90%
3. Exactly 90\%. 90%
4. At least 40\%. 40%
5. At least 40\%. 40%

40%

B)

For each itemset of size k, for each possible item, generate an itemset of size k+1 by adding this item to the set. If this itemset is infrequent, ignore it and proceed. By the downward closure lemma, this will derive all itemsets of size k+1.

C)

\theta_1 = \{y/A\}

\theta_2 = \{\}

C_2 = P(A, x) \lor \lnot T(A)

D)

\theta_1 = \{w/y\}

\theta_2 = \{\}

C_2^{(1)} = R(z) \lor Q(A, x, B)

C_2^{(2)} = R(z) \lor Q(A, x, B) \lor S(y, B)

Q2

A)

Each condition is a node. Each boolean is a leaf/classification.

- if X=0 then
  - if Y=0 then
    - false
  - else
    - if Z = 0 then
      - false
    - else
      - true

- else
    - if $Y = 0$ then
        - if $Z = 0$ then
            - false
        - else
            - true
    - else
        - if $Z = 1$ then
            - false
        - else
            - true

B)

if $X=1 \land Y=1 \land Z=1$ then false else if $X=1 \land Y=1$ then true else if $X=1 \land Z=1$ then true else if $Y=1 \land Z=1$ then true else false

C)

Label the instances #1..#8 for convenience.

Instance left out	2-NN	Classification	Score
#1	#2, #3	false	1
#2	#3, #5	false	1
#3	#2, #5	false	1
#4	#6, #7	true	1
#5	#2, #3	false	1
#6	#4, #7	true	1
#7	#4, #6	true	1
#8	#4, #6	true	0
ACCURACY			**0.875**

LOOCV error: 0.125.

D)

The classification rules model, at it is concise, complete, 100% accurate and interpretable.

Q3

A)

Initially, $G = \{[1,5]\}, S = \{[]\}$.

1. 1 vote positive, 1 vote negative. Verdict positive. **Mistake made.**

   $G = \{[2,5]\}, S = \{[]\}$.

2. 1 vote positive, 1 vote negative. Verdict positive.

   $G = \{[2,5]\}, S = \{[2, 2], [2, 3], [2, 4], [2, 5]\}$.

3. 4 votes positive, 1 vote negative. Verdict positive.

   $G = \{[2,5]\}, S = \{[2, 3], [2, 4], [2, 5]\}$.

4. 3 votes positive, 1 vote negative. Verdict positive.

   $G = \{[2,5]\}, S = \{[2, 4], [2, 5]\}$.

5. 2 votes positive, 1 vote negative. Verdict positive. **Mistake made.**

   $G = \{[2,5]\}, S = \{[2, 5]\}$.

B)

The halving algorithm makes at most $\left\lfloor\log_2 \left\vert{H}\right\vert\right\rfloor$ mistakes. The hypothesis space contains $5+4+3+2+1+1$ hypotheses (for each startpoint $1 \leq a \leq 5$, for each endpoint $a \leq b \leq 5$, as well as the empty interval), and so we conclude that the worst-case mistake bound is $\left\lfloor\log_2 16\right\rfloor = 4$.

Q4

A)

1. Construct a complete graph over all instances.
2. While there are less than k connected components, where k is the desired number of clusters:
   1. Remove the longest edge.
3. Each component is now a cluster.

B)

1. Randomly position each cluster centre.
2. Initialise the edges of the bipartite graph so that each instance has one edge to its closest cluster.

3. While the graph continues to change:
    1. Move the centre of each cluster to the centroid of the instances to which it is connected.
    2. Remove all edges of the graph and create an edge from each instance to its closest cluster.

C)

Category utility attempts to maximise the probability that items in the same category have similar attribute values, and items in different categories have different attribute values. From the perspective of Bayesian learning, it tries to maximise the conditional probability that attribute a_i has value v_j given a category C_k, across all attributes, values and categories.

Q5

A)

An arbitrarily large set of instances can be shattered by H.

Since each binary string of length m can be encoded as a walk down a binary tree of height m, taking the left branch for every 0 digit and the right branch for every 1 digit, a decision tree of height m where all the nodes tests at level k test attribute k is capable of specifying the precise class for any binary string.

Hence, all 2^{2^m} dichotomies of 2^m binary strings of length m are expressible by H.

B)

The size of the learner's hypothesis space is $3^8=6561$.

Set $\delta=0.05, \epsilon=0.1$. The formula for the number of examples m is

$$m \geq \frac{1}{0.1}\left(\ln 6561 + \ln \frac{1}{0.05} \right)$$

C)

(d), and possibly also (c) and (e)?.

Q6

A)

Bias decreases.

Variance increases.

Predictive accuracy on training data increases.

Predictive accuracy on test data increases to a point (while bias is decreasing), then decreases as variance begins to dominate.

B)

No. Decision tree learning can have very high variance, because the choice of the root node at the start is very much dependent on the training set and greatly influences the performance of the tree. Decision tree learning can have very low bias, and algorithm like ID3 can run until they have no error on the training set, but pruning is usually employed to increase the bias, which decreases the probability of overfitting.

C)

Yes. Nearest neighbour has high bias and low variance, it is likely that an instance will be surrounded by similar instances (and hence, similar classes) regardless of the training set. Unlike decision tree learning, there is effectively no way to increase the variance of nearest-neighbour, except by decreasing k.

D)

No. By averaging over multiple different models, the bias is actually more prominent than in any constituent model; unlike, for instance, boosting.

E)

Yes. By averaging over multiple models that were trained over multiple training sets, the effects of any particular training set (the variance) decrease.

Q7

A)

The H_MAP hypothesis is the hypothesis that maximises $P(D \mid h)P(h)$. The H_ML hypothesis assumes that all priors are equally likely and maximises only $P(D \mid h)$.

B)

Three nodes, and two arcs $Cloudy \rightarrow Play\ tennis$ and $Windy \rightarrow Play\ tennis$.

C)

$P(\oplus) = \frac{1}{3}$

$$P(\ominus) = \frac{2}{3}$$

$$P(Cloudy \mid \oplus) = \frac{1}{2}$$

$$P(Windy \mid \oplus) = \frac{1}{2}$$

$$P(Cloudy \mid \ominus) = \frac{1}{4}$$

$$P(Windy \mid \ominus) = \frac{1}{2}$$

D)

Excluding denominators,

$$\begin{align} P(\oplus \mid \lnot Cloudy, \lnot Windy) &= P(\lnot Cloudy \mid \oplus) P(\lnot Windy \mid \oplus) P(\oplus)\\ &=\frac{1}{2} \times \frac{1}{2} \times \frac{1}{3} = \frac{1}{12} \end{align}$$

$$\begin{align} P(\ominus \mid \lnot Cloudy, \lnot Windy) &= P(\lnot Cloudy \mid \ominus) P(\lnot Windy \mid \ominus) P(\ominus)\\ &=\frac{3}{4} \times \frac{1}{2} \times \frac{2}{3} = \frac{1}{4} \end{align}$$

$$\therefore \ominus$$

$$\begin{align} P(\oplus \mid \lnot Cloudy, Windy) &= P(\lnot Cloudy \mid \oplus) P(Windy \mid \oplus) P(\oplus)\\ &=\frac{1}{2} \times \frac{1}{2} \times \frac{1}{3} = \frac{1}{12} \end{align}$$

\begin{align} P(\ominus \mid \lnot Cloudy, Windy) &= P(\lnot Cloudy \mid \ominus) P(Windy \mid \ominus) P(\ominus)\\ &=\frac{3}{4} \times \frac{1}{2} \times \frac{2}{3} = \frac{1}{4} \end{align}

\therefore \ominus