**Model selection:**

**Complexity vs. goodness of fit**

- We can think of finding the best hypothesis as two tasks:
  - **Model selection** defines the hypothesis space and
  - **Optimization** finds the best hypothesis within that space
- How to select among models that are parameterized by size
  - With polynomials we have $size = 1$ for linear functions, $size = 2$ for quadratics, and so on
  - For decision trees, the size could be the number of nodes in the tree
- We want to find the value of the size parameter that best balances underfitting and overfitting to give the best test set accuracy

A wrapper takes a learning algorithm as an argument (DT learning for example)
- The wrapper enumerates models according to the size parameter
- For each size, it uses cross validation (say) on the learner to compute the average error rate on training and test sets
- We start with the smallest, simplest models
  - (which probably underfit the data),
  - and iterate, considering more complex models at each step,
  - until the models start to overfit
- The cross validation picks the value of $size$ with the lowest validation set error
- We then generate a hypothesis of that $size$ using all the data (without holding out any of it; eventually we should evaluate the returned hypothesis on a separate test set)
From error rates to loss

- Consider the problem of classifying emails as spam or non-spam
- It is worse to classify non-spam as spam than vice versa
- So a classifier with a 1% error rate, where almost all errors were classifying spam as non-spam, would be better than a classifier with only a 0.5% error rate, if most of those errors were classifying non-spam as spam
- Utility is what learners – like decision makers – should maximize
- In machine learning it is traditional to express utilities by means of **loss functions**
  - The loss function $L(x, y, \hat{y})$ is defined as the amount of utility lost by predicting $h(x) = \hat{y}$ when the correct answer is $f(x) = y$:
    
    $$
    L(x, y, \hat{y}) = U(\text{result of using } y \text{ given an input } x)
    - U(\text{result of using } \hat{y} \text{ given an input } x)
    $$

- Often a simplified version of the loss function is used: It is 10 times worse to classify non-spam as spam than vice versa:
  
  $L(\text{spam, nonspam}) = 1, \quad L(\text{nonspam, spam}) = 10$

- Note that $L(y, y)$ is always zero
- In general for real-valued data small errors are better than large ones
- Two functions that implement that idea are the absolute value of the difference (called the $L_1$ loss), and the square of the difference (called the $L_2$ loss)
- Minimizing error rate is formulated in the $L_{0/1}$ loss function
  
  **Absolute value loss**: $L_1(y, \hat{y}) = |y - \hat{y}|$

  **Squared error loss**: $L_2(y, \hat{y}) = (y - \hat{y})^2$

  **$0/1$ loss**: $L_{0/1}(y, \hat{y}) = 0$ if $y = \hat{y}$, else 1
• Let $P(X, Y)$ be a prior probability distribution over examples.

• Let $E$ be the set of all possible input-output examples.

• Then the expected **generalization loss** for a hypothesis $h$ (w.r.t. loss function $L$) is:

$$GenLoss_L(h) = \sum_{(x,y) \in E} L(y, h(x)) P(x,y)$$

• The best hypothesis $h^*$ is the one with the minimum expected generalization loss:

$$h^* = \arg \min_{h \in E} GenLoss_L(h)$$

• Because $P(x,y)$ is not known, the learning agent can only estimate generalization loss with **empirical loss** on the set of examples $E$:

$$EmpLoss_{L,E}(h) = \frac{1}{n} \sum_{(x,y) \in E} L(y, h(x))$$

• The estimated best hypothesis $\hat{h}^*$ is then the one with minimum empirical loss:

$$\hat{h}^* = \arg \min_{h \in E} EmpLoss_{L,E}(h)$$

## Regularization

• Earlier on we did model selection with cross-validation on model size.

• An alternative approach is to search for a hypothesis that directly minimizes the weighted sum of empirical loss and the complexity of the hypothesis, which we call the total cost:

$$Cost(h) = EmpLoss(h) + \lambda \text{ Complexity}(h)$$

$$\hat{h}^* = \arg \min_{h \in E} Cost(h)$$

• Here $\lambda$ is a parameter, a positive number that serves as a conversion rate between loss and hypothesis complexity.

• We still need to do cross-validation search to find the hypothesis that generalizes best, but this time with different values of $\lambda$.

• This process of explicitly penalizing complex hypotheses is called **regularization**.
18.5 The Theory of Learning

• Computational Learning Theory (COLT) [Valiant 1984]
• Valiant received the ACM Turing Award in 2010
• Any hypothesis that is seriously wrong will almost certainly be “found out” with high probability after a small number of examples, because it will make an incorrect prediction
• Thus, any hypothesis that is consistent with a sufficiently large set of training examples is unlikely to be seriously wrong: that is, it must be probably approximately correct (PAC)
• PAC-learning is based on the stationarity assumption: Future examples are going to be drawn from the same fixed distribution \( P(\mathcal{E}) = P(X, Y) \) as past examples
• We do not necessarily know the distribution
• Without this connection between the past and the future, learning would be practically impossible

\[ L(\mathcal{H}, \mathcal{N}) = \frac{1}{N} \sum_{i=1}^{N} L_{0/1}(y_i, h(x_i)) P(x_i, y_i) \]

• Let \( \mathcal{X} \) be the instance space from which the examples are drawn from with a stationary distribution
• \( \mathcal{H} \) is the hypothesis class and \( N \) denotes the number of training examples
• We assume that the true function \( f \) is a member of \( \mathcal{H} \)
• The error rate of a hypothesis \( h \) is defined as the expected generalization error for examples drawn from the stationary distribution

\[ error(h) = GenLoss_{L_{0/1}}(h) = \sum_{(x,y)} L_{0/1}(y, h(x)) P(x, y) \]

• A hypothesis \( h \) is called approximately correct if \( error(h) \leq \varepsilon \), where \( \varepsilon \) is a small constant
Let us examine a hypothesis $h_b$, which is not an $\varepsilon$-approximation of the true function.

Then $\text{error}(h_b) > \varepsilon$ and the probability that it agrees with a given example is at most $1 - \varepsilon$.

Since the examples are independent, the bound for $N$ examples is

$$P(h_b \text{ agrees with } N \text{ examples}) \leq (1 - \varepsilon)^N$$

The probability that the hypothesis class $\mathcal{H}$ contains at least one such hypothesis is at most $|\mathcal{H}|(1 - \varepsilon)^N$.

We would like to reduce the probability of this event below some small number $\delta$:

$$|\mathcal{H}|(1 - \varepsilon)^N \leq \delta$$

Then

$$N \geq \left(\frac{1}{-\ln(1 - \varepsilon)}\right)\left(\ln|\mathcal{H}| + \ln\left(\frac{1}{\delta}\right)\right)$$

Because $\ln(1 + \alpha) < \alpha$, we can achieve this if we allow the algorithm to see

$$N \geq \frac{1}{\varepsilon}\left(\ln|\mathcal{H}| + \ln\frac{1}{\delta}\right)$$

examples.

Thus, if a learning algorithm returns a hypothesis that is consistent with this many examples, then with probability at least $1 - \delta$, it has error at most $\varepsilon$.

In other words, it is probably approximately correct.

The number of required examples, as a function of $\varepsilon$ and $\delta$, is called the sample complexity of the hypothesis space.
Learning decision lists

- A *decision list* consists of a series of tests, each of which is a conjunction of literals.
- The tests (rules) are attempted in order.
- If a test succeeds, the decision list specifies the value to be returned.
- If a test fails, processing continues with the next test in the list.

Decision lists resemble decision trees, but their overall structure is simpler: they branch only in one direction.
In contrast, the individual tests are more complex.
Decision lists generalize decision trees.
If we allow tests of arbitrary size, then decision lists can represent any Boolean function.
On the other hand, if we restrict the size of each test to at most $k$ literals ($k$-DL), then it is possible for the learning algorithm to generalize successfully from a small number of examples.
Language $k$-DL includes as a subset the language $k$-DT, the set of all decision trees of depth at most $k$.
Let $k$-DL($n$) denote a $k$-DL language using $n$ Boolean attributes.
Let $\text{Conj}(n,k)$ denote the conjunctions of at most $k$ literals using $n$ attributes.

The number of conjunctions of $k$ literals from $n$ attributes is given by

$$\left| \text{Conj}(n,k) \right| = \sum_{i=1}^{k} \left( \begin{array}{c} 2n \\ i \end{array} \right) = O(n^k)$$

In a decision list each conjunction can be attached to either a $\text{Yes}$ or $\text{No}$ outcome or can be absent from the decision list.

Hence, there are at most $3!^{\text{Conj}(n,k)}$ distinct sets of component tests.

Each of these sets of tests can be in any order, so

$$\left| k-\text{DL}(n) \right| \leq 3^{\text{Conj}(n,k)} \left| \text{Conj}(n,k) \right|!$$

Combining the above estimates yields

$$\left| k-\text{DL}(n) \right| \leq 2^O(n^k \log_2(n^k))$$

We plug this into the formula for sample complexity to show that the number of examples needed for PAC-learning is

$$N \geq (1/\varepsilon) \left( |O(n^k \log_2(n^k))| + \ln(1/\delta) \right)$$

Hence, a $k$-DL function is PAC-learnable from a number of examples polynomial in $n$.

Therefore, any algorithm that returns a consistent decision list will PAC-learn a $k$-DL function in a reasonable number of examples, for small $k$.

The following greedy algorithm will output a consistent decision list (if one exists):
DL-LEARNER(\(S\))

**Input:** \(S\) training examples

**Output:** a decision list or failure

1. if \(S = \emptyset\) then return the trivial decision list \(No\);
2. Choose a test \(t\) that matches a subset of examples \(S_t \neq \emptyset\) such that the members are all positive or negative;
3. if there is no such \(t\) then fail;
4. if all examples in \(S_t\) are positive then \(o \leftarrow Yes\) else \(o \leftarrow No\);
5. \(L \leftarrow DL\-LEARNER(S\setminus S_t)\);
6. return a decision list with initial test \(t\) and outcome \(o\) and remaining tests given by \(L\);

### 18.6 Regression and Classification with Linear Models

- The hypothesis space of linear functions of continuous-valued inputs has been used for hundreds of years
- A univariate linear function (a straight line) with input \(x\) and output \(y\) has the form
  \[ y = w_1 x + w_0 \]
  where \(w_0\) and \(w_1\) are real-valued coefficients to be learned
- Let \(w\) be the vector \([w_0, w_1]\) and define
  \[ h_w(x) = w_1 x + w_0 \]
- The task of finding the \(h_w\) that best fits the data is called linear regression
- To fit a line to the data, all we have to do is find the values of the weights \([w_0, w_1]\) that minimize the empirical loss
It is traditional (going back to Gauss) to use the squared loss function $L_2$ summed over all the training examples

$$\text{Loss}(h_w) = \sum_{j=1}^{N} L_2(y_j, h_w(x_j)) = \sum_{j=1}^{N} (y_j - h_w(x_j))^2 = \sum_{j=1}^{N} (y_j - (w_1x_j + w_0))^2$$

We would like to find $w^* = \arg \min_w \text{Loss}(h_w)$

The sum $\sum_j (y_j - (w_1x_j + w_0))^2$ is minimized when its partial derivatives with respect to $w_0$ and $w_1$ are zero

$$\frac{\partial}{\partial w_0} \sum_{j=1}^{N} (y_j - (w_1x_j + w_0))^2 = 0$$

$$\frac{\partial}{\partial w_1} \sum_{j=1}^{N} (y_j - (w_1x_j + w_0))^2 = 0$$
These equations have a unique solution

\[
w_1 = \frac{N(\sum x_j y_j) - (\sum x_j)(\sum y_j)}{N(\sum x_j^2) - (\sum x_j)^2}
\]

\[
w_0 = \frac{(\sum y_j - w_1(\sum x_j))}{N}
\]

- The weight space defined by \(w_0\) and \(w_1\) is convex
- This is true for every linear regression problem with an \(L_2\) loss function, and it implies that there are no local minima

To go beyond linear models, we will need to face the fact that the equation defining minimum loss will often have no closed-form solution

- Instead, we will face a general optimization search in continuous weight space
- Such problems can be addressed by a hill-climbing algorithm that follows the gradient of the function to be optimized
- Because we are trying to minimize the loss, we will use gradient descent
- We choose any starting point in weight space and then move to a neighboring point that is downhill, repeating until we converge on the minimum possible loss
\(\mathbf{w} \leftarrow \text{any point in the parameter space}\)

\textbf{loop} until convergence \textbf{do}

\textbf{for} each \(w_i\) in \(\mathbf{w}\) \textbf{do}

\[w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} \text{Loss}(\mathbf{w})\]

- The step size parameter \(\alpha\) is usually called the \textit{learning rate} when we are trying to minimize loss in a learning problem.
- It can be a fixed constant, or it can decay over time as the learning problem proceeds.
- For univariate regression, the loss function is a quadratic function, so the partial derivative will be a linear function.

\(\text{Let's consider the case of only one training example, } (x, y):\)

\[\frac{\partial}{\partial w_i} \text{Loss}(\mathbf{w}) = \frac{\partial}{\partial w_i} (y - h_w(x))^2\]

\[= 2(y - h_w(x)) \times \frac{\partial}{\partial w_i} (y - h_w(x))\]

\[= 2(y - h_w(x)) \times \frac{\partial}{\partial w_i} (y - (w_1 x + w_0))\]

- Applying this to both \(w_0\) and \(w_1\) we get:

\[\frac{\partial}{\partial w_0} \text{Loss}(\mathbf{w}) = -2(y - h_w(x))\]

\[\frac{\partial}{\partial w_1} \text{Loss}(\mathbf{w}) = -2(y - h_w(x)) \times x\]
Plugging these values back to the gradient descent update rule (folding the constant 2 into the learning rate $\alpha$), we get the following learning rules for the weights:

$$
\begin{align*}
    w_0 &\leftarrow w_0 + \alpha (y - h_w(x)) \\
    w_1 &\leftarrow w_1 + \alpha (y - h_w(x)) \cdot x
\end{align*}
$$

Intuitively:

- if $h_w(x) > y$ – the output of the hypothesis is too large – reduce $w_0$ a bit
- Reduce $w_j$ if $x$ was a positive input but increase $w_j$ if $x$ was a negative input

For $N$ training examples the derivative of a sum is the sum of the derivatives, and we have

$$
\begin{align*}
    w_0 &\leftarrow w_0 + \alpha \sum_j (y_j - h_w(x_j)) \\
    w_1 &\leftarrow w_1 + \alpha \sum_j (y_j - h_w(x_j)) \cdot x_j
\end{align*}
$$

Multivariate linear regression

- In multivariate linear regression each example $x_j$ is an $n$-element vector
- Our hypothesis space is the set of functions of the form
  $$
  h_{sw}(x_j) = w_0 + w_1 x_{j,1} + \cdots + w_n x_{j,n} = w_0 + \sum_i w_i x_{j,i}
  $$
- To make $w_0$ in par with other weights, we can invent a dummy input attribute $x_{j,0}$ which is defined as always equal to 1
- Then $h$ is simply the dot product of the weights and the input vector (or equivalently, the matrix product of the transpose of the weights and the input vector)
  $$
  h_{sw}(x_j) = w \cdot x_j = w^T x_j = \sum_i w_i x_{j,i}
  $$
- The best vector of weights, $w^*$, minimizes square-error loss over the examples:
  $$
  w^* = \arg \min_w \sum_j L(y_j, w \cdot x_j)
  $$
• Very much like in the univariate case, gradient descent will reach the (unique) minimum of the loss function; the update equation for each weight $w_i$ is
$$w_i \rightarrow w_i + \alpha \sum_j x_{ij}(y_j - h_w(x_j))$$
• It is also possible to solve analytically the $w$ that minimizes loss
• Let $y$ be the vector of outputs for the training examples, and $X$ the data matrix, i.e., the matrix of inputs with one $n$-dimensional example per row
• Then the solution
$$w^* = (X^TX)^{-1}X^Ty$$
minimizes the squared error
• In high-dimensional spaces it is possible that some dimension that is actually irrelevant appears by chance to be useful, resulting in overfitting

Thus, regularization on multivariate linear functions to avoid overfitting is common
• In regularization we minimize the total cost of a hypothesis, counting both the empirical loss and the complexity of the hypothesis
$$Cost(h) = EmpLoss(h) + \lambda Complexity(h)$$
• For linear functions the complexity can be specified as a function of the weights
• We can consider a family of regularization functions:
$$Complexity(h_w) = L_q(w) = \sum_i |w_i|^q$$
• With $q = 1$ we have $L_1$ regularization, which minimizes the sum of the absolute values; with $q = 2$, $L_2$ regularization minimizes the sum of squares
• Loss and regularization functions need not be used in pairs: you could use $L_2$ loss with $L_1$ regularization, or vice versa
• Which regularization function to pick depends on the specific problem
• $L_1$ regularization has an important advantage: it tends to produce sparse models – it often sets many weights to zero, effectively declaring the corresponding attributes to be irrelevant
• Hypotheses that discard attributes can be easier for human to understand, and may be less likely to overfit
• The number of examples required to find a good $h$ is linear in the number of irrelevant features for $L_2$ regularization, but only logarithmic with $L_1$ regularization

Linear classifiers with a hard threshold

• Linear functions can be used to do classification by finding a decision boundary (a linear separator) – a line or a surface in higher dimensions – that separates the two classes (if the data is linearly separable)
  \[ h_w(x) = 1 \text{ if } w \cdot x \geq 0 \text{ and } 0 \text{ otherwise} \]
• We can think of $h$ as passing the linear function $w \cdot x$ through a threshold function
  \[ h_w(x) = \text{Threshold}(w \cdot x) \]
  where $\text{Threshold}(z) = 1 \text{ if } z \geq 0 \text{ and } 0 \text{ otherwise}$
• For regression minimizing the loss could be considered through closed form solution and by gradient descent in weight space
• Here we cannot do either of those things because the gradient is zero almost everywhere in weight space except at those points where $w \cdot x = 0$, and at those points the gradient is undefined
There is a simple weight update rule that converges to a solution. It provides a linear separator that classifies the data perfectly provided the data are linearly separable. For a single example \((x, y)\), we have the \textit{perceptron learning rule}

\[
    w_i \leftarrow w_i + \alpha (y - h_w(x)) \cdot x_i
\]

which is essentially identical to the update rule for linear regression. Because we are considering 0/1 classification problem, the behavior is somewhat different.
Both the true value \( y \) and the hypothesis output \( h_w(x) \) are either 0 or 1:

- If \( y = h_w(x) \) the output is correct, and the weights are not changed.
- If \( y = 1 \) but \( h_w(x) = 0 \), then \( w_i \) is increased when the corresponding input \( x_i \) is positive and decreased when \( x_i \) is negative.
  - This makes sense, because we want to make \( w \cdot x \) bigger so that \( h_w(x) \) outputs a 1.
- If \( y = 0 \) but \( h_w(x) = 1 \), then \( w_i \) is decreased when the corresponding input \( x_i \) is positive and increased when \( x_i \) is negative.
  - This makes sense, because we want to make \( w \cdot x \) smaller so that \( h_w(x) \) outputs a 0.

Typically the learning rule is applied one example at a time, choosing examples at random (stochastic gradient descent).

- The perceptron rule may not converge to a stable solution for fixed learning rate \( \alpha \).
- However, if \( \alpha \) decays as \( O(1/t) \) where \( t \) is the iteration number, then the perceptron learning rule converges to a minimum error solution when examples are presented in a random sequence.
- If data points are not linearly separable, the learning rule may fail to converge.
- Finding the minimum-error solution is NP-hard.