CS 251 Data Analysis and Visualization, Spring 2016

Dr. Bruce A. Maxwell
Department of Computer Science
Colby College

Course Description

This course prepares students to apply computational approaches to data analysis and visualization to real information from a variety of disciplines and applications. Data visualization is the interactive visual exploration of information using 2-D and 3-D graphics using techniques that highlight patterns and relationships. Data analysis incorporates data management, data transformations, statistical analysis, data mining, and machine learning. Through programming projects, students will gain hands-on experience with the fundamentals of data analysis and visualization using data from active research projects at Colby and other institutions.

Prerequisites: CS 231

1 Data Visualization

Data is a multi-faceted word. In philosophy, data are things known or assumed as facts. The original Latin root, *datum*, of which data is the plural, means 'something given'. In computing, data generally means sequences of bits stored in various digital devices.

Data can take many forms. It could be a number, many numbers, a book, a collection of newspapers, readings from a scientific instrument, an access log from a web site, a picture, a sound clip, or a video. Data does not have to be digital, nor does it have to be numeric. Data is most convenient to analyze if it is digital, but it's important to remember that there is valuable data stored in places other than computer hard drives (but there is a huge amount of data on computer hard drives).

One thing to remember is that data is useless without **meta-data**. Meta-data is what tells you the meaning of the numbers or words, or images. Given the list of numbers [69, 72, 73, 64, 66], you can run a number of statistical analyses on them, but you cannot extract any knowledge from the numbers without knowing what they represent, which is given by the meta-data. If you were told they represented the heights of five people in inches, you could then extract knowledge and answer questions about the heights of these individuals. On the other hand, if you were told they represented the number of votes received by each of five candidates in an election for a single position, you would extract very different knowledge from the same set of numbers.

Data visualization is, at its core, the process of connecting data with our brains. While data visualization primarily refers to using our eyes to connect the data with our brains, data visualization is also possible via all of our other senses. The goal of data visualization is to enable the computational machinery in our brains to identify patterns, trends, or other interesting and salient characteristics of the data. While not all of the patterns we find in data are meaningful, our brains have incredible pattern identification and recognition capabilities, and data visualization enables us to make effective use of it.

Data Analysis is the process of computationally extracting information from data. Note that both visualization and analysis have the same ultimate goal: synthesizing knowledge from data. Data visualization enhances the ability of our biological computers to extract knowledge. Data analysis uses the computer to automate the process. Both have a role to play, as it can be difficult to define data analysis strategies without having some idea of the structures that exist in the data. Conversely, visualization can be difficult without some simple computational analyses to make the patterns more obvious.

1.1 Data Terminology

- **data point**, **data vector**, or **feature vector**: one or more numbers representing a single measurement event.
- **variable** or **feature**: a symbol that connects a set of numbers to a meaningful description. A variable/feature usually refers to a single number within a data point or data vector.
- **multi-variable data**: a data set whose data points consist of more than one measurement.
- **dimension**: the number of variables/features/measurements in a data point.
- **min**: the minimum value of a variable within a data set.
- **max**: the maximum value of a variable within a data set.
- **range**: the upper and lower bounds of potential values for a variable, sometimes refers to max - min.
- **independent variable**: a direct measurement or value that does not depend on another value in the data point. In the example $y = mx + b$, the variables m, x and b are independent variables.
- **dependent variable**: a variable calculated from or that is a result of other variables or measurements in the data point. In the example $y = mx + b$, y is the dependent variable because it is completely defined by the variables on the right side of the equation.
- **missing data**: sometimes a data point will not contain all of the measurements that other data points in the set possess.
- **meta-information** or **meta-data**: a description of the variables in a data set, often including their source, method of measurement, valid range, or valid values for the variable.
- **precision**: a description of the number of significant figures in a measurement, which is based on the repeatability, or reproducibility of a measurement. Note that a repeatable measurement is not necessarily correct.
- **accuracy**: a description of how close a measurement is to the true value.

Precision and accuracy are important concepts in data analysis and visualization. A measurement with high precision is not necessarily accurate, and an accurate measurement is not necessarily precise. A data set with high accuracy and high precision is called *valid*.

In many data sets, the original precision of the data gets lost during analysis or transformation by a computer. For example, data that is originally integral, with a precision of ± 0.5 may be converted to a floating point representation as part of a transformation process and stored to a file using six decimal places. The number of significant figures in the data does not change, however, which means that the final representation likely contains meaningless digits. The original precision of the data will usually be lost unless it is included in the meta-data.

Data Types

Data comes in many forms. Not surprisingly, there is generally a correlation between data types in a computer language and data types used in collecting measurements. Data will always have a native format, which defines its **precision** and the number of possible values. Such information should be saved in the meta-information for a data set (but is not always). As data is transformed, or even read from and written back to a file, it can undergo changes. To preserve the quality of data, all intermediate representations of data should be as precise as the original data set, otherwise information will be lost.

Common categories of digital data include:

- numeric: real, integer
- categorical
- binary
- strings

The type of data is independent of the method of storing or organizing the data. Numbers can be written out as text files, organized in databases in binary form, or stored in standard image formats. The form of organization depends largely on how the data was captured and how it will be analyzed or used. Data that needs to be stored, accessed, searched, and backed up is often stored in formal databases.

All data is stored in a binary format within a computer. How the computer generates a visualization from the data is completely up to the programmer. You can create images from the raw data in text strings, if that seems like a good idea.

Data space

Each variable of a data vector is a separate dimension. Real, integral, and categorical variables create different kinds of spaces. Real valued variables create a continuous dimension, while integral and categorical variables create discrete dimensions. The extent of each dimension is determined by the minimum and maximum possible values of the variable.

It can be important to remember that all numbers on a computer are discrete. Floating point and double formats can only represent a fixed set of numbers, although their precision is sufficient for most applications. The number of unique values a 32-bit floating point format can represent is actually fewer than the number of unique values a standard integer can represent.

The data space is the native coordinate system for the data set. The section of the data space where the data resides is arbitrary. The data set does not, for example, have to be anywhere close to the origin. Likewise, the units and values of the native coordinate system are arbitrary, as is the range of the data, even relative to the minimum and maximum possible values. Missing data adds additional complexity to visualization, since not all dimensions of the data point will necessarily be missing. The arbitrary nature of data makes it difficult to design an algorithm that provides an appropriate visualization for any data set.

The general solution to the problem of visualization is to develop a standard visualization process and provide the user control over specific parameters of the process. The user can then control the visualization to suit their needs. Note that there are a number of 'standard' visualization processes and different programs take different approaches.

Data formats:

Once data exists in digital form, it has to be stored. The method of storage has a significant impact on how easily you can write a program to read and analyze the data. For example, consider doing an analysis of word frequency using digitized photos of old texts compared to using a set of web pages. Both sets of data are digital, but one is easy to parse, while the other is difficult.

One reason formal databases are generally the method of choice for large, heavily used data sets is that they are optimized for enabling fast searches and queries for particular pieces of data. Furthermore, they provide the data in a ready-to-use format that interfaces cleanly with scripts and programs.

- Databases: SQL, concept of database, a table, and an entry (or row)
- ASCII files, text files, can always look at them with a text editor
- Excel files, proprietary, CVS (comma-delimited), tab-delimited, space delimited
- XML files, text files with tags that label the contents
- HTML files, a subset of XML used for transferring content over the web
- Code files, data is sometimes stored as computer code
- ArcView files, proprietary, used for geographic information systems
- Binary files, data is packed in a binary format, usually particular to an application
- Images TIFF, GEOTIFF, JPG, GIF, PNG, ...
- Videos MPG, WMV, MOV, GIF, ...

Attributes include the order of the information, the use of comment markers to store meta-data, portability, the use of a variety of separators (tabs, spaces, commas, less-than and greater-than symbols), magic numbers, redundancy, error detection and recovery, and whether the information is human-readable.

Data Precision

All data has a native precision. The native precision depends on the method of data capture and the precision of the measurement. Native precision is not always the same as the number of significant figures used to store the information. We could use six significant figures to store data captured with only three significant figures of precision. In that situation, the extra significant figures fool us into thinking the data is more precise than its native precision. The opposite can also be true. We could take measurements with three significant figures and store only two. In that case, the native precision is reduced by the method of storage.

Whenever data is transformed from one format to another, there is the real danger of the data being changed because of representational errors. The more transformations occur, the more likely that errors will creep into the data. For example, some numbers cannot be exactly represented in a floating point format, or require infinite series of decimals. If the values are written out to a text file with a fixed number of decimals, they get truncated. When the data is read back into the computer, the internal representation of the number is different than it was before the data was written to a file. If the process is repeated several times, the errors accumulate.

Responsible data management requires that we keep the data as close to its original form as possible and in a format that retains at least as much precision as the original measurements. To avoid unnecessary quantization errors, we always want to start with the original source and build our processing chain in as few steps as possible. Any change to the data format introduces quantization errors.

1.2 Visualization

Visualization is the art and science of presenting information to a user. Visualization is a relatively new field, although work in human perception, which is related to visualization, has a long history. Many of the findings in human perception are the basis for techniques used in visualization intended to make certain kinds of information easily accessible.

The purpose of visualization is not to generate pretty pictures, but to present information in such a way that it facilitates the extraction of knowledge from the data. Knowledge comes from seeing and understanding patterns in the data. There are two types of broad goals in visualization, and in data analysis more generally.

1. To answer concrete questions about a given problem.
2. To discover previously unknown facts about a problem.

The motivation for visualization as a tool for answering concrete questions is that it can allow rapid perceptual understanding of the answers. Visualization can also reduce the amount of time required to complete the task [Telea, 2008]. In some cases visualization can take the place of quantitative numerical answers, and in other cases it complements the information.

As a simple example, consider the fact that Tyrannasaurus Rex could reach a size of 13m and 7 metric tons. Just looking at the numbers, you realize that T-Rex was a big animal, but it is difficult to understand exactly how big. The visualization of T-Rex shown in figure 1 provides a much more intuitive understanding of the same facts. You realize very quickly that you would be nothing more than a snack.

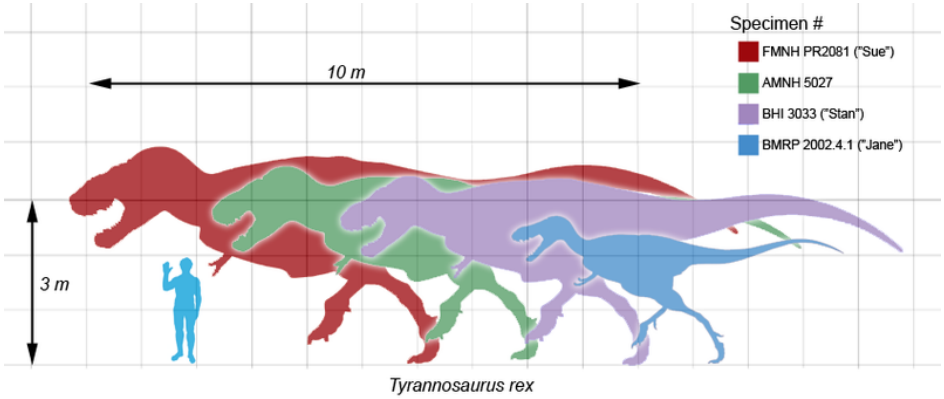

Figure 1: Tyrannosauras Rex shown with a human for scale. [Image from Wikipedia]

The motivation for visualization as a tool for exploration is easier to understand. Much of our brainpower is devoted to understanding visual input, and our brains are very good at identifying structure and patterns in visual data. In fact, we have to be careful when creating visualizations because our brains are quite capable of hallucinating patterns where none actually exist.

Modern visualization systems increasingly focus on making the visualization process interactive. Being able to zoom, rotate, pan, and transform data interactively permits exploration of data sets in ways that were previously impossible. Interactive visualization systems also permit new kinds of representations designed to show structure in complex data sets. Unfortunately, we are still limited to viewing data in a small number of dimensions. However, the number of dimensions may be greater than you expect, as in addition to space and time, there are other visual axes for displaying information such as color, size, and texture.

1.3 Coordinate systems

Coordinate systems are at the heart of visualization techniques. Most of the work involved in generating visualizations is simply transforming data from one coordinate system to another. The starting coordinate system is the data space itself, the native coordinate system of the data. The final coordinate system is the visualization device, for example, a window on a computer screen. It is useful to create a number of intermediate coordinate systems between the initial and final coordinates to make the process of visualization tractable. The following is a fairly standard sequence of coordinate systems used in visualization.

Data Coordinates: the native coordinates of the data space. The max, min, and average values of each variable are defined in this space. For any data set, there is a bounding box in data coordinates within which all of the data resides.

View Volume Coordinates: the volume of the data space the user wants to view. The volume may be defined by the max and min variable values, or it may include only a subset of the data. If the view volume is constrained to be axis-aligned, then it is defined by an origin and an extent in each data variable direction. If the view volume can have arbitrary orientations, then it is defined by an origin and a set of orthonormal axes that define its orientation. The size of the volume is determined by an extent measured along the orthonormal axes.

Normalized Viewing Coordinates: A scaling of the data so that the visible data points–those within the view volume–fit within the range [0, 1] in all dimensions.

Screen Coordinates: A scaling, possibly a translation, and a projection to convert the normalized coordinates into screen coordinates where they are drawn.

1.4 Coordinate Transformations

The three basic transformations we will be using to manipulate data are the rigid geometric transformations: translation, scaling, and rotation. In addition, for viewing three dimensional data we will have to execute a projection from 3 dimensions into 2. Rigid transformations are most easily implemented using matrices, with a data point represented as a column vector. In code, a matrix is simply a 2D array of numbers, and matrix-matrix and matrix-vector multiplication are straightforward to implement.

1.4.1 Matrix Multiplication

Matrix multiplication is built upon the dot project, also called the scalar product or inner product, of two vectors. Given two N-element vectors $\vec{A} = \begin{bmatrix} a_0 & a_1 & \dots & a_{N-1} \end{bmatrix}$ and $\vec{B} = \begin{bmatrix} b_0 & b_1 & \dots & b_{N-1} \end{bmatrix}$ their scalar product is defined as the multiplication of corresponding elements in the two vectors, followed by the summation of the products. The dot product of two vectors is always a single scalar value. The dot product also has a geometric interpretation: its value is equal to the product of the length of the two vectors and the cosine of the angle between them. Therefore, the dot product of two normal vectors (unit length) is a fast way to compute the cosine of the angle between them.

$$d(\vec{A}, \vec{B}) = \sum_{i=0}^{N-1} a_i b_i = ||A||||B|| \cos\theta \tag{1}$$

Each entry in the matrix resulting from the multiplication of two input matrices is the result of a dot product. The dot product uses the corresponding row in the first matrix and the corresponding column in the second. For example, consider two 3x3 matrices given below and their product written as a set of dot products.

$$\begin{bmatrix} d(a_{0,*}, b_{*,0}) & d(a_{0,*}, b_{*,1}) & d(a_{0,*}, b_{*,2}) \\ d(a_{1,*}, b_{*,0}) & d(a_{1,*}, b_{*,1}) & d(a_{1,*}, b_{*,2}) \\ d(a_{2,*}, b_{*,0}) & d(a_{2,*}, b_{*,1}) & d(a_{2,*}, b_{*,2}) \end{bmatrix} = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{1,0} & a_{1,1} & a_{1,2} \\ a_{2,0} & a_{2,1} & a_{2,2} \end{bmatrix} \begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} \\ b_{1,0} & b_{1,1} & b_{1,2} \\ b_{2,0} & b_{2,1} & b_{2,2} \end{bmatrix} \quad (2)$$

The same rule applies for matrices that are not square. The only requirement for matrix multiplication to be valid is that the number of columns in the first matrix must match the number of rows in the second matrix. If the input matrices are RxN and NxC, then the output matrix will be RxC. Matrix multiplication is not commutative, in general.

For our purposes, the most important case of matrix multiplication with non-square matrices is the case of multiplying a matrix and a vector.

$$\begin{bmatrix} d(a_{0,*}, b_{*,0}) \\ d(a_{1,*}, b_{*,0}) \\ d(a_{2,*}, b_{*,0}) \end{bmatrix} = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{1,0} & a_{1,1} & a_{1,2} \\ a_{2,0} & a_{2,1} & a_{2,2} \end{bmatrix} \begin{bmatrix} b_{0,0} \\ b_{1,0} \\ b_{2,0} \end{bmatrix} \quad (3)$$

1.4.2 Homogeneous Coordinates

Homogeneous coordinates are the basis for building a generic manipulation system for rigid objects. The homogeneous coordinates for a 2-D Cartesian point (x_c, y_c) are (x, y, h). The relationship between the homogeneous and standard coordinates is given in (4).

$$\begin{aligned} x_c &= \frac{x}{h} \\ y_c &= \frac{y}{h} \end{aligned} \quad (4)$$

A 3-D Cartesian point (x_c, y_c, z_c) has the homogeneous representation (x, y, z, h), and the normalized form is the same as the 2-D case, just extended to the z-axis.

Note that homogeneous coordinates permit many representations of a single 2-D Cartesian point. In the standard, or normalized form of homogeneous coordinates $h = 1$. In this course, we will always maintain homogeneous coordinates in their normalized form. Using homogeneous coordinates lets us implement transformations such as translation, rotation, and scaling using a single matrix representation.

1.4.3 Cross Product

The cross product, also called the outer product, or vector product generates a third vector that is orthogonal to both original vectors. For 3D vectors, the cross product is defined as in (5).

$$\vec{v}_0 \times \vec{v}_1 = \begin{bmatrix} y_0 z_1 - y_1 z_0 \\ z_0 x_1 - z_1 x_0 \\ x_0 y_1 - x_1 y_0 \end{bmatrix} = \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix} \times \begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} \quad (5)$$

We ignore the homogeneous coordinate when calculating the cross product. As with the dot product, the cross product also has a geometric interpretation. The length of the cross product result is equal to the product of the lengths of the input vectors times the sine of the angle between them.

$$||\vec{v}_0 \times \vec{v}_1|| = ||\vec{v}_0||||\vec{v}_1|| \sin\theta \tag{6}$$

1.4.4 Geometric Transformations

Translation

$$\begin{bmatrix} x_n \\ y_n \\ z_n \\ 1 \end{bmatrix} = \begin{bmatrix} x_0 + t_x \\ y_0 + t_y \\ z_0 + t_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{bmatrix} \tag{7}$$

Rotation about the Z-axis

$$\begin{bmatrix} x_n \\ y_n \\ z_n \\ 1 \end{bmatrix} = \begin{bmatrix} x_0\cos\theta - y_0\sin\theta \\ x_0\sin\theta + y_0\cos\theta \\ z_0 \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{bmatrix} \tag{8}$$

Rotation about the X-axis

$$\begin{bmatrix} x_n \\ y_n \\ z_n \\ 1 \end{bmatrix} = \begin{bmatrix} x_0 \\ y_0\cos\theta - z_0\sin\theta \\ y_0\sin\theta + z_0\cos\theta \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{bmatrix} \tag{9}$$

Rotation about the Y-axis

$$\begin{bmatrix} x_n \\ y_n \\ z_n \\ 1 \end{bmatrix} = \begin{bmatrix} x_0\cos\theta + z_0\sin\theta \\ y_0 \\ -x_0\sin\theta + z_0\cos\theta \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{bmatrix} \tag{10}$$

Scaling

$$\begin{bmatrix} x_n \\ y_n \\ z_n \\ 1 \end{bmatrix} = \begin{bmatrix} x_0 s_x \\ y_0 s_y \\ z_0 s_z \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ 1 \end{bmatrix} \tag{11}$$

Because each 2-D transformation is represented by a 3x3 matrix, we can use matrix multiplication and manipulation to concatenate transformations. Since we use column vectors for our coordinates, the order of operation moves right to left. The right-most transformation executes first.

- Order doesn't matter when concatenating only translations, only rotations, or only scales.
- Order matters when you mix different transformation type.

1.4.5 Coordinate Transformation Process

To change coordinate systems we can follow a three step process.

1. Translate the data so that the new origin becomes zero. If the origin of the view volume is at $(2, 3)$, for example, then by translating every data point by $(-2, -3)$ we move the point $(2, 3)$ in the data space to point $(0, 0)$ in the view volume space.

2. Orient the data space axes to the new coordinate system axes. If the view volume is already axis aligned, then this step is not necessary. But if the view volume can have an arbitrary alignment, then we need to rotate the data.

   In general, if you have a coordinate system defined by three orthonormal vectors (u_x, u_y, u_z), (v_x, v_y, v_z) and (w_x, w_y, w_z), then the matrix to align the coordinate system with the world axes is given below.

$$X_{\text{align}}(\vec{u}, \vec{v}) = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{12}$$

   Note that for 2D data, $u_z = 0$, $v_z = 0$, and $\vec{w} = (0, 0, 1)$.

3. Scale the data. If the new coordinate system uses a different system of units, then scale the data to fit the desired units. For example, when moving to normalized view coordinates, we want the maximum visible variable value along each axis to scale to 1. Therefore, we want to scale each dimension by the maximum visible value in that dimension.

Example: 2D Data manipulation

Consider a data set with a range of [-15, -10] in dimension A and a range of [1, 2] in dimension B. We want to view the whole data set on a screen that is 500 x 300.

1. Convert to view volume coordinates. The origin of the view volume is $(x_0, y_0) = (-15, 1)$ and its extent is $(dx, dy) = (5, 1)$. Therefore, we need to translate the point $(-15, 1)$ in data space to $(0, 0)$ using the transformation $T(15, -1)$. As we're using axis aligned coordinates, we do not need to rotate the data.

2. Convert to normalized view volume coordinates. The range of the data is (5, 1), so we need to scale the data using the transformation $S(\frac{1}{5}, \frac{1}{1})$

3. Convert to screen coordinates. The screen coordinates have their origin in the upper left corner, and the extent of the output image should be (500, 300). The first step is to scale the data to match the output screen size and flip the y-axis using the transformation $S(500, -300)$. The second step is to translate the origin to the lower left corner of the screen using the translation $T(0, 300)$.

Example: 3D Data manipulation

Consider a 3D data set with a range of [6, 10] in dimension A, [10, 20] in dimension B, and [20, 40] in dimension C. The mean value of the data is $\mu = (8, 15, 30)$. You want to set up the view such that the user is looking at the mean data location from the point $v\vec{r}p = (10, 20, 40)$ using a view volume that has an extent of [20, 20, 20]. To properly orient the view volume, we also need to know which direction is up. We can use dimension B as a default up direction, in which case $v\vec{u}p = (0, 1, 0)$.

1. Convert to view volume coordinates. If we position the viewer at the center of one end of the view volume, the process has three steps.
   (a) Translate the view reference point to the origin $T(-10, -20, -40)$
   (b) Orient the axes of the data space and the view volume. The three axes of our view space are the View Plane Normal, which is the direction we're looking, the View Up Vector, and the U vector. We can calculate them using the following process.

$$v\vec{p}n = \text{lookat} - \text{viewer} = (8, 15, 30) - (10, 20, 40) = (-2, -5, -10) \tag{13}$$

$$\vec{u} = v\vec{u}p \times v\vec{p}n \tag{14}$$

$$v\vec{u}p' = v\vec{p}n \times \vec{u} \tag{15}$$

   Use normalized versions of these three vectors to orient the axes.
   (c) Once the axes are oriented, we want to shift the volume so the lower left corner of the viewing face is at the origin. If the viewing face has size (du, dv), then the translation is $T(0.5du, 0.5dv) = T(10, 10)$.
2. Convert to normalized view coordinates. Scale each dimension of the view volume to 1:
   $S(\frac{1}{du}, \frac{1}{dv}, \frac{1}{dw}) = (0.05, 0.05, 0.05)$.
3. Convert to screen coordinates. Scale the two dimensions of the view volume that are perpendicular to the view direction to the screen coordinates, flip the axes, and then translate. Note that both the X and Y axes are reversed from screen coordinates because of the way we set up our view reference coordinate system.
   (a) Scale and flip the coordinate systems: $S(-s_x, -s_y)$
   (b) Translate so the window is all positive: $T(s_x, s_y)$

In summary, 3D viewing from arbitrary locations is a bit more complex than 2D. The important parameters are the **View Reference Point**, which is the center of the view volume face through which the viewer is looking, the **View Up Vector**, which orients the view volume, and the **View Volume Extent**, which is given in View Reference coordinates (which have units similar to the data space). From these parameters we can calculate all of the necessary transformations into normalized view coordinates. Note that, if the data space has drastically different ranges, then arbitrary viewing may not produce good visualizations without scaling the data so that distances in each dimension have roughly similar meanings. We'll be looking at how to do this a bit later.

1.4.6 Generic Orthographic Viewing Pipeline

Parameters

- VRP: view reference point and center of the view window; origin of the view reference coordinates
- $\vec{\text{VPN}}$: view plane normal, direction of viewing
- $\vec{\text{VUP}}$: view up vector, up orientation of the view volume
- $\vec{U}$: x-axis of view reference coordinates
- extent: (E_x, E_y, E_z), size of the bounding box in data space in view reference coordinates
- screen: (s_x, s_y), size of the output device window in pixels

Process

1. Set the view transformation matrix to a 4x4 identity matrix.
$$V = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{16}$$

2. Translate VRP to the origin.
$$V = T(-\text{VRP}_x, -\text{VRP}_y, -\text{VRP}_z)V \tag{17}$$

3. Build the view reference coordinates. Normalize the axes $\vec{U}$, $\vec{\text{VUP}}$, and $\vec{\text{VPN}}$ by dividing each vector by its length (square root of sum of squares).
$$\begin{aligned} \vec{U} &= \vec{\text{VUP}} \times \vec{\text{VPN}} \\ \vec{\text{VUP}}' &= \vec{\text{VPN}} \times \vec{U} \end{aligned} \tag{18}$$

4. Align the axes.
$$V = \begin{bmatrix} U_x & U_y & U_z & 0 \\ \text{VUP}'_x & \text{VUP}'_y & \text{VUP}'_z & 0 \\ \text{VPN}_x & \text{VPN}_y & \text{VPN}_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} V \tag{19}$$

5. Translate the lower left of the view window to the origin. (option 2: skip this step)
$$V = T(\frac{1}{2}E_x, \frac{1}{2}E_y, 0)V \tag{20}$$

6. Scale the view volume to normalize the view volume.
$$V = S(\frac{1}{E_x}, \frac{1}{E_y}, \frac{1}{E_z})V \tag{21}$$

7. Scale to screen coordinates and invert the x and y axes.
$$V = S(-s_x, -s_y, 1)V \tag{22}$$

8. Translate by the screen size (option 2: $T(\frac{1}{2}s_x, \frac{1}{2}s_y, 0)$
$$V = T(s_x, s_y, 0)V \tag{23}$$

1.4.7 Using Matrices

Matrices are useful for representing geometric transformations. However, we have to take care to use them properly and observe certain rules.

- Matrix order is important. In general, matrix operations are not commutative.
- Matrix orientation is important. The matrices defined above assume that the data to be viewed is in column vector form.
- When working with data that is in row-major form (each row is a data point), the data must be transposed prior to executing matrix operations.

When visualizing data using an interactive program we are continually making incremental changes to the viewing conditions. If we use incremental transformations on the data set to update the view, small errors will creep into the data set and continue to grow with each incremental transformation. After a sufficient number of view changes, the error will grow to the point where we are no longer looking at real data.

Therefore, it's important to keep the data around in the original data space. To view data, we first build our view matrix, then transform a copy of the data to produce view coordinates, then plot the transformed data. When the view changes we repeat the process, always going back to the original data set. Under this approach, user actions map directly to changes in the view parameters. The data, on the other hand, remains constant.

An alternative method of thinking about viewing data is to think about transformations modifying the data. Mathematically the concept results in an identical set of matrices. It's the difference between an object sitting on a table and walking around it versus having the object in your hand and rotating it to see all sides.

For some data sets, thinking about the problem as one of manipulating the data is easier. However, thinking of the process as one of changing the view makes it easier to view complex data sets and generate arbitrary viewing paths. For example, implementing a fly-through is much easier when the system is set up as the view reference coordinates moving through the data space.

1.4.8 NumPy, SciPy, and matplotlib

The NumPy, SciPy, and Matplotlib Python packages form a general purpose matrix, mathematical function, and plotting toolbox that provide much of the same functionality as Matlab. NumPy, in particular, provides all of the functionality we need to implement a matrix-based viewing pipeline. The key capabilities we need for viewing are matrix creation and matrix multiplication. Later on we will be using other capabilities for data manipulation, modeling, and data compression.

The examples below assume the NumPy package is imported as `import numpy as np`.

Function	Example	Description
matrix(L)	`M = np.matrix([[1, 0], [0, 1]])`	Creates a matrix out of the given list (of lists)
identity(N)	`I = np.matrix(np.identity(3))`	Creates an identity matrix of the given size
A * B	`A = M * I`	Matrix multiplication
inv(M)	`AI = np.linalg.inv( M )`	returns the inverse of the matrix M
M.I	`AI = M.I`	returns the inverse of the matrix M
M.T	`At = M.T`	returns the transpose of the matrix M

The matplotlib package is a visualization package built on top of NumPy. You import the package using `import pylab`.

Matplotlib has its roots as the visualization half of an interactive shell, similar to Matlab. In its simplest usage, matplotlib handles its own window creation and GUI widgets. You can also embed it into your own UI, including Tk, giving your program control over events.

1.4.9 Connecting User Input and Viewing

Interactive data viewing requires us to connect user input to changes in the view system parameters. User actions generate either specific changes in the view parameters or incremental modifications. The forms of interactive view modification include panning, scaling, and rotating.

- Panning: should move the VRP around the view plane, keeping the orientation of VUP and U fixed.
- Scaling: should increase or decrease the extent of the view volume in the view plane, keeping other parameters fixed.
- Rotation:
  - In 2D, should rotate the VUP vector about the z-axis, modifying the VUP vector and the derived U vector, but leaving their Z values at 0.
  - In 3D, the VRP may move in a circle around a fixation point (e.g. center of the current view volume). The same rotation can be applied to the VPN, VUP, and U vectors. An alternative interface method is to rotate the view direction according to mouse motions, keeping the VRP constant during the rotation.

The mouse gives the user the ability to move within 2 dimensions. Therefore, whether we are translating, rotating, or scaling, we have to have a method for translating user actions into modifications of the view reference coordinates.

- Translation of the mouse: translation of the VRP in the view plane
- Translation of the mouse: rotation of the VUP vector
- Translation of the mouse: scaling of the extent of the view window

A typical mouse has three buttons (left, right, center) which translate into three separate button clicks. Many mice also have a wheel or trackball on them, which provides an additional input device. The keyboard and menu items provide additional inputs that we can use as modifiers to the mouse action.

The key is how we connect the user interface elements to the parameters of the view transformation. Some key design questions include the following.

- Which mouse actions, buttons, or menu states connect to which view parameters?
- How does each action or button press relate to changes in one or more view parameters?
- What is the control law for the relationship?
- What are the control parameters of the relationship?
- Why is this a useful relationship?

Note that the first four questions are up to the programmer. You can make the relationships be whatever you want. The last question is what should guide the design process, and is often the motivation for user studies that look at the effects of different design decisions.

1.4.10 Interfaces as Control Laws

Any time we are connecting an input to an action, we have to write a control law. Consider, for example, connecting the motion of a mouse to panning of the data. The mouse moves in screen coordinates. Therefore, the input to the system is in pixels. Pixels, however, are not a meaningful unit in the data space. We can write the relationship between mouse motion and motion in the data space as follows, where ΔU is the motion in the data space, Δx is motion in screen space, and k_p is a proportional constant relating the two motions.

$$\Delta U = k_p \Delta x \tag{24}$$

We can control the relationship between the two spaces by varying the value of k_p. If we set $k_p = \frac{E_x}{S_x}$, which is the ratio of the data space extent to the screen space, then the data will appear to track the mouse as it moves. However, we could make k_p smaller or larger than that ratio and get different effects. If k_p is smaller, the data will lag the mouse motion. If k_p is bigger, the data will precede the mouse motion. The former is useful for fine tuning of a visualization, the latter is useful for moving quickly between different parts of the data.

We are not limited to a simple proportional relationship, however. Consider, for example, the use of an inertia term.

$$\Delta U_t = k_p \Delta x + k_n \Delta U_{t-1} \tag{25}$$

The inertia term means that the motion of the data is dependent not only on the current user input, but also the prior user input. A more complex form of the motion results if we make k_n dependent upon whether the user is actively controlling the device. Under active control, we can make $k_n = 0$, changing it to something like $k_n = 0.9$ when the user stops actively controlling the mouse. The iPod, and other multi-touch devices make use of this type of relationship to enable quick flipping and scrolling through lists.

Control theory also gives us an intuitive way of understanding the results of different design choices.

1. overdamped
2. critically damped
3. underdamped
4. unstable

Panning within the view plane

- Each change in the mouse position corresponds to a change in the position in the data space
- Horizontal motion in the view plane should move along the U axis
- Vertical motion in the view plane should move along the VUP axis
- The view volume extent and the screen size tell us how to scale pixel motion to data space motion

Process

1. Calculate how much the mouse moved $\Delta h, \Delta v$.
2. Scale the motion into data space by dividing by the screen size and multiplying by the extent.

$$(\Delta u, \Delta v) = (\Delta h \frac{E_x}{S_x}, \Delta v \frac{E_y}{S_y}) \tag{26}$$

3. Multiply the horizontal screen motion by the U axis and the vertical screen motion by the V axis to get the motion of the VRP in data space.

$$\begin{aligned} \Delta\text{VRP}_x &= \Delta u U_x + \Delta v \text{VUP}_x \\ \Delta\text{VRP}_y &= \Delta u U_y + \Delta v \text{VUP}_y \\ \Delta\text{VRP}_z &= \Delta u U_z + \Delta v \text{VUP}_z \end{aligned} \tag{27}$$

4. Add the motions onto the VRP.

$$\text{VRP} = (\text{VRP}_x + \Delta\text{VRP}_x, \text{VRP}_y + \Delta\text{VRP}_y, \text{VRP}_z + \Delta\text{VRP}_z) \tag{28}$$

5. Recalculate the view matrix
6. Calculate the new view locations of the data
7. Adjust the coordinates of the visual objects

Rotating the view up vector

1. Rotation for 2D data reorients the view volume, but the view window stays in the x-y plane.
2. Rotation should not change the location of the VRP or the view direction (VPN)
3. The visual point of rotation is generally the middle of the viewing screen

Process

1. Store the initial click and calculate the angle relative to horizontal. in python, the function

   `math.atan2( y, x)`

   calculates the angle relative to the x-axis defined by the vector (x, y).
2. Store the initial VUP vector: VUP_0
3. For each mouse motion
   (a) Calculate the current mouse angle relative to horizontal using `atan2`.
   (b) Subtract the angles to get the amount of rotation α

(c) Generate a rotation matrix for the angle -α

$$R(-\alpha) = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 & 0 \\ -\sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{29}$$

(d) Rotate the initial VUP vector and update the view object

$$\text{VUP}_\alpha = R(-\alpha)\text{VUP}_0 \tag{30}$$

(e) Recalculate the view matrix

(f) Calculate the new view locations of the data

(g) Adjust the coordinates of the visual objects

Scaling the extent

- The extent should scale uniformly in all directions
- The user should be able to scale up and down using the same motion
- An easy solution is to translate vertical motion into scaling

Process

1. Store the initial mouse click as a reference point P_0
2. Store the initial view extent E_0
3. For each mouse motion
   (a) Calculate the vertical distance between the initial click and the current mouse location.

$$\Delta v = P_{iy} - P_{0y} \tag{31}$$

   (b) Generate a multiplication factor from the difference. Set the value of k to control the speed of the scaling.

$$f = 1.0 + k\Delta v \tag{32}$$

   (c) Bound the factor on the zoom side to a small number larger than zero (e.g. 0.05)

   (d) Multiply the initial view extent by the factor and update the view extent

$$E_i = fE_0 \tag{33}$$

   (e) Recalculate the view matrix

   (f) Calculate the new view locations of the data

   (g) Adjust the coordinates of the visual objects

1.5 3D Camera Control

Translation: using typical keyboard controls (e.g. wasd for forward/left/back/right) works well. Number pad controls also work well.

- Forward/backward moves the VRP along the VPN by some step size γ.

$$\text{VRP}_{t+1} = \text{VRP}_t + \gamma\text{VPN} \tag{34}$$

- Left/right moves the VRP along the U axis.

$$\text{VRP}_{t+1} = \text{VRP}_t + \gamma\text{U} \tag{35}$$

- Up/down moves the VRP along the VUP axis.

$$\text{VRP}_{t+1} = \text{VRP}_t + \gamma\text{VUP} \tag{36}$$

Rotation (fly-through): rotate around VUP, anchored at the VRP, for right-left motion of the mouse and rotate around the U-vector, anchored at the VRP, for up-down motion. The rotations apply only to the orientation of the view coordinate system, which is defined as a set of vectors and is translation invariant.

- Left-right motion: translate VRP to the origin, align the axes, rotate by θ_h about the Y axis, invert the alignment, then translate back. To undo a rotation, we can multiply by the inverse matrix. The inverse of a rotation matrix happens to be its transpose, which makes inverting rotations an easy thing to do.

$$X_h(\theta_h) = T(\text{VRP})R_{xyz}(U, \text{VUP}, \text{VPN})^t R_y(\theta_h) R_{xyz}(U, \text{VUP}, \text{VPN}) T(-\text{VRP}) \tag{37}$$

- Up-down motion: align axes, rotate by θ_v about the X axis, invert the alignment

$$X_v(\theta_v) = T(\text{VRP})R_{xyz}(U, \text{VUP}, \text{VPN})^t R_x(\theta_v) R_{xyz}(U, \text{VUP}, \text{VPN}) T(-\text{VRP}) \tag{38}$$

Use $X_u(\theta_u)$ and $X_v(\theta_v)$ to transform the VRP point (homogeneous coordinate of 1) and the U, VUP, and VPN vectors (homogeneous coordinate of 0). Then rebuild the view matrix to update the data. The effect is that of turning your head around while the world stays fixed. Note that if both θ_u and θv are non-zero, you can do a single transformation process with the two rotation matrices in the center of the expression.

Rotation (Circle around data): rotate the view plane around the center of the view volume.

- Left-right motion: translate the center of view volume extent to the origin, align the VRC axes, rotate around VUP, unalign, untranslate

$$X_h(\theta_h) = T(\text{VRP} + \frac{E_z}{2}\text{VPN}) R^t_{xyz} R_y(\theta_h) R_{xyz} T(-\text{VRP} - \frac{E_z}{2}\text{VPN}) \tag{39}$$

- up-down motion: translate the center of view volume extent to the origin, align the VRC axes, rotate around U, unalign, untranslate

$$X_v(\theta_v) = T(\text{VRP} + \frac{E_z}{2}\text{VPN}) R^t_{xyz} R_x(\theta_v) R_{xyz} T(-\text{VRP} - \frac{E_z}{2}\text{VPN}) \tag{40}$$

Use $X(\theta)$ to transform the VRP point (homogeneous coordinate of 1) and the U, VUP, and VPN vectors (homogeneous coordinate of 0). Then rebuild the view matrix to update the data. The effect is that of the data rotating in space while the user stays fixed.

Scaling: no change compared to 2D scaling: scale the view extent equally in all directions.

1.6 Displaying Higher Dimensions

We can easily display two dimensional data on a plane. With a little more work, we can use the same viewing transformation process to project 3D data onto a plane, enabling us to view it. If we add a time component to the data, we can view four dimensions of data over time. However, an arbitrary fourth dimension is more difficult to represent using a time sequence.

Four, or even five dimensional data, however, is not uncommon. For example, consider a set of water column data taken over a grid. Each data point represents a 2D position plus a depth. A natural way to visualize the locations of the data points is a 3-dimensional grid. However, we would also like to view the value of the measurement taken at each grid point, which requires 4 dimensions. Therefore, we need some method of indicating a value at each 3D point.

What are some visual characteristics we could use to represent the value of a number?

- Color: we could vary the color to represent the measurement value
- Size: we could modify the size of the data point
- Texture: we could modify the visual texture of the data point
- Icons: for discrete data, we can use different icons to represent different values

In addition, we could use non-visual cues to represent information.

- Haptic feedback: we could provide tactile feedback, such as vibration, to represent the value
- Aural feedback: we could use sound to represent the measurement value

Combining multiple cues can permit us to display even higher dimensional data. For example, we could use color and size, or color and icons to represent independent dimensions.

1.6.1 Multi-plots

Sometimes a category is one of the dimensions we want to display. Color or texture are common ways of displaying data from different categories whether using bar charts, scatter plots, or line plots. For example, we could plot the population of several cities over ten years using a line chart with a different color for each city. This is a very common method of plotting three dimensions (category, time, and value).

As the number of categories grows, however, it becomes more difficult to see an individual line. It also becomes difficult to select easily differentiable colors beyond about 7-10. A viewer can no longer quickly pick out the trend for an individual category.

The key to appropriate visualization is identifying the key attribute of the information. In the case of values over time, seeing trends is often the most important attribute. Therefore, being able to see a single line over time, or perhaps two overlaid for comparison is the critical visualization.

One approach to displaying trends over time for multiple categories is to use multiple small plots. A multi-plot visualization is simply a collection of line plots laid out vertically, horizontally, or as a matrix. If the trends over time are the most important attribute, a vertical arrangement allows for direct comparisons. If the magnitudes over time are most important, a horizontal arrangement works well. For a large number of categories, a matrix arrangement is most space effect. Because the trends are the most relevant attribute, the individual plots can be small, with low resolution in time and magnitude.

1.6.2 Colors

RGB: additive color scheme, used internally by computers to represent colors. While, as a programmer, we have to ultimately write pixel colors using RGB, the space is not always appropriate for color selection. In particular, distances in RGB space do not correspond to perceptual distances, as figure 2 demonstrates.

Figure 2: Color distances. The color in the first column is equally different in RGB space from the colors in the second and third column

There are two useful magnitude axes in RGB space. People with normal color vision perceive red and green as opposite colors and yellow and blue as opposite colors. We can use this to display magnitudes on a red-green color axis or yellow-blue color axis.

$$C_{RG} = (\alpha, 0.0, 1 - \alpha) \tag{41}$$

$$C_{BY} = (1 - \alpha, 1 - \alpha, \alpha) \tag{42}$$

HSV: hue channel is the most commonly used color spectrum

- Representation is a cylinder
- Hue is orientation around the circle
- Saturation is distance from the central axis
- Value/intensity is distance along the central axis

CIE-LUV: distances are more perceptually meaningful, two chromaticity channels

- CIE-LUV is a nonlinear transformation of RGB space

Intensity: intensity and color are not completely independent, although they can be used to represent independent dimensions with low precision. When using intensity and color simultaneously, it may be important to adjust the intensity based on the color. Some colors appear inherently brighter than others at the same intensity. Normal human eyes, for example, are more sensitive to green and red than blue.

- Intensity is usually the average value of the color channels $I = (R + G + B)/3$.
- Intensity may also be a weighted average of the color channels $Y = 0.299R + 0.587G + 0.114B$

1.7 Range Selection

How do we automatically select ranges for display? We have to pick ranges for many aspects of a visualization, including spatial extent, colors, sizes, or other methods of representing data values. Part of the purpose of interactive visualization is to avoid having to automatically select ranges, but even if the user has control over the spatial view, we may still have to select ranges for the color and size of the plotted points.

Visualization has several competing goals. A single visualization or series of visualizations has to find an appropriate balance.

- Correct characterization of the data
- Discriminability of relevant characteristics
- Stability and comparability across data sets

If we choose a simple property of the data such as max and min to set the range, then plots of similar kinds of data can look extremely different.

- Outliers in the data control the range of colors, locations, sizes, etc.
- Data with similar values in different data sets may end up with different visualization properties
- Discriminability in the center of the data range may be impossible

We would like to use invariant properties of the data space to select the range for visualization.

- Pick a range based on the mean and standard deviation of the data.

  Data with a Gaussian distribution will be distributed so that most of the data (about 95%) is within 2 standard deviations of the mean. For such data sets, a range such as ± 2 standard deviations will generally produce a consistent, useful visualization with reasonable discriminability.

  Using standard deviation to set the range does not work as well on data that does not follow a Gaussian distribution or has a significant number of outliers. Outliers cause the standard deviation to expand, compressing the middle of the plot and reducing discriminability.

- Pick a range based on the mean and mean absolute deviation (or average absolute deviation).

$$\text{MAD} = \frac{1}{N} \sum_{i=0}^{N-1} |x_i - \mu| \tag{43}$$

  The mean absolute deviation is similar to the standard deviation, but it is less sensitive to outliers. For a Gaussian distribution, the MAD will be approximately 0.8 times the standard deviation.

- Pick the range based on the potential data values.

  For some data sets, the range of possible values is known. Selecting the range base on the set of possible values then enables consistent visualizations across different data sets with the same variables. If the actual data values cluster in one section of the range, this method of range selection will compress them, reducing discriminability of individual data points. However, if the intent is to compare the behavior of data sets and not individual points, this method enables consistent, comparable visualizations.

- Pick the range based on potential values, but use bookend categories that capture outliers.

  For example, pick a color range such as red to green. Let pure red represent any data value larger than an upper bound, and pure green any data value smaller than a lower bound. Pick the range between the upper and lower bound so that it covers most of the data.

  We can achieve the same effect by using a sigmoid function to map data values to visualization values, such as colors. A sigmoid is a squashing function that takes the number line and squashes it to the range [0, 1], maintaining the monotonicity of the original number line. There are a number of functions used as sigmoids, but the most common is the logistics function. All sigmoid functions are s-shaped, mapping large negative numbers to values close to zero and large positive numbers to values close to one.

$$S(x) = \frac{1}{1 + e^{-B(x-x_0)}} \tag{44}$$

  The parameter x_0 corresponds to the central point of the sigmoid curve. The parameter B controls the slope of the central part of the curve. Large B values produce steeper slopes, forcing the curve to be closer to a step function. Small values of B flatten the central slope, slowing the transition of the range from values close to 0 to values close to 1. The sigmoid is a generally useful function in data analysis, and we'll explore it more fully later in the course.

- Pick the range based on desired visualization outcomes.

  For example, if we're plotting temperature for a weather report, any temperature above 95F should appear as hot. Likewise, any temperature below 0F should appear as cold. These are subjective impressions, and in many situations it is not necessary to discriminate further.

- Logarithmic scales to capture measurements with high dynamic range.

  Data with high dynamic range, such as sound amplitudes, challenge linear visualizations. As the amplitude of a lawnmower may be several orders of magnitude higher than the sound of a person talking, trying to plot both on the same linear chart compresses most typical sounds into a very small range. If we were to add the amplitude of a jet engine, the lawnmower would also end up compressed.

  A logarithmic scale enables us to discriminate between different quiet sounds and different loud sounds, but it makes distances less meaningful. Equal distance in log space means equal ratio transitions. Therefore, the change from 1 to 2 is spatially the same as the change from 10 to 20 or 1000 to 2000.

The above concerns apply equally well to color ranges or spatial ranges in visualization.

In good plotting programs, the user can control all aspects of the visualization. There is no universal method of picking a range that will work for all visualizations. There are methods that work reasonably well over a wide range of situations. If you know your data set, you can probably develop an algorithm that will work effectively for the user's purposes. Nevertheless, it is important to build flexibility into a visualization system and enable the user to tweak or adjust most range selection parameters.

1.8 Pre-scaling

Interactive viewing, particularly when it involves rotation, requires that the data space axes all have the same scale. Unit distance along one axis needs to be the same as unit distance along any other axis. The mathematical difficulty of using different scales arises when we try to define the extent of an oriented bounding box as it rotates in the data space. A bounding box of unit length oriented parallel to one axis will shrink or expand when it rotates to orient with a different axis with a different scale.

Many data spaces, however, have different natural scales that are not necessarily related. A data set of height and weight contains two different natural scales (e.g. cm and kg, or inches and pounds). If we have a method of selecting an appropriate range for each axis, then we can pre-scale each axis to the range [0, 1] prior to implementing the view volume.

Note that natural scale is different than range. A data set that contains the height of children might have a natural scale in inches or centimeters. A different data set that contains the height of adults will have the same natural scale, but a different range. We don't want to use different scales when plotting these two data sets together. If we want to see their comparative values, we would also want to use the same range along each axis. Decisions about the natural scale and range are complex and data specific.

We can think of pre-scaling as a step that occurs prior to the viewing transform.

$$\text{Data}_{\text{norm}} = S(s_a, s_b, s_c)\text{Data}_{\text{raw}} \tag{45}$$

Range selection is effected by picking the extent and location of the view volume in the normalized data space. The eye location and direction determines the center of the visualization, and the extent of the bounding box determines the viewing range.

1.9 Scatter Plots

Scatter plots allow the user to view the data in the native data space. So far, we have focused on generating interactive scatter plots that show the data in some representation of the data space.

Axes and titles are important in scatter plots, as are legends that indicate the meaning of the visual representations

- How do we represent arbitrary axes? One method is to keep the axes 2D and constant, Showing the location of the corners in data space. A second method is to place axes in the data space and manipulate them (or at least their endpoints) using the same viewing pipeline we use for the data points.
- What needs to be in a legend? Need an example of each visual representation and a key to its meaning. Legends should be fixed on the screen.
- What is the viewing window? The viewing window is often smaller than the actual window. For example, we don't want the points to move over, or behind the legend or the title. You can build this into the viewing pipeline. Clipping is possible in both normalized screen coordinates or actual screen coordinates.

1.10 Histograms

Viewing the raw data in its native space as a scatter plot is one method of visualization of a data set, and can be very useful as a method of exploration. However, there are many other ways to think about representing characteristics of data.

- Histograms are representations of distributions
- For discrete data, how many elements of each discrete element exist in the data set?
- For continuous data, what is the distribution of data values?

Histograms can be N-dimensional, although beyond 2 or 3 they get unwieldy to represent. 2-D histograms that use color or a 3D representation are common.

For continuous data, we often break the space into bins in order to represent the distribution, counting how many elements fall into each bin. The number of bins to use for a particular data set depends on a number of factors. The issues that arise in bin selection are very similar to those that arise in selecting ranges for visualizing colors, sizes, or locations. If we have bins covering the entire range of possible values, we may lose discriminability in the central portion of the distribution. If the bins are too big or too small, we lose the ability to see coherent patterns in the data. Using bookend bins is a common choice, but can result in strange looking plots where the frequency bumps up on the outermost bins since they cover more of the data space. The following questions can help us to tailor a histogram to the needs of the user.

- What is the frequency of real change in the distribution? $\rightarrow$ Bin size should show real changes.
- What is the precision of the measurement? $\rightarrow$ Bin size should not be finer than precision.
- What is the purpose of the histogram? $\rightarrow$ Bin size and range should support the goals.

If we decide to bin data, we may want to make use of the fact that data has noise characteristics and distribute each data point accordingly. Otherwise, regular binning may lead to **aliasing**.

- Rounding to the nearest bin center is not necessarily correct
- If a measurement has noise that is on the order of the bin size we should distribute the value of the data point between neighboring bins

For large numbers of measurements, discrete histograms become more accurate. For small numbers, we usually want to distribute weight according to the measurement noise. An alternative is to run a smoothing filter over the histogram after it's built.

One method of distributing a count across multiple bins is to use a Gaussian model for the noise in the measurement. The integral under the Gaussian that lies within the central bin is the weight added to that bin. The same calculation provides the weight for the bin's neighbors. Usually, the process is limited to a specific number of bins (2 or 3), or to those within some threshold, such as 2σ. Unfortunately, calculating the integral under a Gaussian can be an expensive computation (it does not have a closed form solution).

One approximation that is often used, is to distribute the weight of a data point between the two nearest bins. This approach is appropriate when the size of the bins is approximately 2σ, where σ is the standard deviation of the noise in the measurement.

Distributing weights across bins is more important for high resolution histograms where the bin size is close to the measurement noise. However, even low-resolution histograms will exhibit aliasing if the count weights are discrete.

1.10.1 Visualizing Histograms

1D distributions

- Vertical axis (or horizontal axis) is frequency, other axis is category
- Frequency may be a count or a percentage (probability)
- Scatter plot
- Bar chart
- Curve
- Filled curve

2D distributions

- 2D grid of categories with vertical bars indicating frequency
- 2D grid with color/size indicating frequency
- 3D scatter plot

3D distributions (e.g. RGB values in an image)

- 3D scatter plot with color/size indicating frequency

1.10.2 Histograms and Probability Distribution Functions

Histograms are a method of estimating probability distribution functions from data.

A probability distribution function specifies the probability of a random variable X taking on a particular value. The most commonly used PDF is the Gaussian distribution, or normal distribution. A PDF for a single random variable can be any function with an integral of 1.

$$\int_{-\infty}^{\infty} PDF(X) = 1 \tag{46}$$

Therefore, we can convert a histogram to a PDF by dividing each bin value by the sum of the bins. PDFs are useful for sampling problems, modeling problems, perception tasks, and learning problems. The characteristics of the PDF are also useful for data analysis and characterization.

1.11 Basic Data Analysis

1.11.1 Mean, Median, Mode

Basic concepts are pretty straightforward.

- Mean: sum of the data values divided by the number of data values

$$\mu = \frac{1}{N} \sum_{i=0}^{N-1} x_i \tag{47}$$

- Median: data value with equal numbers of values above and below it

$$\mu_{\text{median}} = \text{sort}(\{x_0, \ldots, x_{N-1}\})[N/2] \tag{48}$$

- Mode: data value with the highest frequency (usually applied to categorical or binned data)

$$\mu_{\text{mode}} = \max(\text{histogram}(\{x_0, \ldots, x_{N-1}\})) \tag{49}$$

When they fail to be meaningful is a different issue.

- Bi-modal or multi-modal distributions cause problems
- Missing data values can cause problems

1.11.2 Data Variation: Variances and Distributions

Variance is a measure of the distribution of a data set in one dimension. It has a special meaning when the data is well represented by a Guassian distribution. The standard deviation of the best fit distribution will be the square root of the variance of the data, and there are certain rules of thumb the data will follow. For non-Gaussian distributions, it is still a measure of variation, but does not necessarily have the same meaning in terms of how the data is distributed.

A histogram is also a representation of data variance. A normalized histogram–where the sum of the frequencies sum to one–is an expression of probabilities that a data value will fall in particular categories. If we can express the probability of certain data values as a function, this is called the probability distribution function [PDF] of the data.

The entropy of a histogram is also a measure of spread. A low-entropy distribution is concentrated in a few values, while a high entropy distribution is distributed evenly across values. Given a histogram with N bins, the entropy of the histogram is given by (50), where p_i is the probability of bin i.

$$E = \sum_{i=1}^{N} p_i log_2(p_i) \tag{50}$$

The integral of the probability distribution function, also called the cumulative distribution function [CDF], is an alternative method of visualizing a distribution, and one that is applicable to continuous data without binning. The CDF indicates the probability that a value in the data set is below a certain value.

A CDF can be used as a lookup table to emulate drawing data from a particular distribution. First, pick a number uniformly and randomly in the range [0, 1]. Second, find the position on the CDF graph that corresponds to that value on the y-axis. The x-value of the point on the CDF graph corresponds to a value in the data range. Areas of the CDF with steep slopes correspond to highly probable values of the data.

1.11.3 Gaussian Distributions

The Gaussian distribution is the most commonly used probability distribution function for analyzing data.

$$G(x;\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{51}$$

- μ: mean of the distribution
- σ: standard deviation of the distribution

The Gaussian distribution is heavily used in part because of the central limit theorem, which states that the sum of many independent identically distributed random variables will tend to be distributed according to the normal distribution. In other words, the output of any process that is the result many smaller processes, each contributing a similar type noise to the system, will tend to look like a normal distribution.

Fitting a Gaussian function to data can be tricky, as it is a nonlinear function. Simply calculating the mean and standard deviation of the data provides an approximation, but it does not provide the amplitude of the curve in data space. Because the Gaussian is a non-linear function, a common method of finding a good fit is the Levenberg-Marquardt algorithm.

1.11.4 Multi-dimensional Gaussian Distributions

Variables in a data set may be related to one another, or they may be independent. The covariance matrix of a data set Σ gives us an idea of the first order relationships between different dimensions. The covariance matrix is defined by (52).

$$\Sigma(i,j) = \sum_{k=0}^{N} \frac{(x_{i,k} - \mu_i)(x_{j,k} - \mu_j)}{N-1} \tag{52}$$

The diagonal entries of the covariance matrix are the variances of each dimension. The off-diagonals show the relationships between different dimensions. If the off-diagonals are close to zero, then the two dimensions are largely independent. If the off-diagonals are large, then the two dimensions are strongly related. A covariance matrix is symmetric, as the covariance of dimensions i and j is the same as the covariance of dimensions j and i.

We can use the covariance matrix to define multi-dimensional Gaussian distributions.

$$G(\vec{x};\vec{\mu},\Sigma) = \frac{1}{(2\pi)^{N/2}|\Sigma|^{1/2}} e^{-\frac{1}{2}(\vec{x}-\vec{\mu})^t\Sigma^{-1}(\vec{x}-\vec{u})} \tag{53}$$

A reasonable way to think about a multi-dimensional Gaussian in 2D is as an oriented ellipse. In 3D we can visualize it as a an oriented ellipsoid. The long side of the distribution represents the primary axis of variation within the data. The relative size of the axes tells us about the number of important dimensions

of variation within the data. These properties are very important when we start trying to visualize and understand complex, multi-dimensional data sets.

Covariance matrices fall into the category of symmetric, positive-definite matrices. Therefore, we can invert them using a fast algorithm called Cholesky decomposition. That's important if we're using distance metrics to measure the difference between two data points that take into account the covariances.

The covariance matrix is critical for many tasks in data analysis and visualization. In addition to providing information about each dimension of the data individually, it tells us how the various dimensions are related, if at all. If strong relationships exist between two or more dimensions of a data set, then the number of independent dimensions of the data is likely less than the number of features or measurements.

Overall, covariance matrices have four primary uses.

- Understanding: The primary axes of variation give us observational information about our data.
- Compression: We can often usefully represent the data using fewer dimensions than the base data set by finding the primary axes of variation and projecting the data onto those axes.
- Visualization: If we project the data onto the primary axes of variation, then we can display the data along those axes instead of the original data axes. This is one approach to visualizing very high dimensional data that contains redundant information.
- Comparison: distance metrics tell us how similar two points in the data space are. If we use a naïve measurement of distance, such as Euclidean distance, it does not take into account relationships between the various dimensions. If we use a distance metric based on the covariance matrix, we get a better estimate of the true similarity of two points in the data space.

The primary axes of variation of a data set are provided by the eigenvectors of the covariance matrix.

1.12 Data Transformations

We often transform data to make it easier to understand or visualize.

- Combinations of basic measurements to obtain derived values

  Example: an approximation to ocean energy is the product of wave height, tidal variation, and wind speed.

  Exploring combinations of features–ratios, sums, products–is often useful in data analysis, especially when combined with procedures such as linear regression that describe relationships between a dependent variable and one or more independent variables. Computers can quickly, and automatically explore these relationships and identify potential relationships.

- Combination of basic measurements to obtain dimensionless representative numbers

  Reynold's Number: Reynold's number is a dimensionless number that is the ratio of inertial forces to viscous forces in a fluid. It effectively characterizes the behavior of an object moving in the fluid, or fluid moving through a channel, regardless of the actual physical scale.

  Useful dimensionless representative values are often known for engineering problems. Novel computed values–including simple ratios–may provide useful comparative information for analysis or visualization.

- Transformations of individual variables to obtain distributions with well-behaved properties

  Example: logarithmic transformation of a variable to convert a power law distribution into a normal distribution

  Logarithmic transformations can be challenging when used for numbers that are both larger and smaller than one. It is not an appropriate transformation for any feature that has possible real values equal to or less than zero.

- Calculating integrals or gradients of measurements over time or space

  Example: calculating slope from elevation data (gradient) or calculating total energy usage by integrating usage rate information

- Calculating differentials or products relative to a baseline measurement to reduce the effect of noise

  Example: taking an image with the shutter closed to obtain the dark image, then subtracting the dark image from all subsequent normal images.

  Some data sets provide a baseline measurement, or the results may already be the result of comparison to a baseline.

1.12.1 Logarithms, exponentials, and sigmoids

Logarithms are commonly used to transform variables when the ratios of variables are the critical relationship. A logarithmic transformation converts a power law distribution into a linear relationship, for example. A power law distribution occurs when the frequency of an event halves as the strength of the event doubles. Very strong earthquakes, for example, tend to be half as frequent as earthquakes half as strong.

In general, a power law has the form of (54), where $f(x)$ is the frequency of an event of magnitude x, a and k are scaling factors, and $o(x^k)$ is an asymptotically small function of x^k. Taking the logarithm of (54) results in a linear equation with slope k.

$$f(x) = ax^k + o(x^k) \tag{54}$$

$$\log(f(x)) = k\log(x) + \log(a) \tag{55}$$

The flip side of a logarithm is an exponential. Unless the measurement value is already a logarithmic value, we rarely use exponentials to transform data as they spread out the range of the data and create long tails on distributions. However, it is not uncommon to use exponentiation in the context of other types of transformations. For example, a common tactic for converting the output of a process into a likelihood is to use a sigmoid function.

$$P(t) = \frac{1}{1 + e^{-a(x-x_0)}} \tag{56}$$

The sigmoid converts any bump-like distribution into an S-shaped curve. It can be useful for converting values that saturate at large negative or positive values into the range [0, 1]. The slope of the central line is controlled by a, and the placement of the sigmoid is controlled by x_0.

1.12.2 Normalizing Data

Before viewing or manipulating data, we often need to normalize the values of each variable in a data set so that they are comparable. There are a number of methods of normalizing data, and each has its strengths and weaknesses.

- One method is to normalize each dimension based on its min and max values.

$$\hat{x}_i = \frac{x_i - \min}{\max - \min} \tag{57}$$

  The strength of this method is that it is easy to implement and all of the data is contained within the range [0, 1]. A potential problem with using min and max values is that outliers in the data set can compress the middle section of the range, making it difficult to discriminate values.

- A second method, which works best if the data is approximately Gaussian in its distribution, is to subtract off the mean and divide by the standard deviation.

$$z_i = \hat{x}_i = \frac{x_i - \mu}{\sigma} \tag{58}$$

  The transformation in (58) creates what is called a z-value in statistics. It represents how far away from the mean a particular value lies in units of the standard deviation of the distribution. The z-value, therefore, is invariant to the particular mean and standard deviation of the variable. It represents a statistically meaningful differential and indicates where in the distribution the particular value lies.

  When visualizing data normalized by the mean and standard deviation, you usually pick a cutoff, such as $\pm 2\sigma$ as the extent of the view volume. However, this means that not all of the data is initially visible.

- A third commonly used method is to transform the data using a sigmoid function, described above, which converts the real number line into the range (0, 1). It is important to select parameters (α and μ) for the sigmoid function such that most of the data lives on the linear section in the middle of the distribution, saturating only towards the edges.

$$\hat{x}_i = \frac{1}{1 + e^{-\alpha(x_i - \mu)}} \tag{59}$$

  While the sigmoid function is often used to normalize data for machine learning algorithms, it is not as useful for visualization because it is nonlinear. The transformation does not preserve distances between data points, especially outliers.

1.12.3 Gradients

One of the most common operations in filtering, tracking, or control is to calculate the first derivative of a signal (rate of change). Differentiation is inherently a noise amplifying operation, especially for noise that is at a higher frequency than the signal.

Example: Derivative of $\cos(Kx) = -K\sin(Kx)$

When calculating deriatives numerically, you have the added problem of working in the digital domain with periodically spaced samples.

Its useful to look at the definition of a derivative.

$$\frac{dy}{dx} = \frac{\Delta y}{\Delta x} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x} \tag{60}$$

When working with a data stream or digitized function, we have to approximate this as:

$$\frac{\Delta y}{\Delta x} = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} \tag{61}$$

This is known as the forward difference function because it is based on the next sample.

If we want to approximate the true derivative, we need to make the step between x_{i+1} and x_i as small as possible. It turns out this is a really bad idea...

- You subtract two large numbers to get a much smaller number (lose information)
- You divide a large number by a small number (inherently unstable)
- The backward difference is no better, so increasing the sampling rate is not a total solution.

The **central difference** method uses both the forward and backward point to calculate the derivative at the central point.

$$\frac{\Delta y}{\Delta x} = \frac{f(x_{i+1}) - f(x_{i-1})}{x_{i+1} - x_{i-1}} \tag{62}$$

If you use a Taylor series to derive the derivative formula, one thing that falls out is that the error in the derivative is related to the square of the sampling period, which should be a small value.

Another way to think about the central difference formula is as a convolution of a smoothing function with the derivative function.

Consider, for example, the convolution of [1 1] and [-1 1]

- The result is [-1 0 1], which is the central difference theorem

It is possible to extend the concept to larger initial vectors like [1 2 1] and [-1 0 1]

- The result is [-1 -2 0 2 1], which is a smoother version of the central difference.
- Smoothing acts as a low-pass filter, reducing noise in the derivative signal

The derivative function you use for your application, which has to balance two factors:

- **Smoothness**: how noisy is the derivative signal relative to the underlying function
- **Responsiveness**: how quickly can the derivative change in response to the input

A more expensive, but potentially smoother and more accurate method of calculating derivatives is to interpolate the points with an order N polynomial and then use the derivative of the polynomial. You need $N+1$ points for an order N polynomial, which will let you represent derivatives of order $N-1$.

- If you know your function is locally order N or less this can provide an accurate derivative.
- The Taylor expansion methods are actually based on a polynomial fit to the points
- Polynomial fitting is often used on large numbers of data points to get smooth derivatives

1.12.4 Integrals

Integration is another popular thing to do with incoming data

- Measurements of velocity or distance based on acceleration required accurate integration
- Pedometers, for example, use inertial sensors that execute a double-integration
- Energy/water usage computations often require integration of flow measurements

The simple methods for integrating a data series are really simple. These all fall under the rubric of Newton-Cotes formulas, where we approximate the function $f(x)$ with an easily integrable function. The error is related to the quality of the approximation.

One essential requirement for discrete integration to be accurate is that the sampling process has to be done correctly. In particular, the sampling frequency has to be at least twice the highest frequency in the signal being measured. If that is not the case, then the function is not guaranteed to be well-behaved in the interval between two samples.

Rectangular rule: approximate the function as a series of rectangles and sum their area

$$I = \sum_N f(x_i)\Delta x \tag{63}$$

Trapezoidal rule: approximate the function as a series of trapezoids and sum their area

$$I = \sum_N \frac{f(x_i) + f(x_{i+1})}{2}\Delta x \tag{64}$$

Simpsons 1/3 rule: approximate the integral using a series of parabolas (2nd order curves)

$$I = \sum_N \left[f(x_{i-1}) + 4f(x_i) + f(x_{i+1})\right]\frac{1}{3}\Delta x \tag{65}$$

If you work through the Taylor series, you can calculate the error term for each of the methods in terms of the step size. The error being proportional to step size makes sense, as the integral becomes more accurate, approaching the true answer, as the step size decreases. Using more information to approximate the function results in a decrease in error for well behaved functions (we're assuming the signal is sampled sufficiently that it is not changing faster than the sampling process can capture).

- For the rectangle rule, the error is $O(\Delta x)$
- For the trapezoidal rule, the error is $O(\Delta x^2)$
- For both the 1/3 and 3/8 rule the error is $O(\Delta x^4)$

Reducing errors

In order to reduce integration errors you either need to decrease the step size or increase the order of the integration formula

- Decreasing step size at some point starts to increase round-off and numerical errors
- Increasing the order of the integation increases the computational load

It turns out you can use the results of lower order integrations to calculate more accurate results

Richardson extrapolation

If you take two estimates of the integral, but use different step sizes, then you can write an approximation to the error term in terms of the two results. For the trapezoidal rule, the expressions for the integral using the two step sizes are:

$$I \approx I_1(\Delta x_1) + c\Delta x_1^2 \tag{66}$$

$$I \approx I_2(\Delta x_2) + c\Delta x_2^2 \tag{67}$$

These just say that we have two estimates of the true integral I. One is based on time step Δx_1, and one is based on time step Δx_2. Subtracting these two equations gives an estimate for c, which is a multiplier for the error term, in terms of the integral results and the two step sizes.

$$c \approx \frac{I_2(\Delta x_2) - I_1(\Delta x_1)}{\Delta x_1^2 - \Delta x_2^2} \tag{68}$$

If we then solve for I by substituting back into (67), we get an improved estimate of I.

$$I \approx I_2(\Delta x_2) + \frac{I_2(\Delta x_2) - I_1(\Delta x_1)}{\left[\left(\frac{\Delta x_1}{\Delta x_2}\right)^2 - 1\right]} \tag{69}$$

Now if we simplify things by using $\Delta x_2 = \frac{1}{2}\Delta x_1$ then we actually get Simpsons 1/3 rule with step size Δx_2. This means that using a weighted sum of two trapezoidal rule integral results gives us the equivalent of a 2nd order polynomial result and an error of $O(\Delta x^4)$.

Romberg Integration

The end result is that you can define a recursive procedure that calculates increasingly more accurate integrals based on trapezoidal rule results, which are fast to compute.

The general formula for developing higher accuracy results is:

$$I_{k,n} = \frac{4^k I_{k-1,n} - I_{k-1,\frac{n}{2}}}{4^k - 1} \tag{70}$$

k is the order of the integration, and n is the number of samples integrated by the process. A simple trapezoidal rule (order 0) integration result using n steps would be represented as $I_{0,n}$.

Given two trapezoidal rule results $I_{0,n}$ and $I_{0,\frac{n}{2}}$ with step sizes of $\frac{\Delta x}{2}$ and Δx, respectively, you can combine them to get the Simpson's rule (1st order) result using the following.

$$I_{1,n} = \frac{4I_{0,n} - I_{0,\frac{n}{2}}}{4^1 - 1} \tag{71}$$

The formula can be used recursively to generate increasingly accurate estimations of the integral in a tree structure. Each step up the tree results in an accuracy increase on the order of Δx^2.

For example, to calculate a second order result for 16 samples with an error on the order of $O(\Delta x^6)$, represented as $I_{3,16}$, we would make the following calculations.

1. For the order zero level, we would calculate $I_{0,4}$, $I_{0,8}$ and $I_{0,16}$ using the trapezoidal rule. For $I_{0,4}$ we would use four evenly spaced samples out of the original 16 (e.g., 0, 5, 10, and 15). For $I_{0,8}$ we would use eight samples (e.g., 0, 2, 4, 6, 8, 10, 12, and 14), and for $I_{0,16}$ all 16 samples. These would be the only calculations using raw input values.
2. For the order one results, we would calculate $I_{1,8}$, and $I_{1,16}$ using the recursive formula given in (70).
3. For the final order two result, we would calculate $I_{2,16}$ using the recursive formula in (70).

The end result is a more accurate estimate of the true integral while only computing trapezoid rule integrals. Trapezoid rule integrals are easy to do on a microprocessor such as the PIC because they only involve divisions by two, which can be accomplished by a bit shift.

1.13 High-dimensional Data

Visualizing high-dimensional data is a challenging task, especially when the dimensions go well beyond what we can represent using every color, size, texture, arrow, or 3D visualization technique. The basic approach to visualizing high-dimensional data is to compress the data space into a set of dimensions we can view. We want to execute the compression so that as much of the original information is retained in the compressed data space. In particular, points that are significantly different in the original data space should be significantly different in the compressed data space. Likewise, points that are similar in the original data space should still be similar in the compressed space.

The degree of compression required may be small, such as six dimensions of environmental data compressed down to three. At the other extreme, images of faces containing close to 64,000 greyscale values can be compressed and visualized in 3 dimensions with reasonably useful results. The quality of the compression depends largely upon the independence of the original variables. The six environmental variables may represent six independent sources of information, in which case compression quality will be poor. The 64,000 greyscale values in a face image, however, are highly correlated, resulting in a reasonably high quality compression result that preserves the essential information.

The covariance matrix is the basis for one of the most commonly used algorithms for compression and visualization. The covariance matrix tells us which dimensions are correlated and which are independent, or uncorrelated.

1.13.1 Principal Components Analysis

Principal components analysis uses the covariance matrix Σ to identify the primary directions of variation in a set of data. The eigenvectors of the covariance matrix provide a basis space for the data that is tailored specifically to those variation directions. The eigenvalues tell us the relative importance of each of the eigenvectors.

An alternative method of computing similar information is to use SVD, or singular value decomposition to obtain the orthogonal basis vectors representing the primary modes of variation. SVD is the preferred method when the size of each data sample is large compared to the number of samples. For example, when calculating the basis vectors for a set of images of faces, each data sample is an image, potentially consisting of $> 64,000$ pixels, while the whole data set may consist of only 200 images.

When the each data point is a small number of features and there are lots of them, it is easier to calculate the covariance matrix and compute its eigenvalues and eigenvectors.

Calculating the principal components using the covariance matrix:

- Calculate the covariance matrix for the data set. If we are use the convention of a row for each data point, then we need to tell it that the variables are in the columns, not the rows.

  ```
  mcov = numpy.cov( m, rowvar=False )
  ```

- Calculate the eigenvalues and eigenvectors of the covariance matrix. The eig function in the linalg package provides this capability in numpy. The return value is two arrays. The first contains the eigenvalues. The second contains the eigenvectors as columns of the matrix (meigvec[:,i]).

  ```
  (meigval, meigvec) = numpy.linalg.eig( mcov )
  ```

- The eigenvalues tell you the relative importance of the eigenvectors. Looking at the ratio of the eigenvalues is an indication of their relative importance. A commonly used analysis is to look at the

cumulative sum of the eigenvalues from largest to smallest as a fraction of their total sum. If the first few eigenvalues represent 95% of the sum of all the eigenvalues, for example, then the corresponding eigenvectors account for almost all of the variation in the data set.

- The eigenvectors tell you the directions of primary variation within the data. They are orthonormal vectors, which means the dot product of any two eigenvectors is zero and they have unit length.

Calculating the principal components using SVD:

- Calculate the mean data vector $\vec{\mu}$.

```
mu = m.mean( axis = 0 )
```

- Subtract $\vec{\mu}$ from each data point to get a set of differential vectors $\vec{d}$.

```
mdiff = m.copy()
for i in range( mdiff.shape[0] ):
    mdiff[i] = mdiff[i] - mu
```

- Set up a matrix A where each differential vector $\vec{d}$ is a column of A.
- Calculate the SVD of of A, which generates the three matrices U, W, V^t

```
(u, w, vt) = numpy.linalg.svd( mdiff.transpose() )
```

- The columns of U are the basis vectors of the columns of A
- The rows of V^t are the basis vectors of the rows of A
- The values in W are the singular values, and are related to the eigenvalues

$$w_i = \sqrt{e_i} \tag{72}$$

Projecting data onto the principal components

- Select how many principal components to keep. As noted above, it is useful to look at the fraction defined by the cumulative sum of the eigenvalues divided by their total sum. Sometimes you can keep only two or three eigenvectors (for visualization, for example), but other times you may want to choose enough eigenvectors to represent some percentage of the data variation (e.g. 90%).
- Take the dot product of each difference vector with the principal component directions. You can treat the eigenvector matrix as a rotation matrix. Each row of deltadata gets dotted with each column of meigvec.

```
pdata = deltadata * numpy.matrix( meigvec )
```

- The resulting set of feature vectors is a compressed representation of the data.

Reprojecting the data

- Calculate the weighted sum of each element of the compressed vector by its corresponding principal component
- The result constitutes a differential in the data space.
- Add the differential to the original mean value of the data .
- The result is an uncompressed version of the data.
- The data compression-reprojection process is a lossy process as some information is lost.

1.13.2 Singular Value Decomposition

Singular value decomposition [SVD] is a method of factoring a single matrix A into three matrices U, W, and V, each with special properties. Given the factorization of A, it is possible to use the factorization to solve systems of linear equations of the form $Ax = b$. SVD is based on a method called the Householder transform. If you are interested in the details, check out the section on SVD in Numerical Recipes in C [cite].

The primary thing to remember is that we can factor the matrix A as in (73).

$$A = UWV^t \tag{73}$$

Properties of SVD:

- Given: A is an NxM matrix.
- U is an NxM matrix with N-element orthogonal columns.
- W is an MxM diagonal matrix, generally represented as an M-element vector.
- V is an MXM matrix with M-element orthogonal columns.

The columns of U form an orthogonal basis for representing the columns of A. They are identical in direction to the eigenvectors of AA^t. The columns of V form an orthogonal basis for representing the rows of A, and are identical in direction to the eigenvectors of A^tA.

The diagonal elements of the matrix W are called the singular values of A, and are related to the eigenvalues of AA^t (and A^tA). The largest singular value corresponds to the primary direction of variation in the data. The second largest singular value corresponds to the next largest, but orthogonal direction, and so on. Singular values at or near zero correspond to the null space of A. The null space is the set of solutions to the linear system that equal zero.

SVD works regardless of the form of A. However, if we are using SVD to solve a set of linear equations, how we interpret the answer we get depends upon the form of A.

If we think of A as a set of linear equations, then if there are more columns than rows, we have a situation where there are more unknowns than constraints, resulting in an underconstrained linear system. There will be singular values that are zero in this case, and the null-space, represented by the columns of V corresponding to the zero-valued singular values, will be at least one-dimensional.

If A is square, then we ought to be able to solve for the unknowns exactly, so long as A has full rank (e.g. is nonsingular, or each row is independent). In this case, all of the singular values should be non-zero, and the system should have a single solution.

If A has more rows than columns, then the system of linear equations is likely to be overconstrained. Therefore, no exact solution to the linear system is possible. However, the solution provided by using SVD will be the solution that is best in a least-squares sense. In many situations, that is exactly what you want. Only in situations where you have significant outliers does it cause problems.

1.13.3 Example: solving for an N-dimensional line

Given a set of M points P in an N-dimensional space, where $M > N$, we would like to fit a line to the points that minimizes the squared error of the representation. The solution to the problem is actually the first eigenvector of the covariance matrix of the points.

If we first calculate the average point for the set $\bar{p}$, then we can create the difference set $P' = \{p_i - \bar{p}\}$. Let A be the matrix of difference points, with each N-element point corresponding to one row of the matrix.

$$A = \begin{bmatrix} p'_{0,0} & \cdots & p'_{0,N-1} \\ \cdots & \cdots & \cdots \\ p'_{M-1,0} & \cdots & p'_{M-1,N-1} \end{bmatrix} \tag{74}$$

The NxN covariance matrix of the point set is defined as $\Sigma = A^t A$. From the discussion above, the columns of V form the eigenvectors of $A^t A$. Therefore, the line that minimizes the squared error as a representation of the point set is the direction given by the column of V that corresponds to the largest singular value. The line must go through the average point $\bar{p}$.

1.14 Noise

Any discussion of data is incomplete without also discussing noise. Noise in a signal is the fraction of the signal not caused by the thing you want to measure. The signal to noise ratio [SNR] is commonly used to express the amount of noise in a system.

Technically, the SNR is the ratio of the average power of the true signal divided by the average power of the noise signal. The power in an A/C signal is the square of the root mean square [RMS] amplitude. Signal to noise ratios are often large, so engineers commonly use the decibel system to express them.

$$\text{SNR}_{\text{dB}} = 20 log_{10} \left(\frac{A_s}{A_n} \right) \tag{75}$$

One decibel is a ratio of approximately 1.1:1. Twenty decibels is a ratio of 10:1 in amplitude. 100dB is a ratio of 100,000:1.

There are many sources of noise, and there are many types of noise. From a mathematical point of view, we can subdivide noise into categories such as Gaussian, uniform, or falling in some other distribution. We can describe noise as biased or zero-mean, which describe the average value of the noise over time.

In addition, noise is often described using colors. White noise is a noise signal that has equal energy at all frequencies within the bandwidth of interest. Pink noise, on the other hand, decreases in amplitude with higher frequencies. The noise at $2f$ is half the power of the pink noise at f.

Nature imposed noise

Johnson Noise: thermodynamic origin

- A resistor has instantaneous current in it (which over time will create power)
- White noise phenomenon, and so constant in magnitude across the spectrum

Shot Noise: quantization of charge

- A white noise phenomenon
- Arises in situations where there are a small number of items being measured (e.g. photons or electrons), so its strength relative to the signal decreases with increasing numbers of items.

$$\text{SNR} = \frac{N}{\sqrt{N}} = \sqrt{N} \tag{76}$$

Flicker (1/f noise): origin is quantum mechanical

- Pink noise
- Related to the uncertainty principle
- Increases in magnitude with decreasing frequency
- Engineers can always hang the blame on some particular device

These are nature imposed sources and provide a lower bound on how well you can do. It is always possible to do much worse.

Human imposed noise

Rotating machinery	Power Lines	Traffic
Radio	TV	Cell phones
Wi-fi	Microwaves	Radar

Other noise sources

Galactic effects	Continental drift	Seasons
Tides	Diurnal effects	Weather
Light	Heat	Cosmic rays

Consider, for example, how patterns of electrical, cell-phone, or wi-fi usage are influenced by the diurnal cycle and how that may impact the level of noise on a particular sensor system.

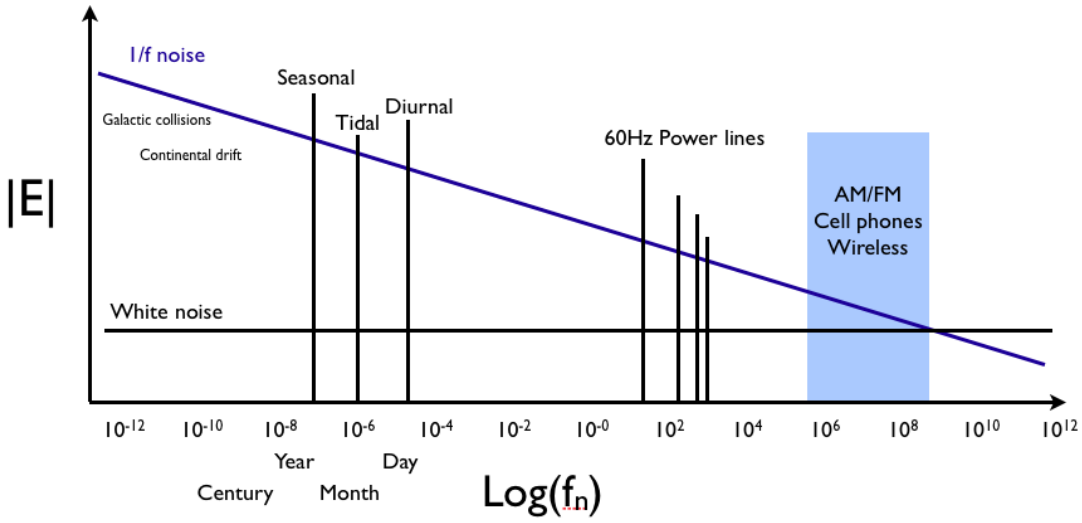

Figure 3: Sources of noise across the frequency spectrum

1.14.1 Measuring Noise

There are a number of methods of measuring noise in a system. The most common is to take a background measurement–or multiple background measurements–and use that as a baseline against which to compare new measurements.

For noise that is unbiased and Gaussian, averaging multiple measurements is the best solution, as the noise tends to cancel itself out. For noise that is non-Gaussian, such as salt-and-pepper noise in an image, using the median of multiple measurements is a better approach, as averaging tends to blur the noise into the signal.

Taking multiple measurements of the background (the measurement system without the primary signal) can also provide an estimate of the variance of the noise in the system. Knowing the level of noise in the system lets us accurately judge when two values measured by the system are the same or significantly different. It can also relate back to picking bin sizes in histograms.

When data sets provide measurements of the background noise levels, we need to make use of it.

1.14.2 Variation versus noise

Noise is something that is not part of the signal we are trying to measure. The variation in a data set does not necessarily correlate with the noise in the data.

- Noise is what you get when the system you are trying to measure is in a constant state, but repeatedly measuring the system produces variation in the results. Therefore, noise limits how accurately you can measure the true state of a system. To call a difference between two measurements significant, in the sense that the difference reflects a change in the true state of the system, the difference in the measurements must be considered relative to the noise levels of the measurement.
- If we could get noise-less measurements, then any variation in the measurements would reflect true variation in the system, and any difference would be significant in the sense that the true value of the system had changed. In real systems, however, the two are conflated.
- Variation in the true system values over a set of measurements causes the measured variance of the data set to be larger than the noise levels. Therefore, we need to be careful about what we mean when we say that two measurements are significantly different in a statistical sense.

Example: Temperature variation

Consider a data set that provides temperature calculations every hour over the course of a year. The true state of the system is the air temperature at the beginning of each hour. The system for measuring temperature inserts a certain amount of noise in the system. As an example, for repeated measurements within a chamber held at a constant temperature the measurements might vary with a standard deviation of 0.05C. Therefore, if we have two measurements that differ by .1C, then we can say the system has likely changed state with a confidence of just over 95% (2 standard deviations).

What are the natural sources of variation in the system?

- Long-term climate change (change over decades)
- Seasonal cycle (1-year period)
- Weather cycles (change over days)
- Diurnal cycle (24 hours)

Note that most of the above sources of variation are much larger than 0.05C. Depending upon what we want to compare, we have to calculate different statistics to talk about significant change. The variance of the data set as a whole tells us about the distribution of temperatures over the course of a year. A temperature reading that is 3σ from the yearly mean will be an unlikely temperature reading for that location. For example, measurements of -40C or 45C or for Maine are unusual in the sense that they do not occur very often.

However, a reading of 45C in July is less significant a change than a reading of 45C in January. If we are talking about the significance of a reading with respect to its context, we have to talk about the variance of the context, not the variance of the entire data set.

---

2 Data Analysis

2.1 Fundamentals of Pattern Recognition and Machine Learning

Pattern recognition is one method of converting data into knowledge. The basic question in pattern recognition is to identify when a new object is sufficiently similar to a previously seen pattern that we can label the new pattern as being the same thing as the known pattern.

One interpretation of pattern recognition is that the computer is subdividing the world of data into a set of classes. In most cases, the classes have some semantic meaning, such as an object or a particular person's face. Another interpretation is that pattern recognition is a method of prediction: if the computer finds a certain set of features, then it predicts, or hypothesizes the existence of a semantic entity.

Pattern recognition and machine learning algorithms require data in order to train and test the resulting classification systems. Organizing and preparing the data plays a significant role in the success of pattern recognition systems.

The most common process for building a pattern recognition system is as follows:

1. Extract features from the raw data.
2. Build a training set of example features from each class we wish to recognize.
3. Build a classifier using the training data.
4. Evaluate the classifier on a test set of labeled data not included in the training set.

Each step above can involve significant complexity, or the complexity may be spread between them. A sufficiently complex and useful feature, for example, may make the remaining steps as trivial as comparing the feature to a threshold. Many medical diagnostics work in this manner where one or more potentially complex measurements are combined to produce a single number or observation, with different values of the number corresponding to different diagnoses.

A large training set can also simplify the remaining steps. A sufficiently large training set generally enables the use of a simpler classifier using simpler features and makes evaluation straightforward. With a sufficiently large data set, visualizations such as histograms provide an accurate representation of the probability of different categories or conditions, enabling accurate functioning of simple probabilistic methods.

A complex classifier may be able to work well with simple features (although it still usually needs plenty of data) because it can consider complex relationships between the features as part of the learning process. Our own brains are an example of such a system. The raw inputs to our senses are fairly simple–activation levels of cells in our ears, eyes, skin, nose, and mouth–yet we extract a complex representation of the world from these signals. Clearly, we require a tremendous amount of data to train our brains.

When working with real data, however, we rarely have enough data, or at least enough labeled data, and we have to balance the complexity of the features with the complexity of the classifier in order to avoid extreme computational requirements. Labeled data is the most valuable commodity in pattern recognition because it is rare, hard to obtain, and greatly speeds learning. Unlabeled data is cheap and easy to generate, but methods of learning from unlabeled data are complex and difficult to get working effectively.

2.1.1 Overview: Building a Pattern Recognition System

The first step in the process is to transform the raw data into a set of features appropriate for learning. The guiding principle of feature extraction should be to design features that are correlated with the desired output. In many cases, the raw data may be only weakly related to particular classes. Transforming the data, or combining it with other variables in the data, may expose the relationship in a way that the machine learning algorithm can more easily use.

Scale is another issue that can arise when dealing with raw data, the same issue as arises in visualization. Some machine learning inputs work best when all input variables are normalized to live within a certain range of values, such as [0, 1]. Section 1.12.2 described three methods for normalizing data sets.

The second step is to generate training data, which is used by the machine learning algorithm to build the classifier that recognizes or predicts events. There are many methods of building training sets.

- For classification problems, a common method of generating training data is to have people label examples of the classes the computer should recognize. The limitation to this method is that people have to manually create the data. This naturally limits the amount of training data that can be produced unless you get very clever about how the labels are generated. For example, if you have one hand labeled example, you may be able to produce multiple other examples through the use of symmetry or perturbation of the original sample. It may also be possible to leverage a process large numbers of people will voluntarily undertake to generate labels.
- For prediction problems, on the other hand, there may be lots and lots of data readily available. Predicting the performance of the stock market, for example, is one of the more lucrative applications of machine learning. In this case, the variable we want to predict–e.g. the DJIA or some other measure–is provided along with all of the other training data. Click-through data on the web is another example of where the label–e.g. whether a person bought a product–can be extracted automatically from the sequence of clicks that led them to the purchase.
- In some cases, we can generate some portion of the training examples automatically. For example, when training classifiers for object recognition in images, it is possible to use random images as negative training examples. It is also possible to bootstrap machine learning systems, using one system to generate labels for learning on a second system and putting the human in the loop only to weed out incorrect labels.
- The most sophisticated machine learning algorithms–but not necessarily the best performing systems–attempt to learn categories and classes automatically from unlabeled or weakly labeled data. These algorithms attempt to find categories of patterns that repeat often in the data. A human in the loop can then attach semantic meaning to particular categories, or provide input that some categories are not meaningful. One issue that arises in learning from unlabeled data is that the learning algorithm may be fooled by correlations or patterns that are not relevant to the actual application.

The third step is to train up the pattern classification or prediction system. We'll go over in detail a few of the more common classifiers and the algorithms for training them.

- Decision Trees
- Cascade Classifiers
- Neural Networks

The final step is to evaluate the classifier or predictor on previously unseen data, or the **test set**. It is important to hold back some of the data, or generate new data so that you can evaluate how well the system does on data it has not seen. The danger with any machine learning algorithm is that it does not generalize to unseen data well enough to be useful.

One way to conceptualize the problem is that each machine learning technique has a certain degree of complexity it can apply to the task of separating the data into categories or predicting a value. Witten and Frank would describe it as the complexity of concepts. Each machine learning technique has parameters that adjust the level of complexity of concepts it can learn. For neural networks, the complexity of concepts it can learn is related to how many nodes and connections are in the network. For decision trees, complexity is also determined by the number of rules in the tree.

In some cases, the problem is so hard that the concepts the ML technique can represent are insufficient to learn the actual task, so it approximates the true task in order to minimize errors on the training set. If the training set is large enough and indicative of typical inputs, then the ML technique will perform similarly on new data as it did on the training data, although in both cases performance may be poor.

In other cases, the problem is so easy, or the training set so sparse, that the ML technique can generate a sufficiently complex concept to correctly classify or predict each instance in the training set. The problem is that when the learned concepts are applied to new data the system is not guaranteed to perform well at all. The system has learned the training set so well that it does not appropriately generalize to new information.

The balancing act of machine learning, therefore, is to adjust the complexity of the ML technique being used to match the task. The role of the training and test set are also tied in with this decision.

- The training set must be large enough to represent the range of typical inputs to the system
- The ML technique should be sufficiently powerful to represent the true concepts in the data, but not so powerful that it can memorize individual instances.
- The test set provides a check on the training process, indicating when overfitting is occurring.

Since most machine learning training techniques are iterative in nature–repeatedly evaluate the learner on the training set and adjust its parameters to improve performance–every few iterations it is important to evaluate the learner on the test set. The performance of the learner on both the training and testing sets should show initial improvement. Usually, performance will begin to bottom out after some number of iterations. If the learner is capable of overfitting the data, then at some point the test set performance will start to go down. The best performing learner is the one just before the drop in performance on the test set.

Figure 4 shows the error on the training and sets sets during learning for a simple neural network being trained to implement the XOR function on two inputs between 0.0 and 1.0. The training set consists of only four examples ((0, 0), (0, 1), (1, 0), (1, 1)), while the test set consists of many more and varied examples. Note how the error on the training set monotonically decreases, while the error on the test set (upper line) starts to increase around iteration 2800.

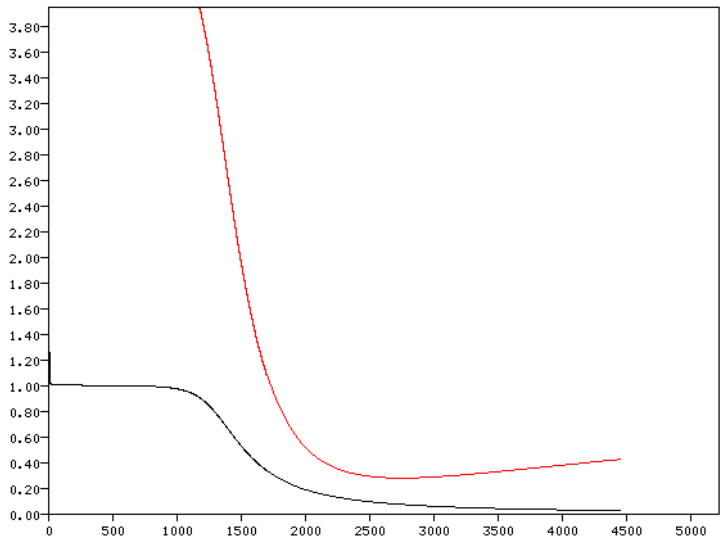

Figure 4: Error rate during training for a four node neural network trying to learn the XOR function.

2.1.2 No Free Lunch Theorem

The 'No Free Lunch Theorem' is one explanation for why machine learning methods require tuning, tweaking, and intelligent selection based on the data and the problem.

The NFL theorem states that all search and optimization algorithms have the same average performance over all problems. In other words, there is no machine learning algorithm–which all fall into the category of search and/or optimization–that will consistently outperform all other algorithms on all problems. Since not all problems are the same, and different ML methods have different internal models, it stands to reason that some ML method is going to be optimal for a problem, but that must be balanced by the same ML method having relatively poorer performance on a different problem.

Note that the NFL theorem makes the assumption that all problems are equally likely. It may be the case that there is a bias to the type of problems that exist in the world, in which case some ML methods may dominate other methods, on average. Factors other than performance–training time, or run time, for example–are not part of the NFL theorem considerations.

It's also very important to keep in mind that factors other than the differences between general learning methods can play a much more significant role in overall performance. Decisions about how to use prior information, the distribution and number of training examples, and the particulars of the cost, error or reward functions can overwhelm any inherent positive or negative impact on performance due to the selected machine learning method.

2.1.3 Ugly Duckling Theorem

The ugly duckling theorem has to do with distance metrics (termed predicates), which are methods used to determine if two feature vectors represent the same category or different categories. The theorem considers all possible ways of measuring similarity and claims that, for any two patterns, the number of possible predicates claiming the patterns are similar is constant. As with the NFL theorem, the ugly duckling theorem implies that careful selection of the predicates for a particular set of categories is critical to the success of a pattern recognition system.

> **Ugly Duckling Theorem**
> Given that we use a finite set of predicates that enables us to distinguish any two patterns under consideration, the number of predicates shared by any two such patterns is constant and independent of the choice of those patterns. Furthermore, if pattern similarity is based on the total number of predicates shared by two patterns, then any two patterns are "equally similar."
> (Duda, Hart, and Stork, Pattern Classification, 2001)

As with the NFL theorem, the UD theorem does not prefer any method of comparing two feature vectors and assumes that every method is equally useful in the space of possible problems. In practice, we tend to prefer distance metrics that prefer to match vectors whose features are similar. We also tend to like distance metrics that obey simple geometric laws, such as the triangle inequality, which says that if the distance between A and B is some distance D_{AB}, then the distances between A and some other point C and between B and C must sum to at least D_{AB}.

$$D(A, B) \leq D(A, C) + D(C, B) \tag{77}$$

2.2 Features

The complexity of a pattern recognition system is distributed across the various stages: feature selection, training set creation, and classifier selection. If you have a sufficiently good feature, then a simple classifier suffices to categorize the data. If you have a sufficiently large training set, then simple features and simple classifiers can learn the categorization. A sufficiently good classifier may be able to learn the categorization using simple features. While the NFL and UD theorems states that no one feature or classifier dominates over all problems, appropriate choices can produce better performance on a specific problem.

Features come in many forms.

- Numeric - integer or real numbers that have meaning on a number line. As all number representations in a computer are digital, there is no effective difference between the two from a feature point of view.
- Categorical data - categorical data may consist of a finite set of strings or integers, neither of which has semantic meaning on a number line. There are many types of data that are categorical. A specific form of binary categorization–the presence or absence of something–is a hybrid of categorical and numeric data when represented as 0 (absence) and 1 (presence) as we can treat the numbers as having meaning on a number line for certain types of machine learning techniques.
- Text data - text data that consists of a finite number of strings is effectively categorical. However, the number of categories may be very large (the size of a complete dictionary of a language). Hence, text is often analyzed using different methods when it does not represent a small set of categories.
- Aggregate types - dates are an example of an aggregate type, as they have three separate fields that need to be handled differently. When comparing dates, a program will often convert the date into a single number–such as seconds since the Epoch (January 1, 1970)–to make the comparison simple. (Dates are a pain to deal with.)

The specific features used to answer a certain problem may or may not incorporate the raw data or measurements. In many cases, we apply our knowledge of the task to create derived features from the raw data. The number of hits on a certain page, for example, is a feature derived from the raw web log data. The first reported arrival of a bird in Maine, given all observer reports, is a derived feature. For the task of object detection in computer vision, the raw pixels values are rarely used as the basis for machine learning. Instead, researchers try to derive features that are more closely related to the object being detected, or that have some properties they believe will make detecting the object in a variety of conditions more robust.

The particular machine learning technique we select for a problem may also determine the kinds of features required. The most common decision tree building algorithm, for example, requires categorical features. Any numerical features must first be converted to a set of categories. A neural network, on the other hand, can accept both categorical and numeric features, so long as both can have a numerical representation, but it will generally work best if all of the features have meaning along a number line and are scaled to a fixed range, such as [0, 1].

Converting numerical data to categories is a potentially complex process. Using an arbitrary categorization process–such as dividing the range of a variable into N equal parts–may be the worst possible categorization strategy. Both choosing N and choosing where to insert the category boundaries are complex problems.

2.2.1 Clustering

Clustering is the process of segmenting data into groups. There are many different clustering approaches, each with their own strengths and weaknesses. In general, most clustering algorithms fall into one of two categories: hierarchical, or partitional. Hierarchical clustering can be a bottom-up agglomerative process or a top-down divisive process. In either case, the algorithm generates a new set of clusters from an existing set of clusters. The initial set of clusters could be the whole data set (top-down) or each data point by itself (bottom-up).

Partitional algorithms generate all of the clusters simultaneously and usually require as input the number of clusters to create. There are also hybrid methods that attempt to combine the two approaches.

Clustering can be part of both the pre-processing stage–creating and generating features–and the basis for a learned classifier. In the pre-processing stage, the purpose of clustering is to identify natural categories within the data. Clustering applies equally to a single variable or to a collection of variables. In the latter case, clustering is a form of data compression, similar in purpose to PCA. Clustering extremely large collections of variables becomes problematic, for a number of reasons, so it is not uncommon for a system designer to first execute PCA on a collection of variables and then generate clusters in a lower dimensional eigenspace.

Some common clustering algorithms including the following:

- Region growing: a form of bottom-up hierarchical clustering, links together chains of data points that are close together. Two data points A and B are in the same group if there is a series of data points connecting A and B such that the distance between any two points along the path is less than a threshold distance ϵ, given some **distance metric** $d(\vec{x_i}, \vec{x_j})$. The strength of region growing is that it is easy to implement and it can identify classes of arbitrary size and shape. The weakness of region growing, or min-link clustering, is that it can link together examples that should be in two different classes if there is a single path connecting them. The phenomenon is also called leakage.

```
Givens:
   a similarity predicate Sim(x_i, x_j)
   a function Neighbor(x_i) that returns a list of the neighbors of x_i
   a set of feature vectors X = {x_i}
   a membership vector M = {m_i} initialized to -1

regioncounter = 0
stack = empty stack
while M has an unmarked entry
  seed = x_i corresponding to the next unmarked entry in M
  M[seed] = regioncounter
  stack.push( seed )

  while the stack is not empty
     cur = stack.pop()
     nlist = Neighbor( cur )
     for each neighbor in nlist
         if Sim( cur, neighbor )
             M[ neighbor ] = regioncounter
             stack.push( neighbor )

  regioncounter++
```

- Agglomerative clustering: a form of bottom-up hierarchical clustering, examines all pairwise distances between data points and, at each step, links the two closest points that are not already linked. Agglomerative clustering produces a dendogram, or tree, which represents a whole set of possible clusterings from each data point as a cluster to the whole data set as one cluster. Each level of the tree corresponds to a different number of clusters.

  Agglomerative clustering can be computationally expensive if, when two leaves join, the algorithm recomputes the distance between the new cluster and all other leaves, using the average value of the new cluster to compute the distances, a process called average-link clustering. Using **average-link** clustering is less susceptible to leakage than **min-link** clustering, where the minimum distance between any two points in a group is the basis for merging.

  Picking the proper number of clusters from the dendogram is challenging. In general, the goal is to balance the simplicity gained by having fewer clusters and the representation error. As the number of clusters decreases, the representation error increases. At some point, the increase in the error is not worth having fewer clusters.

```
Givens:
    a set of feature vectors X = {x_i}
    a distance metric D(x_i, x_j) that obeys the Triangle Inequality
    a set of clusters C, initialized so each contains one feature vector
    a function Merge(c_i, c_j) that merges two clusters
    a set of matches CD, initialized to the distances between all clusters

while the number of clusters is greater than the goal number
    c_i, c_j = clusters from smallest distance in CD
    c_new = merge( c_i, c_j )
    remove c_i and c_j from C
    add c_new to C
    recompute all entries in CD involving c_i or c_j using c_new
    sort CD
```

- Hierarchical subdivision: a form of top-down clustering, divides the data set into two parts and then recursively subdivides the new clusters. The procedure can be breadth-first, in which case the tree stays balanced, or the procedure can always subdivide the cluster with the greatest variation, in which case the clustering tree may be unbalanced. There are many algorithms for implementing the subdivision process that emphasize different characteristics of the resulting clusters. Hierarchical subdivision is not commonly used, but if the subdivision process is handled with care, it can provide a fairly robust segmentation of the data.

- Online clustering: a form of bottom-up hierarchical clustering, does not require the entire data set to be in memory. Instead, online clustering views data one instance at a time. If a new data point is close to an existing cluster, it adds the point to the cluster. If a new data point is far from any existing cluster, the point becomes the seed for a new cluster.

  The number of clusters produced by online clustering depends upon the threshold used to determine if a new data point is sufficiently close to an existing cluster. A tighter threshold results in more clusters. Online clustering is often used for very large data sets when other methods are not practical. Online clustering works event for data sets of millions or billions of points. Because it handles the data one feature vector at a time, it can also be used to cluster data collected in real time.

```
Givens:
    a set of feature vectors X = {x_i}
    a distance metric D(x_i, x_j) that obeys the Triangle Inequality
    a set of clusters C, initially empty
    a predicate Like( c_i, x_i ), true if x_i is close enough to c_i
    a function Add( c_i, x_i) that adds x_i to cluster c_i

for all feature vectors x_i
    found = False
    for all clusters c_i in C
       if Like( c_i, x_i )
           Add( c_i, x_i )
           found = True

    if not found
        c_new = create a new cluster from x_i
        add c_new to C
```

- K-means clustering: a form of partitional clustering, iteratively subdivides the data set into a set of K clusters, where K is predetermined. The algorithm begins by picking an initial set of K cluster means. It then iterates between two steps: 1) assign each data point to a cluster mean, 2) update each cluster mean to be the average of the data points assigned to the class. K-means is relatively fast and generally performs well in practice if K is appropriate and the algorithm uses a reasonable method of picking the initial means.

```
Givens:
  a set of feature vectors X = {x_i}
  a membership vector M = {m_i} initialized to -1
  a distance metric D(c_i, x_j) that obeys the Triangle Inequality
  a set of K clusters C
  a set of K clusters C_old, set to zeros
  a function Change(C, C_old), returns the distance between cluster sets

build C from K feature vectors
while Change(C, C_old) is bigger than a (small) threshold
    for each feature vector x_i in X
        closestID = c_0.id
        closestMean = c_0.mean
        for each cluster c_j in C
            d = D( c_j, x_i )
            if d < D( closestMeant, x_i)
                closest = c_j.id
                closestMean = c_j.mean

        M[ i ] = closest.id

    C_old = C
    C = Compute new cluster means given X, M
```

Picking an appropriate K can be complex. Each data set will have a different number of natural clusters. In some cases, visual exploration of the data reveals an appropriate K. Researchers have also developed a number of different measures of the quality of a particular clustering. Comparing the quality of the clusters from various values of K is one method of picking the best K for a particular data set.

Initializing the K-means clustering algorithm usually involves a random process. For real data sets, this means that a particular clustering may not be representative. Some data sets exhibit stability regardless of the initial cluster choices, while others are highly dependent upon them. It is possible to estimate cluster stability by executing the algorithm multiple times and counting how often each point is grouped with the same set of points.

- Fuzzy C-means: a form of partitional clustering, is a continuous variation of the discrete K-means algorithm. Instead of using a binary classification of each data as being in a class or not, the fuzzy c-means algorithm assigns a membership weight to each data point for each cluster by its distance from the cluster mean. The algorithm then updates the cluster means using weighted averages of all points in the data set. Fuzzy C-means is a specific instance of an optimization method called the Expectation-Maximization, or the EM algorithm.

  Since each data point is a member–with some weight–of each cluster, the set of membership values makes a potentially useful feature vector itself.

  The algorithm for fuzzy C-means is almost identical to K-means. The difference is that instead of testing only which cluster is closest, the membership vector holds the distance from every cluster to every point. The distances then affect the new cluster computation as each feature vector contributes something to each cluster mean (although it can be very small).

```
Givens:
  a set of N feature vectors X = {x_i}
  a N x K membership vector M = {m_i}
  a distance metric D(c_i, x_j) that obeys the Triangle Inequality
  a set of K clusters C
  a set of K clusters C_old, set to zeros
  a function Change(C, C_old), returns the distance between cluster sets
  a function Member(d), converts a distance into a membership

build C from K feature vectors
while Change(C, C_old) is bigger than a (small) threshold
    for each feature vector x_i in X
        for each cluster c_j in C
            M[x_i.id][c_j.id] = Member( D( c_j.mean , x_i ) )

    C_old = C
    C = Compute new cluster means given X, M
```

- Spectral clustering: a form of partitional clustering, represents the data as a graph of relationships and identifies the subdivision of the graph that minimizes the cost of cutting connections. Spectral clustering uses the eigenvectors of the data when represented as a matrix of relationships to determine where to make the cut. There are a number of different spectral clustering methods that optimize different criteria for making the cut.

The output of any clustering method is a set of **class descriptors**–usually cluster means–and **membership labels** for the training data. The set of class descriptors represent a set of concepts. We can use the cluster labels as features for a classifier that requires categories, or we can use the cluster means themselves as a simple classifier, attaching to the a new data point the label of the cluster mean to which it is closest. In the case of fuzzy C-means, we can use the membership weights as a feature vector for another classifier, or categorize a feature vector as its highest membership cluster.

Mathematically, each cluster, or concept is represented by a feature vector. Often, however, we may also want to assign a semantic concept to each cluster. In most cases, a person assigns a meaningful semantic label to each cluster. However, it is also possible to assign labels automatically by analyzing the key characteristics of a cluster that differentiate it from all other clusters. One algorithm that seems to work reasonably well for clusters built from multi-variable data sets is as follows. The end result is a label that contains from one to three features with the adjectives low, medium, or high.

```
Given: a set of cluster means with N features

For each cluster i
    Initialize a list C of N integers to 0
    For each cluster j where j != i
        calculate the feature where i and j are most different
        increment C[j]
    Search C to identify up to three features with the highest non-zero counts
    For each label feature divide the range into thirds
    Give each label feature the adjective low, medium, high
```

Clustering is often used as an exploratory method when analyzing data, although it is also a common first step in analysis of images or other natural data. Clusters provide aggregations of basic data upon which we can run more complex algorithms that look at aggregate properties.

No clustering algorithm is best in all situations. Given very large data sets, online clustering may be the only option computationally. K-means clustering tends to work well in most situations and is a useful all-purpose clustering tool. Fortunately, it is also simple to implement.

2.3 Selecting an Optimal Number of Clusters

What is the right number of clusters?

- Balance representation error and model complexity
- Look for the point where the increase in model complexity is greater than the reduction in error

Rissanen's Minimum Description Length

- Term 1: $-\log_2 P(x^n|\Theta)$ the log probability of the data given the model

  If the distances are calculated in a space scaled by the standard deviations of the various dimensions, then the log probability is the the sum of the squared distances from each point to its associated cluster mean times the dimension of the data D: $D * SSE$.

- Term 2: $\frac{Dk}{2} \log_2 n$ the number of clusters times the dimension of the data times the log of the number of points

  Note, since the dimension of the data D appears in both terms, we can leave it out of the description length equation, since it does not change the balance between the two terms.

$$DL = -\log_2 P(x^n|\Theta) + \frac{k}{2} \log_2 n = SSE + \frac{k}{2} \log_2 n \tag{78}$$

Krzanowski and Lai, "A Criterion for Determining the Number of Groups in a Data Set Using Sum-of Squares Clustering", Biometrics 44, 23-34, March 1988.

The measure is based on how the representation error (SSE) changes as the number of clusters increases. It defines the DIFF(k) function as

$$\text{DIFF}(k) = (k-1)^{\frac{2}{p}} SSE(k-1) - k^{\frac{2}{p}} SSE(k) \tag{79}$$

$$\text{R\&L}(k) = \left| \frac{\text{DIFF}(k)}{\text{DIFF}(k+1)} \right| \tag{80}$$

where k is the number of clusters, p is the number of variables, and SSE(k) is the SSE with k clusters.

The goal is to maximize this value. Basically, this means that if the SSE drops a lot from k-1 to k clusters but drops very little from k to k+1, then stopping at k is probably a good idea. However, if the SSE drops very little from k-1 to k clusters but drops considerably from k to k+1, then it makes more sense to use k+1 clusters than k clusters.

Ray and Turi, "Determination of Number of Clusters in K-Means Clustering and Applications in Colour Image Segmentation", 2000

The logic of Ray and Turi is to find the ratio between the intra-cluster distances–the average distance of points to their associated cluster mean–and the inter-cluster distances. To compute the inter-cluster distance, they propose using the minimum distance between any two clusters.

$$\text{intra} = \frac{1}{N} \sum_{i=0}^{K-1} \sum_{x \in C_i} ||\vec{x} - \vec{z}_i||^2 \tag{81}$$

$$\text{inter} = \min_{\substack{i=0,\ldots,K-2 \\ j=i+1,\ldots,K-1}} (||z_i - x_j||^2) \tag{82}$$

$$RL = \frac{\text{intra}}{\text{inter}} \tag{83}$$

2.4 Distance Metrics

Classifiers have a hierarchy of complexity and take many different approaches to making decisions. One way to think about the complexity is by thinking of the feature space as an N-dimensional space with boundaries between the different classes. A more complex classifier permits more complex boundaries to define the difference between classes. In some cases this is a good thing, and permits carving out small, oddly shaped areas of the feature space. In other cases, we want smoother boundary shapes in order to avoid learning things that are too specific about a class.

There are many ways to measure similarity between feature vectors. Some are more appropriate than others, and the particular distance measure often depends upon the type of data being compared.

For starters, we would like distance metrics to obey the *Triangle Inequality* for a metric space.

$$D(\vec{x}, \vec{z}) \leq d(\vec{x}, \vec{y}) + d(\vec{y}, \vec{z}) \tag{84}$$

The triangle inequality is important because it means that distances are meaningful, and there are no shortcuts or wormholes in the space. We would also like distance metrics to be commutative so that $d(a, b) = d(b, a)$.

Mean absolute distance (also called L1 distance): not as sensitive to outliers as higher order distances.

$$D_{\text{L1}}(\vec{x}, \vec{y}) = \sum_{i=1}^{N} |x_i - y_i| \tag{85}$$

Normalized mean absolute distance: takes into account the variances in each dimension

$$D_{\hat{\text{L1}}}(\vec{x}, \vec{y}) = \sum_{i=1}^{N} \frac{|x_i - y_i|}{\sigma_i} \tag{86}$$

Euclidean distance (also called L2 distance): optimal distance measure for Gaussian noise.

$$D_{\text{L2}}(\vec{x}, \vec{y}) = \sum_{i=1}^{N} (x_i - y_i)^2 \tag{87}$$

Normalized Euclidean distance: takes into account the variances of the different dimensions.

$$D_{\hat{\text{L2}}}(\vec{x}, \vec{y}) = \sum_{i=1}^{N} \frac{(x_i - y_i)^2}{\sigma_i^2} \tag{88}$$

Mahalanobis distance: takes into account the covariances of the different dimensions.

$$D_{\text{Maha}}(\vec{x}, \vec{y}) = (\vec{x} - \vec{y})^T \Sigma^{-1} (\vec{x} - \vec{y}) \tag{89}$$

Hausdorff distance: defined as the maximum distance between corresponding elements of a set. Hausdorff distance has been shown to be useful in matching geometric shapes.

The family of distance metrics like mean absolute distance and Euclidean distance are called LN metrics. Mean absolute distance is L1, Euclidean is L2. Hausdorff distance is effectively an $L\infty$ metric, which means the largest difference between any two elements of the feature vectors dominates the distance.

Binary distance: returns the number of similar features in two vectors. The similarity metric–whether two values are the same or not–can use any other distance metric to make the determination. The similarity metric can also be crisp (return 0 or 1) or fuzzy (return a continuous number between 0 and 1). Binary distance is useful for matching collections of heterogeneous features that include enumerated or non-numeric types where the difference between two values on a number line is not meaningful.

Intersection distance: defined as the sum of the minimum value of corresponding elements. Useful for matching histograms.

Earth Mover's Distance: defined as the minimum amount of change required to convert one set into another. EMD is often used as a robust distance measure for histograms.

Edit Distance: defined on strings as the number of operations required to transform one string into another. It is similar to EMD. There are a number of different specific edit distance metrics that depend on the types of editing operations allowed: replace, delete, insert, transpose, skip, etc.. The most commonly used edit distance is Levenshtein distance, which allows insertion, deletion, or substitution of a single character and counts the number of operations.

Correlation distance: a normalized Pearson's Correlation Coefficient applied to corresponding elements of the two vectors. If the two feature vectors represent a time series, this can be a useful metric for measuring their similarity independent of the absolute scale of the values. Two vectors will have a small distance if the corresponding elements of the feature vector vary together.

$$d_r(\vec{x},\vec{y}) = 1 - \frac{\sum_{i=1}^{N}(x_i-\hat{x})(y_i-\hat{y})}{\sqrt{\sum_{i=1}^{N}(x_i-\hat{x})^2}\sqrt{\sum_{i=1}^{N}(y_i-\hat{y})^2}} = 1 - \frac{\sum x_i y_i - n\hat{x}\hat{y}}{\sqrt{\sum x_i^2 - n\hat{x}^2}\sqrt{\sum y_i^2 - n\hat{y}^2}} \tag{90}$$

Cosine distance/angular distance: measure the angle between two vectors. Cosine distance is useful for situations where the feature vectors can be thought of as vectors radiating from the origin and the angle between the vectors is the critical element. This is also useful for comparing points arranged on a spherical surface.

$$d_{cos}(\vec{x},\vec{y}) = \frac{\vec{x}\cdot\vec{y}}{||\vec{x}||||\vec{y}||} \tag{91}$$

R	GR	GR	GW	GW	GW	GG	GG
W	WR	WR	WW	WW	WW	WG	WG
G	RR	RR	RW	RW	RW	RG	RG
	R	R	W	W	W	G	G

Figure 5: Dynamic time warping matrix for two patterns.

Dynamic Time Warping: An algorithm for measuring the similarity between vectors of potentially differing length, where each element of the vector represents a location in time or space.

DTW is one method of defining a framework that provides a rigorous definitions of symbols and how they can match up with an exemplar symbol. While dynamic time warping was originally designed to compare one-dimensional time signals, such as digitized speech, it is also applicable to multi-dimensional signals with spatial extent. The fundamental concept in DTW is that a symbol in a gallery pattern can match more than one symbol, or no symbols, in the probe pattern, but the ordering of the symbols must be preserved.

For example, given the gallery pattern RWG and the probe pattern $RRWWWGG$, the two patterns would have a low error in DTW matching because each duplicated symbol in the probe would match up with the same symbol in the gallery. However, if the probe pattern looked like $RWRWGWG$, the match would be poor, because at least two the symbols would match incorrectly. There is no way to stretch RWG to match perfectly a pattern with interleaved symbols.

Formally, dynamic time warping returns the minimum cost path that matches the gallery and probe images, starting at the first element of each pattern and ending at the final element of each pattern. Visually, if we lay out one pattern along the X-axis and the second along the Y-axis, they form a matrix with each location in the matrix defining a potential match two symbols, as shown in figure 5. The minimum cost path minimizes both the error between the symbols and the cost of moving through the matrix.

A recursive definition of the DTW cost from the origin of the matrix to any other location in the pattern match matrix is given in (**??**).

$$D(x_i, y_i) = \gamma(x_i, y_i) + \min_{x', y'} \left(D(x', y') + \zeta((x', y'), (x_i, y_i)) \right) \tag{92}$$

$\gamma(x_i, y_i)$ is the cost of matching the the ith element of x and y.

$\zeta((x', y'), (x_i, y_i))$ is the cost of moving from a prior element (x', y') to element (x_i, y_i).

$D(x', y')$ is the cost of the path from the start to element (x', y'). The recursive way to think about DTW is that the optimal path ending in the upper right corner of the diagram is the minimum sum of the optimal path to any possible prior node plus the cost of getting to the last node. The recursion ends when you get to the first node, whose cost is simply the matching cost of the first elements of each vector.

There are many variations on DTW, primarily variations on the rules for moving from node to node and the relative costs. For example, in some situations it may be permissible to skip a symbol.

2.5 Classifiers

Once we convert our raw data into a set of feature vectors, we generally create some type of classifier in order to learn patterns in the data. There are many types of classifiers with different requirements for training data.

One important attribute of classifiers is whether they fall into the category of **supervised** or **unsupervised** learning systems. A supervised learning system has input-output examples. For each input, the training set includes the expected output. Decision trees and feedforward neural networks are examples of supervised learning systems. The training process learns their structure and functionality from the training set.

An unsupervised learning system has only input feature vectors. The training set contains no indication of the appropriate output for a given input example. K-means clustering is one example of an unsupervised classifier. The training set contains only data, and the resulting clusters form naturally from the data topology. The cluster id for each feature vector is a form of classification. We can classify new data by finding the closest cluster mean.

While unsupervised learning is generally more difficult than supervised learning, and its outputs do not always have clear semantic meaning, the machine learning community continues to pursue unsupervised learning algorithms. Unsupervised learning is appealing for two primary reasons.

- There is lots of data in the world, but few labels or input-output examples.
- Natural learning systems (e.g. people) often use unsupervised learning.

Researchers have also developed hybrid methods that bootstrap an unsupervised learning system for a large data set using a small labeled input-output data set. [need a reference or two here]

In the following sections we'll take a look at several types of supervised classifiers, their structure, how they learn from data, and how we make use of them on new data.

2.5.1 Naïve Bayes

If we have input-output examples, one way to approach the problem of classification is through probability. Given a feature vector $\vec{x}$, what is the probability of an output A given $\vec{x}$? If we know the set of possible processes that could have produced $\vec{x}$, then we can estimate the probability of each output given the feature vector.

Example data set:

The follow samples are drawing from two Gaussian distributions. The mean of the A category is $(1, 1, 1)$, and the mean of the B category is $(2, 2, 2)$. The standard deviations of the three dimensions are $(2.0, 1.5, 0.5)$. The H and L values indicate whether the corresponding D value is above or below 1.5.

Class	D1	D2	D3	C1	C2	C3
A	2.60	1.91	1.10	H	H	L
A	1.61	3.34	0.56	H	H	L
B	2.44	3.83	1.95	H	H	H
A	2.21	1.76	2.12	H	H	H
B	3.25	-0.52	2.15	H	L	H
B	3.61	1.67	1.02	H	H	L
B	4.33	3.06	1.72	H	H	H
A	-1.09	0.36	1.04	L	L	L
B	3.67	0.59	1.66	H	L	H
B	6.16	4.15	1.35	H	H	L
B	-0.32	3.04	1.73	L	H	H
A	1.50	1.17	0.86	L	L	L
A	-0.14	0.13	0.45	L	L	L
A	0.51	1.99	0.99	L	H	L
A	1.83	3.03	0.92	H	H	L
B	3.85	1.18	2.58	H	L	H
B	2.57	2.32	2.11	H	H	H
B	5.37	2.11	2.45	H	H	H
A	0.43	1.11	0.65	L	L	L
B	-2.02	1.64	0.74	L	H	L

p(A) 0.45
p(B) 0.55

Dim	$p(A\|L)$	$p(A\|H)$	$p(B\|L)$	$p(B\|H)$
0	0.71	0.31	0.29	0.69
1	0.57	0.38	0.43	0.62
2	0.73	0.11	0.27	0.89

Dim	$p(L\|A)$	$p(H\|A)$	$p(L\|B)$	$p(H\|B)$
0	0.56	0.44	0.18	0.82
1	0.44	0.56	0.27	0.73
2	0.89	0.11	0.27	0.73

If we have a set of measurements $\vec{B}$ and we are trying to predict some outcome A, a simple way to approach the problem is to use training data to estimate the probability of an outcome given the measurements, or $P(A|\vec{B})$, which is given in the first table above. It turns out it is difficult to directly estimate the conditional probability of a class given a set of measurements. The issue is that there are many possible combinations of measurements, and to correctly estimate the probability of a class given the measurements there must be many examples of each possible input.

Fortunately, Bayes rule gives us a relationship between the conditional probability we want to discover and probabilities that are much easier to measure from training data.

Bayes rule

$$P(A|\vec{B}) = \frac{P(\vec{B}|A)P(A)}{P(\vec{B})} \tag{93}$$

The probabilities in the numerator on the right side turn out to be easy to compute, and are given in the

second table above. To obtain the probability of a class, we just look at the prevalence of the class in the training data (which needs to be well sampled). We can estimate the conditional probability $P(\vec{B}|A)$ by looking at all of the class A training instances and calculating the probability of each possible measurement value. If the measurements are enumerated, the probabilities are simple to calculate. If the measurements are numeric, then we can either model the dimension using a probability distribution function (e.g. a Gaussian) or we can explicitly represent the distribution of measurements as a normalized histogram. In either case, calculating the probability is straightforward given the training data.

The one caveat to the above statement is that if the input measurements are not independent variables, then the estimation of the conditional probability becomes much more compex. A common heuristic is to assume that the dimensions of $\vec{B}$ are independent, in which case we can calculate each dimension's probabilities independently. Making the assumption that the measurements in $\vec{B}$ are independent gives you a method called naïve Bayes.

If we assume measurements are independent (which is why the method is called naïve Bayes), then we can multiply the conditional probabilities for each dimension to get the conditional term in the numerator of the right side. After calculating the numerator for each possible output class, we normalize so the sum of the probabilities is one. The label with the highest probability given the data should be the label for the unknown example.

As an example, consider the input string LHH from the example above. The probabilities tell us that the measurements were more likely from a category B object.

$$\begin{aligned} P(A|\vec{X}) &= \frac{P(\vec{X}|A)P(A)}{P(\vec{X})} \\ &= \frac{P(\vec{X}_0|A)P(\vec{X}_1|A)P(\vec{X}_2|A)P(A)}{P(\vec{X})} \\ &= \frac{0.56 \times 0.56 \times 0.11 \times 0.45}{P(\vec{X})} = \frac{0.015}{P(\vec{X})} \end{aligned} \tag{94}$$

$$\begin{aligned} P(B|\vec{X}) &= \frac{P(\vec{X}|B)P(B)}{P(\vec{X})} \\ &= \frac{P(\vec{X}_0|B)P(\vec{X}_1|B)P(\vec{X}_2|B)P(B)}{P(\vec{X})} \\ &= \frac{0.18 \times 0.73 \times 0.73 \times 0.55}{P(\vec{X})} = \frac{0.053}{P(\vec{X})} \end{aligned} \tag{95}$$

A second example shows that the string LLL is more likely to be from an A category object.

$$\begin{aligned} P(A|\vec{X}) &= \frac{P(\vec{X}_0|A)P(\vec{X}_1|A)P(\vec{X}_2|A)P(A)}{P(\vec{X})} \\ &= \frac{0.56 \times 0.44 \times 0.89 \times 0.45}{P(\vec{X})} = \frac{0.099}{P(\vec{X})} \end{aligned} \tag{96}$$

$$\begin{aligned} P(B|\vec{X}) &= \frac{P(\vec{X}_0|B)P(\vec{X}_1|B)P(\vec{X}_2|B)P(B)}{P(\vec{X})} \\ &= \frac{0.18 \times 0.27 \times 0.27 \times 0.55}{P(\vec{X})} = \frac{0.0072}{P(\vec{X})} \end{aligned} \tag{97}$$

Topics:

- Review building a classifier (e.g. two category problem) with a single measurement
- Calculating $p(x|\omega_i)$ using a histogram
- Calculating $p(x|\omega_i)$ by fitting a 1-D Gaussian
- Building a classifier with two or more measurements
  - If they are independent, multiply the items in the numerator
  - denominator is a sum term

$$p(\vec{x}) = \sum p(\vec{x}|\omega_i)p(\omega_i) \tag{98}$$

- If they are dependent, model as a single multi-dimensional Gaussian distribution (or sum of Gaussians); or apply PCA to the input data space and project onto the important eigenvectors.

2.5.2 Decision Trees

Decisions trees are similar to the game 20 questions, but the questions don't have to be yes/no. The questions form a tree structure, with the first question being the starting node. Each leaf of the tree represents a label for the input feature vector. The particular set of questions in the tree depends on the form of the feature vector and the training data.

- A decision tree is a set of rules organized in a tree structure with a root node and leaves.
- Each node is a classification rule that subdivides the data set into two or more parts
- Each leaf is associated with an output rule
- To classify a novel instance, begin at the root node and use the results of each classification rule to get to a leaf node and its output rule.

Decisions trees can use many different types of classification rules.

- Enumerated types: one branch for each type, or one branch for each of a set of mutually exclusive subsets of the categories. If there is one branch for each type, the variable will not be used again lower down in the tree.
- Numeric types:
  - A simple threshold test of a variable with a constant value: two branches
  - An interval test with different branches for above and below the interval: three branches
  - Comparisons between different variables
- Missing data:
  - Make a special branch in the tree for the missing data category
  - Use the most popular branch for real data given the training set
  - Split the instance and send it down multiple branches, then recombine the outputs using the likelihood of each branch given the training set

Each leaf of a decision tree represents a single path through the tree from the root node. The collection of rules along the path form a single rule we could write as a (potentially complex) if statement. If we were to write down a rule for each leaf node, we would end up with a set of classification rules. We can take a novel instance and implement the same function as the decision tree using the set of leaf rules. One important characteristic of this set of rules is that the order in which they are applied to the instance is immaterial. Every instance will meet the criteria of one and only one rule in the set.

The set of if-then rules in a decision tree subdivides the problem space into many small pieces. The expectation is that, if the features are related to the desired output classes, then it should be possible to subdivide the space so the small pieces are close to uniform. There are many forms of decision trees. The key parameter of a decision tree is it size, or complexity. A more complex decision tree divides the input space into many smaller pieces, giving it the ability to specify complex and precise boundaries. However, a simpler decision tree may often capture the decision boundary more accurately and better generalize to new data.

When building decision trees, we are always trying to choose what question to ask next. When picking a question, there are two factors we want to measure. First, we would like to maximize how much we learn from the question, regardless of whether the answer is yes or no. One way of thinking about the information

gain is by considering how much of the input space we discard, depending on the answer. The best we can do is discard half of the possible input space no matter what the answer is.

Second, we want the likelihood of the correct class to increase in the input space that remains. In other words, we are both subdividing the problem and making it more likely that if we make a guess about the output class that we'll be correct. The training set provides us with data with which we can evaluate different questions and choose what is the best question to ask next. The training set also tells us when a particular subset of the input space is likely to have a single output class, in which case we don't have to add more nodes to that part of the decision tree.

Information Content is a way of measuring how many more nodes, or questions, we are likely to need to complete a branch of a decision tree. The fewer additional nodes we need, the better the original question. A branch that is close to pure (only one output class) is likely to need fewer questions to complete. A branch that has training samples with approximately equal numbers of two output classes is likely to need more questions in order to separate the two classes. The best possible question would separate the training data into one branch per output class, in which case the tree requires no additional questions and is complete.

Information theory tells us that **entropy** is the function that best captures the idea of information content. Entropy is defined as in (99).

$$E(x) = -p(x)\log_2(p(x)) \tag{99}$$

The entropy of a branch of a decision node is the sum of the entropies of each output class. For example, consider a branch that contains training examples from two output classes, 6 from one class and 4 from another. The probability of each output class is 0.6 and 0.4, respectively.

$$E([0.6, 0.4]) = -0.6\log_2(0.6) - 0.4\log_2(0.4) = 0.971 \tag{100}$$

Now that we can measure the information content in a single branch, we can calculate the information content of all of the branches of a decision node. Multiplying each branch by the probability the branch will be taken provides the total information content of the node. A node whose branches contain zero information content has pure branches: there is only one output class in each branch, so there is no need to ask any more questions.

The information gain due to a split at a node is the difference between the entropy of the branch data coming into the node and the weighted sum of the entropies of the branches leaving the node. For example, consider a node receiving 20 training samples of two classes that divide evenly as [10, 10]. The question at the node divides the data into 3 branches $[6, 7, 7]$. Within each branch, the division into two classes breaks down as $([4, 2], [5, 2], [1, 6])$.

The information content in the incoming data collection is $E([10, 10]) = -0.5\log_{0.5} -0.5\log 0.5 = 1.0$. The information content in the outgoing stream is given by the entropy of the outgoing streams multiplied by their likelihood in the training set.

$$\begin{aligned} \text{IC} &= \frac{6}{20}E([4,2]) + \frac{7}{20}E([5,2]) + \frac{7}{20}E([1,6]) \\ &= (6 * 0.918 + 7 * 0.863 + 7 * 0.591)/20 \\ &= 0.785 \end{aligned} \tag{101}$$

The **information gain** is the difference in the incoming and outgoing branches. In this case, the information gain is $1.0 - 0.785 = 0.215$. Consider the same data, but a question that divides it into two branches with the subdivision into classes as $([9, 1], [1, 9])$.

$$\begin{aligned} \text{IC} &= \frac{10}{20}E([9,1]) + \frac{10}{20}E([1,9]) \\ &= (10 * 0.469 + 10 * 0.469)/20 \\ &= 0.469 \end{aligned} \tag{102}$$

The alternative split would provide an information gain of 0.531, or more than twice the information gain of the first branching. Therefore, the second branching is likely a better choice.

One problem with using information gain directly is that it tends to prefer decision trees that subdivide the task into many small pieces. A method often used to counter this tendency is to express the information gain as the ratio of the node's information gain to the entropy of how the data gets split by the question (intrinsic information). The entropy of the data split is simply the information content of the subdivision of the data into the branches regardless of class. A question that subdivides all of the incoming training samples into equal parts has a high entropy. A question that sends a large number of samples into one branch and few into another has a low entropy. Using the ratio of information gain to intrinsic information favors nodes that produce fewer divisions, but balances that with the ability of the question to subdivide the task in a useful manner.

The intrinsic information in the 3-way split is E(6, 7, 7) = 1.718. The intrinsic information in the 2-way split is E(10, 10) = 1. Therefore, even if the two questions produced similar information gains, the two-way split gives a smaller denominator and should be preferred because it gives a larger information gain ratio.

In some cases the gain ratio overcompensates, so a heuristic is to select the node with the largest gain ratio, so long as its information gain is at least as big as the average information gain for all nodes considered. That way a node with a small information gain, but a large gain ratio, does not dominate over a node that has a higher than average information gain, but not as equal a division of the data.

We now have enough knowledge to begin constructing decision trees.

- Build a node for each attribute and calculate the information gain, intrinsic information, and gain ratio
- Pick the node with the largest gain ratio (see caveat above) and insert it into the tree
- Any branch with zero information content gets an output label
- Keep adding nodes until there are no more attributes to select or every leaf node has an output label

While the above procedure creates a perfectly fine decision tree, it does not balance generalization with specificity. The best decision tree algorithms use additional information to choose when to subdivide a node, and they prune branches from trees once they are completed. Otherwise, every single detail in the training data will contribute to the tree, whether or not it is relevant to the task.

It is important to note two attributes of the process.

- The decision about whether to build a node, and what question to ask is part of a local search process, or a **greedy search** process. That means the tree is built through a series of decisions that do not take into account global information about the structure of the tree, but consider only the immediate value of a question on the components of the training set going through that branch.

- The decision process itself is heuristic, which means there is no provably correct method of picking a decision tree node. Experience and experimentation have yielded an algorithm that generally performs well within a greedy search process. However, the free lunch theorem suggests that certain parameters of the tree building process should be evaluated when building a tree for a particular task.

1R Trees: Calculate a rule for each attribute and pick the best one. In some cases, a complex decision tree is not warranted. For many tasks, there may be a single variable in the data set that acts as an effective proxy for the desired output classes. A 1R tree is a single decision node. In some cases, a 1R tree works as well or better than other classifier methods, especially for small training sets.

The information gain is a useful method of selecting which 1R tree to use. For the example data set, there are three possible 1R trees. Each dimension divides the data set into two parts. The initial information content of the data set is $[9, 11] = 0.993$. The information gain for each of the 1R trees is given in table 1. From the table, category three is clearly the best option and provides the highest information gain.

Category	L Branch	L IC	H Branch	H IC	Information Gain
1	[5, 2]	0.863	[4, 9]	0.890	0.112
2	[4, 3]	0.985	[5, 8]	0.961	0.024
3	[8, 3]	0.845	[1, 8]	0.503	0.302

Table 1: Information gain of 1R trees from example data set

Numeric attributes are common features in data, but so far we have examined only enumerated attributes in building decision trees. To make use of numeric attributes we have to determine a method of separating the data into two or more branches. There are many ways to go about making a decision node.

- Fix the number of branches (e.g. 2) and then use the information gain of the possible divisions to determine where best to split the data. The cost of building a node this way is O(N).
- Fix the number of branches and use a clustering algorithm (K=2) to select the division point. This has the benefit of separating the data in a meaningful way, but doesn't necessarily do anything to improve the classification performance. Clustering is also approximately O(N).
- Calculate the average value of the variable for each output class and pick decision points based on the means and standard deviations of the distributions. This is also an O(N) computation.

One difference between numeric attributes and categorical ones is that a numeric attribute may be used in several different nodes within a branch. Since categorical outputs are all the same in a sub-branch, there is no further need for subdivisions.

Missing values can present problems when building a decision tree. As noted earlier, one solution to missing attributes is to send the instance down two branches and then combine the results of the resulting leaves using a weighted average. It is possible to do the same thing when building the decision tree by using non-integer branch weights when calculating the information gain for a node. For sub-trees after a split decision, the training sample still contributes via its likelihood of going down that branch.

Pruning decision trees is the trickiest part of building useful trees. Almost all data sets contain some noise, so creating a perfect decision may be either impossible or undesirable from a generalization standpoint. It may be better to stop subdividing a particular branch of the tree well before all of the attributes have been used or the branch is pure. Pruning uses two different methods of reducing the complexity of trees.

- Subtree replacement: eliminate a subtree and replace it with a single leaf node
- Subtree raising: replace a parent node by one of its subtrees

Subtree replacement, which is basically a traditional tree pruning operation, is the most common and effective method of decision tree pruning. The basic idea is to look at the expected error rate of a subtree and compare it to the expected error rate of replacing the subtree with a single leaf node. If the expected error rate of the subtree is not significantly better than the error rate of the single leaf node, then prune away the branch.

We can estimate the error rate of a node by using the formula for occurrences of a class resulting from a Bernoulli process. A Bernoulli process models coin flipping multiple times, possibly with an unfair coin. Given a confidence limit c, we can use a lookup table to find the z-value that corresponds to it. For example, a 25% confidence limit is a z-value of 0.69, or 0.69 standard deviations. Because we end up using training data to estimate the expected error rate (not a great idea) we have to use a pessimistic estimate of the error, which is basically an upper bound on the error (with c confidence). If f is the error rate, N is the number of samples, and z is the z-value corresponding to our chosen confidence, the upper bound on the error is given by (103).

$$e = \frac{f + \frac{z^2}{2N} + z\sqrt{\frac{f}{N} - \frac{f^2}{N} + \frac{z^2}{4N^2}}}{1 + \frac{z^2}{N}} \tag{103}$$

If a node has 5 examples of class A and 14 examples of class B, then replacing the decision node by a leaf that guesses B results in 5 errors, or an error rate of $f = 5/14$. Using a confidence of 25%, or $z = 0.69$, the according to (103), the upper bound on the error at the node is 0.46. If the node has N branches, we can then calculate the upper bound of the error on each branch, combine the estimates using a weighted average, and compare the result to the estimated error of the node itself. If the combined error of the branches is less than the error at the node, then we do not prune away the children. If the error of the children is greater than the estimated error of the parent, however, then we can prune away the children and replace the node with a leaf.

Pruning is an essential part of improving the performance of decisions on independent test data. Pruning can also generate much simpler subtrees, improving both the ability of people to understand the rules and the overall speed of the classifier.

Example: Building a decision tree for the 3D example data

As we calculated above, the attribute with the greatest information gain is category 3, so we pick that attribute for the first subdivision of the tree. Examining the remaining two attributes, category 2 has the greatest information gain on both branches, resulting in the divisions shown in figure 6.

Completing the tree with category 1 results in the full decision tree shown. However, note that in both cases, the two leaves of the category 1 branch result in a tie or the same value. Therefore, there is no improvement in the error rate by including the decision node and they can be pruned. The same result occurs with the category 2 branches–the error rates do not improve–so both of the category 2 nodes can be pruned, resulting in a single rule tree.

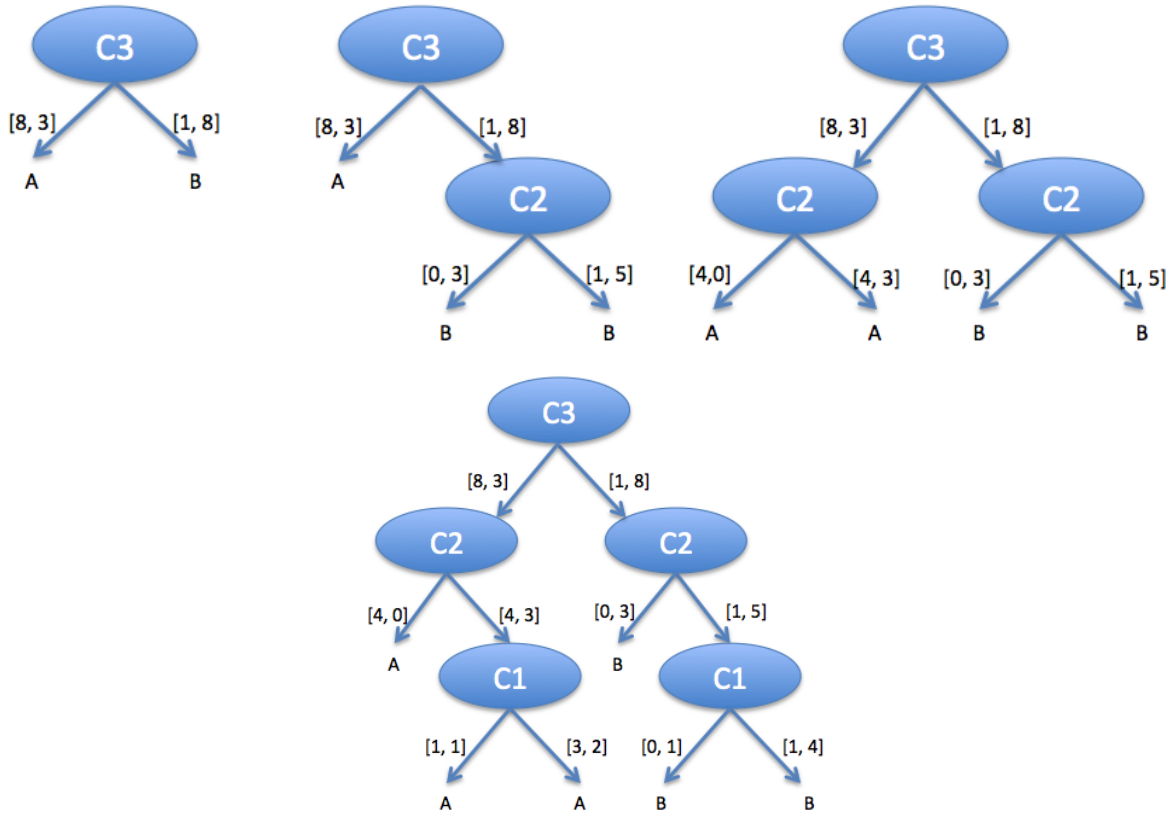

Figure 6: Building up a decision tree

2.5.3 Evaluating a Classifier

Now that we have gone through a few examples of classifiers, it's important to take a step back and figure out how we evaluate a classifier. How do we know it is working? How do we characterize the kinds of mistakes it makes?

In order to make an estimate of the quality of a classifier, we want to evaluate its performance on a test set, which should be data the classifier did not use during its training process. After running the data through the classifier, there are a number of methods of analyzing the results.

One of the most common methods of representing classifier results is as a **confusion matrix**. A confusion matrix indicates not only how many instances of output category X_i the classifier correctly labeled, but also how many instances of category X_i were labeled as each of the other classes X_j. For a 2-class problem, the confusion matrix is a 2x2 matrix. For an N class problem, the confusion matrix is NxN. For example, on a face detection task the confusion matrix may look like any of the three cases below.

Label	A	B
True A	0.60	0.40
True B	0.01	0.99

Label	A	B
True A	0.85	0.15
True B	0.13	0.87

Label	A	B
True A	0.98	0.02
True B	0.45	0.55

In the left-most case, the system finds only 60% of the true faces, but it mostly avoids incorrectly detecting a face when none exists. In the middle case, the false negatives and false positives are balanced: the system makes false positive and false negative errors at about the same rate. In the right-most case, the system finds almost all of the faces in the data set, but it also classifies many non-face examples as faces.

Almost all machine learning methods have some type of parameter that lets the developer tune the performance of the system to encourage or discourage false positives or negatives. This means there are many possible operating points. The purpose of the system defines which operating point is most appropriate.

A common method of representing classifier results to show the range of possible operating points is as a **receiver-operator curve**, or ROC curve. An ROC curve is a way of demonstrating the performance of a classifier as a function of the false positive rate. Generally, the false positive rate is shown on the X-axis, and the true positive rate is shown on the Y-axis. The **precision** of a classifier is determined by the ratio of true positive to false positive values, while the **recall** of the classifier is defined as the percent of true positive input cases correctly labeled. For a two-class problem, the precision and recall values are sufficient to describe the whole 2x2 confusion matrix.

In a recognition scenario, we always have to decide if we have seen an instance of the category. Many classifiers return a continuous number indicating confidence in a category. Comparing the confidence to a threshold, we determine if the item is in the category. As we loosen the threshold, we see more of the proper item, but we also see more false positives.

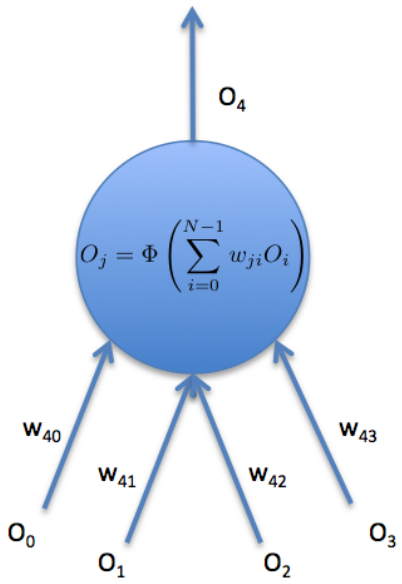

Figure 7: Artificial neural network node showing inputs, input weights, and the output.

2.5.4 Neural Networks

Neural networks are structures of interconnected nodes.

- Information flows between the nodes
- A weight is associated with each input to a node
- The output of a node is a function of its weighted inputs
- For general-purpose ANNs, the function at each node is nonlinear

A feedforward ANN is a subset of the possible ANN structures. A feedforward layer has three distinct types of nodes: input, hidden, and output.

- Information in a feedforward ANN starts from an input layer, to which an input sample is applied.
- The information then flows through one or more hidden layers
- The result of the computation is accessed in the output layer.

A feedforward ANN is like a combinational circuit in digital logic; its ouput is purely a function of its input. There are other kinds of ANNs where the output of at least one nodes feeds back into a prior node. These are called recurrent networks and function as sequential circuits with the outputs being a function of both the input and the prior output of the ANN.

The function executed by node j to generate its output O_j is generally of the form

$$O_j = \Phi\left(\sum_{i=0}^{N-1} w_{ji}O_i\right) \tag{104}$$

An important practical aspect of ANNs is that each node generally has a bias term, or constant term in the summation. In other words, if there are N-1 actual input values (O_1, O_{N-1}), then there is one other input

$O_0 = 1$ with a constant value. The value of the bias term weight w_{j0} is learned in exactly the same manner as every other weight in the network. Each node in the network has a connection, and therefore a weight, to the bias node.

The nonlinear function $\Phi(x)$ is generally some kind of squashing function that produces a value within a fixed range from an arbitrary range input. The sigmoid function, for example, is commonly used.

$$\Phi(x) = \frac{1}{1 + e^{-x}} \tag{105}$$

The sigmoid function takes values on the real number line and modifies them to be in the range $(0, 1)$.

Example

Below is a simple example of a feedforward neural network trained on the XOR function.

- Two input nodes accept the input pattern.
- Two hidden nodes are fully connected to both input nodes.
- One output node provides the result of the network

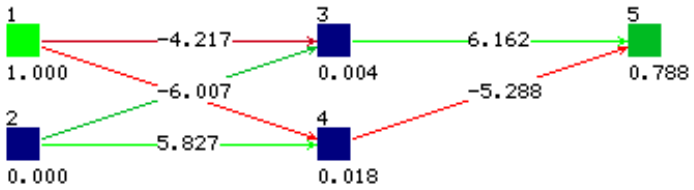

Figure 8: Artificial neural network trained to learn the XOR function

The weights in the network are as follows.

Target	Source	Weight
1	Bias	1.00
2	Bias	1.00
3	1	-4.22
3	2	3.99
3	Bias	-1.42
4	1	-6.01
4	2	5.83
4	Bias	2.00
5	3	6.16
5	4	-5.29
5	Bias	1.39

Key factors in ANN design

- Architecture complexity: big enough to learn the task, small enough to train and run

  Since the input and output layers are generally determined by the task, the complexity of the ANN is determined by the number and structure of the hidden nodes. Some networks reduce complexity (the number of weights) by not fully connecting the layers. On a fully connected network, reducing or increasing the complexity of the network means adding or deleting hidden nodes.

  An empirical method of network design is to start with a small number of hidden nodes and add nodes until overtraining is possible: when the training and test set errors begin to diverge after sufficient training. Alternatively, if the initial guess at the number of hidden nodes enables overtraining, then reduce the number of hidden nodes until the network's performance begins to degrade.

- Creating adequate training and testing sets: both quantity and coverage of likely inputs

  ANNs learn based on their training data. They are good at learning a function in areas where there is lots of training data, and they can generalize to nearby parts of the input space. Inputs that are very different from any of their training examples will produce an output, but the output is not likely to be meaningful. To guard against outputs that are meaningless, the network either needs training examples from all parts of the the input space or it needs to produce some kind of confidence measure that indicates whether it knows how to process the input.

- Building in expert knowledge

  If you know something about the structure of the problem or relationships in the input, then you can sometimes build that knowledge into the structure of the network. A hidden node in a network works similarly to a compression method like PCA. The training process forces it to learn the set of connection weights that minimize the total error of the network. In doing so, it is compressing a set of values–the inputs–to a single value.

  If you have prior knowledge that a set of inputs are strongly related, it is worth considering a network architecture where only those inputs connect to one or more hidden nodes in a pipeline separate from other inputs. The output of the specialized hidden node connects with the rest of the network at a higher layer. This architecture forces the network to learn the relationships between the limited set of inputs before learning the group's relationship to other inputs. If the prior knowledge is correct, then the network may train faster and work better than a more generic network trained on the same data.

  Given an infinite amount of data, a generic architecture can learn as well or better than a specialized architecture. However, given a specific data set, researchers have shown that a modular, specialized network will generally perform better than a generic architecture.

- Confidence measures

  Careful design of the network outputs enables the network to provide an indication as to whether its output is meaningful. For example, when dealing with a two class problem, it is possible to design the network with a single output node, using a low value to indicate one class and a high value to indicate the other. A single output, however, gives no indication of the confidence the network has in the value.

  An alternative is to design the network with two outputs, one for each class, and train it to make only one of the outputs high. In that situation, two high outputs or two low outputs indicates that the network is not well trained for that input pattern. While it is also possible for the network to give a meaningless 1/0 or 0/1 output, having the two outputs makes that less likely.

Supervised Training

While one could create an artificial neural network by hand, it is very difficult to set the weights of a network appropriately. Instead, we want to use a training method that learns the weights from a training set consisting of labeled examples.

- Training set: a set of labeled data containing examples of all classes the network needs to see.
- Testing set: a similarly labeled data set on which the network never trains, but which is used to evaluate the network's performance.

The most common training method for feedforward neural networks is called backpropagation. It is a gradient descent algorithm that modifies the weights to improve the performance of the network on the training set. Gradient descent algorithms are greedy iterative algorithms that take one step towards the solution each iteration, making the change that maximally minimizes an error metric.

The process of training using a single pattern is as follows:

- Apply the training pattern to the network
- Calculate the output of the network and compare it to the desired output
- Calculate the change to the weights for the output layer to make the actual and desired outputs closer.
- Move back through the network, calculating how to change the weights at each layer.

Overall, a common version of backpropagation is:

- Apply all of the training patterns as above and store the sum of the weight changes
- Apply the weight changes to the network, modified by a learning rate
- Repeat thousands of times

Applying all of the training patterns to the network is called an **epoch**. Training can often take hundreds or thousands of epochs.

You can visualize the error surface of the network as a set of valleys, hills, and plateaus. Each point on the error surface represents a set of values for the network weights. The best network, given the training and testing data, is the one at the bottom of the deepest valley, the lowest point on the error surface.

Backpropagation training generally begins with the weights of the network set to small, non-zero random values. Imagine the network starting at some randomly selected point on the error surface. By calculating the impact on the error rate of changing each weight in a specific manner, the backpropagation algorithm can determine which changes correspond to the optimal step down the error surface. By iterating that process, the network moves down the error surface towards the nearest valley.

Problems can arise with any greedy gradient-descent algorithm when the error surface is bumpy or has lots of valleys other than the deepest one. Since the network moves towards the closest valley, where the network starts can determine whether it can learn the problem optimally. The situation is similar to K-means clustering, where the initial cluster means end up determining how the data set is divided.

As with K-means clustering, one method of finding the global optimum is to train the network multiple times, starting it in different locations on the error surface each time. Other methods modify the backpropagation algorithm by introducing terms into the optimization that allow it to skip over or bounce out of local minima (valleys).

Algorithm for a single training pattern

1. Execute a forward pass on a training pattern to calculate the output of the network O.
2. Calculate the mean squared error between the output O and the desired output T.

$$E = \frac{1}{N}\sum_{i=1}^{N}(O_i - T_i)^2 \tag{106}$$

3. Calculate the weight changes required to move O closer to T
   - Weight changes are calculated using the **generalized delta rule**

$$\Delta w_{ji} = \eta \delta_j o_i \tag{107}$$

     - w_{ji} is the the weight going from node i to node j
     - Δw_{ji} is the change that should be made to w_{ji}
     - η is the learning constant, and governs the magnitude of the weight changes
     - δ_j relates the weight to its influence on the output
     - o_i is the output of node i
   - The weight change ought to be proportional to the amount of change in the overall error realized by changing the weight. In other words, we need to know the derivative of the error with respect to changes in the weight.

$$\Delta w_{ji} \propto -\frac{\partial E}{\partial w_{ji}} \tag{108}$$

   - To figure out this derivative, we need to have a complete expression connecting the inputs and outputs of a node. The overall input to a node is a weighted sum of the outputs of all connected nodes.

$$I_j = \sum_i w_{ji} o_i \tag{109}$$

   - Given the expression for the overall input to a node, the output of a node is simply

$$o_j = \Phi(I_j) = \frac{1}{1 + e^{-I}} \tag{110}$$

   - The derivative of the sigmoid function is simple to calculate.

$$\Phi'(x) = \Phi(x)(1 - \Phi(x)) \tag{111}$$

   - Now we can divide the derivative from above into two parts, reflecting the change in the error with respect to the input of a node and the change in the input to a node with respect to a single weight.

$$-\frac{\partial E}{\partial w_{ji}} = -\frac{\partial E}{\partial I_j}\frac{\partial I_j}{w_{ji}} \tag{112}$$

- The second term in (112) is the derivative of the input equation (109)

$$\frac{\partial I_j}{\partial w_{ji}} = \frac{\partial}{\partial w_{ji}} \sum_i w_{ji} o_i = o_i \tag{113}$$

  and explains the o_i in the generalized delta rule.

- The δ term in the delta rule corresponds to the remaining partial derivative, which we can again expand.

$$-\frac{\partial E}{\partial I_j} = -\frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial I_j} = -\frac{\partial E}{\partial o_j}\Phi'(I_j) \tag{114}$$

- The second term of (114) is given by $\Phi'(I_j)$. The first term is different for output units and hidden units.

- For an output node, the differential of the error term with respect to its output is given by (115).

$$\frac{\partial E}{\partial o_j} = \frac{\partial}{\partial o_j}\frac{1}{N}\sum_{i=1}^{N}(O_i - T_i)^2 = -(t_j - o_j) \tag{115}$$

  In other words, the weight change for a link coming into an output node depends upon how different the output is from the target.

  Combining (115) and (114), the delta value for an output node is given by

$$\delta_j = (t_j - o_j)\Phi'(I_j) \tag{116}$$

- Computing the delta value for a hidden node is more challenging, because it must take into account all of the ways in which a single weight can affect the different between the network outputs and the target. If a hidden node connects to multiple output nodes, then changing a weight of a connection coming into a hidden node changes all of the output node values.

  If there are K output nodes, we can use the chain rule to identify the relationships.

$$\frac{\partial E}{\partial o_j} = \sum_{k=1}^{K}\frac{\partial E}{\partial I_k}\frac{\partial}{\partial o_j}\sum_{i=1}^{N} w_{ki} o_j = \sum_{k=1}^{K}\frac{\partial E}{\partial I_k} w_{kj} = -\sum_{k=1}^{K}\delta_k w_{kj} \tag{117}$$

  The way to interpret (117) is that the δ for a weight going into the hidden node is the sum of the δ values of all the connections leaving the node multiplied by their respective weights.

- The generalized delta rule for a hidden layer is, therefore, as follows.

$$\delta_j = \Phi'(I_j)\sum_{k=1}^{K}\delta_k w_{kj} \tag{118}$$

- The update for a weight connected to an output node is given in (119).

$$\Delta w_{ji} = \eta(t_j - o_j)\Phi'(I_j) o_i \tag{119}$$

- The update for a weight connected to a hidden node is given in (120).

$$\Delta w_{ji} = \eta \left(\Phi'(I_j) \sum_{k=1}^{K} \delta_k w_{kj} \right) o_i \tag{120}$$

Intuitively, the update for a weight w_{ij} connecting node i to hidden node j is proportional to o_i modified by the impact of any change in the weight on all of the K output nodes it affects.

One thing to note is that in both cases if $o_i = 0$ then there will be no weight adjustment. Therefore, that suggests avoiding training examples where the inputs are zero. For example, the network learns much better when the training samples for the XOR problem avoid exact zero as an input, instead using numbers close to zero.

Likewise, trying to train a network to achieve a true 0.0 or a true 1.0 is difficult, and often leads to overtraining because it pushes the outputs far into the saturation zone of the sigmoid function. Instead, having training examples where the maximum desired output is 0.9, for example, rather than 1.0, tends to result in faster training and a more generalized network.

4. After evaluating the adjustments for each weight in the network, apply the adjustments. When training with multiple patterns, sometimes the preferred method is to hold off on modifying the network until the adjustments for every training example have been calculated. Once the system has applied the entire training set, it adjusts the weights by using the mean or median value suggested by the training examples.

Variations on Backpropagation

Methods of improving training:

- More training data, with more coverage of the range of possible inputs.
- Update training and testing sets during training to emphasize problem cases.
- Boltzmann training: similar to simulated annealing. Add a random offset to the weights that decreases over time as training progresses.

$$\Delta W_{ji} = \Delta w_{ji} + G(0, \sigma(t)) \tag{121}$$

- Momentum: add a fraction of the weight change in the prior step to the current step

$$\Delta W_{ji}^t = w_{ji}^t + \beta w_{ji}^{t-1} \tag{122}$$

Some networks use alternative squashing function, such as the hyperbolic tangent. The hyperbolic tangent looks like the sigmoid function, but it its range is $(-1, 1)$. Sometimes using the hyperbolic tangent can improve training times.

$$\text{atanh}(bv) = a \left[\frac{1 - e^{-bv}}{1 + e^{-bv}} \right] = \frac{2a}{1 + e^{-bv}} - a \tag{123}$$

ANN Case Study: ALVINN

ALVINN was one of the first successful ANN systems used to control a robot. Developed by Dean Pomerleau at Carnegie Mellon University, ALVINN was trained to choose a steering direction given a single greyscale image taken from a camera mounted below the rearview mirror. The system scaled the input image to 30x32, and each greyscale value generated one input to the network.

The network architecture was a 3-level fully connected feed-forward network. The input layer used a 30x32 grid of nodes, the hidden layer used 4 hidden units, and the output layer consisted of 30 output units. The network contained approximately 4000 weights.

The output layer design was intended to provide a measure of confidence in the steering direction. Each training example consisted of an input image and a desired steering direction. The network was trained to produce a Gaussian distribution of a certain standard deviation, centered on the appropriate steering direction. This form of output enables estimation of the confidence of the network's answer by fitting a Gaussian to the output values and examining the standard deviation and the error of the fit. If, for example, the network is producing mostly small values or mostly large values on the output nodes, then it is clear that the network is not representing the desired function appropriately in that part of the input space.

As part of the network design process, Pomerleau also experimented with having the network reproduce its input at the output. In addition to the steering direction output, he also trained the network to reproduce the input image on a 30x32 output grid. For parts of the input space where the network was well trained, the network was able to effectively recreate the input. Areas of the input space not represented in the training data did not produce similar outputs.

The idea of testing the error between the input and the recreated input is also used in PCA compression algorithms. For example, we can build a PCA space, or eigenspace for images of faces and represent each face image with just a few values (e.g. 10-12 numbers). If we use the compressed values to recreate the original image we will get a picture of a face that looks similar to the original. If we apply the compression process to a picture of a flower, however, then when we recreate the original image we will still get something that looks like a face. Since the recreated output does not look like the input, that means that the flower image is outside of the input space used in the training set.

After training, ALVINN was able to steer a car 98% of the way from coast to coast.

What is an ANN learning?

- In what is probably the classic pattern recognition fiasco, the army developed a tank classifier that was shown examples of Soviet tanks and US tanks from many different angles and distances. Their classifier worked extremely well on the training set. When it was taken into the field, however, it failed completely. It turned out the the Soviet tank pictures were all taken on a sunny day, and all of the US tank pictures were taken on a cloudy day. The network learned to predict the weather conditions.
- When developing ALVINN, Pomerleau studied what factors the hidden nodes were emphasizing by looking at which weights were most active. They found some surprising things. For example, when the network was trained on a stretch of road with guard rails, the network put heavy weights on the inputs from pixels corresponding to the guard rail locations and small weights elsewhere in the image. To avoid the network performing poorly when the guard rails ended, he modified the input images so that parts of the image above and to the side of the road were effectively blocked out or reduced in importance in the final network.

Summary

- An ANN is capable of learning arbitrary classification functions if trained on enough data
- An ANN can often provide good performance and generalize to previously unseen data
- Most ANN configurations are straightforward to train using backpropagation or a variant of it
- The structure of an ANN affects its performance and the speed with which it trains
- The inputs to an ANN need to be pre-processed and transformed into forms it can use effectively
- It can be difficult to identify what factors the ANN is using to solve the problem
- Real systems often make use of multiple ANNs since training is a stochastic process

2.5.5 Nearest Neighbor Classification

So far, we have examined three different forms of classification.

- Bayesian classifiers: combine prior probabilities and evidence to select the most probable class.
- Decision trees: subdivide the world based on features until each individual subdivision is a single class.
- Neural networks: approximate the classification task as a function and learn the function

Nearest neighbor classification takes a fourth approach: example-based classification. The main idea is to build a library of example cases and then classify new cases based on which labeled case is most similar. In some cases, a new example may be labeled as 'none of the above' if the distance between it and any of the example cases is too big.

Nearest neighbor classification is related to clustering, and clustering is one approach to identifying the library of example cases. The set of cluster means are an example-based representation of a set of data, and classifying a new feature vector by comparing it to the cluster means divides the data space into contiguous partitions.

Nearest neighbor classification is not, however, limited to using a single sample to represent each class. Given a set of labeled training data, a classifier can store and use the entire training set as examples. Using multiple examples for each class allows for disconnected regions and arbitrary decision boundaries, given enough data.

As with any classifier, there is a tradeoff between specificity and generality. There is also a tradeoff between the complexity of the decision boundaries and speed. Clearly, using a single example of each class in the classifier is the fastest possible case, but it also produces the simplest decision boundaries and does not permit discontiguous regions. Using all of the labeled examples of a class has the potential to produce the most complex decision boundaries, but is the most computationally expensive.

In addition to how many samples per class, the other free parameter of a nearest neighbor classifier is the distance metric. The only constraints on a distance metric are that it be non-negative and obey the triangle inequality. As posited by the Ugly Duckling theorem, each particular problem will have an ideal distance metric, and no single metric will be equally useful in all problems.

Two common variations on nearest-neighbor classifiers are **K-Nearest Neighbor Classifiers** and **Support Vector Machines**. A K-nearest neighbor classifier calculates, for each class, the distance between the new feature vector and the K closest examples. The classification decision is based on the sum of the distances to the K closest examples in each class. K-nearest neighbors can produce more complex boundaries with smaller numbers of examples per class than using only the closest neighbor.

The design of a support vector machine [SVM] is based on two observations: the boundary cases of each class are the only ones that really matter in classification, and the decision boundary is probably simple given the right distance metric.

Conceptually, an SVM tries to identify and keep only the boundary cases from the training data for each class. Then it uses a mathematical process to evaluate a set of possible distance metrics and identify the mapping that results in the simplest boundary. Clearly, the two concepts are linked: the actual location of the boundary is defined by the distance metric. SVMs have proven to be a successful classification method. While they are mathematically more complex than simple nearest neighbor or KNN classification, the concept is the same: find examples of each class and use them to classify new cases.

2.6 Numerical Prediction

In the prior section we looked at methods capable of estimating a discrete output class. Predicting numeric values is somewhat more complicated, but possible using similar techniques. ANNs, for example, are capable of learning numeric functions as easily–or more easily–than discrete output categories. Other methods, such as decision trees or K-Nearest Neighbors, require modifications to both the classification process and the building/training process.

2.6.1 Linear Regression

Basic linear regression is fitting a constant, line, plane, or hyper-plane to the data. Linear regression minimizes the error in the dependent variable relative to the predicted value given the line or plane fit. Note that this is different from finding the best fit model to the data set, which involves finding the line or plane that minimizes the distance of each point in the data space from the model.

The easiest way to visualize the difference is to consider linear regression with one variable predicting a second using a line fit. Linear regression minimizes the vertical distance (dependent variable error) between the estimated line and the data points. When the dependent variable is very sensitive to the independent variable (big slopes) this can result in unexpected line fits because of noise.

Finding the best fit line to the set of (x, y) points, however, minimizes the perpendicular distance between each data point and the line model. The latter requires solving a set of–generally overconstrained–linear equations. Note that the two problems are fundamentally different, although the amount and type of data is identical.

In the case of regression, one or more variables $(\vec{x})$ are considered independent, while the dependent variable (y) is modeled as a function of $\vec{x}$. Regression identifies the set of parameters $\vec{p_r}$ for a model of the form $y = f(\vec{p_r}, \vec{x})$.

Finding the best model for a set of points $\{\vec{x_i}\}$, however, is the problem of finding the set of parameters for a function $\vec{x} = f(\vec{p})$ that minimizes the error between the model and the data points.

Note that the linear in linear regression refers to the parameters that relate the independent variables to the dependent variables. A polynomial regression model, described as $y = \beta_0 + \beta_1 x + \beta_2 x^2$, is a perfectly fine candidate for linear regression as the dependent variable y is a linear function of the parameters $\{\beta_i\}$

The general multi-dimensional equation to solve for linear regression is given by (124).

$$X^t y = (X^t X)\vec{\beta} \tag{124}$$

X is the data matrix, which is $N \times M$ where N is the number of training samples and $M - 1$ is the number of features used for prediction. The first column of X is all 1's, which represents the constant coefficient for β_0. The value of each element of the square matrix $(X^t X)$ is given by (125).

$$x_{ij} = \sum_{k=0}^{N-1} x_{ki} x_{kj} \tag{125}$$

The left side of (124) is given by (126).

$$z_i = \sum_{k=0}^{N-1} x_{ki} y_k \tag{126}$$

Solving the resulting linear matrix equation for the parameters provides the least squares estimates for the $\{\beta_i\}$. Singular value decomposition [SVD] is a good choice for doing the estimation because it will correctly handle singular matrices. In particular, if two of the 'independent' variables are, in fact, completely dependent, then SVD will give a singular value close to or equal to 0 for the NULL space–the direction or space of variation that has no impact on the solution–and return the least squares estimate for the remaining independent directions.

Linear regression can make use of categorical data by treating each category as a single binary variable that has the value one or zero. Likewise, linear regression can classify feature vectors into binary categories by training it to match a function that outputs a 0 for one category and 1 for the other.

2.6.2 Model and Regression Trees

- Put a regression equation at each leaf node
- Requires building the tree to minimize variation in the dependent attribute
- Instead of splitting on information gain, we split on standard deviation reduction. The standard deviation of each subtree is weighted by the number of training examples $|T_i|$ passed on to the subtree.

$$\text{SDR} = sd(T) - \sum_i \frac{|T_i|}{|T|} sd(T_i) \tag{127}$$

- The splitting process terminates when the standard deviation of the examples in the leaf node is small compared to the original data set (e.g. $<5\%$)
- Each leaf uses a regression model based on the attributes not used along the path to the root.
- The pruning process is similar to a standard decision tree.

One problem with regression trees is that two data points that are extremely close in the data space could use radically different linear regression models. For example, consider a tree with one node that splits an attribute on the threshold value X. Two data points with values $X - \epsilon$ and $X + \epsilon$ can be very close in the data space. Because they go to different leaves, however, the regression tree will use different regression models to predict their values. This can result in discontinuities in the prediction space between subtrees.

The solution to the problem of discontinuities is to smooth the result. Instead of storing regressions at just the leaves, the regression tree can store a regression model at each node. The actual predicted value is then a weighted combination of the predictions along the path from the leaf to the root. At each step up from child to parent, the new predicted value p' is a weighted sum of the child's prediction value p and the parent's prediction value q.

$$p' = \frac{np + kq}{n + k} \tag{128}$$

The number of training examples that reached the child node is n, and k is a smoothing constant. The relative values of n and k determines the degree of smoothing. Researchers have shown that smoothing can significantly improve prediction performance.

In machine learning terminology, a model tree uses a regression equation at each leaf. By contrast, a regression tree has a single constant value at each leaf. Both types of trees make use of smoothing to avoid discontinuities in the learned function.

Enumerated types cause problems for linear regression because they must be converted to numeric values. One approach to conversion is to calculate the average predicted value for each of the k values in the enumerated variable. Then sort the k averages and select a position in the list on which to divide the enumerated variable into a binary variable, which can then be treated as a numeric value for regression.

2.6.3 Locally Weighted Linear Regression

The basic idea of a regression tree is to divide the data space into small pieces that are easier to represent as a linear function. Any nonlinear function, if divided into sufficiently small pieces, can be represented as a set of linear hyperplanes: all curves are locally linear.

Instance-based learning can make use of the same concept to locally represent the data space as a set of hyperplanes. Consider a set of N training instances distributed around the data space. For an unknown pattern, we can find the closest K training instances and fit a plane to them, weighting each point by its inverse distance from the unknown pattern. The fit plane then provides a prediction for the dependent variable for the unknown pattern.

Except for choosing the training data, locally weighted linear regression does all computation at prediction time, dynamically selected which training points to use for a given new pattern.

2.6.4 Generalizing Local Weighting

It is possible to improve the performance of many learning methods using local weighting. Naïve Bayes, for example, tends to perform much better when the prediction is based on locally weighted training examples. As with locally weighted linear regression, the predictor function is generated at prediction time, not during any training phase.

1. Given: a pattern P to be evaluated
2. Identify all training examples within a certain distance of P and generate a training set where each example is associated with a weight.

$$T_P = \{(t_0, w_0), (t_1, w_1), \ldots (t_{N-1}, w_{N-1})\} \tag{129}$$

3. Build the classifier or predictor using the weighted training set. For example, when computing probabilities for Naïve Bayes, each training example counts as a fraction rather than an integer value.
4. Use the trained classifier or predictor to generate the output class or value

Local weighting, which divides up the input space into smaller sections, enables simple classifiers and predictors to perform better because the output function is more likely to be smooth. While the system still needs sufficient training data, the simpler form of the output function means that much less training data is required to learn the function.

Note that the radius of influence of the training samples may vary depending upon the local density of training samples. In areas of low density, it may be necessary to increase the radius of influence around the unknown pattern in order to obtain enough training examples to generate the classifier or predictor.

2.6.5 Data structures for Nearest Neighbor Search

Simple nearest-neighbor search through a large data set is $O(N)$ in the number of data points. For large data sets this can lead to unacceptable performance. There are several approaches to making nearest neighbor search faster. One is to grid the data into regular bins, which reduces the number of points that need to be searched to a subset of the entire data set: the bin in which the data point resides and possibly neighboring bins. In some cases, this is the best option, as discussed below. For many real data sets, however, we can build a tree structure that permits nearest-neighbor searches in $O(\log N)$ time, which is generally much faster than even gridded data.

K-D Tree

For a single dimension (values on a number line), the most efficient data structure for identifying nearest neighbors is a binary tree, which leads to $O(\log N)$ time in the worst case for a balanced tree. For data with multiple dimensions, we can build a K-D tree. A K-D tree is also a binary tree, but each node in the tree uses a different dimension of the data, and the dimensions rotate with depth.

Each node in a K-D tree is a data point in the set. The value of one dimension of the data set determines how the rest of the data is split into the two child trees. The root node uses dimension 0, its children use dimension 1, and so on until the dimension wraps back to 0. When building a tree, we can use one of two methods.

- Randomly select a data point and use its value to divide the data.
- Select the data point with the median value in the splitting dimension.

Both techniques have their pros and cons. Randomly selecting the node points is fast, but may lead to unbalanced trees. Selecting the median takes more time, but will produce more balanced trees.

Once the tree is built, identifying the nearest neighbor is straightforward.

Nearest neighbor search:

- Search down to the leaf that best approximates the sample. The total distance between the leaf and the sample defines a radius within which the nearest neighbor must exist.
- Walk back up the tree, at each node Q testing how close it is to the sample and whether the other child needs to be searched. If the total distance between Q and the sample is smaller than the smallest distance seen so far, then save Q as the potential nearest neighbor and tighten the search radius. We can prune away the other child node only if the distance in the single splitting dimension between Q and the sample is larger than the distance to the closest point found so far.

To implement K-nearest neighbor search, on the walk back up the tree the algorithm just needs to keep track of the closest K points. The system searches a child branch only if the distance in the splitting dimension between the parent and the Kth best point found so far is smaller than the total distance between the sample point and the Kth best point.

2.6.6 Robust Regression

Linear regression makes use of all of the data in the training set to estimate the best fit hyper-plane to the data. Unfortunately, because it minimizes squared error over the training set, just a few outliers can cause the resulting fit to move away from what might be called the consensus solution. While minimizing sum-squared error is the optimal method of handling Gaussian errors, it is not optimal for errors that fit other distributions.

Robust regression is the descriptor for a set of methods that fit models to data in ways that attempt to ignore outliers. For data that does not have a Gaussian error distribution, robust regression can often produce fits that are more general and accurate. Two robust regression techniques are **Least Median of Squares** and **RANSAC**. Least Median of Squares is a more specific variant of RANSAC.

The basic idea with Least Median of Squares is to find the regression model that minimizes the median squared error for the data set. The process is not deterministic, but stochastic. In general, it is not guaranteed to find the best fit model given the criterion, but it generally does a good job of finding an adequate model.

Least Median of Squares Algorithm

1. Given: data set with N samples $\{s_i\}$ and M-1 independent dimensions
2. $minMedianError \leftarrow \infty$
3. For K iterations, or until $minMedianError < \epsilon$
   (a) Randomly pick M points from the N samples
   (b) Fit a hyperplane model $R(s)$ to the M points
   (c) Calculate the sum-squared error for each data point $e_i = (o_i - M(s_i))^2$
   (d) Sort the points and identify the median error e_m
   (e) if $e_m < minMedianError$ then update $minMedianError$ and store $R(s)$
4. Return $minMedianError$ and the corresponding model $R(s)$

Least Median of Squares is commonly used to fit regression lines to data (M = 2). For data with Gaussian errors, the difference between LMS and regular linear regression is small. For data with non-Gaussian errors, LMS can often produce a result that makes more sense in the context of the data as it is not affected by outliers, unless they make up for than 50% of the data. The number of iterations used to build the LMS model determines the likelihood that the process will find a good model. The higher the dimensionality of the model (M) and the more outliers exist in the data set, the more iterations are required to identify a good solution. One way to think about it is that to build a good model the algorithm must select M non-outlier points randomly. The larger M is, and the more outliers exist in the data set, the harder it is to pick M non-outlier points.

RANSAC, which stands for Random Sampling and Consensus, is a more general version of Least Median of Squares. It is commonly used in situations where there are non-Gaussian outliers in the data set, and it is able to produce reasonable results even when the percentage of outliers is potentially larger than 50%. The algorithm is almost identical to LMS, except that instead of looking at the median error, RANSAC uses an error threshold to count the number of inliers for a model.

For RANSAC, the key parameters are:

- An estimate of the percentage of inliers w
- The number of data points required to estimate a model n
- The desired probability that the algorithm will find a good solution p
- The number of iterations to run the process k

The probability of randomly picking an inlier point is given by w, so the probability of picking all inliers is w^n, and the probability of picking at least one outlier is $(1 - w^n)$. The probability of executing k iterations and never picking a good model is $(1 - w^n)^k$. Therefore, if we want to pick a good model with probability p, we can identify k as follows.

$$\begin{aligned} \text{probability of picking all bad models} &= 1 \text{ minus probability of picking a good model} \\ (1 - w^n)^k &= (1 - p) \\ k \log(1 - w^n) &= \log(1 - p) \\ k &= \frac{\log(1 - p)}{\log(1 - w^n)} \end{aligned} \tag{130}$$

The full algorithm is given below. Note that, unlike the LMS process, the last step in RANSAC is to use all of the inliers to calculate a final model in a least-squares sense. The assumption is that errors within the inlier group are likely to be Gaussian, in which case a least-squares fit model for the inliers is the optimal model given the training data.

RANSAC

1. Given: data set with N samples $\{s_i\}$ and a model $R(s)$ with n degrees of freedom.
2. $maxInliers \leftarrow 0$
3. For k iterations, or until $maxInliers > d$
   (a) Randomly pick n points from the N samples
   (b) Fit a model R_k to the n points
   (c) Calculate the how many points C are close enough to the model
   (d) if $C < maxInliers$ then update $maxInliers$ and store R_k in R_C
4. Using all of the inliers for the best model, calculate the optimal model R_o in a least squares sense.
5. Return $maxInliers$ and the corresponding model R_o

2.7 Meta-Learning

2.7.1 Bagging

Bagging: training many classifiers on randomly sampled subsets of the data, then using majority vote or averaging numerical estimates.

Bias-variance decomposition: errors come from several causes: training data containing noise, inability of a classifier to learn the problem, and the fact that training data is finite. We can't do much about data containing noise, or necessarily the capability of the classifier (bias errors), but we may be able to do something about selecting the training data used by the classifier.

Bagging tries to reduce the variance in classifier performance by randomly selecting samples of training data and building many classifiers. As each classifier sees a different subset of the training data, each will perform slightly differently. As a whole, the expectation is that performance will improve, and generally will not decrease. Decisions for bagging are made using simple majority voting.

One improvement to Bagging is to make use of confidence probabilities to weight votes, rather than using binary voting. Decision trees, for example, can provide confidence estimates in their outputs by simply returning the percentage of training examples at a leaf that represent the output class. A leaf with close to uniform training examples would have a high confidence, while a leaf with almost even numbers of each output class in the training set would have a low confidence.

With the Kinect example, each leaf stored the distribution of categories from the training data. To find the overall class, they combined the distributions from from three trees and used that to classify the input.

Bagging works best on classifiers that are sensitive to the training data. Nearest-neighbor classifiers, for example, are pretty stable. Decision trees, however, can be quite unstable. In fact, bagging sometimes works better using decision trees that are not pruned, as the selection of the last few leaves is quite sensitive to the specific training data.

2.7.2 Randomization

Randomization: Insert randomization into the process of building classifiers, then build lots of them and let them vote.

Randomization works better on classifiers that are more stable with respect to training data, but can also work well with unstable classifiers. The kind of randomization inserted into the training process for the classifier can be small (e.g. initial weights on ANNs) or large (e.g. building random decision trees). Decision tree forests can be built using randomization in different ways: limiting which attributes are available or building trees where the nodes are selected randomly instead of using information gain.

How could we insert randomization into the following methods?

- Decision trees: randomly select features, depth of the tree, decision points, pruning
- Neural networks: initialization, network structure
- Naïve Bayes: training set selection
- Linear Regression: training set selection

2.7.3 Boosting

Boosting: training many classifiers using weights on the training data errors to emphasize data on which the prior classifiers fared poorly, then using weighted majority voting to make an ensemble decision.

Boosting Learning Algorithm (Witten & Frank)

1. Assign equal weights $\{w_i\}$ to each training instance
2. For t iterations
   (a) Normalize the weights so $\sum w_i = 1$
   (b) Learn a model M_i using the weighted data
   (c) Compute the error rate of the model e_i
   (d) If $e_i = 0.0$ or $e_i \geq 0.5$ then terminate model generation
   (e) For each example in the data set
      - If example x_i was classified correctly, multiply w_i by $e/(1-e)$

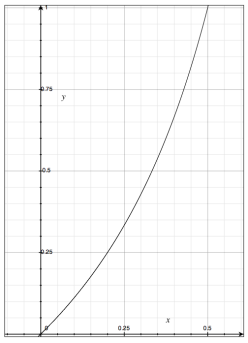

Figure 9: $w = e/(1-e)$.

Boosting Classification Algorithm

1. Assign weights of 0 to all output classes o_t
2. For each model M_i
   (a) Add $\Delta = -\log(e/(1-e))$ to the weight of the output class predicted by M_i
3. Return the output class with the highest weight

Note that boosting does not depend as much on classifiers that are high quality because each classifier's final vote is weighted by its performance on the training set. Any classifier that achieves a classification rate better than chance (e.g. 50% in a two-class problem) may be useful. Boosting also focuses specifically on adding new classifiers that handle cases not handled properly by the existing ensemble. Therefore, it is often useful to set up the classifiers so they are different from one another: training the same kind of classifier on data that is weighted only slightly differently does not necessarily generate a very different classifier. Classifiers like decision stumps, random forests, or intentionally simplified neural networks are more likely to be sensitive to the training data and produce a variety of classifiers for the ensemble.

2.7.4 Cascade Classifiers

For many problems, output categories occur at similar rates, in which case the classifier is trying to differentiate between cases that are about equally likely. For some problems, however, one output category may occur at a very small rate compared to the other. Visual object detection, such as detecting faces in images, is an example of one such task. In a picture of a person standing on the beach, there is one box of pixels (perhaps a dozen with small shifts and scales) that contains the face. There are millions of other boxes in the image that do not contain a face. If we gave a standard classifier a set of randomly sampled boxes from the image with their output labels (face or no face), then a classifier that always guessed no face would have a 99.99% performance rate, or better. It would also be completely useless.

One approach to detecting unlikely events is to build a classifier that uses many simple features to carve away parts of the input space where faces don't exist. Each feature does not have to be good at detecting faces, it just has to be good at not discarding inputs of actual faces. If we put enough simple features in a cascade, or linear tree, and each one is good at not discarding faces, then a sufficient number of simple classifiers can perform as well or better than a single complex feature. In addition, since many instances can be rejected using a few simple features, the overall detection process may be faster than having a single complex analysis function.

One of the keys to using lots and lots of simple classifiers in a cascade network is that each one has to be set up so that it almost never rejects real instances of the object. The false positive rate can be high, but the false negative rate must be close to zero. All of the real instances must make it all the way through the chain. If some of the false positives are discarded at each step, then the number of false positives at the end of the chain ought to be low. The other key factor is that all of the simple features need to be different, in the sense that they are rejecting different parts of the input space.

As an example of simple features, consider boxes of various sizes that represent area sums. A simple feature is the difference of the area sums between adjacent boxes of different sizes and configurations. It turns out that such simple features can be computed very quickly using the **integral image**.

Viola and Jones used simple box features of varying sizes in configurations that included: side-by-side, up-down, center subtract, and 2x2 checkerboard. The base resolution of the detector was 24x24, so the set of all box features of any size or location that fit within a 24x24 box was approximately 180,000. The challenge then becomes, how do you pick which features of the 180,000 should be strung together to form the cascade classifier?

Viola-Jones Adaboost Variation

1. Givens:
   - Example images $(x_1, y_1, w_{t,1}), \ldots, (x_n, y_n, w_{t,n})$ where y_i is 0 for negative examples and 1 for positive examples and w_i is the weight of the training example at step t during the training process.
   - A feature set with a classifier h_j associated with each feature j.
2. Initialize training example weights $w_{1,i} = \frac{1}{2m}$ for the m negative examples and $w_{1,i} = \frac{1}{2l}$ for the l positive examples.
3. For $t = 1, \ldots, T$:
   (a) Normalize the weights so they sum to one.
   (b) For each classifier h_j, train it to minimize the weighted performance on the training set.
   $$e_j = \textstyle\sum_i w_i \left| h_j(x_i) - y_i \right|$$
   (c) Let h_t be the classifier with the lowest error e_t.
   (d) Update the weights on the training set:
   $$w_{t+1,i} = w_{t,i} \left(\frac{e_t}{1-e_t} \right)^q$$
   where $q = 0$ if example x_i was correctly classified by h_t and $q = 1$ otherwise.
4. The final strong classifier H uses a weighted vote of the ensemble of weak classifiers.
   $$H(x) = \begin{cases} 1 & \sum_{h_j \in H} \alpha_t h_j(x) \geq \frac{1}{2} \sum_{t=1}^{T} \alpha_t \\ 0 & \text{otherwise} \end{cases}$$
   where $\alpha_t = \log \frac{1-e_t}{e_t}$.

The algorithm above generates a single classifier that combines a number of weak classifiers to generate a much stronger classifier. The single classifier is a set of individual features/tests represented by the h_j whose votes are weighted. Classifiers with better performance get weighted more heavily. The collection of features is often called an ensemble.

To obtain a cascade, Viola-Jones used a sequence of these ensemble classifiers until they achieved a sufficient false positive rate. Any single ensemble does not have a sufficiently small false positive rate to work by itself.

Cascade Classifier Variation: Wu and Rehg

1. Givens: a training set with positive and negative examples, a minimum detection rate d (e.g. 0.999), and a maximum false positive rate f (e.g. 1×10^{-7}).
2. For each feature j, train a weak classifier h_j with a false positive rate of f.
3. Initialize an ensemble classifier $H \leftarrow$, step $t \leftarrow 0$, current detection rate $d_t = 0.0$, and current false positive rate $f_t = 1.0$.
4. while $d_t < d$ or $f_t > f$
   - if $d_t < d$, then find the feature k with classifier h_k such that, by adding it to H, the detection rate of the new ensemble of classifiers d_{t+1} is maximized.
   - else, find the feature k with classifier h_k such that, by adding to H, the false positive rate of the new ensemble of classifiers f_{t+1} is minimized.
   - $t \leftarrow t + 1$, $H \leftarrow H \cup \{h_k\}$.
5. The decision of the ensemble classifier is formed by a majority voting of weak classifiers in H.

$$H(x) = \begin{cases} 1 & \sum_{h_j \in H} h_j(x) \geq \theta \\ 0 & \text{otherwise} \end{cases}$$

   The threshold θ is initially set to half the number of classifiers in the ensemble, but may be decreased if necessary to achieve a different balance between d_T and f_T.

Note that all of these methods are simply classifier methods that work on a feature vector. All of the above examples use images as the basis for the feature vectors, but they also apply equally well to gesture and action recognition, speech recognition (where HMMs are one of the most successful methods), or any other situation where a classifier may be useful.

2.7.5 Sampling from a non-uniform distribution with replacement

If the learning method is able to use weights on training samples, then the system can be used directly in boosting. Sometimes, however, it is difficult to weight training samples given the learning algorithm. In that situation, it is useful to be able to generate a training set by drawing samples from a distribution, where each sample has a different probability of being selected.

The simplest algorithm for sampling a weighted distribution is to normalize the sum of the weights to one, draw a random number α from the range [0, 1], and then find the sample k where the sum of the weights from 1 to k reaches α. That algorithm is given below.

1. Weight each sample in the source population
2. Normalize the weights so the sum of the weights is 1
3. Iterate N times
   (a) Generate a random number β in the range [0, 1]

(b) Starting with the first sample in the source population, sum the weights until the sum of the weights of the first Q source samples is larger than or equal to β.

$$\sum_{i=0}^{Q} w_i \geq \beta \tag{131}$$

(c) Put sample Q into the new population

4. Return the new population

If you visualize the source population as a set of different sized blocks, you can visualize this version as picking a height, and then stacking the blocks in order until some block reaches or exceeds the target height and adding a copy of that block to the new population. Each time you pick a sample for the new population you have to build a new tower. Unfortunately, the simple algorithm has time complexity $O(N^2)$, where N is the size of the source population–the population from which samples are drawn. This can be prohibitively expensive for real time applications that use lots of samples.

It turns out that there is an $O(N)$ version of the algorithm that works just as well.

1. Weight each sample in the source population
2. Normalize the weights so the sum of the weights is 1
3. Calculate a step size $s = 1/N$
4. Pick a random starting value β in the range [0, s]
5. Let $\gamma = 0$ be the sum of the weights seen so far
6. Let $i = 0$ be the index of the current weight
7. Iterate N times
   (a) While $\gamma \leq \beta$ add the w_i to γ and increment i, wrapping back to 0 if $\gamma > 1$
   (b) Add sample i to the new population
   (c) Increment $\beta = \beta + s$, wrapping back to 0 if $\beta > 1$
8. Return the new population

The second version of the algorithm is like building a single tower out of all the original blocks–each a different size–and then taking a comb with equally-spaced fingers and placing it in a random location alongside the tower. The blocks copied into the new population are those touching a finger of the comb. Sometimes several fingers will touch a single block; sometimes the spacing of the fingers will skip over one or more blocks. The random placement of the comb, however, means each block has a chance of being touched by a finger that is proportional to the weight of the block.

2.8 Application: Bio-informatics

Bioinformatics is one area where data visualization, data analysis, and machine learning techniques play an important role. There are many open questions in bioinformatics and computational biology.

- Sequence analysis: comparing sequences of base pairs (DNA) or amino acids (Proteins) within or between organisms. The challenge is matching strings in a way that usefully handles insertions, deletions, and swaps.
- Genome annotation: identifying locations in a DNA sequence that serve specific purposes such as encoding proteins, transfer RNA, or other features. There is still a significant amount of DNA whose function, if any, is still unknown.
- Computational evolutionary biology: building evolutionary trees based on the changes in DNA over time.
- Biodiversity analysis: measuring the genetic diversity of a single species or a whole ecosystem. Visualization is a key aspect of the problem.
- Analysis of gene expression: analysis and visualization of how the expression of genes changes in different types of cells or under different conditions.
- Analysis of regulation: connecting patterns in the DNA with the regulation of a protein. Clustering is one method used to identify co-expressed genes.
- Analysis of protein expression: similar to analysis of regulation, the data here is large amounts of mass spectrometry data providing amounts of peptides that come from different proteins. The goal is to identify which proteins are present given which peptides appear in the mass spec data (think Naïve Bayes).
- Prediction of protein structure: both heuristic and physics-based models of protein folding attempt to predict protein structure and function.
- Modeling biological systems: computer simulation of dynamic models of biological function. Visualization is a key component of computational modeling.
- Biomedical imaging: using computers to automate various processes in bioinformatics that involve visual inspection. Machine learning for object detection and visualization are key to developing useful processes.

2.8.1 Sequence Analysis

Sequence analysis requires a distance metric for comparing strings. DNA sequences have four possible values in each location $\{A, G, C, T\}$. Challenges arise because the strings can be different lengths, and the strings can contain deletions, insertions, or swaps.

- Deletion: one or more bases in the string are missing relative to the other string.
- Insertion: one or more bases have been inserted into the string relative to the other string.
- Swaps: one base has been substituted for another. Some base swaps are more likely than others.

One method of sequence analysis uses a dynamic programming approach to find the optimal global alignment of two sequences. The technique is similar to the dynamic time warping [DTW] distance metric.

Sequence alignment changes two of the assumptions of standard DTW. First, a gallery symbol can match only one probe symbol, so repeated matches of one gallery symbol to multiple probe symbols are replaced by the insertion symbol. Likewise, a probe symbol can match only one gallery symbol, so repeated matches of one probe symbol to multiple gallery symbols are replaced by the deletion symbol (the insertion and deletion symbol are both gaps, just in opposite strings). The cost of a swap is defined by a similarity matrix that relates each of the based pairs to one another. An example similarity matrix is given in table 2.

Base	A	G	C	T
A	10	-1	-3	-4
G	-1	7	-5	-3
C	-3	-5	9	0
T	-4	-3	0	8

Table 2: Example similarity matrix for DNA base pairs.

The Needleman-Wunsch algorithm is identical to dynamic time warping, except that the cost of moving horizontally or vertically is given by the gap penalty (e.g. -5, which is equal to the worst match), while the cost of moving diagonally is given by the similarity matrix. The other difference is that each string begins with a gap to allow the first character of the probe to begin on any character of the gallery. The first row and first column of the DTW matrix, called the F matrix in the literature, are the cost of increasing numbers of gaps. The recursive definition of the algorithm is given by (132).

$$F_{ij} = \max(F_{i-1,j-1} + S(A_i, B_j), F_{i,j-1} + d, f_{i-1,j} + d) \tag{132}$$

The algorithm can be implemented as a recursion, or by filling up the matrix from the $(0, 0)$ location.

```
d = gap cost
for i in length(A+1)
  F(i, 0) = d*i
for j in length(B+1)
  F(0, j) = d*j
for i in 1 to length(A+1)
  for j in 1 to length(B+1)
    C1 = F[i-1, j-1] + S(A[i-1], B[i-1]])
    C2 = F[i, j-1] + d
    C3 = F[i-1, j] + d
    F[i, j] = max(C1, C2, C3)
```

Example: Global string matching

Match the two strings: AAGCTTG with AGCCTA. Starting in the upper left, we can fill in the matrix using the algorithm above.

-	0	A	A	G	C	T	T	G
0	**0**	-5	-10	-15	-20	-25	-30	-35
A	-5	**10**	5	0	-5	-10	-15	-20
G	-10	5	**9**	**12**	7	2	-3	-8
C	-15	0	4	7	**21**	16	11	6
C	-20	-5	-1	2	16	**21**	16	11
T	- 25	-10	-6	-3	11	**24**	**29**	24
A	-30	-15	0	-5	6	19	24	**28**

Moving backwards from the lower right, we get the following match.

```
AAGC-TTG
A-GCC-TA
```

The Smith-Waterman algorithm is a variation of the Needleman-Wunsch algorithm. The main difference is that any cell that would receive a negative score instead receives a score of zero. To obtain matches, the backtracking process starts at the highest scoring cell and proceeds until the process encounters a cell with a zero value. Alignments generated using the Smith-Waterman algorithm do not necessarily provide a complete match between the two sequences.

```
di = insertion cost
dd = deletion cost
for i in length(A+1)
  F(i, 0) = d*i
for j in length(B+1)
  F(0, j) = d*j
for i in 1 to length(A+1)
  for j in 1 to length(B+1)
    C1 = F[i-1, j-1] + S(A[i-1], B[i-1]])
    C2 = F[i, j-1] + di (insertion cost)
    C3 = F[i-1, j] + dd (deletion cost)
    F[i, j] = max(0, C1, C2, C3)
```

2.8.2 Basic Local Alignment Search Tool [BLAST]

BLAST is the most commonly used tool for identifying homologies that exist between protein or DNA sequences. It is a heuristic method of identifying matches that uses four stages to filter out unlikely matches, not unlike a cascade classifier. The idea is to do fast and simple processes to identify likely matching locations and then increase the complexity and accuracy of the matching algorithms in each subsequent stage, using only the match locations that pass through the prior stages.

Given two full sequences A and B, the BLAST process is as follows. Note that the gallery sequence B can be much longer than A, as the process does not require A to stretch out to completely match B.

BLAST Algorithm

1. Discard low-complexity regions of A. For example, repeated elements, or sequences made up of only a few values that are not discriminative.

2. Use matches between short, fixed-length sequences to identify which sequences commonly occur in both A and B.

   Divide A into overlapping sequences of length N. For example, for $N = 3$ the sequence $AGCCTG$ would become the set of strings $S = \{AGC, GCC, CCT, CTG\}$. Then compare each subsequence S_i against every possible location in B. Each comparison receives a score using a similarity matrix, such as the one given above for the Needleman-Wunsch algorithm. For a given location in B, the algorithm records those S_i that receive a sufficiently high score. At the end of this process, each S_i has a total score T_i. Calculate the set of high scoring subsequences $S' = \{S_i | T_i > T_c\}$.

3. Identify locations in B where the sequences in S' match exactly. These form seed locations in the F matrix that is the space of possible matches between A and B.

4. Look for places in the F matrix where two matches occur in the same diagonal within a distance A and perform an ungapped, or exact alignment between the two sequences and extend it as far as possible until the match becomes poor. This forms a seed match for the next stage. Only those seed matches whose total scores exceed a threshold move on to the next stage.

5. For each ungapped extension from the prior stage, perform a gapped alignment to generate a complete alignment between A and B. Only those gapped alignment whose scores exceed a threshold move on to the final stage.

   A gapped alignment permits gaps, insertions, and replacements. The gapped alignment process forces the solution to go through at least one point on the seed match. It also uses a dropoff parameter that halts the search if the best score drops too far below the best score found so far. The dropoff parameter limits the search to an area around the optimal path found so far and avoids computing the entire F matrix. The gapped alignment algorithm is another variant of DTW that uses different scores for starting an insertion or deletion versus continuing an insertion or deletion. Otherwise, it is similar to the algorithms described above.

6. The final stage is to perform a final gapped alignment, but with a larger dropoff value, and to traceback the exact match sequence. The user ends up seeing the best K matches (e.g. K = 500).

One proposed extension to the BLAST algorithm that enables it to run faster is to insert another stage after the ungapped matching that executes what is called a semi-gapped match that allows insertions and deletions only between fixed length ungapped matches. This reduces the complexity of the paths through the F matrix, which makes semi-gapped matching faster than gapped matching. By inserting it between the ungapped and gapped matches, it reduces the number of matches passed on to the gapped matching stage without filtering out likely good matches.

Note the similarity in goals between the BLAST algorithm sequence of stages and a cascade classifier. Both are looking for unlikely events and make use of a sequence of stages to filter out unlikely low probability matches. At each stage, the most important concern is not filtering out potentially good matches.

The heuristic method is justified because the computational cost of finding all good matches of length L out of two strings of length $N > L$ is $N^2(N-L)(N-L) = O(N^4)$, which is infeasible for long strings.

3 Appendix: Python GUI Development

Structure of Python Applications

Python is an object-oriented language, if you choose to use it that way. Object-oriented design is an effective method of organizing applications. At the top level, we'll create a class that represents the application. All of the capabilities of the application will be methods of the application class. The application's init method will handle program startup, and its main method will begin the event loop that waits for user input.

Elements of a GUI:

- **Root**: the base object for a Tkinter application. It is the meta-manager for windows, menus, and events. In Tkinter, the root can also be attached to a visible window, but you can make the root window invisible and create other windows that are separate from the root window.
- **Windows**: visual window elements for the application. Windows are containers for the Canvas and other GUI elements like buttons and sliders.
- **Canvas**: a virtual drawing space on which visual elements, graphics, and images reside. The Canvas is separate from, and can be much larger than the window. Only the part of the Canvas within the window can be seen.
- **Menus**: drop-down or pop-up lists of commands a user can select using a mouse.
- **Events**: a keypress, mouse click, mouse drag, menu selection, or other events such as a window coming into the foreground.
- **Callbacks**: a function that is called when an event occurs.

3.0.1 Application Design

Packages: The packages we will need include the Tkinter and tkFileDialog packages. We may add others as we add capabilities. The Tkinter package is the general package for windows, menus, events and other widgets. The tkFileDialog handles giving the user a standard file selection dialog.

Application class: A good design strategy is to encapsulate your application as a class. The init method for the class sets up the application, and all of the callbacks and handler functions are methods. All of the application state information is stored as fields of the application class, which guarantees that all class methods have access to application state data. If you don't use a class, you'll end up creating a class to hold that information anyway.

Initialization: The `__init__` method has to do a number of things to set up the application.

- Create a root widget
  `self.root = Tkinter.Tk()`
- Set up the geometry of the window
  `self.root.geometry( "%dx%d+50+30" % (dx, dy) )` (note the X-window geometry)
- Set the window title
  `self.root.title( "Display App" )`
- Set the maximum size of the window for resizing
  `self.root.maxsize( 1024, 768 )`

- Bring the window to the front
  `self.root.lift()`
- Make the menus (write a function to handle it)
- Create the Canvas widge (write a function to handle it)
- Set the key and button bindings (write a function to handle it)
- Set up any state information required by the application (write a function to handle it)

Menus: The menus are set up as a tree. At the root is the menu bar. The main pull-down menus are children of the menu bar, and fold-out menus can be inserted into them. All of the elements of the tree are Menu objects, even the menu bar. The following sequence creates the menu bar, sets it as the main menu widget for the Tk root, then creates a file menu and a command menu and adds them into the menu bar.

```
# create a new menu
self.menu = tk.Menu(self.root)

# set the root menu to our new menu
self.root.config(menu = self.menu)

# create a file menu
filemenu = tk.Menu( self.menu )
self.menu.add_cascade( label = "File", menu = filemenu )

# create a command menu
cmdmenu = tk.Menu( self.menu )
self.menu.add_cascade( label = "Command", menu = cmdmenu )
```

To add items to a menu we use the `add_command` and `add_separator` methods. For example, to build a file menu with open and quit options, we could write the following. The calls to add_command tell Tk to call the handleOpen and handleQuit methods when the user selects Open and Quit, respectively.

```
filemenu.add_command( label = "Open...", command = self.handleOpen )
filemenu.add_separator()
filemenu.add_command( label = "Quit", command = self.handleQuit )
```

Canvas: The Canvas object provides the drawing surface on which we create widgets (controls) and visual elements such as lines, dots, circles, etc. If we have no other objects to create, then all that needs to occur for initialization is creating and packing the canvas. The Canvas size should, in general, be at least as big as the window.

```
self.canvas = tk.Canvas( self.root, width=dx, height=dy )
self.canvas.pack( expand=tk.YES, fill=tk.BOTH )
```

Bindings: A binding connects a user action, such as a mouse button click or keypress, to a specific function. Almost any user action can be bound to a function. In the example below, the left mouse button (button 1) and motion while the left mouse button is held down are bound to two separate functions.

```
self.root.bind( '<Button-1>', self.handleButton1 )
self.root.bind( '<B1-Motion>', self.handleButton1Motion )
```

Functions that handle events must take an event object as a parameter. The event object has many useful fields that provide information about the event. The method handleButton1, for example, might look like the following.

```
def handleButton1(self, event):
    print 'handle button 1: %d %d' % (event.x, event.y)
```

You can bind functions to regular keyboard events as well as events where the user is holding down a modifier key. For example, to catch the event where the user holds down the command key and the q key, you would make the following binding.

```
self.root.bind( '<Command-q>', self.handleModQ )
```

You can also use the terms `Option-q` and `Control-q` for the option and control keys, respectively.

Drawing Objects: Tkinter allows you to make simple graphical elements on a Canvas. Standard Canvas graphical objects include arcs, images, lines, ovals, polygons, rectangles, and text. Each of these objects has many options that control its appearance. As an example, the following creates a plus sign around a given point using two lines. Note that the code stores a reference to each line created in the list `self.lines`. If you want to edit or change the appearance of objects, you have to store a reference to the object. It's useful to initialize whatever list or data structure you use to store graphical objects in the initialization function.

```
curline = self.canvas.create_line( event.x-5, event.y, event.x+5, event.y )
self.lines.append(curline)
curline = self.canvas.create_line( event.x, event.y-5, event.x, event.y+5 )
self.lines.append(curline)
```

If we keep around the references to all of the lines we create, it means we can later move them using the coords method or delete them when we're done. The following example shows how you would traverse the list and delete all of the line elements.

```
for line in self.lines:
    self.canvas.delete( line )
```

Moving Objects: Tkinter provides a function `coords` that lets you both query and move objects. If you use the function with only the graphics object as the argument, it returns the coordinates of that object. If you use the function with the graphics object and new coordinates as arguments, it moves the object.

```
# get the coordinates of the line
loc = self.canvas.coords( line )

# shift the line two pixels down and to the right
 self.canvas.coords( line, loc[0]+2, loc[1]+2, loc[2] + 2, loc[3] + 2 )
```

Window Coordinates: The coordinate system of the window has its origin in the upper left corner, with positive X going right and positive Y going down. This configuration is slightly different than we're used to thinking in that Y increases as you move down instead of up. It's an artifact of the fact that cathode ray tube displays were designed to draw pixels on the screen beginning in the upper left and ending in the lower right.