A

Mathematics and Algorithms for Interactive Systems

There are a variety of mathematical concepts and basic algorithms that are useful in interactive systems. Most of these concepts are taught in various computer science courses and were developed in areas related to interaction. This appendix collects together the most useful algorithms along with some examples of their use. Many people wanting to work in interactive systems will know many of these concepts already. Because of this, a front to back read of this material may not be necessary. It is assumed that readers will dip in and out of this section in various places to find the information that they need. To support such opportunistic reading, each concept is presented in the following format:

- Prerequisites – these are other concepts that are assumed in the current discussion.
- Overview – this is a brief description of the concept and what it is good for.
- Description – details of the concept with illustrated examples
- Implementation – psuedo-code for implementation of the concept

The psuedo-code is in C/C++/Java syntax with some modifications. Because arrays are frequently used, the code uses standard multidimensional array notation ([x,y] rather than [x][y]). In addition, array parameters are specified with their lengths. For example:

```
double dotProduct( double A[n], double B[n])
```

indicates two, one-dimensional array arguments that both have a length of n. The declaration

```
double [r, c] innerProduct( double A[r, n ], double B[n, c] )
```

indicates two, two-dimensional array arguments where the number of columns in A is equal to the number of rows in B and the dimension of the result is r by c.

While non-standard for C/C++/Java, this notation greatly simplifies the algorithms.

All mathematical concepts are presented both in mathematical notation and as functional expressions or as procedures. Many programmers find the functional/procedural notation easier to read even if it is more verbose.

The concepts are grouped into categories with introductions for each category. The major categories are:

- Basic mathematics – vectors, matrices, linear equations, geometry
- Basic algorithms – priority queues, minimal path searching
- Classifiers – vector classifiers, sequence classifiers.

A1 – Mathematics

The mathematics of vectors, matrices and linear equations form the basis for many of the algorithms in interactive systems. Because so many interfaces are based on graphical images, knowledge of 2D and 3D geometry is also helpful. Because many graphical user interfaces are based on manipulation of points, it is also helpful to compute several properties of a collection of points.

A1.1 – Vectors

Prerequisites

Arithmetic

Overview

A vector is simply a one dimensional array of real numbers [x_1, x_2, x_3, . . .x_n]. Vectors are used to represent geometric points as well as features of objects. In geometric situations we use vectors with 3 or 4 elements. In other situations the vectors may be very long. For example text documents are frequently classified by the count of the number of occurrences of each word in the document. Therefore a document feature vector has one element for each possible word (a very long vector). Operations on vectors are the basis for a variety of other algorithms. All of the algorithms in this section are O(N), where N is the number of elements in a vector.

Description

The following notational conventions are common for vectors. Names of vectors are generally given with an arrow over the name ($\vec{v}$) and are represented as column vectors. One can think of a column vector as an N by 1 dimensional array (N rows and 1 column). The elements of the vector are represented by the name of the vector and the index of the element as a subscript. Subscripts by convention start with one.

$$\vec{v} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix}$$

The transpose of a vector is a row vector (1 by N). When dealing with simple vectors, the row or column nature is not important. However, it does become important when using vectors with matrices.

$$\vec{v}^{\,T} = \begin{bmatrix} v_1 & v_2 & v_3 & v_4 \end{bmatrix}$$

A1.1a – Vector Arithmetic

Vectors are added and subtracted by adding and subtracting their components. One can also add or subtract scalar values

$$\vec{a} + s = \begin{bmatrix} a_1 + s & a_2 + s & a_3 + s & \ldots & a_n + s \end{bmatrix}$$
$$\vec{a} - s = \begin{bmatrix} a_1 - s & a_2 - s & a_3 - s & \ldots & a_n - s \end{bmatrix}$$

$$\vec{a} + \vec{b} = \begin{bmatrix} a_1 + b_1 & a_2 + b_2 & a_3 + b_3 & \ldots & a_n + b_n \end{bmatrix}$$
$$\vec{a} - \vec{b} = \begin{bmatrix} a_1 - b_1 & a_2 - b_2 & a_3 - b_3 & \ldots & a_n - b_n \end{bmatrix}$$

Implementation

```
double [n] add(double V[n], double S)
{    double rslt[n];
     for (int i=0; i<n; i++)
     {    rslt[i]=V[i]+S;  }
     return rslt;
}

double [n] add(double A[n], double B[n])
{    double rslt[n];
     for (int i=0; i<n; i++)
     {    rslt[i]=A[i]+B[i];  }
     return rslt;
}
```

A1.1b - Vector Length

A vector can be thought of as a line extending from the origin to a point in n-dimensional space. The length of a vector $\vec{v}$ with N elements is represented by $\|\vec{v}\|$.

$$length(\vec{v}) = \|\vec{v}\| = \sqrt{\sum_{i=1}^{N} v_i^2}$$

Implementation

```
double length( double V[n])
{    double dSum=0;
     for (int i=0; i<n; i++)
     {    dSum+=V[i]*V[i];  }
     return Math.sqrt(dSum);
}
```

A1.1c – Unit Vectors

Unit vectors are vectors whose length is one. For any non-zero vector $\vec{v}$ we can compute a unit vector $\vec{u}$ in the same direction.

$$\vec{u} = unit(\vec{v}) = \frac{\vec{v}}{\|\vec{v}\|}$$

Implementation

```
double [n] unit ( double V[n] )
{    double rslt[n];
     double len = length(V);
     for (int i=0; i<n; i++)
     {    rslt[i]=V[i]/len;  }
     return rslt;
}
```

A1.1d - Dot Product

The dot product of two vectors is found in many applications. Its most common use is to combine a vector of variables with a vector of constant coefficients to produce a linear equation. For example $\vec{C} \bullet \vec{V} = 10$ would be the same as $c_1 v_1 + c_2 v_2 + c_3 v_3 = 10$. The dot product is defined as:

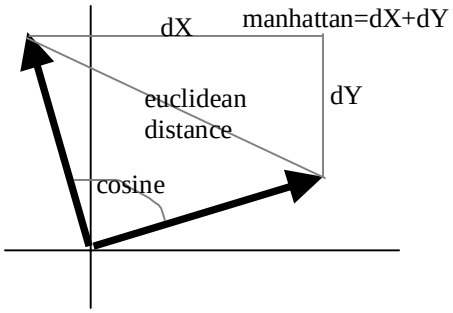
$$\vec{a} \bullet \vec{b} = \sum_{i=1}^{N} a_i b_i$$

Implementation

```
double dotProduct(double A[n], double B[n])
{    double sum=0;
     for(int i=0; i<n; i=+)
     {    sum+=A[i]*B[i];  }
     return sum;
}
```

A1.1e – Comparing Vectors

It is frequently helpful to compare two vectors to see how similar they might be. There are three basic approaches for comparing two vectors. They are the Manhattan distance, the Euclidean distance and the cosine. The following figure in 2D illustrates these three forms of distance between two vectors.



Manhattan Distance

It is called the Manhattan distance because it is the distance one would walk between two points in downtown Manhattan. It is not possible to walk directly between two points, one must follow the rectangular streets.

$$manh(\vec{a},\vec{b}) = \sum_{i=0}^{n-1} \left| a_i - b_i \right|$$

Implementation

```
double manhattan(double A[n], double B[n])
{    double dist=0;
     for (int i=0; i<n; i++)
     {    if (A[i]>B[i])
          {    dist+=A[i]-B[i];  }
          else
          {    dist+=B[i]-A[i];  }
     }
     return dist;
}
```

Euclidean Distance

This is geometrically the shortest distance between two points.

$$dist(\vec{a},\vec{b}) = \left\|\vec{a} - \vec{b}\right\|$$

Implementation

```
double distance(double A[n], double B[n])
{    return length(subtract(A,B));  }
```

Cosine Distance

The cosine of the angle between any two vectors is the dot product of their unit vectors. The cosine has several interesting properties. If two vectors are parallel to each other, then the cosine of their angle is 1. If they are perpendicular (completely unrelated) then their cosine is 0. If two vectors are directly opposite of each other then their cosine is –1. The cosine of the angle between two vectors is useful in 2D and 3D geometry, but it is also a useful comparison between two vectors in general. It is sometimes used in text retrieval to compare the word frequency vectors of two documents. One obvious optimization of this computation is to keep the unit vectors rather than the original vectors, so that the cosine can be calculated directly by the dot product.

$\vec{a} \bullet \vec{b} = \cos(\theta) \cdot \left\|\vec{a}\right\| \cdot \left\|\vec{b}\right\|$ where θ is the angle between $\vec{a}$ and $\vec{b}$.

therefore

$$\cos(\theta) = unit(\vec{a}) \bullet unit(\vec{b})$$

Implementation

```
double cosine(double A[n], double B[n])
{    return dotProduct( unit(A), unit(B) );  }
```

A1.2 – Matrices

Prerequisites

Vectors (A1.1,A1.1d)

Overview

Matrices are used to represent a large variety of linear equations and linear transformations. Matrix multiplication, matrix inverse and solving systems of linear equations are all quite straightforward using matrices. The key algorithms for these are presented here. A matrix is a two dimensional array of real values.

$$M = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix}$$

A1.2a - Transpose

The transpose of a matrix is simply a reversal of rows and columns.

$$M^T = \begin{bmatrix} a_{1,1} & a_{2,1} & a_{3,1} \\ a_{1,2} & a_{2,2} & a_{3,2} \\ a_{1,3} & a_{2,3} & a_{3,3} \end{bmatrix}$$

A1.2b - Matrix multiply

Matrix multiply combines an (*L*x*M*) with an (*M*x*N*) matrix to produce an (*L*x*N*) matrix. Each element (*i*, *j*) of the resulting matrix is the dot product of the *i*th row of the first matrix and the *j*th column of the second matrix.

- Matrix multiplication is associative. $(A \bullet B) \bullet C = A \bullet (B \bullet C)$
- Matrix multiplication is **not** commutative. $A \bullet B \neq B \bullet A$ in many cases.

Multiplying an *N*x*N* matrix times a vector of length *N* will produce a new column vector which is a linear transformation of the original vector. Matrix multiplication can thus be used to model any linear transformation of a vector of values. Where such transforms occur in sequence, multiplication of their matrices together will produce a single matrix that is the concatenation of all the

transforms. This is very useful in computer graphics as well as in a variety of other situations requiring the transformation of data.

A most useful matrix is the identity matrix *I*, which has 1.0 on all diagonal elements and zeroes everywhere else. The identity matrix has the property $I \bullet M = M$.

It is also true that if (*A*=*B*) then $(M \bullet A = M \bullet B)$ and $(A \bullet M = B \bullet M)$.

Implementation

```
double [l,n] matrixMultiply(double A[l,m], double B[m,n])
{    double result[ ]=new double[l,n];
     for (int r=0;  r<l;  r++)
     {    for (int c=0; c<n; c++)
          {    result[r,c]=0.0;
               for (int i=0; i<m; i++)
               {    result[r,c]=result[r,c]+A[r,i]*B[i,c]; }
          }
     }
     return result;
}
```

A1.2c - Solving linear equations

Any system of linear equations can be modeled as the following matrix equation

$$M\vec{V} = \vec{C}$$

Where $\vec{V}$ is a vector of *N* values, *M* is an *N*x*N* matrix of coefficients and $\vec{C}$ is a vector of resultant values. Take for example the following system of equations.

$$2x + 3y - 4z = 10$$
$$3x + 2y - z = 4$$
$$x - 7y + 2z = -2$$

This system can be modeled using the following matrix equation where each row of the matrices *M* and $\vec{C}$ corresponds to an equation and each column of *M* corresponds to one of the variables.

$$\begin{bmatrix} 2 & 3 & -4 \\ 3 & 2 & -1 \\ 1 & -7 & 2 \end{bmatrix} \bullet \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 10 \\ 4 \\ -2 \end{bmatrix}$$

The simplest way to solve this system is to use the Gauss-Jordan reduction method. This method is based on two properties of linear equations. The first is that any linear equation can be multiplied or divided by any non-zero value without changing the set of points that satisfy the equation. The second property is that two equations can be subtracted from each other to produce a third equation that is consistent with the system of equations. We use these two properties to transform the matrix *M* into an identity matrix. We divide a row of the matrix by its diagonal value (which makes the diagonal value a 1) and then we multiply and subtract that equation from all other equations so that every other element in that column becomes a zero. Having done that we will have $I \bullet \vec{V} = \vec{C}'$, where $\vec{C}'$ is a column vector of the solution values. This algorithm for solving linear equations is $O(N^3)$, where *N* is the number of variables.

Implementation

Note that this algorithm modifies both M and C in the course of its operation.

```
double [n] solveLinear(double M[n,n], double C[n])
{    for (int r=0; r<n; r++)
     {    if (M[r,r]==0.0)
          {    generate an error. There is no solution }
          double div=M[r,r];
          for (int c=0; c<n; c++)
          {    M[r,c]=M[r,c]/div; }
          C[r]=C[r]/div;

          for (int r2=0; r2<n; r2++)
          {    if (r2!=r)
               {    double mul = M[r2,r];
                    for (int c=0; c<n; c++)
                    {    M[r2,c]=M[r2,c]-M[r,c]*mul; }
                    C[r2]=C[r2]-C[r]*mul;
               }
          }
     }
     return C;
}
```

A1.2e - Inverting a matrix

It is sometimes useful to compute the inverse of a matrix. The inverse of a matrix (M^{-1}) is defined such that $MM^{-1} = I$ and $M^{-1}M = I$. We compute the

inverse by replacing the vector $\vec{C}$ with an identity matrix and using the same techniques that used in solving a set of linear equations. If we can compute the inverse of a matrix, we can use it to solve linear equations. Not all matrices are invertible. This happens when the one or more of the equations is some linear combination of the others. The algorithm given below will detect this condition. Matrix inversion is $O(N^3)$, where *N* is the number of variables.

$$M\vec{V} = \vec{C}$$
$$M^{-1}M\vec{V} = M^{-1}\vec{C}$$
$$I\vec{V} = M^{-1}\vec{C}$$
$$\vec{V} = M^{-1}\vec{C}$$

Implementation

```
        Note that this algorithm modifies the matrix M
    double [n,n] invertMatrix(double M[n,n] )
    {   double R[ ][ ]=identityMatrix(n);
        for (int r=0; r<n; r++)
        {   if (M[r,r]==0.0)
            {   generate error. There is no inverse. }
            double div = M[r,r];
            for (int c=0; c<n; c++)
            {   M[r,c]=M[r,c]/div;
                R[r,c]=R[r,c]/div;
            }

            for (int r2=0; r2<n; r2++)
            {   if (r!=r2)
                {   double mul=M[r2,r];
                    for (int c=0; c<n; c++)
                    {   M[r2,c]=M[r2,c]-M[r,c]*mul;
                        R[r2,c]=R[r2,c]-R[r,c]*mul;
                    }
                }
            }
        }
        return R;
    }

    double [n,n] identityMatrix(int n)
    {   double R[ ]= new double[n,n];
        for (int r=0; r<n; r++)
        {   for (int c=0; c<n; c++)
            {   if (r==c)
                {   R[r,c]=1.0;  }
                else
                {   R[r,c]=0.0;  }
            }
        }
        return R;
    }
```

A1.2f – Linear approximation

Prerequisites

Vectors(xx), Matrix Inverse(xx), Homogeneous coordinates(xx)

Overview

It is sometimes necessary to find a linear function that approximates a set of points. When there are *N* variables and more than *N* points the linear function will in general only approximate those points rather than exactly pass through them. To do this we compute the linear function and measure its error from the desired outcome. We try to minimize the sum of the squares of these errors (least squares method). Finding an approximation involves creating a matrix of all of the point data. This creation step is $O(V^2P)$ where *V* is the number of variables and *P* is the number of sample points. The resulting matrix must then be inverted which is $O(V^3)$.

Description

In a system of two variables the linear function that we are trying to discover can take the following form.

$$ax + by + c = d$$

In vector form of this equation would be

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \bullet \begin{bmatrix} a \\ b \\ c \end{bmatrix} = d$$

To use this particular form of linear equation, we transform [*x y*] into the **homogeneous** point [*x y* 1]. The purpose of the additional 1 is to incorporate the offset coefficient *c* into the vector equation. The general *N* dimensional form of the equation is $\vec{v} \bullet \vec{C} = d$, where $\vec{v}$ is a homogenous point, $\vec{C}$ is a vector of coefficients and *d* is the desired outcome of the function. The coefficient vector completely characterizes the linear function. For many situations, the desired outcome *d* is zero.

We can also use this method to approximate non-linear functions that use only linear coefficients. Suppose we have two coordinates [*x y*], we might use an equation of the form

$$\begin{bmatrix} x & y & x^2 & y^2 & xy & \frac{1}{x} & \sin(y) & 1 \end{bmatrix} \bullet \begin{bmatrix} e \\ f \\ g \\ h \\ i \\ j \\ k \\ l \end{bmatrix} = d$$

Because the values for *x* and *y* are known for any sample point, the non-linear values are also known. Therefore, the equation is still linear for the coefficients *e-l*. We have simply converted the 2 dimensional space of *x, y* into a 7 dimensional space, by computing the new values.

Since the linear function will only approximate the points, each point will have an error. This error e is calculated as $\vec{v} \bullet \vec{C} - d = e$. Since *e* may be positive or negative, we generally use e^2, which is differentiable and always positive. For some set of *P* points, we want to calculate the vector $\vec{C}$ that minimizes the sum of the squared errors. We set up our error calculation as follows.

$$B = \begin{bmatrix} x_{1,1} & x_{1,2} & \ldots & 1 \\ x_{2,1} & x_{2,2} & \ldots & 1 \\ \ldots & \ldots & \ldots & 1 \\ x_{P,1} & x_{P,2} & \ldots & 1 \end{bmatrix}$$ where $x_{i,j}$ is the *j*th coordinate of the *i*th point

$$\vec{Y} = \begin{bmatrix} d_1 \\ d_2 \\ \ldots \\ d_P \end{bmatrix}$$ where d_i is the desired outcome for the *i*th point

The error *E* is calculated as

$$E = \vec{Y} - B \bullet \vec{C}$$

Given this matrix formulation for *E* we can calculate a vector $\vec{C}$ that will minimize the sum of the squares of *E*. This is taken from [Castleman96 p502].

$$\vec{C} = \left(B^T \bullet B\right)^{-1} \bullet (B^T \bullet \vec{Y})$$

For an *N* dimensional space, $(B^T \bullet B)$ is an (*N*+1) x (*N*+1) matrix *W* of the form

$$W = (B^T \bullet B) = \begin{bmatrix} \sum_{i=1}^{P} x_{i,1}x_{i,1} & \sum_{i=1}^{P} x_{i,1}x_{i,2} & \cdots & \sum_{i=1}^{P} x_{i,1}x_{i,N} & \sum_{i=1}^{P} x_{i,1} \\ \sum_{i=1}^{P} x_{i,2}x_{i,1} & \sum_{i=1}^{P} x_{i,2}x_{i,2} & \cdots & \sum_{i=1}^{P} x_{i,2}x_{i,N} & \sum_{i=1}^{P} x_{i,2} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \sum_{i=1}^{P} x_{i,N}x_{i,1} & \sum_{i=1}^{P} x_{i,N}x_{i,2} & \cdots & \sum_{i=1}^{P} x_{i,N}x_{i,N} & \sum_{i=1}^{P} x_{i,N} \\ \sum_{i=1}^{P} x_{i,1} & \sum_{i=1}^{P} x_{i,2} & \cdots & \sum_{i=1}^{P} x_{i,N} & P \end{bmatrix}$$

For the same space, $(B^T \bullet Y)$ is a vector $\vec{V}$ of length (*N*+1), which has the form

$$\vec{V} = (B^T \bullet Y) = \begin{bmatrix} \sum_{i=1}^{P} x_{i,1}d_i \\ \sum_{i=1}^{P} x_{i,2}d_i \\ \cdots \\ \sum_{i=1}^{P} x_{i,N}d_i \\ \sum_{i=1}^{P} d_i \end{bmatrix}$$

The algorithm, then is to calculate *W* and $\vec{V}$, compute W^{-1} and multiply them together to compute $\vec{C}$.

Implementation

```
double [ ] leastSquares(double B[P,N], double d[P])
{     double W[ ][ ] = new double[N+1,N+1];
      double V[ ]=new double[N+1];

      for (int i=0; i<=N; i++)
      {     V[i]=0.0;
            for (int j=0; j<=N; j++)
            {     W[i,j]=0.0;  }
      }

      for (int p=0; p<P; p++)
      {
            for ( int i=0; i<N; i++)
            {     V[i]=V[i]+B[p,i];
            }

      }
}
```

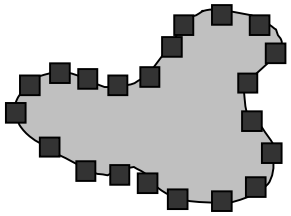
A1.3 - Clouds of points

Prerequisites

Vectors(xx)

Overview

When working with geometric entities and inputs, one frequently encounters a collection of points. These points might be all of the points in an ink stroke, the vertices of a polygon, all of the pixels that make up a shape, or all of the purple pixels in an image. All of these can be modeled as a list of [*x y*] or [*x y z*] coordinates.



There are a variety of features of such clouds that are helpful. The center of gravity is a useful reference point from which to base these features. The angle of

eccentricity (angle of the longest axis on the cloud) is useful when determining orientation of a shape, the size and elongation are also useful features.

A1.3a - Center of gravity

The simplest feature for a cloud of points is its center of gravity. If we assume that all points have equal weight then this is simply the average of all of the points.

$$Cx = \frac{1}{N}\sum_{i=1}^{N} x_i$$

$$Cy = \frac{1}{N}\sum_{i=1}^{N} y_i$$

Implementation

```
Point center(Point points[n])
{    Point C;
     C.x=0.0;
     C.y=0.0;
     for (int i=0; i<n; i++)
     {    C.x=C.x+points[i].x;
          C.y=C.y+points[i].y;
     }
     C.x=C.x/n;
     C.y=C.y/n;
     return C;
}
```

A1.3b - Moments

The shape of a set of points can be described by a set of moments [Castleman96 xx]. The most interesting moments are relative to the center of the shape. Computing the moments relative to the center makes the values of the moments invariant with respect to translation (the position of the object). There is a whole family of these moments that have been defined. This family is parameterized by two integers *p* and *q*. These moments were designed for evaluating the shapes of gray scale images[Pav.. see Castleman96 xx]. However, for our purposes the following formula again assumes that the weights of all the points are equal.

$$m_{p,q} = \sum_{i=1}^{N} (x_i - Cx)^p (y_i - Cy)^q$$

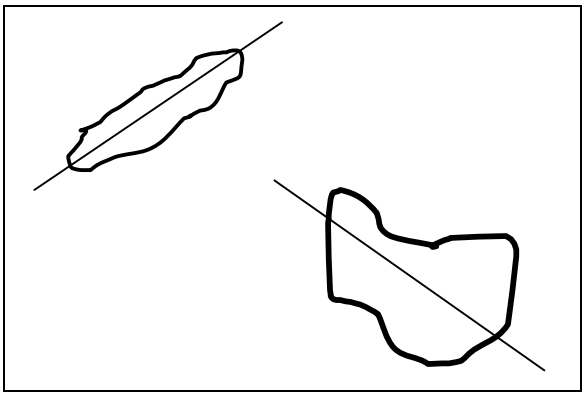
Implementation

the following are the more commonly used moments

```
double moment11(Point points[n],Point center)
{    double sum=0.0;
     for (int i=0; i<n; i++)
     {    sum=sum+(points[i].x-center.x)*points[i].y-center.y);  }
     return sum;
}

double moment20(Point points[n], Point center)
{    double sum=0.0;
     for (int=0; i<n; i++)
     {    double diff=points[i].x-center.x;
          sum=sum+diff*diff;
     }
     return sum;
}

double moment02(Point points[n], Point center)
{    double sum=0.0;
     for (int=0; i<n; i++)
     {    double diff=points[i].y-center.y;
          sum=sum+diff*diff;
     }
     return sum;
}
```

A1.3c - Primary Angle

A very useful measurement is the primary axis of a shape. The primary axis is the angle of the greatest extent of the shape. The following shapes are examples.



The angle θ of the primary axis can be computed from the first and second degree moments of the shape.

$$\theta = \frac{1}{2}\tan^{-1}\left(\frac{2m_{1,1}}{m_{2,0} - m_{0,2}}\right)$$

Note that the primary axis has two possible primary angles differing by 180 degrees. When comparing two shapes it is useful to normalize both shapes by rotating the primary angle of each of them to the positive X axis and then comparing. To do this, one must remember to make two comparisons so as to compensate for the ambiguity in the primary angle.

Since the tangent function has singularities when the denominator approaches zero, a good implementation will use the ATAN2(dx,dy) function found on most systems. ATAN2 checks all of the various reflections so as avoid the singularities caused for the normal arctan function.

Implementation

```
double primaryAngle(Point points[n])
{    Point C = center(points);
     return      atan2(      moment20(points,C)-moment02(points,C),
2*moment11(points,C) )/2;
}
```

Another approach for determining the primary angle is to use the vector from the center to the point that is farthest from the center [Kurlander xx]. This strategy,

however, is more sensitive to outliners than computing that angle from the moments.

A1.3d - Dimensions of a cloud

It is frequently helpful to determine the "size" of a cloud of points. The simplest computation of size is to use a bounding rectangle. The bounding rectangle can easily be computed by taking the maximums and minimums in X and Y. This is frequently done after the points have been rotated so that the primary angle is on the X axis. Using max and min is subject to distortions of outlier points. This frequently does not matter, however, if outliers are concern, then we can use the average distance from the center in X and Y.

$$SizeX = \frac{1}{N}\sum_{i=1}^{N}\left|x_i - Cx\right|$$

$$SizeY = \frac{1}{N}\sum_{i=1}^{N}\left|y_i - Cy\right|$$

These two size estimates can also be used to compute the elongation or eccentricity of a shape as $\frac{SizeY}{SizeX}$. If the cloud has been rotated so that the primary angle is along the X axis, this measure of elongation will lie between zero and one. This elongation measure is also size invariant.

A2 – Algorithms

Prerequisites

Basic data structures

Overview

These algorithms are drawn from basic heuristic search methods. They are frequently used in various optimization algorithms to compute a minimal sequence of operations.

A2.1 - Priority queue

Overview

The priority queue is a simple structure for storing objects that have a priority. A priority is represented as an integer or floating point number. Items are inserted into the queue in any order. When an item is removed from the queue it is either the item with the largest or smallest priority, depending on how the queue is designed. For minimization problems we use a queue that always returns the smallest priority item in the queue. In our discussions we will return the smallest priority item. The implementation for returning the largest item simply reverses the less than comparisons into greater than.

Our implementation of a priority queue uses a structure called a heap. A heap is a complete binary tree such that any node in the tree always has less than or equal priority than either of its subnodes. Because our binary tree is complete, we can represent the tree in a simple array and compute the tree relations in the following way.

For any node at index i in the tree

parent(i)=(i-1)/2
*left(i)=i*2+1*
*right(i)=i*2+2*

The root of the tree is at index 0. When we insert a new item, we insert it at the end of the array, at the very bottom of the tree. We then compare it with its parent and if it has lesser priority, we exchange it with its parent. This compare/exchange process continues until either the root of the tree is reached or the inserted object reaches a point in the tree where its parent has a lesser priority. There will be at most $\log_2(N)$ comparisons and exchanges, where N is the number of items currently in the queue.

To remove an item from the queue we take the root item, since by induction the root has a lesser priority than all other items in the queue. We then take the last item in the queue and place it at the root. This item will almost always be out of place. This root item is compared with its left and right children. If either is less than the root item, then the root is exchanged with the smallest of the two. The original item is now in a new position and is then compared to its children. This compare and exchange process continues until the item is less than both of its children, or it has no children. This will take at most $\log_2(N)$ steps.

Implementation

```
interface PriorityObject
{    double priority();  }

class PriorityQueue
{    PriorityObject items[ ];
     int next=0;

     void insert(PriorityObject obj)
     {    items[next]=obj;
          int idx = next;
          next=next+1;
          int parent = (idx-1)/2;
          while (idx>0 && items[idx].priority()<items[parent].priority() )
          {    PriorityObject temp = items[parent];
               items[parent]=items[idx];
               items[idx]=temp;
               idx=parent;
               parent = (idx-1)/2;
          }
     }
```

```
    PriorityObject remove()
    {   PriorityObject result;
        PriorityObject temp;
        if (next==0)
        {   return null;  }
        result=items[0];
        next=next-1;
        items[0]=items[next];
        int idx=0;
        while (true)
        {   left = idx*2+1;
            right=left+1;
            if (left<next && items[left].priority()<items[idx].priority() )
            {   if (right<next && items[right].priority()<items[left].priority()
)
                {   temp=items[idx];
                    items[idx]=items[right];
                    items[right]=temp;
                    idx=right;
                }
                else
                {   temp=items[idx];
                    items[idx]=items[left];
                    items[left]=temp;
                    idx=left;
                }
            }
            else if (right<next &&
items[right].priority()<items[idx].priority() )
            {   temp=items[idx];
                items[idx]=items[right];
                items[left]=temp;
                idx=left;
            }
            else
            {   return result;  }
        }
    }
    boolean empty()
    {   return next==0;  }
}
```

A2.2 - Least-cost path

Prerequisites

Priority Queue(xx)

Overview

There are many problems that can be cast as a graph searching problem. Such problems are characterized by a state (some set of data values) from which there are several possible choices. Each choice leads to another state and each choice has a cost. The states and choices constitute the nodes and arcs of a weighted, directed graph. A series of such choices is a path. The cost of a path is the sum of the cost of its choices. What is desired is to make a series of choices that produces the least-cost path from some initial state to some goal state.

There are some problems that are defined not in terms of cost but in terms of benefit. It would seem in those cases that finding the maximal path would be appropriate. However, in a graph that has cycles, the maximal path is infinite. Such problems, however, can be recast into a minimal cost algorithm by computing *cost=maxBenefit-benefit*. We can now use these costs to compute a minimal cost path which will be the maximum benefit, acyclic path.

The algorithm works by placing each candidate path on a priority queue weighted by that path's cost so far. We then take the least cost path off of the queue and expand that path's choices. By always selecting the least-cost path and expanding it, we are guaranteed to find the shortest path if there is one. If there is no path, then the priority queue will become empty before the goal state is located. Each path placed on the queue retains a link to the path it extends. Once the goal state is reached the least cost path can be extracted by following these links back to the start state.

Implementation

```
interface State
{    Boolean matches(State S) returns true if this object "matches" S
     int nChoices() returns the number of choices leaving this state
     State choice(int N) returns the next state for choice N
     double choiceCost(int N)  returns the cost of choice N
}

class Path implements PriorityObject
{    double pathLength;
     double choiceCost;
     State nextState;
     Path previousState;
     Path (double cost, Path previous, State next)
     {    if (previous==null)
          {    pathLength=0;  }
          else
          {    pathLength = previous.pathLength+cost;  }
          choiceCost = cost;
          nextState = next;
          previousState = previous;
     }
     double priority()
     {    return pathLength;  }
}

Path leastCostPath( State start, State goal )
{    PriorityQueue Q = new PriorityQueue();
     Path startPath = new Path(0.0,null, start);
     Q.add(startPath)
     while (!Q.empty() )
     {    Path curPath = Q.remove();
          if (goal.matches(curPath))
          {    return curPath;  }
          for (int c=0; c<curPath.nChoices();  c++)
          {    if ( notAlreadyVisited(curPath.choice(c)) )
               {    Path next = new Path(curPath.choiceCost(c),
                                         curPath,
                                         curPath.choice(c) );
                    Q.add(next);
               }
          }
     }
}
```

The least cost path will never have a cycle. Therefore the call to **notAlreadyVisited** can both check to see if the next state has already been visited (in which case a path of equal or lower cost is already underway) and also add the state to the list of visited states. Since the number of visited states may be large, this test can be omitted without affecting the correctness of the algorithm.

A3 – Classifiers

Prerequisites

Vectors (A1.1), Matrices (A1.2), some of Algorithms(A2).

Overview

As interactive systems move beyond simple point and click interfaces there is an increasing need to recognize what the user has done, to recognize situations in which the user is acting and to identify the kinds of objects that the user is acting upon. This is particularly true when using cameras, microphones, gestures and speech as a mechanism for interacting. At the heart of most such advanced interaction techniques is the concept of a classifier. For example a user may sketch a character using a pen. The system may then want to classify that pen stroke as an "a", "b" or "c" or perhaps as "insert", "delete" or "paste". A speech system takes an audio input and attempts to classify it as "help", "save", "run" or "don't know". The range and depth of issues involved in classifier design is quite broad and can be the subject of multiple university courses. However, there are a small number of quite straightforward classifiers that can be readily programmed and applied to a variety of interactive applications.

Description

The goal of a classifier is to take some object *O* that has a vector of features $\vec{f}$ that are used to characterize the object *O*. A classifier has 2 or more classes of objects. By looking at the feature vector, the classifier attempts to identify which class the object *O* belongs to. Thus a classifier is a function $C(\vec{f}) \rightarrow class$. A class is simply a small finite enumeration of possibilities.

For example we may take the set of features [hasFeathers, number of legs, color] and attempt to classify animals as one of the set {pig, chicken, dog}. We could write our classifier by saying "if it has feathers and two legs, then it is a chicken otherwise if it is pink it is a pig, otherwise it is a dog." This is a relatively simple classifier. We are generally interested in more complicated classifications.

We are most interested in trainable classifiers. Trainable classifiers are general in nature in that they can compute a wide space of classifications depending upon the parameters. Thus our classifier becomes $C(\vec{f}, P) \rightarrow class$, where *P* is a parameter structure that controls how the classifier performs its classification. We train a classifier by taking a set of training examples and computing an appropriate parameter *P* that is consistent with the examples and usable by our

classifier. A training example is simply a feature vector for some sample object, with an associated class. Our training process attempts to generalize from the training examples to produce a classifier that works well for a wide range of feature vectors. We can categorize classifiers by the kinds of features and feature vectors that they can handle, and by the way in which they compute their parameter from a training set.

One of the biggest differences is between vector classifiers and sequence classifiers. A vector classifier assumes that for a given problem there is a finite set of features and every object has exactly the same number of features. Thus our classification can work by comparing feature values. Our simple animal classifier described above is a vector classifier. There are exactly three features. The sequence classifiers handle an arbitrary number of features in the feature vector with different objects having different numbers of features. In sequence classifiers the order or sequence in which values appear greatly determines the classification. String parsing is a sequence classifier. The features are all characters and it is the order of those characters that determines the parse. The algorithms for fixed vectors and sequences are very different.

Feature Selection

It is only possible to cover a small number of the most common classifiers here. There are many possible classifiers [Mitchell xx, Duda xx] each with its own strengths and weaknesses. In most interactive situations, the classifiers described here will suffice. Generally classification performance can be improved more by providing new features or combinations of features than by new classifier algorithms. If a classifier is not performing correctly it is usually helpful to identify cases where there is confusion and then try to identify some feature or features that would distinguish those cases. Adding the new features can frequently solve the problem.

In expanding the feature set for a problem, one should also recognize the “curse of dimensionality,” which is that the more features in a problem, the more training data is required for accurate learning. Adding features increases the number of dimensions in the feature space, and thus increases the number of examples required to effectively populate that space.

Types of features

We categorize features into three types: real numbers, nominal (enumerated) and distance comparable. The **real numbers** are one of the easiest types of features because we can talk about ranges of numbers, compute the distance between two

numbers and average a set of numbers. All of these are useful when building various types of classifiers. One of the problems with real numbers is that there are infinitely many of them. Even once we have a bound on the set of numbers (for example between –1.0 and 2.4) there are infinitely many possibilities inside. This is problematic for some types of classifiers.

Nominal or **enumerated** features have a fixed set of values. For example hasFeathers is either true or false. A political affiliation feature may be {Republican, Democrat, Libertarian, Other}. The advantage of such features is that they can be used to model a variety of concepts and for the purposes of our algorithms; the possible values of such a feature can be fully enumerated. It is possible to convert a numeric feature into a nominal one by *quantizing* the values. For example we could convert weight in pounds into {0-100=light, 100-200=medium, 200+ = heavy}. This would allow us to use weight for a classifier that requires nominal attributes. The problem with such quantizing is with the boundaries. Our classifier would treat a person weighing 99 pounds very differently from one weighing 101 pounds even though most people could not tell the difference by looking at them.

We can also convert a nominal feature into numeric one by assigning numeric values to each of the possibilities. For example {Republican=1, Democrat=2, Libertarian=3, Other=4}. The problem with this conversion is that many numeric algorithms perform generalizations based on numeric values. When nominal features are encoded numerically, these generalizations may not reflect the nature of the features. For example, it is not valid to say that the average of a Republican(1) and a Libertarian(3) is a Democrat(2). It is even worse to place Libertarian between Democrat and Other. We can assign the numeric encodings, but reasoning about them along the real number line is not conceptually valid.

The **distance comparable** features are not numeric, but they can be compared and a distance between two values can be assigned. An example might be the set of strings. We can create a “substitution cost” distance for strings. For example comparing “Heat” with “heat” might produce a very small distance because substituting “H” for “h” is very cheap. The strings “heet” and “heat” might be farther apart because substituting “e” for “a” is not too bad, but they are not as close as “H” and “h”. The strings “heap” and “heat” might be even farther apart for similar reasons. Another distance comparable feature might be animal species like {pigs, chickens, cows, elk, . . .}. We can assign a distance between any two species based on differences in their chromosome makeup. Though we can determine a distance between any two values for such a feature, there is no clear ordering of the features, nor can we compute an average. The fact that there is a

distance has value in some classifier algorithms, but they cannot be used in as many situations as numeric features. It is possible to take any sequence of nominal values and convert it into one or more distance comparable features using a distance metric such as minimal edit distance (A3.2a xx).

A3.1 - Vector Classifiers

Prerequisites

Vectors (xx), Matrices(xx), Classifiers(A3)

Overview

The vector classifiers are quite general and can solve a variety of problems. Many of them are quite easy to implement and to train. The ability to apply vector classifiers to a problem will greatly enhance the number of options that you have in detecting and responding to interactive human behavior.

In a vector classifier problem, every object has exactly the same number of features and all values for a feature are of the same type. This is in contrast to sequence classifiers that handle varying numbers of features. Example uses of vector classifiers are:

- using [hue, saturation] to classify image pixels as human skin or not.
- using [bodyTemperature, pulseRate, bloodPressure] to decide if a patient is healthy or sick.
- using [height, width, strokeLength, startPoint, endPoint] of an ink gesture to decide if it means insert, delete, bold, italic, or justify.

Description

There are two broad categories of vector classifiers: those that make a binary decision between two classes (sick/healthy, skin/not skin) and those that can classify any number of classes {insert,delete,bold,italic,justify}. Most binary classifiers return some confidence value of how much they support their conclusions. We can use this to turn a binary classifier into a multi-class classifier. If there are N classes, we can train N separate binary classifiers each of which decides Yes/No between one class and all of the other classes. We then run our feature vector through all N classifiers and take the class that yields the most confident yes.

Each classifier is trained using a training set. A training set is simply an unordered list of training examples. Each training example is a tuple

$[\vec{f}, c]$ where $\vec{f}$ is a feature vector and *c* is a class that corresponds to that combination of features.

Classifier algorithms vary in the way in which they *generalize* from training examples. In most realistic situations, the space of possible feature vectors is very large. In many cases we have far fewer training examples than there are possible points in the feature space. What we want is to generalize from the training examples into neighboring regions of the feature space. Various classifiers generalize in different ways. The way in which a particular type of classifier generalizes is called its inductive bias [Mitchell97 p. 39].

Classifier efficiency is measured both in time to train the classifier and in time to classify a particular feature vector, once the classifier is trained. The parameters for measuring such efficiency are the number of training examples *E*, the number of features *F*, and the number of possible classes *C*.

The classifiers described in this section are:

- **Naïve Bayes** – a relatively simple learning algorithm based primarily on counting and voting. Training is quite simple and the algorithm can learn a large number of patterns. This algorithm is primarily useful when many features must work together to build confidence in a conclusion. The training time is *O(EF)*, and the classification time is *O(F)*.
- **Decision tree** – larger number of features. All features must be nominal. Decision trees are little more complicated to train and implement than Parzen windows. Decision trees can learn arbitrarily complicated decision patterns. Training time is $O(EF)$ and classification time is $O(F)$.
- **Nearest neighbor** – any number of features, which must be numeric or distance-comparable. Can learn arbitrarily complicated decision patterns. Training time is zero because all training examples are saved. Classification time is $O(EF)$, which is a problem.
- **Clustering** – this is not truly a trainable classifier, but it can be used to reduce the classification time of a nearest neighbor classifier as well as serve other purposes.
- **Linear Classifiers –** these can handle any number of numeric features. They can only be trained on problems where classes are linearly separable. Training time is variable because the algorithm iterates until it converges or reaches some maximum number of iterations. Classification time is $O(FC)$.

A3.1a – Naïve Bayes

Prerequisites

Arrays, Classifiers(A3)

Overview

This is a relatively simple classifier that an assemble evidence from a variety of features to produce a classification. The simplest form of this algorithm uses all of the known features and thus can be a little slow to classify. The algorithm is easy to implement and very easy to train. The training time is *O(EF)*, and the classification time is *O(F)*. This algorithm can be augmented with a feature selection step that will improve classification times.

Description

The foundation of this algorithm is Bayes' Law for computing conditional probabilities. The idea is to compute the probability of each class C_i given a feature vector *V*. The class with the highest probability is the final classification. The approach begins with using Bayes' Law to compute the probability of C_i given a single feature V_f. Bayes' Law is as follows.

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}$$

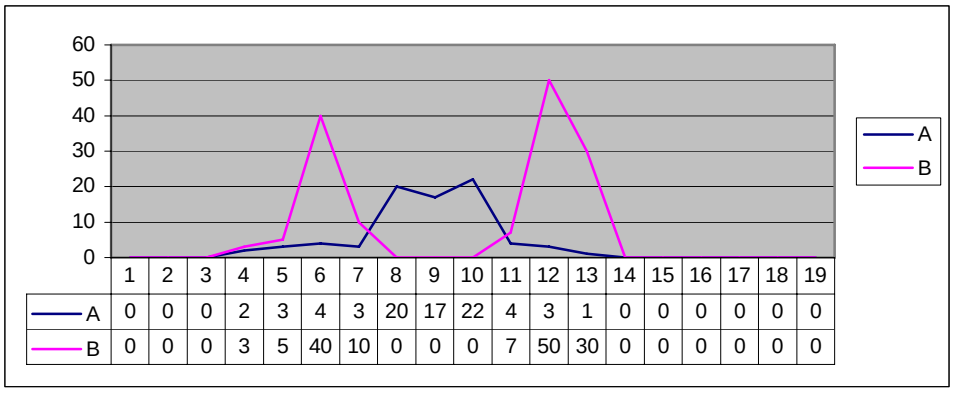
Conditional probability *P(A|B)* is the probability that *A* is true given that *B* is known to be true. In our classification problem *B* would be the facts that we know from our feature vector *V* and *A* would be one of the classes to be classified. What we want to know for each class C_i, what is the probability $P(C_i|V)$, or what is the probability of C_i being the correct classification given the feature vector *V*.

The first step that we will need is to compute discrete probabilities. Though it is possible to compute exact probabilities for some problems using calculus, we will rarely use that approach in user interfaces. To use calculus to solve the problem we need continuous functions for the various probabilities. This requires more understanding of the problem and its feature than most users have. For our implementation will convert the problem to discrete probabilities and use simple histograms. Our output classes are already discrete. If we have nominal features then we are already prepared. However, many problems have real valued features (Naïve Bayes is rarely used with distance comparable features). To get nominal features from real-valued features we must *quantize* them in to discrete "buckets". The quantization problem and only be briefly discussed here. Once all features are nominal we can compute the various probabilities using Bayes' Law

on each feature. We then combine the probabilities of the various features to produce a final probability for each class.

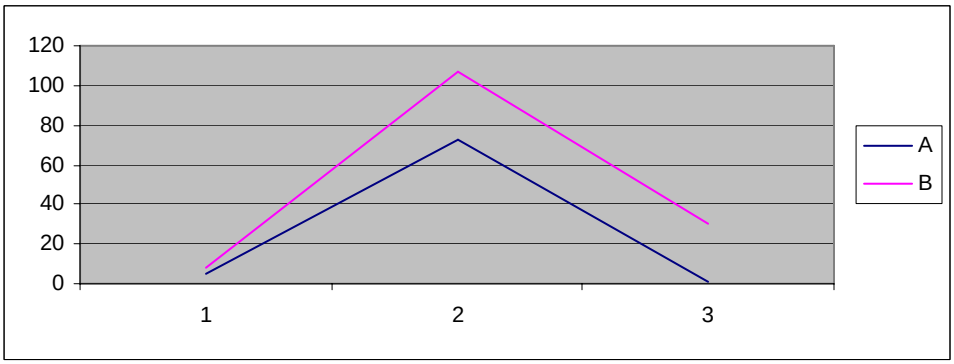
Quantization

The quantization problem occurs when numeric values are incorrectly quantized into nominal features. Consider a one-feature/two-class problem where the distribution of 224 training examples in two classes (A and B) is shown below.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
A	0	0	0	2	3	4	3	20	17	22	4	3	1	0	0	0	0	0	0
B	0	0	0	3	5	40	10	0	0	0	7	50	30	0	0	0	0	0	0

Based on this data, values 1-3 and 14-19 will be undecided. Values 4-7 and 11-13 will be class B and values 8-10 will be class A. Suppose, however, to keep the size of our classifier small, we quantized this data into three values. The resulting distribution is shown below.



With this quantization, all values now vote for class B no matter what. The valley in class B where class A resides has been averaged away.

If we quantized to the other extreme with 300 possible values, our training set of 224 examples will guarantee that 76 of the feature values are unsampled and if there are multiple samples per value, an even larger percentage of the feature space will be unaccounted for. The simplest quantization is to pick a "bin" size such that there are many samples per bin, but there is still sufficient information to separate the classes. By quantizing all real-valued features we now have all nominal features that can be used with simple counting techniques to compute probabilities.

Feature/Class Probabilities

We first use Bayes' Law to compute the probability of a class C_i given a single discrete feature value F_v.

$$P(C_i \mid F_v) = \frac{P(F_v \mid C_i)P(C_i)}{P(F_v)}$$

Remember that our goal is to find the class that has the highest probability. What we want to do is compare $P(C_1|F_v) > P(C_2|F_v)$. Using Bayes' Law the comparison would be:

$$\frac{P(F_v \mid C_1)P(C_1)}{P(F_v)} > \frac{P(F_v \mid C_2)P(C_2)}{P(F_v)}$$

Notice that $P(F_v)$ appears in the denominator on both sides of the comparison. Because $P(F_v)$ is positive, we can multiply it times both sides of the inequality to produce:

$$P(F_v \mid C_1)P(C_1) > P(F_v \mid C_2)P(C_2)$$

This forms our classification test for a two class case. For multiple classes we simply compute the max of all of the classes.

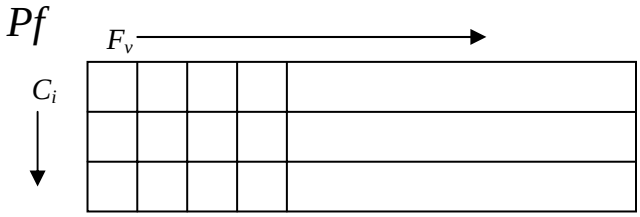
Computing the probability of a class $P(C_i)$ is simple. We count the number of instances of each class in our training set and divide by the number of items in the training set. The purpose of this term in computing the probability is to account for classes that occur very infrequently. For example, when trying to diagnose a disease (class) from a set of symptoms and test results (features) we would want very strong support from the feature before we published a diagnosis of smallpox because smallpox is almost completely eradicated.

Sometimes the class probability term can cause problems. Suppose that we are trying to create a laser pointer tracker using a camera. We need a classifier that will separate laser spots from normal pixels. Many images will have a million or more pixels with only 3-5 of those pixels belonging to the laser spot. This means that the probability of a laser spot as compared to a non-laser spot will be 5 in a million. It will require a lot of evidence from the features to overcome that

hurdle. In many cases we are using a classifier to find something that is relatively rare in the training set. In those cases, discarding the class probability term will produce a better classifier for the task even though it is not a true probability. Our classification test becomes:

$$P(F_v \mid C_1) > P(F_v \mid C_2)$$

Regardless of whether we use class probabilities, the key term for classification is $P(F_v|C_i)$. This is computed from the training data in a simple two dimensional matrix *Pf*:



For each training example we take the output class and the discrete value of V_f to subscript into the array and increment the count. We then divide each row by the number of training examples for that class to get the probabilities. For a given feature value F_v, we index the appropriate column in the array and take the class with the largest probability.

Combined Probabilities

The discussion above will give us the probability of a class given the value of one feature. What we need is the probability given all of the features. The first step is to compute the matrix for every feature. We then need to combine these feature probability into a single composite probability for each class. The approach is to "and" all of the feature probabilities together by multiplying them. This computes the probability of the condition where all of the feature values in the input correspond to a given class. The simple multiplication technique is based on the "naïve" assumption that all features are independent. This is generally not true. For example the weight and height of a person are not independent features. Taller people generally weigh more than shorter people. However, the "naïve" assumption works just fine in many cases. Hence we combine evidence from all of the feature using the formula

$$P(C_i \mid V) = \prod_f P(C_i \mid V_f)$$

In many cases the probabilities can be small and the number of features large. When this occurs, the product of the probabilities can overflow the floating point representations of most machines. To resolve this case we use the sum of the

logarithms of the probabilities rather than the product of the probabilities. This does not change the comparison. We can classify using the following formula:

$$\arg\max_{C_i} \left(\sum_f \log(P(C_i \mid V_f) \right)$$

For efficiency we replace the probabilities in the matrices with the logarithm of the probabilities. Classification is simply a matter of summing the matrix values for each feature/class and selecting the class with the largest value. On occasion some of the probability values will be zero in some bins. The value of log(0) is negative infinity. To resolve this problem we initialize *Pf* to 1 rather than zero before counting. This only very slightly distorts the results and prevents numeric problems.

Implementation

As with all classifiers, there are two algorithms: the training algorithm that produces matrices from the training set and the classification algorithm that takes a feature vector and returns a class. The code below includes the class probabilities. They are easily removed if not desired. It is assumed that the features have already been converted to discrete integer values where needed.

```
class Example
{    int theClass;      // the class that should be the result of this example
     int features[];    // features for the example
}
class NaiveBayes
{    int nClasses;      // number of output classes
     int nFeatures;     // number of features per example
     int nValues;       // maximum number of discrete values per feature
     float logClassProb [nClasses];
          // logarithm of the probability of each class.
     float logFeatureProb [nFeatures,nClasses,nValues];
          // contains the probabilities trained from the example set
}
NaiveBayes train( Example examples[nExamples], NaiveBayes data)
{    // Assumes that data already contains nClasses, nFeatures
     // and nValues. Will generate logClassProb and logFeatureProb.
     classProb (examples,data);
     featureProb (examples, data);
     return data;
}
classProb (Example examples[nExamples], NaiveBayes data)
{    for (int c=0;c<data.nClasses;c++)
          data.logClassProb [c]=1;
     foreach ( e in examples )
          data.logClassProb [e.theClass] += 1;
     for (int c=0;c<data.nClasses;c++)
          data.logClassProb [c] =
               log(data.logClassProb [c]/(nExamples+nClasses));
}
```

```
featureProb( Example examples[nExamples], NaiveBayes data)
{
    for (int f=0;f<data.nFeatures; f++)
        for (int c=0;c<data.nClasses; c++)
            for (int v=0;v<data.nValues; v++)
                logFeatureProb [f,c,v]=1;
    foreach (e in examples )
    {   for (int f=0;f<data.nFeatures;f++)
            logFeatureProb [f, e.theClass, data.features[f] ];
    }
    int nCounts=nExamples+
        data.nFeatures*data.nClasses*data.nValues;
    for (int f=0;f<data.nFeatures; f++)
        for (int c=0;c<data.nClasses; c++)
            for (int v=0;v<data.nValues; v++)
                data.logFeatureProb [f,c,v]=
                    log(data.logFeatureProb [f,c,v]/(nSamples));
}
```

The train() algorithm above will load a data structure with the computed probabilities from the training data this can then be used to classify new feature vectors.

```
int classify( int features[nFeatures], NaiveBayes data )
{
    float classLogSum[nClasses];
    for (int c=0;c<nClasses;c++)
        classLogSum[c]=0;
    for (int f=0;f<nFeatures;f++)
    {
        for (int c=0;c<nClasses;c++)
            classLogSum[c]+=data.logFeatureProb[f,c,features[f];
    }
    float maxSum=classLogSum[0];
    int maxClass=0;
    for (int c=1;c<nClasses;c++)
        if (classLogSum[c]>maxSum)
        {   maxSum=classLogSum[c];
            maxClass=c;
        }
    return c;
}
```

A3.1b - Decision tree

Prerequisites

Classifier features (A3)

Overview

Decision trees most naturally use nominal features. There are decision tree variations that allow for numeric and distance comparable features. They are somewhat sensitive to quantization. The time of classification is linear in the number features. The size of the classifier depends upon the complexity of the classification function to be learned. A decision tree is never larger than the feature space (product of the ranges of the features) and is frequently very much smaller. This classifier learns the same set of decision functions as Parzen windows. However, the memory requirement is typically much smaller and the number of required training examples is smaller because decision trees can generalize across ranges of features.

Description

A decision tree works by selecting a feature and using that feature to divide a set of training examples into groups based on the value of that feature. Each such group then recursively forms a subtree. There are two primary strategies for selecting features and doing the decomposition. The first involves strictly nominal features and is the easiest to program. However, there are many cases

when numeric features are important. The use of numeric features will be discussed after the nominal features algorithm.

To illustrate how the tree is constructed using nominal features, consider the following set of training data.

1 - { tall, heavy, 3, red } -> good
2 - { tall, light, 2, red } -> good
3 - { medium, veryHeavy, 1, green} ->OK
4 - { short, light, 3, green } -> OK
5 - { medium, heavy, 3, red } -> good
6 - { medium, light, 1, red} -> bad
7 - { medium, heavy, 1, red} -> bad

Using this data we can build the top node of our tree using the first feature.

short
4 - { **short**, light, 3, green } -> OK
medium
3 - { **medium**, veryHeavy, 1, green} ->OK
5 - { **medium**, heavy, 3, red } -> good
6 - { **medium**, light, 1, red} -> bad
7 - { **medium**, heavy, 1, red} -> bad

tall
1 - { **tall**, heavy, 3, red } -> good
2 - { **tall**, light, 2, red } -> good

Note that the tree under "short" contains only the class "OK" and the tree under "tall" contains only the class "good." These subsets of the training set need no further refinement. Such subsets are consider pure because they only contain one class. They can simply report their class. The training examples under "medium" contain all three classes and must be subdivided further. We can continue breaking down our tree using the second feature as shown below.

short ->OK
medium
light
6 - { medium, **light**, 1, red} -> bad
heavy
5 - { medium, **heavy**, 3, red } -> good

7 - { medium, **heavy**, 1, red} -> bad
veryHeavy
3 - { medium, **veryHeavy**, 1, green} ->OK
tall -> good

We can continue this process using the third feature.

short ->OK
medium
light ->bad
heavy
1
7 - { medium, heavy, **1**, red} -> bad
2
3
5 - { medium, heavy, **3**, red } -> good
veryHeavy -> OK
tall -> good

We can now produce our final decision tree for the data.

short ->OK
medium
light ->bad
heavy
1 -> bad
2 -> ?
3 -> good
veryHeavy -> OK
tall -> good

Classification using a nominal feature decision tree

Each node of a decision tree has the following structure.

```
class DecisionTree
{    int featureIndex;
          // index of the feature for subdividing the tree
          // if this is –1 then there are no subtrees and theClass should be
          // reported
     DecisionTree subtrees[ ];
          // array of decision trees indexed by the values of the
          // feature indicated by featureIndex
     int theClass;
          // if there are no subtrees for this node, then this is the class that
          // corresponds to this part of the tree.
}
```

Using this structure, our classifier algorithm is as follows.

```
int classify(nominalFeature features[n], DecisionTree tree)
{    if (tree.featureIndex== -1)
     {    // this is a leaf of the tree
          return tree.theClass;
     }
     else
     {    featureValue = features[tree.featureIndex];
          return classify( features, tree.subtrees[featureValue]);
     }
}
```

Building the decision tree

Building the decision tree is a recursive process. At each step we select a feature to use to partition our trainingSet.

```
decisionTree buildTree( example trainingSet[n])
{    decisionTree result;

     if ( hasOnlyOneClass(trainingSet) )
     {    result.featureIndex = -1;
          result.theClass = trainingSet[0].class;
          return result;
     }
     else
     {
          int feature = pickAFeature(trainingSet);
          if (feature is no feature)
          {    result.featureIndex = -1;
               result.theClass = undecided;
               return result;
          }
          result.featureIndex=feature;
          int nVals = howManyValuesForFeature(feature);
          result.subTrees = new decisionTree[nVals];
          for each value V for feature
          {    result.subTrees[V] = buildTree(
                         matchingExamples( trainingSet, feature, V));
          }
          return result;
     }
}

example [ ] matchingExamples( example trainingSet[n], int feature,
nominalFeature Val)
{
     return all examples from trainingSet for which the specified feature
has the
          specified value
}
```

The key to the algorithm is the order in which we pick features to break down the training set. This is controlled by how we implement **pickAFeature**. The simplest implementation is to use the first feature that has more than one value in the training set. Selecting such a feature guarantees that each invocation of **buildTree** will have a smaller training set to work on and eventually the process will converge. We may in some cases reach a training set for which all feature vectors are the same, but the classes are not. We can resolve this by voting,

(which helps with noisy data) or by reporting no class, which will leave the result undetermined for those cases.

Simply selecting the first usable feature is not a very good approach. It leads to larger tree sizes than necessary and less effective generalization from training examples. What we want is to select a feature that best divides the training set along the categories that we are ultimately trying to classify. The classic algorithm for decision feature selection is ID3 [Quinlan93].

Evaluating which feature to use is based on a measure of the "impurity" of a set of training examples. A set of training examples is pure (impurity=0) if all training examples in the set have the same class. If there is some mixture of classes in the training set, then that set is considered more impure. A node of a decision tree should use the feature that will divide its training set into subsets that will reduce impurity as much as possible.

We can compute the impurity of a set of training examples in a variety of ways [Duda01 p.399]. The simplest is $imp(T) = 1 - \max_c(P(T_c))$ where T is the training set, $P(T_c)$ is the fraction of *T* that are in class *C*.

Given some feature *F* we can compute the impurity after splitting the training set on *F* as follows.

$$imp(F) = \sum_{v=1}^{values(F)} P_v imp(T_v)$$

Where P_v is the fraction of the training examples that have value *V* for feature *F* and T_v is the set of training examples that have value *V* for feature *F*.

This measure of impurity is somewhat of a problem because it favors features that split many ways over features that split a few. Where there are many splits, the probability of each split is somewhat lower than if there are just a few ways to split. This makes $imp(T_v)$ lower for features that split many ways. We can compensate for this by using the scaling formula

$$imp_g(F) = \frac{imp(F)}{-\sum_{v=1}^{values(F)} P_v \log_2 P_v}$$

To pick a feature we compute $imp_g(F)$ for each feature and pick the feature that produces the least impurity.

```
int pickAFeature( example trainingSet[n])
{    int pickedFeature=-1;
     double pickedImpurity = 1.0;
     for each feature F
     {    int nVals = howManyValuesForFeature(F)
          int counts[ ][ ]=new int[nVals, nClasses];
          set counts to zero

          for (i=0; i<n; i++)
          {    int val = trainingSet[i].features[F];
               counts[val, trainingSet[i].class]++;
          }

          double impF=0.0;
          double scale=0.0;
          for (v = 0; v<nVals; v++)
          {    int countV = 0;
               int max=0;
               for (c=0; c<nClasses; c++)
               {    countV=countV+counts[v,c];
                    if (counts[v,c]>max)
                    {    max=counts[v,c];  }
               }
               double impTv= 1.0-(max/countV);
               double Pv=countV/n;
               impF=impF+Pv*impTv;
               scale=scale-Pv*log2(Pv);
          }
          impF=impF/scale;

          if (impF<pickedImpurity)
          {    pickedImpurity = impF;
               pickedFeature = F;
          }
     }

     return pickedFeature;
}
```

Decision trees with numeric features

Decision trees with nominal features build a tree where each node is indexed by the value of some feature. This is not acceptable when there are numeric features because there are potentially an infinite number of numeric values for a feature. In describing decision trees with numeric features, we will assume that all features are numeric. Algorithms for mixing numeric and nominal features are easily created once the two types are independently understood.

The general approach for creating decision tree nodes for numeric features is to select a feature *F* and a value *V* for that feature. We then can divide our training set into two subsets *L* and *G* (less and greater) such that

$$L = \text{all training examples } T_i \text{ such that } T_i.F < V$$

$$G = \text{all training examples } T_i \text{ such that } T_i.F \geq V$$

The class for defining a decision tree can be modified in the following way.

```
class DecisionTree
{    int featureIndex;
          // index of the feature for subdividing the tree
          // if this is –1 then there are no subtrees and theClass should be
          // reported
     float V
          // the value to be used in partitioning the tree according to the
feature
     DecisionTree less;
          // A decision tree for all cases where the feature is less than V
     DecisionTree greater;
          // A decision tree for all cases where the feature is greater than
or equal to V
     int theClass;
          // if there are no subtrees for this node, then this is the class that
          // corresponds to this part of the tree.
}
```

The classification algorithm is quite simple. We simply walk the tree comparing feature values as we go until we reach a leaf. We then return the class for that leaf.

```
int classify(float features[n], DecisionTree tree)
{    if (tree.featureIndex== -1)
     {    // this is a leaf of the tree
          return tree.theClass;
     }
     else if (features[tree.featureIndex]<tree.V)
     {    return classify(features,tree.less); }
     else
     {    return classify(features,tree.greater); }
}
```

The challenge lies in building the tree. We not only must select a feature but we must select a value for that feature to use as our split value to divide the training set. A very simple approach is to select the mean for each feature. We can then compute the new impurity for each feature division and select the feature that produces the least impurity. Using the mean is easy to compute and tends to divide the training set into roughly equal halves. This means that the decision time will be at worst *nlog(n)* where *n* is the number of training examples. The problem is that the mean has little to do with where a good place to divide the feature so as to produce a good classification. Good feature choices will be ignored because their appropriate split point is not near the mean.

The best way to produce such a decision tree is to take each feature and each value in that feature and pick the one that produces the lowest impurity.

```
decisionTree buildTree( example trainingSet[n])
{    decisionTree result;

     if ( hasOnlyOneClass(trainingSet) )
     {    result.featureIndex = -1;
          result.theClass = trainingSet[0].class;
          return result;
     }
     else
     {    int bestFeature = -1;
          float leastImpurity = infinity;
          float bestValue = 0.0;
          example bestL[];
          example bestG[];
          for each feature F
          {    for each value V for feature F
               {    example L[]=empty;
                    example G[]=empty;
                    for each training example T
                    {    if (T[F]< V)
                         {    L.add(T); }
                         else
                         {    G.add(T); }
                    }
                    float splitImpurity = impurity(L) + impurity(G);
                    if (splitImpurity < leastImpurity)
                    {    leastImpurity=splitImpurity;
                         bestFeature=F;
                         bestValue=V;
                         bestL=L;
                         bestG=G;
                    }
               }
          }
          result.featureIndex = bestFeature;
          result.V = bestValue;
          result.less=buildTree(bestL);
          result.greater=buildTree(bestG);
          return result;
     }
}
```

This algorithm can be very slow to build a tree if there are many features and many training examples. This speed can be improved in a number of ways. The

simplest improvement is to sample the training examples rather than use all of them. We do not need an exact split value. An approximately good one will do because subsequent layers of the decision tree can correct any inaccuracies. If at each buildTree we select 100 samples from the training set and consider only their feature values then we have a fixed bound on the number values to consider. When we actually produce the partitioned sets for building lower nodes in the tree and for calculating impurity, we still use the full training set. This approach can be significantly faster (depending on the sampling size used) and will produce very close to the same quality of decision. However, there will frequently be multiple nodes for a feature as multiple layers are used to find the best split point.

When considering a feature for possible split, it can help to first sort the training examples by that feature. The values can then be considered in ascending order. This has several advantages. 1) It is easy to ignore duplicate values. 2) Using the mean between two successive values will in general produce a better split in the presence of sparse data than just using either value. 3) When moving from training example T_i *to* T_{i+1} in the sorted list it is simple to increment the counts for the various classes in calculating impurity rather than retest all of the training examples. Because the list is already sorted, only the training example being considered changes subsets with the new value. The two sets L and R are easily represented as two ranges in the training set array rather than building new structures. The sorting approach is more complicated and does not improve the quality of the resulting classifier, but it can significantly increase the speed of the classifier.

Decision trees with distance comparable features

We can also handle distance-comparable features by using exemplar partitions. If we want to divide a feature into K partitions, we can get K exemplars (one for each partition). We then partition the feature values based on which of the K exemplars they are closest to. Selecting appropriate exemplars is a problem. For ideas on how to deal with this see the section on clustering (xx). One possible strategy is to pick one training example from the set for each class represented in the set. Use the feature value of each such example as an exemplar. This will tend to cluster similar classes together, but may still produce less efficient trees.

A3.1c - Nearest neighbor

Prerequisites

Classifier features (A3), Vectors (A1.1), Comparing Vectors(A1.1e)

Overview

The nearest neighbor classifier works by storing all of the examples from the training set and comparing a new vector to be classified against all of the stored vectors. It uses any of the vector distance metrics (A1.1e) for comparison. It finds the closest neighbor (least distance) out of the set of examples and returns the class of that example. This algorithm can learn decision functions of any complexity. Its classification time is $O(EF)$, which tends to make the classification slow. This algorithm generally requires features that are distance comparable or numeric. There are variations of the algorithm that can use purely nominal features.

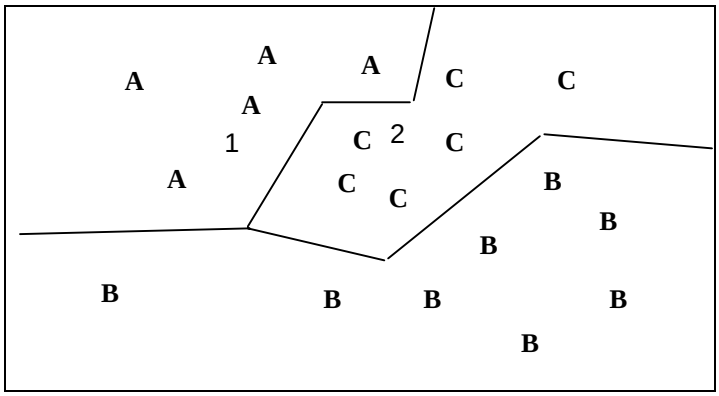
Description

The nearest neighbor classifier is conceptually quite simple. We store the training set and then compare a feature vector to be classified against all of the examples in the training set, and return the one that is the nearest. Any of the vector distance functions described in A1.1e can be used for the distance.

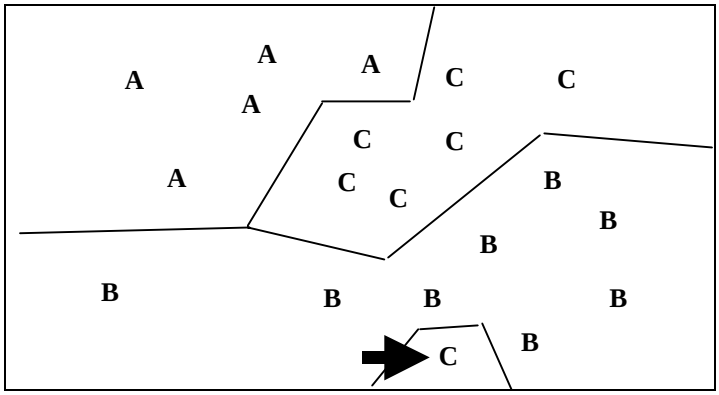
Implementation

```
class nearestNeighbor( Feature features[n], example trainingSet[nEx] )
{    example E=trainingSet[0];
     double minDist = distance(features, E.features);
     class curClass = E.class;
     for (int i=1; i<nEx; i++)
     {    double tmp = distance(features,trainingSet[i].features);
          if (tmp<minDist)
          {    minDist = tmp;
               curClass = trainingSet[i].class;
          }
     }
     return curClass;
}
```

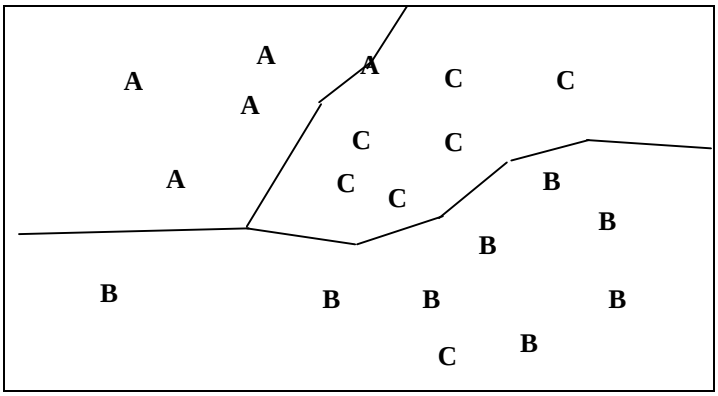
The following picture shows roughly how nearest neighbor works with two features and three classes (A,B,C). The nearest neighbor surfaces are slightly more complicated than what is shown, but not much. Point 1 is closer to an A than anything else and point 2 is closer to a C.



This simple algorithm is somewhat sensitive to noise. That is, if some of the training examples are incorrect, this algorithm will learn those incorrect data. Suppose we had an erroneous C as shown below. This would create a small pocket of C class space in the middle of B.



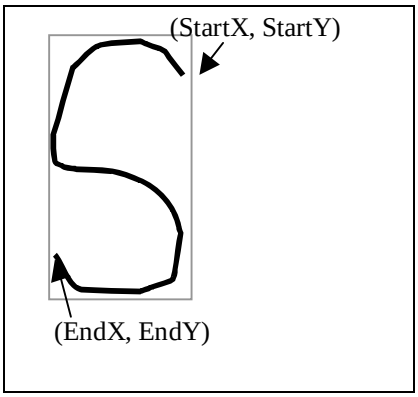
The noise problem can be overcome by finding the K nearest neighbors of a feature vector and having them vote on which class would be right. In our example above, using a K of three would eliminate the effect of the spurious C but would also push back some of the corners as shown below.



The averaging effect of using K larger than 1 can be overcome with more training data. Another approach is to weight the influence of each of the K examples by their distance from the point being classified. Thus closer examples have more influence than those farther away.

Normalizing features

One of the problems with simple nearest neighbor is that not all features have the same range. Take, for example, a simple character classifier that uses the features StartX, StartY, EndX, EndY and StrokeLength.



Generally strokes for recognition are scaled to the unit square before features are computed so that the size of the character will not matter. In such a case, StartX, StartY, EndX and EndY will lie in the range 0.0->1.0. However StrokeLength can very much more widely. In the example of an “S”, the StrokeLength will be close to 4.0. When computing distances between feature vectors, the

StrokeLength will have much greater distances and therefore will separate objects much more widely than the other four features. We would like to eliminate such bias.

One simple normalization is to compute maximum and minimum values for each feature and then normalize the data values into classifier values.

$$classVal = \frac{dataVal}{\max_f - \min_f}$$

where $\max_f$ and $\min_f$ are the maximum and minimum value for feature *f*.

This method is susceptible to outliers. If there were one really extreme example in the training set, it would bias the range. If our values follow a normal distribution (which many do) then 95% of all values lie within 4 times the standard deviation. A more robust normalization then is:

$$classVal = \frac{dataVal}{4\sigma_f}$$

where σ_f is the standard deviation of all values of the feature *f*.

Distance-comparable features

Both of these normalizations work for numeric features but not for distance-comparable features. Such features do not have maximum values or standard deviations. However, we can compare all feature values to each other (which is $O(N^2)$ in the number of values) and compute an average distance among values. This average distance is roughly comparable to 2 times the standard deviation. Thus we can normalize distance-comparable features by:

$$classVal = \frac{dataVal}{2ad_f}$$

where ad_f is the average distance between values for feature *f*.

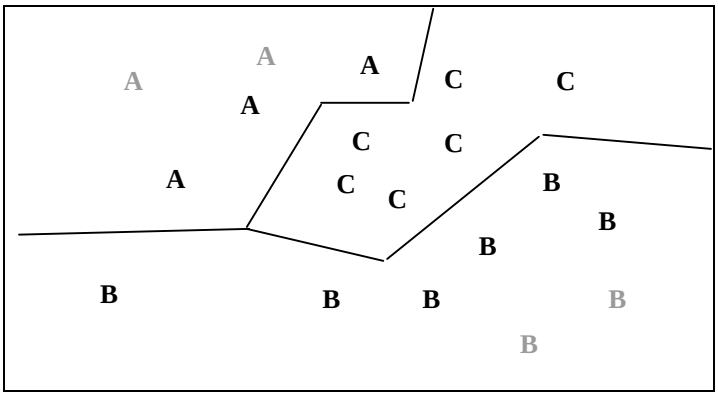
Nominal features

Nominal features do not have distances at all. One very simple distance metric is to return 0 if two values are the same and 1 if they are different. There are other mechanisms for computing distances for nominal features based on how well they separate classes [WILSON xx].

These normalizations are only intended to bring values into similar ranges. Exact normalizations are not important. We only want to prevent wild distortions.

Pruning examples

One of the big problems with nearest neighbor classifiers is that large numbers of training examples can slow down classification time and take up a lot of space. Both of these factors can be improved by pruning out examples that do not change the outcome of the classifier. In the following 2D classifier, the gray examples could be removed without changing the result.



A variety of algorithms have been developed for performing such pruning. See [Wilson xx] for an analysis of such techniques. One simple technique is trial deletion. We remove a training example from the set and see if it is still classified correctly when its own example is missing. If it is still classified correctly, then remove it, otherwise keep it. Another approach is trial insertion. We start with an empty set of examples. One by one we test each new training example to see if it will classify correctly. If it does not classify correctly, then we add it to the set of examples. We repeat this process until no new training examples are added.

A3.1d – Clustering – K means

Prerequisites

Vector distances(xx), Nearest Neighbor Classifier(xx)

Overview

In many situations, the amount of training data or the number of features is too large or expensive to use directly. One of the ways to deal with this is to break the data into a set of *K* clusters, with each cluster being represented by a mean vector. For this kind of clustering all features must be numeric. A feature vector

can be classified into one of the *K* clusters by finding which of the K-means has the smallest distance to the feature vector. The key problem is discovering an appropriate set of mean vectors.

Description

The mean vector of a set of vectors is determined by computing the average value for each feature.

$$mean_f = \frac{\sum_{i=1}^{N} items[i]_f}{N}$$

where *f* is a feature, items is an array of vectors and *N* is the number of vectors.

We can use mean vectors as a way to cluster data using a distance metric. If we are initially given a set of *K* vectors, we can cluster a set of vectors according to which of the *K* vectors each one is nearest to. We can then compute a new set of *K* vectors by taking the means of each cluster. We repeat this until the membership in the clusters does not change. We then can retain the *K* mean vectors as the representation of each of the classes.

The ability of this algorithm to produce good clusters depends upon the choice of the original *K* vectors. One way to produce a good set of *K* vectors is to start with a single mean vector for the entire set. We can divide the cluster by computing a new variant vector where each feature is a very small value different from the original vector. With these two vectors that are very close, but not identical, we can produce two new clusters. These new clusters will converge to their own means. If we have *K* clusters and we want *K*+1 clusters, we can pick the largest cluster, compute a new variant of its mean and then do K-means again until the clusters converge. Proceeding in this way, we can eventually produce any number of clusters.

Nearest neighbor classifiers suffer from poor classification speed because of the need to compute the distance to each member of the training set. We can improve this performance by replacing the training data with the mean vector for each class. This, however, will produce misclassifications in cases where the decision space is more complex than a simple distance. We can handle this by computing the *K* means starting with the means of each classifier. Any cluster that is impure (has a mixed set of classes in that cluster) is divided into clusters based on the classes in that cluster. Mean vectors are computed for the new clusters and the K-means algorithm is repeated. This splitting process proceeds until every cluster contains training data for only one class. In the worst case there will be one

cluster for every training item, but in many cases there will be far fewer items than in the original training data.

Vector quantization

Clustering is also used to produce a "vector quantization" that reduces the feature vector to one of a set of sample classes. The approach is to compute a set of *K* mean vectors that cluster the vector data into *K* classes. We can then take a new vector *V*, compute which of the *K* means is nearest to it and then replace that vector with the index of the nearest mean. This in essence quantizes *V* into one of a small set of possibilities. This quantized value can then be used in later stages of a recognition process. This is frequently used to convert a series of vector features into a sequence of quantized values that are then fed to a sequence classifier.

A3.1e - Linear classifiers

Prerequisites

Vectors(xx)

Overview

Linear classifiers use a simple linear function for each class. To classify a point we compute the function for each class and take the class that produces the largest result. To produce the desired linear function for each class we use an iterative gradient descent process. Linear classifiers cannot always learn every decision surface. In some cases the decisions are not linear. This problem can be accommodated by computing additional non-linear features (see xx).

Description

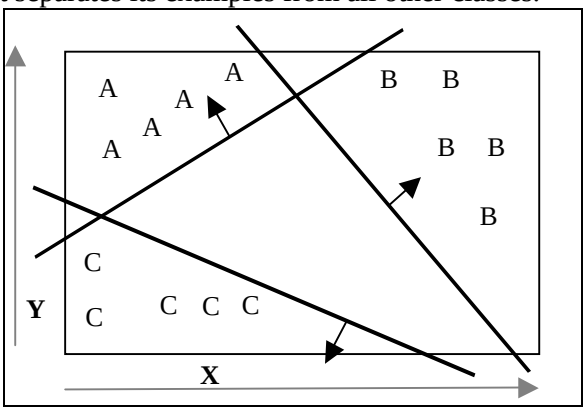
For our linear functions, we use the same homogenous formulation of a linear function as discussed in the section on linear approximation (xx). That is each feature vector acquires an additional coordinate 1, to accommodate a constant offset in the linear equation. Thus the linear function for a three feature system becomes:

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} \bullet \begin{bmatrix} p \\ q \\ r \\ s \end{bmatrix} = d$$

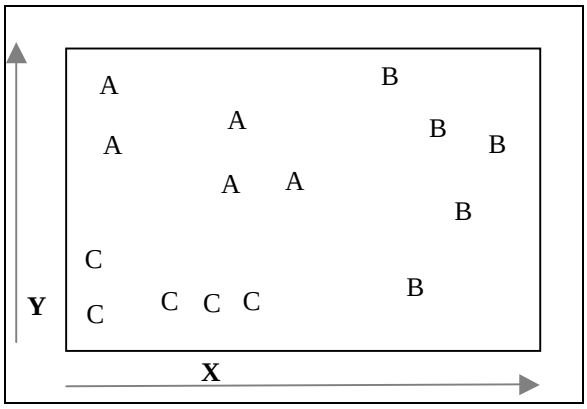
where *p*-*s* are the coefficients and *d* is the desired outcome for the function. The general form of our function is $\vec{V}_T \bullet \vec{C}_c = d_{T,c}$, where $\vec{V}_T$ is the homogeneous

feature vector for training example *T*, $\vec{C}_c$ is the coefficient vector for class *c*, and $d_{T,c}$ is the desired outcome for training example *T* for class *c*.

Linear classifiers can only solve problems that are linearly-separable. That is when a linear hyperplane clearly divides classes from each other. Consider the following problem which has a two-dimensional feature space and three classes A, B and C. If the training examples fall into the feature space as shown in the following diagram, then each class can have its own hyperplane (line in 2 dimensions) that separates its examples from all other classes.



The arrows in the above diagram show the direction relative to the hyperplane where positive values of *d* (the result of the dot product) are found. We can find the class A by using the hyperplane in the upper left and run any feature vector through that equation. If the result is positive, then the feature vector belongs to class A. The same can be done with the other two hyperplanes. Our classification algorithm then is to run a feature vector through the hyperplane for each class and choose the class whose hyperplane returns the largest value. Many problems are not linearly separable. Take the following distribution of our same three classes. Note that although the classes are clearly clumped together there is no straight line that will separate class A from all other classes. Remember that although our diagrams are 2 dimensional, the classifier problem is N-dimensional depending on the number of features.



Although there is no line that will separate the classes it is possible that a series of parabolas would to the job. We can accommodate this by replacing our $[X \quad Y]$ feature vector with the 5 dimensional feature vector $[X \quad Y \quad X^2 \quad Y^2 \quad XY]$.

If our classes are linearly separable, then our problem is to come up with an algorithm that will learn the linear coefficients of the necessary hyperplanes from the training data. The perceptron algorithm works by randomly initializing our hyperplanes and then evaluating them against each element of our training set. If a hyperplane gives the wrong answer we slightly modify the coefficients so that the hyperplane will give a more correct answer. We continue doing this with repeated passes against all of the training data. If the classes are linearly separable the coefficients will converge. If they are not then we must detect that no progress is being made and stop.

We start our algorithm by initializing all of the coefficients in all of our $\vec{C}_c$ vectors to some small random number. For each training example *T* we compute d_c. Given that $T.class$ is the correct class for example *T*, then $d_{T.class}$ should be the largest outcome. If this is so, then the coefficients are left unchanged. If $d_{T.class}$ is not the largest outcome, then we must modify the coefficients. Suppose that $d_{\max}$ is the largest outcome for a given training example. We want to modify $\vec{C}_{T.class}$ so that it will produce a higher outcome that is larger than $d_{\max}$. We also want to modify the coefficients of all classes with larger outcomes than *T.class* so

that they will produce lower outcomes that are less than $d_{T.class}$. We do this with the following perceptron training rule [Mitchell97 p88]

$$\vec{C}_c \leftarrow \vec{C}_c + \eta(t_c - d_c)\vec{V}_T$$

where t_c is the target outcome for class *c* and η is the learning rate. The learning rate η must be between 0.0 and 1.0 and is usually relatively small (0.1). The learning rate prevents extreme movements of the coefficients that overshoot the desired values.

For class *T.class* we modify the coefficients to be

$$\vec{C}_{T.class} = \vec{C}_{T.class} + \eta(d_{\max} + 1 - d_{T.class})\vec{V}_T$$

This will move the outcome for *T.class* towards a value larger than $d_{\max}$. For all classes *L* such that $d_L \geq d_{T.class}$ we modify the coefficients according to the rule

$$\vec{C}_L = \vec{C}_L + \eta(d_{T.class} - 1 - d_L)\vec{V}_T$$

This will move the outcomes for these classes towards a value smaller than $d_{T.class}$. For all classes *S* such that $d_S < d_{T.class}$ we leave $\vec{C}_S$ unchanged because those functions produce correct outcomes.

This algorithm will converge if η is sufficiently small and linear functions can correctly separate the classes. If linear functions are not sufficient, then this algorithm will not converge. Nonconvergence is generally detected when some maximum number of iterations has been exceeded.

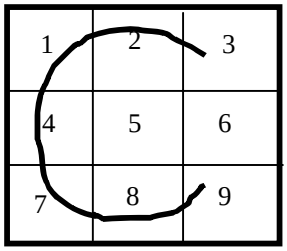
Implementation

```
int classify( double coef[C,F+1], double features[F])
{    int cls = -1;
     double dMax = smallest possible value;
     for (int c=0; c<C; c++)
     {    double d = coef[c,F];
          for (int f=0; f<F; f++)
          {    d=d+coef[c,f]*features[f];  }
          if (d>dMax)
          {    dMax = d;
               cls = c;
          }
     }
     return cls;
}

double [nClasses,nFeatures+1] trainLinear( example trainingSet[nEx])
{
     double coef[ ][ ]=new double[nClasses,nFeatures+1];
     for (int c=0; c<nClasses; c++)
     {    for (int f=0; f<=nFeatures; f++)
          {    coef[c,f]=a small positive random number;  }
     }

     for (int iter=0; iter<maximumIterations;  iter++)
     {    int nErrors=0;
          for (int ex=0; ex<nEx; ex++)
          {    nErrors=nErrors+trainExample(trainingSet[ex],coef)
          }
          if (nErrors==0)
          {    return coef;  }
     }
     return coef;
}
```

```
int trainExample(example E, double coef[nClasses, nFeatures+1] )
{    double d[ ]=new double[nClasses];
     for (int c=0; c<nClasses; c++)
     {    d[c]=coef[c,nFeatures];
          for (int f=0; f<nFeatures; f++)
          {    d[c]=d[c]+coef[c,f]*E.features[f];  }
     }

     int errCount=0;
     double L=the desired learning rate;
     double dMax=the smallest possible number;
     double dClass=d[E.class];
     for (int c=0; c<nClasses; c++)
     {    if(c!=E.class && d[c]>=dClass)
          {    double mov = L*(dClass-1-d[c]);
               for (int f=0; f<nFeatures; f++)
               {    coef[c,f]=coef[c,f]+mov*E.features[f];  }
               coef[c,nFeatures]=coef[c,nFeatures]+mov;
               if (d[c]>dMax)
               {    dMax=d[c];  }
               errCount=errCount+1;
          }
     }

     if (errCount>0)
     {    double mov=L*(dMax+1-dClass);
          for (int f=0; f<nFeatures; f++)
          {    coef[E.class,f]=coef[E.class,f]+mov*E.features[f];  }
          coef[E.class,nFeatures]=coef[E.class,nFeatures]+mov;
     }
     return errCount;
}
```

A3.2 - Sequence Classifiers

Sequence classifiers describe objects by a sequence of tokens drawn from some finite alphabet of tokens. Unlike vector classifiers, the number of tokens in the sequence may vary. Generally in sequence classifiers are nominal features with the order carrying much of the information about the object.



In the following discussion of sequence classifiers we can use strings of characters as example feature sequences. We will also use a simple stroke recognizer as an example. As shown in figure xx, we can define nine regions that a stroke of digital ink must pass through and characterize that stroke by the sequence of regions [Teitelman 64]. For example, the character C drawn in figure xx would be encoded as the sequence [3,2,1,4,7,8,9]. In this case the alphabet of the sequence is the digits 1 through 9. Character strings and stroke sequences can serve as examples for our discussion of sequence classifiers.

A3.2a - Simple String Matching

The simplest sequence classifier is simple matching. Suppose, for example, that we wanted to classify sequences into cats and dogs. Our training set would be a set of strings each accompanied by the class that it belongs to. For example:

Chow -> Dog
Collie -> Dog
Cheetah -> Cat
Lion -> Cat
Lurcher -> Dog
Leopard -> Cat
Poodle -> Dog
Puma -> Cat

We can build a similar training set for our character recognizer. We get multiple sequences mapping to the same characters because of irregularities in user input.

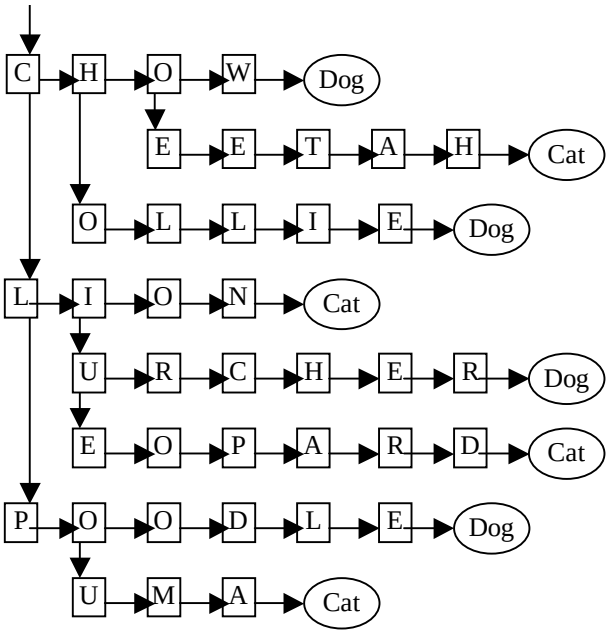
[3,2,1,4,7,8,9] -> C
[6,3,2,1,4,7,8,9] -> C
[1,4,7,4,1,2,3,6,5,6,9,8,7] ->B
[4,7,4,1,2,3,6,5,6,9,8,7] ->B
[1,4,7,4,1,2,3,6,5,6,9,8] ->B
[1,4,7,4,1,2,3,6,5,4,5,6,9,8,7] ->B

The classification algorithm is a simple one. We take the input sequence and compare it against each of the training sequences. If it matches one of them then we return the corresponding class. If it does not match then we return "I don't know."

This algorithm has several problems. The first is that both the classification time and the space requirements are $O(TC)$ where TC is the total number of characters in the training set. The second problem is that the recognition is very brittle. If the user generates an input that is slightly different from any of the training data, the classifier fails. For example entering "lions" will fail, given our training data. The four instances of B in our character classifier are due to minor variations in user input.

A3.2b - Trie classification

We can reduce the classification time problem by using a structure called a Trie. This structure simply combines common prefixes of the sequences and thus eliminates redundant comparisons. This approach is very similar to decision trees except that inputs are always handled in first to last order rather than selecting for minimal impurity. The training data for cats and dogs would produce a tree of the following form. If a string matches the character in a node then the algorithm moves to the right. If there is a mismatch, then we follow the down link. If there is no down link, then the match fails. If the algorithm moves to the right and finds a class and the sequence is at the end, then there is a match and the class is returned. The classification time is $O(AL)$ where A is the number of tokens in the alphabet and L is the length of the sequence being classified. This is much faster than the simple string match, but still has the same brittleness in that if fails for any sequence not in the training set.


C
H
O
W
Dog
E
E
T
A
H
Cat
O
L
L
I
E
Dog
L
I
O
N
Cat
U
R
C
H
E
R
Dog
E
O
P
A
R
D
Cat
P
O
O
D
L
E
Dog
U
M
A
Cat

Implementation

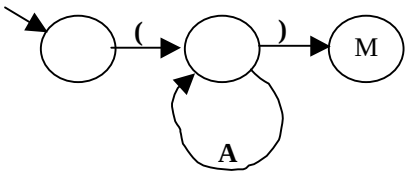
```
class TrieNode
{   int token, theClass;
    TrieNode matchLink, failLink;

    void addTrainingSequence(int seq[L],int cls)
    {   addTrainingSequence(0,seq,cls }

    void addTrainingSequence(int tokenIdx, int seq[L], int cls)
    {   if (tokenIdx>=L)
        {   theClass=cls; }
        else if (seq[tokenIdx]==token)
        {   if (matchLink==null)
            {   matchLink=createMatchChain(tokenIdx+1,seq,cls); }
            else
            {   matchLink.addTrainingSequence(tokenIdx+1,seq,cls); }
        }
        else if (failLink==null)
        {   failLink=createMatchChain(tokenIdx,seq,cls); }
        else
        {   failLink.addTrainingSequence(tokenIdx,seq,cls); }
    }

    TrieNode createMatchChain(int tokenIdx, int seq[L], int cls)
    {   TrieNode tmp = new TrieNode();
        if (tokenIdx>=L)
        {   tmp.theClass=cls;
            tmp.token=-1;
            tmp.matchLink=null;
            tmp.failLink=null;
            return tmp;
        }
        else
        {   tmp.theClass=-1;
            tmp.token=seq[tokenIdx];
            tmp.matchLink=createMatchChain(tokenIdx+1,seq,cls);
            tmp.failLInk=null;
            return tmp;
        }
    }
    int classify(int tokenIdx, int seq[L])
    {   if (tokenIdx>=L)
        {   return theClass; }
        else if (seq[tokenIdx]==theToken)
```

```
            {   if (matchLink==null)
                {   return –1; }
                else
                {   return matchLink.classify(tokenIdx+1,seq); }
            }
            else if (failLink==null)
            {   return –1; }
            else
            {   return failLink.classify(tokenIdx,seq); }
        }
    }
```

The implementation can be optimized for speed if each node has an array of matching links that can be indexed by the token number. This consumes more space, but makes the algorithm $O(L)$, where L is the length of the sequence to be classified.

A3.2c - Finite state machines

The trie is a special case of the more general class of finite state machines. A finite state machine has a start state and zero or more additional states. Each state can have a class that is recognized if a sequence terminates in that state. There are transitions between states. Each transition has one of the alphabet tokens for that transition. The following is a simple state machine that recognizes an opening parenthesis, any number of the letter A, and a closing parenthesis.



Finite state machines have classification time of $O(L)$, where L is the length of the sequence to be classified. There are deterministic and non-deterministic finite state machines. A deterministic machine has at most one transition leaving a state for a given input token. A non-deterministic machine can have many transitions leaving a state for the same input token. A non-deterministic machine may have a classification time of $O(SL)$, where S is the number of states and L is the length of the sequence. This is much worse than for a deterministic machine.

Finite state machines have many useful algorithmic properties including: the ability to convert any non-deterministic machine into a deterministic machine, create a new machine that is the union or intersection of any two finite state machines, and a finite state machine can be reduced to a unique machine for any recognized set of strings [reference to an algorithms book]. Finite state machines are widely used for recognizing a variety of input sequences.

A3.2d - Minimal Edit distance

The introduction of the trie and the finite state machine resolves our classification time problems and our minimal representation problem, but they do not resolve the brittleness problem. All of these classifiers fail if the sequence to be classified is even slightly different from the sequences recognized by the classifier.

What we need is a measure of how close two sequences are to each other. If we have a measure of a distance between two sequences, we can use classification techniques similar to nearest neighbor. This will be much more robust than simple matching. We could compare the two sequences token by token and compute a distance similar to the Manhattan distance [A1.1e]. However, the sequences are frequently not the same length. Simple token, by token comparison is not adequate.

The most common measure of a distance between two sequences is the minimal edit distance. We match two sequences by performing a series of edits on one or both of the sequences until they match. We attach a cost to each editing operation and compute the edit distance as the sum of those costs. The minimal edit distance is the cost for those edits that will produce a match with the least cost.

In most uses, there are three editing operations: substituting one token for another, deleting a token from a sequence, or inserting a token into a sequence. The following are examples of possible edits:

Hat, hat -> substitute “H” for “h”
meet, meat -> substitute “e” for “a”
hat, heat -> either insert an “e” into the first string or delete “e” from the second

The algorithm is driven by three arrays that are indexed by alphabet tokens.

- *sub*[*a*,*b*], which is the cost of substituting token *a* for token *b*. *sub*[*a*,*a*] should be zero for all *a*.
- *ins*[*a*], which is the cost of inserting token *a* into a sequence.

- *del*[*a*], which is the cost of deleting token *a* from a sequence.

In most instances the array *sub* is symmetric (the cost of substituting "H" for "h" is the same as substituting "h" for "H"). It is also frequently true that the values of *ins* and *del* are identical. This is because inserting a token into one sequence generally produces the same results as deleting that token from the other sequence. The algorithm, however, does not require these symmetries.

In the case of string matching we may make the same cost of deleting or inserting any character. We might make the cost of substituting an upper case letter for its lower case equivalent very low. We might give a medium cost when substituting vowels for each other or substituting "c" for "s" since they frequently make the same sound. We would put a higher cost on substituting "t" for "o". In the case of our gesture classifier that encodes a gesture as a sequence of region crossings, we might assign substitution costs based on adjacency of the regions with insertion and deletion costs being higher still. The cost assignment approach will vary from application to application.

The algorithm works by matching tokens from the two sequences starting at index 0 in both sequences. For a given pair of indices *i* and *j*, have three choices: substitute, delete, or insert. Note that substitution of matching tokens has zero cost. Thus using [*i*,*j*] as our state representation and our choices of editing operations as transitions, the minimal edit distance becomes a least-cost-path problem [A2.2].

Implementation

```
class EditState implements State //see section A2.2
{   int i,j;

    Boolean matches(EditState S)
    {   return i==S.i && j==S.j; }

    int nChoices()
    {   return 3; ]

    EditState choice(int N)
    {   if (N==0) //a substitution
        {   return new EditState(i+1,j+1); }
        else if (N==1) //delete from seq1
        {   return new EditState(i+1,j); }
        else //insert into seq1
        {   return new EditState(i,j+1); }
    }

    double choiceCost(int N)
    {   if (N==0) //a substitution
        {   return sub[seq1[i], seq2[j] ]; }
        else if (N==1) //a delete from seq1
        {   return del[seq1[i]]; }
        else if (N==2) //an insert into seq1
        {   return ins[seq2[j]]; }
    }
}
```

As described in [A2.2] this algorithm can be optimized by a **notAlreadyVisited** function so that states are not repeatedly explored. For relatively short sequences this function can be implemented with a two dimensional Boolean array using the indices of the two sequences. The array is initialized to all false and then set to true whenever the combination of two indices is attempted.

A3.2e - Nearest Neighbor Sequence Classifier

We can use minimal edit distance to compare an input sequence against each of the training examples as in the nearest-neighbor vector classifier [A3.1c]. However, this can be relatively expensive. If we take our training data and build a trie or a finite state machine we can adapt our minimal edit distance algorithm. Since a trie is a special case of finite state machines, we will use finite state machines for our implementation. Instead of characterizing our match process by the indices of the two strings being compared, we will use [*s,i*] where *s* is a state

in the finite state machine and *i* is an index into the sequence. We start with the start state of the machine and the index 0.

At a given state we have several choices. We can take each of the transitions leaving state *s* using the cost of *sub*[*seq*[*i*],*t*] where *t* is the token for that transition. We can stay in state *s* and increment *i* using the cost *del*[*seq*[*i*]]. We can also take each transition without moving *i*. In this last case we use the cost *ins*[*t*] where *t* is the token for the transition taken.

Implementation

```
class MatchState implements State //see section A2.2
{   int s; //state in the Finite State Machine
    int i; //index into the sequence being matched

    Boolean matches(MatchState S)
    {   return i==S.i && s==S.s; }

    int nChoices()
    {   return 2*nTransitionsLeaving(s)+1; ]

    MatchState choice(int N)
    {   if (N==0) //deleting a token from seq
        {   return new MatchState(s,i+1); }
        else if (N<=nTransitionsLeaving(s)) //follow transition using substitution
        {   return new MatchState( transition[s,N].next, i+1); }
        else  //follow transition by inserting its token into seq
        {   N = N-nTransitionsLeaving(s);
            return new MatchState( transition[s,N].token, i);
        }
    }

    double choiceCost(int N)
    {   if (N==0) //cost of deletion
        {   return del[seq[i]]; }
        else if (N<=nTransitionsLeaving(s)) // cost of substitution
        {   return sub[transition[s,N].token, seq[i]]; }
        else //cost of insertion
        {   N=N-nTransitionsLeaving(s);
            return ins[transition[s,N].token];
        }
    }
}
```

If we are working with a trie where each node can only be reached in one way, then we can assign an index to each trie node and can use that index along with the sequence index to subscript a two dimensional **notAlreadyVisited** array as we did with minimal edit distance. However, because the number of trie nodes may become large and because we may use finite state machines, we will need a hash table to take care of our repeated visits. The key to the hash table is composed from the state index and the sequence index. Remember that **notAlreadyVisited** is not required for the correct functioning of the least-cost path algorithm, but if space is not a problem it can sharply reduce the amount of time required for the match.

A 3.3 – Statistical Sequence Classifiers

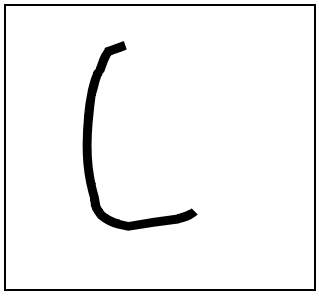
In the preceding section, we discussed classifiers where the desired sequences were clearly defined and the inputs received were known exactly. There are many situations, however, where the desired input sequence is not clearly specified. In some systems the input that the user generates is not known for sure. We use probabilistic methods for dealing with these issues. Statistical recognition is a huge part of pattern recognition and beyond the scope of this work. However, there are three algorithms that are quite useful in interactive systems. They are:

- *probabilistic state machines* – when the desired sequence is modeled as a state machine, but the inputs are uncertain and represented as a vector of probabilities.
- *Ngrams* – where the desired sequences are not clearly defined and the inputs may or may not be known exactly
- *Hidden Markov Models* – where we only know example inputs and must develop a state model that captures those inputs.

A 3.3a – Probabilistic State Machines

The use of such state machine methods in interactive settings was pioneered by Hudson[Hudson's probabilistic paper]. These methods apply when we can clearly define the required set of inputs as a state machine, but our input tokens are uncertain. For example, we may have a trie derived from dog and cat breeds as described in A 3.2 but our inputs come from a character recognizer, which has uncertain results. We may have a state machine of legal command phrases, but our speech recognizer is uncertain about which word the user actually spoke.

In our dog/cat example, the character recognizer may have a hard time deciding between C and L. We can model this by having the recognizer return a vector of probabilities rather than just a single token. Given the stroke below the following probabilities might be returned.



Error	0.12
. . . .	
C	0.50
. . . .	
L	0.30
. . . .	

O	0.05
. . . .	
U	0.03

. . . .	

The recognizer believes that the stroke is a C but is uncertain. This uncertainty is reflected in the probabilities, including the fact that the stroke may be in error. If our state machine is currently in a state that will accept either C, L or U we are therefore uncertain which next state we should be in as well as whether we should accept the error and remain in the current state. We therefore represent our current state not as a specific value but rather as an array of probabilities. We can use the probabilities of our current state, the legal transitions and the probabilities from our recognizer to work out our uncertainties. Our current most probable state may not have a transition on C and therefore will only lend its probability to L. The next most probable state may be very close in probability and it may have a transition on C. The high opinion that the recognizer has of C may override. Instead of forcing the recognizer to make the decision, the probabilistic state machine can take past state information, legal inputs and the recognizer's opinion all into account is choosing a classification result. For more examples see [Hudson xx].

We can build a classifier algorithm using these principles. Given a set of states *S* we define an array *cur[S]*. *cur[s]* is the probability that the machine is currently in state *s*. Given an alphabet of inputs *I* we can define an array *in[I]* such that *in[i]* is the probability that the last input received from the user was *i*. The key to probabilistic state machines is the algorithm for computing state transitions. We will define our state machine as a set of transitions $t(curState, input, nextState) \in T$. The probability that a transition *t* will be taken is the probability of *t.input* times the probability of *t.currentState*. Since there may be many transitions leading into a given state we will need to sum the probabilities of all such transitions.

Implementation

```
class Transition
{    int currentState;
     int input;
     int nextState;
}

double cur[S];
Transition trans[T];

void  stateTransition( double in[I] )
{    double next[S];

     double errorProb=1.0; // probability of incorrect input
     for (int i=0;i<I;i++)    { errorProb=errorProb-in[i]; }

     // probability of no transition because of error
     for (int s=0;s<S;s++) { next[s]=cur[s]*errorProb; }

     // transition probabilities
     for (int t=0;t<T;t++)
     {    int curS = trans[t].currentState;
          double inProb = in[ trans[t].input ];
          int nextS = trans[t].nextState;
          next[nextS]=next[nextS]+inProb*cur[curs];
     }

     // normalize cur to sum to 1.0
     double sum=0.0;
     for ( int s=0;s<S;s++) { sum=sum+next[s]; }
     for (int s=0;s<S;s++) { cur[s]=next[s] / sum; }
}
```

Note that if the input probabilities do not sum to 1.0 (as when the recognizer believes there is an error in the input) or some of the inputs are not acceptable to any of the non-zero states, the total probability in *cur[S]* will steadily decline. We accommodate this by dividing every element of *next* by the sum of *next* at the end of each transition.

A 3.3b – N-Grams

In some cases we may not have a state machine that clearly defines the set of legal inputs. For example, it is very difficult to create a state machine or even a context-free grammar that can accurately represent English syntax. Even if we

had such a grammar a syntactically correct sentence may not make any sense or many colloquialisms would be outside of the grammar. We may not want to model all French words by a huge dictionary because new French words would fail or the size may be prohibitive. What we need is a model of valid sequences that does not require that we formulate a state machine.

A common technique for such a model is N-grams. We form N-grams by taking a corpus of valid inputs and finding all input sequences of length N. We count how often each combination of N inputs occurs. Generally we normalize our N-gram model by dividing each count by the sum of all counts. This gives us a probability for each sequence of length N.

For example, if we use bi-grams (N=2) we can take our last input and use the bi-gram probabilities to predict what our next input should be. If our inputs are from a character recognizer we can use this information to refine our choice of characters. A “t” never follows a “q” in English. We can be even more accurate if we use tri-grams (N=3). In this case we use our last two inputs to determine the probabilities for our next input. Our N-gram statistics are filling the place of the state machine in helping discriminate among uncertain inputs.

The huge advantage of N-grams over state machines is that we can derive them from examples rather than carefully design them. Rather than build a state machine of possible words we can compute the letter tri-grams for all words in a dictionary. Rather than build a grammar for English we can compute word tri-grams for all words in the New York Times.

A major problem with N-grams is their size. If we use letters of the alphabet, there are 676 bi-grams, 17,576 possible tri-grams, and 456,976 quad-grams. These sizes are generally acceptable, except in very small machines. However, if we want to use a 20,000 word dictionary as our set of possible inputs, then there are 400 million possible bi-grams and 8 trillion possible tri-grams. In real problems the N-gram model is quite sparse. There are not 400 million valid word pairs in English. Though this sparseness eliminates a simple array implementation of N-grams it does make many such problems tractable.

Local predictors

One use of Ngrams is as a local predictor. In this algorithm the Ngrams are used to predict what the next input will be. No history is keep and no future inputs are taken into account. Take for example a character recognizer. We can use a tri-gram model of three characters and the last two inputs to predict what the next

input should be. If we combine the prediction probabilities with the input probabilities from the recognizer we can improve our system's ability to select the correct input. This assumes that the history (the last two inputs) is correct cannot take future inputs into account in making the decision. However, in this example, if the last two inputs were incorrect the user would have eliminated them and the future inputs do not yet exist.

The following implementation assumes a tri-gram model. It can easily be reduced to bi-grams or extended to larger numbers than three.

Implementation

```
    // interface to some previously obtained trigram model
interface Trigram()
{
    double getProb(int i1, i2, i3) // probability of the trigram (i1,i2,i3)
}

class LocalPredictor
{
    Trigram tg;      // the trigram model to be used.
    int i2;          // the last input recognized
    int i1;              // the input before last

        // Takes an array of input probabilities that is indexed by the
input class
        // number. It uses the tg model and these inputs to predict what
the correct
        // input should be.
    int predictInput( double inputProbabilities[nInputs] )
    {
        double maxProb=0.0;
        int selectedInput=-1;
        for (int i=0;i<nInputs;i++)
        {   double prob = inputProbabilities[i]*tg.getProb(i1,i2,i);
            if (prob>maxProb)
            {   selectedInput=i;
                maxProb=prob;
            }
        }
        i1=i2;
        i2=i;
        return i;
    }
}
```

Globally Optimal Input Sequence

N-grams can also be used to evaluate a sequence of uncertain inputs to select the sequence of inputs that is globally optimal relative to some N-gram model. Unlike the local predictor, this approach must have the entire input sequence before starting and takes the entire sequence into account before selecting the most plausible set of inputs.

As with probabilistic state machines and the local predictor, this algorithm models an input as a vector of probabilities indexed by the input class. We use the least-cost-path algorithm to select the most likely sequence of inputs. Each step of the least-cost-path is characterized by the following tuple :

- The index of the input being considered
- The last N-1 inputs that were chosen on this path
- The sum of all costs along this path

The cost of making a particular choice is *(1.0-choiceProbability)* or the probability that the choice was wrong. The *choiceProbability* is the N-gram probability of this choice, times the recognizer probability that this was the input generated by the user. The following algorithm will use tri-grams as the example.

When using N-grams we have the problem of starting and ending the sequence. For this we use a special empty input Λ. The sequence always starts with N-1 empty inputs and ends with N-1 empty inputs. In our algorithms we will use input class zero for our empty input.

Implementation

```
// See implementation for LeastCostPath

    // A RecognizerInput is a vector of probabilities. These are the
recognizer's opinion
    // on which input the user entered.
class RecognizerInput
{   double probs[nInputClasses];  }

    // interface to some previously obtained trigram model
interface Trigram()
{   double getProb(int i1, i2, i3); }        // probability of the trigram
(i1,i2,i3)

class GlobalOptimalSequence
{
    Trigram tg;
    RecognizerInput inputs[nUserInputs];

    TrigramState extends State // see LeastCostPath
    {
        int i2;        // the last input selected
        int i1;        // second to the last input selected
        int idx;       // index into inputs for the user input being
considered for this state

        TrigramState( int BeforeLast, Last, CurrentInput)
        {   i1=BeforeLast;
            i2=Last;
            idx=CurrentInput;
        }

        boolean matches(TrigramState S)
        {   if (idx>=nUserInputs && S.idx>=nUserInputs)
            {   return true; }                    // this is a goal state
            return (i1==S.i1 && i2==S.i2 && idx==S.idx);
        }

        int nChoices() = nInputClasses;

        TrigramState choice(int InputChosen)
        {   return new TrigramState(i2,InputChosen,idx+1); }

        double choiceCost(int InputChosen)
```

```
            {   double trigramProb = tg.getProb(i1,i2,InputChosen);
                double inputProb =  inputs[idx].progs[InputChosen];
                double choiceProb = trigramProb * inputProb;
                return 1.0-choiceProb;
            }
        } // end TrigramState

        Path chooseSequence(RecognizerInput in[])
        {   inputs = in;
            TrigramState goal=new TrigramState(0,0,in.length);
            TrigramState start=new TrigramState(0,0,0);
            return leastCostPath(start,goal);
        }
    }
```

A4 – Color

A 4.1 - RGB to HSB

This algorithm assumes that RGB is represented as three numbers between 0.0 and 1.0. The conversion is to HSB where each component varies from 0.0 to 1.0. The RGB is in the variables red, green and blue with the result stored in the variables hue, sat and bright.

Implemementation

```
rgbMin = min( red, green, blue);
rgbMax = max( red, green ,bue);
delta = rgbMax – rgbMin;

bright = rgbMax;
if (delta== 0 )
{    hue = 0;
     sat = 0;
}
else
{    sat = delta/rgbMax;

     dRed=    ( ( ( rgbMax-red ) / 6 ) + ( delta / 2 ) ) / delta;
     dGreen= ( ( ( rgbMax-green) / 6 ) + ( delta / 2 ) ) / delta;
     dBlue =  ( ( ( rgbMax-blue ) / 6 ) + (delta / 2 ) ) / delta;

     if ( red == rgbMax )
          hue = dBlue-dGreen;
     else if ( green == rgbMax )
          hue = ( 1 / 3 ) + dRed – dBlue;
     else
          hue = ( 2 / 3 ) + dGreen – dRed;

     if (hue < 0 ) hue = hue+1;
     if (hue > 1 ) hue = hue-1;
}
```

A 4.2 – HSB to RGB

This algorithm assumes that HSB and RGB are represented by values between 0.0 and 1.0. HSB is stored in the variables hue, sat and bright, and RGB is stored in the variables red, green and blue.

Implementation

```
if (sat == 0 )
{     red = bright;
      green = bright;
      blue = bright;
}
else
{     if (hue == 1.0)
            varInt = 0;
      else
            varInt = int(hue*6);
      var1 = bright*(1-sat);
      var2 = bright*( 1 – (sat * (hue*6 – varInt) ) );
      var3 = bright*( 1 – (sat * ( 1 – (hue*6 – varInt) ) ) );

      if (varInt == 0) { red = bright; green = var3; blue = var1; }
      else if ( varInt == 1) { red = var2; green = bright; blue = var1; }
      else if (varInt == 2) { red = var1; green = bright; blue = var3; }
      else if (varInt == 3) { red = var1; green = var2; blue = bright; }
      else if (varInt == 4 ) { red = var3; green = var1; blue = bright; }
      else { red = bright; green = var1; blue = var2; }
}
```