Introduction to Neural Networks

Supervised learning: neural networks, statistics, & machine learning

Initialize:

```
Off[General::spell1];

SetOptions[ListDensityPlot, ImageSize → Small];
SetOptions[DensityPlot, ImageSize → Small];
SetOptions[ContourPlot, ImageSize → Small];
```

Overview

Previously...

Introduced the notions of loss and risk for inference.

Rationale for Bayesian from the point of view of supervised learning that minimizes empirical risk.
We did this by starting from a goal to learn a decision rule (or estimator) to minimize average loss over a set of training pairs (the empirical risk), and then showed that this is equivalent (in the limit) to making a Bayes optimal decision.

Today

Set neural network supervised learning in the context of various statistical/machine learning methods.
Rationale: The brain is complicated. Good to understand bottom-up, from neurons to behavior. But also good to understand top-down, from behavior to quantitative models with as few free parameters as possible. But with a view to plausible neural network functioning.

In other words, always a good idea to try the simplest model first.

Introduction to scientific python

Nearest neighbor demonstration: “random Xor”

k nearest neighbors or k-NN for short. Has no problem with Xor.

Define $\mathcal{D}$ to be a mixture distribution of bivariate normals

```
𝒟 = MixtureDistribution[{1, 1, 1, 1},
   {MultinormalDistribution[{0, 0.}, {{1., 0}, {0, 1.}}],
    MultinormalDistribution[{0, 10.}, {{1., 0}, {0, 1.}}],
    MultinormalDistribution[{10, 0.}, {{1., 0}, {0, 1.}}],
    MultinormalDistribution[{10, 10.}, {{1., 0}, {0, 1.}}]}];
```

Here’s a sample:

```
RandomVariate[𝒟]
```

```
{-0.151706, 8.49553}
```

We can generate n = 100 samples:

```
n = 100;
```

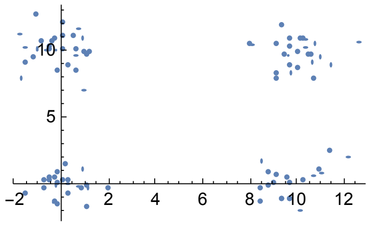
```
somedata = Table[RandomVariate[𝒟], {n}];
```

And then artificially assign labels, either 0 or 1 to each of the 100 points.

Exercise: change below for Xor, Or...And...

```
labels = Which[EuclideanDistance[#, {0, 0}] < 5, 0,
     EuclideanDistance[#, {0, 10}] < 5, 1, EuclideanDistance[#, {10, 0}] < 5,
     1, EuclideanDistance[#, {10, 10}] < 5, 0] &  /@ somedata;
```

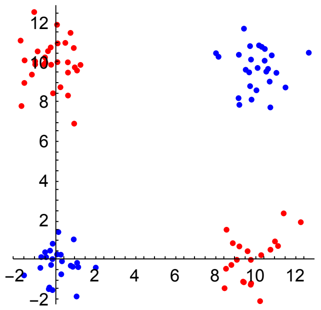
```
ListPlot[somedata]
```



We can color the plots with the labels, red for 1’s and blue for 0’s.

```
colorf = Blend[{{0, Blue}, {1, Red}}, #] &
pl = Graphics[MapThread[{colorf[#1], Point[#2]} &, {labels, somedata}],
  Axes → True, AspectRatio → 1]
```

```
Blend[{{0, Blue}, {1, Red}}, #1] &
```



Estimate nearest neighbor boundaries

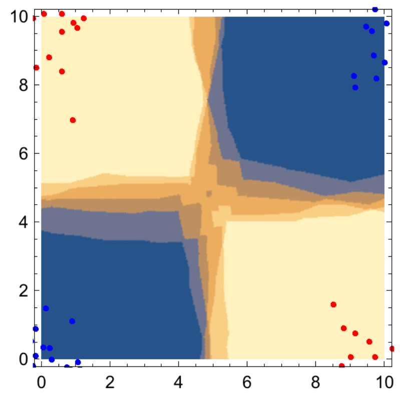
OK, so we have our artificial data and a pretty plot. Let’s classify.

Run through lots of points inside {{0,10},{0,10}}. For each point find the k nearest neighbors, and count how many have labels of 1 (i.e. how many dots are “red”). Plot up the count proportions.

```
nf2[x_, k_] := Total[
    Nearest[Flatten[Table[{somedata[[i]] → labels[[i]]}, {i, 1, n}], 1], x, k]] / k;
```

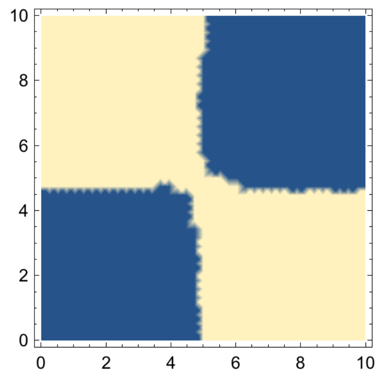
Here’s a plot of the proportion of 0s or 1s falling within a circle containing 5 neighbors, for each (x,y) point.
(“circle” is implicit in the default distance function used by Nearest[]).

```
Show[{DensityPlot[nf2[{x, y}, 5], {x, 0, 10}, {y, 0, 10}, PlotPoints → n], pl}]
```



Make the decision: if the proportion is more than 1/2 decide 1, otherwise 0:

```
k = 5;
DensityPlot[If[nf2[{x, y}, k] > 1 / 2, 1, 0], {x, 0, 10}, {y, 0, 10}, PlotPoints → 20]
```



k=1 and Voronoi diagrams

Nearest neighbor regression

“If something is similar to something else in one respect, it is likely to be similar in another respect.”

Discriminant functions

Before looking at linear discriminants (Fisher and SVM), let’s build our geometric intuitions of what a simple threshold logic or perceptron unit does by viewing it from a more formal point of view. Perceptron learning is an example of nonparametric statistical learning, because it doesn't require knowledge of the underlying probability distributions generating the data (such distributions are characterized by a relatively small number of "parameters", such as the mean and variance of a Gaussian distribution). Of course, how well it does will depend on the generative structure of the data. Much of the material below is covered in Duda and Hart (1978).

Linear discriminant functions: Two category case

A discriminant function, g(**x**) divides input space into two category regions depending on whether g(**x**)>0 or g(**x**)<0. (We've switched notation, **x**=**f**). This is the hard threshold case, i.e. a step function.

This linear case corresponds to the simple perceptron unit we studied earlier:

$$g(\mathbf{x}) = \mathbf{w.x} + w_0$$

where **w** is the weight vector and $\mathbf{w_0}$ is the (scalar) threshold (which we've called bias, although this "bias" has nothing to do with statistical "bias").

Discriminant functions can be generalized, for example to quadratic decision surfaces:

$$g(\mathbf{x}) = w_0 + \sum_{i=1} w_i x_i + \sum_{i=1}\sum_{j=1} w_{ij} x_i x_j$$

where **x** = $\{x_1, x_2, x_3 ...\}$. g(**x**)=0 defines a decision surface which in the linear case is a hyperplane.

Suppose ***x*** $_1$and ***x***$_2$ are vectors, with endpoints sitting on the hyperplane (or in the simple case of the red line below), then their difference is a vector lying in the hyperplane

$$\mathbf{w.x_1 + w_0} = \mathbf{w.x_2 + w_0}$$
$$\mathbf{w.(x_1 - x_2)} = 0$$

so because the dot product is zero, the weight vector **w** is normal to any vector lying in the hyperplane. Thus **w** determines how the plane is oriented. The normal vector **w** points into the region for which g(**x**)>0, and **-w** points into the region for which g(**x**)<0.

Again, let **x** be a point on the hyperplane. If we project **x** onto the normalized weight vector **x.w/|w|**, we have the normal distance of the hyperplane from the origin equal to:

$$\mathbf{w.x} / |\mathbf{w}| = -\mathbf{w_0} / |\mathbf{w}|$$

Thus, the threshold determines the position of the hyperplane.

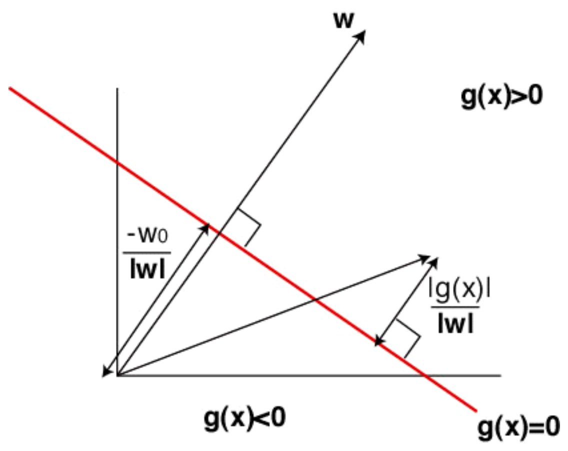
One can also show that the normal distance of x to the hyperplane is given by:

$$\mathbf{g(x)} / |\mathbf{w}|$$

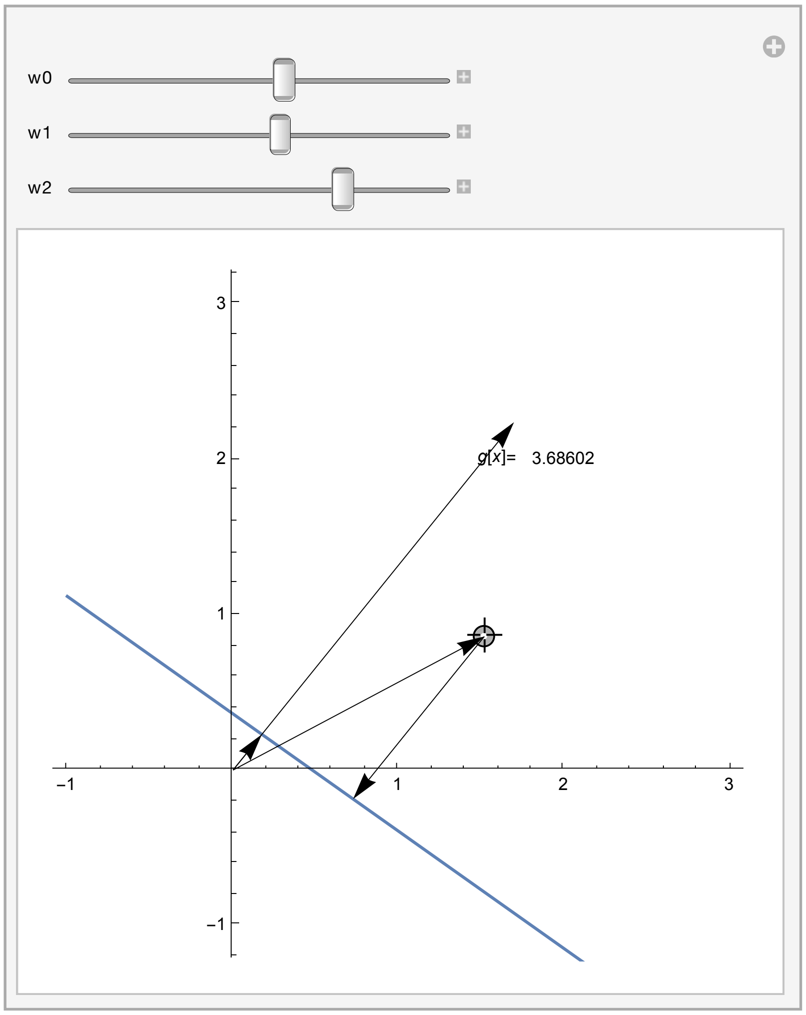
To summarize:

1) disriminant function divides the input space by a hyperplane decision surface;
2) The orientation of the surface is determined by the weight vector w;
3) the location is determined by the threshold w_0;
4) the discriminant function gives a measure of how far an input vector is from the hyperplane.

The figure summarizes the basic properties of the linear discriminant.


w
g(x)>0
-w0 / lwl
|g(x)| / lwl
g(x)<0
g(x)=0

```
Clear[w1, w2, w, w0];
Manipulate[
 (* x0={2,1};*)
 w = {w1, w2}; wn = w / Norm[w];
 g[x_] := {w1, w2}.x + w0;
 gg = Plot[Tooltip[
    x2 /. Solve[{w1, w2}.{x1, x2} + w0 == 0, x2], "discriminant"], {x1, -1, 3}];
 ggg = Graphics[g[Dynamic[MousePosition["Graphics"]]]];
 Show[{gg, Graphics[Inset["g[x]=", {1.6, 2}]],
   Graphics[Inset[ToString[g[x0]], {2, 2}]], Graphics[
    {Tooltip[Arrow[{{0, 0}, w}], "w"], Tooltip[Arrow[{{0, 0}, (-w0 / Norm[w]) * wn}],
      "-w0/|w|"], Tooltip[{Arrow[{{0, 0}, x0}]}, "x"],
     Tooltip[{Arrow[{x0, x0 - wn * g[x0] / Norm[w]}]}, "g(x)/|w|"]}]},
  PlotRange → {{-1, 3}, {-1, 3}}, AxesOrigin → {0, 0}, Axes → True,
  AspectRatio → 1, ImageSize → Medium],
 {{w0, -2.5}, -6, 3}, {{w1, 1}, 0, 3}, {{w2, 2}, 0, 3},
 {{x0, {2, 1}}, Locator}]
```



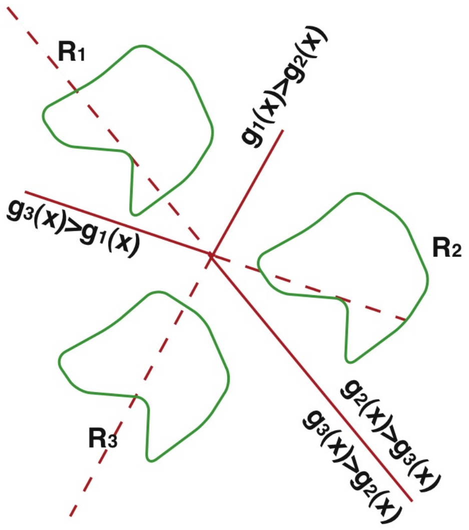
Multiple classes

Suppose there are c classes. There are a number of ways to define multiple class discriminant rules.

One way that avoids undefined regions is:

$$\mathbf{g}_i(\mathbf{x}) = \mathbf{w}_i . \mathbf{x} + \mathbf{w}_{i0},\quad i = 1, \ldots, c$$

Assign $\mathbf{x}$ to the ith class if : $g_i(\mathbf{x}) > g_j(x)$ for all $j \neq i$.



(Adapted from Duda & Hart, 1973)

It can be shown that this classifier partitions the input space into simply connected convex regions. This means that if you connect any two feature vectors belonging to the same class by a line, all points on the line are in the same class. Thus this linear classifier won't be able to handle problems for which there are disconnected clusters of features that all belong to the same class. Also, from a probabilistic perspective, if the underlying generative probability model for a given class has multiple peaks, this linear classifier won't do a good job either.

Fisher linear discriminant

Read in Statistical Add-in packages:

```
In[55]:= Off[General::"spell1"];
         << "MultivariateStatistics`";
         << "ComputationalGeometry`"
```

```
Ellipsoid::shdw :
  Symbol Ellipsoid appears in multiple contexts {MultivariateStatistics`, System`}; definitions in context
      MultivariateStatistics` may shadow or be shadowed by other definitions. >>
```

Basic idea: find a projection that minimizes the within-class variance, while maximizing the between-class variance.

Task-dependent Dimensionality reduction

Motivation

Later, when we consider the problem of dimensionality reduction, we will take another look at hyper-planes. But here the idea will be to find hyperplanes onto which we can project our input data, and from there divide up the hyperplane into decision regions. The idea is that the original input space may be impractically huge, but if we can find a subspace (hyperplane) that preserves the distinctions between categories as well as possible, we can make our decisions in smaller space. We will derive the Fisher linear "discriminant".

This is closely related to the psychology idea of finding "distinctive" features. E.g. consider bird identifica-tion. If I want to discriminate cardinals from other birds in my backyard, I can make use of the fact that (males) cardinals may be the only birds that are red. So even tho' the image of a bird can have lots of dimensions, if I project the image on to the "red" axis, I can do fairly well with just one number. How about male vs. female human faces?

A conceptually similar idea was used to learn diagnostic features in complex images (cf. Ullman et al., Hegde et al.).

We'll look at this more later when we learn about how to discover feature hierarchies.

Fisher's linear discriminant: building intuitions

Generative model: two nearby gaussian classes

Define two bivariate base distributions

```
In[144]:= (ar = {{1, 0.99}, {0.99, 1}};
          ndista = MultinormalDistribution[{0, -1}, ar];)
          (br = {{1, .9}, {.9, 2}};
          ndistb = MultinormalDistribution[{0, 1}, br];)
```

Find the expression for the probability distribution function of ndista

```
pdf = PDF[ndista, {x1, x2}]
```

$$1.12822\, e^{\frac{1}{2}(-x1\,(50.2513\,x1-49.7487\,(x2+1))-(x2+1)\,(50.2513\,(x2+1)-49.7487\,x1))}$$

Use Mean[ ] and CovarianceMatrix[ndista] to verify the population mean and the covariance matrix of ndistb

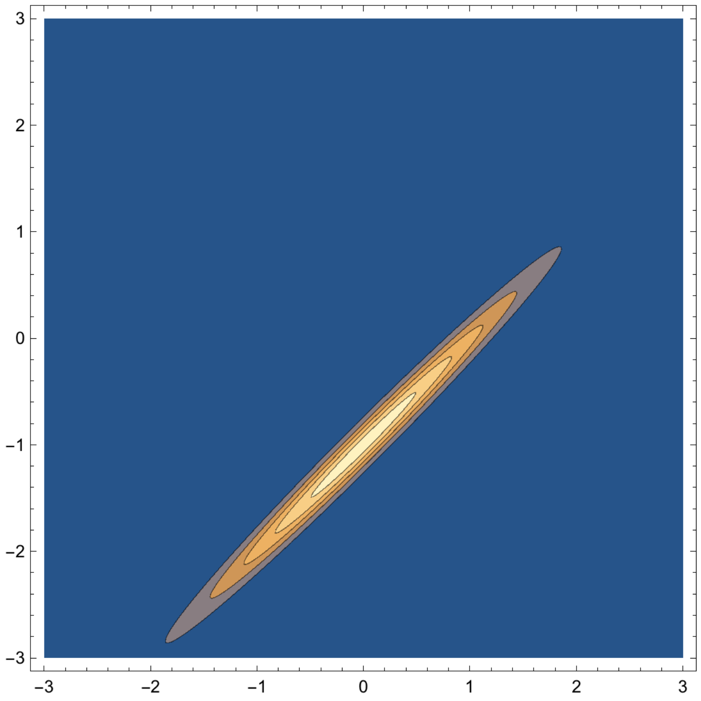
```
In[146]:= Mean[ndistb]

Out[146]= {0, 1}
```

```
In[147]:= Covariance[ndista]

Out[147]= {{1, 0.99}, {0.99, 1}}
```

Try different covariant matrices. Should they be symmetric? Constraints on the determinant of ar, br?
Make a contour plot of the PDF ndista

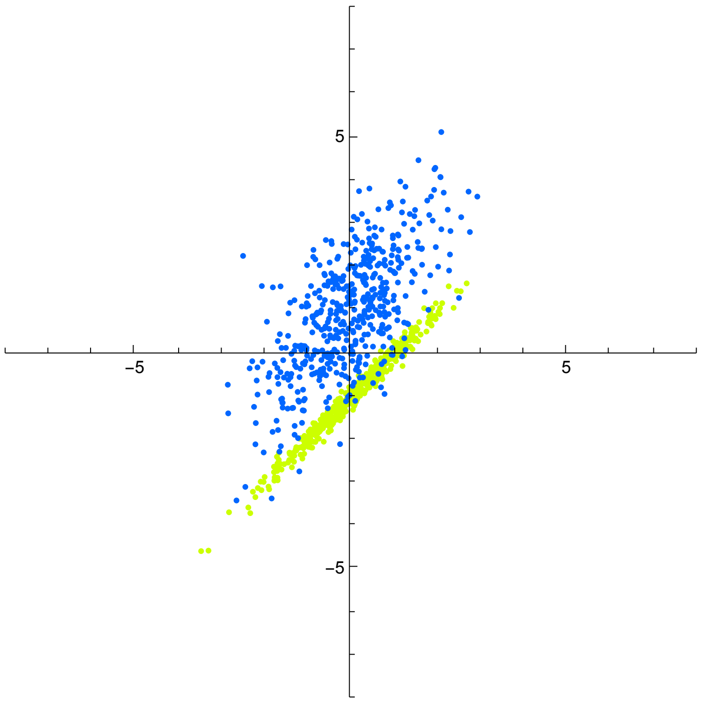
```
In[148]:= pdfa = PDF[ndista, {x1, x2}];
          ContourPlot[pdfa, {x1, -3, 3}, {x2, -3, 3}, PlotPoints → 64, PlotRange → All]
```

Out[149]=



```
In[150]:= nsamples = 500;
          a = Table[Random[ndista], {nsamples}];
          ga =
            ListPlot[a, PlotRange → {{-8, 8}, {-8, 8}}, AspectRatio → 1, PlotStyle → Hue[0.2`]];
          b = Table[Random[ndistb], {nsamples}];
          gb =
            ListPlot[b, PlotRange → {{-8, 8}, {-8, 8}}, AspectRatio → 1, PlotStyle → Hue[0.6`]];
          Show[
           ga,
           gb]
```

Out[155]=



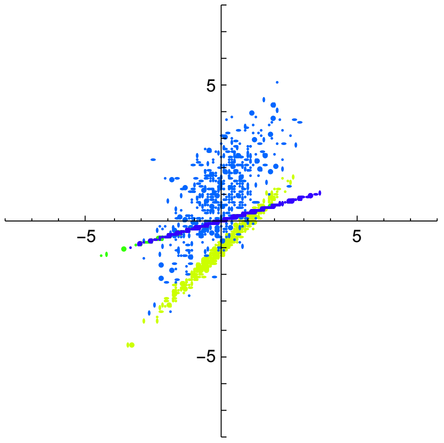
Use Mean[ ] to find the *sample* mean of b. Whats is the sample *covariance* of b?

```
In[156]:= Mean[b]

Out[156]= {-0.0618827, 0.939583}

In[157]:= Covariance[b]

Out[157]= {{1.06342, 0.945131}, {0.945131, 1.90154}}
```

Try out different projections of the data by varying the slope (m) of the discriminant line

```
In[158]:= m = 2 / 7; (*Hint: m=-2/3 is pretty good*)
          wnvec = {1, m} / Sqrt[1 + m^2];
```

In[159]:=
```
aproj = (#1 wnvec &) /@ (a.wnvec);
gaproj = ListPlot[aproj, AspectRatio → 1,
   PlotStyle → Hue[0.3`], DisplayFunction → Identity];
bproj = (#1 wnvec &) /@ (b.wnvec);
gbproj = ListPlot[bproj, AspectRatio → 1,
   PlotStyle → Hue[0.7`], DisplayFunction → Identity];
Show[ga, gb, gaproj, gbproj, DisplayFunction → $DisplayFunction,
 AspectRatio → Automatic]
```

Out[163]=


By trial and error, find a value of m that separates the classes well along the projection line. Plot out the marginal distributions relative to this line.

Calculate the "signal-to-noise" ratio along the projection line. Do this by taking the difference between the means divided by the square root of the product of the standard deviations along the line.

Theory and program demo for simple 2-class case

(see Duda and Hart for general case)

A measure of the separation between the projections is the difference between the means:

$$|w.(m_a - m_b)|$$

and

$$m_a = \frac{1}{N}\sum_{i=1}^{N} x, \quad \text{summed over the N x's from class a}$$

$$m_b = \frac{1}{M}\sum x, \quad \text{summed over the M x's from class b}$$

where w (**wnvec**) is the unknown unit vector along the discriminant line .

In our case above, the vector difference between the means is:

In[80]:=
```
Mean[a] - Mean[b]
```

Out[80]=
```
{0.0135485, -1.98083}
```

and the difference between the means projected onto a discriminant line is:

In[81]:=
```
wnvec.(Mean[a] - Mean[b])
```
Out[81]=
```
0.557202
```

To improve separation, we can't just scale w, because the noise scales too.

We'd like the difference between the means to be large relative to the variation for each class. We can define a measure of the scatter for the projected samples in say class a (a==1), by:

$$\sum_{y \in \text{class a}} (y - \tilde{m}_a)^2$$

where $\tilde{m}_a$is the sample mean of the points from class a projected onto discriminant line and y=**w.*x***
Or in terms of the Mathematic example:

In[82]:=
```
Apply[Plus, aproj - wnvec.Mean[a]];
```

The total scatter S is defined by the sum of the scatters for both classes (a and b).

$$S = \sum_{y \in \text{class a}} (y - \tilde{m}_a)^2 + \sum_{y \in \text{class b}} (y - \tilde{m}_b)^2$$

If we divide the above number by the total number of points, we have an estimate of the variance of the combined data along the projected axis.

We now have the basic ingredients behind intuition for the Fisher linear discriminant. We'd like to find that **w** for which J:

$$J(w) = \frac{|\tilde{m}_a - \tilde{m}_b|^2}{S} = \frac{|\mathbf{w}.(m_a - m_b)|^2}{S}$$

is biggest. We want to maximize the difference between the projected class means, while minimizing the dispersion of the data on the projected line.

One can show that S = $\mathbf{w}^{\mathbf{T}}.\mathbf{S}_{\mathbf{W}}.\mathbf{w}$, where

$S_W$is measure of within-class variation called the *within-class scatter matrix:*

$$S_W = \sum_{i=1}^{2} \sum_{x \in \text{Class i}} (x - m_i)(x - m_i)^T$$

For the numerator, a measure of between class variation is the *between-class scatter matrix*:

$$S_B = (m_1 - m_2).(m_1 - m_2)^T$$

and the difference between the projected means can be show to be:

$$|\tilde{m}_a - \tilde{m}_b|^2 = w^T.S_B.w$$

Find **w** (corresponding to slope) to maximize the criterion function

$$J(w) = \frac{w^T.S_B.w}{w^T.S_W.w}$$

Answer:

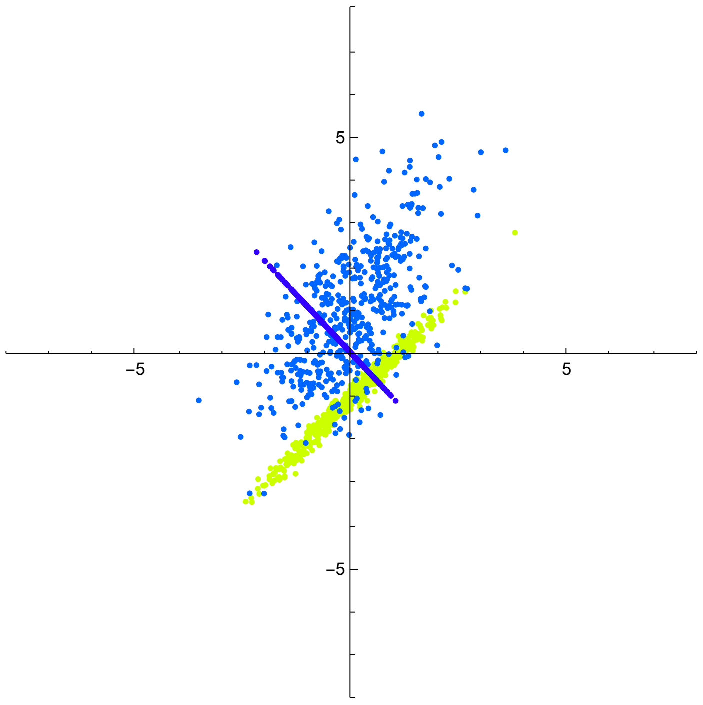
$$w = S_W^{-1}.(m_a - m_b)$$

Demo: Program to finding Fisher's linear discriminant

In[83]:=
```
normalize[x_] := x / Sqrt[x.x];
```

```
In[84]:= ma = Mean[a];
         mb = Mean[b];

In[86]:= Sa = Sum[Outer[Times, a[[i]] - ma, a[[i]] - ma], {i, 1, nsamples}];
         Sb = Sum[Outer[Times, b[[i]] - mb, b[[i]] - mb], {i, 1, nsamples}];
         Sw = Sa + Sb;
         wldf = normalize[Inverse[Sw].(ma - mb)]

Out[89]= {0.682891, -0.73052}

In[90]:= aproj = (#1 wldf &) /@ (a.wldf);
         gaproj = ListPlot[aproj, AspectRatio → 1,
            PlotStyle → Hue[0.3`], DisplayFunction → Identity];
         bproj = (#1 wldf &) /@ (b.wldf);
         gbproj = ListPlot[bproj, AspectRatio → 1,
            PlotStyle → Hue[0.7`], DisplayFunction → Identity];
         Show[ga, gb, gaproj, gbproj, DisplayFunction → $DisplayFunction]
```

Out[94]=

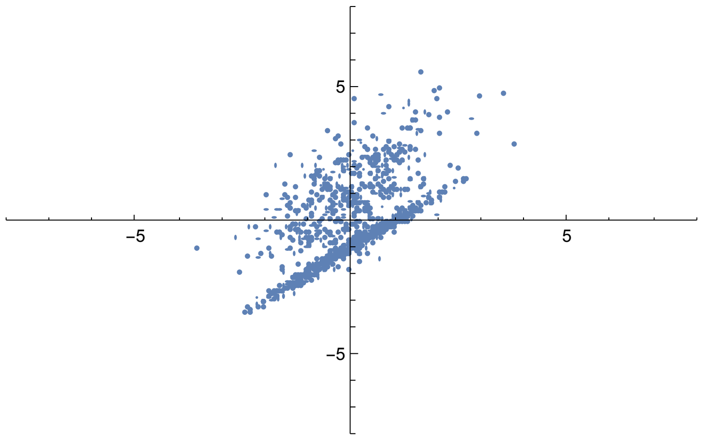


We started off with a 2-dimensional input problem and turned it into a 1-D problem. For the n-dimensional case, see Duda and Hart.

Compare with the principal component axes

```
In[95]:=  c = Join[a, b];
          ListPlot[c, PlotRange → {{-8, 8}, {-8, 8}}]
```

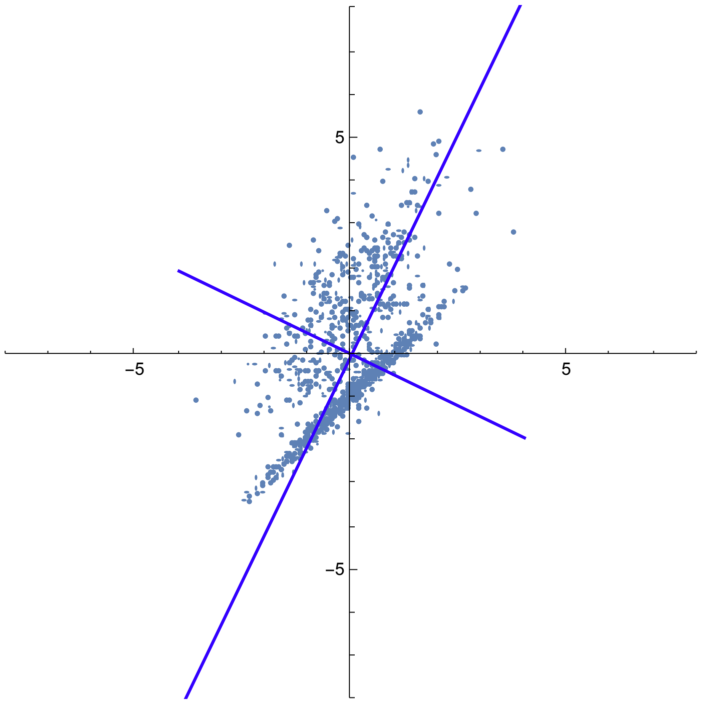
Out[96]=



```
In[97]:=  g1 = ListPlot[c, PlotRange → {{-8, 8}, {-8, 8}},
              AspectRatio → 1, DisplayFunction → Identity];

In[98]:=  auto = Covariance[c]
          eigvalues = Eigenvalues[auto]
          eigauto = Eigenvectors[auto]

Out[98]=  {{1.00703, 0.93032}, {0.93032, 2.48545}}

Out[99]=  {2.93449, 0.557994}

Out[100]= {{0.434683, 0.900584}, {-0.900584, 0.434683}}

In[101]:= gPCA  = Plot[{eigauto[[1,2]]/eigauto[[1,1]] x,
              eigauto[[2,2]]/eigauto[[2,1]] x},
                  {x,-4,4}, AspectRatio->1,
                  DisplayFunction->Identity,
                  PlotStyle->{RGBColor[.2,0,1]}];
```

```
In[102]:= Show[g1, gPCA, DisplayFunction → $DisplayFunction]
```

Out[102]=



How does the principal component (biggest variance) compare with the Fisher discriminant line?

Support vector machines (SVMs) and kernel methods

Initialize

Read in Add-in packages:

```
In[165]:= Off[General::"spell1"];
          << "ErrorBarPlots`";
          << "MultivariateStatistics`";
```

```
Ellipsoid::shdw :
   Symbol Ellipsoid appears in multiple contexts {MultivariateStatistics`, System`}; definitions in context
       MultivariateStatistics` may shadow or be shadowed by other definitions. ≫
```

Make sure the SVM package is downloaded in the default directory

```
In[168]:= SetDirectory[NotebookDirectory[]]

Out[168]= /Users/kersten/Sites/kersten-lab/courses/Psy5038WF2014/Lectures/Lect_18_Python
```

```
In[169]:= << MathSVMv7`
```

Some other useful directory functions:

```
FileNames[];
Directory[];
```

We assume a simple perceptron TLU to classify vector data x into one of two classes depending on the sign of g(x):

decision(x) = sign($w.x + b$).

Given $g(\mathbf{x}) = \mathbf{w.x} + b$, recall that g(x)/||w|| is the distance of a data point x from a plane defined by g(x) = 0. In support vector machines, the goal is to find the separating plane (i.e. find w and b) that is as far as possible from any of the data points. The intuition is that this will minimize the probability of making wrong classifications when given new data at some future point.

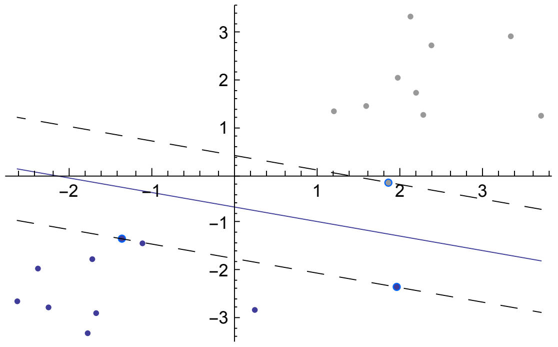
Formally, we want to solve"

$$\max_{(w,b)}\left(\min_i d\left(\Pi_{w,b}, x_i\right)\right), \qquad (1)$$

where $d(\Pi_{w,b}, x_i) = g(x) / || w ||$, i.e.

$d(\Pi_{w,b}, x_i) = g(x) / || w || = | w.x_i + b | / || w ||$

In[8]:=



Two-class data (black and grey dots), their optimal separating hyperplane (continuous line), and support vectors (circled in blue). This is an example output of the `SVMPlot` function in *MathSVM*. The width of the “corridor” defined by the two dotted lines connecting the support vectors is the margin of the optimal separating hyperplane. (From Nilsson et al., 2006)

The Primal Problem

It can be shown that the optimal separating hyperplane solving (1) can be found as the solution to the equivalent optimization problem

$$\min_{w,b} \frac{1}{2} || w ||^2$$

$$\text{subject to } y_i\left(w^T x_i + b\right) \geq 1,$$

Where y_i is the class label, either 1 or -1, for the ith point. Typically, equality will hold for a relatively small number of the data vectors. These data are termed *support vectors.* The solution (w, b) depends only on these specific points, and in effect contain all the information for the decision rule. The "dual problem".

A Simple linear SVM Example (from Nilsson et al. 2006)

Here's a demo of a simple SVM problem. It uses the add on package *MathSVMv7 written by Nilsson et al.*

In[172]:=
```
len = 20;
X = Join[
   RandomReal[NormalDistribution[-2, 1], {len / 2, 2}],
   RandomReal[NormalDistribution[2, 1], {len / 2, 2}]];
y = Join[Table[1, {len / 2}], Table[-1, {len / 2}]];
```

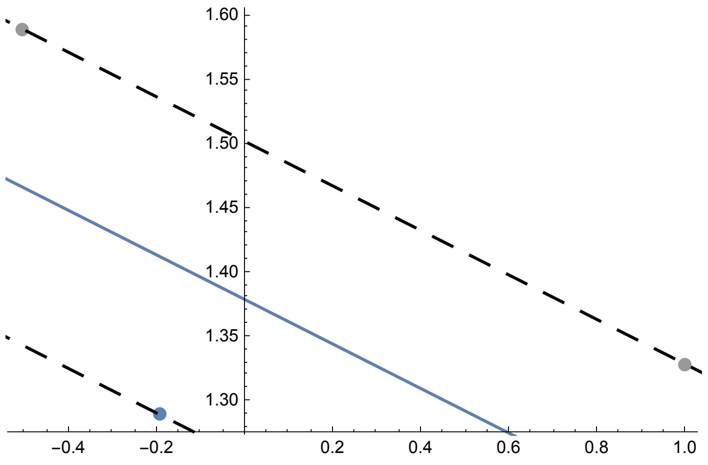
We use the simple SVM formulation provided in *MathSVM* by the `SeparableSVM` function.

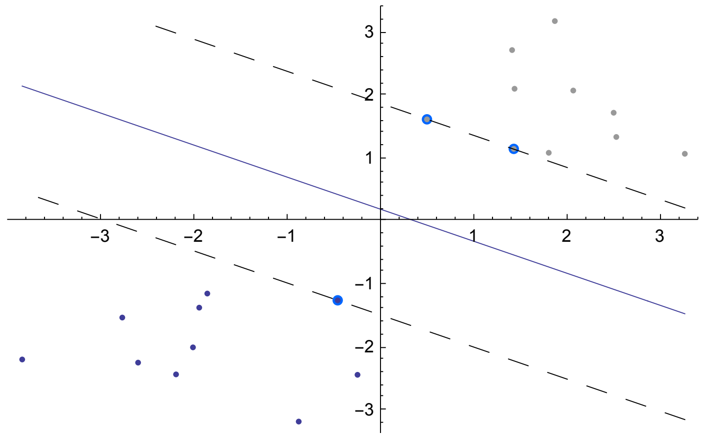
```
In[175]:= τ = 0.01;
          α = SeparableSVM[X, y, τ]
Out[176]= {0, 0, 34.1686, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 8.02851, 0, 0, 0, 26.1401, 0}
```

In the output figure below, the solid line marks the optimal hyperplane, and dotted lines mark the width of the corridor that joins support vectors (highlighted in blue).

```
In[179]:= SVMPlot[α, X, y]
```

Out[179]=





A Nonlinear Example: Using Kernels (from Nilsson et al. 2006)

What if the data are not linearly separable? The essential idea is to map the data (through some non-linear mapping, e.g. polynomial) to a higher-dimensional "feature" space to find the optimal hyperplane separating the data. The dot product gets replaced by a non-linear kernel function. For example, the polynomial kernel is given by:

In[15]:= $k(\mathbf{x_i}, \mathbf{x_j}) = (\mathbf{x_i} \cdot \mathbf{x_j})^d$

Out[15]= $k(\mathbf{x_i}, \mathbf{x_j}) = (\mathbf{x_i} \cdot \mathbf{x_j})^d$

If d = 1, we have the standard dot product, but for d = 2, 3, etc.. we have polynomial functions of the elements of the vectors x. See Nilsson et al, and paper by Jäkel (2009) for more information on kernels.

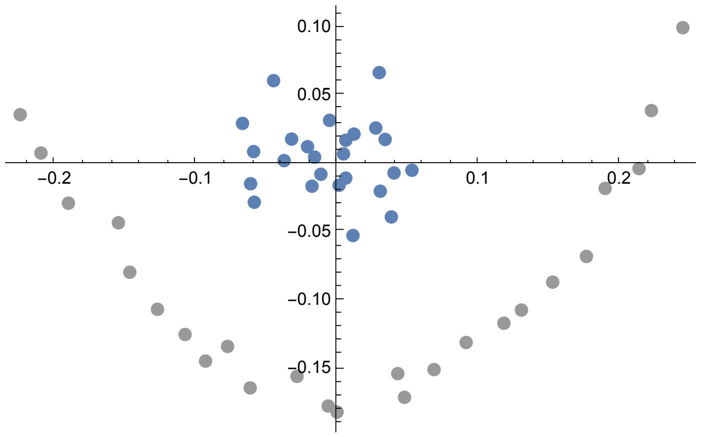
Here is a demo for an application for nonlinear classification . We'll use second-degree kernel, e.g.:

```
In[183]:= PolynomialKernel[{xx1, xx2, xx3}, {1, -1, -1}, 2]
```

Some synthetic data which is not linearly separable.

```
In[17]:= len = 50;
X = Join[
   RandomReal[NormalDistribution[0, 0.03], {len / 2, 2}],
   Table[
    {RandomReal[NormalDistribution[i / len - 1 / 4, 0.01]],
     Random[NormalDistribution[(2 i / len - 1 / 2)^2 - 1 / 6, 0.01]]},
    {i, len / 2}]];
y = Join[Table[1, {len / 2}], Table[-1, {len / 2}]];
SVMDataPlot[X, y, PlotRange → All]
```

Out[20]=



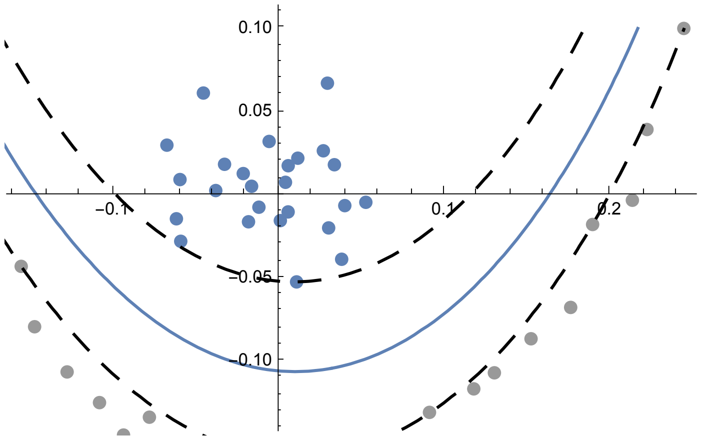
We use the `KernelFunction` to specify the kernel type and run **SeparableSVM[]**.

```
In[21]:= τ = 0.01;
pk = PolynomialKernel[#1, #2, 2] &;
α = SeparableSVM[X, y, τ, KernelFunction → pk]

Out[23]= {0, 0, 0, 3327.96, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  1494.97, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 867.358, 0, 0, 0, 0, 861.755, 0, 0, 103.874}
```

When visualizing the results, `SVMPlot` can use the kernel functions to draw any nonlinear decision curves.

```
In[24]:= SVMPlot[α, X, y, KernelFunction → pk]
```

Out[24]=



```
In[25]:= Clear[len, X, y, α, pk]
```

More information

The wiki entry for SVMs has a fairly good introduction (As of fall 2014).

To go to the source see Vapnik (1995)

http : // svm.first.gmd.de/

Demo links:

http://svm.dcs.rhbnc.ac.uk/pagesnew/GPat.shtml

And for applications of kernel methods more generally, to cognitive and neuroscience see reviews by Jäkel et al. (2006; 2009).

Logistic regression, and the generalized linear model

The single (weight) layer perceptron is a special case of logistic regression, which in turn is a special case of a *generalized linear model*. Logistic regression has been studied by statisticians since the 1950s.

The starting point is wanting to model a binary response Y (think action potential on or off) by the conditional probability:

$$P(Y=1 \mid X=x),$$

where X are the input features. Let’s assume that

$$P(Y=1 \mid X=x) = p(x;w)$$

for some parameters w. How to model this function p?

As we’ve done before, let’s assume the simplest: we want a linear function. But over what function of p?

$p(x) = w.x + b$ has problems because the left side is bounded between 0 and 1, and the right side is unbounded.

Log p(x) = w.x + b may seem nice because adding input features multiplies probabilities, but is bounded on only one side. A solution is to model log odds as a linear function:

$$\text{Log}\,\frac{p(x)}{1-p(x)} = w.x + b$$

and after solving for p(x),

$$p(x) = \frac{1}{1+e^{-(w.x+b)}}$$

Looking familiar? This says that the probability of Y=1 is determined by taking a linear weighted sum of the inputs plus b (think bias) and deciding “yes” if this is bigger than zero. This is a linear classifier. This is also the probabilistic update rule we used for the Boltzmann machine.

To make things look even more familiar let’s calculate the expected value of Y=1 (of a neuron firing). First note that we can write the probability of Y= y_i, whose values are 1 or 0 in a concise summary expression as:

$$p(y_i \mid x) = \frac{e^{y_i\,(w.x+b)}}{1+e^{(w.x+b)}}$$

Then the probability of firing, i.e. $y_i = 1$ is:

$$p(y_i = 1 \mid x) = \frac{1}{1+e^{-(w.x+b)}} = \sigma(\text{w.x + b})$$

By definition, the average rate is:

$$\sum_{y_i=0}^{1} y_i\, p\,(y_i \mid x) = 0 \times p(y_i = 0 \mid x) + 1 \times p(y_i = 1 \mid x) = \sigma(\text{w.x + b})$$

This is stage 1 and 2 of our original generic neuron model.
The logistic regression model can be extended to multiple class decisions.

These models in turn can be viewed as special cases of generalized linear models.

Note: The general linear model is different! It really is linear. The generalized linear model is in general not. For applications of the generalized linear model to computational neuroscience, see Pillow (2007).

Other methods

Naive Bayes. Assume we have a class variable C that takes on values corresponding to labels. And assume a set of features $\{F_i\}$. Naive Bayes assumes that the probabilities of features are all conditionally independent of the class they came from:

$$p(C|F_1,\dots,F_n) = \frac{1}{Z} p(C) \prod_{i=1}^{n} p(F_i|C)$$

http://en.wikipedia.org/wiki/Naive_Bayes_classifier

And others, AdaBoost, http://en.wikipedia.org/wiki/AdaBoost, and Random Forest, This Random Forest link also has a concise description of k-NN. http://en.wikipedia.org/wiki/Random_forest

Mathematica's Classify[ ] function demo

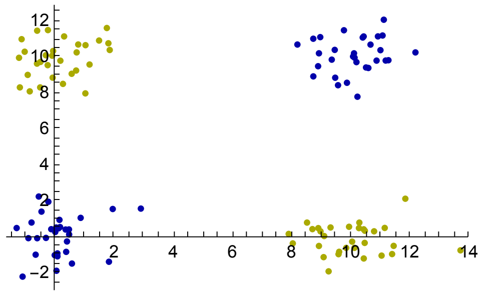
Here we adapt code from one of *Mathematica*'s examples to apply it to the noisy Xor problem.

Define clusters sampled from normal distributions:

```
sampledata[center_] :=
  RandomVariate[MultinormalDistribution[center, IdentityMatrix[2]], 30];
```

Generate clusters, assign labels to each and plot;

```
clusters = sampledata /@ {{0, 0}, {0, 10}, {10, 0}, {10, 10}};
colors = {Blue, Yellow, Yellow, Blue};
ListPlot[clusters, PlotStyle → Darker@colors]
```



```
Dimensions[clusters]
```

```
{4, 30, 2}
```

The finagling in the next line is to assign all of the 2D points to their corresponding colors, and then running the associations through the Classifier.

```
Flatten[Thread[Transpose[clusters][[#]] → colors] & /@ Range[30], 1]
```

```
c3 = Classify[Flatten[Thread[Transpose[clusters][[#]] → colors] & /@
     Range[Dimensions[clusters][[2]]], 1]]
```

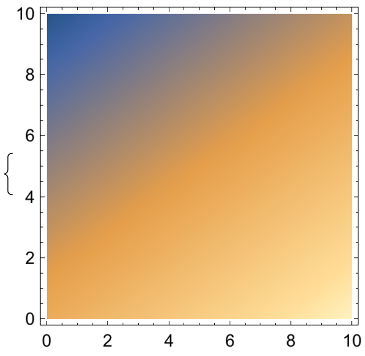
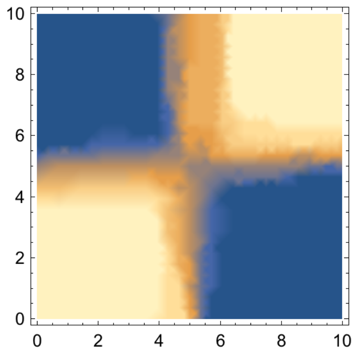
```
ClassifierFunction[ Method: NearestNeighbors
                    Number of classes: 2 ]
```

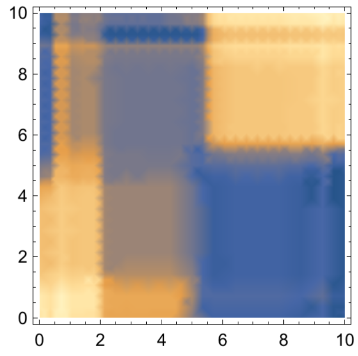
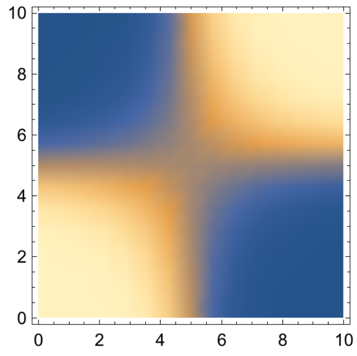
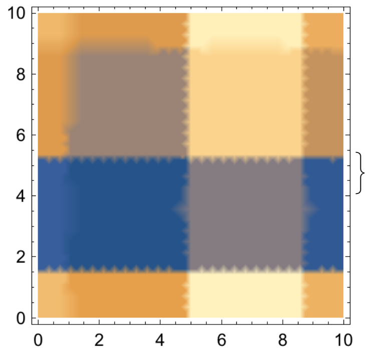
How do probabilities get calculated?

```
Normal@c3[{2, 7}, "Probabilities"][[2]]
```

```
0.97619
```

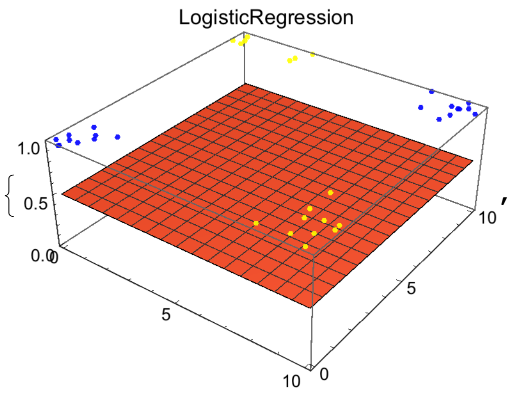
```
plotprob[method_] := Module[{},
    c3 =
     Classify[Flatten[Thread[Transpose[clusters][[#]] → colors] & /@ Range[30], 1],
      Method → method];
    DensityPlot[Normal@c3[{x, y}, "Probabilities"][[1]], {x, 0, 10}, {y, 0, 10}]
   ];

plotprob /@ {"LogisticRegression", "NearestNeighbors",
   "RandomForest", "SupportVectorMachine", "NaiveBayes"}
```

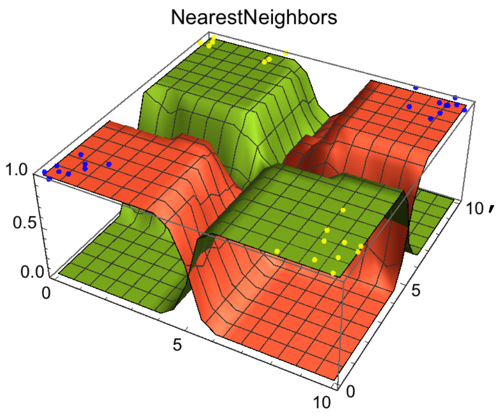
{, ,

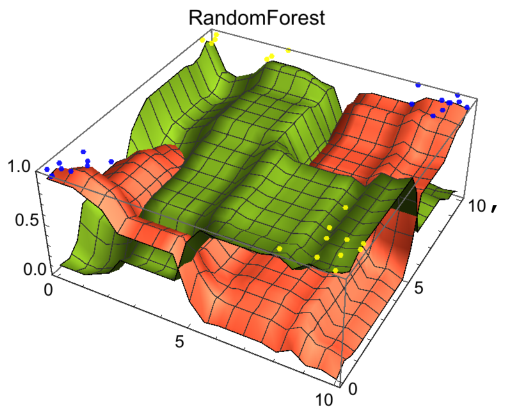
, , }

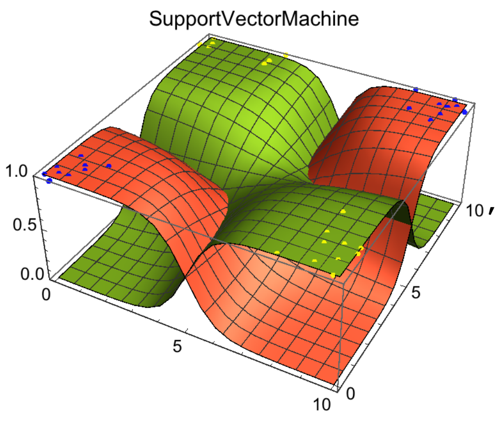
```
plotprobabilities[method_] := Module[{c, proba},
    c3 = Classify[Flatten[
        Thread[Transpose[clusters][[#]] → colors] & /@ Range[30], 1], Method → method];
    proba = Table[Append[{x, y}, #] & /@ Lookup[c3[{x, y}, "Probabilities"], colors],
      {x, 0, 10, .5}, {y, 0, 10, .5}];
    proba = Flatten[#, 1] & /@ Transpose[proba, {3, 2, 1, 4}];
    Show[
     ListPlot3D[proba, PlotRange → {0, 1}, PlotLabel → method],
     ListPointPlot3D[Map[Append[#, 1] &, clusters, {2}],
      PlotStyle → ({Opacity[.9], #} & /@ colors)], ImageSize → 250
    ]
];
```

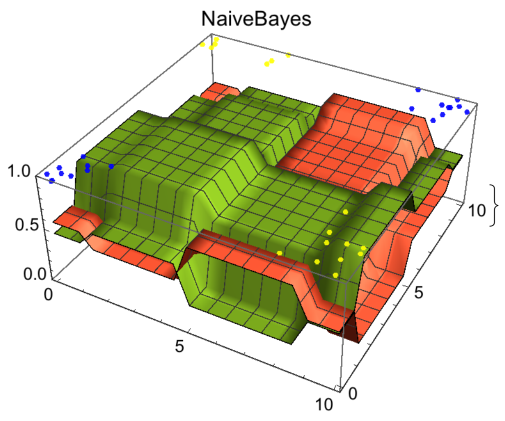
```
plotprobabilities /@ {"LogisticRegression", "NearestNeighbors",
  "RandomForest", "SupportVectorMachine", "NaiveBayes"}
```











Transitioning to Python

Why?

Pros:

  Free

Rapid growth in packages for scientific computation, including spiking neural networks (http://briansimulator.org), machine learning, image processing, computer vision, and psychophysics.

Lots of packages, examples, tutorials. More specialized packages than *Mathematica*. For example, deep convolutional networks: http://deeplearning.net/tutorial/lenet.html.

Cons:
Lots and lots of packages, examples, tutorials. Packages are a set of moving targets that are may live or fade away and die. Not as many specialized packages as matlab. Installation can be a pain.

Scientific python: numpy, scipy, ...

Raw python is too basic. We need add on modules. So in order of importance, we will always start by loading numpy for creating and manipulating numerical arrays, scipy for scientific functions (e.g. optimization, ..), and matplotlib for graphics:

We'll also use IPython which brings the notebook functionality of *Mathematica* to python.

In addition the modules, pandas, scikit-learn, and scikit-image, provide a wide range of machine learning and an image processing functions. Lots of others, such as neurolab.

In this course, the main purpose for python for is for the lecture material on MCMC sampling where we'll be using pymc.

In order to get started, we'll first go to a terminal window and see if python is there. Execute a demo program. Then
we'll open a new IPython notebook and bring the python program into the notebook.

References

Duda, R. O., Hart, P. E., & Stork, D. G. (2001). *Pattern classification* (2nd ed.). New York: Wiley. (Amazon.com)
Hegdé, J., Bart, E., & Kersten, D. (2008). Fragment-based learning of visual object categories. Current Biology, 18(8), 597–601. doi:10.1016/j.cub.2008.03.058
Jakel, F., Schölkopf, B., & Wichmann, F. A. (2013). Generalization and similarity in exemplar models of categorization: Insights from machine learning. Psychonomic Bulletin & Review, 15(2), 256–271. doi:10.3758/PBR.15.2.256
Vapnik, V. N. (1995). *The nature of statistical learning*. New York: Springer-Verlag.
http://neuron.eng.wayne.edu/software.html
Ullman, S., Vidal-Naquet, M., & Sali, E. (2002). Visual features of intermediate complexity and their use in classification. Nature Neuroscience, 5(7), 682–687. doi:10.1038/nn870
Yuille, A., Coughlan J., Kersten D. (1998) (pdf)