Fitting the 4-Parameter Lineal Basis Model

Barrie J Stokes

Clinical Pharmacology
Faculty of Health
University of Newcastle, NSW, Australia

Trevor N Moffiet
Statistics
Faculty of Science & Information Technology
University of Newcastle, NSW, Australia

Initialization.

In[1]:=
```
Needs["mathStatica`"]
FancyMatrix[Off]
SuperLog[On]
```

```
FancyMatrix is now Off.

SuperLog is now On.
```

Options.

In[4]:=
```
SetOptions[Plot, Axes → False, Frame → True , ImageSize → 400];
SetOptions[ListPlot, Axes → False, Frame → True, ImageSize → 400];
SetOptions[ParametricPlot, Axes → False, Frame → True, ImageSize → 400, AspectRatio → GoldenRatio^-1];
```

Packages.

In[7]:=
```
Needs["MultivariateStatistics`"]
```

In[8]:=
```
?MultinormalDistribution
```

MultinormalDistribution[μ, Σ] represents a multivariate
normal (Gaussian) distribution with mean vector μ and covariance matrix Σ. ≫

Functions.

hessianH[f_, x_List?VectorQ]

In[9]:=
```
ClearAll[hessianH]
hessianH[f_, x_List?VectorQ] := D[f, {x, 2}]
?hessianH
```

```
Global`hessianH
```

```
hessianH[f_, x_List?VectorQ] := D[f, {x, 2}]
```

In[12]:=
```
hessianH[p^2 q^2, {p, q}]
```

Out[12]=
$$\{\{2\,q^2,\ 4\,p\,q\},\ \{4\,p\,q,\ 2\,p^2\}\}$$

In[13]:=
```
Hessian[p^2 q^2, {p, q}] (* The mathStatica Hessian function *)
```

Out[13]=
$$\{\{2\,q^2,\ 4\,p\,q\},\ \{4\,p\,q,\ 2\,p^2\}\}$$

restrictedParameters[allParamsPointEstimates_, cvm_]

In[14]:=
```
ClearAll[restrictedParameters]
restrictedParameters[allParamsPointEstimates_, cvm_] := Module[{n, rns}, n = 1;
  While[True, (rns = RandomReal[MultinormalDistribution[allParamsPointEstimates, cvm]];
    If[rns[[1]] > 0 && rns[[2]] > 0, Break[]]); n++]; rns]
?restrictedParameters
```

```
Global`restrictedParameters
```

```
restrictedParameters[allParamsPointEstimates_, cvm_] := Module[{n, rns}, n = 1;
  While[True, (rns = RandomReal[MultinormalDistribution[allParamsPointEstimates, cvm]];
    If[rns[[1]] > 0 && rns[[2]] > 0, Break[]]); n++]; rns]
```

varCovarMatrixtoCorrMatrix[cvm_]

In[17]:=
```
ClearAll[ varCovarMatrixtoCorrMatrix ]
varCovarMatrixtoCorrMatrix[ cvm_ ] := Transpose[ Transpose[ cvm / Sqrt[ Diagonal[ cvm ] ] ] / Sqrt[ Diagonal[ cvm ] ] ]
?varCovarMatrixtoCorrMatrix
```

```
Global`varCovarMatrixtoCorrMatrix
```

```
varCovarMatrixtoCorrMatrix[cvm_] := Transpose[ Transpose[ cvm / Sqrt[ Diagonal[cvm] ] ] / Sqrt[ Diagonal[cvm] ] ]
```

In[20]:=
```
testMat = {{σ1^2, ρ12 σ1 σ2, ρ13 σ1 σ3}, {ρ12 σ1 σ2, σ2^2, ρ23 σ2 σ3}, {ρ13 σ1 σ3, ρ23 σ2 σ3, σ3^2}};
% // MatrixForm
```

Out[21]//MatrixForm=

$$\begin{pmatrix} \sigma1^2 & \rho12\,\sigma1\,\sigma2 & \rho13\,\sigma1\,\sigma3 \\ \rho12\,\sigma1\,\sigma2 & \sigma2^2 & \rho23\,\sigma2\,\sigma3 \\ \rho13\,\sigma1\,\sigma3 & \rho23\,\sigma2\,\sigma3 & \sigma3^2 \end{pmatrix}$$

In[22]:=
```
varCovarMatrixtoCorrMatrix[ testMat ] /. {1/Sqrt[z_^2] -> 1/z} // MatrixForm
```

Out[22]//MatrixForm=

$$\begin{pmatrix} 1 & \rho12 & \rho13 \\ \rho12 & 1 & \rho23 \\ \rho13 & \rho23 & 1 \end{pmatrix}$$

distance[{{x1_, y1_}, {x2_, y2_}}]

In[23]:=
```
ClearAll[ distance ]
distance[ {{x1_, y1_}, {x2_, y2_}} ] := Sqrt[(x1 - x2)^2 + (y1 - y2)^2]
?distance
```

```
Global`distance
```

```
distance[{{x1_, y1_}, {x2_, y2_}}] := Sqrt[(x1 - x2)^2 + (y1 - y2)^2]
```

circumCircleRadius[{{x1_, y1_}, {x2_, y2_}, {x3_, y3_}}]

In[26]:=
```
ClearAll[ circumCircleRadius ]
circumCircleRadius[ {{x1_, y1_}, {x2_, y2_}, {x3_, y3_}} ] :=
 1/2 Sqrt[(((x1 - x2)^2 + (y1 - y2)^2) ((x1 - x3)^2 + (y1 - y3)^2) ((x2 - x3)^2 + (y2 - y3)^2)) /
     (x3 (-y1 + y2) + x2 (y1 - y3) + x1 (-y2 + y3))^2]
?circumCircleRadius
```

```
Global`circumCircleRadius
```

```
circumCircleRadius[{{x1_, y1_}, {x2_, y2_}, {x3_, y3_}}] :=
```

$$\frac{1}{2}\sqrt{\frac{\left((x1-x2)^2+(y1-y2)^2\right)\left((x1-x3)^2+(y1-y3)^2\right)\left((x2-x3)^2+(y2-y3)^2\right)}{(x3\,(-y1+y2)+x2\,(y1-y3)+x1\,(-y2+y3))^2}}$$

emptyQ[center_, alpha_, plist_]

In[29]:=
```
(*Main function def*)
(*0 for interior bar,1 for boundary bar,2 for bar to be erased*)
```

This function determines whether any points in the list "pList" are inside a circle centre "center" and radius "alpha".
If not, it returns True, i.e., the circle is empty of points in "pList".

In[30]:=
```
ClearAll[emptyQ];
emptyQ[center_, alpha_, plist_] := Module[
  {empty = True, n = 1},
  While[empty && n ≤ Length[plist], empty = Norm[plist[[n]] - center] > alpha; n++];
  empty]
?emptyQ
```

```
Global`emptyQ
```

```
emptyQ[center_, alpha_, plist_] := Module[{empty = True, n = 1},
  While[empty && n ≤ Length[plist], empty = Norm[plist[[n]] - center] > alpha; n++]; empty]
```

functionAlpha[alpha_, plist_, {id1_, id2_}]

In[33]:=
```
ClearAll[functionAlpha];
functionAlpha[alpha_, plist_, {id1_, id2_}] := Module[
   {p1 = plist[[id1]], p2 = plist[[id2]], center1, center2, lhalf, center1emptyQ, center2emptyQ, property},
   lhalf = Norm[p2 - p1] / 2; (* the 1/2 length of the edge {p1,p2} *)
   center1 = (p2 + p1) / 2 + Sqrt[(alpha / lhalf)^2 - 1] {{0, -1}, {1, 0}}.((p2 - p1) / 2);
   center2 = (p2 + p1) / 2 + Sqrt[(alpha / lhalf)^2 - 1] {{0, 1}, {-1, 0}}.((p2 - p1) / 2);
   (* check that no other points are inside the circles at centre1 and centre2 *)
   center1emptyQ = emptyQ[center1, alpha, Delete[plist, {{id1}, {id2}}]];
   center2emptyQ = emptyQ[center2, alpha, Delete[plist, {{id1}, {id2}}]];
   (* assign 0 or 1 or 2 according to
    "0 for interior bar,1 for boundary bar,2 for bar to be erased",
   i.e., ..... *)
   Which[center1emptyQ ∧ center2emptyQ, 2, (! center1emptyQ) ∧ (! center2emptyQ),
    0, (center1emptyQ ∧ (! center2emptyQ)) ∨ ((! center1emptyQ) ∧ center2emptyQ), 1]
   (* Print[property];
   Show[Graphics[{Red,Circle[center1,alpha],Circle[center2,alpha]}],
    ListPlot[plist,AspectRatio→1 ] ] *)
  ] /; alpha > Norm[plist[[id1]] - plist[[id2]]] / 2
functionAlpha[alpha_, plist_, {id1_, id2_}] := 2 /; alpha < Norm[plist[[id1]] - plist[[id2]]] / 2
?functionAlpha
```

```
Global`functionAlpha
```

```
functionAlpha[alpha_, plist_, {id1_, id2_}] :=
 Module[{p1 = plist[[id1]], p2 = plist[[id2]], center1, center2,
    lhalf, center1emptyQ, center2emptyQ, property},
   lhalf = 1/2 Norm[p2 - p1]; center1 = (p2+p1)/2 + Sqrt[(alpha/lhalf)^2 - 1] {{0, -1}, {1, 0}}.(p2-p1)/2;
   center2 = (p2+p1)/2 + Sqrt[(alpha/lhalf)^2 - 1] {{0, 1}, {-1, 0}}.(p2-p1)/2;
   center1emptyQ = emptyQ[center1, alpha, Delete[plist, {{id1}, {id2}}]];
   center2emptyQ = emptyQ[center2, alpha, Delete[plist, {{id1}, {id2}}]];
   Which[center1emptyQ && center2emptyQ, 2, ! center1emptyQ && ! center2emptyQ, 0,
    (center1emptyQ && ! center2emptyQ) || (! center1emptyQ && center2emptyQ), 1]] /;
  alpha > 1/2 Norm[plist[[id1]] - plist[[id2]]]

functionAlpha[alpha_, plist_, {id1_, id2_}] := 2 /; alpha < 1/2 Norm[plist[[id1]] - plist[[id2]]]
```

pointSetDiameterFunction[data_]

In[37]:=
```
ClearAll[pointSetDiameterFunction]
pointSetDiameterFunction[data_] := Module[{convexhull, allConvexHullPairs},
  convexhull = ConvexHull[data];
  allConvexHullPairs = Map[(data[[#]]) &, Flatten[Outer[List, convexhull, convexhull], 1]];
  Max[Map[distance, allConvexHullPairs]]
 ]
?pointSetDiameterFunction
```

```
Global`pointSetDiameterFunction
```

```
pointSetDiameterFunction[data_] :=
 Module[{convexhull, allConvexHullPairs}, convexhull = ConvexHull[data];
  allConvexHullPairs = (data[[#1]] &) /@ Flatten[Outer[List, convexhull, convexhull], 1];
  Max[distance /@ allConvexHullPairs]]
```

Abstract.

This *Mathematica* 7 Notebook develops a novel regression method applied to bivariate data on a bounded space (we will work in the canonical {0,1}×{0,1} space). This approach was developed by [Moffiet, 2008] in the context of analysis of satellite remote sensing data. A distinguishing feature of the development presented here is that the X and Y variables are treated with complete symmetry; neither variable takes dependent or independent roles. The fitted "regression line", which we call a Lineal Basis, is described parametrically, i.e., as {X = X[s, …], y = y{ s, … }}, with 0 ≤ s ≤ 1 and the constraints {X[0, …], Y[0, …]} = {0, 0} and {X[1, …], Y[1, …]} = {1, 1}, rather than conventionally as Y = f[x, …] with f[0] = 0 and f[1] = 1 (ellipsis … denotes parameters).

This Lineal Basis is fitted to the data according to a model in which for each observed data point $\{x_i, y_i\}$ there is a corresponding "generating" point $\{X_i, Y_i\}$, *which lies on the Lineal Basis*. The difference vector between an $\{X_i, Y_i\}$ and its corresponding $\{x_i, y_i\}$is modelled as a sample from a bivariate distribution which here is taken as a product of two independent Beta[α, β] distributions, using the notation Beta[αX, βX]×Beta[αY, βY]. Further, we will allow αX, βX, αY, and βY to vary with s, i.e., we have four functions αX[s, …], βX[s, …], αY[s, …], and βY[s, …].

Since the mean of the Beta[α, β] distribution is α / (α + β), the parameters of these four function must be such that for a given s, the joint Beta-Beta mean point {αX[s, …] / (αX[s, …] + βX[s, …]), αY[s, …] / (αY[s, …] + βY[s, …]) lies on the Lineal Basis. It turns out that simple linear functions suffice to fit many of the data sets typically encountered in these bounded spaces.

Interesting computational issues arise when constructing the "mean prediction region" and the"single prediction region" for a Lineal Basis model, analogous to the "mean prediction bands" and the"single prediction bands" of simple linear regression. Concepts from computational geometry are employed, and in particular the logic of a key calculation is verified via a **Manipulate[]** (Appendix 1).

Section 1 - Data Setup.

The "Get Coordinates" menu item in the dropdown menu resulting from a right-click on any graphic is an easy way to generate the list of {x,y} coordinate pairs of a constructed set of points.
We set up the canonical fully bounded space in a simple plot, so that with "Get Coordinates" the coordinates of datasets of various shapes and sizes to be generated quickly and easily.

In[40]:=
```
ListPlot[{{0, 0}, {1, 1}},
  FrameLabel→{Style["X", 10, FontFamily→"Times"], Style["Y", 10, FontFamily→"Times"]}];
Labeled[%, Style["          Figure 1. The Canonical Bounded X-Y Space",
  13, FontFamily→"Book Antiqua"]]
```

Out[41]=

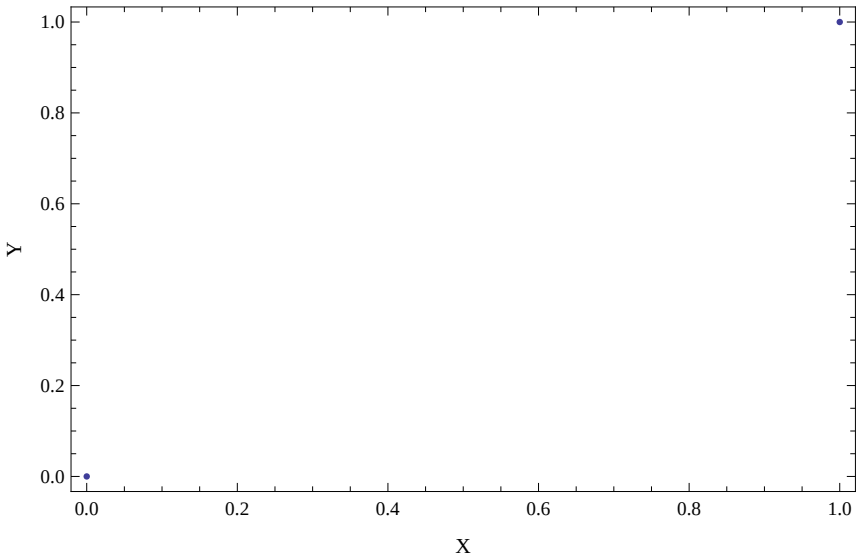


Figure 1. The Canonical Bounded X–Y Space

We will use the following data set in our development of Lineal Basis modelling.

In[42]:=
```
data = {{0.01866, 0.01223}, {0.003469, 0.03738}, {0.01562, 0.07261}, {0.03992, 0.08267}, {0.03992, 0.133}, {0.04904, 0.1581},
    {0.046, 0.2236}, {0.1159, 0.2236}, {0.08549, 0.2839}, {0.1311, 0.3645}, {0.1432, 0.3594}, {0.1736, 0.4097},
    {0.1705, 0.455}, {0.2343, 0.45}, {0.2252, 0.5053}, {0.2556, 0.5557}, {0.2799, 0.5557}, {0.2921, 0.5104}, {0.3285, 0.5255},
    {0.3073, 0.5959}, {0.3467, 0.6311}, {0.3771, 0.5959}, {0.365, 0.6462}, {0.4166, 0.6966}, {0.4136, 0.6311}, {0.4348, 0.6462},
    {0.45, 0.7519}, {0.4652, 0.7519}, {0.4865, 0.7922}, {0.5017, 0.7318}, {0.5351, 0.7318}, {0.5533, 0.7871}, {0.611, 0.8123},
    {0.5624, 0.8978}, {0.5898, 0.8475}, {0.6171, 0.7871}, {0.6475, 0.7871}, {0.6505, 0.8324}, {0.6991, 0.8324},
    {0.6748, 0.8727}, {0.7599, 0.8878}, {0.7234, 0.8878}, {0.7356, 0.9381}, {0.7508, 0.8727}, {0.769, 0.8777},
    {0.8024, 0.8978}, {0.8085, 0.918}, {0.8328, 0.8978}, {0.8419, 0.9129}, {0.8753, 0.9381}, {0.8905, 0.9532},
    {0.9057, 0.9582}, {0.927, 0.9632}, {0.9482, 0.9381}, {0.9634, 0.9481}, {0.9452, 0.9733}, {0.5928, 0.7418}, {0.526, 0.6815},
    {0.4926, 0.616}, {0.4227, 0.5657}, {0.3285, 0.4802}, {0.2283, 0.3594}, {0.1827, 0.3292}, {0.125, 0.294}, {0.1341, 0.4248},
    {0.1979, 0.5053}, {0.2404, 0.6211}, {0.2617, 0.6865}, {0.3194, 0.7318}, {0.371, 0.762}, {0.4348, 0.8073}, {0.4926, 0.8173},
    {0.5199, 0.8223}, {0.3164, 0.6815}, {0.289, 0.6211}, {0.6201, 0.8878}, {0.6384, 0.9029}, {0.7751, 0.933}} // Sort;
```

We store the size of the dataset in a variable for the later convenience in the calculation of degrees of freedom:

In[43]:=
```
nPoints = Length[ data ]
```

Out[43]= 78

Here is our development data set:

In[44]:=
```
dataPlot = ListPlot[ data ,
   FrameLabel → {Style[ "X", 10, FontFamily → "Times" ], Style[ "Y", 10, FontFamily → "Times" ]}];
Labeled[ %,  Style["          Figure 2. The Development Dataset", 13, FontFamily → "Book Antiqua"] ]
```

Out[45]=

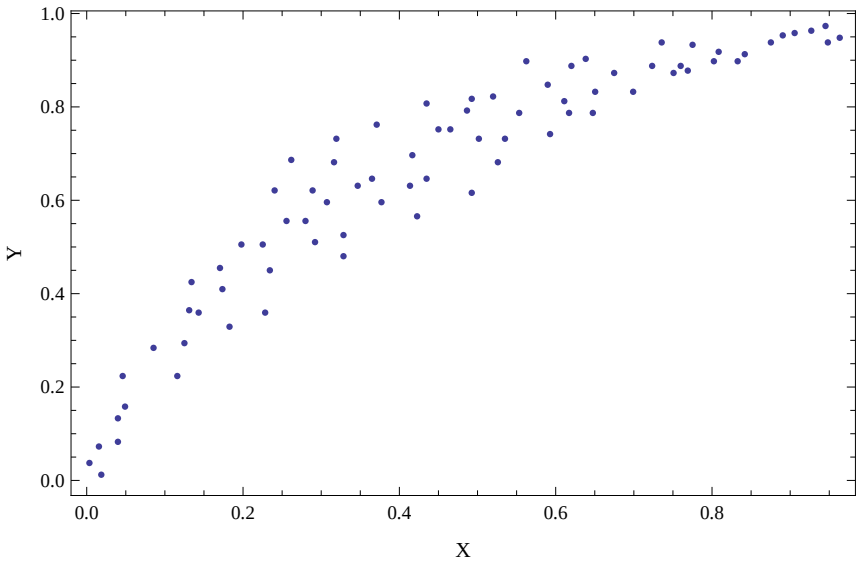


Figure 2. The Development Dataset

Section 2 - Modelling.

Introduction.

The essential feature of Lineal Basis modelling is that the treatment of the X and Y variables is completely symmetric. In orthodox linear regression, X and Y take the roles of an independent and a dependent variable respectively, the measurements of X are assumed to be without "noise" or "error", and in the case of simple linear regression for an observed data point {X, Y} the Y observation is modelled as a random sample from a Normal distribution N[a X + b, σ], where the mean is specified as the linear function a X + b of the abscissa X, with standard deviation σ that is constant for all relevant values of X (the "homoscedastic" assumption).

In Lineal Basis modelling, the underlying relationship between X and Y is modelled by a parametric form where a general point on the Lineal Basis—which will plot as a smooth curve from {0,0} to {1,1} in the canonical bounded space—is given as {X[s, …], Y[s, …]}, with s ranging from 0 to 1, similar to the way for example a curve is specified when being plotted by *Mathematica*'s **`ParametricPlot[]`** function.

A critical feature of Lineal Basis modelling is that, whereas in simple linear regression the "noise" or "error" is typically modelled by a univariate Normal distribution, here the "noise" or "error"—we call it the "observation" error—is modelled by a bivariate distribution. For this development the bivariate distribution is a product of two independent Beta distributions, i.e., a Beta-Beta distribution. For each observed data point we assume there is a corresponding "generating" point, and the observed data point is a random sample from a Beta-Beta *whose mean—the corresponding "generating" point—lies somewhere on the Lineal Basis*. Further, we will assume that, writing the bivariate distribution more fully as

Beta[α_X, β_X]×Beta[α_Y, β_Y],

the α_X, β_X, α_Y, and β_Y are all function of s (and other parameters), so that a more complete form for the Beta-Beta distribution is

Beta[α_X[s, …], β_X[s, …]]×Beta[α_Y[s, …], β_Y[s, …]], the ellipsis … denoting other parameters.

Since the mean of the Beta[α, β] distribution is $\frac{\alpha}{\alpha+\beta}$, and the joint mean of the Beta-Beta must be {0,0} when s=0 and {1,1} when s=1, the functions α_X[s, …], β_X[s, …], α_Y[s, …], and β_Y[s, …] are constrained such that:

(i) α_X[0, …] / (α_X[0, …] + β_X[0, …]) = 0, and α_Y[1, …] / (α_Y[1, …] + β_Y[1, …]) = 0; and
(ii) α_X[1, …] / (α_X[1, …] + β_X[1, …]) = 1, and α_Y[1, …] / (α_Y[1, …] + β_Y[1, …]) = 1.

In developing Lineal Basis modelling we have found that these requirements can be met most simply with linear functions of the form

α_X[s, …] = a s, β_X[s, …] = b (1 - s), α_Y[s, …] = c s, and β_Y[s, …] = d (1 - s).

The two resulting mean expressions are then

$\frac{a\,s}{a\,s+b(1-s)}$ and $\frac{c\,s}{c\,s+d(1-s)}$.

We see immediately that we can for example divide though the first expression by b and the second expression by d, showing that each expression has only one free parameter. For simplicity we will rewrite these expressions by setting b and d both to 1. Failure to notice the essential one-free-parameter form of these expressions, i.e., attempts to obtain absolute values of a, b, c, and d leads to numerical instabilities warning of the ill-conditioning of the problem of fitting the 4 parameters.

The complete Lineal Basis model thus requires, for N data points, the determination of parameters a and c—jointly governing the shape of the Lineal Basis—and N parameters s_i that determine the positions on the Lineal Basis of the N "generating" points. Note that we are thus employing 2 N data values (N coordinate pairs) to fit N+2 parameters, so the problem is comfortably over-determined.

Lineal Basis Model Setup.

This constructs an indexed list of (N=78) s parameters:

In[46]:=
```
sParams = Array[ s, nPoints ]
```

Out[46]=
```
{s[1], s[2], s[3], s[4], s[5], s[6], s[7], s[8], s[9], s[10], s[11], s[12], s[13],
 s[14], s[15], s[16], s[17], s[18], s[19], s[20], s[21], s[22], s[23], s[24], s[25], s[26],
 s[27], s[28], s[29], s[30], s[31], s[32], s[33], s[34], s[35], s[36], s[37], s[38], s[39],
 s[40], s[41], s[42], s[43], s[44], s[45], s[46], s[47], s[48], s[49], s[50], s[51], s[52],
 s[53], s[54], s[55], s[56], s[57], s[58], s[59], s[60], s[61], s[62], s[63], s[64], s[65],
 s[66], s[67], s[68], s[69], s[70], s[71], s[72], s[73], s[74], s[75], s[76], s[77], s[78]}
```

We will need a list of all the parameters to be fitted:

In[47]:=
```
allParams = {a, c} ~ Join ~ sParams;
% // Short
```

Out[48]//Short=
```
{a, c, s[1], s[2], s[3], s[4], s[5], s[6], s[7], s[8], <<60>>,
 s[69], s[70], s[71], s[72], s[73], s[74], s[75], s[76], s[77], s[78]}
```

Here we construct the one-parameter form of the expression for the mean of the X-component of the Beta-Beta distributions:

In[49]:=
$$\frac{a\,s}{a\,s + b\,(1 - s)} /. \{b \to 1\}$$

Out[49]=
$$\frac{a\,s}{1 - s + a\,s}$$

This form has the required properties, i.e., it is 0 at s=0 and 1 at s=1:

In[50]:=
$$\left\{\frac{a\,s}{a\,s + b\,(1 - s)} /. \{b \to 1,\ s \to 0\},\ \frac{a\,s}{a\,s + b\,(1 - s)} /. \{b \to 1,\ s \to 1\}\right\}$$

Out[50]=
```
{0, 1}
```

By mapping the appropriate pure function over the indexed list of s parameters we get a list of symbolic modelled X-coordinates of the N "generating" points:

In[51]:=
$$\text{Xmodelled} = \text{Map}\left[\left(\frac{a\,\#}{1 - \# + a\,\#}\right)\&,\ \text{sParams}\right];$$
```
% // Short
```

Out[52]//Short=
$$\left\{\frac{a\,s[1]}{1 - s[1] + a\,s[1]},\ \frac{a\,s[2]}{1 - s[2] + a\,s[2]},\ \text{<<74>>},\ \frac{a\,s[77]}{1 - s[77] + a\,s[77]},\ \frac{a\,s[78]}{1 - s[78] + a\,s[78]}\right\}$$

We now construct the one-parameter form of the expression for the mean of the Y-component of the Beta-Beta distributions:

In[53]:=
$$\frac{c\,s}{c\,s + d\,(1 - s)} /. \{d \to 1\}$$

Out[53]=
$$\frac{c\,s}{1 - s + c\,s}$$

This form also has the required properties:

In[54]:=
$$\left\{\frac{c\,s}{c\,s + d\,(1 - s)} /. \{d \to 1,\ s \to 0\},\ \frac{c\,s}{c\,s + d\,(1 - s)} /. \{d \to 1,\ s \to 1\}\right\}$$

Out[54]=
```
{0, 1}
```

Again mapping the appropriate pure function over the indexed list of s parameters we get a list of symbolic modelled Y-coordinates of the N "generating" points:

In[55]:=
```
Ymodelled = Map[ (c #)/(1 - # + c #) &, sParams ];
% // Short
```

Out[56]//Short=

$$\left\{\frac{c\,s[1]}{1-s[1]+c\,s[1]},\ \frac{c\,s[2]}{1-s[2]+c\,s[2]},\ \text{<<74>>},\ \frac{c\,s[77]}{1-s[77]+c\,s[77]},\ \frac{c\,s[78]}{1-s[78]+c\,s[78]}\right\}$$

Given the unusual form of our model, we will proceed to fit it to the development data by finding the values of a, c, and the s_i parameters such that the value of an expression which can be conveniently referred to as the total sum-of-squared-errors is minimized.

It is most important to note at this point that, in the fully Bayesian context, this model is fitted using MCMC methods implemented in WinBUGs [WinBUGS, 2010], and the description and fitting of the model is somewhat different. Three further parameters are introduced that essentially account for the absolute values of our earlier parameters a, b, c, and d, and allow the generation of samples from the Bayesian posterior distributions of all parameters to proceed without numerical difficulties.

This produces the complete total sum-of-squared-errors expression (only partly shown):

In[57]:=
```
sseXY = Total[ (data[[All, 1]] - Xmodelled)^2 ] + Total[ (data[[All, 2]] - Ymodelled)^2 ];
% // Short
```

Out[58]//Short=

$$\left(0.003469-\frac{a\,s[1]}{1-s[1]+a\,s[1]}\right)^2+\left(0.03738-\frac{\text{<<1>>}}{\text{<<1>>}}\right)^2+$$

$$(\text{<<1>>})^2+\text{<<150>>}+(\text{<<1>>}\ \text{<<1>>}^2+(\text{<<7>>}-\text{<<1>>})^2+\left(0.9481-\frac{c\ \text{<<1>>}}{\text{<<1>>}}\right)^2$$

Model Fitting.

Mathematica's **FindMinimum[]** function finds the minimising values of a, c, and the 78 s_i's almost instantly:

In[59]:=
```
{sseXYMin, solSSExy} = FindMinimum[ {sseXY, a > 0, c > 0}, allParams, Method → "Automatic" ]
```

Out[59]=
```
{0.146142, {a → 0.767195, c → 2.42107, s[1] → 0.0146657, s[2] → 0.0301212, s[3] → 0.00690679, s[4] → 0.0376475,
  s[5] → 0.0585306, s[6] → 0.098851, s[7] → 0.0707359, s[8] → 0.134633, s[9] → 0.112926, s[10] → 0.148709,
  s[11] → 0.185055, s[12] → 0.216086, s[13] → 0.18585, s[14] → 0.242971, s[15] → 0.220549, s[16] → 0.181527,
  s[17] → 0.278672, s[18] → 0.288962, s[19] → 0.211287, s[20] → 0.262643, s[21] → 0.355679,
  s[22] → 0.328148, s[23] → 0.39804, s[24] → 0.338698, s[25] → 0.377637, s[26] → 0.319177, s[27] → 0.372965,
  s[28] → 0.421978, s[29] → 0.448325, s[30] → 0.318157, s[31] → 0.344177, s[32] → 0.411343,
  s[33] → 0.428959, s[34] → 0.490163, s[35] → 0.408236, s[36] → 0.44779, s[37] → 0.483904,
  s[38] → 0.416312, s[39] → 0.468309, s[40] → 0.547566, s[41] → 0.530998, s[42] → 0.540293,
  s[43] → 0.572267, s[44] → 0.487536, s[45] → 0.587279, s[46] → 0.553671, s[47] → 0.606807,
  s[48] → 0.544596, s[49] → 0.575623, s[50] → 0.613329, s[51] → 0.665221, s[52] → 0.663487,
  s[53] → 0.619999, s[54] → 0.663596, s[55] → 0.65725, s[56] → 0.699727, s[57] → 0.717768,
  s[58] → 0.678937, s[59] → 0.699442, s[60] → 0.731901, s[61] → 0.73427, s[62] → 0.77171, s[63] → 0.797068,
  s[64] → 0.786242, s[65] → 0.797901, s[66] → 0.801142, s[67] → 0.823271, s[68] → 0.831887,
  s[69] → 0.842517, s[70] → 0.854006, s[71] → 0.865113, s[72] → 0.896365, s[73] → 0.911323,
  s[74] → 0.923474, s[75] → 0.93991, s[76] → 0.955312, s[77] → 0.948786, s[78] → 0.962146}}
```

In[60]:=
```
allParamsPointEstimates = allParams /. solSSExy;
```

In order to display the fitted Lineal Basis we need a list of the fitted "generating" points:

In[61]:=
```
dataModelled = Transpose[ {Xmodelled , Ymodelled } ] /. solSSExy;
% // Short
```

Out[62]//Short=
```
{{0.01129, 0.0347819}, <<76>>, {0.95122, 0.98401}}
```

If desired, we can easily fit a second order `InterpolatingFunction` through the list of fitted "generating" points:

In[63]:=
```
dataModelledInterpolation = Interpolation[ dataModelled, InterpolationOrder → 2 ]
```

Out[63]=
```
InterpolatingFunction[{{0.00530739, 0.95122}}, <>]
```

Here we plot the data points, the "generating", the fitted Lineal Basis, and an extrapolation to {0,0} and {1,1}.

In[64]:=
```
Off[InterpolatingFunction::"dmval"]
dataModelledInterpolationPlot =
  Plot[dataModelledInterpolation[s], {s, 0, 1}, PlotStyle → {Gray, Thickness[0.015], Opacity[0.2]}];
fittedLinealBasisPlot = ListPlot[dataModelled // Sort, Joined → True,
   PlotStyle → {Blue}, PlotRange → {{0, 1}, {0, 1}}];
generatingPointsPlot = ListPlot[dataModelled, PlotStyle → {Red}, PlotRange → {{0, 1}, {0, 1}}];
Show[{dataPlot, dataModelledInterpolationPlot, fittedLinealBasisPlot, generatingPointsPlot}];
Labeled[%, Style[
  "        Figure 3. Data & Generating Points, with Extrapolation", 13, FontFamily → "Book Antiqua"]]
```

Out[69]=

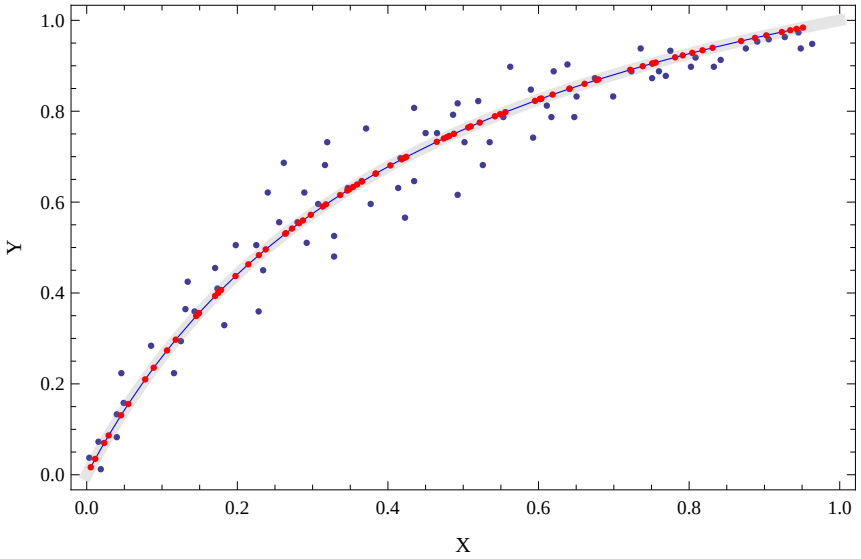


Figure 3. Data & Generating Points, with Extrapolation

Here we have connected each data point with its corresponding "generating" point. The minimum value of the total-sum-of-squared-errors is the sum of the squared lengths of all these connecting lines.

In[70]:=
```
lines = Map[Line, Transpose[{data, dataModelled}]];
Show[{dataPlot, generatingPointsPlot}, Epilog → {lines}];
Labeled[%, Style[
  "        Figure 4. Data & Corresponding Generating Points Connected", 13, FontFamily → "Book Antiqua"]]
```

Out[72]=

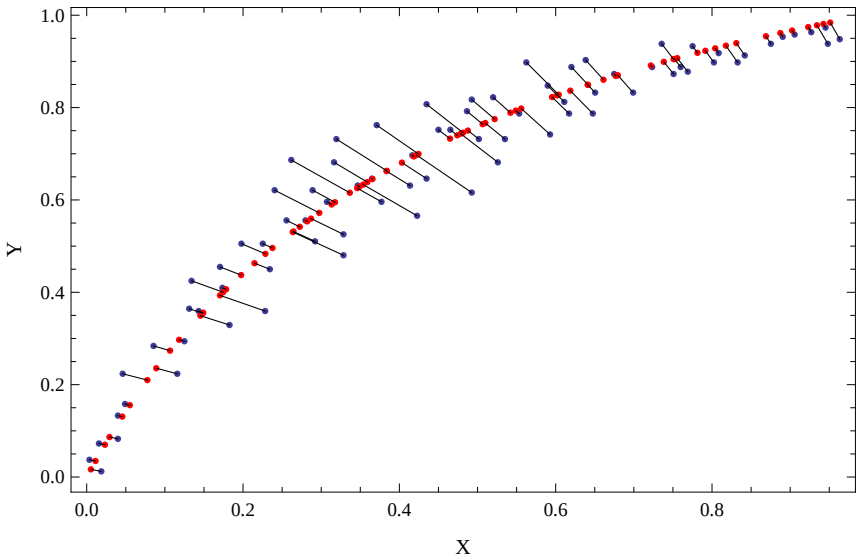


Figure 4. Data & Corresponding Generating Points Connected

These are the fitted s_i values:

In[73]:=
```
ListPlot[ sParams /. solSSExy];
Labeled[ %,  Style["        Figure 5. The Fitted s_i Values", 13, FontFamily → "Book Antiqua"] ]
```

Out[74]=

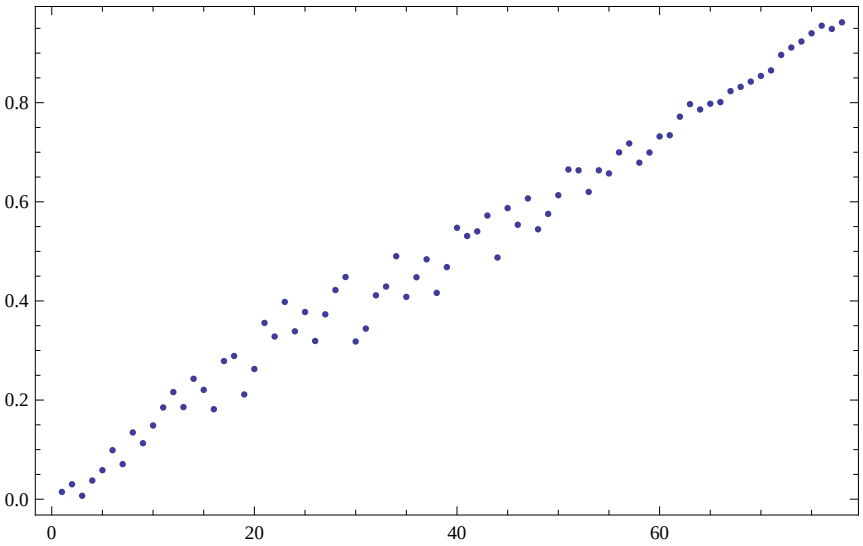


Figure 5. The Fitted s_i Values

Note that by the construction of the Lineal Basis model, we have $0 \le s_i \le 1$ for all s_i.

In[75]:=
```
{Min[ sParams /. solSSExy ], Max[ sParams /. solSSExy ]}
```

Out[75]=
```
{0.00690679, 0.962146}
```

Model Precision.

In numerical minimisation, the Hessian is the symmetric matrix of second derivatives of the objective function with respect to the independent variables, evaluated at the minimum. If the objective function is, say f(x, y, z, w), the Hessian matrix H is

$$H = \begin{pmatrix} \partial_{x,x} f & \partial_{x,y} f & \partial_{x,z} f & \partial_{x,w} f \\ \partial_{y,x} f & \partial_{y,y} f & \partial_{y,z} f & \partial_{y,w} f \\ \partial_{z,x} f & \partial_{z,y} f & \partial_{z,z} f & \partial_{z,w} f \\ \partial_{w,x} f & \partial_{w,y} f & \partial_{w,z} f & \partial_{w,w} f \end{pmatrix}$$ evaluated at {x, y, z, w} = {x_{min}, y_{min}, z_{min}, w_{min}}.

In nonlinear model fitting, to quote [Gregory, 2005], "… , we expect that sufficiently close to χ^2_{min}, χ^2 will be approximately quadratic so we should be able to ignore higher order terms in the Taylor expansion" (χ^2 here corresponds to our "total-sum-of-squared-errors"). This assumption is that in the region of the minimum, the objective function surface or space is locally quadratic. The *Mathematica* documentation reveals that **`FindMinimum[]`** has essentially five different ways of setting up such a "local quadratic model".

MLE (Maximum Likelihood Estimation) theory shows that under the weak assumption that the ML estimator is consistent (which we assume here), asymptotically the distribution of the MLE estimator is multivariate Normal with mean the vector of true parameter values, and variance-covariance matrix the inverse of the Hessian H, i.e., an estimate of the variance-covariance matrix associated with the estimates {x_{min}, y_{min}, z_{min}, w_{min}} is H^{-1}.

In a statistical model fitting application, the Hessian of the "total-sum-of-squared-errors" is proportional to the Hessian of the likelihood.

We can calculate the Hessian using the defined function **`hessianH[]`** (or else *mathStatica*'s **`Hessian[]`**), and apply the appropriate scaling factor. We find its determinant, and check its **`Dimensions[]`**.
Since the Hessian is an 80×80 matrix, we show only the upper-left 10×10 submatrix:

In[76]:=
```
hessian = (2 sseXYMin / (nPoints - 2))^-1 × hessianH[ sseXY, allParams ] /. solSSExy;
Take[ hessian, {1, 10}, {1, 10} ] // MatrixForm
Det[ hessian ]
Dimensions @ hessian
```

Out[77]//MatrixForm=

2508.57	0	9.84756	15.8363	−4.13809	9.59904	25.9487	53.5642	30.9616	62.0167
0	217.202	15.547	30.0032	10.4399	38.5235	49.6893	63.1043	55.6847	71.1174
9.84756	15.547	3127.38	0	0	0	0	0	0	0
15.8363	30.0032	0	2902.49	0	0	0	0	0	0
−4.13809	10.4399	0	0	3222.01	0	0	0	0	0
9.59904	38.5235	0	0	0	2777.89	0	0	0	0
25.9487	49.6893	0	0	0	0	2544.97	0	0	0
53.5642	63.1043	0	0	0	0	0	2177.26	0	0
30.9616	55.6847	0	0	0	0	0	0	2413.06	0
62.0167	71.1174	0	0	0	0	0	0	0	1887.11

Out[78]= 4.8565×10^{238}

Out[79]= {80, 80}

As noted, the required variance-covariance matrix is H^{-1}. We ensure numerical symmetry by simply averaging H^{-1} and its **Transpose[]**. We check that it is indeed symmetric, and also positive definite, which is a requirement of a variance-covariance matrix and necessary for statements such as **RandomReal[Multinormal[mean,cvm]]** to execute without error.
Again we show only the upper-left 10×10 submatrix:

In[80]:=
```
cvm = (Inverse[ hessian ] + Transpose[ Inverse[ hessian ] ]) / 2;
Take[ cvm, {1, 10}, {1, 10} ] // MatrixForm
{SymmetricMatrixQ[ cvm ], PositiveDefiniteMatrixQ[cvm ]}
```

Out[81]//MatrixForm=

0.286654	0.902124	−0.00538732	−0.0108893	−0.00255488	−0.0135011	−0.0205362	−0.0331987	−0.0
0.902124	2.84911	−0.0170043	−0.0343735	−0.008073	−0.0426285	−0.0648256	−0.10477	−0.0
−0.00538732	−0.0170043	0.000421253	0.000205168	0.0000481779	0.00025443	0.00038693	0.000625379	0.000
−0.0108893	−0.0343735	0.000205168	0.000759265	0.0000973906	0.000514316	0.000782153	0.00126415	0.000
−0.00255488	−0.008073	0.0000481779	0.0000973906	0.000333242	0.000120784	0.000183671	0.000296836	0.000
−0.0135011	−0.0426285	0.00025443	0.000514316	0.000120784	0.000997807	0.000969957	0.00156766	0.00
−0.0205362	−0.0648256	0.00038693	0.000782153	0.000183671	0.000969957	0.00186801	0.00238408	0.00
−0.0331987	−0.10477	0.000625379	0.00126415	0.000296836	0.00156766	0.00238408	0.00431263	0.0
−0.0244958	−0.0773224	0.000461523	0.000932937	0.000219077	0.00115694	0.00175944	0.0028437	0.00
−0.0434177	−0.137018	0.000817868	0.00165325	0.0003882	0.00205018	0.0031179	0.00503939	0.00

Out[82]= {True, True}

In order to see the relationships between the various parameter more clearly, we can apply the function **varCovarMatrixtoCorrMatrix[]** defined above to **cvm** to see the correlations between the parameters, showing the upper-left 10×10 submatrix only:

In[83]:=
```
corrm = varCovarMatrixtoCorrMatrix[ cvm ];
Take[ corrm, {1, 10}, {1, 10} ] // MatrixForm
```

Out[84]//MatrixForm=

1.	0.998233	−0.490255	−0.738115	−0.261403	−0.798301	−0.887468	−0.944215	−0.912668	−0.961027
0.998233	1.	−0.490832	−0.739047	−0.261999	−0.799507	−0.888592	−0.945176	−0.913799	−0.96199
−0.490255	−0.490832	1.	0.362779	0.128587	0.392441	0.436186	0.463981	0.448562	0.472236
−0.738115	−0.739047	0.362779	1.	0.193616	0.590895	0.656758	0.698605	0.675392	0.711034
−0.261403	−0.261999	0.128587	0.193616	1.	0.209462	0.232794	0.247609	0.239397	0.252013
−0.798301	−0.799507	0.392441	0.590895	0.209462	1.	0.710461	0.755716	0.730617	0.769161
−0.887468	−0.888592	0.436186	0.656758	0.232794	0.710461	1.	0.839966	0.812056	0.85491
−0.944215	−0.945176	0.463981	0.698605	0.247609	0.755716	0.839966	1.	0.863799	0.909398
−0.912668	−0.913799	0.448562	0.675392	0.239397	0.730617	0.812056	0.863799	1.	0.879167
−0.961027	−0.96199	0.472236	0.711034	0.252013	0.769161	0.85491	0.909398	0.879167	1.

This reveals some important details:
(i) the parameters "a" and "c" are very highly correlated, which is clear by inspection of the symbolic **sseXY** as there is a term involving "a" and a term involving "c" for each data point;
(ii) the s_i parameters are all positively correlated with each other; and
(iii) "a" and "c" are generally highly negatively correlated with the s_i parameters.

It is always a good idea to check a calculated matrix inverse; we look at the upper-left 10×10 submatrices only:

In[85]:=
```
Take[cvm.hessian, {1, 10}, {1, 10}] // MatrixForm // Chop
```

Out[85]//MatrixForm=

1.	0	0	0	0	0	0	0	0	0
0	1.	0	0	0	0	0	0	0	0
0	0	1.	0	0	0	0	0	0	0
0	0	0	1.	0	0	0	0	0	0
0	0	0	0	1.	0	0	0	0	0
0	0	0	0	0	1.	0	0	0	0
0	0	0	0	0	0	1.	0	0	0
0	0	0	0	0	0	0	1.	0	0
0	0	0	0	0	0	0	0	1.	0
0	0	0	0	0	0	0	0	0	1.

In[86]:=
```
Take[cvm.hessian, {71, 80}, {71, 80}] // MatrixForm // Chop
```

Out[86]//MatrixForm=

1.	0	0	0	0	0	0	0	0	0
0	1.	0	0	0	0	0	0	0	0
0	0	1.	0	0	0	0	0	0	0
0	0	0	1.	0	0	0	0	0	0
0	0	0	0	1.	0	0	0	0	0
0	0	0	0	0	1.	0	0	0	0
0	0	0	0	0	0	1.	0	0	0
0	0	0	0	0	0	0	1.	0	0
0	0	0	0	0	0	0	0	1.	0
0	0	0	0	0	0	0	0	0	1.

Parameter Confidence Regions.

Following are plots of samples from the bivariate "uncertainty" distributions for selected pairs of parameters.
The parameters "a" and "c" are shown to be very strongly correlated, as we have seen:

In[87]:=
```
ListPlot[ Table[ restrictedParameters[ allParamsPointEstimates,  cvm][[ {1, 2} ]], {nSamples = 1000} ],
  AspectRatio → 1 ,  PlotRange → All, ImageSize → 300,
  FrameLabel → {Style[ "a", 12, FontFamily → "Times" ],  Style[ "c", 12, FontFamily → "Times" ]} ];
Labeled[ %,  Style[ StringForm[ "Figure `1`. Parameter Correlation Plot: c
     vs a (`2` samples)\n                                    (correlation  = `3`)",
  7,  nSamples, NumberForm[ corrm[[ 1, 2 ]],  3 ] ], 13, FontFamily → "Book Antiqua"] ]
```

Out[88]=

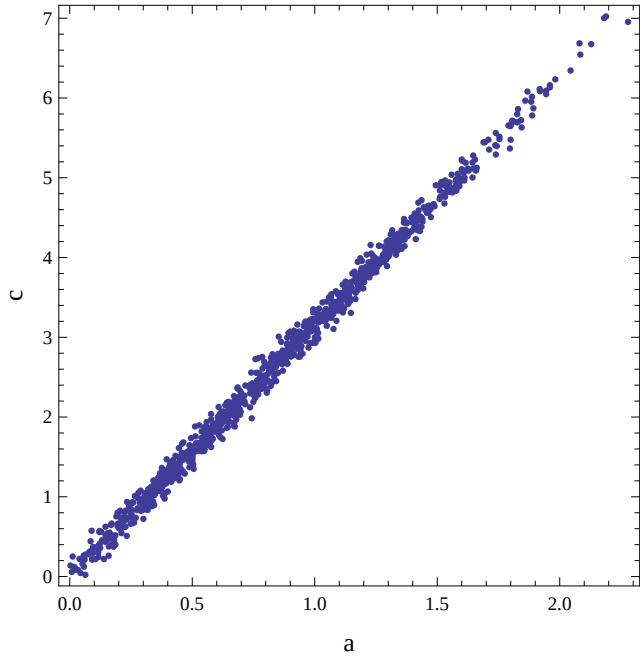


Figure 7. Parameter Correlation Plot: c vs a (1000 samples)
(correlation = 0.998)

This shows the relation between "a" and "s_1", the positional parameter for the first "generating" point:

In[89]:=
```
ListPlot[ Table[ restrictedParameters[ allParamsPointEstimates,  cvm][[ {1, 3} ]], {nSamples = 1000} ],
  AspectRatio → 1 ,  PlotRange → All, ImageSize → 300,
  FrameLabel → {Style[ "a", 12, FontFamily → "Times" ],  Style[ "s₁", 12, FontFamily → "Times" ]} ];
Labeled[ %,  Style[ StringForm[ "Figure `1`. Parameter Correlation Plot: s₁
     vs a (`2` samples)\n                                    (correlation  = `3`)",
  8,  nSamples, NumberForm[ corrm[[ 1, 3 ]],  3 ] ], 13, FontFamily → "Book Antiqua"] ]
```

Out[90]=

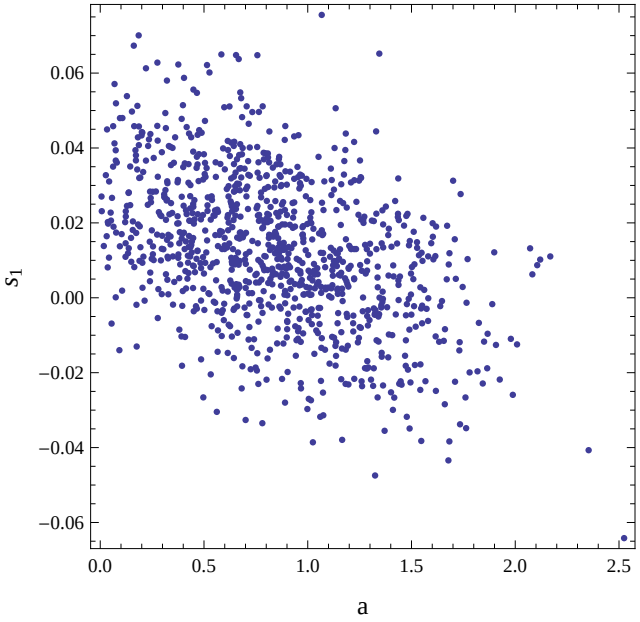


Figure 8. Parameter Correlation Plot: s_1 vs a (1000 samples)
(correlation = −0.49)

This shows the relation between "s_1" and "s_2":

In[91]:=
```
ListPlot[ Table[ restrictedParameters[ allParamsPointEstimates,  cvm][[ {3, 4} ]], {nSamples = 1000} ],
  AspectRatio → 1 ,  PlotRange → All, ImageSize → 300,
  FrameLabel → {Style[ "s1", 12, FontFamily → "Times" ],  Style[ "s2", 12, FontFamily → "Times" ]} ];
Labeled[ %,  Style[ StringForm[ "Figure `1`. Parameter Correlation Plot: s2
     vs s1 (`2` samples)\n                                    (correlation  = `3`)",
  9,  nSamples, NumberForm[ corrm[[ 3, 4 ]],  3 ] ], 13, FontFamily → "Book Antiqua"] ]
```

Out[92]=

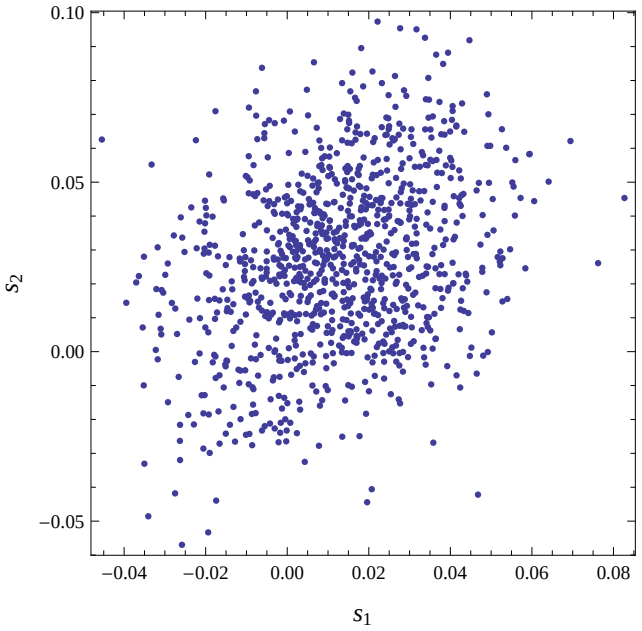


Figure 9. Parameter Correlation Plot: s_2 vs s_1 (1000 samples) (correlation = 0.363)

And finally this shows the relation between "s_{77}" and "s_{78}", the last two s_i parameters:

In[93]:=
```
ListPlot[ Table[ restrictedParameters[ allParamsPointEstimates,  cvm][[ {nPoints - 1, nPoints} ]],
    {nSamples = 1000} ],  AspectRatio → 1 ,  PlotRange → All, ImageSize → 300,
  FrameLabel → {Style[ "s77", 12, FontFamily → "Times" ],  Style[ "s78", 12, FontFamily → "Times" ]} ];
Labeled[ %,  Style[ StringForm[ "Figure `1`. Parameter Correlation Plot: s78 vs
     s77 (`2` samples)\n                                (correlation  = `3`)",
  10,  nSamples, NumberForm[ corrm[[ 77, 78 ]],  3 ] ], 13, FontFamily → "Book Antiqua"] ]
```

Out[94]=

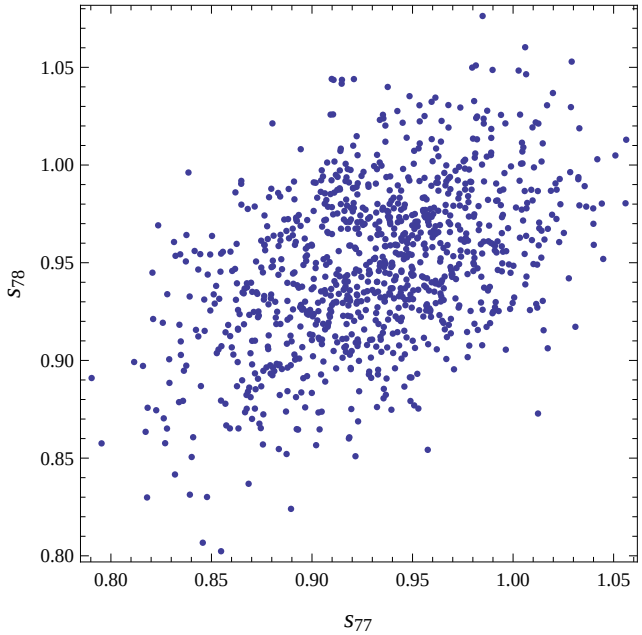


Figure 10. Parameter Correlation Plot: s_{78} vs s_{77} (1000 samples)
(correlation = 0.516)

The "Mean Prediction" Region.

We are now in a position to generate random samples from the "uncertainty" distributions for the a, c, and s_i parameters. Plotting the resulting Lineal Bases together will give a clear picture of the extent of the 95% "mean prediction" region.
The point estimates list is the mean vector of the asymptotic multivariate Normal distribution—with variance-covariance matrix **cvm**—from which we wish to sample:

In[95]:=
```
allParamsPointEstimates = allParams /. solSSExy;
% // Short
```

Out[96]//Short=
```
{0.767195, 2.42107, <<76>>, 0.948786, 0.962146}
```

This constructs a list of replacement rules which we will use repeatedly to generate sampled Lineal Bases:

In[97]:=
```
parameterSubstitutionRule = Map[ (#[[1]] → #[[2]]) &,
   Transpose[ {allParams, RandomReal[ MultinormalDistribution[ allParamsPointEstimates,  cvm], 1 ][[1]]} ]];
% //
 Short
```

Out[98]//Short=
```
{a → 0.415477, c → 1.14601, <<76>>, s[77] → 0.963705, s[78] → 0.957755}
```

First we generate a single sampled Lineal Basis and plot it along with the original data:

In[99]:=
```
parameterSubstitutionRule = Map[ (#[[1]] → #[[2]]) &,
   Transpose[ {allParams, RandomReal[ MultinormalDistribution[ allParamsPointEstimates,  cvm], 1 ][[1]]} ]];
Print[ "{a, c} = ", {a, c} /. parameterSubstitutionRule ]
Show[ dataPlot, ListPlot[ Transpose[ {Xmodelled, Ymodelled} ] /. parameterSubstitutionRule // Sort,
   Joined → True, PlotStyle → {Orange} ] ,
  FrameLabel → {Style[ "X", 10, FontFamily → "Times" ], Style[ "Y", 10, FontFamily → "Times" ]},
  PlotRange → {{0, 1}, {0, 1}}];
Labeled[ %,  Style[ StringForm[ "Figure `1`. Single Mean Prediction Plot for a
     Lineal Basis Model\n                                    {a,c = {`2`, `3`}}", 6,
   NumberForm[ a /. parameterSubstitutionRule, 3 ], NumberForm[ c /. parameterSubstitutionRule, 3 ] ],
  13, FontFamily → "Book Antiqua"] ]
```

{a, c} = {0.449359, 1.50926}

Out[102]=

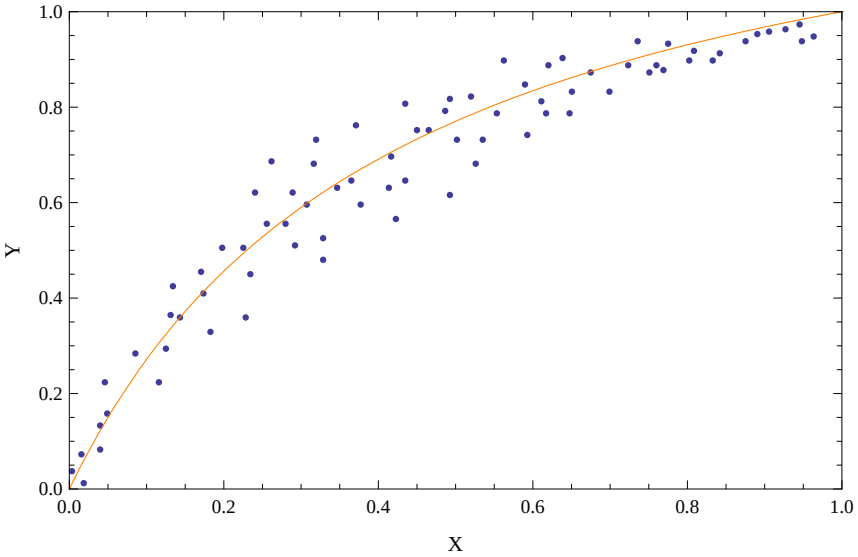


Figure 6. Single Mean Prediction Plot for a Lineal Basis Model
{a,c = {0.449, 1.51}}

Note that in the preceding code we are drawing a single multivariate sample from an unrestricted distribution via **`RandomReal[MultinormalDistribution[allParamsPointEstimates,cvm],1]`**.
On occasion we can obtain a plot such as the following, due to one or both (in this case) of a and c being negative.

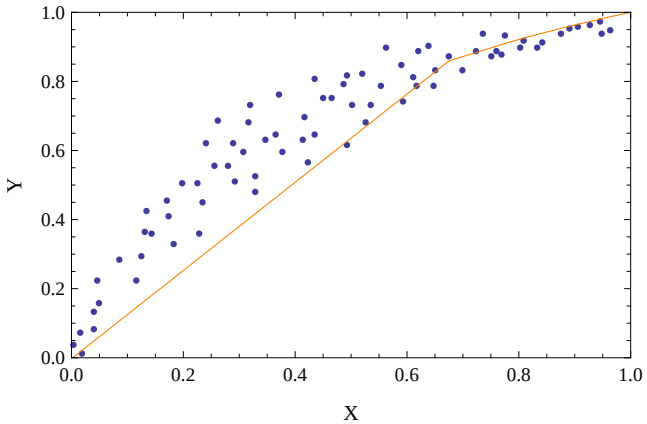


Figure 6a. Single Mean Prediction Plot for a Lineal Basis Model
{a,c = {−0.179, −0.525}}

To avoid this we have coded the defined function **`restrictedParameters[]`** above, which samples from **`MultinormalDistribution[allParamsPointEstimates,cvm]`** until a>0 and c>0. We use this function in visualising some of the correlations between the a, c, and s_i parameters.

Here finally we can show a set of models where the "a", "b", and "s_i" parameters for each model were drawn from the (restricted) multivariate Normal distribution with mean vector equal to the vector of minimum sum-of-squared-errors parameter values, and variance-covariance matrix derived above via the inverse of the scaled Hessian.
The scatter of models shown here is directly analogous the the familiar "hyperbolic hourglass" shape of the 95% "mean prediction bands" in standard simple linear regression.

In[103]:=
```
meanPredictionPlot = Show[ ListPlot[
     Map[ (Transpose[ {Xmodelled, Ymodelled} ] /. Map[ (#[[1]] → #[[2]]) &,  Transpose[ {allParams, #} ]] // Sort) &,
      Table[ restrictedParameters[ allParamsPointEstimates,  cvm ], {nSamples = 200} ] ] ,
     Joined → True,  PlotStyle → {{Opacity[ 0.3 ], Orange}} ],
   fittedLinealBasisPlot, dataPlot,  PlotRange → {{0, 1}, {0, 1}},
   FrameLabel → {Style[ "X", 10, FontFamily → "Times" ],  Style[ "Y", 10, FontFamily → "Times" ]} ];
Labeled[ %,  Style[ StringForm[
   "Figure `1`. Mean Prediction Plot for a Lineal Basis Model: (`2` samples)",
   11,  nSamples ], 13, FontFamily → "Book Antiqua"] ]
```

Out[104]=

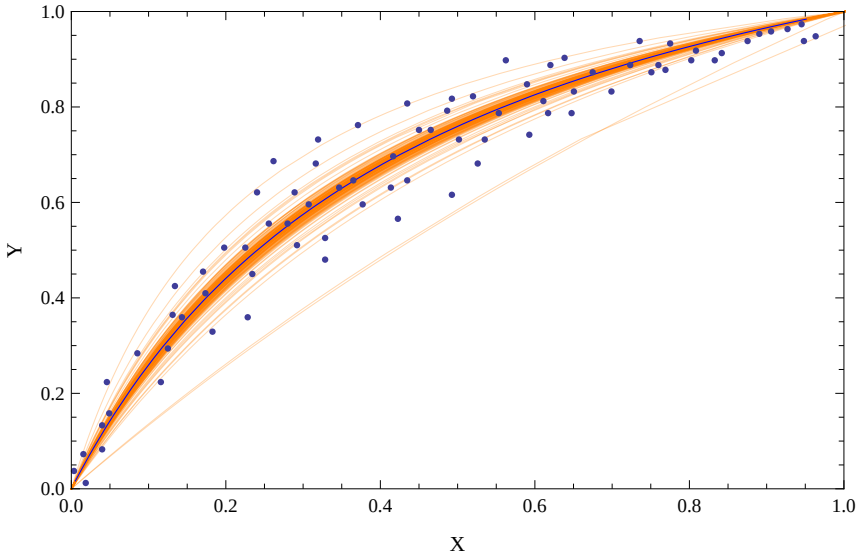


Figure 11. Mean Prediction Plot for a Lineal Basis Model: (200 samples)

Section 3 - Model Exploration.

Recall that the analytic form of these Beta-Beta means was derived from the assumption that sufficiently flexible forms for the functions αX[s, …] etc., were linear: α_X[s, …] = a s, β_X[s, …] = b (1 - s), α_Y[s, …] = c s, and β_Y[s, …] = d (1 - s).

We saw that the mean point of the Beta-Beta would then take the form $\{\frac{a\,s}{a\,s+b(1-s)}, \frac{c\,s}{c\,s+d(1-s)}\}$, but because a rescaling of the {a,b} or {c,d} pairs leaves the mathematical form of these expressions unchanged, numerical optimization required reducing each expression to an equivalent one-parameter form.

For any value of s, 0≤ s ≤ 1, the mean point of the Beta-Beta distribution was thus specified as

$$\left\{\frac{a\,s}{1-s+a\,s}, \frac{a\,s}{1-s+a\,s}\right\}.$$

Substituting the solution vales of a and c, we get …

In[105]:=
```
{a s / (1 - s + a s), c s / (1 - s + c s)} /. solSSExy
```

Out[105]=
$$\left\{\frac{0.767195\,s}{1-0.232805\,s}, \frac{2.42107\,s}{1+1.42107\,s}\right\}$$

… as the parametric description of the best fit Lineal Basis. So, we can use **ParametricPlot[]** to display this curve:

In[106]:=
```
linealBasisParametricPlot = ParametricPlot[ {as/(1 - s + as), cs/(1 - s + cs)} /. solSSExy, {s, 0, 1},
   FrameLabel → {Style[ "X", 10, FontFamily → "Times" ], Style[ "Y", 10, FontFamily → "Times" ]} ];
Labeled[ %, Style[ StringForm[ "Figure `1`. Parametric Plot of Best Fit Lineal Basis", 12 ],
  13, FontFamily → "Book Antiqua"] ]
```

Out[107]=

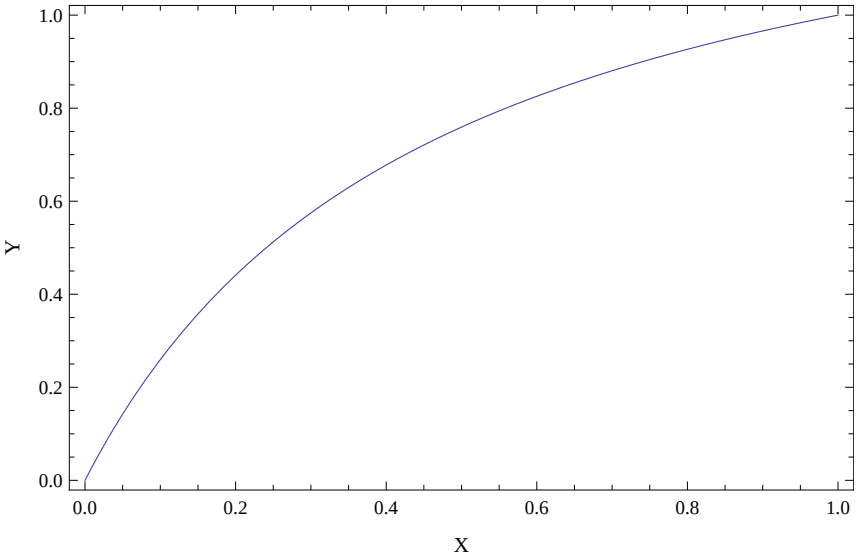


Figure 12. Parametric Plot of Best Fit Lineal Basis

For the purpose of exploring the behaviour of the Beta-Beta distributions as s varies from 0 to 1, we now define four separate functions for αX, βX, αY, and βY, using the values of the linear coefficients in the αX and βX functions **a/.solSSExy** (the fitted value of a) and 1, respectively, and for the values of the linear coefficients in the αY and βY functions **c/.solSSExy** (the fitted value of a) and 1, respectively.

In[108]:=
```
ClearAll[αX];
αX[s_] := a s /. solSSExy // Evaluate
?αX
```

```
Global`αX
```

```
αX[s_] := 0.767195 s
```

In[111]:=
```
ClearAll[βX];
βX[s_] := 1 (1 - s)
?βX
```

```
Global`βX
```

```
βX[s_] := 1 (1 - s)
```

In[114]:=
```
ClearAll[αY];
αY[s_] := c s /. solSSExy // Evaluate
?αY
```

```
Global`αY
```

```
αY[s_] := 2.42107 s
```

In[117]:=
```
ClearAll[βY];
βY[s_] := 1 (1 - s)
?βY
```

```
Global`βY
```

```
βY[s_] := 1 (1 - s)
```

The fitting procedure we used to determine the Lineal Basis parameters a, c, and the s_i found a best fit Lineal Basis using these "normalised" αX[s, … functions] and so did not yield estimates of the absolute values of the original parameters a, b, c, and d. Now that we know the *ratio* of a to b—i.e., the best fit estimate of a in the "normalised" αX function—and the *ratio* of c to d—i.e., the best fit estimate of c in the "normalised" αX function, we need a method of estimating the absolute values of a, b, c, and d while preserving the fitted ratios a/b and c/d.

Here are the absolute discrepancies between the x-coordinates of the data points and the x-coordinates of their corresponding "generating" points, plotted against the s value for the corresponding "generating" point. This is then the absolute departure—parallel to the X-axis— of the data from their "generating" points as we move along the Lineal Basis from s=0 to s=1.

In[120]:=
```
ListPlot[
  Transpose[{sParams, Abs[data[[All, 1]] - Xmodelled]}] /. solSSExy // Sort, Joined → True, FrameLabel →
   {Style["s", 10, FontFamily → "Times"], Style["Absolute X Error", 10, FontFamily → "Times"]}];
Labeled[%, Style[StringForm["Figure `1`. Absolute X Coordinate Differences (data-'generating') vs s",
   13], 13, FontFamily → "Book Antiqua"]]
```

Out[121]=

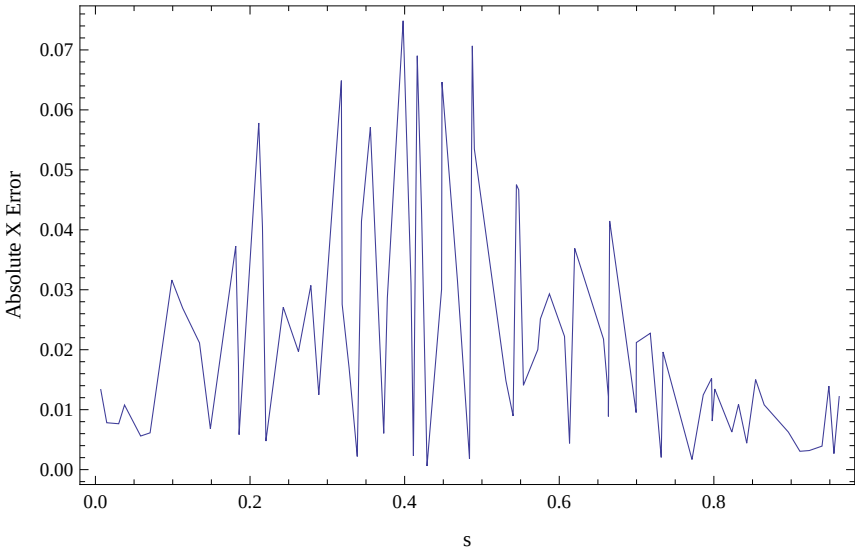


Figure 13. Absolute X Coordinate Differences (data−'generating') vs s

Here next are the steps by which *mathStatica* can determine the expected absolute difference between a random sample from the general Beta[α, β] distribution and the mean of the distribution.
This defines the general Beta[α, β] PDF:

In[122]:=
```
f = x^(α-1) (1 - x)^(β-1) / Beta[α, β];   domain[f] = {x, 0, 1} && {α > 0, β > 0};
```

The mean is confirmed via *mathStatica*'s **Expect[]** function …

In[123]:=
```
Expect[x, f]
```

Out[123]=
$$\frac{\alpha}{\alpha + \beta}$$

… and the general expression for the required expected absolute difference is found via:

In[124]:=
```
expectedBetaAbsError = Expect[Abs[x - α/(α + β)], f]
```

Out[124]=
$$\frac{2\,\alpha\,\text{Beta}\left[\frac{\alpha}{\alpha+\beta}, \alpha, \beta\right] - 2\,(\alpha+\beta)\,\text{Beta}\left[\frac{\alpha}{\alpha+\beta}, 1+\alpha, \beta\right]}{(\alpha+\beta)\,\text{Beta}[\alpha, \beta]}$$

We now propose to substitute into this expression the "normalised" forms for αX[s, …] and βX[s, …] using the best fit values for a and c, but with a "scaling factor" we will call **sf**, i.e., instead of **a/.solSSExy** we will have **sf × a/.solSSExy**, and instead of 1 we will have **sf**. We will then calculate the expected absolute errors at each of the "generating" points using the resulting expression, each of which will be a function of **sf**. Then we can again calculate a total-sum-of-squared-differences between these values and those plotted in Figure 13, i..e, the corresponding observed absolute errors, and find the best fit value of **sf**. We argue that this is a reasonable method for recovering good estimates of a and b, since it is based on the best fit value for a in the "normalised" version of αX[s, ...], which gives the ratio a/b, and makes use of the variation about the Lineal Basis (separately in each of the X and Y dimensions).

To find the expected absolute errors at each of the "generating" points, we will **Map[]** a pure function over the list of best fit s_i values, and we can construct the form of the required pure function in one step:

In[125]:=
```
expectedBetaAbsError /. {α → sf a s, β → sf (1 - s)} /. solSSExy /. {s → #}
```

Out[125]=
$$\left(1.53439\ \text{sf}\ \text{Beta}\left[\frac{0.767195\ \text{sf}\ \#1}{\text{sf}\ (1-\#1)+0.767195\ \text{sf}\ \#1},\ 0.767195\ \text{sf}\ \#1,\ \text{sf}\ (1-\#1)\right]\&1 - 2\ \text{Beta}\left[\frac{0.767195\ \text{sf}\ \#1}{\text{sf}\ (1-\#1)+0.767195\ \text{sf}\ \#1},\ 1+0.767195\ \text{sf}\ \#1,\ \text{sf}\ (1-\#1)\right]\&1\ (\text{sf}\ (1-\#1)+0.767195\ \text{sf}\ \#1)\right) \Big/ \left(\text{Beta}[0.767195\ \text{sf}\ \#1,\ \text{sf}\ (1-\#1)]\ (\text{sf}\ (1-\#1)+0.767195\ \text{sf}\ \#1)\right)$$

These are the expected absolute errors at each of the "generating" points; **sf** is now the only free parameter:

In[126]:=
```
expectedBetaAbsoluteXErrors =
  Map[ (expectedBetaAbsError /. {α → sf a s, β → sf (1 - s)} /. solSSExy /. {s → #}) &, sParams /. solSSExy ];
% //
 Short
```

Out[127]//Short=
$$\left\{\frac{1.00343\ (\text{<<1>>}+\text{<<21>>}\ \text{sf}\ \text{Beta}[\text{<<1>>}])}{\text{sf}\ \text{Beta}[0.0112515\ \text{sf},\ \text{<<19>>}\ \text{sf}]},\ \text{<<76>>},\ \frac{\text{<<17>>}\ \text{<<1>>}}{\text{sf}\ \text{<<1>>}}\right\}$$

These are the observed absolute errors, ...

In[128]:=
```
observedBetaAbsoluteXErrors = Abs[data[[All, 1]] - Xmodelled] /. solSSExy;
% // Short
```

Out[129]//Short=
```
{0.00782104, 0.00765206, <<74>>, 0.0139329, 0.0121802}
```

... this constructs the total sum-of-squared-differences we would like to minimize with respect to **sf**, ...

In[130]:=
```
sseBetaAbsoluteXErrors = Total[ (expectedBetaAbsoluteXErrors - observedBetaAbsoluteXErrors)^2 ];
% // Short
```

Out[131]//Short=
$$\left(-0.0133526+\frac{\text{<<19>>}\ \text{<<1>>}}{\text{sf}\ \text{<<1>>}}\right)^2+\left(-\text{<<21>>}+\frac{\text{<<1>>}}{\text{<<1>>}}\right)^2+(\text{<<1>>}\ \text{<<1>>}^2+\text{<<72>>}+\text{<<1>>}+(\text{<<1>>})^2+(-0.0121802+\text{<<1>>})^2$$

... and this finds the best fit value of **sf**:

In[132]:=
```
{sseBetaX, solSSEBetaX} = FindMinimum[ sseBetaAbsoluteXErrors, {sf, 200, 300} ]
```

Out[132]=
```
{0.0250731, {sf → 251.583}}
```

Now we can re-define the αX[s, …] and βX[s, …] functions using this value of **sf**:

In[133]:=
```
ClearAll[αX];
αX[s_] := sf a s /. solSSExy /. solSSEBetaX // Evaluate
?αX
```

```
Global`αX
```

```
αX[s_] := 193.013 s
```

In[136]:=
```
ClearAll[βX];
βX[s_] := sf (1 - s) /. solSSEBetaX // Evaluate
?βX
```

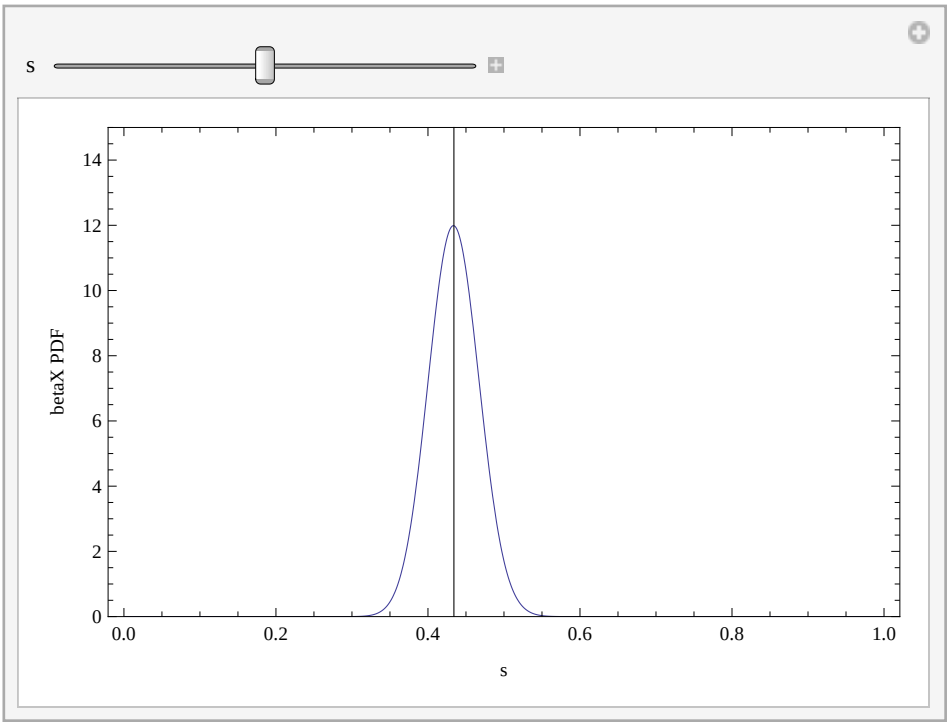
```
Global`βX
```

$\beta X[s\_] := 251.583\ (1 - s)$

This **Manipulate[]** shows how the betaX PDF and its mean varies as s ranges over {0, 1}, i.e., along the fitted Lineal Basis from s=0 to s=1. The mean is indicated by the vertical black line.

In[139]:=
```
Manipulate[
 Plot[PDF[BetaDistribution[αX[s], βX[s]], x],
  {x, 0, 1}, PlotRange → {0, 15}, FrameLabel → {"s", "betaX PDF"},
  Epilog → {Line[{{αX[s]/(αX[s] + βX[s]), 0}, {αX[s]/(αX[s] + βX[s]), 15}}]}], {{s, 0.5}, 0.001, 0.999}]
```

Out[139]=



We now repeat this procedure for $\alpha Y[\ s, \ldots\ ]$ and $\beta Y[\ s, \ldots\ ]$. (We condense the code somewhat to save space.)

In[140]:=
```
expectedBetaAbsoluteYErrors =
  Map[(expectedBetaAbsError /. {α → sf c s, β → sf (1 - s)} /. solSSExy /. {s → #}) &, sParams /. solSSExy];
observedBetaAbsoluteYErrors = Abs[data[[All, 1]] - Xmodelled] /. solSSExy;
sseBetaAbsoluteXErrors = Total[(expectedBetaAbsoluteYErrors - observedBetaAbsoluteYErrors)^2];
{sseBetaY, solSSEBetaY} = FindMinimum[sseBetaAbsoluteXErrors, {sf, 200, 300}];
```

In[144]:=
```
ClearAll[αY];
αY[s_] := sf c s /. solSSExy /. solSSEBetaY // Evaluate
?αY
```

```
Global`αY
```

$\alpha Y[s\_] := 288.376\ s$

In[147]:=
```
ClearAll[βY];
βY[s_] := sf (1 - s) /. solSSEBetaY // Evaluate
?βY
```

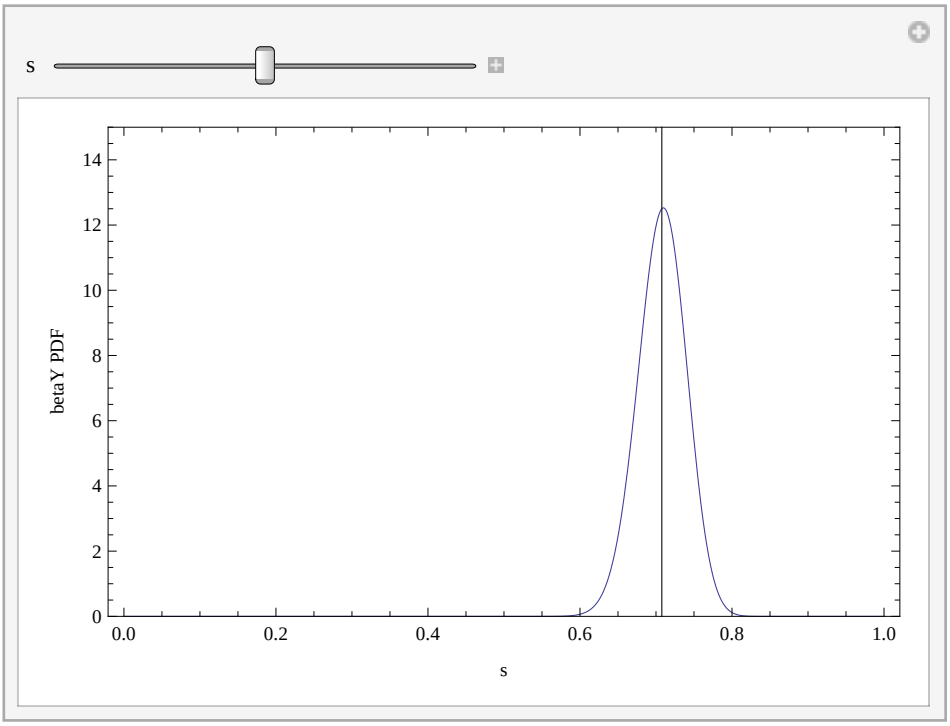
```
Global`βY
```

```
βY[s_] := 119.111 (1 - s)
```

This **Manipulate[]** shows in turn how the betaY PDF and its mean varies as s ranges over {0, 1}, i.e., along the fitted Lineal Basis from s=0 to s=1.
The mean is indicated by the vertical black line.

In[150]:=
```
Manipulate[
 Plot[PDF[BetaDistribution[αY[s], βY[s]], x],
  {x, 0, 1}, PlotRange → {0, 15}, FrameLabel → {"s", "betaY PDF"},
  Epilog → {Line[{{αY[s]/(αY[s] + βY[s]), 0}, {αY[s]/(αY[s] + βY[s]), 15}}]}], {{s, 0.5}, 0.001, 0.999}]
```

Out[150]=



This shows the two linear functions αX[s] and βX[s] …

In[151]:=
```
Plot[{αX[s], βX[s]}, {s, 0, 1},
  FrameLabel → {Style[ "s", 10, FontFamily → "Times" ], Style[ "αX and  βX", 10, FontFamily → "Times" ]} ];
Labeled[ %,  Style[ StringForm[ "Figure `1`. αX (increasing) and βX (decreasing) as functions of s", 14 ],
  13, FontFamily → "Book Antiqua"] ]
```

Out[152]=

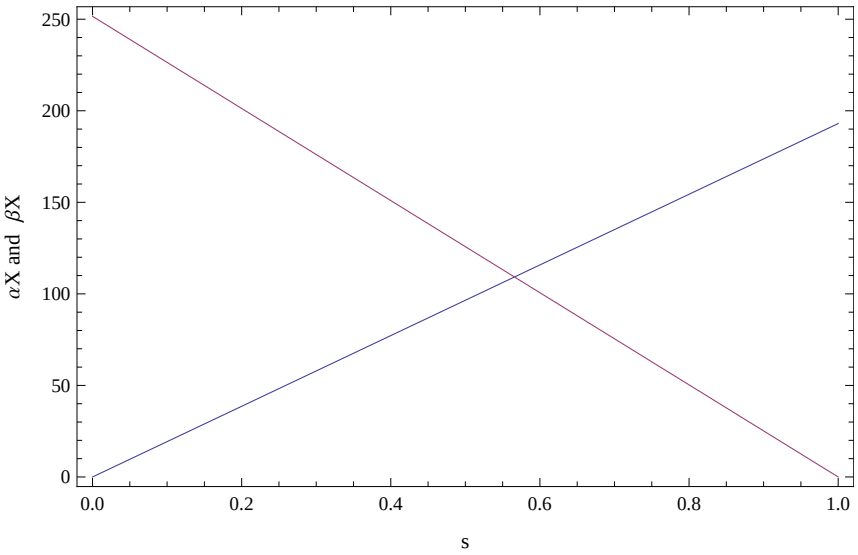


Figure 14. αX (increasing) and βX (decreasing) as functions of s

… and these are the two linear functions αX[s] and βX[s].

In[153]:=
```
Plot[{αY[s], βY[s]}, {s, 0, 1},
  FrameLabel → {Style[ "s", 10, FontFamily → "Times" ], Style[ "αX and  βX", 10, FontFamily → "Times" ]} ];
Labeled[ %,  Style[ StringForm[ "Figure `1`. αY (increasing) and βY (decreasing) as functions of s", 15 ],
  13, FontFamily → "Book Antiqua"] ]
```

Out[154]=

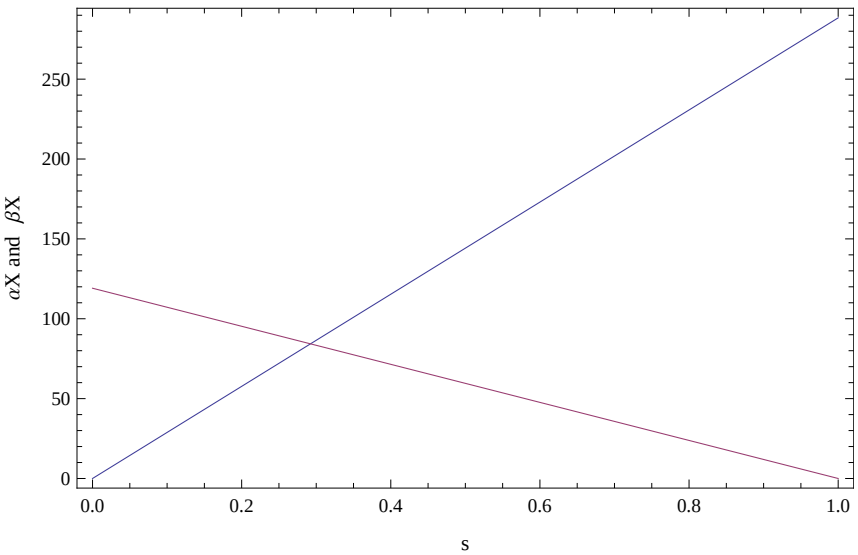


Figure 15. αY (increasing) and βY (decreasing) as functions of s

Finally in this section we show how the joint 95 % - containing region for the bivariate Beta-Beta distribution varies as X varies from 0 to 1.

A rectangular—not the smallest area—region that contains 95% of the joint Beta×Beta distribution is one bounded by the limits of a product of central confidence intervals for each of BetaX and BetaY that each contain …

In[155]:=
```
√0.95
```

Out[155]= 0.974679

… of each distribution. The two corresponding CDF values are thus:

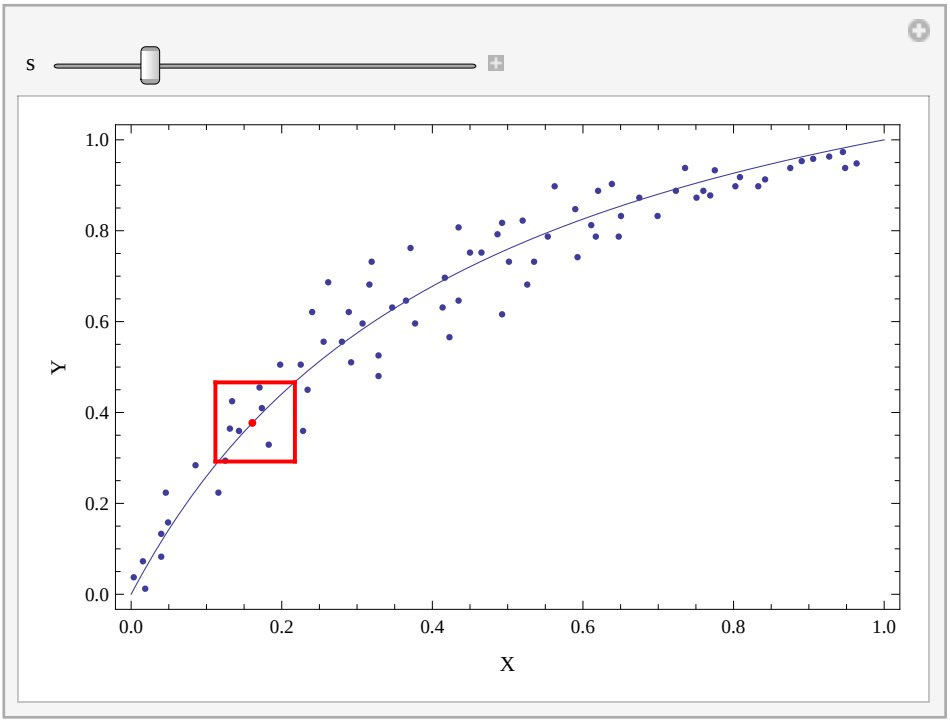
In[156]:=
$$\{cdfLo,\ cdfHi\} = \left\{\frac{1-\sqrt{0.95}}{2},\ 1-\frac{1-\sqrt{0.95}}{2}\right\}$$

Out[156]= {0.0126603, 0.98734}

A **Manipulate[]** allows us to see clearly that as s ranges from 0 to 1, the four linear functions αX[s,a], βX[s,b], αY[s,c], and βY[s,d] vary in such a way that the joint mean (indicated by the red dot) of the Beta-Beta distribution tracks along the fitted Lineal Basis, and the rectangular region with 95% coverage for any given value of s expands and contracts appropriately. As will be seen below, the calculation of a "mean prediction" region for the complete model must be based on samples from the complete model—with s ranging from 0 to 1—considered *in toto*.

In[157]:=
```
Manipulate[
 x95Lo = Quantile[BetaDistribution[αX[s], βX[s]], cdfLo];
 x95Hi = Quantile[BetaDistribution[αX[s], βX[s]], cdfHi];
 y95Lo = Quantile[BetaDistribution[αY[s], βY[s]], cdfLo];
 y95Hi = Quantile[BetaDistribution[αY[s], βY[s]], cdfHi];
 Show[ dataPlot, linealBasisParametricPlot,
  Epilog → {Red, PointSize[ 0.01], Point[ {αX[s]/(αX[s] + βX[s]), αY[s]/(αY[s] + βY[s])} ],
    Thickness[ 0.005 ],
    Line[ {{x95Lo, y95Lo}, {x95Hi, y95Lo}} ], Line[ {{x95Hi, y95Lo}, {x95Hi, y95Hi}} ],
    Line[ {{x95Hi, y95Hi}, {x95Lo, y95Hi}} ], Line[ {{x95Lo, y95Hi}, {x95Lo, y95Lo}} ]
   }]
, {{s, 0.2}, 0.001, 0.999, 0.001}]
```

Out[157]=



Section 4 - Monte Carlo Sampling for the Single Prediction Region.

Single Prediction Region Monte Carlo Sampling.

Next we generate a large sample of simulated data from the statistical model we have developed.
Setting the sample size …

In[158]:=
```
sampleSize = 1000;
```

… we can plot a set of **sampleSize** (x,y) pairs so that consecutive samples are from the slightly different BetaX×BetaY joint distributions that obtain from **sampleSize** s values ranging from $\frac{1}{\text{sampleSize}+1}$ to $\frac{\text{sampleSize}}{\text{sampleSize}+1}$ in steps of $\frac{1}{\text{sampleSize}+1}$.

In[159]:=
```
samplePlot = ListPlot[ mcSampleXYtotal = predictiveSampleData =
    Table[ {RandomReal[BetaDistribution[αX[s], βX[s]] ], RandomReal[BetaDistribution[αY[s], βY[s]]]},
     {s, 1 / (sampleSize + 1), 1 - 1 / (sampleSize + 1), 1 / (sampleSize + 1)}],
   PlotRange → All,  PlotStyle → {PointSize[ 0.0001 ]} ];
Labeled[ %, Style[ StringForm[ "Figure `1`. `2` samples from the complete Beta∗Beta statistical model",
   16, NumberForm[ sampleSize, 3 ] ], FontSize → 14, FontFamily → "Sylfaen" ] ]
```

Out[160]=

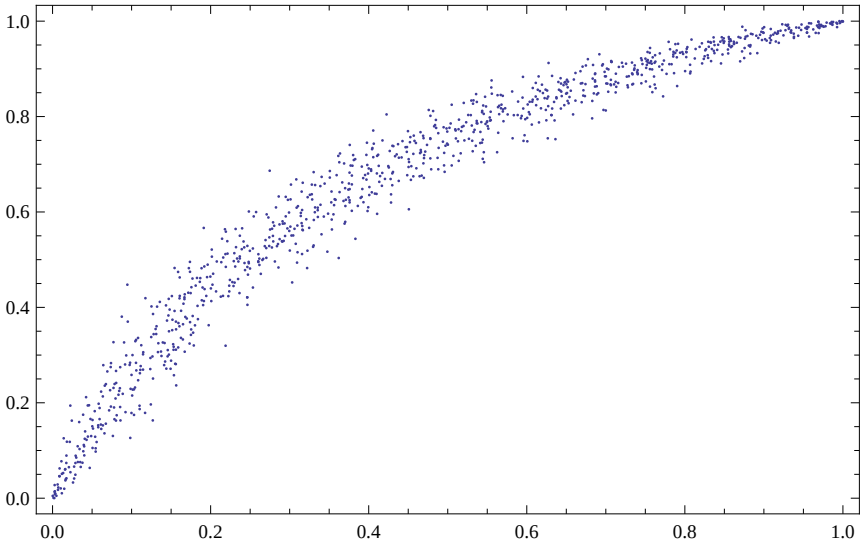
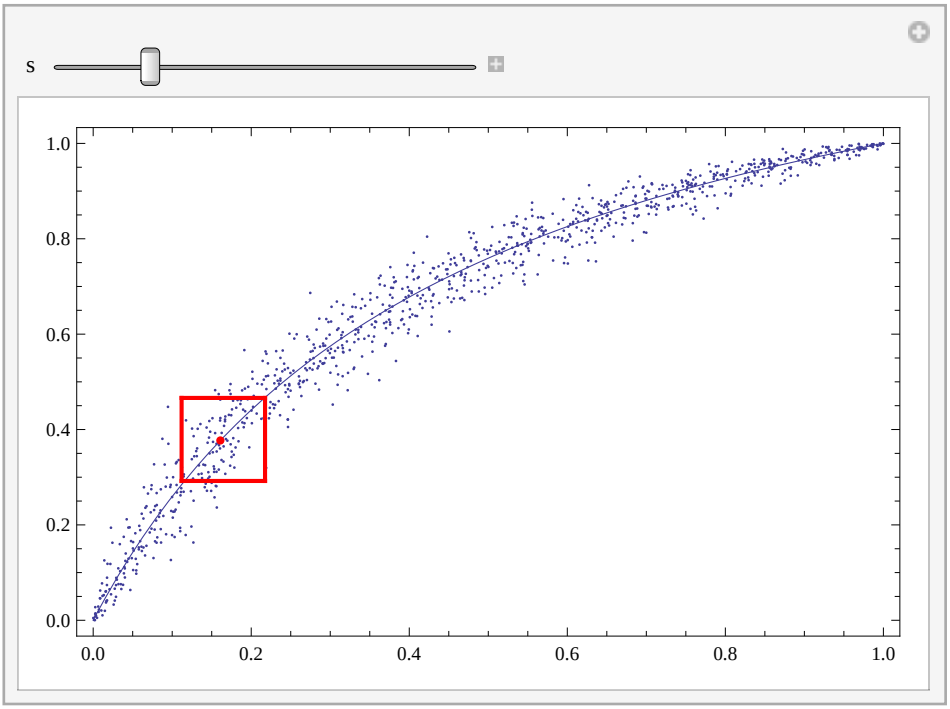


Figure 16. 1000 samples from the complete Beta∗Beta statistical model

This **Manipulate[]** is the same as the preceding one except with the large sample of simulated data shown in Figure 16.
We note that there is good visual agreement between the spread of the simulated data and the size of the 95% coverage rectangle for each s value.

In[161]:=
```
Manipulate[
 x95Lo = Quantile[BetaDistribution[αX[s], βX[s]], cdfLo];
 x95Hi = Quantile[BetaDistribution[αX[s], βX[s]], cdfHi];
 y95Lo = Quantile[BetaDistribution[αY[s], βY[s]], cdfLo];
 y95Hi = Quantile[BetaDistribution[αY[s], βY[s]], cdfHi];
 Show[ samplePlot, linealBasisParametricPlot,
  Epilog → {Red, PointSize[ 0.01], Point[ { αX[s]/(αX[s] + βX[s]), αY[s]/(αY[s] + βY[s]) } ],
    Thickness[ 0.005 ],
    Line[ {{x95Lo, y95Lo}, {x95Hi, y95Lo}} ], Line[ {{x95Hi, y95Lo}, {x95Hi, y95Hi}} ],
    Line[ {{x95Hi, y95Hi}, {x95Lo, y95Hi}} ], Line[ {{x95Lo, y95Hi}, {x95Lo, y95Lo}} ]
   }]
 , {{s, 0.2}, 0.001, 0.999, 0.001}]
```

Out[161]=



The Concave-Hull-Stripping Method for Defining a Single Prediction Region.

The technique we apply in order to determine a defined single prediction region for our Lineal Basis model is a modification of a numerical approach we will refer to as concave-hull-stripping. A set of single predictions is generated via random sampling from the "observation" Beta-Beta distribution using the fitted model parameters.

Although for most nonlinear models an analytic expression for a 95%-containing region is not generally available, a 95% region can often be approximated by determining the convex hull of a large single prediction region sample and then successively stripping away just the points on the convex hull. This is repeated until approximately 95% of the original point set remains. The convex hull of the remaining point set is then taken as a good approximation to the desired 95% single prediction region. Since the number of points in each successive convex hull is not known in advance, the precision with which the final point set can be determined to contain 95% of the original points is governed by the size of the original point set.

Note that *Mathematica* offers the function **PolytopeQuantiles[*points*,*q*]** within the **MultivariateStatistics`** package. From the *Mathematica* documentation: "Outlying vectors are removed by repeatedly peeling off layers of convex hulls from the data until at most a fraction q are left. **PolytopeQuantile[*points*,*q*]** interpolates between the convex hulls of the points remaining before and after the last layer is removed."

Our problem is that the standard theory is not available for Lineal Basis modelling, and as shown our single prediction point set above is not convex. We therefor employ the defined function **functionAlpha[*α*,*pointSet*,*dtEdge*]**, which uses concepts from computational geometry, to determine analogues of convex hulls, called alpha hulls. Applied to an edge in the Delaunay Triangulation of the point set, and for a given value of α, **functionAlpha[]** determines whether the edge does or does not belong to the alpha hull for that α.

Alpha hulls are typically concave. Here is a test point set with alpha hulls for the indicated values of alpha:

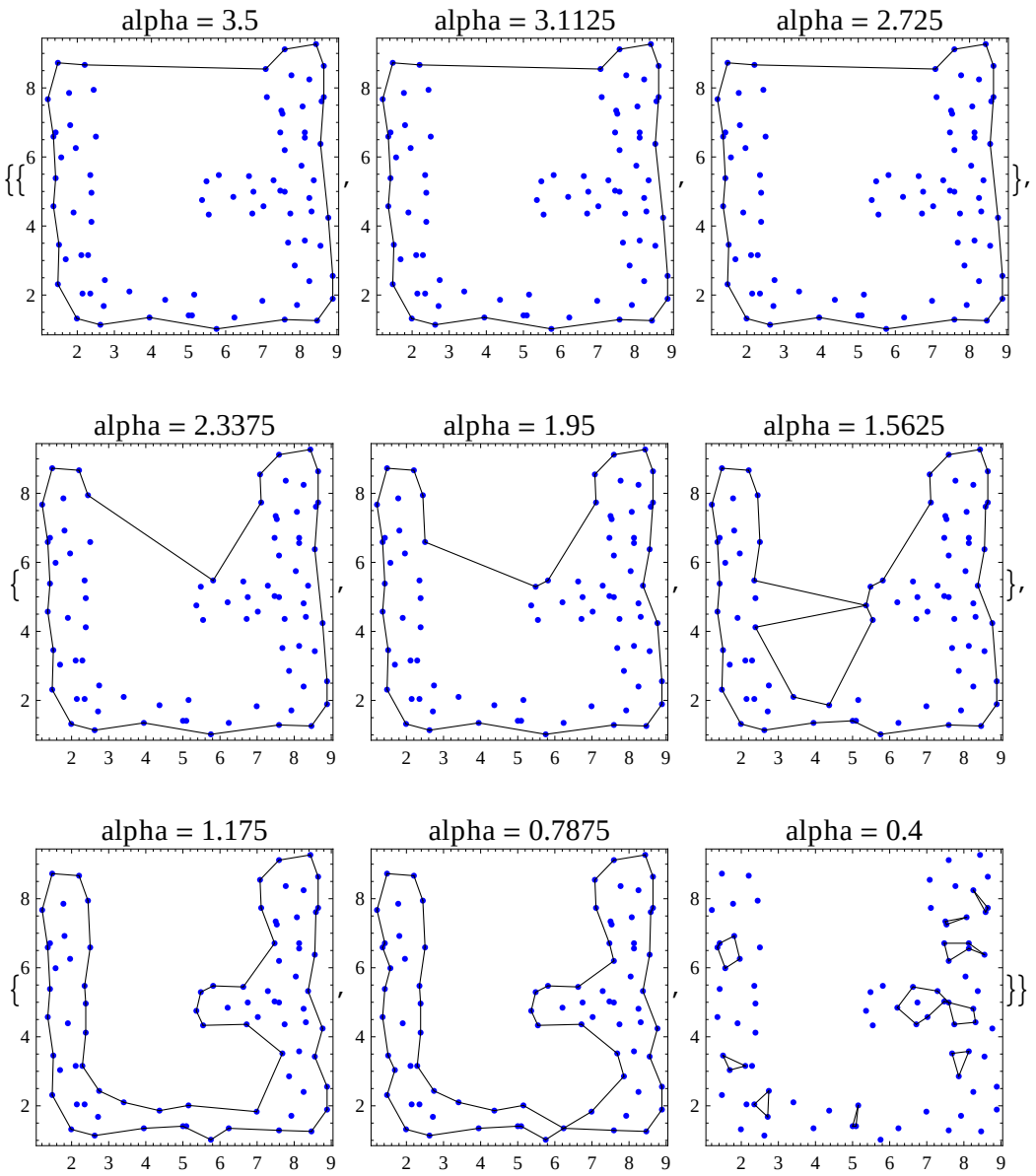


Figure 17. A Range of Alpha Hulls

It is clear that for an appropriate value of alpha, the alpha hull coincides closely with the outline of even highly concave point sets as perceived by most observers. (These ideas are widely used in character recognition.) Note that when alpha is sufficiently large the alpha hull is the standard convex hull (as explained further in Appendix 1), and when sufficiently small the alpha hull fragments, and consists of many small disjoint polygons.

For the purposes of obtaining a sequence of alpha hulls that can be successively stripped to arrive at a reasonable approximation to the 95% single prediction region, we have found empirically that a suitable value of alpha for the point sets with which we are concerned is around half of the diameter of the current point set. (The function **pointSetDiameterFunction[]** defined above finds the largest point-to-point distance for all possible pairs of points.)

We will work with the single prediction region sample assigned above to **predictiveSampleData** to illustrate concave-hull-stripping.

We first generate a new plot of the single prediction region sample:

```
In[162]:= predictiveSampleDataPlot = ListPlot[predictiveSampleData,
            PlotRange -> Full, AspectRatio -> 1, PlotStyle → {Blue, PointSize[ 0.01 ]},
            FrameLabel → {Style[ "X", 10, FontFamily → "Times" ], Style[ "Y", 10, FontFamily → "Times" ]}];
```

We will need the point set diameter:

```
In[163]:= pointSetDiameter = pointSetDiameterFunction[predictiveSampleData]

Out[163]= 1.41237
```

We next use the **DelaunayTriangulation[]** function from *Mathematica*'s **ComputationalGeometry`** package to arrive at a list of paired pointers to the endpoints of all the edges in the Delaunay triangulation:

```
In[164]:= tri = DelaunayTriangulation[ predictiveSampleData ];
          % // Short
          edgePointers = Thread[List @@ #] & /@ tri // Flatten[#, 1] & ;
          edgePointers = Union[Sort /@ edgePointers];
          % // Short

Out[165]//Short=
          {{1, {994, 5, 7, 6}}, {2, {8, 15, 10, 5, 12, 18}}, <<996>>,
           {999, {997, 1000, 993, 992, 994}}, {1000, {996, 993, 999, 997}}}

Out[168]//Short=
          {{1, 5}, {1, 6}, {1, 7}, {1, 994}, {2, 5}, {2, 8}, <<2967>>,
           {994, 999}, {995, 998}, {996, 1000}, {997, 999}, {997, 1000}, {999, 1000}}
```

With alpha set to half the point set diameter, the variable **requiredDTedges** lists only the edges in the corresponding alpha hull.
The graphic shows that this choice of alpha is a good compromise between larger values which miss the concavity of the "underside" of the point set, and the small values that produce the fragmentation shown above for alpha=0.4 applied to the test point set.

```
In[169]:= requiredDTedges = Extract[edgePointers,
             Position[functionAlpha[alf = pointSetDiameter / 2.01, predictiveSampleData, #] & /@ edgePointers, 1]];
          Show[ predictiveSampleDataPlot, Graphics[GraphicsComplex[
              predictiveSampleData, Line[requiredDTedges]], PlotRange → All ] ];
          Labeled[ %, Style[ StringForm[ "Figure `1`. Single Prediction Sample with Alpha Hull; alpha = `2`",
             18, NumberForm[ alf, 3 ] ], FontSize → 14, FontFamily → "Sylfaen" ] ]
```

Out[171]=

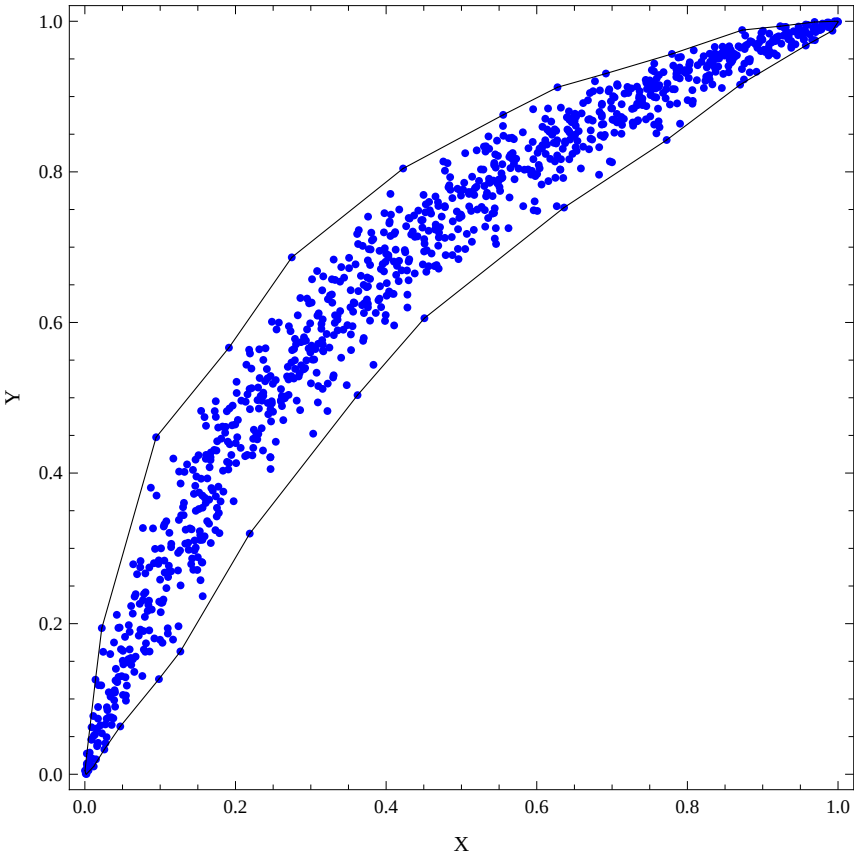


Figure 18. Single Prediction Sample with Alpha Hull; alpha = 0.703

In order to begin "concave-hull-stripping" this data set, the following steps are taken.
We can easily determine and count the points on this alpha hull:

```
In[172]:= outermostPoints = Flatten[
            Map[ ({predictiveSampleData[[#[[1]]], predictiveSampleData[[#[[2]]]]}) &,  requiredDTedges ], 1 ] // Union;
```

We keep a progressive collection of all the points discarded as we repeatedly remove each new alpha hull (and construct a plot of these discarded points):

```
In[173]:= outerMostPointsForPlotting = outermostPoints;
          outermostPointsPlot = ListPlot[outerMostPointsForPlotting,
             PlotRange -> Full, AspectRatio -> 1, PlotStyle → {Orange, PointSize[ 0.01 ]}];
```

Now we repeat the above calculations, printing out the percentage coverage of the new alpha hull.
The discarded points are shown in orange, to place each new concave hull with decreasing coverage in the context of the original single prediction sample.

This defines the new point set with the current alpha hull points removed, and creates a new plot:

```
In[175]:= newData = Complement[ predictiveSampleData, outermostPoints ];
```

```
In[176]:= newDataPlot =
            ListPlot[newData, PlotRange -> Full, AspectRatio -> 1, PlotStyle → {Blue, PointSize[ 0.01 ]},
             FrameLabel → {Style[ "X", 10, FontFamily → "Times" ], Style[ "Y", 10, FontFamily → "Times" ]}];
```

We will need the new point set diameter:

```
In[177]:= newPointSetDiameter = pointSetDiameterFunction[newData ]

Out[177]= 1.40318
```

We again use the **`DelaunayTriangulation[]`** function to obtain a new list of pointers to the pairs of endpoints of all the edges in the Delaunay triangulation, and with the new diameter we calculate and display the new alpha hull:

In[178]:=
```
newTri = DelaunayTriangulation[ newData ];
newEdgePointers = Thread[List @@ #] & /@ newTri // Flatten[#, 1] & ;
newEdgePointers = Union[Sort /@ newEdgePointers];
newRequiredDTedges = Extract[newEdgePointers,
   Position[functionAlpha[alf = newPointSetDiameter / 2.01, newData, #] & /@ newEdgePointers, 1]];
Show[ newDataPlot, outermostPointsPlot, Graphics[
   GraphicsComplex[ newData, Line[newRequiredDTedges]], PlotRange → All ] ];
Labeled[ %, Style[ StringForm[ "Figure `1`. Single Prediction Sample with Alpha Hull; alpha = `2`",
   19, NumberForm[ alf, 3 ] ], FontSize → 14, FontFamily → "Sylfaen" ] ]
Print[ "Coverage = ", Length[ newData] / Length[ predictiveSampleData ] 100.0, "%" ]
```

Out[183]=

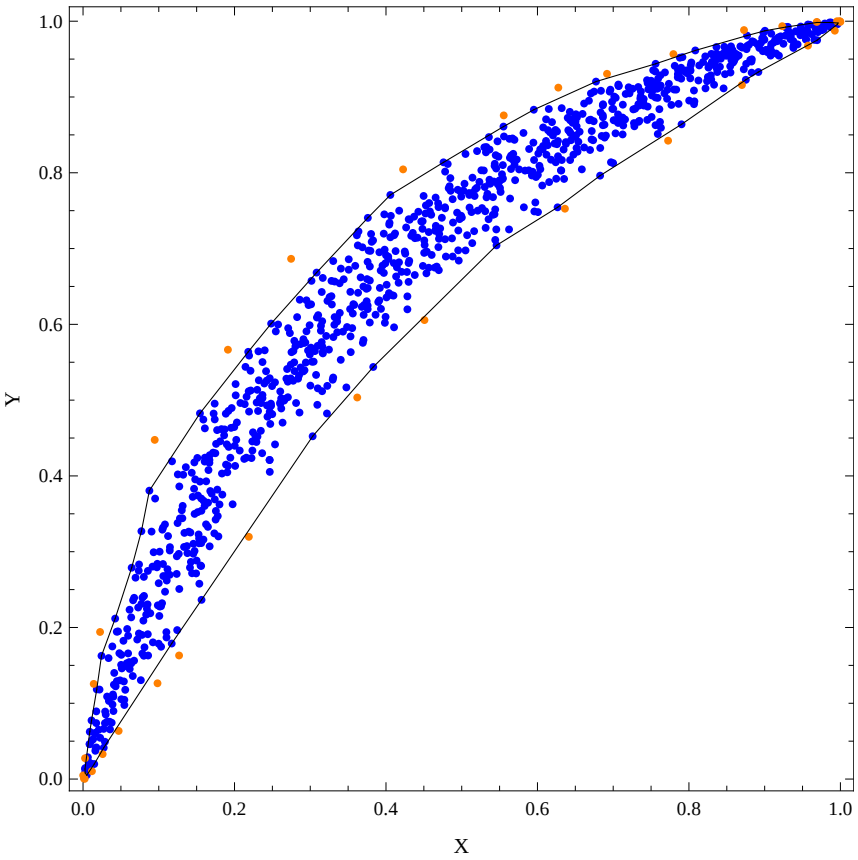


Figure 19. Single Prediction Sample with Alpha Hull; alpha = 0.698

```
Coverage = 96.6%
```

At this point we can simply re-run the above (condensed) code as many time as we wish, to produce a sequence of concave single prediction regions with decreasing coverage:

In[185]:=
```
outermostPoints = Flatten[ Map[ ({newData[[#[[1]]]], newData[[#[[2]]]]}) &,  newRequiredDTedges ], 1 ] // Union;
outerMostPointsForPlotting = outerMostPointsForPlotting ~ Join ~ outermostPoints;
outermostPointsPlot =
  ListPlot[outerMostPointsForPlotting, PlotRange -> Full, AspectRatio -> 1, PlotStyle → {Orange, PointSize[ 0.01 ]}];
newData = Complement[ newData, outermostPoints ];
newDataPlot = ListPlot[newData, PlotRange -> Full, AspectRatio -> 1, PlotStyle → {Blue, PointSize[ 0.01 ]},
    FrameLabel → {Style[ "X", 10, FontFamily → "Times" ], Style[ "Y", 10, FontFamily → "Times" ]}];
newPointSetDiameter = pointSetDiameterFunction[newData ];
newTri = DelaunayTriangulation[ newData ];
newEdgePointers = Thread[List @@ # ] & /@ newTri // Flatten[#, 1] & ;
newEdgePointers = Union[Sort /@ newEdgePointers];
newRequiredDTedges =
  Extract[newEdgePointers, Position[functionAlpha[alf = newPointSetDiameter / 2.01, newData, # ] & /@ newEdgePointers, 1]];
Show[ newDataPlot, outermostPointsPlot, Graphics[GraphicsComplex[ newData, Line[newRequiredDTedges]], PlotRange → All ] ];
Labeled[ %, Style[ StringForm[ "Figure `1`. Single Prediction Sample with Alpha Hull; alpha = `2`", 20, NumberForm[ alf, 3 ] ],
  FontSize → 14, FontFamily → "Sylfaen" ] ]
Print[ "Coverage = ", Length[ newData] / Length[ predictiveSampleData ] 100.0, "%" ]
```

Out[196]=

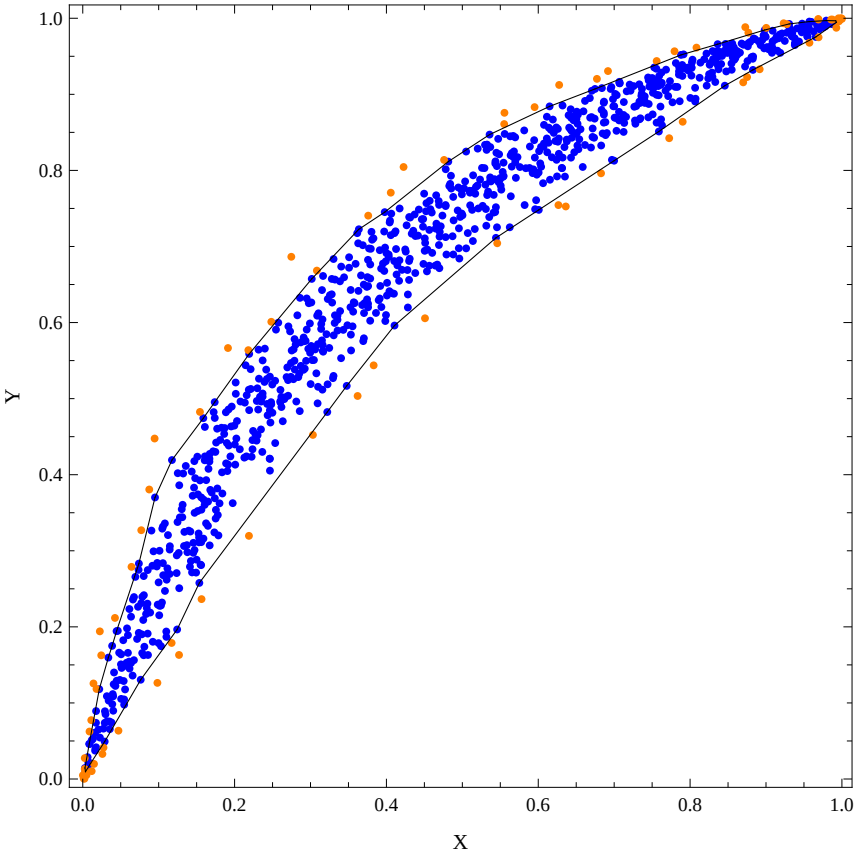


Figure 20. Single Prediction Sample with Alpha Hull; alpha = 0.695

```
Coverage = 92.2%
```

In[198]:=
```
outermostPoints = Flatten[ Map[ ({newData[[#[[1]]], newData[[#[[2]]]]}) &,  newRequiredDTedges ], 1 ] // Union;
outerMostPointsForPlotting = outerMostPointsForPlotting ~ Join ~ outermostPoints;
outermostPointsPlot =
  ListPlot[outerMostPointsForPlotting, PlotRange -> Full, AspectRatio -> 1, PlotStyle → {Orange, PointSize[ 0.01 ]}];
newData = Complement[ newData, outermostPoints ];
newDataPlot = ListPlot[newData, PlotRange -> Full, AspectRatio -> 1, PlotStyle → {Blue, PointSize[ 0.01 ]},
   FrameLabel → {Style[ "X", 10, FontFamily → "Times" ], Style[ "Y", 10, FontFamily → "Times" ]}];
newPointSetDiameter = pointSetDiameterFunction[newData ];
newTri = DelaunayTriangulation[ newData ];
newEdgePointers = Thread[List @@ #] & /@ newTri // Flatten[#, 1] & ;
newEdgePointers = Union[Sort /@ newEdgePointers];
newRequiredDTedges =
  Extract[newEdgePointers, Position[functionAlpha[alf = newPointSetDiameter / 2.01, newData, #] & /@ newEdgePointers, 1]];
Show[ newDataPlot, outermostPointsPlot, Graphics[GraphicsComplex[ newData, Line[newRequiredDTedges]], PlotRange → All ] ];
Labeled[ %, Style[ StringForm[ "Figure `1`. Single Prediction Sample with Alpha Hull; alpha = `2`", 21, NumberForm[ alf, 3 ] ],
  FontSize → 14, FontFamily → "Sylfaen" ] ]
Print[ "Coverage = ", Length[ newData] / Length[ predictiveSampleData ] 100.0, "%" ]
```

Out[209]=

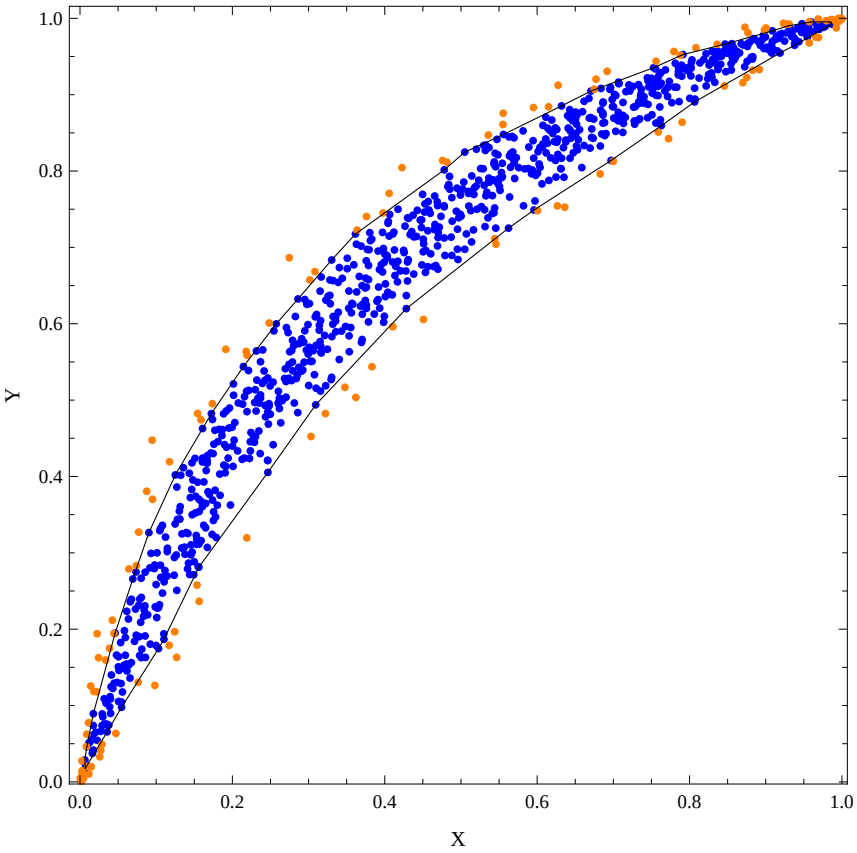


Figure 21. Single Prediction Sample with Alpha Hull; alpha = 0.688

```
Coverage = 87.9%
```

In[211]:=
```
outermostPoints = Flatten[ Map[ ({newData[[#[[1]]]], newData[[#[[2]]]]}) &,  newRequiredDTedges ], 1 ] // Union;
outerMostPointsForPlotting = outerMostPointsForPlotting ~ Join ~ outermostPoints;
outermostPointsPlot =
  ListPlot[outerMostPointsForPlotting, PlotRange -> Full, AspectRatio -> 1, PlotStyle → {Orange, PointSize[ 0.01 ]}];
newData = Complement[ newData, outermostPoints ];
newDataPlot = ListPlot[newData, PlotRange -> Full, AspectRatio -> 1, PlotStyle → {Blue, PointSize[ 0.01 ]},
    FrameLabel → {Style[ "X", 10, FontFamily → "Times" ], Style[ "Y", 10, FontFamily → "Times" ]}];
newPointSetDiameter = pointSetDiameterFunction[newData ];
newTri = DelaunayTriangulation[ newData ];
newEdgePointers = Thread[List @@ # ] & /@ newTri // Flatten[#, 1] & ;
newEdgePointers = Union[Sort /@ newEdgePointers];
newRequiredDTedges =
  Extract[newEdgePointers, Position[functionAlpha[alf = newPointSetDiameter / 2.01, newData, # ] & /@ newEdgePointers, 1]];
Show[ newDataPlot, outermostPointsPlot, Graphics[GraphicsComplex[ newData, Line[newRequiredDTedges]], PlotRange → All ] ];
Labeled[ %, Style[ StringForm[ "Figure `1`. Single Prediction Sample with Alpha Hull; alpha = `2`", 22, NumberForm[ alf, 3 ] ],
  FontSize → 14, FontFamily → "Sylfaen" ] ]
Print[ "Coverage = ", Length[ newData] / Length[ predictiveSampleData ] 100.0, "%" ]
```

Out[222]=

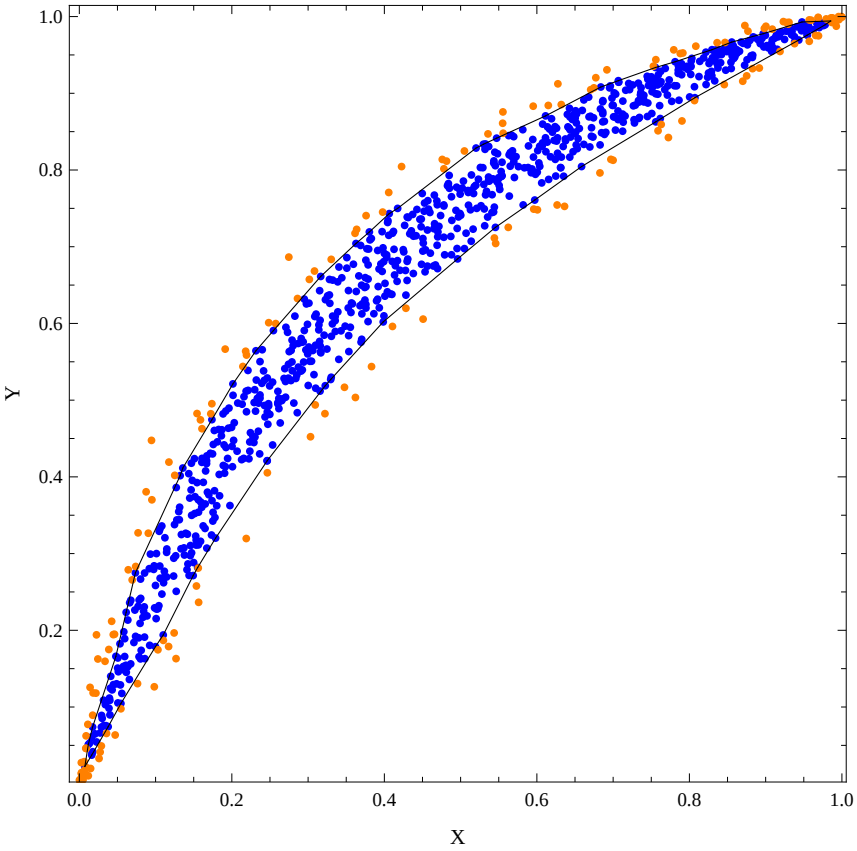


Figure 22. Single Prediction Sample with Alpha Hull; alpha = 0.686

```
Coverage = 83.5%
```

Conclusion.

We have been able to fit a Lineal Basis model to a development data set, and using the combined symbolic, numeric, and graphical features of *Mathematica,* confidence regions for the fitted parameters, and mean and single prediction regions have been constructed. The analytic insight available through the ***mathStatica`*** package [Rose & Smith, 2002] was essential in this process, as also was the intuitive insight provided by the **Manipulate[]** function.

References.

[Moffiet, 2008]. Moffiet, T., 2008. Bivariate relationship modelling on bounded spaces with application to the estimation of forest foliage cover by Landsat satellite ETM-plus sensor. PhD Thesis. University of Newcastle, Callaghan, NSW, Australia. http://hdl.handle.net/1959.13/29220.

[WinBUGS, 2010]. http://www.mrc-bsu.cam.ac.uk/bugs/

[Gregory, 2005]. Gregory, P.C. 2005. Bayesian Logical Data Analysis for the Physical Sciences. Cambridge University Press.

Appendix 1.

These two **Manipulate[]**s illustrate dynamically how the logic of **functionAlpha[]** works. For a given alpha value, **functionAlpha[α,*points*,{e_1,e_2}]** decides whether the edge {e_1,e_2} in the Delaunay Triangulation of the given point set is part of the alpha hull. The test is as follows. Two circles of radius α are constructed, with centres on the orthogonal bisector of the edge with endpoints {e_1,e_2}, and passing through $e_1$and e_2. If it is then the case that *one* of the circles contains other points in the point set but the other does not, then the edge is part of the alpha hull. If both circles are empty, or both contain other points, the edge is not part of the alpha hull. A hint that this logic might produce some sort of generalisation of the convex hull is that, if α is set at a very large value compared with the scale (e.g, the diameter) of the point set, locally the circles associated with each edge will be almost coincident with the edge, and in the limit as α approaches ∞ will become infinite extensions of the edge. In that case, the test amounts to determining whether there are other points of the points set on both sides of the edge or only on one side, and it is obvious that only those edges in the convex hull will pass the "only-on-one-side" test.

If alpha is less than the half-length of an edge, the centres of its two circles coincide at the edge midpoint, so the two circles also coincide. The two circles separate into an "inner" and an "outer" circles just as alpha equals the edge half-length.

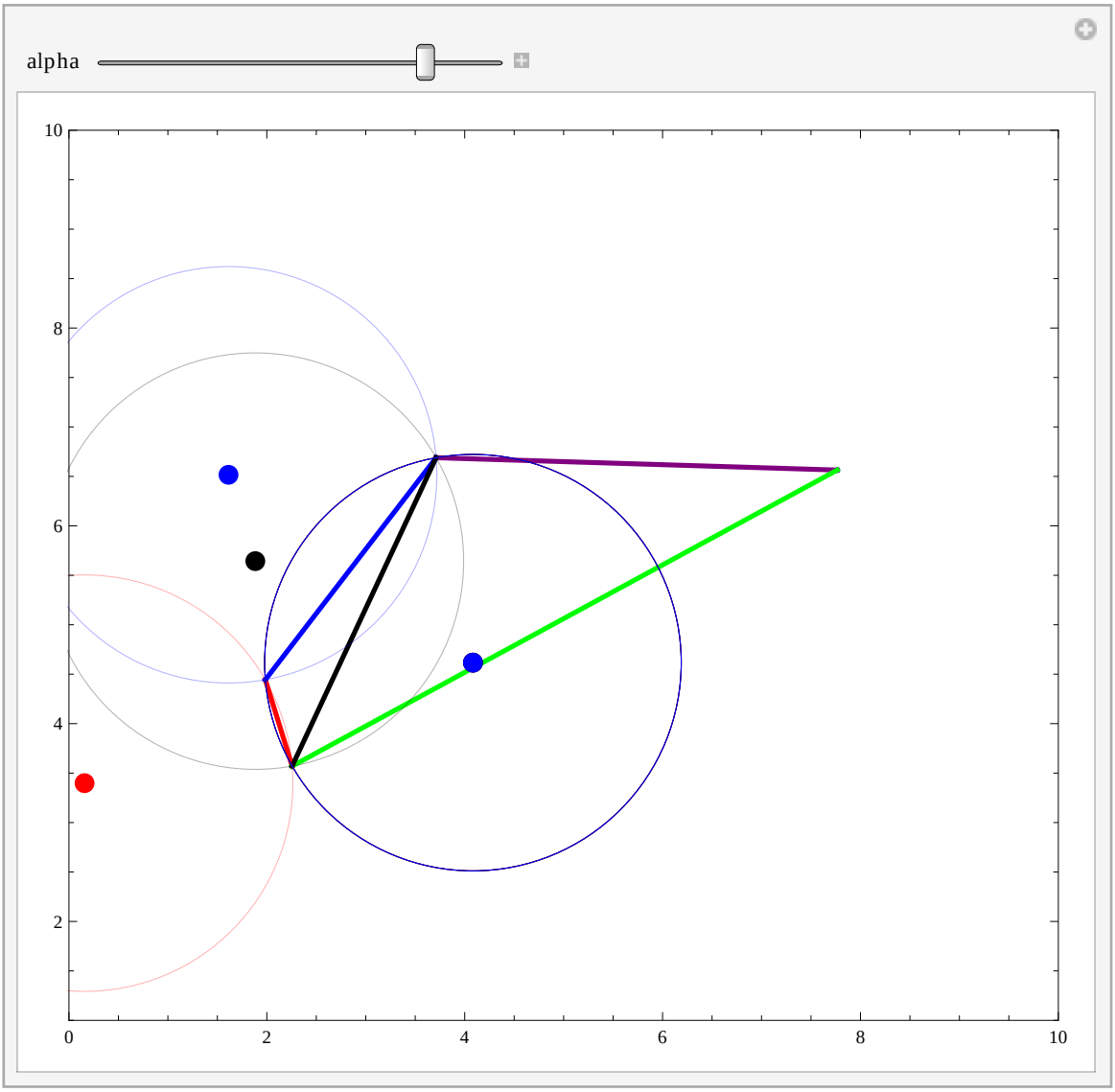
In this first animation, we only show the circles—and their centres—associated with the three sides of the red-blue-black triangle. We have set the starting value of the manipulable variable, alpha, at the value of the radius of the circumcircle of the red-blue-black triangle, so at first we see only one circle, since at that value of alpha the three "inner" circles coincide; they are all circumcircles. The three "outer" circles are shown with reduced opacity.

If alpha is taken to zero all the circles disappear, but as alpha is increased slowly it is apparent that at first all the circles are empty. It is only when alpha reaches the value of the circumcircle radius that each of the three "inner" circles pass through the three vertices of the triangle, and the centres of the "inner" circles coincide. For slightly larger alpha values all three edges belong to the current alpha hull, as each "inner" circle now contains a vertex other than its edge endpoints, while the each paler "outer" circle does not.

```
In[224]:= pointsPlot =
            ListPlot[ points = {p1, p2, p3, p4} = {{2.256, 3.571}, {1.985, 4.444}, {3.711, 6.689}, {7.765, 6.564}},
              Axes → False, Frame → True, AspectRatio → Automatic ];

In[225]:= ccr1 = circumCircleRadius[ {p1, p2, p3} ]

Out[225]= 2.10538

In[226]:= ccr2 = circumCircleRadius[ {p1, p3, p4} ]

Out[226]= 3.41184
```

In[227]:=
```
quadrilateral = Graphics[ Flatten[Transpose[ {{{Thickness[0.005], Red}, Blue, Purple, Green},
      Map[ Line, Partition[ points ~Join~ {First[ points ]}, 2, 1 ] ]} ], 2] ];
edges = Partition[ points ~Join~ {First[ points ]}, 2, 1 ];
Manipulate[
 (sol12 = Solve[ {Total[ ({x, y} - points[[1]])^2 ] == alpha^2, Total[ ({x, y} - points[[2]])^2 ] == alpha^2} ];
  centres12 =
   If[ FreeQ[ {x, y} /. sol12, Complex] , {x, y} /. sol12, {(points[[1]] + points[[2]])/2, (points[[1]] + points[[2]])/2}];
  sol13 = Solve[ {Total[ ({x, y} - points[[1]])^2 ] == alpha^2, Total[ ({x, y} - points[[3]])^2 ] == alpha^2} ];
centres13 =
   If[ FreeQ[ {x, y} /. sol13, Complex] , {x, y} /. sol13, {(points[[1]] + points[[3]])/2, (points[[1]] + points[[3]])/2}];
  sol23 = Solve[ {Total[ ({x, y} - points[[2]])^2 ] == alpha^2, Total[ ({x, y} - points[[3]])^2 ] == alpha^2} ];
centres23 =
   If[ FreeQ[ {x, y} /. sol23, Complex] , {x, y} /. sol23, {(points[[2]] + points[[3]])/2, (points[[2]] + points[[3]])/2}];
  sol34 = Solve[ {Total[ ({x, y} - points[[3]])^2 ] == alpha^2, Total[ ({x, y} - points[[4]])^2 ] == alpha^2} ];
centres34 =
   If[ FreeQ[ {x, y} /. sol34, Complex] , {x, y} /. sol34, {(points[[3]] + points[[4]])/2, (points[[3]] + points[[4]])/2}];
  sol41 = Solve[ {Total[ ({x, y} - points[[4]])^2 ] == alpha^2, Total[ ({x, y} - points[[1]])^2 ] == alpha^2} ];
centres41 = If[ FreeQ[ {x, y} /. sol41, Complex] , {x, y} /. sol41,
    {(points[[4]] + points[[1]])/2, (points[[4]] + points[[1]])/2}]; Show[ pointsPlot, quadrilateral,
   Graphics[ {Thickness[0.005], Line[ {points[[1]], points[[3]]} ]} ],
   centreDots = Graphics[ {PointSize[ 0.02 ],
      Transpose[ {{Red, Black, Blue}, Map[ Point, {centres12, centres13, centres23} ]}]} ],
   Graphics[ {{
      {Red, Opacity[0.3], Circle[ centres12[[1]], alpha ]},
      {Red, Opacity[1.0], Circle[ centres12[[2]], alpha ]},
      {Black, Opacity[0.3], Circle[ centres13[[1]], alpha ]},
      {Black, Opacity[1.0], Circle[ centres13[[2]], alpha ]},
      {Blue, Opacity[0.3], Circle[ centres23[[1]], alpha ]},
      {Blue, Opacity[1.0], Circle[ centres23[[2]], alpha ]} (*,
      {Purple,Opacity[0.3],Circle[ centres34[[2]], alpha ]},
      {Purple,Opacity[1.0],Circle[ centres34[[1]], alpha ]},
      {Green, Opacity[0.3],Circle[ centres41[[1]], alpha ]},
      {Green, Opacity[1.0],Circle[ centres41[[2]], alpha ]} *)
     }} ], PlotRange -> {{0, 10}, {1, 10}}, ImageSize -> 500 ])
 , {{alpha, ccr1}, 0.05, 2.5, 0.01}]
```

Out[229]=



In this second animation, we start with alpha at the value of the circumcircle of the red-blue-black triangle, and as alpha is increased to around 3.13 the two centres of the green circles appear (the centres of the purple circles appeared earlier). Further increase of alpha show the centre of the black "inner" circle moving toward a second coincidence with the centres of the purple and green circles. At the point of coincidence, alpha equals the circumcircle radius of the black-purple-green triangle, and an increase in alpha at that point means that both of the black circles now contain other points in the set, so the black edge is no longer in the alpha hull, while the purple and green circle pairs now each fulfill the criterion. The alpha hull then consists of the red, blue, purple, and green edges.

This same process of growth of the alpha hull continues as alpha increases until, as noted above the alpha hull becomes the standard convex hull for values above the half-length of the longest edge in the convex hull.

In[230]:=
```
quadrilateral = Graphics[ Flatten[Transpose[ {{{Thickness[0.005], Red}, Blue, Purple, Green},
      Map[ Line, Partition[ points ~Join~ {First[ points ]}, 2, 1 ] ]} ], 2] ];
edges = Partition[ points ~Join~ {First[ points ]}, 2, 1 ];
Manipulate[
 (sol12 = Solve[ {Total[ ({x, y} - points[[1]])^2 ] == alpha^2, Total[ ({x, y} - points[[2]])^2 ] == alpha^2} ];
  centres12 =
   If[ FreeQ[ {x, y} /. sol12, Complex] , {x, y} /. sol12, {(points[[1]] + points[[2]])/2, (points[[1]] + points[[2]])/2}];
  sol13 = Solve[ {Total[ ({x, y} - points[[1]])^2 ] == alpha^2, Total[ ({x, y} - points[[3]])^2 ] == alpha^2} ];
centres13 =
   If[ FreeQ[ {x, y} /. sol13, Complex] , {x, y} /. sol13, {(points[[1]] + points[[3]])/2, (points[[1]] + points[[3]])/2}];
  sol23 = Solve[ {Total[ ({x, y} - points[[2]])^2 ] == alpha^2, Total[ ({x, y} - points[[3]])^2 ] == alpha^2} ];
centres23 =
   If[ FreeQ[ {x, y} /. sol23, Complex] , {x, y} /. sol23, {(points[[2]] + points[[3]])/2, (points[[2]] + points[[3]])/2}];
  sol34 = Solve[ {Total[ ({x, y} - points[[3]])^2 ] == alpha^2, Total[ ({x, y} - points[[4]])^2 ] == alpha^2} ];
centres34 =
   If[ FreeQ[ {x, y} /. sol34, Complex] , {x, y} /. sol34, {(points[[3]] + points[[4]])/2, (points[[3]] + points[[4]])/2}];
  sol41 = Solve[ {Total[ ({x, y} - points[[4]])^2 ] == alpha^2, Total[ ({x, y} - points[[1]])^2 ] == alpha^2} ];
centres41 = If[ FreeQ[ {x, y} /. sol41, Complex] , {x, y} /. sol41,
    {(points[[4]] + points[[1]])/2, (points[[4]] + points[[1]])/2}]; Show[ pointsPlot, quadrilateral,
   Graphics[ {Thickness[0.005], Line[ {points[[1]], points[[3]]} ]} ],
   centreDots = Graphics[ {PointSize[ 0.02 ], Transpose[ {{Red, Black, Blue, Purple, Green},
        Map[ Point, {centres12, centres13, centres23, centres34, centres41} ]}]} ],
   Graphics[ {{
      {Red, Opacity[0.3], Circle[ centres12[[1]], alpha ]},
      {Red, Opacity[1.0], Circle[ centres12[[2]], alpha ]},
      {Black, Opacity[0.3], Circle[ centres13[[1]], alpha ]},
      {Black, Opacity[1.0], Circle[ centres13[[2]], alpha ]},
      {Blue, Opacity[0.3], Circle[ centres23[[1]], alpha ]},
      {Blue, Opacity[1.0], Circle[ centres23[[2]], alpha ]} ,
      {Purple, Opacity[0.3], Circle[ centres34[[2]], alpha ]},
      {Purple, Opacity[1.0], Circle[ centres34[[1]], alpha ]},
      {Green, Opacity[0.3], Circle[ centres41[[1]], alpha ]},
      {Green, Opacity[1.0], Circle[ centres41[[2]], alpha ]}
     }} ], PlotRange -> {{0, 10}, {1, 10}}, ImageSize -> 500 ])
 , {{alpha, ccr1}, ccr1, 1.1*ccr2, 0.01}]
```

Out[232]=