In [1]:
```
%use kotlin-dl
```

In [2]:
```
// con tan pocos datos, los que hemos usado en regresiones tradicionales, la predicción de la RRNN es incorrecta
//     val x = listOf(1.0, 2.0, 3.0, 4.0, 5.0).toDoubleArray()
//     val y = listOf(2.0, 3.0, 5.0, 4.0, 6.0).toDoubleArray()
```

In [3]:
```
// Ejemplo para crear datos sintéticos para la regresión lineal
//                 y = (0.9 * x) + 1.3 + ruido
val x = (1..100).map { it / 100.0 }.toDoubleArray()
val y = x.map { 0.9 * it + 1.3 + Math.random() / 10 }
```

In [4]:
```
// Crear un modelo secuencial con una capa densa de una neurona
val model = Sequential.of(
    Input(1),
    Dense(1)
)
```

In [5]:
```
// Compilar el modelo con una función de pérdida de error cuadrático medio
// y un optimizador de descenso de gradiente estocástico
model.compile(
    optimizer = SGD(),
    loss = Losses.MSE,
    metric = Metrics.MSE // Calcular el R-cuadrado como métrica
)
```

In [6]:
```
// Crear un conjunto de datos en memoria con los datos de entrada y salida
val xFloat = x.map { it.toFloat() }.toFloatArray()
val yFloat = y.map { it.toFloat() }.toFloatArray()
val features = xFloat.map { floatArrayOf(it) }.toTypedArray()
val dataset = OnHeapDataset.create(features, yFloat)
```

In [7]:
```
// Entrenar el modelo durante 100 épocas con un tamaño de lote de 8
model.fit(dataset, epochs = 100, batchSize = 8)
```

Out[7]:
```
org.jetbrains.kotlinx.dl.api.core.history.TrainingHistory@37714eba
```

In [8]:
```
/*
 [m] es el valor del único peso dense_2_dense_kernel y [b] es el valor del único sesgo dense_2_dense_bias.
*/
```

```
println("Weights: ${model.layers[1].weights}")

val dense2DenseKernel = model.layers[1].weights["dense_2_dense_kernel"] as Array<FloatArray>
var m = dense2DenseKernel[0][0]
println("m: $m")


val dense2DenseBias = model.layers[1].weights["dense_2_dense_bias"] as Array<Float>
var b = dense2DenseBias[0]
println("b: $b")
```

```
Weights: {dense_2_dense_kernel=[[F@6e930091, dense_2_dense_bias=[Ljava.lang.Float;@77ab6061}
m: 0.8917015
b: 1.3563751
```

In [9]:
```
println("El modelo de regresión lineal resultante es ->   Y = "+m+"*X + "+ b)
```

```
El modelo de regresión lineal resultante es ->   Y = 0.8917015*X + 1.3563751
```

Dibujamos la recta del modelo matemático

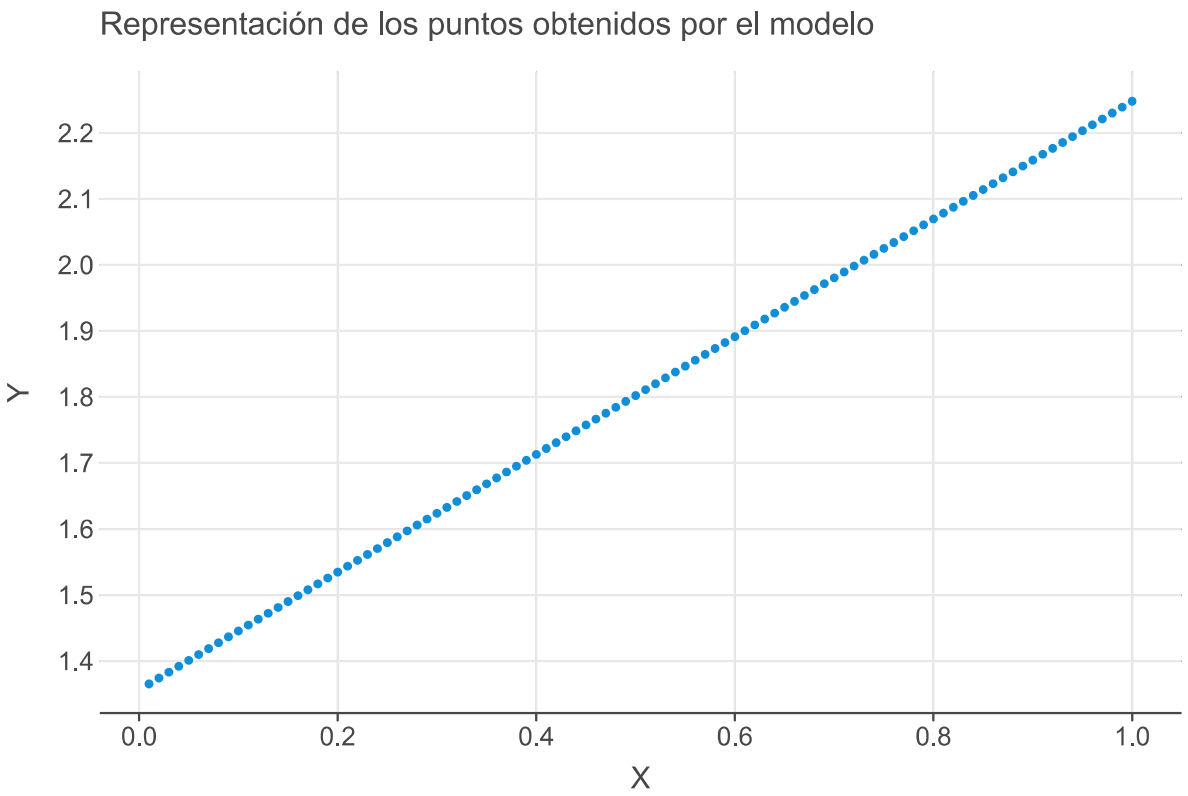
In [10]:
```
%use lets-plot
```

In [11]:
```
val d = mapOf(
    "xd" to (1..100).map { it / 100.0 }.toList(),
    "yd" to x.map { m * it + b }
)

val dOrigen =mapOf(
    "xo" to (1..100).map { it / 100.0 }.toList(),
    "yo" to y
)
```

In [12]:
```
val p = letsPlot(d) {x="xd"; y="yd"}
p + geomPoint() +
labs(title = "Representación de los puntos obtenidos por el modelo", x = "X", y = "Y", color = "yd")
```

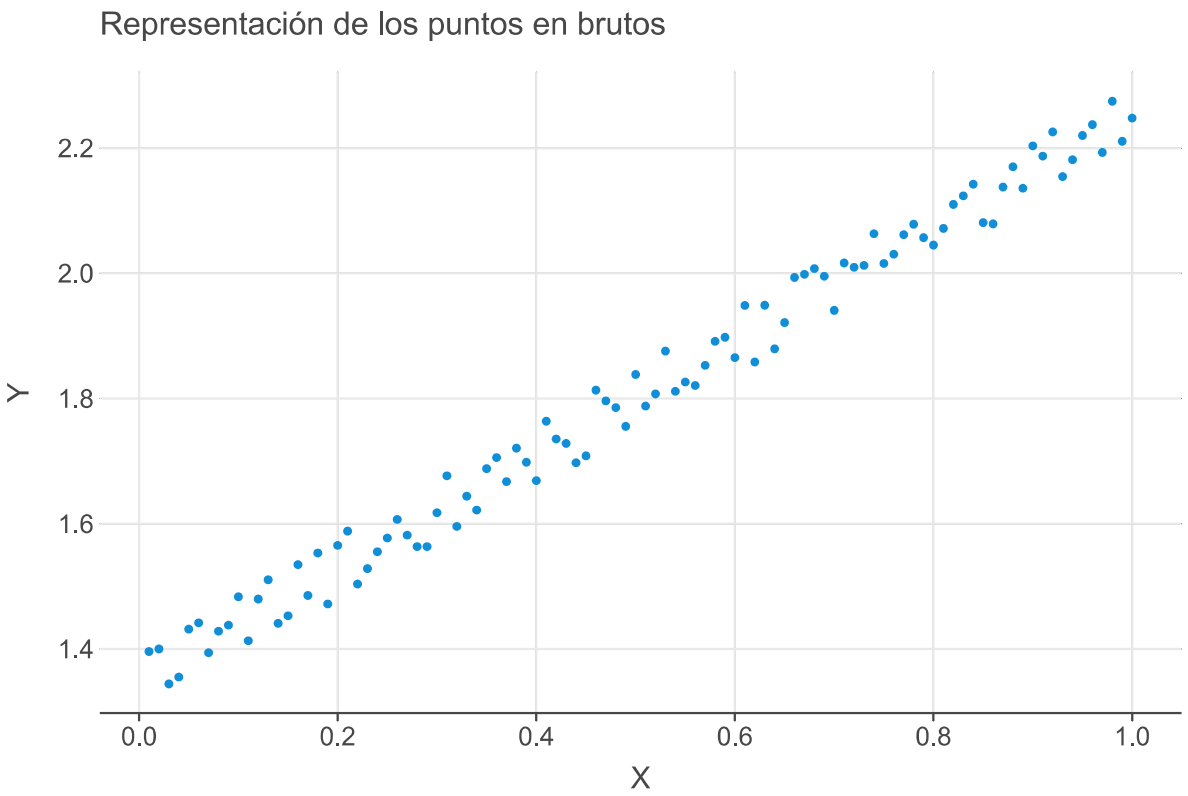
Out[12]:



In [13]:

```
val po = letsPlot(dOrigen) {x="xo"; y="yo"}
po + geomPoint() +
labs(title = "Representación de los puntos en brutos", x = "X", y = "Y", color = "y")
```

Out[13]:

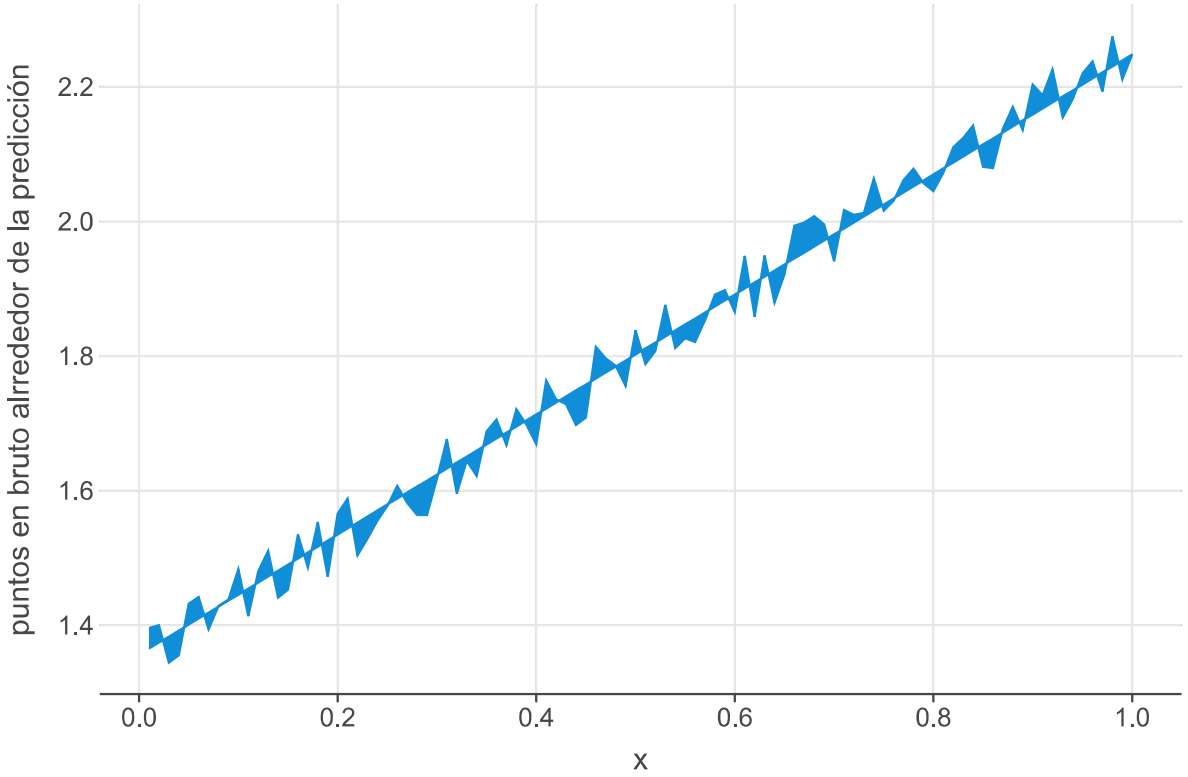


Usamos "ribbon" que nos sirve para marcar el área entre dos curvas

In [14]:
```
val dTodo  = mapOf(
    "xd" to (1..100).map { it / 100.0 }.toList(),
    "yd" to x.map { m * it + b },
    "yo" to y
)
```

In [15]:
```
letsPlot(dTodo) +
    geomRibbon() { x = "xd"; ymin = "yd"; ymax = "yo" } +
    xlab("x") + ylab("puntos en bruto alrrededor de la predicción")
```

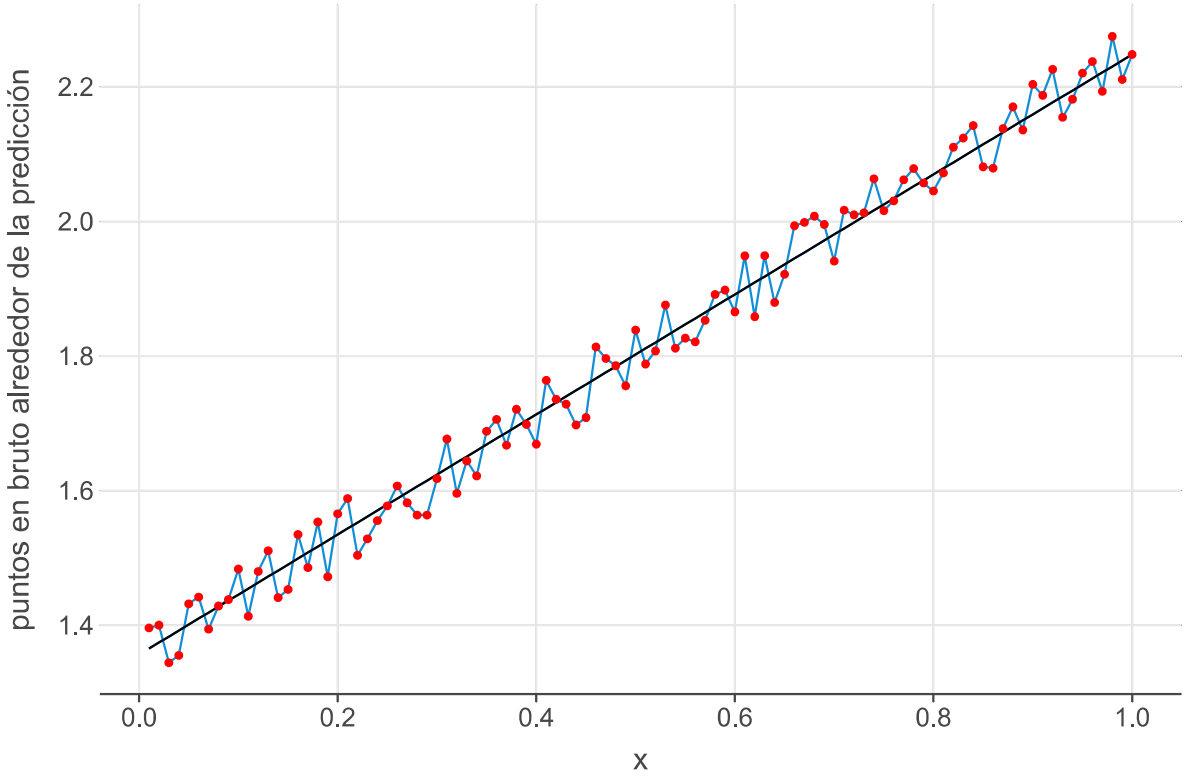
Out[15]:



In [16]:
```
val plot = letsPlot(dTodo) +
           geomRibbon(alpha = 0.0) { x = "xd"; ymin = "yd"; ymax = "yo" } +
           geomLine (color = "black") { x = "xd"; y = "yd" } +
           geomPoint (color = "red") { x = "xd"; y = "yo" } +
           labs(x = "x", y = "puntos en bruto alrededor de la predicción")
```

In [17]:
```
plot
```

Out[17]:


puntos en bruto alrededor de la predicción
2.2
2.0
1.8
1.6
1.4
0.0
0.2
0.4
0.6
0.8
1.0
x

In [ ]: