Algunos campos de actuación de ML. Ejemplos de implementación en Vertica, contrastados en Python y R.

ÍNDICE

ÍNDICE 2
1 INTRODUCCIÓN 3
2 INTRODUCCIÓN A LA LIBRERÍA VERTICA-ML-PYTHON 3
2.1 CONEXIÓN A BASE DE DATOS 4
2.2 VDATAFRAME Y VCOLUMN 4
2.3 MODELOS MACHINE LEARNING 4
3 DETALLES DE LA DATABASE 5
3.1 DW_FT_VENTAS (INDICACIONES) 5
4 KMEANS VERTICA 5
4.1 INTRODUCCIÓN 6
4.2 SINTAXIS EN VERTICA 6
4.3 EJEMPLO KMEANS (1) 7
4.4 EJEMPLO KMEANS (2) 9
4.5 EJEMPLO KMEANS (2) EN R 11
4.6 EJEMPLO KMEANS (2) EN PYTHON Y VERTICA-ML-PYTHON 13
4.7 CONTRASTE EJEMPLO KMEANS (2) 14
4.8 KMEANS + PCA 16
5 REGRESIÓN LINEAL 17
5.1 SINTAXIS 17
5.2 CONTRASTANTO RESULTADOS Y RENDIMIENTO CON R 18
5.3 EJEMPLO REGRESIÓN LINEAL EN PYTHON 21
5.4 CONTRASTE EJEMPLO REGRESIÓN LINEAL 22
6 RANDOM FOREST CLASIFICACIÓN: RIESGO CREDITICIO 23
6.1 SINTAXIS 23
6.2 EJEMPLO EN R: RIESGO CREDITICIO 24
6.3 EJEMPLO EN VERTICA 9.2: RIESGO CREDITICIO 31
6.4 EJEMPLO RANDOM FOREST EN PYTHON 34
6.5 CONTRASTE EJEMPLO RANDOM FOREST 35

1 INTRODUCCIÓN

En este documento estudiaremos posibles campos de actuación de algunos algoritmos de Machine Learning (ML). Buscamos estudiar la implementación de los algoritmos en Vertica y la librería Vertica-ML-Python con casos sencillos y comprobamos su rendimiento con las implementaciones en Python y R.

Enumeremos los algoritmos ML disponibles en Vertica:

1. KMEANS
2. REGRESIÓN LINEAL
3. CLASIFICACIÓN RANDOM FOREST
4. REGRESIÓN RANDOM FOREST
5. CLASIFICADOR SVM
6. REGRESOR SVM
7. NAIVE_BAYES

2 INTRODUCCIÓN A LA LIBRERÍA VERTICA-ML-PYTHON

Vertica_ml_python es una librería de Python que permite explotar las ventajas de Vertica, aplicando herramientas analíticas con la flexibilidad que proporciona Python, junto con la velocidad y escalabilidad de la base de datos de Vertica. La idea es que Python trabaja en memoria, a la hora de cargar los datos y procesarlos. La librería permite llevar la lógica del análisis a la fuente de datos (Vertica) y procesarlos allí, con un gasto de memoria y un tiempo de ejecución inferior a si utilizásemos únicamente Python. El problema que se busca resolver es poder enfocar la flexibilidad y utilidad de Python a entornos de Big Data, el crecimiento del uso de memoria y tiempo de ejecución escala de manera muy rápida conforme aumentamos el tamaño de la fuente de datos.

El objeto fundamental de la librería es el vDataframe, junto con sus vColumn. También podemos crear modelos de Machine Learning directamente en la base de datos, junto con algunas funcionalidades interesantes, como puede ser la lectura de un csv y cargarlo directamente en la base de datos.

En la documentación oficial se exploran algunas de las diferencias entre utilizar Vertica + Python vs utilizar únicamente librerías de Python (Pandas + Scikit). Los resultados más destacados son: tiempo de carga inferior en Vertica puesto que los datos se manipulan en el origen, no hace falta cargarlos en memoria como hace pandas; tamaño en memoria menor con la librería Vertica, el objeto vDataframe no guarda todos los datos como haría el Dataframe de pandas; tiempo de ejecución similar, Pandas tiene mejores resultados en ciertas funciones, aunque se puede aumentar la velocidad aumentando el número de nodos de Vertica.

2.1 CONEXIÓN A BASE DE DATOS

Para conectarse a la base de datos de Vertica para utilizar Vertica-ML-Python, tenemos varias opciones:

- JDBC: mediante la librería jaydebeapi. Sin embargo, no funciona bien con la librería Vertica-ML-Python, pues para crear un vDataframe en el código usa una concatenación de métodos del cursor. El método execute() del cursor de jaydebeapi devuelve None en lugar del propio cursor y da error.
- ODBC: Tenemos dos opciones y además podemos crear un DSN que facilite la conexión a la base de datos. En mis pruebas no he conseguido que con el DSN funcione en Windows, pues la librería Vertica-ML-Python tarde o temprano busca el DSN en ODBC.INI, y Windows gestiona los DSN de otra manera. Con lo cual, escribiendo la información de la conexión en el código, podemos crear el cursor necesario con vertica_python o con pyodbc.

```
conn_info = {'host': "", 'port': 5433,
             'user': "", 'password': "",
             'database': ""}
conn = vertica_python.connect(** conn_info)
cur = conn.cursor()
```

2.2 VDATAFRAME Y VCOLUMN

vertica_ml_python.vdataframe:

vDataframe es un objeto de Python que recuerda las modificaciones del usuario para usar una consulta SQL optimizada (cuando el usuario filtra los datos, vDataframe mantiene todas las transformaciones en memoria). Para cada columna de la relación, vDataframe crea una vColumn que almacena para cada columna su nombre y sus transformaciones; vDataframe permite preparación y exploración de los datos sin modificarlos.

vColumn y vDataframe usan la información del otro; es importante conocer ambas estructuras para saber cómo utilizar el objeto. En la documentación oficial vienen listados todos y cada uno de los métodos disponibles (además de tenerlos como código abierto), podemos destacar funciones de agregación, de dibujo de gráficos, aplicar una función a los elementos, rellenar campos vacíos…

2.3 MODELOS MACHINE LEARNING

vertica_ml_python.learn:

También podemos crear modelos directamente en la base de datos y entrenarlos. Según he visto, el código interno lo que hace es ejecutar la sentencia del modelo seleccionado en Vertica (es decir, las funcionalidades que también vamos a estudiar aquí), sin embargo, también añade algunas funcionalidades de estudio de los modelos que Vertica no dispone de manera natural, por ejemplo, el coeficiente de determinación R^2 en Regresión Lineal.

Además de crearlos, podemos cargarlos o borrarlos de la base de datos.

3 DETALLES DE LA DATABASE

3.1 Dw_ft_ventas (indicaciones)

De la tabla de hechos estas son las columnas que más me pueden interesar. Entiendo que son las variables relacionadas con las ventas de neumáticos y sus valores cuantificables de interés. Me fijaré en el cliente 415,107.

1. Tarifa_codigo: un mismo producto (neumático) se puede vender a distintos precios. Por ejemplo, puedes tener la tarifa A de Venta al Público, con PVP a 100 y la tarifa B de Grandes Clientes con PVP a 90.
2. Tar_pcoste: coste del producto. El coste total será, por tanto, tar_pcoste*cantidad. En nuestro caso, Tar_pcoste= 23.965. Coste total del producto o importe de coste = 23.965 * 4 = 95.86
3. Precio: precio definitivo de venta del artículo. En nuestro caso, precio = 32.58. Precio total de venta = precio * 4 = 32.58 * 4 = 130.32. Aplicando el descuento del 5%. Precio total de venta = 123.804.
4. Importe = 123.804
5. Dto: descuento aplicado al precio de venta final. Es del 5%
6. Margen = Importe – Importe de Coste = 95.86

Según JC lo importante son las siguientes columnas:

1. Cantidad = 4
2. Precio de Coste tar_pcoste
3. Precio
4. Importe de coste = tar_pcoste * cantidad
5. Importe
6. Margen de importe:

4 KMEANS VERTICA

4.1 INTRODUCCIÓN

El procedimiento que llevaré a cabo para analizar la utilidad de las implementaciones ML integradas en Vertica consistirá en la realización de casos sencillos y el contraste de los resultados obtenidos con los correspondientes en R.

4.2 SINTAXIS EN VERTICA

1. En primer lugar, consultemos la sintaxis empleada por Vertica del algoritmo:

Syntax

```
KMEANS ( 'model-name', 'input-relation', 'input-columns', 'num-clusters'
         [ USING PARAMETERS [exclude_columns='excluded-columns']
                            [, max_iterations=max-iterations]
                            [, epsilon=epsilon-value]
                            [, {init_method='init-method'} | {initial_centers_table='init-table'} ]
                            [, output_view='output-view']
                            [, key_columns='key-columns'] ] )
```

Las explicaciones sobre los argumentos obligatorios y los opcionales las iremos haciendo sobre la marcha.

Tratemos de construir una clasificación no supervisada de los clientes de la base de datos. Esta clasificación depende también de la iniciativa de la empresa. Atendiendo a las columnas de la tabla de hechos dw_ft_ventas la variable que mayor interés pudiera tener de cara a la segregación del tipo de clientes es "importe total de todas las compras efectuadas por el cliente". Así, tendremos una clasificación en función del desembolso económico que lleve a cabo para la empresa, esto es, si el número de clusters resultantes es de 3, podríamos considerar la clasificación de cliente importante, mediano y pequeño. Otra posibilidad es verlo en función del beneficio, es posible que para una compra de menor importe exista un mayor margen de beneficio frente a una compra de mayor importe.

Si bien, se pueden tener otros factores en cuenta, e.g., no deberíamos considerar un cliente únicamente por el desempeño económico del mismo, sino también por su frecuencia ya que no es lo mismo para el negocio que un cliente haga una aportación de 1000 euros en una sola compra a que efectúe 10 compras de 100 euros en distintas ocasiones. Con esto último, de salir 3 clusters resultantes, tendríamos la clasificación de cliente recurrente, de media frecuencia u ocasional. Habría que definir también un lapso conveniente para establecer las frecuencias: 1 mes, 2 meses, 1 año, etc.

Podríamos establecer los grupos por separados, pero podrían no coincidir, lo mejor es combinar estos 2 criterios para formar los clusters de forma conjunta y formar otros grupos: cliente importante-ocasional, cliente importante-de media frecuencia, cliente importante-recurrente, cliente medio-de media frecuencia, cliente medio-ocasional, etc.

La naturaleza de los grupos requiere de una observación posterior por parte del analista.

A continuación, veamos dos ejemplos en Vertica:

Los 3 primeros párrafos declarativos se corresponden con la preparación de la tabla que se usará en el algoritmo.

4.3 EJEMPLO KMEANS (1)

```
CREATE TABLE machine_learning.auxiliar1 AS
SELECT DISTINCT id_dim_cliente AS "cliente", "precio" AS precio
FROM machine_learning.dw_ft_ventas
WHERE id_fecha > 20130101 AND id_fecha < 20130401

DELETE
FROM machine_learning.auxiliar1
WHERE precio is NULL

CREATE TABLE machine_learning.pruebaJ1 AS
SELECT DISTINCT cliente, sum(precio) AS "TOTAL"
FROM machine_learning.auxiliar1
GROUP BY cliente
ORDER BY cliente

SELECT KMEANS('PRUEBA_CLUSTER1','pruebaJ1', '*', 3
USING PARAMETERS exclude_columns = 'cliente', max_iterations = 25, output_view = 'KMEANS1')
```

Si nos fijamos, en el primer párrafo declarativo tenemos las ventas y los clientes correspondientes a un trimestre, hemos borrado posibles valores nulos que KMEANS no aceptará. En el segundo y en el tercero hemos calculado los desembolsos totales por cada cliente. En el cuarto párrafo, hacemos una llamada al algoritmo, donde figuran 4 argumentos obligatorios:

1. "PRUEBA_CLUSTER1": El nombre que asignamos al modelo.
2. 'pruebaJ1': la tabla donde el algoritmo buscará los datos de interés.
3. El tercer parámetro es '*' y sirve para informar de cuántas filas queremos que participen en el modelo.
4. El número 3 indica el número de grupos en los que queremos la segregación de los datos.

En cuanto al conjunto de argumentos opcionales su función la indica su etiqueta. Con el último argumento opcional obtenemos una vista cuyo nombre es output_view = 'KMEANS1'.

Ejecutando todo obtenemos la siguiente vista:

123 total	123 cluster_id
12,070.33	2
20,394.262	1
273.09	0
739.92	0
2,646.14	2
1,196.42	0

Observamos que se han formado los grupos 0,1,2. Deducimos ya que el grupo de los grandes clientes es el 1, que el 2 es el de los clientes medianos y que 0 es el de los clientes pequeños.

Si calculamos los centros se tiene que:

123 cluster_id	123 CMONEY	123 CARDINAL
0	573.49	1,319
1	23,781.38	4
2	4,527.36	93

Observamos los centros y concluimos que 1319 clientes se han considerado por el algoritmo como clientes pequeños, que 93 se consideran medianos, y que tan solo 4 se consideran clientes grandes en comparación al resto. A continuación, contrastaremos el resumen del método con los cálculos recientemente hechos para tener fiabilidad de los resultados.

También podemos realizar la consulta que sigue:

```
SELECT GET_MODEL_SUMMARY(USING PARAMETERS model_name='PRUEBA_CLUSTER1')
```

Y obtenemos la siguiente lectura:

```
-------
 total
-----------
 573.48518
23781.37800
4527.35656


=======
metrics
=======
Evaluation metrics:
   Total Sum of Squares: 4.5522115e+09
   Within-Cluster Sum of Squares:
      Cluster 0: 4.2124331e+08
      Cluster 1: 2.0295006e+08
      Cluster 2: 4.6950196e+08
   Total Within-Cluster Sum of Squares: 1.0936953e+09
   Between-Cluster Sum of Squares: 3.4585162e+09
   Between-Cluster SS / Total SS: 75.97%
 Number of iterations performed: 18
 Converged: True
 Call:
kmeans('machine_learning.PRUEBA_CLUSTER1', 'pruebaJ1', '*', 3
USING PARAMETERS exclude_columns='cliente', max_iterations=25, epsilon=0.0001, init_method='kmeanspp', distance_method='euclidean', output_view='KMEANS1')
```

```
-------
 total
-----------
 573.48518
23781.37800
4527.35656
```

Nos fijamos en que lo primero que nos ofrece son los centros que coinciden con los que he calculado. También nos dice que el método ha sido convergente entre otros muchos otros datos.

OBSERVACIONES IMPORTANTES:

1. Veo un gran inconveniente a todo esto. Dicho inconveniente es que en la sintaxis y, en concreto, en el uso de KMEANS hay que transmitirle al algoritmo el número de grupos que quieres que se

formen. Esta decisión a menudo puede depender del criterio de un analista. Sin embargo, a veces este criterio puede ir contra la naturaleza de los grupos y forzarlos. Pongamos un ejemplo, lo deseado según el ejemplo anterior es que se formen 9 grupos como los que hemos comentado, no obstante, estos no tienen por qué existir de forma natural. Quizás no existan clientes cuya clasificación de frecuencia sea media, ya que las frecuencias tengan mucha dispersión y, en tal caso, lo natural es dividir en 2 grupos. Al transmitirle que se formen 3 grupos estaríamos forzando la clasificación.

2. Para solucionar lo anterior, se suele emplear clustering jerárquico y dendogramas. Estos últimos te ayudan a visualizar la clasificación natural de los clusters y adelantar un número adecuado a la clasificación por KMEANS. Es decir, es común que estos dos métodos se complementen en el orden clustering no jerárquico y un uso posterior de KMEANS.
3. Vertica no dispone de clustering no jerárquico eso supone emplear KMEANS a ciegas. Podríamos usar el clustering no jerárquico en R, pero perdería el sentido recurrir a R cuando lo que queremos es implementar el ML integrado en Vertica.

4.4 EJEMPLO KMEANS (2)

```
CREATE TABLE machine_learning.auxiliar2 AS
SELECT DISTINCT id_dim_cliente AS "cliente", sum(precio) AS precio, 1 AS "UNOS"
FROM machine_learning.dw_ft_ventas
WHERE id_fecha > 20130101 AND id_fecha < 20130401
GROUP BY id_dim_cliente, id_fecha

DELETE
FROM machine_learning.auxiliar2
WHERE precio is NULL

CREATE TABLE machine_learning.pruebaJ2 AS
SELECT DISTINCT cliente, sum(UNOS) AS "Frecuencia", sum(precio) AS "TOTAL"
FROM machine_learning.auxiliar
GROUP BY cliente
ORDER BY cliente
```

Calculemos los grupos en función de la frecuencia de sus compras, y el desembolso económico total:

```
SELECT *
FROM clasificacion_clientes
ORDER BY cluster_id, frecuencia, total
```

Implementamos ahora el algoritmo:

```
SELECT KMEANS('CLUSTER_PRUEBA2','pruebaJ2', '*', 9
USING PARAMETERS exclude_columns = 'cliente', max_iterations = 25, output_view = 'KMEANS2')
```

Calculamos sus centros:

```
SELECT cluster_id,
CAST(SUM(frecuencia)/COUNT(frecuencia)/90 AS DECIMAL(8,2)) AS 'CFRECUENCIA',
CAST(SUM(frecuencia)/COUNT(frecuencia) AS DECIMAL(8,2)) AS 'x1_ABSFRECUENCIA',
ROUND(SUM(total)/COUNT(total),2) AS 'x2_CMONEY', COUNT(cluster_id) AS 'CARDINAL'
FROM KMEANS2
GROUP BY cluster_id
ORDER BY cluster_id
```

Y observamos la query:

	123 cluster_id	123 CFRECUENCIA	123 x1_ABSFRECUENCIA	123 x2_CMONEY	123 CARDINAL
1	0	0.03	2.51	320.47	997
2	1	0.17	15.25	5,938.87	36
3	2	0.66	59.00	54,734.42	1
4	3	0.07	6.14	1,437.7	272
5	4	0.12	10.55	3,185.74	94
6	5	0.60	54.00	27,412.57	2
7	6	0.23	21.00	19,494.24	2
8	7	0.20	18.40	10,195.32	5
9	8	0.32	29.14	13,769.33	7

Vemos los 9 grupos que hemos querido crear con medias en (x_1, x_2).

Nota: Deberíamos haber considerado estandarizar los datos debido a la diferencia de magnitud entre una variable y otra.

Con la declaración:

```
SELECT GET_MODEL_SUMMARY(USING PARAMETERS model_name='CLUSTER_PRUEBA2')
```

obtenemos la lectura:

```
=======
centers
=======
frecuencia|  total
----------+-----------
 2.51254 | 320.47036
15.57143 |5977.26354
59.00000 |54734.41700
 6.13603 |1437.69888
10.48421 |3200.57105
54.00000 |27412.57100
21.00000 |19494.24000
18.40000 |10195.31960
29.14286 |13769.32943
=======
metrics
=======
Evaluation metrics:
   Total Sum of Squares: 8.5532417e+09
   Within-Cluster Sum of Squares:
      Cluster 0: 52751959
      Cluster 1: 33651678
      Cluster 2: 0
      Cluster 3: 43229034
      Cluster 4: 38204852
      Cluster 5: 7980997.3
```

Coinciden como era de esperar. Sin embargo, me llama la atención de que en el segundo centro veo una diferencia en la segunda coordenada de dicho centro. Por lo demás todo es correcto.

Vemos que tenemos 4 elementos de un solo elemento. Esto podría sugerir que hemos forzado 9 grupos cuando lo natural podría resultar que fueran 4 o 5. Sin embargo, también es posible que se traten de valores atípicos.

En cualquier caso, ya vemos una clasificación que podría ser interesante. Por ejemplo, los clientes que se agrupan entorno a medias $(2.51, 320.47)$ son los clientes pequeños y ocasionales son los que más predominan y al ser el cardinal de 997 suponen un aporte económico importante para la empresa y no se deben desatender. Por otra parte, también podemos visualizar que uno de los grupos de cardinal 1 (formado por un solo elemento) es de medias $(59, 54734.42)$, esto es, se trata de un cliente que aporta los mayores beneficios en comparación al resto de los clientes y cuya frecuencia es la más importante. Este último se clasificaría como un cliente importante y recurrente. Si quisiéramos saber quién es este distinguido cliente fijémenos en su cluster_id y hagamos la query correspondiente:

```
SELECT cliente_nombre
FROM machine_learning.dw_dim_cliente
INNER JOIN (
SELECT *
FROM KMEANS3
WHERE cluster_id=2 ) AS aux
ON dw_dim_cliente.id_dim_cliente = aux.cliente
```

Para un posterior contraste de resultados con R, necesitaremos fijar los mismos centros semilla en Vertica que en R y utilizar la distancia euclídea en ambas.

4.5 EJEMPLO KMEANS (2) EN R

Utilizaremos esta sección para expresar las capacidades de R para con ello poner de manifiesto las limitaciones de Vertica 9.2 en cuanto al algoritmo KMEANS se refiere.

Con los mismos datos calculados con Vertica en el ejemplo anterior:

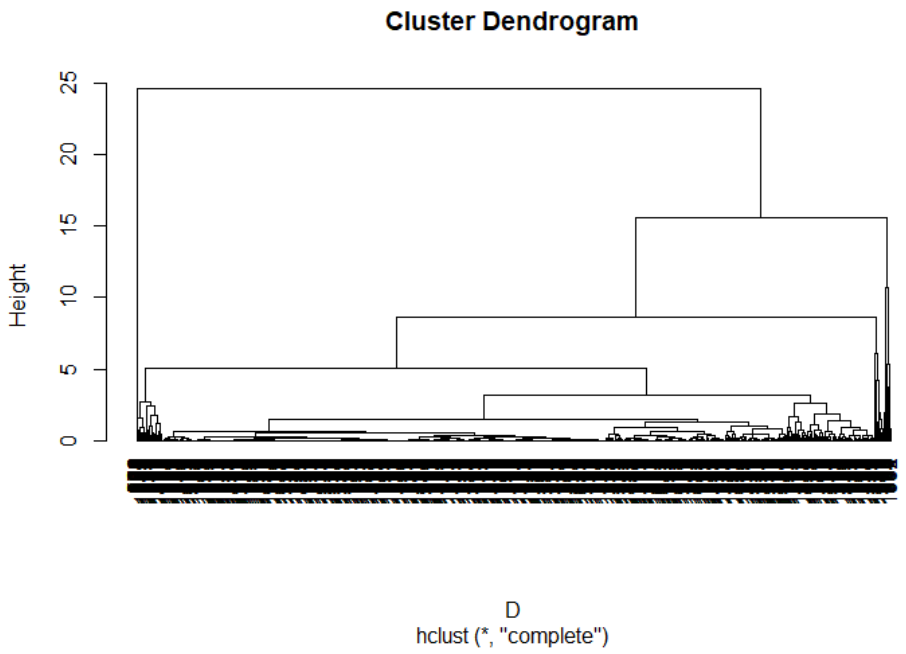
```
1. EJEMPLO <- read.csv("EJEMPLO_KMEANS.csv", sep=",",
   header = TRUE)
2. EJEMPLO <- EJEMPLO[,2:3]
3. head(EJEMPLO)
4. EJEMPLO<-scale(EJEMPLO)
```

```
  Frecuencia    TOTAL
1         37 18215.81
2         57 29410.19
3          4   273.09
4         14   773.82
5          9  3192.52
6         21  1262.24
```

Primero hagamos un análisis jerárquico del dataset:

Calculamos la matriz de distancias correspondiente y empleamos el modelo jerárquico:

```
1. D<-dist(EJEMPLO)
2. PJ<-hclust(D)
3. plot(PJ, hang=-1)
```



En el dendograma podemos ver que lo natural es considerar 3-5 grupos.

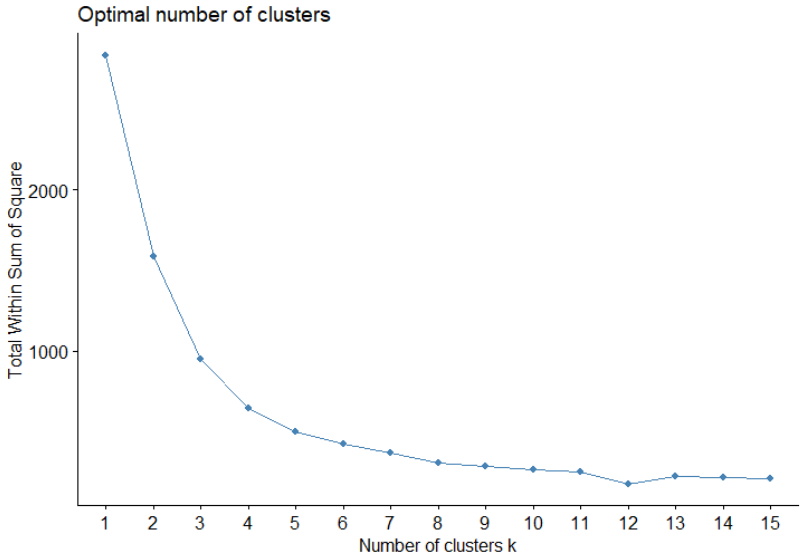
```
1. KM<-kmeans(EJEMPLO,5)
2. KM$cluster
3. table(KM$cluster)
```

```
> table(KM$cluster)

   1    2    3    4    5
 281  103   16 1013    3
```

Veamos ahora cuál es el número óptimo de clusters con el análisis de R:

```
1. library(cluster)
2. library(factoextra)
3. fviz_nbclust(x = EJEMPLO, FUNcluster = kmeans, method = "wss",
   k.max = 15,
4.              diss = get_dist(EJEMPLO, method = "euclidean"),
   nstart = 50)
```



Vemos que a partir de 5 empieza a estabilizarse por lo que nuestra elección no fue equivocada.

4.6 EJEMPLO KMEANS (2) EN PYTHON Y VERTICA-ML-PYTHON

Vamos a implementar el modelo en Python, con la librería de Vertica y sin la librería (usando Scikit-learn). En el siguiente apartado vemos los resultados y hacemos el contraste global.

Vertica-ml-python:

```
#### KMEANS ####
from vertica_ml_python.learn.cluster import KMeans
#model = load_model(name = "machine_learning.kmeans_pruebaJ2", cursor = cur)
model = KMeans(name = "machine_learning.kmeans_pruebaJ2", cursor = cur, n_cluster = 9)
model.fit("machine_learning.pruebaJ2", ["Frecuencia", "TOTAL"])
print(model.metrics)
print(model.cluster_centers)
```

Como vemos, no hace falta cargar ningún dato. En el método fit le pasamos la tabla de la base de datos donde se alojan nuestros datos de entrenamiento para ajustar el modelo.

Python:

```
import pandas as pd
from sklearn.cluster import KMeans

tableExample = pd.read_csv(r'C:\Users\Kevin Moreno\Documents\verticaML\tabla_ejemploKmeans.csv')
df = pd.DataFrame(tableExample, columns = ['Frecuencia', 'TOTAL'])
kmeans = KMeans(n_clusters=9).fit(df)

print('------------CENTROS-------------')
print(kmeans.cluster_centers_)
print('------------INERCIA-------------')
totSS = df.var().sum()*(df.shape[0]-1)
betSS = totSS - kmeans.inertia_
witSS = kmeans.inertia_
print('Total Sum of Squares                          ', totSS)
print('Total Within-Cluster Sum of Squares           ', witSS)
print('Total Between-Cluster Sum of Squares          ', betSS)
print('Between-Cluster / Total SS                    ', betSS/totSS)
```

Con Python hay que cargar el csv y guardarlo en memoria en un DataFrame de Pandas.

4.7 CONTRASTE EJEMPLO KMEANS (2)

Con los datos del ejemplo KMeans2, realizo un modelo sencillo en las 4 plataformas y estudio las diferencias de los resultados.

Vertica:

```
centers
=======
frecuencia|    total
----------+----------
  6.12868 |1443.11722
 57.00000 |29410.19200
 10.61458 |3240.24615
 59.00000 |54734.41700
 16.00000 |6220.25634
  2.51403 | 321.03249
 21.00000 |19494.24000
 51.00000 |25414.95000
 25.60000 |12949.25060
```

```
=======
metrics
=======
Evaluation metrics:
     Total Sum of Squares: 8.5532417e+09
     Within-Cluster Sum of Squares:
         Cluster 0: 42921581
         Cluster 1: 0
         Cluster 2: 42216230
         Cluster 3: 0
         Cluster 4: 46355250
         Cluster 5: 53066380
         Cluster 6: 3269278.5
         Cluster 7: 0
         Cluster 8: 21279124
     Total Within-Cluster Sum of Squares: 2.0910784e+08
     Between-Cluster Sum of Squares: 8.3441339e+09
     Between-Cluster SS / Total SS: 97.56%
```

Vertica ML Python:

```
                                                    value
Between-Cluster Sum of Squares                 8363130000.0
Total Sum of Squares                           8553241700.0
Total Within-Cluster Sum of Squares             190111700.0
Between-Cluster SS / Total SS            0.9777731406795157
converged                                             True
            frecuencia               total
0     10.5376344086022     3170.7957311828
1                 18.4          10195.3196
2     2.51253761283852    320.470357071214
3                 59.0           54734.417
4     6.13602941176471    1437.69888235294
5                 54.0           27412.571
6                 21.0            19494.24
7     29.1428571428571    13769.3294285714
8     15.1621621621622    5902.01272972973
```

R:

```
> cat("Between-Cluster Sum of Squares            ", KM$betweenss)
Between-Cluster Sum of Squares            8378053880
> cat("Total Sum of Squares                      ", KM$totss)
Total Sum of Squares                      8553241696
> cat("Total Within-Cluster Sum of Squares       ", KM$tot.withinss)
Total Within-Cluster Sum of Squares       175187816
> cat("Between-Cluster SS / Total SS             ", KM$betweenss/KM$totss)
Between-Cluster SS / Total SS             0.979518
> KM$centers
  Frecuencia      TOTAL
1   6.142857  1403.1007
2   4.027778   656.6447
3  26.636364 13428.0287
4  15.300000  6420.5482
5   8.604167  2432.4018
6  59.000000 54734.4170
7  37.666667 25199.2707
8  12.795918  3996.4908
9   1.864035   187.1044
```

Python:

```
------------CENTROS-------------
[[2.23121387e+00 2.56333232e+02]
 [2.10000000e+01 1.94942400e+04]
 [1.57812500e+01 6.33970631e+03]
 [5.90000000e+01 5.47344170e+04]
 [8.11764706e+00 2.14206976e+03]
 [2.56000000e+01 1.29492506e+04]
 [4.96153846e+00 1.00110822e+03]
 [1.19285714e+01 3.81115741e+03]
 [5.40000000e+01 2.74125710e+04]]
------------INERCIA-------------
Total Sum of Squares                          8553241695.926128
Total Within-Cluster Sum of Squares           155544996.73001078
Total Between-Cluster Sum of Squares          8397696699.196117
Between-Cluster / Total SS                    0.9818144976770508
```

En este ejemplo concreto, todos los resultados son muy parecidos. Sin embargo, aunque la diferencia sea sutil, se ve una pequeña diferencia del ajuste del modelo entre el algoritmo de Vertica, Vertica-ml-python y la implementación en R y Python. La implementación de Vertica-ml-python también tiene una ligera mejora contra el algoritmo de Vertica. (El ajuste viene dado por el valor Between-Cluster / Total SS)

4.8 KMEANS + PCA

En ocasiones, el número de variables partícipes en un KMEANS puede resultar muy elevado con una consecuente ralentización en su ejecución. No obstante, Vértica también incluye PCA (Análisis de Componentes Principales). Si bien, aunque PCA se utilice muy a menudo en revelaciones de relaciones ocultas sobre unas determinadas variables, también es útil a la hora de complementarse con KMEANS. Si PCA consigue recoger un buen porcentaje de variabilidad de las variables primitivas, se puede aplicar un KMEANS sobre las variables generadas por PCA (las proyecciones) con un número menor de variables. El único inconveniente es que la interpretabilidad de estas variables se pierde al proyectar, pero se puede resolver regresando a las coordenadas originales, algo para lo que Vertica proporciona una función:

Syntax

```
APPLY_PCA ( input-columns
                    USING PARAMETERS model_name='model-name'
                                     [, num_components=num-components]
                                     [, cutoff=cutoff-value]
                                     [, match_by_pos=match-by-position]
                                     [, exclude_columns='excluded-columns']
                                     [, key_columns='key-columns'] )
```

5 REGRESIÓN LINEAL

5.1 SINTAXIS

Syntax

```
LINEAR_REG ( 'model-name', 'input-relation', 'response-column', 'predictor-columns'
              [ USING PARAMETERS
                              [exclude_columns='excluded-columns']
                              [, optimizer='optimizer-method']
                              [, regularization='regularization-method']
                              [, epsilon=epsilon-value]
                              [, max_iterations=iterations]
                              [, lambda=lamda-value]
                              [, alpha=alpha-value] ] )
```

Sobre los argumentos obligatorios:

1. Model-name: el nombre que queremos dar al modelo.
2. Input-relation: se trata de la tabla con la que vamos a construir el modelo.
3. Response-column: la variable dependiente deseada para la predicción.
4. Predictor-columns: predictores que se van a usar.

Sobre los argumentos opcionales más representativos:

1. Exclude_columns: columnas que quieras excluir como predictoras
2. Optimizer: método de optimización para entrenar el modelo.
3. epsilon: cuánta precisión se desea.

OBSERVACIÓN: me resultan llamativos estos parámetros opcionales debido a que la regresión lineal tal y como yo la conozco nos permite realizar un cálculo determinado de los coeficientes estimadores mediante la fórmula:

$$\hat{\beta} = (X'X)^{-1}X'y$$

donde $\boldsymbol{X}$ es la matriz de predictores e $\boldsymbol{y}$ la de observaciones de la variable respuesta. Por lo tanto, desconozco la necesidad de aplicar un método iterativo como sustitución de un simple cálculo matricial.

INCONVENIENTES: un claro inconveniente para la aplicación de un modelo de regresión lineal es que el aprendizaje es supervisado y, con ello, menos susceptible de automatización. Hay que validar los supuestos sobre los que se construye para garantizar el éxito del mismo. Por tanto, no termino de comprender dónde radica el potencial de implementar este modelo ML en Vertica. Por ejemplo, en un caso de regresión lineal múltiple hay que verificar:

1. Tendencia lineal con representaciones.
2. No colinealidad entre los predictores.

3. Homocedasticidad de los residuos.
4. Normalidad de los residuos.
5. Independencia de los residuos.
6. Valores influyentes y outliers.
7. Principio de parsimonia en la selección de los predictores.
8. El número de observaciones ha de ser de hasta 15 o 20 veces el número de predictores.

A excepción de la validación 8, los supuestos 1-7 en principio creo que no se pueden validar en Vertica. Esto supone que cuando apliquemos un modelo de regresión lineal múltiple desde Vertica lo hagamos a ciegas. De hecho, una de las herramientas que nos proporciona rigor en la validación de tales supuestos son los contrastes de hipótesis y creo que Vertica no dispone de estos. Por consiguiente, para validar estos supuestos deberíamos emigrar a R o algún programa estadístico similar y, con ello, el sentido de utilizar ML integrado en Vertica para evitar una dependencia externa se pierde.

Una vez mencionado lo anterior, me dispongo al menos a contrastar los resultados de una regresión lineal múltiple en Vertica con los resultados que ofrece R. Utilizaré un ejemplo conocido en el que tales supuestos se verifican con R con el objetivo de verificar el éxito del algoritmo integrado en Vertica cuando las condiciones de su aplicación son ideales.

5.2 CONTRASTANTO RESULTADOS Y RENDIMIENTO CON R

No he encontrado en la naturaleza de los datos de la base de datos sobre la que hemos trabajado en los ejemplos de K-means ningún rango de aplicación de una regresión lineal múltiple. Esto se debe a que la base de datos contempla las ventas de neumáticos y una regresión lineal múltiple en una predicción de ventas cuando estas fluctúan por múltiples factores no me parece plausible.

Por lo tanto, utilizaré los datos de un ejemplo simulado y ya conocido. Los datos son los siguientes:

	INC_RATING	SOLVENCIA	CAV
1	30.53	100.00	58.33
2	34.56	50.00	72.22
3	30.16	78.57	38.89
4	15.14	35.71	27.78
5	-10.86	40.00	33.33
6	26.50	61.43	55.56
7	-26.25	21.43	13.89
8	35.29	47.14	77.78
9	1.96	32.86	25.00
10	-19.29	5.71	16.67
11	22.47	64.29	27.78
12	0.86	28.57	22.22
13	58.00	80.00	100.00
14	-8.67	42.86	30.56
15	14.78	58.57	61.11

Tenemos dos variables independientes SOLVENCIA Y CAV que intentan predecir el incremento de un rating (INC_RATING). Guardamos el csv en un txt y con una transformación de spoon los migramos a la base de datos.

	123 INC_RATING	123 SOLVENCIA	123 CAV
1	1.96	32.86	25
2	22.47	64.29	27.78
3	26.5	61.43	55.56
4	30.16	78.57	38.89
5	30.53	100	58.33
6	34.56	50	72.22
7	58	80	100
8	14.78	58.57	61.11
9	35.29	47.14	77.78
10	-26.25	21.43	13.89
11	-19.29	5.71	16.67
12	-10.86	40	33.33
13	-8.67	42.86	30.56
14	0.86	28.57	22.22
15	15.14	35.71	27.78

SOLVENCIA Y CAV (crecimiento acumulado de ventas) son las variables predictoras, INC_RATING la variable que queremos predecir. Hagamos la consulta correspondiente para emplear el algoritmo:

```
SELECT LINEAR_REG('REG_LINEAL_MODEL4', 'REG_LINEAL_AUXILIAR', 'INC_RATING', '*'
USING PARAMETERS exclude_columns = 'INC_RATING', max_iterations = 25)
```

Obtengamos un resumen del modelo creado:

```
--RESUMEN DEL MODELO
SELECT GET_MODEL_SUMMARY(USING PARAMETERS model_name='REG_LINEAL_MODEL4')
```

```
=======
details
=======
predictor|coefficient|std_err |t_value |p_value
---------+-----------+--------+--------+--------
Intercept| -29.90742 | 6.61988|-4.51782| 0.00070
solvencia|  0.38443  | 0.14855| 2.58782| 0.02375
   cav   |  0.55447  | 0.14420| 3.84520| 0.00233
==============
regularization
==============
type| lambda
----+--------
none| 1.00000
===========
call_string
===========
linear_reg(
machine_learning.REG_LINEAL_MODEL4
,
REG_LINEAL_AUXILIAR
,
"inc_rating"
,
*
USING PARAMETERS exclude_columns=
```

Observamos que el resumen nos ofrece los coeficientes estimadores, sin embargo, no nos proporciona el coeficiente de determinación R^2 ni el ajustado. Luego los calcularemos.

Si quisiéramos las predicciones en Vertica, mediante la consulta que sigue las obtendríamos con la sintaxis que sigue:

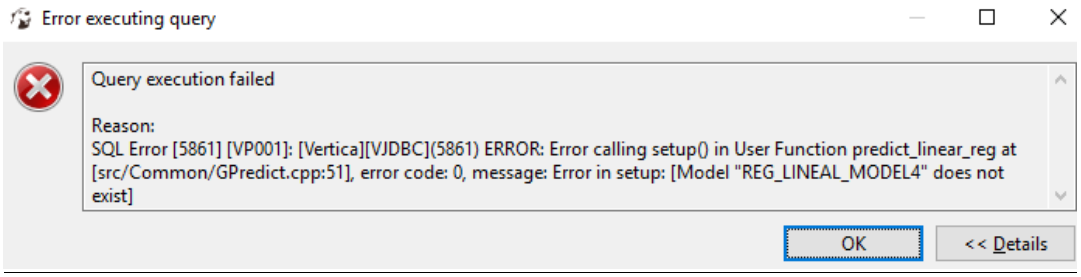
Syntax

```
PREDICT_LINEAR_REG ( input-columns
                     USING PARAMETERS model_name='model-name'
                                      [, match_by_pos=match-by-position] )
```

Procedemos:

```
SELECT PREDICT_LINEAR_REG(SOLVENCIA, CAV USING PARAMETERS model_name='REG_LINEAL_MODEL4')
FROM machine_learning.REG_LINEAL_AUXILIAR;
```

Y obtenemos el siguiente error:



He consultado el error en internet, también he consultado otras fuentes para ver si la sintaxis se trataba de la misma pero no he conseguido solucionarlo, por lo que deduzco que se trata de un error en Vertica. Por tanto, no puedo contrastar el método con R mediante los coeficientes de determinación. Sin embargo, para predicciones con constantes sí funciona y mantengo casi la sintaxis intacta lo que refuerza mi idea de que es un error de Vertica y no mío:

Tomando el primer dato de la tabla $(32.86, 25)$:

```
SELECT PREDICT_LINEAR_REG(32.86, 25 USING PARAMETERS model_name='REG_LINEAL_MODEL4',
match_by_pos='true')
FROM machine_learning.REG_LINEAL_AUXILIAR
LIMIT 1;
```

	123 PREDICT_LINEAR_REG
1	-3.4134086344

Comparamos con R:

```
> x0<-data.frame(SOLVENCIA=32.86,CAV=25) #Una variable nuevas
> predict(modelo,x0,interval="confidence")
       fit       lwr      upr
1 -3.413409 -11.22024 4.393423
```

Los valores son iguales. Concluimos, al menos, que la recta ajustada es la misma según la lectura y los resultados.

CONCLUSION: parece que la regresión lineal múltiple en Vertica ajusta bien el modelo.

COMPARATIVA DE TIEMPO: otra observación que podemos hacer es que el tiempo empleado en Vertica es mayor para construir el modelo que imagino que será por el método iterativo que emplea. En Vertica he observado para distintas ejecuciones con el mismo training dataset que los tiempos de construcción oscilan en 0.8-3 segundos pronunciándose en 0.8. En R, sin embargo, el tiempo es nulo o casi nulo:

```
> TimeRL<-system.time(
+   modelo<-lm(INC_RATING~SOLVENCIA+CAV,data=Enisa_Simu
+ )
> TimeRL
   user  system elapsed
   0.00    0.02    0.03
>
```

5.3 EJEMPLO REGRESIÓN LINEAL EN PYTHON

Vertica-ml-python:

```
### LINEAR REGRESSION ###
from vertica_ml_python.learn.linear_model import LinearRegression
drop_model(name = "machine_learning.RLpy", cursor = cur)
modelRL = LinearRegression(name = "machine_learning.RLpy", cursor = cur)
modelRL.fit(input_relation = "machine_learning.REG_LINEAL_AUXILIAR",
            X = ["SOLVENCIA", "CAV"],
            y = "INC_RATING")
print(modelRL.coef)
print(modelRL.regression_report())
```

Python:

```
import pandas as pd
from sklearn.linear_model import LinearRegression

tableExample = pd.read_csv(r'C:\Users\Kevin Moreno\Documents\verticaML\tabla_ejemploRL.csv')
X = pd.DataFrame(tableExample, columns = ['SOLVENCIA','CAV'])
y = pd.DataFrame(tableExample, columns= ['INC_RATING'])
#df = pd.DataFrame(tableExample, columns = ['Frecuencia', 'TOTAL'])
reg = LinearRegression().fit(X,y)
print('----------VALOR R2----------')
print(reg.score(X,y))
print('----------COEFICIENTES---------')
print(reg.coef_)
print('----------INTERCEPT-----------')
print(reg.intercept_)
```

5.4 CONTRASTE EJEMPLO REGRESIÓN LINEAL

Vertica:

```
predictor|coefficient|std_err |t_value |p_value
---------+-----------+--------+--------+--------
Intercept| -29.90742 | 6.61988|-4.51782| 0.00070
solvencia|  0.38443  | 0.14855| 2.58782| 0.02375
   cav   |  0.55447  | 0.14420| 3.84520| 0.00233
```

Vertica-ML-Python:

```
     predictor          coefficient              std_err              t_value   \\
0    Intercept    -29.9074180678622     6.61987851198836    -4.51781977776494   \\
1    solvencia    0.384430385437321    0.148553830817263     2.58781872754403   \\
2          cav    0.554465078718375    0.144196721984534     3.84519891359143   \\
                  p_value
0    0.000704468812005089
1      0.0237531755324546
2     0.00233037306374696
                                     value
explained_variance          0.8232631544202
max_error                   15.916369117099
median_absolute_error      8.29957577269426
mean_absolute_error        8.28753278624023
mean_squared_error         91.3443245510478
r2                          0.8232631544202
```

R:

```
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -29.9074     6.6199  -4.518 0.000704 ***
SOLVENCIA     0.3844     0.1486   2.588 0.023753 *
CAV           0.5545     0.1442   3.845 0.002330 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 10.69 on 12 degrees of freedom
Multiple R-squared:  0.8233,    Adjusted R-squared:  0.7938
F-statistic: 27.95 on 2 and 12 DF,  p-value: 3.048e-05
```

Python:

```
----------VALOR R2----------
0.8232631544201996
----------COEFICIENTES---------
[[0.38443039 0.55446508]]
----------INTERCEPT----------
[-29.90741807]
```

En este ejemplo concreto, al ser muy sencillo, el algoritmo nos devuelve el mismo resultado en todas las plataformas. Cabe mencionar que Vertica no nos proporciona el valor R^2, pero la librería Vertica-ML-Python sí.

6 RANDOM FOREST CLASIFICACIÓN: RIESGO CREDITICIO

6.1 SINTAXIS

La sintaxis de un Random Forest de Clasificación en Vertica es la siguiente:

Syntax

```
RF_CLASSIFIER ( 'model-name', input-relation, 'response-column', 'predictor-columns'
             [ USING PARAMETERS [exclude_columns='excluded-columns']
                                [, ntree=num-trees]
                                [, mtry=num-features]
                                [, sampling_size=sampling-size]
                                [, max_depth=depth]
                                [, max_breadth=breadth]
                                [, min_leaf_size=leaf_size]
                                [, min_info_gain=threshold]
                                [, nbins=num-bins] ] )
```

Argumentos obligatorios:

1. Model-name: el nombre que quieres asignar al modelo

2. Input-relation: la tabla que le introduces para aprendizaje
3. Response-column: variable respuesta u objetivo
4. Predictor-columns: variables columnas predictoras

Argumentos opcionales:

1. Exclude_columns = columnas predictoras que no quieres que participen en el modelo
2. Ntree = número de árboles utilizados en el modelo
3. Mtry = número de variables aleatorias como candidatas en cada ramificación
4. Sampling_size = la proporción del conjunto de datos de entrada que se selecciona aleatoriamente para entrenar cada árbol.
5. Max_depth = el máximo nivel de profundidad que puede alcanzar cada árbol.
6. Max_breadth = máximo número de nodos terminales que puede tener cada árbol.
7. Min_leaf_size = el mínimo tamaño que pueden alcanzar los nodos de cada árbol (las hojas).
8. Min_info_gain = umbral mínimo para incluir una división.

Default values:

1. $0 \leq$ ntree=20 $\leq \mathbf{1000}$
2. mtry = $\sqrt{\{number\ of\ predictors\}}$
3. $0 \leq$ sampling_size = 0.632 ≤ 1
4. $\mathbf{0} \leq$ max_depth = 5 ≤ 5
5. $\mathbf{1} \leq$ max_breadth=32 $\leq$ 1e9
6. $\mathbf{1} \leq$ min_leaf_size = 1 $\leq$ 1e6
7. $0 \leq$ min_info_gain = 0.0 ≤ 1

6.2 EJEMPLO EN R: RIESGO CREDITICIO

Uno de las posibles aplicaciones de un Random Forest es poder predecir si un prestatario no podrá pagar un futuro préstamo. Para ilustrar este caso utilizaremos un conjunto de datos que representa 9578 préstamos a 3 años que fueron financiados a través de la plataforma LendingClub.com entre mayo de 2007 y febrero de 2010. La variable dependiente not_fully_paid indica que el préstamo no se devolvió en su totalidad, esto es, el prestatario se marcó por la empresa como poco probable para la devolución del préstamo con sus intereses. Para predecir esta variable de interés, nosotros usaremos las siguientes variables independientes disponibles:

1. Credit.policy: 1 si el cliente cumple con los criterios de suscripción de crédito de LendingClub.com, y 0 en caso contrario.
2. Purpose: propósito del prestatario y préstamo: "credit_card", "debt_consolidation", "education", "major_purchase", "small_business", "all_other".
3. Int.rate: la tasa de interés del préstamo como proporción (una tasa del 11% se almacenaría como 0.11). A los prestatarios que LendingClub.com considera con mayor riesgo les asigna tasas de interés más altas.
4. Installment: las cuentas mensuales adeudadas por el prestatario si el préstamo decide financiarse.

5. Log.annual.inc: los ingresos anuales declarados por el prestatario.
6. DtI: La relación deuda-ingreso del prestatario (cantidad de deuda dividida por los ingresos anuales)
7. Fico: la puntuación de crédito FICO. El número de días que el prestatario ha tenido una línea de crédito.
8. Revol.bal: The borrower's revolving balance (amount unpaid at the end of the credit card billing cycle).
9. Revol.util: the borrower's revolving line utilization rate (the amount of the credit line used relative to total credit available).
10. Inq.last.6mths: the borrower's number of inquiries by creditors in the last 6 months.
11. Delinq.2yrs: el número de veces que el posible prestatario ha estado al menos hasta un mes en estado de impago con respecto a alguna deuda durante los últimos 2 años.
12. Pub.rec: The borrower's nunmber of derogatory public records (bankruptcy filings, tax liens or judgments).

Notemos todo lo que podemos hacer en R para posteriormente notar las limitaciones (o no) de Vertica si se emplea en solitario.

A. PREPARAMOS Y EXPLORAMOS LOS DATOS:

Cargamos los datos en R:

```
1. library(readr)
2. loans1 <- read.csv("loans.csv")
3. str(loans1)
4. summary(loans1)
```

```
   'data.frame':        9578 obs. of  14 variables:
 $ credit.policy    : int  1 1 1 1 1 1 1 1 1 1 ...
 $ purpose          : Factor w/ 7 levels "all_other","credit_card",..: 3
2 3 3 2 2 3 1 5 3 ...
 $ int.rate         : num  0.119 0.107 0.136 0.101 0.143 ...
 $ installment      : num  829 228 367 162 103 ...
 $ log.annual.inc   : num  11.4 11.1 10.4 11.4 11.3 ...
 $ dti              : num  19.5 14.3 11.6 8.1 15 ...
 $ fico             : int  737 707 682 712 667 727 667 722 682 707 ...
 $ days.with.cr.line: num  5640 2760 4710 2700 4066 ...
 $ revol.bal        : int  28854 33623 3511 33667 4740 50807 3839 24220 6
9909 5630 ...
 $ revol.util       : num  52.1 76.7 25.6 73.2 39.5 51 76.8 68.6 51.1 23
...
 $ inq.last.6mths   : int  0 0 1 1 0 0 0 0 1 1 ...
 $ delinq.2yrs      : int  0 0 0 0 1 0 0 0 0 0 ...
 $ pub.rec          : int  0 0 0 0 0 0 1 0 0 0 ...
 $ not.fully.paid   : int  0 0 0 0 0 0 1 1 0 0 ...
 credit.policy                 purpose        int.rate       installment

 Min.   :0.000   all_other         :2331   Min.   :0.0600   Min.   : 15.6
7
```

```
 1st Qu.:1.000   credit_card       :1262   1st Qu.:0.1039   1st Qu.:163.7
7
 Median :1.000   debt_consolidation:3957   Median :0.1221   Median :268.9
5
 Mean   :0.805   educational       : 343   Mean   :0.1226   Mean   :319.0
9
 3rd Qu.:1.000   home_improvement  : 629   3rd Qu.:0.1407   3rd Qu.:432.7
6
 Max.   :1.000   major_purchase    : 437   Max.   :0.2164   Max.   :940.1
4
                 small_business    : 619

 log.annual.inc        dti              fico       days.with.cr.line
 Min.   : 7.548   Min.   : 0.000   Min.   :612.0   Min.   :  179
 1st Qu.:10.558   1st Qu.: 7.213   1st Qu.:682.0   1st Qu.: 2820
 Median :10.928   Median :12.665   Median :707.0   Median : 4140
 Mean   :10.932   Mean   :12.607   Mean   :710.8   Mean   : 4562
 3rd Qu.:11.290   3rd Qu.:17.950   3rd Qu.:737.0   3rd Qu.: 5730
 Max.   :14.528   Max.   :29.960   Max.   :827.0   Max.   :17640
 NA's   :4                                         NA's   :29
   revol.bal         revol.util     inq.last.6mths    delinq.2yrs
 Min.   :      0   Min.   :  0.00   Min.   : 0.000   Min.   : 0.0000
 1st Qu.:   3187   1st Qu.: 22.70   1st Qu.: 0.000   1st Qu.: 0.0000
 Median :   8596   Median : 46.40   Median : 1.000   Median : 0.0000
 Mean   :  16914   Mean   : 46.87   Mean   : 1.572   Mean   : 0.1638
 3rd Qu.:  18250   3rd Qu.: 71.00   3rd Qu.: 2.000   3rd Qu.: 0.0000
 Max.   :1207359   Max.   :119.00   Max.   :33.000   Max.   :13.0000
                   NA's   :62       NA's   :29       NA's   :29
    pub.rec       not.fully.paid
 Min.   :0.0000   Min.   :0.0000
 1st Qu.:0.0000   1st Qu.:0.0000
 Median :0.0000   Median :0.0000
 Mean   :0.0621   Mean   :0.1601
 3rd Qu.:0.0000   3rd Qu.:0.0000
 Max.   :5.0000   Max.   :1.0000
 NA's   :29
```

Detecto un error en el dataset que se puede resolver. Este se trata de que las variables credit.policy y not.fully.paid deberían ser tratadas como factores. El trato de la primera no influye tanto como la segunda ya que se usa como predictora, sin embargo, si no cambiamos la naturaleza de la variable respuesta el Random Forest actuará como una regresión y sus resultados no serán los valores absolutos 1 ó 0 sino que serán aproximaciones. De hecho, si esto pasa inadvertido, cuando construimos el modelo el algoritmo de R tiene la capacidad de darse cuenta de que la variable respuesta está considerada como entera pero con tan solo dos valores y deduce que probablemente quieres una clasificación y no una regresión. Acabará preguntándote el propio algoritmo. Quizás con más categorías R no se percate y puedes terminar incurriendo en error, por lo que hay que estar atento.

Veamos ahora la proporción de impagos:

```
1. # Proportion of loans not paid fully
2. table(loans1$not.fully.paid)
```

Tenemos:

```
1.    0    1
2. 8045 1533
```

Por lo que tenemos un porcentaje de impagos de un 16%:

$$\frac{1533}{9578} \times 100 = 16\%$$

Tenemos un claro caso de unbalanced data. Esto significa que cuando empleemos un Random Forest tendremos un sesgo a favor de la clase mayoritaria, esto es, la de los buenos pagadores. De hecho, no me percaté, lo implementé, y me salió lo siguiente:

```
Call:
 randomForest(formula = not.fully.paid ~ ., data = loans_train)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 3

        OOB estimate of  error rate: 16.18%
Confusion matrix:
     0  1 class.error
0 5563 24 0.004295686
1 1054 20 0.981378026
```

Como podemos ver tenemos un error del casi 99% en los no pagadores. Esto es peor incluso, ya que es mucho peor clasificar un mal pagador como buen pagador que viceversa. Así que no podemos tomar el dataset tal y como está. Para esto hay dos opciones:

1. Submuestreo: eliminar aleatoriamente observaciones de la clase con más instancias para equilibrar los pesos. Sin embargo, estaríamos perdiendo información potencialmente relevante que con la muestra original sí se incluye.
2. Sobremuestreo: aumentar aleatoriamente observaciones de la clase con menos instancias para equilibrar los pesos. Con esto se evita la pérdida de información, pero se corre el riesgo de sobreajustar el modelo. No obstante, suele dar mejor resultados.

De cualquier manera, lo correcto sería realizar validaciones cruzadas y comparar ambos posibles modelos. Obviaré esto último, y realizaré directamente un sobremuestreo.

Veamos cuántos valores NA hay por cada atributo:

```
apply(loans1,2,function(x){ sum(is.na(x))})
```

```
1.      credit.policy           purpose          int.rate
2.                  0                 0                 0
3.       installment    log.annual.inc               dti
4.                  0                 4                 0
5.              fico days.with.cr.line         revol.bal
6.                  0                29                 0
7.        revol.util    inq.last.6mths       delinq.2yrs
8.                62                29                29
9.           pub.rec    not.fully.paid
```

```
10.                29                   0
```

Y veamos cuántas filas se ven afectadas:

```
1. > missing <- subset(loans1, is.na(log.annual.inc) | is.na(days.w
   ith.cr.line) | is.na(revol.util) | is.na(inq.last.6mths) | is.na
   (delinq.2yrs) | is.na(pub.rec))
2. > nrow(missing)
3. [1] 62
4. > table(missing$not.fully.paid)
5.
6.  0  1
7. 50 12
```

Tenemos 62 filas afectadas de las cuales 50 resultaron ser con devolución de préstamo y 12 sin devolución de préstamo. Teniendo en cuenta el número de observaciones con el que tratamos eliminar estas pocas filas con valores nulos no debería afectar a los resultados de un modelo predictivo ya que relativamente son muy pocas. Una solución posible es predecir esos resultados con el resto de variables independientes de las cuales sí que disponemos de toda la información. Sin embargo, como ya hemos dicho, al ser tan pocas filas las implicadas, optaré por excluirlas del entrenamiento del modelo.

```
1. > loans2 <- na.omit(loans1)
2. > apply(loans2,2,function(x){ sum(is.na(x))})
3.     credit.policy           purpose          int.rate
4.                 0                 0                 0
5.       installment    log.annual.inc               dti
6.                 0                 0                 0
7.              fico days.with.cr.line         revol.bal
8.                 0                 0                 0
9.        revol.util    inq.last.6mths       delinq.2yrs
10.                0                 0                 0
11.          pub.rec    not.fully.paid
12.                0                 0

1. > loans2$not.fully.paid <- as.factor(loans2$not.fully.paid)
2. > loans2$credit.policy <- as.factor(loans2$credit.policy)
```

Ahora apliquemos un sobremuestreo aleatorio:

```
1. loans <- loans2
2.
3. Y<-which(loans2$not.fully.paid==1)
4.
5. loans_unbalanced <- loans
6.
7.       Y1 = sample(Y)
8.       Y2 = sample(Y)
9.       Y3 = sample(Y)
10.      Y4 = sample(Y)
11.
12.      loans_balanced <- rbind(loans,loans[Y1,],loans[Y2,],
   loans[Y3,], loans[Y4,])
```

```
> table(loans_balanced$not.fully.paid)

   0    1
7995 7605
```

Particionemos el conjunto aleatoriamente en un conjunto de entrenamiento (70%) y otro que sirva de test del modelo (30%) para validarlo.

```
1. # Unbalanced_data
2. set.seed(2019)
3. train=sample(1:nrow(loans_unbalanced),round(0.7*nrow(loans_unbal
   anced)))
4. loans_unbalanced_train = loans_unbalanced[train,]
5. dim(loans_unbalanced_train)
6. loans_unbalanced_test = loans_unbalanced[-train,]
7. loans_test_aux = loans_unbalanced[-train,]
8. dim(loans_unbalanced_test)
9. nrow(loans_unbalanced_train)+nrow(loans_unbalanced_test)
10.
11.
12.      # Balanced_data
13.      set.seed(2019)
14.      train=sample(1:nrow(loans_balanced),round(0.7*nrow(loans_ba
   lanced)))
15.      loans_balanced_train = loans_balanced[train,]
16.      dim(loans_balanced_train)
17.      loans_balanced_test = loans_balanced[-train,]
18.      loans_test_aux = loans_balanced[-train,]
19.      dim(loans_balanced_test)
20.      nrow(loans_balanced_train)+nrow(loans_balanced_test)
```

B. Es conveniente, a modo orientativo, realizar un estudio previo visual para detectar posibles candidatos a mejores predictores y otras características que definan el modelo y cuyas conclusiones se han de contrastar posteriormente con las del modelo. Así, tendremos un poco más de certeza de que el modelo funciona bien o, como sucede en algunos casos, sorprendernos con predictores que a priori no parecieron de tanta relevancia y que, si todo se ha hecho correctamente, han resultado inadvertidos. De todas formas, este apartado lo dejo pendiente por el momento ya que para los objetivos de este documento no resulta estrictamente necesario.

C. CONSTRUCCIÓN DEL MODELO:

```
1. set.seed(2019)
2. library(randomForest)
3. Modelo1 <- randomForest(
4.   formula = not.fully.paid ~ .,
5.   data = loans_balanced_train
6. )
```

```
> Modelo1

Call:
 randomForest(formula = not.fully.paid ~ ., data = loans_balanced_train
)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 3

        OOB estimate of  error rate: 3.19%
Confusion matrix:
     0    1 class.error
0 5303  313 0.055733618
1   35 5269 0.006598793
```

Comprobemos el accuracy con los conjuntos test, tanto con el original como el balanceado:

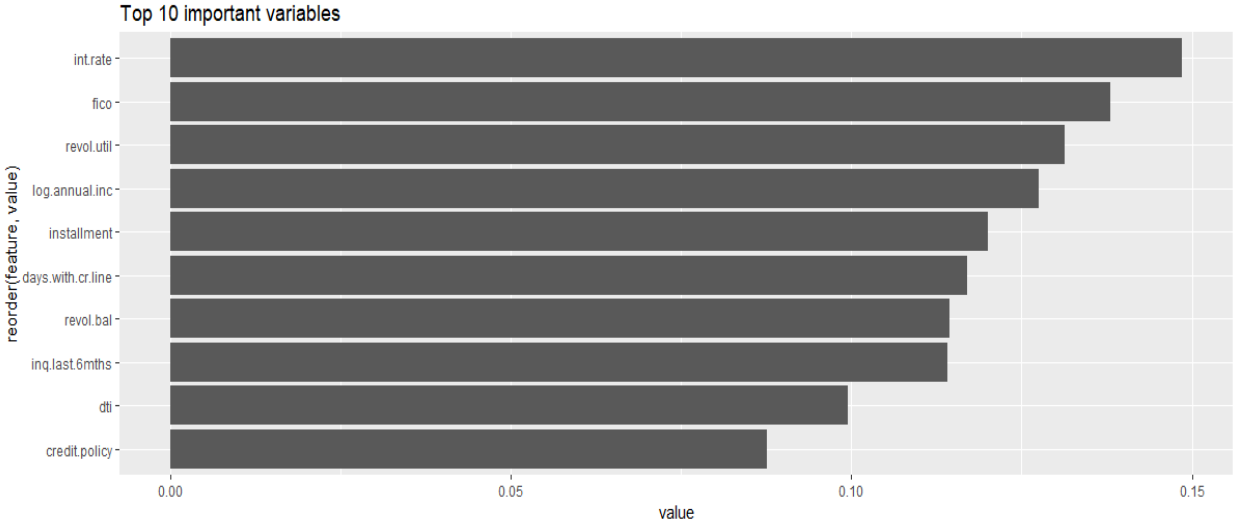
```
1. Test1 <- predict(Modelo1, loans_unbalanced_test, type = "class")
2. Test2 <- predict(Modelo1, loans_balanced_test, type = "class")
3.
4. accuracy_unbalanced_train <- mean(Test1 == loans_unbalanced_test
   $not.fully.paid)
5. accuracy_balanced_train <- mean(Test2 == loans_balanced_test$not
   .fully.paid)
```

```
> data.frame(accuracy_unbalanced_train,accuracy_balanced_train)
  accuracy_unbalanced_train accuracy_balanced_train
1                 0.9810858                 0.966453
```

Tenemos un buen accuracy en ambos casos utilizando el mismo modelo.

Si queremos ahora ver la importancia de las variables:

```
1. library(ranger)
2. Modelo1 <- ranger(
3.   formula = not.fully.paid ~ .,
4.   data = loans_balanced_train,
5.   num.trees = 500,
6.   importance = 'permutation'
7. )
8.
9. library(ggplot2)
10.      library(dplyr)
11.      list_features=data.frame("feature"= names(Modelo1$variable.
   importance),
12.                               "value"=unname(Modelo1$variable.im
   portance))
13.      list_features %>%
14.        dplyr::arrange(desc(value)) %>%
15.        dplyr::top_n(10) %>%
16.        ggplot(aes(reorder(feature, value), value)) +
17.        geom_col() +
18.        coord_flip() +
19.        ggtitle("Top 10 important variables")
```



Esta es una de las limitaciones que creo que presentará Vertica. Como poco, tendrá las limitaciones en cuanto a visualizaciones.

6.3 EJEMPLO EN VERTICA 9.2: RIESGO CREDITICIO

Partimos de los datos balanceados del documento (LOANS_BALANCED). Tomamos una muestra aleatoria para el algoritmo de Random Forest y el resto la utilizamos como muestra de test. Una vez la creamos la exportamos a csv y con PDI la metemos en la base de datos para Vertica, ya que de lo contrario no sé cómo en Vertica podemos coger una muestra aleatoria. Además, así tenemos consistencia con el modelo, pues si generamos la muestra varias veces, obtendremos modelos distintos.

Una vez lo tenemos en csv, importado a Vertica y a R, vamos a construir los modelos de Random Forest para la clasificación.

Primero en R: Es muy importante poner nuestra variable respuesta como factor, para que el algoritmo Random Forest se detecte como clasificación y no regresión.

```
loans_balanced_train$not.fully.paid <- factor(loans_balanced_train$not.fully.paid)
loans_balanced_test$not.fully.paid <- factor(loans_balanced_test$not.fully.paid)
```

Una vez tenemos esto, creamos el modelo. He creado dos modelos para comparar la eficiencia del algoritmo según el número de árboles escogido:

```
library(randomForest)
Modelo200 <- randomForest(
  formula = not.fully.paid~.,
  data = loans_balanced_train,
  ntree = 200
)

Modelo500 <- randomForest(
  formula = not.fully.paid~.,
  data = loans_balanced_train
)
```

De primeras podemos comparar el tiempo de entrenamiento del algoritmo:

```
TimeRF200 <- system.time(
  Modelo200 <- randomForest(
    formula = not.fully.paid~.,
    data = loans_balanced_train,
    ntree = 200
  )
)

TimeRF500 <- system.time(
  Modelo500 <- randomForest(
    formula = not.fully.paid~.,
    data = loans_balanced_train
  )
)
```

```
> TimeRF200
   user  system elapsed
  10.51    0.14   11.11
> TimeRF500
   user  system elapsed
  24.78    0.34   25.62
```

Una vez hecho esto, vamos a ver la efectividad del modelo en nuestras muestras test:

```
predTest <- predict(Modelo200, loans_balanced_test, type= "class")
summary(predTest)
mean(predTest == loans_balanced_test$not.fully.paid)
table(predTest, loans_balanced_test$not.fully.paid)
```

```
> predTest <- predict(Modelo200, loans_balanced_test, type= "class")
> mean(predTest == loans_balanced_test$not.fully.paid)
[1] 0.9662393
> table(predTest, loans_balanced_test$not.fully.paid)

predTest    0    1
       0 2227   30
       1  128 2295
```

Mientras que para el de 500 árboles:

```
> predTest500 <- predict(Modelo500, loans_balanced_test, type= "class")
> mean(predTest500 == loans_balanced_test$not.fully.paid)
[1] 0.9643162
> table(predTest500, loans_balanced_test$not.fully.paid)

predTest500    0    1
          0 2218   30
          1  137 2295
```

Destacar que se obtiene precisión similar, aunque un poco más baja en el algoritmo de 200 árboles. *Posiblemente se deba a que, a mayor número de árboles, más se tiende a hacer overfit en el modelo. De todas maneras, es solo un ejemplo con una muestra concreta, lo usual es tomar varias muestras del 70% y comparar.*

La tabla nos da el número de coincidencias de la predicción de clasificación con el valor real de la muestra test.

Vamos a Vertica:

```
SELECT RF_CLASSIFIER('RF_LOANS', 'LOANS_TRAIN', 'not.fully.paid', '*'
USING PARAMETERS exclude_columns = 'not.fully.paid', ntree=200);
```

El modelo con 200 árboles tiene un tiempo de ejecución de: 1 row(s) fetched - 8.516s, que comparado a R es ligeramente inferior.

El modelo de 500 árboles tiene un tiempo de ejecución de: 1 row(s) fetched - 24.245s, que, aunque inferior al de R, la diferencia es menor que en el de 200.

Ahora aplicamos el modelo a la muestra test:

```
CREATE TABLE PREDICT_LOANS_TEST as
(SELECT PREDICT_RF_CLASSIFIER(* USING PARAMETERS model_name = 'RF_LOANS') as pred
FROM LOANS_TEST_2);

SELECT pred, COUNT(pred)
FROM PREDICT_LOANS_TEST
GROUP BY pred;
```

(LOANS_TEST_2 es la muestra test sin la columna 'not.fully.paid' por comodidad)

La consulta nos proporciona este resultado:

	ABC pred	123 COUNT
1	0	3.049
2	1	1.631

Para el modelo de 500 árboles, tenemos una sentencia análoga.

```
DROP TABLE PREDICT_LOANS_TEST500;
CREATE TABLE PREDICT_LOANS_TEST500 as
(SELECT PREDICT_RF_CLASSIFIER(* USING PARAMETERS model_name = 'RF_LOANS500') as pred
FROM LOANS_TEST_2);

SELECT pred, COUNT(pred)
FROM PREDICT_LOANS_TEST500
GROUP BY pred;
```

La consulta nos da:

	ABC pred	123 COUNT
1	0	3.050
2	1	1.630

Resultados muy similares. Sin embargo, observamos lo siguiente:

```
SELECT "not.fully.paid", COUNT("not.fully.paid")
FROM LOANS_TEST
GROUP BY "not.fully.paid";
```

	123 not.fully.paid	123 COUNT
1	1	2.325
2	0	2.355

Es decir, la precisión del algoritmo es muchísimo menor en Vertica y en R. No sé si se deberá a mi implementación del algoritmo, o quizá se deba a la potencia de Vertica o R.

6.4 EJEMPLO RANDOM FOREST EN PYTHON

Vertica-ML-Python:

```
### RF CLASSIFIER ###
from vertica_ml_python.learn.ensemble import RandomForestClassifier
drop_model(name = "machine_learning.RFpy2", cursor = cur)
RFtest = vDataframe("machine_learning.LOANS_TEST", cursor = cur)
modelRF = RandomForestClassifier(name = "machine_learning.RFpy2", cursor = cur, n_estimators = 200)
#modelRF = load_model(name = "machine_learning.RFpy", cursor = cur)
modelRF.fit(input_relation = "machine_learning.LOANS_TRAIN",
            X = ["credit.policy", "purpose", "int.rate", "installment", "log.annual.inc", "dti", "fico",
            "days.with.cr.line", "revol.bal", "revol.util", "inq.last.6mths", "deling.2yrs", "pub.rec"],
            y = "not.fully.paid")
modelRF.add_to_vdf(RFtest)
print(RFtest)
```

Destacar que no sé muy bien cómo aplicar el modelo a mi muestra de test en la librería. Este sería un ejemplo aproximado de cómo pienso que debería funcionar. Sin embargo, el código crea y entrena el modelo y lo almacena en la base de datos, podemos acceder desde allí para ver el resultado.

Python:

```
import pandas as pd
from sklearn.ensemble import RandomForestClassifier

tableExampleTraining = pd.read_csv(r'C:\Users\Kevin Moreno\Documents\verticaML\loans_balanced_train.csv')
tableExampleTest = pd.read_csv(r'C:\Users\Kevin Moreno\Documents\verticaML\loans_balanced_test.csv')
predictorsTraining = tableExampleTraining.loc[:,'credit.policy':'pub.rec']
responseTraining = tableExampleTraining.loc[:,'not.fully.paid']
predictorsTest = tableExampleTest.loc[:,'credit.policy':'pub.rec']
responseTest = tableExampleTest.loc[:,'not.fully.paid']

predictorsTrainPrueba = pd.get_dummies(predictorsTraining)
predictorsTestPrueba = pd.get_dummies(predictorsTest)
#print(predictorsPrueba)

rf200 = RandomForestClassifier(n_estimators = 200)
rf200.fit(predictorsTrainPrueba,responseTraining)
pred = rf200.predict(predictorsTestPrueba)
confusionMatrix = pd.crosstab(responseTest, pred, rownames=['valor real'], colnames=['valor predicción'])
print(confusionMatrix)
```

Scikit no permite el uso de variables categóricas en los predictores, de ahí que necesitemos utilizar get_dummies.

6.5 CONTRASTE EJEMPLO RANDOM FOREST

PRUEBA RANDOM FOREST

Python:

```
valor predicción     0     1
valor real
0                 2232   123
1                   30  2295
```

R:

```
predTest    0    1
       0 2222   30
       1  133 2295
```

Vertica:

	123 actual_class	123 predicted_0	123 predicted_1
1	0	1.823	532
2	1	1.226	1.099

Vertica ML Python:

	123 actual_class	123 predicted_0	123 predicted_1
1	0	1.835	520
2	1	1.246	1.079

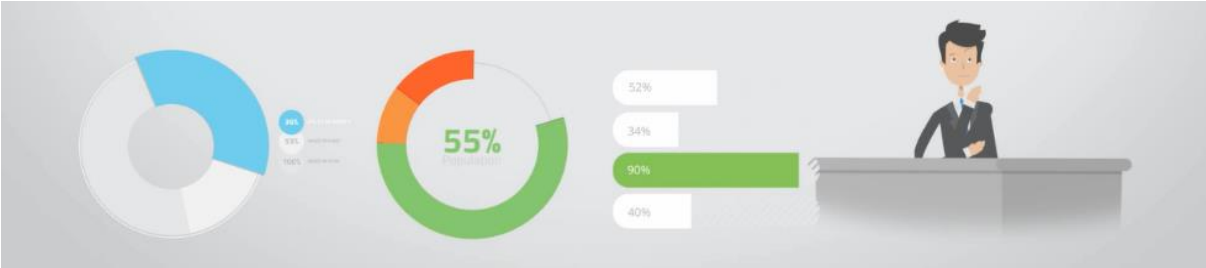
Vemos que los resultados en R y Python son similares y con poco margen de error, mientras que en Vertica los resultados tienen muchísimas clasificaciones erróneas. En resumen, para una clasificación mediante RandomForests sería mejor utilizar Python o R.

7 DESCRIPCIÓN STRATEBI

En Stratebi ofrecemos **gran cantidad de soluciones analíticas** por una compañía de **rápido crecimiento,** innovando en las áreas tecnológicas de mayor desarrollo en la actualidad: **Business Intelligence, Big Data y Social Intelligence**, muchas de ellas, basadas en soluciones **Open Source**.

Además, somos **Partners Certificados en Microsoft PowerBI y Vertica**, con gran número de proyectos con ámbas tecnologías

Contamos con oficinas en diversas ciudades de España: Madrid, Barcelona, Sevilla, Alicante y Latam



Desarrollamos nuevas soluciones analíticas basadas en Open Source, para la generación de Cuadros de Mando en tiempo real, con tecnologías IoT para SmartCities, machine learning, etc...

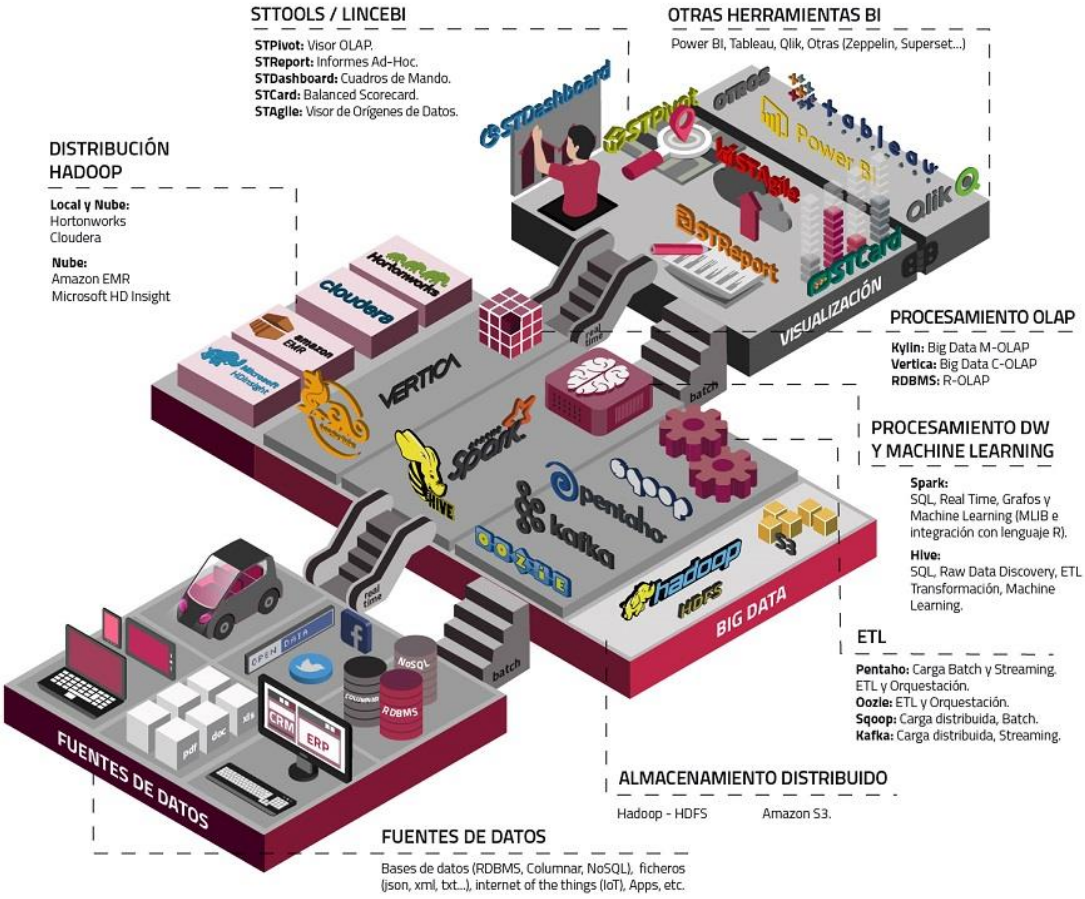
LinceBI
The best open source BI based solution

8 TECNOLOGÍAS

Recientemente, hemos sido nombrados Partners Certificados de Vertica, Talend, Microsoft, Snowflake, Kylligence, Pentaho, etc…

9 DESCRIPCIÓN DE LAS SOLUCIONES

Stratebi son los creadores de la solución LinceBI.com sobre la que podrás desarrollar e innovar.



10 INFORMACIÓN SOBRE STRATEBI

Stratebi es una empresa española, con sede en Madrid y oficinas en Barcelona, Alicante y Sevilla, creada por un grupo de profesionales con amplia experiencia en sistemas de información, soluciones tecnológicas y procesos relacionados con soluciones de Open Source y de inteligencia de Negocio.

Esta experiencia, adquirida durante la participación en proyectos estratégicos en compañías de reconocido prestigio a nivel internacional, se ha puesto a disposición de nuestros clientes a través de Stratebi.

Stratebi es la única empresa española que ha estado presente todos los Pentaho Developers celebrados en Europa (Mainz-Alemania, Barcelona, Lisboa, Roma, Amsterdam, Sintra, Amberes (2) Londres…), habiendo organizado el de Barcelona.

En Stratebi nos planteamos como **objetivo** dotar a las compañías e instituciones, de herramientas escalables y adaptadas a sus necesidades, que conformen una estrategia Business Intelligence capaz de rentabilizar la información disponible. Para ello, nos basamos en el desarrollo de soluciones de Inteligencia de Negocio, mediante tecnología Open Source.

Stratebi **son** profesores y responsables de proyectos **del Master en Business Intelligence de la Universidad UOC, UCAM, EOI…**

Los profesionales de Stratebi son los creadores y autores del primer weblog en español sobre el mundo del Business Intelligence, Data Warehouse, CRM, Dashboards, Scorecard y Open Source.

Stratebi es partner de las principales soluciones Analytics: Microsoft PowerBI, Talend, Pentaho, Vertica, Snowflake, Kyligence, Cloudera…

Todo Bi, se ha convertido en una referencia para el conocimiento y divulgación del Business Intelligence en español.

11 REFERENCIAS STRATEBI

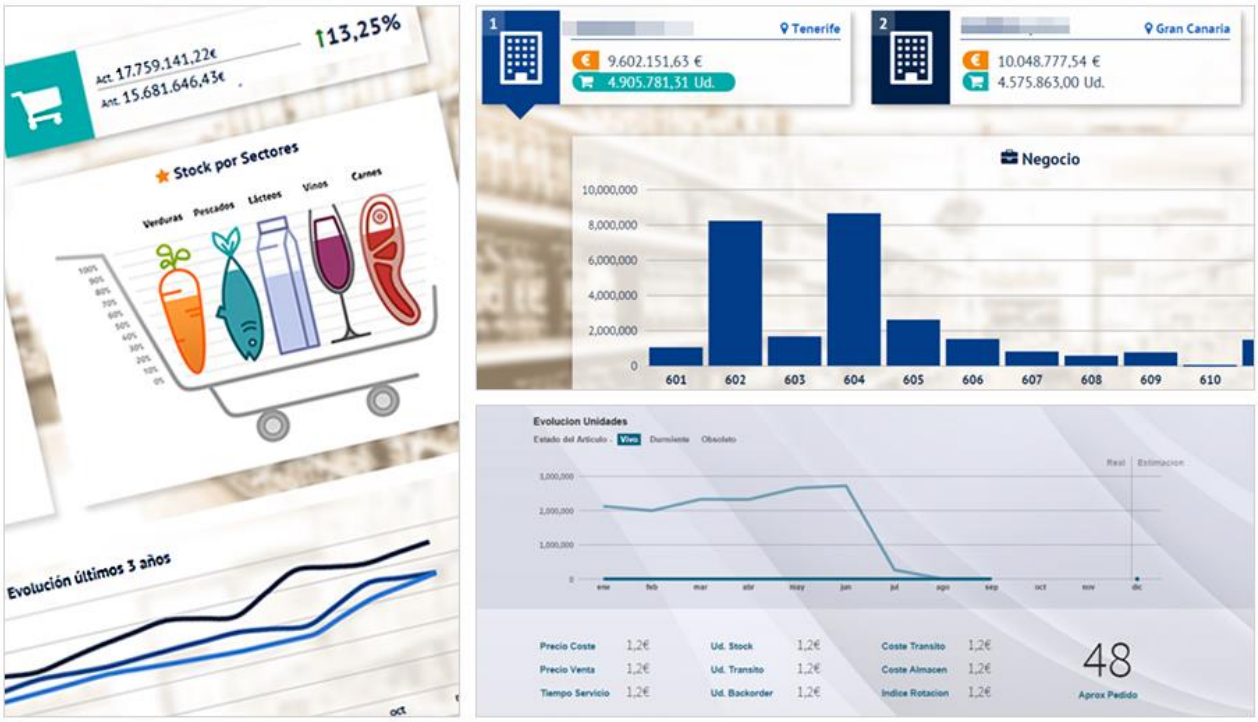
Trabajamos en los principales sectores y con algunas de las compañías y organizaciones más importantes de España.

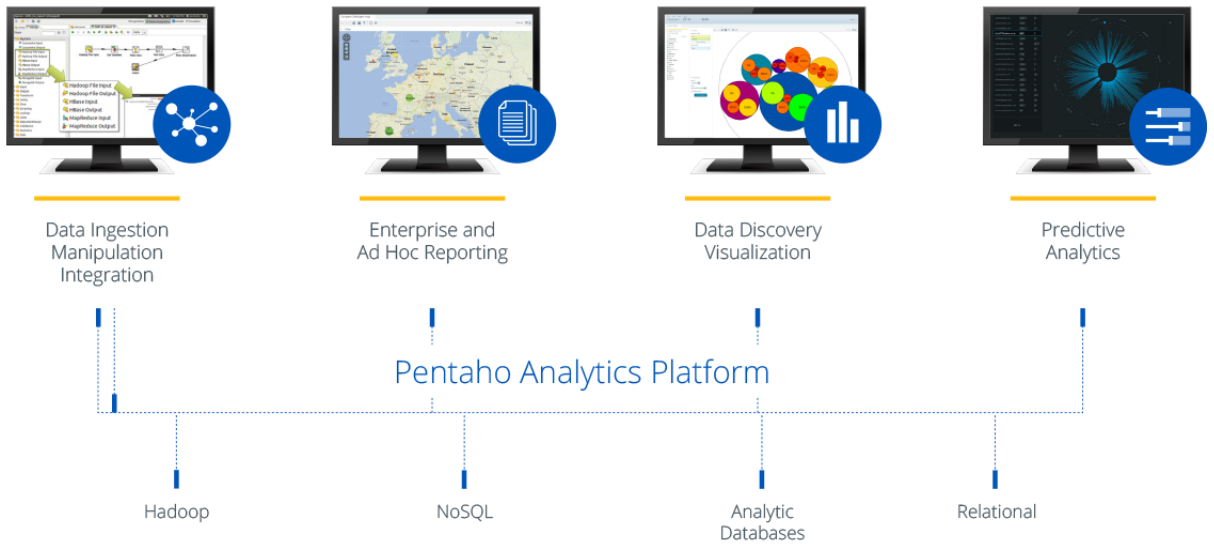
SECTOR PRIVADO

SECTOR PÚBLICO

12 EJEMPLOS DE DESARROLLOS ANALYTICS

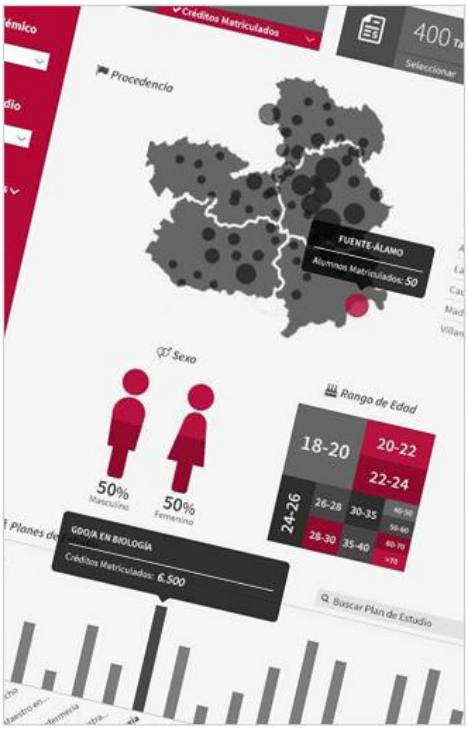
A continuación se presentan **ejemplos de algunos screenshots** de cuadros de mando diseñados por Stratebi, con el fin de dar a conocer lo que se puede llegar a obtener, así como Demos Online en la web de Stratebi:



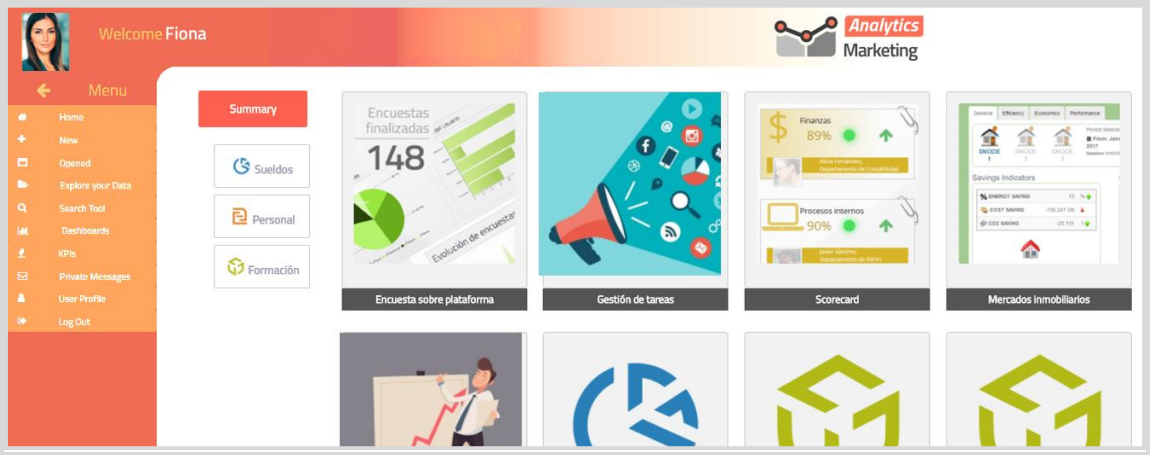


facebook DASHBOARD COMENTARIOS
Sexo
Sentimiento
Localización
Tipo de recurso
Evolución Comentarios por días

smart MADRID
Último Actualización
18:24
TOP 5 PUNTOS DE MEDIDA
PROXIMIDAD DE PUNTOS DE MEDIDA
EVOLUCIÓN INTENSIDAD DE TRÁFICO


Créditos Matriculados
Procedencia
FUENTE-ÁLAMO
Alumnos Matriculados: 50
Sexo
50% Masculino
50% Femenino
Rango de Edad
18-20
20-22
22-24
24-26
26-28
28-30
30-35
35-40
GDO/A EN BIOLOGÍA
Créditos Matriculados: 6.500
Buscar Plan de Estudio

181 Inscritos en Cursos
8 Desinscritos en Cursos
181 Inicios de Cursos
110 Inscritos en Eventos Virtuales
63 Inscritos en Debates
181 Inscritos en Talleres
USUARIOS ACTIVOS POR PAÍS
INSCRITOS EN CURSOS POR COMUNIDAD EDU
JORNADA COMPLETA
IDIOMAS
TIPO DE CENTRO
TIPO DE AUSENCIA
Nº de profesores por sexo
36%
64%
Nº total de profesores
1.587
Media de edad
43 años
Búsqueda


Welcome Fiona
Analytics Marketing
Menu
Home
New
Opened
Explore your Data
Search Tool
Dashboards
KPIs
Private Messages
User Profile
Log Out
Summary
Sueldos
Personal
Formación
Encuestas finalizadas
148
Encuesta sobre plataforma
Gestión de tareas
Finanzas 89%
Procesos internos 90%
Scorecard
Mercados inmobiliarios