Emimon

July 2, 2017

1 Notebook für die Smart-Data-Veranstaltung

Dieses Notebook ist die Vorlage für die Aufgabe für die Smart-Data-Blockveranstaltung im Sommersemester 2017.

Die Aufgabe besteht aus den folgenden Schritten:

- 1) Laden der Daten als Spark Dataframe
- 2) Aufbereiten der Daten zu einem "Unique Customer View"
- 3) Deskriptive Analysen der Daten, um interessante Variablen für ein Predictive Model zu finden
- 4) Schätzen eines Predictive Model, um die besten Kunden für eine Facebook-Aktion zu finden
- 5) Interpretieren der Koeefizienten
- 6) Evaluieren des Trainings- und des Testdatensatzes hinsichtlich der ROC-Kurve

2 Hinweise zur Durchführung und zur Benotung

- Das Notebook kann gern als Vorlage für Eure eigene Aufgabe genutzt werden. Beachtet bitte, dass Eure Daten sich von denen hier unterscheiden, so dass Ihr bitte die für Ihnen zur Verfügung gestellten Daten nutzt.Die Daten, die hier importiert werden, sind auch auf der VM nicht vorhanden, so dass der Befehl zum Einlesen failen wird.
- Da dieses Notebook als Vorlage gegeben wird, ist das bloSSe Kopieren und Ausführen des Notebooks ohne weitere Untersuchungen vorzunehmen, natürlich nicht unbedingt eine Basis für eine gute Note :). Insbesondere die Kommentare im Text zu den Daten und Ergebnissen sollten in Ihrem Notebook nicht mehr meine, sondern Eure sein!
- Es geht vielmehr darum, selber in die Daten reinzuschauen und evtl. mehr oder andere Daten für die Analyse heranzuziehen sowie generell zu versuchen, aus den Daten zu lernen. Dabei ist es vor allen Dingen wichtig, dass der Analyseprozess nachvollziehbar ist und die Gedankengänge dabei in der Präsentation erläutert werden können.

- In jedem Abschnitt gebe ich Erweiterungshinweise, die Vorschläge sind, was man hier an Eigenleistung einbringen könnte. Um eine gute Note zu bekommen, ist es nicht Pflicht alle Erweiterungshinweise zu befolgen, es geht mir eher um ein durchdachtes Vorgehen und den Versuch, ein möglichst gutes Predictive Model zu bauen. Wer allerdings keine Erweiterungshinweise beachtet, kann keine gute Note erwarten.
- Als Abschlussleistung sind das das fertige Notebook als .html-Datei und die Präsentation abzugeben. Die Präsentation kann auch anhand des Notebooks erfolgen, in diesem Fall ist dann kein separates PDF mit der Präsentation notwendig. Bitte nutzt die Möglichkeit, über die Markdown-Felder (so wie dieses hier) eigene Texte und Interpretationen zu den jeweiligen Schritten zu erstellen. Je mehr Ihr zu den Ergebnissen ihrer Analysen in den Abschnitten schreibt, um so eher kann ich sehen, wie viel Arbeit Ihr in das Notebook gesteckt habt.

2.1 Vorbereitungen

Als erstes wird das Notebook vorbereitet. Dazu müssen die notwendigen Pakete importiert werden: - pyspark ist die Schnittstelle zu Apache Spark - pandas ist ein Python-Paket, mit dem man Daten einfach bearbeiten kann

```
In [1]: import pyspark
        import pandas
```

2.1.1 Erstellen der Spark-Kontexte

- Ein SparkContext registriert die Schnittstelle zu Apache Spark
- Ein SQLContext registriert die Schnittstelle zur SQL-API von Apache Spark. Hiermit lassen sich SQL-Queries auf Spark-Daten ausführen.

```
In [2]: sc = pyspark.SparkContext('local[*]')
        sq = pyspark.sql.SQLContext(sc)
```

3 Schritt 1: Laden der Daten

3.1 1.1 Einlesen der Daten

Die Daten bestehen aus zwei Dateien:

- customers.csv
- transactions.csv

Die customers-Datei enthält die Stammdaten der Kunden, wie z.B. das Alter oder das Geschlecht. Die transactions-Datei enthält alle Transaktionen der Kunden.

```
In [3]: customers = sq.read.load("../aufgabe/daten/cus_model_until_170228.parquet/")
```

```
In [4]: orders = sq.read.load("../aufgabe/daten/tt_model_until_170228.parquet/")
```

3.2 1.2 Erster Check der Daten

```
In [5]: customers.count()
```

```
Out[5]: 88032
```

```
In [6]: customers.show(10)
```

```
+-----------+--------------+---------------+------------+-------+--------+
|Customer_id|Customer_since|Customer_gender|Customer_age|Zipcode|   Phone|
+-----------+--------------+---------------+------------+-------+--------+
|   25461829|    2011-08-09|           male|  1974-02-27|   3752|(06234) |
|   25461830|    2015-05-26|           male|  1994-12-11|  35862|+49-831-|
|   25461831|    2014-01-25|         female|  1990-06-19|  42761|(0155) 2|
|   25461832|    2016-07-22|           male|  1974-05-04|  37164|(0376) 5|
|   25461834|    2010-01-28|         female|  1960-04-14|  48335|+49-737-|
|   25461835|    2014-08-20|           male|  1977-03-15|   1552|(0538) 3|
|   25461837|    2014-05-31|           male|  1980-02-20|   6404|(00846) |
|   25461838|    2011-10-29|           male|  1964-08-30|  64651|(0012) 1|
|   25461839|    2016-10-02|         female|  1983-02-08|  13000|(08784) |
|   25461840|    2011-10-20|         female|  1948-02-09|  52181|+49-8511|
+-----------+--------------+---------------+------------+-------+--------+
only showing top 10 rows
```

```
In [7]: customers.printSchema()
```

```
root
 |-- Customer_id: long (nullable = true)
 |-- Customer_since: string (nullable = true)
 |-- Customer_gender: string (nullable = true)
 |-- Customer_age: string (nullable = true)
 |-- Zipcode: long (nullable = true)
 |-- Phone: string (nullable = true)
```

Tipp Anstelle von show() kann man sich die Daten auch mit limit(xx).toPandas() etwas schöner in Pandas anschauen. Das limit(xx)-Statement beschränkt die Anzahl der ausgegebenen Zeilen auf xx.

```
In [8]: customers.limit(10).toPandas()
```

```
Out[8]:    Customer_id Customer_since Customer_gender Customer_age  Zipcode     Phone
        0     25461829     2011-08-09            male   1974-02-27     3752  (06234)
        1     25461830     2015-05-26            male   1994-12-11    35862  +49-831-
        2     25461831     2014-01-25          female   1990-06-19    42761  (0155) 2
        3     25461832     2016-07-22            male   1974-05-04    37164  (0376) 5
```

```
        4     25461834     2010-01-28          female   1960-04-14    48335  +49-737-
        5     25461835     2014-08-20            male   1977-03-15     1552  (0538) 3
        6     25461837     2014-05-31            male   1980-02-20     6404  (00846)
        7     25461838     2011-10-29            male   1964-08-30    64651  (0012) 1
        8     25461839     2016-10-02          female   1983-02-08    13000  (08784)
        9     25461840     2011-10-20          female   1948-02-09    52181  +49-8511
```

```
In [9]: orders.count()
```

```
Out[9]: 444945
```

```
In [10]: customers.crosstab('Customer_age', 'Customer_gender').show()
```

```
+--------------------------+------+----+
|Customer_age_Customer_gender|female|male|
+--------------------------+------+----+
|                  1960-07-23|     2|   1|
|                  1966-05-03|     0|   3|
|                  1985-04-02|     6|   3|
|                  1975-08-01|     3|   2|
|                  1989-05-09|     4|   4|
|                  1950-03-30|     1|   0|
|                  1990-04-10|     2|   1|
|                  1978-04-05|     5|   2|
|                  1952-01-19|     1|   0|
|                  1987-07-26|     1|   4|
|                  1974-08-17|     3|   2|
|                  1993-09-21|     2|   1|
|                  1993-06-10|     0|   1|
|                  1951-06-27|     1|   0|
|                  1982-05-16|     1|   4|
|                  1988-02-05|     2|   3|
|                  1962-03-20|     4|   0|
|                  1963-10-28|     3|   0|
|                  1952-05-03|     1|   3|
|                  1955-06-06|     0|   3|
+--------------------------+------+----+
only showing top 20 rows
```

```
In [11]: orders.printSchema()
```

```
root
 |-- Customer_id: long (nullable = true)
 |-- Invoice_id: string (nullable = true)
 |-- Item_number: string (nullable = true)
 |-- Date: string (nullable = true)
 |-- Quantity: long (nullable = true)
```

```
 |-- Price: double (nullable = true)
 |-- Order_type: string (nullable = true)
 |-- Product_groupid: string (nullable = true)
```

```
In [12]: orders.limit(10).toPandas()
```

Out[12]:

	Customer_id	Invoice_id	Item_number	Date	Quantity	Price	Order_type	Product_groupid
0	25402226	fd47c076	1	2010-11-21	1	45.05	Telephone	Woman trend
1	25402227	650fe715	1	2010-01-28	1	1.00	Internet	Kids Style
2	25402227	650fe715	2	2010-01-28	1	39.38	Internet	Woman trend
3	25402227	650fe715	r_2	2010-02-02	1	-39.38	cancellation	Woman trend
4	25402228	90b0e13a	1	2010-06-21	1	66.93	Internet	Men casual
5	25402228	90b0e13a	2	2010-06-21	1	18.34	Internet	Woman trend
6	25402228	90b0e13a	3	2010-06-21	1	16.41	Internet	Men trend
7	25402230	86e62f78	1	2010-09-28	1	24.29	Telephone	Woman trend
8	25402230	86e62f78	2	2010-09-28	1	43.55	Telephone	Woman casual
9	25402231	027f7fba	1	2013-12-29	1	40.67	Internet	Kids Style

```
In [13]: orders.describe('Quantity','Price').show()
```

```
+-------+-------------------+------------------+
|summary|           Quantity|             Price|
+-------+-------------------+------------------+
|  count|             444945|            444945|
|   mean|  1.02388609828181|18.350279158098047|
| stddev|0.16132593852848948| 30.88087939236928|
|    min|                  1|           -112.33|
|    max|                  3|            127.07|
+-------+-------------------+------------------+
```

3.2.1 *Erweiterungshinweis 1*

Um die Rohdaten besser zu verstehen, könnt Ihr hier gern weitere Ansichten erstellen, zum Beispiel das Gruppieren der Orders nach Produktgruppen, nach Order-Typen, nach Zeiten (Jahre,

Saison). Im folgenden Notebook findet Ihr viele der Techniken, die dafür notwendig sind eläutert und mit einem Beispiel. Wendet diese auch auf die Rohdaten hier an, insbesondere um nach dem ersten Modellversuch zu lernen, was in den Daten noch drin stecken könnte.

Dieses Notebook erläuter sehr gut die Konzepte von PySpark-Dataframes. Dataframes sind die Datensätze, die wir hier analysieren. http://nbviewer.jupyter.org/github/jkthompson/pyspark-pictures/blob/master/pyspark-pictures-dataframes.ipynb. Dort sind viel mehr Konzepte erläutert als hier genutzt werden.

Wenn Sie noch tiefer in das gesamte Thema Machine Learning mit Notebooks einsteigen möchten, empfehle ich diesen Link: https://github.com/jakevdp/PythonDataScienceHandbook. Diese Inhalte fokussieren sich allerdings auf die Möglichkeiten innerhlab von Python und nich auf Apache Spark.

```
In [14]: orders.groupBy("Product_groupid").count().toPandas()
```

```
Out[14]:   Product_groupid   count
         0       Men trend   55666
         1    Woman casual  111048
         2      Kids Style   55589
         3     Woman trend   55394
         4      Men casual   57518
         5            Kids   55402
         6           Shoes   54328
```

```
In [15]: orders.groupBy("Order_type").count().toPandas()
```

```
Out[15]:      Order_type   count
         0  cancellation   89289
         1     Telephone  178297
         2      Internet  177359
```

```
In [16]: season = orders.withColumn('sellin_season',orders.Date.substr(6,2))
```

```
In [17]: season.limit(10).toPandas()
```

```
Out[17]:    Customer_id Invoice_id Item_number        Date  Quantity  Price  \
         0     25402226   fd47c076           1  2010-11-21         1  45.05
         1     25402227   650fe715           1  2010-01-28         1   1.00
         2     25402227   650fe715           2  2010-01-28         1  39.38
         3     25402227   650fe715         r_2  2010-02-02         1 -39.38
         4     25402228   90b0e13a           1  2010-06-21         1  66.93
         5     25402228   90b0e13a           2  2010-06-21         1  18.34
         6     25402228   90b0e13a           3  2010-06-21         1  16.41
         7     25402230   86e62f78           1  2010-09-28         1  24.29
         8     25402230   86e62f78           2  2010-09-28         1  43.55
         9     25402231   027f7fba           1  2013-12-29         1  40.67

              Order_type Product_groupid sellin_season
         0     Telephone     Woman trend            11
         1      Internet      Kids Style            01
```

```
2      Internet     Woman trend            01
3  cancellation     Woman trend            02
4      Internet      Men casual            06
5      Internet     Woman trend            06
6      Internet       Men trend            06
7     Telephone     Woman trend            09
8     Telephone    Woman casual            09
9      Internet      Kids Style            12
```

```
In [18]: season.groupBy("sellin_season").count().toPandas()
```

```
Out[18]:    sellin_season  count
          0             07  38185
          1             11  35285
          2             01  39523
          3             09  36725
          4             05  37295
          5             08  35420
          6             03  35631
          7             02  39903
          8             06  35946
          9             10  36538
          10            12  37750
          11            04  36744
```

4 Schritt 2: Erzeugen des Unique Customer View

Der unique customer view (UCV) ist die Basis für alle Analysen, die auf den Kundendaten durchgeführt werden. Diese Sicht bedeutet, dass man den Kunden zu einem bestimmten Zeitpunkt anschaut und Merkmale dieses Kunden zu diesem Zeitpunkt berechnet.

Um einen unique customer view für ein Predictive Model zu berechnen, benötigt man auSSerdem noch eine Erfolgsvariable, auf die das Modell kalibriert werden kann. Diese wird anhand des Verhaltens des Kunden NACH diesem Zeitpunkt (beobachtet für eine gewisse Zeit) berechnet.

4.1 2.1 Typische Merkmale in einem UCV

- Kundenattribute wie Alter, Adresse, Geschlecht, Länge der Kundenbeziehung
- Transaktionsattribute wie "Zeit seit dem letztem Kauf", "Umsatz in den letzten 12 Monaten", "Anzahl der Bestellungen in den letzten 24 Monaten"
- Attribute des Marketings wie "Wie viele Gutscheine hat der Kunden in den letzten 12 Monaten bekommen"

4.2 2.2 Berechnung eines UCV

Ein UCV hat immer genau eine Zeile für jeden Kunden, der zu einem Zeitpunkt betrachtet werden soll. Da z.B. die Transaktionen mehrere Zeilen für jeden Kunden haben können, müssen für das UCV Aggregationen vorgenommen werden, d.h., es müssen anhand bestimmter Rechenvorschriften Fakten zusammengefasst werden.

Damit das UCV für eine Modellschätzung nutzbar ist, müssen die Werte in dem UCV entweder numerisch oder kategorial sein. Dies bedeutet, dass z.B. ein Datumsfeld nicht in Frage kommt. Daher müssen alle Werte, die auf einem Datum basieren, in Zahlen, z.b. das Alter in Jahren konvertiert werden.

Typischerweise ist die Erzeugung des UCV die aufwändigste Phase eines Predictive Analytics-Projekts.

```
In [19]: # Schritt 0: Die Transaktionstabelle auf den 31.12.2016 "abschneiden"

         orders_valid = orders.where("Date <= '2016-12-31'")

In [20]: orders_valid.limit(10).toPandas()
```

Out[20]:

	Customer_id	Invoice_id	Item_number	Date	Quantity	Price	Order_type	Product_groupid
0	25402226	fd47c076	1	2010-11-21	1	45.05	Telephone	Woman trend
1	25402227	650fe715	1	2010-01-28	1	1.00	Internet	Kids Style
2	25402227	650fe715	2	2010-01-28	1	39.38	Internet	Woman trend
3	25402227	650fe715	r_2	2010-02-02	1	-39.38	cancellation	Woman trend
4	25402228	90b0e13a	1	2010-06-21	1	66.93	Internet	Men casual
5	25402228	90b0e13a	2	2010-06-21	1	18.34	Internet	Woman trend
6	25402228	90b0e13a	3	2010-06-21	1	16.41	Internet	Men trend
7	25402230	86e62f78	1	2010-09-28	1	24.29	Telephone	Woman trend
8	25402230	86e62f78	2	2010-09-28	1	43.55	Telephone	Woman casual
9	25402231	027f7fba	1	2013-12-29	1	40.67	Internet	Kids Style

```
In [21]: # Schritt #1: Alle Kunden, die am 31.12.2016 mindestens eine Transaktion hatten, als Ba

         custs_with_orders = orders_valid.select("customer_id").distinct()

In [22]: custs_with_orders.count()

Out[22]: 87275
```

Dies ist die Basis des UCV. Damit besteht dieser nun aus einem Datenfeld, welches eine unique Auflistung aller Kunden ist, die am 31.12.2016 mindestens bereits einmal eine Order getätigt haben.

```
In [23]: ucv = custs_with_orders
```

```
In [24]: ucv.limit(10).toPandas()
```

```
Out[24]:    customer_id
         0     25402550
         1     25402650
         2     25403321
         3     25403599
         4     25404111
         5     25404911
         6     25405236
         7     25406003
         8     25406911
         9     25407133
```

4.3 2.3 Berechnung der fixen Kunden-Attribute

Nun geht es darum, weitere Merkmale an diesen UCV heranzuspielen. Dafür starten wir zunächst mit den fixen Attributen aus der Kunden-Tabelle. Wir nehmen hier die erste Stelle der Postleitzahl und das Geschlecht.

```
In [25]: cust_fixed_attrs = customers.select(
             "Customer_id",
             "Customer_gender",
             "Zipcode",
             "Phone"
         )
```

Um die erste Stelle der Postleitzahl herauszubekommen, braucht man SQL-Funktionen, mit denen man Felder manipulieren kann. Diese sind sehr ausführlich in http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html dokumentiert.

Es gibt Funktionen, die direkt Felder manipulieren. Diese stehen einfach in der Syntax von PySpark zur Verfügung und müssen nicht extra importiert werden. Diese sind hier dokumentiert: http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html#pyspark.sql.Column

Ferner gibt es spezielle Funktionen, mit denen man komplexere SQL-Funktionen, die sich auch über mehrere Felder erstrecken können, abbilden kann. Diese müssen zunächst importiert werden. Wir laden zunächst das Paket hierfür und vergeben als Alias für dieses Paket "F", damit wir es später kurz und knapp referenzieren können. Diese Funktionen sind hier beschrieben: http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html#module-pyspark.sql.functions

```
In [26]: import pyspark.sql.functions as F
```

Die Funktion, mit der man die erste Stelle des Zipcodes herausbekommen kann, heisst 'substr'. Diese wird in dem folgenden pyspark-SQL-Ausdruck verwendet. Diese Funktion ist eine Feld-Funktion und muss daher nicht mit F. referenziert werden.

Bitte beachten, dass man in dem Ausdruck selbst immer den vollständigen Namen des Dataframes (hier 'cust_fixed_attrs') mit angeben muss, daher wird das Feld 'Zipcode' hier als 'cust_fixed_attrs.Zipcode' referenziert.

```
In [27]: cust_fixed_attrs_zip = cust_fixed_attrs.withColumn("zip1", cust_fixed_attrs.Zipcode.sub
                    ).drop("Zipcode")
```

Der "drop"-Befehl am Ende sorgt dafür, dass das Original-Feld aus dem Datensatz entfernt wird.

```
In [28]: cust_fixed_attrs_zip.limit(10).toPandas()
```

Out[28]:

	Customer_id	Customer_gender	Phone	zip1
0	25461829	male	(06234)	3
1	25461830	male	+49-831-	3
2	25461831	female	(0155) 2	4
3	25461832	male	(0376) 5	3
4	25461834	female	+49-737-	4
5	25461835	male	(0538) 3	1
6	25461837	male	(00846)	6
7	25461838	male	(0012) 1	6
8	25461839	female	(08784)	1
9	25461840	female	+49-8511	5

```
In [29]: cust_fixed_attrs_zip.select('Customer_id').count()
```

```
Out[29]: 88032
```

Diese zwei neuen Attributen verknüpfen wir jetzt mit dem bereits bestehenden UCV, welches zurzeit nur aus der Customer_id besteht. Dies geschieht über einen so genannten Join, genauer gesagt einen inner Join. Dies ist eine SQL-Operation, die für jeden Datensatz der gleichen ID in zwei Tabellen diese beiden Tabellen zusammenführt ("joined").

https://en.wikipedia.org/wiki/Join_(SQL)

Die Funktion in Pyspark heisst ebenfalls 'join'.

```
In [30]: ucv = ucv.join(cust_fixed_attrs_zip, "Customer_id", "left_outer")
```

```
In [31]: ucv.limit(10).toPandas()
```

Out[31]:

	customer_id	Customer_gender	Phone	zip1
0	25402550	female	+49-122-	7
1	25402650	female	(0101) 8	6
2	25403321	male	+49-7515	8
3	25403599	male	(0002) 4	4
4	25404111	male	(0685) 5	1
5	25404911	female	+49-123-	5
6	25405236	female	+49-706-	8
7	25406003	female	+49-110-	3
8	25406911	female	+49-300-	6
9	25407133	female	(06655)	2

Nach jedem Join muss geprüft werden, ob die Anzahl der Zeilen im UCV der ursprünglichen Anzahl entspricht. Wenn dies nicht der Fall ist, so ist der Join fehlerhaft gewesen.

Sind nach dem Join mehr Zeilen im UCV als vorher, liegt dies typischerweise daran, dass der Schlüssel, auf den der join gemacht wurde (hier 'Customer_id') nicht in beiden Tabellen ein eindeutiger Schlüssel ist, d.h., in einer oder beiden der Tabellen der Schlüssel mehrmals vorkommt.

Sind nach dem Join weniger Zeilen im UCV als vorher, liegt das daran, dass der 'inner Join' nicht für jeden Eintrag in der Basistabelle (ucv) einen Eintrag in der zu joinenden Tabelle (hier: 'cust_fixed_attrs_zip') hatte.

Dies kann sehr häufig passieren. Daher verwenden wir ab jetzt nicht mehr den 'inner Join', sondern einen so genannten 'left outer Join', der dafür sorgt, dass keine Zeilen aus der ursprünglichen Tabelle (ucv) gelöscht werden, wenn zu dem Schlüssel keine Zeilen in der zu joinenden Tabelle vorhanden sind. Statt dessen wird dafür dann der spezifische Wert NULL eingesetzt, der bedeutet, dass das jeweilige Feld undefiniert ist.

https://en.wikipedia.org/wiki/Join_(SQL)#Left_outer_join

```
In [32]: ucv.count()
```

```
Out[32]: 87275
```

Diese Zahl stimmt mit dem originalen UCV überein.

4.3.1 *Erweiterungshinweis 2*

Dies ist nur ein minimales Set (Customer_gender, zip1). Auch aus der Telefonnummer könnte man noch Variablen für den UCV erzeugen... Wegen der Schreibweise kann man nicht viel mit Telefonnummer machen.

4.4 2.4 Berechnung der zeitabhängigen Kundenattribute

Nun geht es darum, das Alter des Kunden zum Zeitpunkt des UCV zu berechnen. In den Daten liegt lediglich das Geburtsdatum vor, so dass das Alter anhand einer Datumsdifferenz berechnet werden muss.

Dafür sind zwei Schritte erforderlich: Zunächst muss das Text-Feld "Date" in ein Datum umgewandelt werden. Dann muss die Differenz dieses Datums und des Betrachtungsdatums berechnet werden.

```
In [33]: cust_bdate = customers.select(
             "Customer_id",
             "Customer_age"
         )
```

```
In [34]: cust_bdate.limit(5).toPandas()
```

```
Out[34]:    Customer_id Customer_age
         0     25461829   1974-02-27
         1     25461830   1994-12-11
         2     25461831   1990-06-19
         3     25461832   1974-05-04
         4     25461834   1960-04-14
```

Für den ersten Schritt ist die Funktion 'pyspark.sql.functions.to_date(col)' zuständig. Wir nutzen hier eine sehr angenehme Eigenschaft von PySparkSQL, nämlich die Möglichkeit des Verkettens von SQL-Anweisungen.

Die folgende Zelle nimmt folgende Änderungen vor:

- Hinzufügen des Feldes "bdate" mit dem Inhalt des "Date" als echtes Datum
- Hinzufügen eines weiteren Feldes mit dem Wert '2016-12-31' als echtes Datum (Die Funktion lit erzeugt ein Feld mit einem Wert, in diesem Fall den String '2016-12-31', und to_date konvertiert diesen Wert dann in ein Datum.
- Hinzufügen des Feldes "current_age" mit Hilfe der Funktion 'months_between', die den Abstand in Monaten zwischen zwei Datumsfeldern berechnet. Das Alter in Jahren wird dann durch das Teilen und Abrunden (floor) der Monate durch 12 errechnet.

```
In [35]: cust_bdate = cust_bdate.withColumn("bdate",F.to_date(cust_bdate.Customer_age)
                                  ).withColumn("ucv_date", F.to_date(F.lit("2016-12-31"))
                                  ).withColumn("current_age", F.floor(F.months_between("ucv_date"
```

```
In [36]: cust_bdate.limit(5).toPandas()
```

```
Out[36]:
```

	Customer_id	Customer_age	bdate	ucv_date	current_age
0	25461829	1974-02-27	1974-02-27	2016-12-31	42
1	25461830	1994-12-11	1994-12-11	2016-12-31	22
2	25461831	1990-06-19	1990-06-19	2016-12-31	26
3	25461832	1974-05-04	1974-05-04	2016-12-31	42
4	25461834	1960-04-14	1960-04-14	2016-12-31	56

Da uns nur das Alter interessiert, löschen wir die Felder, die wir für das Anspielen des Alters an das ucv benötigen, wieder heraus. Dies geht wie oben erwähnt mit der Funktion 'drop()'

```
In [37]: cust_bdate = cust_bdate.drop("Customer_age"
                                    ).drop("bdate"
                                    ).drop("ucv_date")
```

4.4.1 *Erweiterungshinweis 3*

Genauso sinnvoll wie das Alter könnte das Ausrechnen der Zeit seit Beginn der Kundenbeziehung sein (Customer_since)

```
In [38]: dauer_kundb=customers.withColumn("Customer_since",F.to_date(customers.Customer_since)
                                  ).withColumn("ucv_date", F.to_date(F.lit("2016-12-31"))
                                  ).withColumn("dauer_kundb", F.floor(F.months_between("ucv_date"
```

```
In [39]: dauer_kundb = dauer_kundb.drop("Customer_since").drop("Customer_gender").drop("Customer
```

```
In [40]: dauer_kundb.limit(10).toPandas()
```

```
Out[40]:
```

	Customer_id	dauer_kundb
0	25461829	5
1	25461830	1
2	25461831	2

```
3     25461832            0
4     25461834            6
5     25461835            2
6     25461837            2
7     25461838            5
8     25461839            0
9     25461840            5
```

Wir erweitern das UCV nun um das neu berechnete Feld:

```
In [41]: ucv = ucv.join(cust_bdate, "Customer_id", "left_outer")

In [42]: ucv.limit(5).toPandas()

Out[42]:    customer_id Customer_gender     Phone zip1  current_age
         0     25402550          female  +49-122-    7           42
         1     25402650          female  (0101) 8    6           40
         2     25403321            male  +49-7515    8           38
         3     25403599            male  (0002) 4    4           46
         4     25404111            male  (0685) 5    1           47

In [43]: ucv.crosstab('current_age', 'Customer_gender').show()

+--------------------------+------+----+----+
|current_age_Customer_gender|female|male|null|
+--------------------------+------+----+----+
|                        69|   142| 143|   0|
|                      null|     0|   0|   3|
|                        88|     6|   5|   0|
|                        56|   657| 677|   0|
|                        42|  1329|1360|   0|
|                        24|   427| 448|   0|
|                        37|  1306|1310|   0|
|                        25|   492| 501|   0|
|                        52|   874|1009|   0|
|                        46|  1253|1239|   0|
|                        93|     2|   1|   0|
|                        57|   634| 600|   0|
|                        78|    37|  27|   0|
|                        29|   865| 802|   0|
|                        84|    17|  11|   0|
|                        61|   423| 407|   0|
|                        89|     4|   3|   0|
|                        74|    71|  75|   0|
|                        60|   428| 474|   0|
|                        85|    10|  17|   0|
+--------------------------+------+----+----+
only showing top 20 rows
```

4.5 2.5 Berechnung der Attribute aus den Orders

Nun kommt der spannende Teil der Aufbereitung. Im Marketing für Bestandskunden ist es bekannt, dass die wichtigsten Attribute für das Abschätzen des weiteren Verhaltens von Kunden die so genannten RFM-Variablen sind:

- Recency: Wie lange ist es her, dass der Kunde eine Bestellung getätigt hat?
- Frequency: Wie oft tätigt der Kunde Bestellungen pro Zeiteinheit?
- Monetary Value: Wie viel Geld hat der Kunde bereits umgesetzt?

Um diese Kennzahlen zu ermitteln, werden die Informationen in der Orders-Tabelle herangezogen. Diese Tabelle hat mitunter mehrere Einträge für jeden Kunden, da diese ja mehrere Artikel pro Order und/oder mehrere Orders durchführen können. Hier ist also eine geeignete Aggregation gefragt.

Zudem muss man sich noch entscheiden, wie weit man in der Historie des Kunden zurück gehen möchte. Im folgenden Beispiel gehen wir ein Jahr zurück, also 12 Monate bis 01.01.2016.

Um die Orders zu erhalten, die für diese Berechnung relevant sind, filtern wir zunächst die Tabelle der Orders auf alle Transaktionen zwischen dem 01.01.2016 und dem 31.12.2016

```
In [44]: rel_orders = orders.where("Date >= '2016-01-01' AND Date <= '2016-12-31'")

In [45]: rel_orders.count()

Out[45]: 59645

In [46]: rel_orders.select("Customer_id").distinct().count()

Out[46]: 17980
```

Wir haben also ca. 59645 Orders, die wir für die Analyse nutzen können, allerdings haben nur ca. 17980 der insgesamt ca. 87275 Kunden in diesem Zeitraum eine Transaktion gehabt, so dass das Modell für die meisten Kunden aus den Transaktionen nichts lernen kann.

Wir berechnen jetzt ein R, F, M - Kennzeichen ein Attribut welches wir unserem UCV hinzufügen. Für die Recency schauen wir zunächst, wann der jeweilige Kunde den letzten Kauf (Wenn er einen hatte) im Jahr 2016 durchgeführt hat. Danach berechnen wir die Anzahl der Monate, die am 31.12.2016 seit dem letzten Kauf vergangen sind.

Diese Berechnung nehmen wir analog zur Berechnung des Alters des Kunden von weiter oben vor.

```
In [47]: rel_orders = rel_orders.withColumn("t0", F.to_date(F.lit("2016-12-31"))
                                ).withColumn("order_date", F.to_date(rel_orders.Date)
                                ).withColumn("recency_in_1year", F.months_between("t0", "order_da

In [48]: rel_orders.limit(5).toPandas()

Out[48]:    Customer_id Invoice_id Item_number        Date  Quantity  Price  \
         0     25402265   fce026a5           1  2016-02-02         1  63.03
         1     25402265   fce026a5           2  2016-02-02         1   8.52
         2     25402265   fce026a5           3  2016-02-02         1   4.75
         3     25402265   50dc453e           1  2016-05-20         1  18.77
```

```
4     25402265   50dc453e         r_1  2016-05-26         1 -18.77

     Order_type Product_groupid          t0  order_date  recency_in_1year
0     Telephone           Shoes  2016-12-31  2016-02-02         10.935484
1     Telephone      Kids Style  2016-12-31  2016-02-02         10.935484
2     Telephone            Kids  2016-12-31  2016-02-02         10.935484
3     Telephone            Kids  2016-12-31  2016-05-20          7.354839
4  cancellation            Kids  2016-12-31  2016-05-26          7.161290
```

Im Gegensatz zu der Kunden-Datei kann man an dem obigen Output sehen, dass ein Kunde hier mehrmals vorkommen kann. Um also ein Attribut für unser UCV zu erreichen, müssen wir diese potentiell verschiedenen Werte noch aggregieren.

Es macht Sinn, das Recency-Kennzeichen auf dem zuletzt erfolgten Kauf zu berechnen. Dies bedeutet, dass wir den minimalen Wert der 'recency_in_1year' erhalten möchten.

Aggregationen in PysparkSQL sehen so aus:

```
In [49]: recency_1year = rel_orders.groupBy("Customer_id"
                                    ).agg(F.min(rel_orders.recency_in_1year).alias("recency_1year"
                                    )

In [50]: recency_1year.limit(5).toPandas()

Out[50]:    Customer_id  recency_1year
         0     25402650      11.806452
         1     25407318       5.290323
         2     25408027       4.548387
         3     25408268       4.354839
         4     25409570       3.225806
```

Hier macht es Sinn, zu überprüfen, ob die Customer_id wirklich ein uniquer Schlüssel ist nach der Aggregation. Dies lässt sich recht einfach bewerkstelligen:

```
In [51]: recency_1year_count = recency_1year.count()
         recency_1year_distinct_custs = recency_1year.select("Customer_id").distinct().count()

         print("Anzahl Zeilen in der Tabelle: {} - Anzahl uniquer Kunden-IDs in der Tabelle: {}"
                     recency_1year_count, recency_1year_distinct_custs))

Anzahl Zeilen in der Tabelle: 17980 - Anzahl uniquer Kunden-IDs in der Tabelle: 17980
```

Die beiden Zahlen in dem ausgegebenen Text sollten übereinstimmen.

Nach der Prüfung fügen wir diese Variable wieder dem UCV zu. Diesmal dürfen wir keinen inner Join durchführen, da sehr viele Kunden in den letzten 12 Monaten keine Orders hatten. Bei einem inner Join würden diese Kunden aus dem UCV entfernt werden. Das UCV soll jedoch alle Kunden enthalten, die am 31.12.2016 mindestens eine Order getäigt hatten, egal wie lange diese Order nun her ist.

Daher wird hier ein Left outer Join durchgeführt, diesen erzeugt man durch das Keyword "left_outer".

```
In [52]: ucv = ucv.join(recency_1year, "Customer_id", "left_outer")

In [53]: ucv.limit(10).toPandas()
```

Out[53]:

	customer_id	Customer_gender	Phone	zip1	current_age	recency_1year
0	25402550	female	+49-122-	7	42	NaN
1	25402650	female	(0101) 8	6	40	11.806452
2	25403321	male	+49-7515	8	38	NaN
3	25403599	male	(0002) 4	4	46	NaN
4	25404111	male	(0685) 5	1	47	NaN
5	25404911	female	+49-123-	5	36	NaN
6	25405236	female	+49-706-	8	39	NaN
7	25406003	female	+49-110-	3	52	NaN
8	25406911	female	+49-300-	6	63	NaN
9	25407133	female	(06655)	2	31	NaN

Wir sehen, dass nur für wenige Kunden das Recency-Kriterium gefüllt ist (Dies sind die Felder, in denen NaN steht => 'Not a Number'). Dies liegt daran, dass die Analyse sich nur die Käufe der letzten 12 Monate anschaut, während viele Kunden in der Datenbank schon länger nicht mehr gekauft, haben.

4.5.1 *Erweiterungshinweis 4*

Nur ein Jahr zurück zu gehen ist eventuell ja nicht genug, um die Kunden gut einschätzen zu können. Es könnte ja auch Sinn machen, sich den Kunden etwas länger in die Vergangenheit zurück anzuschauen?

Dies ist eine wichtige Aufgabe: Ohne Merkmale, die etwas länger zurück reichen, insbesondere für die Kennzahl "Recency" wird es schwierig werden, ein gutes Modell zu erstellen. Es empfiehlt sich, daher den vollen Datenzeitraum ab dem 01.01.2010 auszuschöpfen, z.B. mit RFM-Variablen vom Typ '_7years'

```
In [54]: rel_orders7 = orders.where("Date >= '2010-01-01' AND Date <= '2016-12-31'")

In [55]: rel_orders7.count()
```

Out[55]: 436059

```
In [56]: rel_orders7.select("Customer_id").distinct().count()
```

Out[56]: 87275

```
In [57]: rel_orders7 = rel_orders.withColumn("t0", F.to_date(F.lit("2016-12-31"))
                                 ).withColumn("order_date", F.to_date(rel_orders.Date)
                                 ).withColumn("recency_in_7year", F.months_between("t0", "order_da

In [58]: rel_orders7.limit(10).toPandas()
```

Out[58]:

	Customer_id	Invoice_id	Item_number	Date	Quantity	Price	\
0	25402265	fce026a5	1	2016-02-02	1	63.03	
1	25402265	fce026a5	2	2016-02-02	1	8.52	

```
2     25402265   fce026a5           3  2016-02-02         1   4.75
3     25402265   50dc453e           1  2016-05-20         1  18.77
4     25402265   50dc453e         r_1  2016-05-26         1 -18.77
5     25402265   50dc453e           2  2016-05-20         1  15.24
6     25402266   84d62139           1  2016-01-15         1   7.68
7     25402266   9c14f7bf           1  2016-05-02         1  36.30
8     25402266   9c14f7bf           2  2016-05-02         1  45.20
9     25402266   3381c756           1  2016-10-29         1  47.05

     Order_type Product_groupid          t0  order_date  recency_in_1year  \
0     Telephone           Shoes  2016-12-31  2016-02-02         10.935484
1     Telephone      Kids Style  2016-12-31  2016-02-02         10.935484
2     Telephone            Kids  2016-12-31  2016-02-02         10.935484
3     Telephone            Kids  2016-12-31  2016-05-20          7.354839
4  cancellation            Kids  2016-12-31  2016-05-26          7.161290
5     Telephone     Woman trend  2016-12-31  2016-05-20          7.354839
6     Telephone    Woman casual  2016-12-31  2016-01-15         11.516129
7      Internet            Kids  2016-12-31  2016-05-02          7.935484
8      Internet       Men trend  2016-12-31  2016-05-02          7.935484
9     Telephone       Men trend  2016-12-31  2016-10-29          2.064516

   recency_in_7year
0         10.935484
1         10.935484
2         10.935484
3          7.354839
4          7.161290
5          7.354839
6         11.516129
7          7.935484
8          7.935484
9          2.064516
```

```
In [59]: recency_7year = rel_orders7.groupBy("Customer_id"
                              ).agg(F.min(rel_orders7.recency_in_7year).alias("recency_7year
                              )
```

```
In [60]: recency_7year.limit(10).toPandas()
```

```
Out[60]:    Customer_id  recency_7year
         0     25402650      11.806452
         1     25407318       5.290323
         2     25408027       4.548387
         3     25408268       4.354839
         4     25409570       3.225806
         5     25413591       2.645161
         6     25415099       2.935484
         7     25415403       6.774194
```

```
8     25416163       8.161290
9     25418099       4.096774
```

```
In [61]: recency_7year_count = recency_7year.count()
         recency_7year_distinct_custs = recency_7year.select("Customer_id").distinct().count()

         print("Anzahl Zeilen in der Tabelle: {} - Anzahl uniquer Kunden-IDs in der Tabelle: {}"
                     recency_7year_count, recency_7year_distinct_custs))
```

```
Anzahl Zeilen in der Tabelle: 17980 - Anzahl uniquer Kunden-IDs in der Tabelle: 17980
```

```
In [62]: ucv = ucv.join(recency_7year, "Customer_id", "left_outer")
```

```
In [63]: ucv.limit(10).toPandas()
```

```
Out[63]:    customer_id Customer_gender     Phone zip1  current_age  recency_1year  \
         0     25402550          female  +49-122-    7           42            NaN
         1     25402650          female  (0101) 8    6           40      11.806452
         2     25403321            male  +49-7515    8           38            NaN
         3     25403599            male  (0002) 4    4           46            NaN
         4     25404111            male  (0685) 5    1           47            NaN
         5     25404911          female  +49-123-    5           36            NaN
         6     25405236          female  +49-706-    8           39            NaN
         7     25406003          female  +49-110-    3           52            NaN
         8     25406911          female  +49-300-    6           63            NaN
         9     25407133          female  (06655)     2           31            NaN

            recency_7year
         0            NaN
         1      11.806452
         2            NaN
         3            NaN
         4            NaN
         5            NaN
         6            NaN
         7            NaN
         8            NaN
         9            NaN
```

4.5.2 *Erweiterungshinweis 5*

Von den RFM-Kriterien wurde bis jetzt nur das "R" implementiert. Es macht viel Sinn, auch die anderen Kriterien noch in den UCV aufzunehmen.

Für die 'frequency_1year' ist eine Aggregation nötig, die die Anzahl der verschiedenen Invoice_IDs pro Kunden zählt. Für das Attribut 'monetary_1year' ist eine Summe über (quantity * price) des jeweiligen Kunden notwendig. Wichtig ist es, die Multiplikation von quantity und price vor der Aggregation durchzuführen.

Neben den RFM-Werten stecken in den Order-Daten noch weitere interessante Informationen. Zum Beispiel gibt es verschiedene Order-Typen wie Telephone, Internet und Cancellation. Es könnte Sinn machen, für diese auch R,F,M-Kritierien aufzubereiten.

Um ein ordentliches Predictive Model zu erstellen, sollten hier auf jeden Fall noch einige weitere Attribute aufgebaut werden. Es müssen allerdings auch nicht hunderte von Attributen werden.

```
In [64]: frequency_1year = rel_orders.groupBy("Customer_id"
                                ).agg(F.count(rel_orders.Invoice_id).alias("frequency_1year")
                                )

In [65]: frequency_1year.limit(10).toPandas()
```

Out[65]:

	Customer_id	frequency_1year
0	25402650	1
1	25407318	1
2	25408027	5
3	25408268	4
4	25409570	2
5	25413591	1
6	25415099	3
7	25415403	1
8	25416163	5
9	25418099	3

```
In [66]: ucv = ucv.join(frequency_1year, "Customer_id", "left_outer")

In [67]: rel_orders.limit(5).toPandas()
```

Out[67]:

	Customer_id	Invoice_id	Item_number	Date	Quantity	Price	\
0	25402265	fce026a5	1	2016-02-02	1	63.03	
1	25402265	fce026a5	2	2016-02-02	1	8.52	
2	25402265	fce026a5	3	2016-02-02	1	4.75	
3	25402265	50dc453e	1	2016-05-20	1	18.77	
4	25402265	50dc453e	r_1	2016-05-26	1	-18.77	

	Order_type	Product_groupid	t0	order_date	recency_in_1year
0	Telephone	Shoes	2016-12-31	2016-02-02	10.935484
1	Telephone	Kids Style	2016-12-31	2016-02-02	10.935484
2	Telephone	Kids	2016-12-31	2016-02-02	10.935484
3	Telephone	Kids	2016-12-31	2016-05-20	7.354839
4	cancellation	Kids	2016-12-31	2016-05-26	7.161290

```
In [68]: rel_orders = rel_orders.withColumn("sum_money1",rel_orders.Quantity * rel_orders.Price)

In [69]: rel_orders.limit(5).toPandas()
```

Out[69]:

	Customer_id	Invoice_id	Item_number	Date	Quantity	Price	\
0	25402265	fce026a5	1	2016-02-02	1	63.03	

```
1     25402265   fce026a5           2  2016-02-02         1   8.52
2     25402265   fce026a5           3  2016-02-02         1   4.75
3     25402265   50dc453e           1  2016-05-20         1  18.77
4     25402265   50dc453e         r_1  2016-05-26         1 -18.77

     Order_type Product_groupid          t0  order_date  recency_in_1year  \
0     Telephone           Shoes  2016-12-31  2016-02-02         10.935484
1     Telephone      Kids Style  2016-12-31  2016-02-02         10.935484
2     Telephone            Kids  2016-12-31  2016-02-02         10.935484
3     Telephone            Kids  2016-12-31  2016-05-20          7.354839
4  cancellation            Kids  2016-12-31  2016-05-26          7.161290

   sum_money1
0       63.03
1        8.52
2        4.75
3       18.77
4      -18.77
```

```
In [70]: monetary_1year = rel_orders.groupBy("Customer_id"
                               ).agg(F.sum(rel_orders.sum_money1).alias("monetary_1year")
                               )
```

```
In [71]: monetary_1year.limit(5).toPandas()
```

```
Out[71]:    Customer_id  monetary_1year
         0     25402650           38.62
         1     25407318           36.45
         2     25408027           42.22
         3     25408268            0.00
         4     25409570            0.00
```

```
In [72]: ucv = ucv.join(monetary_1year, "Customer_id", "left_outer")
```

```
In [73]: ucv.limit(5).toPandas()
```

```
Out[73]:    customer_id Customer_gender     Phone zip1  current_age  recency_1year  \
         0     25402550          female  +49-122-    7           42            NaN
         1     25402650          female  (0101) 8    6           40      11.806452
         2     25403321            male  +49-7515    8           38            NaN
         3     25403599            male  (0002) 4    4           46            NaN
         4     25404111            male  (0685) 5    1           47            NaN

            recency_7year  frequency_1year  monetary_1year
         0            NaN              NaN             NaN
         1      11.806452              1.0           38.62
         2            NaN              NaN             NaN
         3            NaN              NaN             NaN
         4            NaN              NaN             NaN
```

This is RFM for Order_type

```
In [74]: frequency_1year_orders = rel_orders.groupBy("Order_type"
                                 ).agg(F.count(rel_orders.Customer_id).alias("frequency_1year")
                                 )

In [75]: frequency_1year_orders.limit(5).toPandas()

Out[75]:      Order_type  frequency_1year
         0  cancellation            11923
         1     Telephone            23639
         2      Internet            24083

In [76]: monetary_1year_orders = rel_orders.groupBy("Order_type"
                                 ).agg(F.sum(rel_orders.sum_money1).alias("monetary_1year")
                                 )

In [77]: monetary_1year_orders.limit(5).toPandas()

Out[77]:      Order_type  monetary_1year
         0  cancellation      -369250.24
         1     Telephone       738439.78
         2      Internet       748982.68

In [78]: recency_1year_orders = rel_orders.groupBy("Order_type"
                                 ).agg(F.min(rel_orders.recency_in_1year).alias("recency_1year"
                                 )

In [79]: recency_1year_orders.limit(5).toPandas()

Out[79]:      Order_type  recency_1year
         0  cancellation            0.0
         1     Telephone            0.0
         2      Internet            0.0

In [80]: order_type = frequency_1year_orders.join(monetary_1year_orders, "Order_type", "left_out

In [81]: order_type = order_type.withColumn("mean_order",order_type.monetary_1year / order_type.

In [82]: order_type.limit(5).toPandas()

Out[82]:      Order_type  frequency_1year  monetary_1year  mean_order
         0  cancellation            11923      -369250.24  -30.969575
         1     Telephone            23639       738439.78   31.238199
         2      Internet            24083       748982.68   31.100057
```

4.6 2.6 Ersetzen von fehlenden Werten

Um den Bau des UCV auf der Attributseite abzuschlieSSen, sollten wir jetzt noch die undefinierten Werte mit sinnvollen Platzhaltern zu ersetzen.

Oft ist dieser Platzhalter schlicht die '0'. Wenn zum Beispiel kein Kauf durch den Kunden erfolgte oder kein Umsatz getätigt wurde, so ist 0-mal etwas passiert und es wurden 0 € Umsatz erzeugt. Bei der Recency ist die 0 nicht korrekt, denn dort würde eni Wert von 0 ja bedeuten, dass der Kunde gerade eben eine Order getätigt hat. Hier sollte man eine hohe Zahl nehmen, z.B. 9999. Dies stellt sicher, dass diese Kunden nicht mit Kunden vermischt werden, die einen Wert bei Recency stehen haben.

Diese Ersetzungen kann man mit einem CASE WHEN-Statement vornehmen. Die Syntax in PySparkSQL ist dafür etwas gewöhnungsbedürftig, aber dies hat man schnell raus. Wie man unter sieht, kann man in PySparkSQL Felder "on the fly" ersetzen, so dass man das Feld, welches man überschreibt während der Aktion noch als Quellfeld benutzen kann.

```
In [83]: ucv = ucv.withColumn("recency_1year", F.when(ucv.recency_1year > -1, ucv.recency_1year)

In [84]: ucv = ucv.withColumn("recency_7year", F.when(ucv.recency_7year > -1, ucv.recency_7year)

In [85]: ucv = ucv.withColumn("frequency_1year", F.when(ucv.frequency_1year > -1, ucv.frequency_

In [86]: ucv = ucv.withColumn("monetary_1year", F.when(ucv.monetary_1year > -1, ucv.monetary_1ye

In [87]: ucv.limit(10).toPandas()
```

Out[87]:

	customer_id	Customer_gender	Phone	zip1	current_age	recency_1year	\
0	25402550	female	+49-122-	7	42	9999.000000	
1	25402650	female	(0101) 8	6	40	11.806452	
2	25403321	male	+49-7515	8	38	9999.000000	
3	25403599	male	(0002) 4	4	46	9999.000000	
4	25404111	male	(0685) 5	1	47	9999.000000	
5	25404911	female	+49-123-	5	36	9999.000000	
6	25405236	female	+49-706-	8	39	9999.000000	
7	25406003	female	+49-110-	3	52	9999.000000	
8	25406911	female	+49-300-	6	63	9999.000000	
9	25407133	female	(06655)	2	31	9999.000000	

	recency_7year	frequency_1year	monetary_1year
0	9999.000000	9999	0.00
1	11.806452	1	38.62
2	9999.000000	9999	0.00
3	9999.000000	9999	0.00
4	9999.000000	9999	0.00
5	9999.000000	9999	0.00
6	9999.000000	9999	0.00
7	9999.000000	9999	0.00
8	9999.000000	9999	0.00
9	9999.000000	9999	0.00

4.7 2.7 Ermittlung der Erfolgsvariable für das Training des Modells

Für die Fertigstellung des UCV fehlt nun noch die Ergebnis- oder Erfolgsvariable, auf die der Algorithmus trainiert werden kann.

Hier soll dies eine einfache numerische Variable sein, die immer dann den Wert 0 hat, wenn der Kunde im Zielzeitraum keine positiven Umsätze gemacht hat, und dann den Wert 1 hat, wenn der Kunde positive Umsätze gemacht hat.

Daraus folgt, dass ein Kunde, der z.B. eine Hose für 69.95€ bestellt und diese dann wieder zurückschickt, eine 0 bekommt, während ein Kunde, der zusätzlich zu der Hose noch einen Gürtel für 29.95€ bestellt hat und diesen behält, eine 1 bekommt.

Diese Variable kann wieder durch Aggregation ermittelt werden. Diesmal müssen allerdings die Order-Daten auf den Zielzeitraum eingeschränkt werden, den wir auf die Monate Januar 2017 und Februar 2017 gesetzt haben.

```
In [88]: target_orders = orders.where("Date > '2016-12-31' and Date < '2017-03-01'")

In [89]: target_orders.count()

Out[89]: 8886

In [90]: target_orders.select("Customer_id").distinct().count()

Out[90]: 3932
```

Wir berechnen zunächst den Wert einer Order-Zeile als price * quantity

```
In [91]: target_orders = target_orders.withColumn("value", target_orders.Price*target_orders.Qua

In [92]: target_orders.limit(5).toPandas()

Out[92]:    Customer_id Invoice_id Item_number        Date  Quantity  Price  \
         0     25402285   86fa2e9f           1  2017-02-14         1   1.00
         1     25402285   86fa2e9f           2  2017-02-14         1   9.40
         2     25402285   86fa2e9f           3  2017-02-14         1  39.94
         3     25402285   86fa2e9f         r_3  2017-02-22         1 -39.94
         4     25402318   757edef4           1  2017-02-23         1  39.55

              Order_type Product_groupid  value
         0     Telephone      Kids Style   1.00
         1     Telephone      Kids Style   9.40
         2     Telephone            Kids  39.94
         3  cancellation            Kids -39.94
         4     Telephone      Men casual  39.55
```

Nun können wir diese Tabelle auf die Customer_id aggregieren und als Summe aggregieren.

```
In [93]: target_var = target_orders.groupBy("Customer_id").agg(F.sum(target_orders.value).alias(

In [94]: target_var.limit(3).toPandas()
```

```
Out[94]:    Customer_id  sum_value
         0     25405830     130.99
         1     25408027      21.38
         2     25416163      32.16
```

Um nun unsere Zielvariable zu erhalten, wird wieder ein Case When-Statement genutzt:

```
In [95]: target_var = target_var.withColumn("target", F.when(target_var.sum_value > 0, 1).otherw
                                          ).drop("sum_value")

In [96]: target_var.limit(10).toPandas()

Out[96]:    Customer_id  target
         0     25405830       1
         1     25408027       1
         2     25416163       1
         3     25434916       1
         4     25438084       1
         5     25440550       1
         6     25445347       0
         7     25446935       1
         8     25449131       1
         9     25463895       1
```

Nun haben wir die Erfolgsvariable abgeleitet und müssen diese nun noch an das UCV anjoinen - Wieder per left outer Join.

```
In [97]: target_var.count()

Out[97]: 3932

In [99]: ucv.select("Customer_id").distinct().count()

Out[99]: 87275

In [100]: ucv = ucv.join(target_var, "Customer_id", "left_outer")

In [101]: ucv.limit(5).toPandas()

Out[101]:    customer_id Customer_gender     Phone zip1  current_age  recency_1year  \
          0     25402550          female  +49-122-    7           42    9999.000000
          1     25402650          female  (0101) 8    6           40      11.806452
          2     25403321            male  +49-7515    8           38    9999.000000
          3     25403599            male  (0002) 4    4           46    9999.000000
          4     25404111            male  (0685) 5    1           47    9999.000000

             recency_7year  frequency_1year  monetary_1year target
          0    9999.000000             9999            0.00   None
          1      11.806452                1           38.62   None
          2    9999.000000             9999            0.00   None
          3    9999.000000             9999            0.00   None
          4    9999.000000             9999            0.00   None
```

Da nur sehr wenige der Bestandskunden im Beobachtungszeitraum kaufen (Leider ein sehr realistisches Szenario im e-Commerce) sind hier viele Werte noch fehlend. Wir ersetzen diese wieder, analog zu dem Vorgehen bei der Recency-Variable, diesmal allerdings mit 0, da das Fehlen einer Kaufinformation bedeutet, das nicht gekauft wurde.

```
In [102]: ucv = ucv.withColumn("target", F.when(ucv.target > -1, ucv.target).otherwise(0))
```

```
In [103]: ucv.limit(5).toPandas()
```

```
Out[103]:    customer_id Customer_gender     Phone zip1  current_age  recency_1year  \
          0     25402550          female  +49-122-    7           42    9999.000000
          1     25402650          female  (0101) 8    6           40      11.806452
          2     25403321            male  +49-7515    8           38    9999.000000
          3     25403599            male  (0002) 4    4           46    9999.000000
          4     25404111            male  (0685) 5    1           47    9999.000000

             recency_7year  frequency_1year  monetary_1year  target
          0    9999.000000             9999            0.00       0
          1      11.806452                1           38.62       0
          2    9999.000000             9999            0.00       0
          3    9999.000000             9999            0.00       0
          4    9999.000000             9999            0.00       0
```

Damit ist der zweite Block, die Erstellung des Flatfiles abgeschlossen

---

5 Schritt 3: Deskriptive Analyse der Attribute

5.1 3.1. Allgemeine Analysen

In diesem Block geht es darum, ein "Gefühl" für die Daten zu bekommen. Daher werden wir einige deskriptive Analysen machen, um zu sehen, welche Attribute für ein Predictive Model interessant sein könnten.

Zunächst schauen wir uns einmal die Verteilung der Erfolgsvariable an. Dafür erstellen wir eine Häufigkeitstabelle der Nullen und Einsen im Datensatz

```
In [104]: ucv.groupBy("target").count().toPandas()
```

```
Out[104]:    target  count
          0       1   2546
          1       0  84729
```

Wie man sieht, sind die allermeisten Fälle der Nichtkauf. Dies ist absolut typisch für e-Commerce-Daten.

Um ein etwas besseres Gefühl dafür zu bekommen, wie die Daten verteilt sind, schauen wir uns die Daten einmal generell mit der 'describe()'-Funktion an.

```
In [105]: ucv.describe().toPandas()
```

Out[105]:

	summary	customer_id	Customer_gender	Phone	zip1
0	count	87275	87272	87272	87272
1	mean	2.5462205393033516E7	None	None	4.497009350077917
2	stddev	34668.30151510231	None	None	2.289638121938325
3	min	25402226	female	(0000) 0	1
4	max	25522222	male	+49-8888	8

	current_age	recency_1year	recency_7year
0	87272	87275	87275
1	41.8024796039967	7940.07986139482	7940.07986139482
2	12.382089618354067	4042.016935027861	4042.016935027861
3	22	0.0	0.0
4	100	9999.0	9999.0

	frequency_1year	monetary_1year	target
0	87275	87275	87275
1	7939.734746490977	12.850919851045536	0.029172156975078772
2	4042.6942894914796	36.150984850228895	0.16828982958234823
3	1	-0.07000000000000028	0
4	9999	500.57000000000005	1

Das mittlere Alter seint etwa 41 Jahre zu sein, mit einer Standardabweichung von ca 12 Jahren nach oben und unten.

5.2 3.2 Kreuztabellen

Um den Einfluss von kategorialen Attributen anzuschauen, eignen sich Kreuztabellen. Im folgenden schauen wir uns eine Kreuztabelle der Zielvariable nach dem Attribut "Customer_gender" an. Kreutabellen sind dadurch gekennzeichnet, dass man die ursprünglichen Daten nach zwei Attributen gruppiert, im folgenden "target" und "Customer_gender".

```
In [106]: ucv.groupBy("target", "Customer_gender").count().orderBy("Customer_gender", "target").
```

Out[106]:

	target	Customer_gender	count
0	0	None	3
1	0	female	42180
2	1	female	1263
3	0	male	42546
4	1	male	1283

Wie man hier sieht, scheint das Geschlecht als einzelnes Attribut keinen starken Einfluss auf den Kauf zu haben. Von den Frauen haben 1263/(1263+42180) = 2,91 % gekauft, von den Männern haben 1283/(1283+42546) = 2,93% Prozent. Dies ist zwar ein kleiner, aber kein signifikanter Unterschied.

5.3 3.3 Boxplots von metrischen Variablen

Um für metrische Attribute den univariaten Zusammenhang mit der Zielvariable zu sehen, empfiehlt sich ein Boxplot, der einmal auf die 0 und einmal auf die 1 gemacht wird.

Ein Boxplot (https://de.wikipedia.org/wiki/Boxplot) zeigt die Verteilung der Werte von metrischen Variablen an. Wenn man nun für jedes metrische Attribut einen Boxplot für Datensätze, bei denen die Zielvariable jeweils 0/1 ist, macht, so kann man visuell analysieren, ob die Verteilung für die beiden Ergebnisse verschieden ist.

Um diese Plots mit Pandas zu machen, muss zunächst das pyplot-Paket importiert werden.

```
In [107]: import matplotlib.pyplot as plt
```

Um in Pandas einen Plot mehrmals pro Attribut auszuführen, muss das Attribut kategorial sein, also erzeugen wir ein neues Attribut "target_boolean", welches die Zielvariable als String anstelle als Zahl beinhaltet.

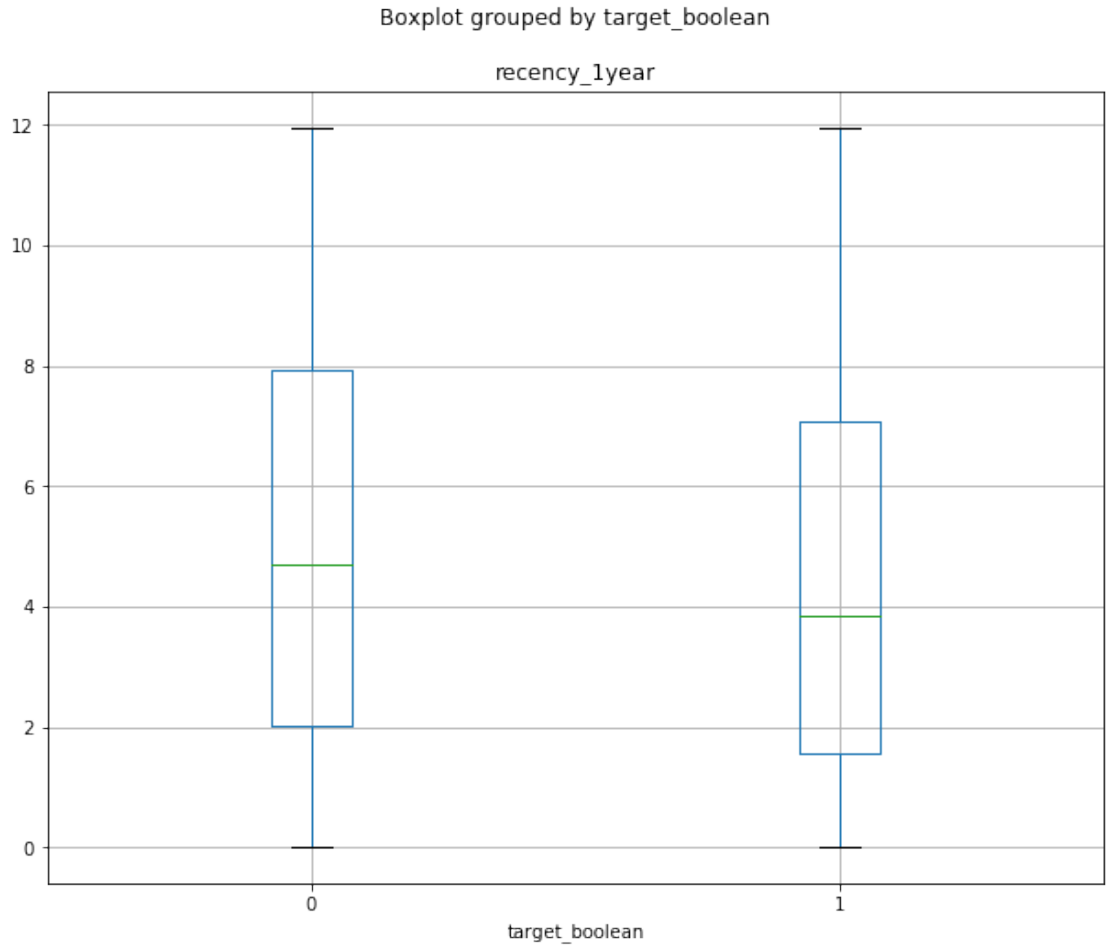
```
In [108]: ucv_des = ucv.withColumn("target_boolean", ucv.target.cast("string"))
```

Wir benutzen die Matplotlib-Bibliothek und müssen daher noch Plots für dieses Notebook aktivieren:

```
In [109]: %matplotlib inline
```

```
In [110]: ucv_des.where("recency_1year < 13").toPandas( #Alle 9999-Werte ausschließen, da sie so
                      ).boxplot(by='target_boolean',  #nach der Zielvariable gruppieren
                                column='recency_1year', #Feld, welches anylsiert werden soll
                                figsize = [10,8]     # Den Plot etwas größer machen
                      )
```

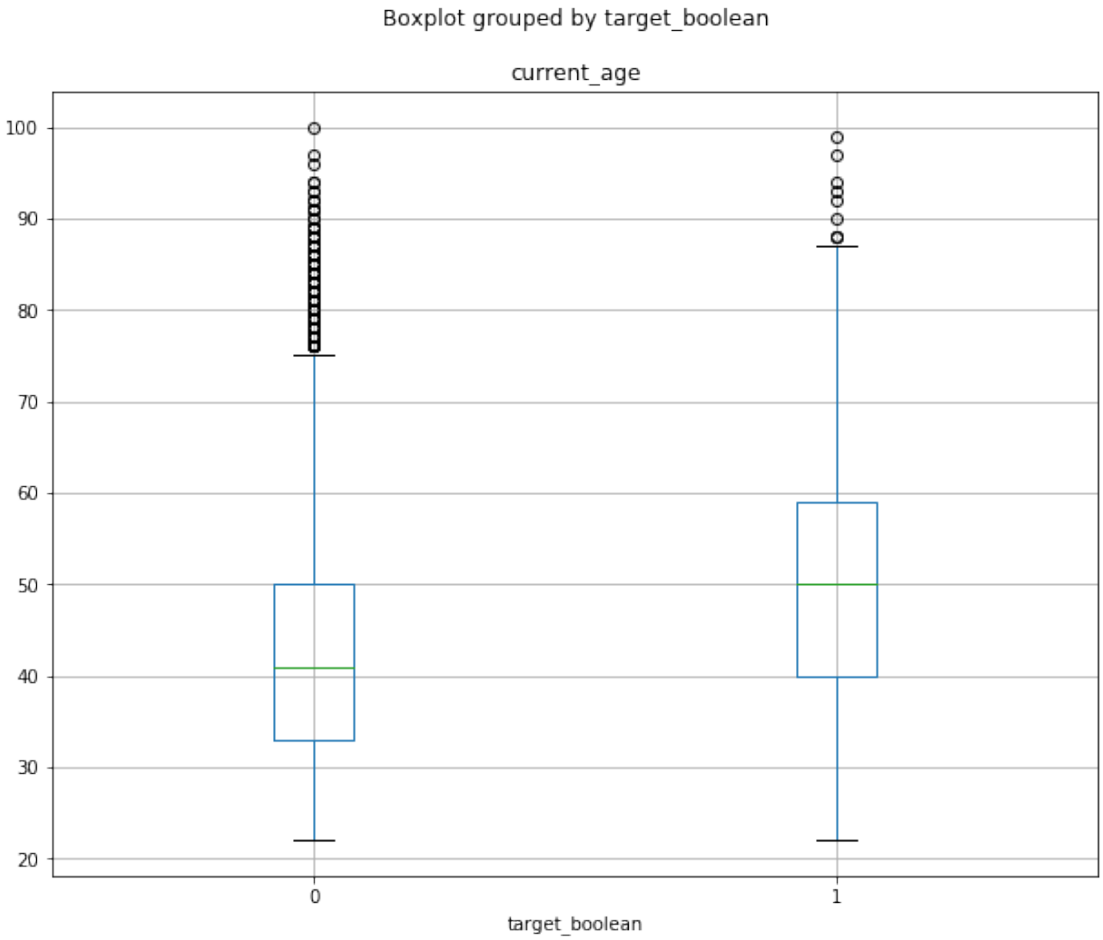
```
Out[110]: <matplotlib.axes._subplots.AxesSubplot at 0x7fa9bbd797f0>
```



Hier ist recht deutlich zu sehen, dass die Recencies für die Kunden, die gekauft haben, niedriger sind als für die, die nicht gekauft haben. Daher scheint die Zeit seit dem letzten Kauf ein interessanter Faktor für ein Predictive Model zu sein.

Ein weiterer Faktor, den man untersuchen könnte, wäre das Alter der Kunden. Wir plotten einmal auf die gleiche Weise das Alter gegen die kategoriale Zielvariable. Im Gegensatz zur Recency müssen wir hier keine Werte herausfiltern, so dass der Befehl etwas einfacher ist.

```
In [111]: ucv_des.toPandas().boxplot(by='target_boolean', column='current_age', figsize = [10,8]
```



Wie man hier sieht, scheint auch das Alter eine gute Vorhersagekraft auf den Verkaufserfolg zu haben. Bei den Kunden, die gekauft haben, liegt das mittlere Alter bei etwas über 50, bei denen, die nicht gekauft haben liegt es bei etwa Mitte 40. Auch dieses Attribut könnte also interessant für ein Predictive Model sein.

5.3.1 *Erweiterungshinweis 6*

Um zu entscheiden, welche Attribute in das Modell aufgenommen werden sollen, solltet Ihr hier auch weitere, eventuell von Ihnen zusätzlich erzeugte Attribute entweder mit Hilfe von Kreuztabellen oder von Boxplots oder von weiteren Visualisierungen untersuchen.

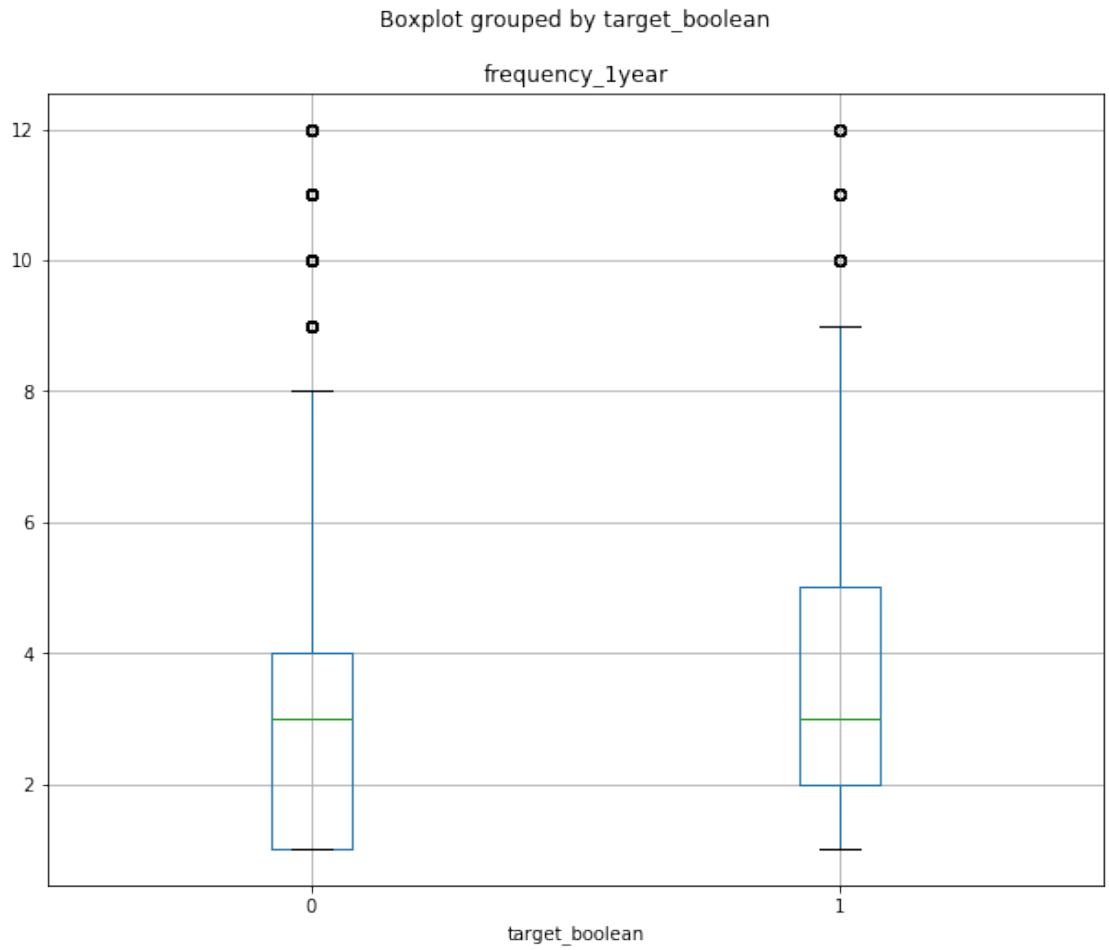
```
In [112]: ucv.groupBy("target", "zip1").count().orderBy("zip1", "target").toPandas()
```

```
Out[112]:     target  zip1  count
          0       0  None      3
          1       0     1  10605
          2       1     1    359
          3       0     2  10528
          4       1     2    284
```

```
5          0      3  10670
6          1      3    326
7          0      4  10490
8          1      4    337
9          0      5  10702
10         1      5    336
11         0      6  10574
12         1      6    321
13         0      7  10601
14         1      7    300
15         0      8  10556
16         1      8    283
```

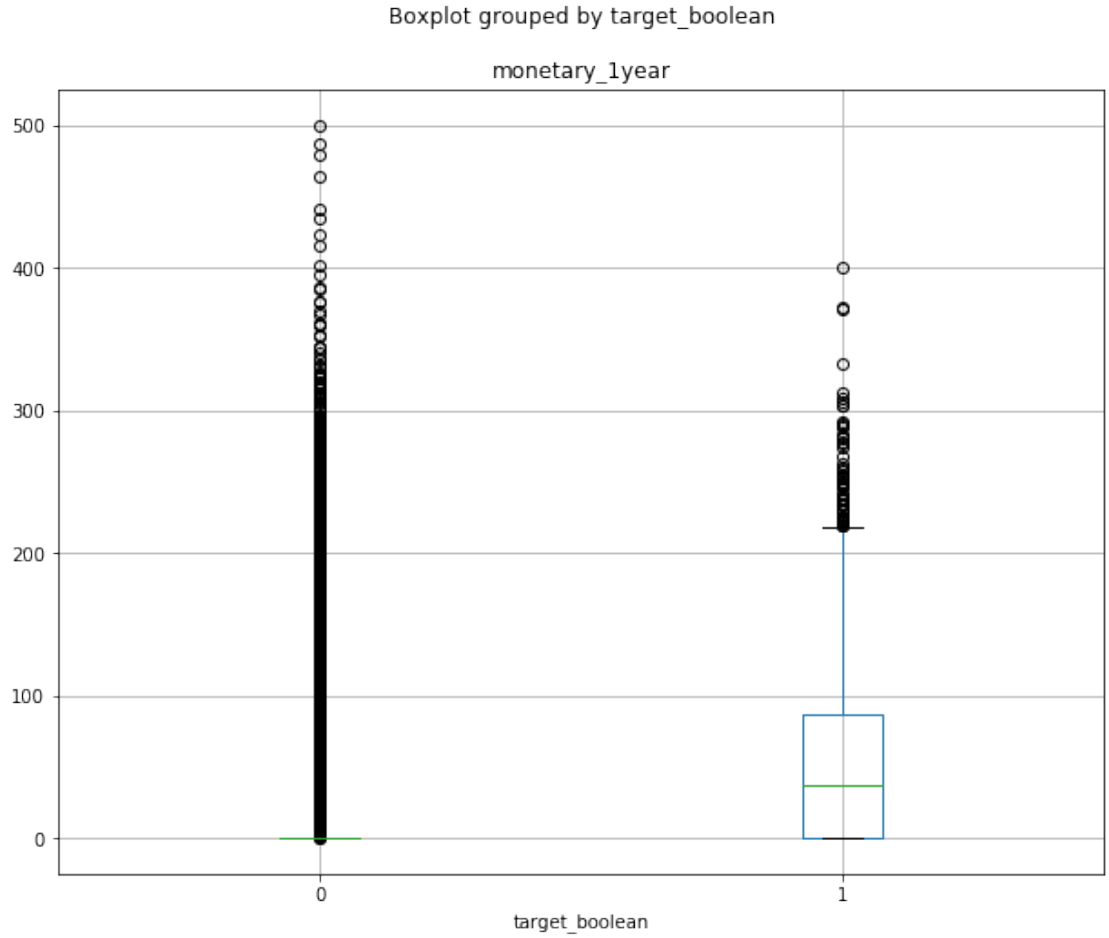
In [113]:
```
ucv_des.where("frequency_1year < 13").toPandas( #Alle 9999-Werte ausschließen, da sie
                    ).boxplot(by='target_boolean',  #nach der Zielvariable gruppieren
                              column='frequency_1year', #Feld, welches anylsiert werden soll
                              figsize = [10,8]      # Den Plot etwas größer machen
                    )
```

Out[113]: <matplotlib.axes._subplots.AxesSubplot at 0x7fa9c0e4b860>



```
In [115]: ucv_des.toPandas( #Alle 9999-Werte ausschließen, da sie sonst die Grafik ruinieren
                      ).boxplot(by='target_boolean',  #nach der Zielvariable gruppieren
                                column='monetary_1year', #Feld, welches anylsiert werden soll
                                figsize = [10,8]     # Den Plot etwas größer machen
                      )
```

```
Out[115]: <matplotlib.axes._subplots.AxesSubplot at 0x7fa9b8ed8400>
```



---

6 Schritt 4: Schätzen eines Predictive Models

Mit dem UCV welches wir erstellt haben, können wir direkt ein Predictive Model erstellen, da alle notwendigen Informationen bereits enthalten sind.

Ein Predictive Model erstellt anhand eines Kalibrierungsalgorithmus einen Zusammenhang zwischen den vergangenen Attributen (hier z.B. Alter, Geschlecht etc.) und einem in der Zukunft liegenden Ereignis (hier der Kauf). Dafür muss es auf bereits vergangene Episoden trainiert werden.

Wir verwenden hier eine der Basismethoden für die Kalibrierung von Modellen, die als Zielvariable eine 0 oder eine 1 haben: Die Logistische Regression. Dieses Verfahren ist ein klassisches Werkzeug der Statistik und ähnelt dem ältesten und bekanntesten Verfahren, der Linearen Regression recht stark. Im Rahmen dieser Veranstaltung gehen wir hier nicht weiter auf die Methode ein, der Wikipedia-Artikel dazu ist recht gut gemacht: https://de.wikipedia.org/wiki/Logistische_Regression

6.1 4.1 Vorbereitungen

Wir werden das Modell mit der MLLIB-Bibliothek von Apache Spark schätzen. Dieses Vorgehen erfordert einige Vorbereitungen.

Wir importieren zunächst die passenden Pakete aus PySpark, um diese im folgenden nutzen zu können.

```
In [116]: from pyspark.ml.classification import LogisticRegression
          from pyspark.ml import Pipeline
          from pyspark.ml.feature import VectorAssembler, StringIndexer, OneHotEncoder
```

Wir müssen wir das Feld mit dem Zielwert in 'label' umbenennen, da die Machine-Learning-Library von Apache Spark ein solches Feld erwartet.

```
In [117]: ucv = ucv.withColumnRenamed("target", "label")
```

6.2 4.2 Aufteilung der Daten in Training/Test

Als nächstes müssen wir die Daten in ein Trainings- und ein Testset aufteilen, um das Modell hinterher evaluieren zu können. Wir schlagen eine Aufteilung 75/25 (Training/Test) vor, aber es ist letzlich Ihre Entscheidung. Sie können die Aufteilung ggf. hier ändern, einfach einen anderen Werte für 'fraction' eintragen.

```
In [118]: train = ucv.sample(fraction=0.75, withReplacement=False)
```

Das Test-Sample wird erstellt, in dem wir das Trainings-Sample vom Gesamt-Datensatz abziehen. Übrig bleibt dann der Test-Datensatz

```
In [119]: test = ucv.subtract(train)
```

6.3 4.3 Vorbereitung der Daten für die Modellierung => Pipeline

MLLIB benötigt die Daten für die Modellierung in einem ganz bestimmten Format. Dazu müssen die Trainings-Werte in einem Feld mit dem Namen 'label' als numerische Werte vorliegen. Dies haben wir oben bereits erledigt. Zudem müssen alle Attribute in einem Feld mit dem Namen 'features' vorliegen.

Es müssen also die einzelnen Felder des UCV in ein einzelnes Feld 'verdichtet' werden. Der typische Weg, dies in Apache Spark zu machen, ist eine so genannte Pipeline. In einer Pipeline

werden nacheinander verschiedene Datenaufbereitungsschritte vorgenommen, wobei als Input für den nächsten Schritt das Ergebnis des vorherigen Schritts genutzt wird.

Eine tief gehende Dokumentation findet sich unter https://spark.apache.org/docs/latest/ml-pipeline.html

Der Hauptgrund, eine Pipeline einzusetzen, besteht darin, dass das Verfahren der logistischen Regression ausschlieSSlich numerische Attribute verwenden kann. Wir haben jedoch in unserem UCV verschiedene Attribute, die kategoriale Werte haben, wie zum Beispiel den Zipcode und das Geschlecht.

Damit wir ein logistisches Regressionsmodell schätzen können, müssen wir also zunächst diese kategorialen Ausprägungen in Zahlen umwandeln. Dafür stellt Apache Spark verschiedene Mechanismen zur Verfügung, von denen hier nur zwei notwendig sind: Der StringIndexer (https://spark.apache.org/docs/latest/ml-features.html#stringindexer) und der OneHotEncoder (https://spark.apache.org/docs/latest/ml-features.html#onehotencoder).

Der StringIndexer erzeugt aus einem String-Attribut einen numerischen Index, indem er jedem Wert eines Attributs eine Zahl als Index zuordnet. So wird aus "female" z.B. der Index 1 und aus "male" der Index 0. Damit ist der erste Schritt zu einer Umwandlung in ein numerisches Attribut erledigt.

In einem zweiten Schritt müssen nun die einzelnen Werte der Indizes in weitere Attribute aufgeteilt werden. Dies ist notwendig, da ansonsten die Werte der Indizes eine geordnete Bedeutung bekommen würden. Dies würde zum Beispiel den Postleitzahlen bedeuten, dass ein Index von 3 eine höhere Bewertung als ein Index von 2 hätte. Tatsächlich implizieren die Indizes aber keine Rangreihenfolge, sondern nur eine Zuordnung. Daher müssen die verschiedenen Werte in verschiedene Felder aufgesplittet werden.

Diese Aufgabe übernimmt der OneHotEncoder. Er erzeugt aus einem StringIndexer-Attribut mit z.B. drei verschiedenen Werten insgesamt drei neue Felder, die immer 0 sind, wenn der Werte zutrifft und 1 sind, wenn der Wert nicht zutrifft.

Am Beispiel des Geschlechts:

Aus

Geschlecht
male
female
unknown

macht der StringIndexer:

Geschlecht	GeschlechtIndex
male	0
female	1
unknown	2

Der OneHotEncoder wiederum erzeugt dann diese Felder:

Geschlecht	GeschlechtIndex	GeschlechtIndex_female	GeschlechtIndex_unknown
male	0	1	0
female	1	0	1

Geschlecht	Geschlecht_index	Geschlecht_female	Geschlecht_male	Geschlecht_unknown
unknown	2	0	0	1

Wir verwenden hier zwei kategoriale Attribute, den Zip-Code und das Geschlecht. Dafür kreieren wir jeweils einen Indexer und einen Encoder. Wichtig ist hier, dass die outputCol des vorhergehenden Schritts mit der inputCol des folgenden Schrittes übereinstimmt.

```
In [120]: zipIndexer = StringIndexer(inputCol="zip1", outputCol="zipIndex")
          zipEncoder = OneHotEncoder(inputCol="zipIndex", outputCol="zipVec")
```

```
In [121]: genderIndexer = StringIndexer(inputCol="Customer_gender", outputCol="genderIndex")
          genderEncoder = OneHotEncoder(inputCol="genderIndex", outputCol="genderVec")
```

6.3.1 *Erweiterungshinweis 7*

Hier können natürlich noch weitere kategorische Variablen erstellt werden. Basierend auf dem, was Ihr vorher in dem UCV aufbereitet habt, ist die Auswahl hier recht groSS. Am besten ist es, sich hier an die Ergebnisse aus der deskriptiven Analyse zu erinnern und viel versprechende Kandidaten hier mit dem StringIndexer und dem OneHotEncoder zu erstellen.

Ebenso können und sollten hier weitere metrische Variablen, z.B. aus dem RFM-Bereich erstellt werden.

```
In [131]: train.printSchema()
```

```
root
 |-- customer_id: long (nullable = true)
 |-- Customer_gender: string (nullable = true)
 |-- Phone: string (nullable = true)
 |-- zip1: string (nullable = true)
 |-- current_age: long (nullable = true)
 |-- recency_1year: double (nullable = true)
 |-- recency_7year: double (nullable = true)
 |-- frequency_1year: long (nullable = true)
 |-- monetary_1year: double (nullable = true)
 |-- label: integer (nullable = true)
```

Für bereits numerische Attribute muss kein Pre-Processing mehr durchgeführt werden. Diese können direkt, zusammen mit den durch Indexer und Encoder erzeugten Attributen in den finalen Schritt, den featureAssembler gegeben werden.

Der featureAssembler fügt einfach alle Attribute, die ihm übergeben werden, in ein Datenfeld namens "features" zusammen, so dass der Datensatz dann bereit für die Modellierung ist.

```
In [124]: phoneIndexer = StringIndexer(inputCol="Phone", outputCol="phoneIndex")
          phoneEncoder = OneHotEncoder(inputCol="phoneIndex", outputCol="phoneVec")
```

```
In [125]: featureAssembler = VectorAssembler(
              inputCols=["current_age", "recency_1year", "zipVec", "genderVec","frequency_1year"
              outputCol="features")
```

6.4 4.4 Schätzen des Modells => Ausführen der Pipeline

Die eigentliche logistische Regression ist dann einfach der letzte Schritt in der Pipeline, der ausgeführt wird. Es gibt viele Möglichkeiten, die logistische Regression zu konfigurieren, diese sind unter https://spark.apache.org/docs/latest/api/python/pyspark.ml.html#pyspark.ml.classification.LogisticRegressi ausführlich erklärt.

```
In [126]: lr = LogisticRegression(maxIter=10, regParam=0.01)
```

Zum Schluss wird die Pipeline als ganzes definiert, indem einfach ein Pipeline-Objekt erzeugt wird, dem die vorher konfigurierten Schritte in der Reihenfolge, in der sie ausgeführt werden sollen, übergeben werden:

```
In [127]: pipeline = Pipeline(stages=[zipIndexer, zipEncoder, genderIndexer, genderEncoder, feat
```

Bei der Konfiguration der Pipeline gilt zu beachten, dass bis jetzt noch nichts ausgeführt wurde. Wir haben lediglich ein Objekt 'pipeline' definiert, welches bestimmte Ablaufschritte beschreibt.

Jede Pipeline hat zwei Methoden:

- 'fit' führt dazu, dass die Pipeline trainiert wird. Dies heisst, dass anhand der in dem Datensatz enthaltenen Daten die einzelnen Schritte kalibriert werden. Insbesondere wird im letzten Schritt die logistische Regression trainiert.
- 'transform' wendet das vorher ge-fittete Modell auf die selben oder neue Daten an. Somit kann man sehr einfach z.B. den Test-Datensatz scoren, um die Güte des Modells zu erhalten.

```
In [134]: # Dies führt die die Vorbereitung der Daten und die Schätzung des Modells aus
          model = pipeline.fit(train)
```

```
---------------------------------------------------------------------------
Py4JJavaError                             Traceback (most recent call last)

<ipython-input-134-6b100d257325> in <module>()
      1 # Dies führt die die Vorbereitung der Daten und die Schätzung des Modells aus
----> 2 model = pipeline.fit(train)


/usr/local/spark/python/pyspark/ml/base.py in fit(self, dataset, params)
     62                 return self.copy(params)._fit(dataset)
     63             else:
---> 64                 return self._fit(dataset)
     65         else:
     66             raise ValueError("Params must be either a param map or a list/tuple of p


/usr/local/spark/python/pyspark/ml/pipeline.py in _fit(self, dataset)
```

```
    109                     transformers.append(model)
    110                     if i < indexOfLastEstimator:
--> 111                         dataset = model.transform(dataset)
    112             else:
    113                 transformers.append(stage)


    /usr/local/spark/python/pyspark/ml/base.py in transform(self, dataset, params)
    103                 return self.copy(params)._transform(dataset)
    104             else:
--> 105                 return self._transform(dataset)
    106         else:
    107             raise ValueError("Params must be a param map but got %s." % type(params)


    /usr/local/spark/python/pyspark/ml/wrapper.py in _transform(self, dataset)
    250     def _transform(self, dataset):
    251         self._transfer_params_to_java()
--> 252         return DataFrame(self._java_obj.transform(dataset._jdf), dataset.sql_ctx)
    253
    254


    /usr/local/spark/python/lib/py4j-0.10.4-src.zip/py4j/java_gateway.py in __call__(self, *
   1131         answer = self.gateway_client.send_command(command)
   1132         return_value = get_return_value(
-> 1133             answer, self.gateway_client, self.target_id, self.name)
   1134
   1135         for temp_arg in temp_args:


    /usr/local/spark/python/pyspark/sql/utils.py in deco(*a, **kw)
     61     def deco(*a, **kw):
     62         try:
---> 63             return f(*a, **kw)
     64         except py4j.protocol.Py4JJavaError as e:
     65             s = e.java_exception.toString()


    /usr/local/spark/python/lib/py4j-0.10.4-src.zip/py4j/protocol.py in get_return_value(ans
    317                 raise Py4JJavaError(
    318                     "An error occurred while calling {0}{1}{2}.\n".
--> 319                     format(target_id, ".", name), value)
    320             else:
    321                 raise Py4JError(


    Py4JJavaError: An error occurred while calling o660.transform.
```

```
: java.lang.NullPointerException
        at org.apache.spark.sql.types.Metadata$.org$apache$spark$sql$types$Metadata$$hash(Me
        at org.apache.spark.sql.types.Metadata$$anonfun$org$apache$spark$sql$types$Metadata$
        at org.apache.spark.sql.types.Metadata$$anonfun$org$apache$spark$sql$types$Metadata$
        at scala.collection.TraversableLike$$anonfun$map$1.apply(TraversableLike.scala:234)
        at scala.collection.TraversableLike$$anonfun$map$1.apply(TraversableLike.scala:234)
        at scala.collection.IndexedSeqOptimized$class.foreach(IndexedSeqOptimized.scala:33)
        at scala.collection.mutable.WrappedArray.foreach(WrappedArray.scala:35)
        at scala.collection.TraversableLike$class.map(TraversableLike.scala:234)
        at scala.collection.AbstractTraversable.map(Traversable.scala:104)
        at org.apache.spark.sql.types.Metadata$.org$apache$spark$sql$types$Metadata$$hash(Me
        at org.apache.spark.sql.types.Metadata$$anonfun$org$apache$spark$sql$types$Metadata$
        at org.apache.spark.sql.types.Metadata$$anonfun$org$apache$spark$sql$types$Metadata$
        at scala.collection.MapLike$MappedValues$$anonfun$foreach$3.apply(MapLike.scala:245)
        at scala.collection.MapLike$MappedValues$$anonfun$foreach$3.apply(MapLike.scala:245)
        at scala.collection.TraversableLike$WithFilter$$anonfun$foreach$1.apply(TraversableL
        at scala.collection.immutable.Map$Map3.foreach(Map.scala:161)
        at scala.collection.TraversableLike$WithFilter.foreach(TraversableLike.scala:732)
        at scala.collection.MapLike$MappedValues.foreach(MapLike.scala:245)
        at scala.util.hashing.MurmurHash3.unorderedHash(MurmurHash3.scala:91)
        at scala.util.hashing.MurmurHash3$.mapHash(MurmurHash3.scala:222)
        at scala.collection.GenMapLike$class.hashCode(GenMapLike.scala:35)
        at scala.collection.AbstractMap.hashCode(Map.scala:59)
        at scala.runtime.ScalaRunTime$.hash(ScalaRunTime.scala:206)
        at org.apache.spark.sql.types.Metadata$.org$apache$spark$sql$types$Metadata$$hash(Me
        at org.apache.spark.sql.types.Metadata$$anonfun$org$apache$spark$sql$types$Metadata$
        at org.apache.spark.sql.types.Metadata$$anonfun$org$apache$spark$sql$types$Metadata$
        at scala.collection.MapLike$MappedValues$$anonfun$foreach$3.apply(MapLike.scala:245)
        at scala.collection.MapLike$MappedValues$$anonfun$foreach$3.apply(MapLike.scala:245)
        at scala.collection.TraversableLike$WithFilter$$anonfun$foreach$1.apply(TraversableL
        at scala.collection.immutable.Map$Map1.foreach(Map.scala:116)
        at scala.collection.TraversableLike$WithFilter.foreach(TraversableLike.scala:732)
        at scala.collection.MapLike$MappedValues.foreach(MapLike.scala:245)
        at scala.util.hashing.MurmurHash3.unorderedHash(MurmurHash3.scala:91)
        at scala.util.hashing.MurmurHash3$.mapHash(MurmurHash3.scala:222)
        at scala.collection.GenMapLike$class.hashCode(GenMapLike.scala:35)
        at scala.collection.AbstractMap.hashCode(Map.scala:59)
        at scala.runtime.ScalaRunTime$.hash(ScalaRunTime.scala:206)
        at org.apache.spark.sql.types.Metadata$.org$apache$spark$sql$types$Metadata$$hash(Me
        at org.apache.spark.sql.types.Metadata._hashCode$lzycompute(Metadata.scala:107)
        at org.apache.spark.sql.types.Metadata._hashCode(Metadata.scala:107)
        at org.apache.spark.sql.types.Metadata.hashCode(Metadata.scala:108)
        at org.apache.spark.sql.catalyst.expressions.AttributeReference.hashCode(namedExpres
        at scala.runtime.ScalaRunTime$.hash(ScalaRunTime.scala:206)
        at scala.collection.immutable.HashSet.elemHashCode(HashSet.scala:177)
        at scala.collection.immutable.HashSet.computeHash(HashSet.scala:186)
        at scala.collection.immutable.HashSet.$plus(HashSet.scala:84)
        at scala.collection.immutable.HashSet.$plus(HashSet.scala:35)
```

```
at scala.collection.mutable.SetBuilder.$plus$eq(SetBuilder.scala:22)
at scala.collection.mutable.SetBuilder.$plus$eq(SetBuilder.scala:20)
at scala.collection.generic.Growable$class.loop$1(Growable.scala:53)
at scala.collection.generic.Growable$class.$plus$plus$eq(Growable.scala:57)
at scala.collection.mutable.SetBuilder.$plus$plus$eq(SetBuilder.scala:20)
at scala.collection.TraversableLike$class.to(TraversableLike.scala:590)
at scala.collection.AbstractTraversable.to(Traversable.scala:104)
at scala.collection.TraversableOnce$class.toSet(TraversableOnce.scala:304)
at scala.collection.AbstractTraversable.toSet(Traversable.scala:104)
at org.apache.spark.sql.catalyst.trees.TreeNode.containsChild$lzycompute(TreeNode.sc
at org.apache.spark.sql.catalyst.trees.TreeNode.containsChild(TreeNode.scala:89)
at org.apache.spark.sql.catalyst.trees.TreeNode$$anonfun$5$$anonfun$apply$11.apply(T
at scala.collection.TraversableLike$$anonfun$map$1.apply(TraversableLike.scala:234)
at scala.collection.TraversableLike$$anonfun$map$1.apply(TraversableLike.scala:234)
at scala.collection.immutable.List.foreach(List.scala:381)
at scala.collection.TraversableLike$class.map(TraversableLike.scala:234)
at scala.collection.immutable.List.map(List.scala:285)
at org.apache.spark.sql.catalyst.trees.TreeNode$$anonfun$5.apply(TreeNode.scala:358)
at org.apache.spark.sql.catalyst.trees.TreeNode.mapProductIterator(TreeNode.scala:18
at org.apache.spark.sql.catalyst.trees.TreeNode.transformChildren(TreeNode.scala:329
at org.apache.spark.sql.catalyst.trees.TreeNode.transformDown(TreeNode.scala:295)
at org.apache.spark.sql.catalyst.plans.QueryPlan.transformExpressionDown$1(QueryPlan
at org.apache.spark.sql.catalyst.plans.QueryPlan.org$apache$spark$sql$catalyst$plans
at org.apache.spark.sql.catalyst.plans.QueryPlan$$anonfun$6.apply(QueryPlan.scala:26
at org.apache.spark.sql.catalyst.trees.TreeNode.mapProductIterator(TreeNode.scala:18
at org.apache.spark.sql.catalyst.plans.QueryPlan.transformExpressionsDown(QueryPlan.
at org.apache.spark.sql.catalyst.plans.QueryPlan.transformExpressions(QueryPlan.scal
at org.apache.spark.sql.catalyst.analysis.Analyzer$ResolveDeserializer$$anonfun$appl
at org.apache.spark.sql.catalyst.analysis.Analyzer$ResolveDeserializer$$anonfun$appl
at org.apache.spark.sql.catalyst.plans.logical.LogicalPlan$$anonfun$resolveOperators
at org.apache.spark.sql.catalyst.plans.logical.LogicalPlan$$anonfun$resolveOperators
at org.apache.spark.sql.catalyst.trees.CurrentOrigin$.withOrigin(TreeNode.scala:70)
at org.apache.spark.sql.catalyst.plans.logical.LogicalPlan.resolveOperators(LogicalP
at org.apache.spark.sql.catalyst.analysis.Analyzer$ResolveDeserializer$.apply(Analyz
at org.apache.spark.sql.catalyst.analysis.Analyzer$ResolveDeserializer$.apply(Analyz
at org.apache.spark.sql.catalyst.rules.RuleExecutor$$anonfun$execute$1$$anonfun$appl
at org.apache.spark.sql.catalyst.rules.RuleExecutor$$anonfun$execute$1$$anonfun$appl
at scala.collection.LinearSeqOptimized$class.foldLeft(LinearSeqOptimized.scala:124)
at scala.collection.immutable.List.foldLeft(List.scala:84)
at org.apache.spark.sql.catalyst.rules.RuleExecutor$$anonfun$execute$1.apply(RuleExe
at org.apache.spark.sql.catalyst.rules.RuleExecutor$$anonfun$execute$1.apply(RuleExe
at scala.collection.immutable.List.foreach(List.scala:381)
at org.apache.spark.sql.catalyst.rules.RuleExecutor.execute(RuleExecutor.scala:74)
at org.apache.spark.sql.catalyst.encoders.ExpressionEncoder.resolveAndBind(Expressio
at org.apache.spark.sql.Dataset.<init>(Dataset.scala:209)
at org.apache.spark.sql.Dataset$.ofRows(Dataset.scala:64)
at org.apache.spark.sql.Dataset.org$apache$spark$sql$Dataset$$withPlan(Dataset.scala
at org.apache.spark.sql.Dataset.select(Dataset.scala:1121)
```

```
at org.apache.spark.ml.feature.StringIndexerModel.transform(StringIndexer.scala:185)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java
at java.lang.reflect.Method.invoke(Method.java:498)
at py4j.reflection.MethodInvoker.invoke(MethodInvoker.java:244)
at py4j.reflection.ReflectionEngine.invoke(ReflectionEngine.java:357)
at py4j.Gateway.invoke(Gateway.java:280)
at py4j.commands.AbstractCommand.invokeMethod(AbstractCommand.java:132)
at py4j.commands.CallCommand.execute(CallCommand.java:79)
at py4j.GatewayConnection.run(GatewayConnection.java:214)
at java.lang.Thread.run(Thread.java:748)
```

---

7 Schritt 5: Interpretation des Modells

Das Ergebnis der logistischen Regression ist eine Gleichung, die für jedes in das Modell eingangene Attribut ein Gewicht spezifiziert.

Dabei zeigt ein positiver Koeeffizient an, dass die Kaufwahrscheinlichkeit von diesem Attribut positiv beeinflusst wird. Ist der Koeffizient negativ, so beinflusst dieses Attribut die Kaufwahrscheinlichkeit negativ.

7.1 5.1. Interpretieren der Koeffizienten

Die Koeffizienten kann man direkt aus den geschätzen Modell-Objekt abrufen:

```
In [147]: # Das Modell extrahieren
          lastStage = len(model.stages)
          lrmod = model.stages[lastStage-1]

In [128]: lrmod.coefficients

Out[128]: DenseVector([0.0291, -0.0001, -0.0288, -0.051, 0.2798, -0.0645, -0.0071, 0.0198, 0.012
```

Dieser Vektor hilft einem erst einmal noch nicht so viel weiter. Für die Interpretation der Ergebnisse müssen in dem folgenden Datensatz die Indizes der kategorialen Variablen wieder auf die ursprünglichen Kategorien zurück gemappt werden.

Dafür macht es Sinn, die ursprünglichen Attribute mit ihren StringIndexes zu gruppieren, so dass man die Attribute dem Ergebnis zuordnen kann.

Der groSSe Vorteil einer Pipeline ist, dass das Bewerten von neuen Datensätzen mit dem Modell nun sehr einfach ist. Man führt einfach die Transform-Methode für den gewünschten Datensatz aus. Hier der Datensatz 'train'. Mit Hilfe der 'transform'-Methode können wir alle Attribute für den Trainingsdatensatz generieren, um die StringIndexes mit den ursprünglichen Werten zu verbinden.

```
In [151]: result_train = model.transform(train)
```

Dies ist die Zuordnungstabelle für die Zipcodes

```
In [152]: result_train.groupBy("zipIndex", "zip1").count().toPandas()
```

```
Out[152]:    zipIndex zip1  count
          0  3.0       8    1040
          1  4.0       7    1035
          2  2.0       2    1080
          3  6.0       3    1022
          4  1.0       4    1082
          5  8.0       0    1
          6  5.0       6    1032
          7  7.0       5    1015
          8  0.0       1    1109
```

Dies ist die Zuordnungstabelle für das Geschlecht

```
In [127]: result_train.groupBy("genderIndex", "Customer_gender").count().toPandas()
```

```
Out[127]:    genderIndex Customer_gender  count
          0          0.0            male   4276
          1          1.0          female   4140
```

Im folgenden schauen wir uns die erste Zeile des Datensatzes an, damit wir einfacher die Koeffizienten den Attributen zu ordnen können. Wichtig für die Zuordnung ist die Reihenfolge der Werte im dem Feld 'features'.

Wie man sieht, ist das erste Feature das Alter und das zweite Feature recency_1year. Danach kommen die Felder für den Zipcode und das Geschlecht.

```
In [154]: pandas.set_option('display.max_colwidth', -1) #nur mit dieser Option wird alles angeze
          result_train.select("*").drop("rawPrediction", "probability", "prediction"
                                        ).limit(1).toPandas()
```

```
Out[154]:    customer_id Customer_gender zip1  current_age  recency_1year  label  \
          0  45413558     female          1    55           9999.0         0

             zipIndex                                    zipVec  genderIndex genderVec  \
          0  0.0       (1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)  1.0          (0.0)

                                                                 features
          0  (55.0, 9999.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
```

Für die korrekte Interpretation muss man nun noch wissen, dass das One-Hot-Encoding immer den Wert mit dem Index 0 als die Basiskategorie annimmt. Das bedeutet, dass für diesen Attribut-Wert der Koeeffizient per Definition 0 ist und nicht angezeigt wird. Das heiSSt, dass es für den Zipcode in diesem Fall nur 8 Parameter gibt, obwohl es 9 Ausprägungen gibt, und dass es für das Geschlecht nur einen Parameter gibt, der für das Attribut "weiblich" steht.

Insgesamt sollte es also 11 Parameter geben: current_age, recency_1year, 8x zip1 und 1x Customer_gender.

```
In [156]: len(lrmod.coefficients)
```

```
Out[156]: 11
```

Dies scheint korrekt zu sein, schauen wir uns die Koeffizienten einmal an:

```
In [128]: lrmod.coefficients
```

```
Out[128]: DenseVector([0.0291, -0.0001, -0.0288, -0.051, 0.2798, -0.0645, -0.0071, 0.0198, 0.012
```

Für die oben angegebenen Koeffizienten könnte die Interpretation wie folgt sein:

```
- Das Alter erhöht die Kaufwahrscheinlichkeit (Koeffizient 0.0291)
- Eine längere Recency senkt die Kaufwahrscheinlichkeit (-0.0001)
- PLZ: Gegenüber der Basis-PLZ 1 wirken die PLZ 4, 2, 7 und 0 negativ und die PLZ 8, 3 und 5 pos
- Geschlecht: Frauen haben eine niedrigere Kaufwahrscheinlichkeit als Männer.
```

7.2 5.2 Anschauen der Scores für das Traingsset

Wir können uns nun die Bewertungen anschauen, diese sind in der Spalte 'probability' zu finden. Für jeden Kunden werden zwei Werte berechnet, der erste Wert ist Wahrscheinlichkeit für den Nicht-Kauf, der zweite ist die Wahrscheinlichkeit für den Kauf.

```
In [146]: result_train.select(
              "Customer_id",
              "Customer_gender",
              "Zip1",
              "recency_1year",
              "probability"
          ).limit(5).toPandas()
```

```
Out[146]:    Customer_id Customer_gender Zip1  recency_1year  \
          0  45413558     female          1    9999.0
          1  45413786     female          2    9999.0
          2  45413902     male            8    9999.0
          3  45413916     male            8    9999.0
          4  45414329     male            8    9999.0

                                  probability
          0  [0.982889393563, 0.0171106064365]
          1  [0.986935834237, 0.0130641657629]
          2  [0.992957898576, 0.0070421014237]
          3  [0.982306171558, 0.0176938284421]
          4  [0.983762657751, 0.0162373422494]
```

Durch Sortierung nach probability können wir nun den besten Kunden im Trainingsset herausfinden. Hierbei machen wir uns zu nutze, dass die Wahrscheinlichkeit für den Nicht-Kauf die erste Zahl in dem probability-Vektor ist. Daher sortieren wir aufsteigend, d.h., der Kunde mit der kleinsten Wahrscheinlichkeit für den Nicht-Kauf steht oben. Dies ist gleichzeitig der Kunde mit der grösSten Wahrscheinlichkeit für den Kauf.

```
In [158]: result_train.select( "Customer_id",
              "Customer_gender",
              "Zip1",
              "recency_1year",
              "probability").orderBy("probability").limit(5).toPandas()
```

```
Out[158]:    Customer_id Customer_gender Zip1  recency_1year  \
          0  45420598     male           2    6.483871
          1  45423300     male           2    3.354839
          2  45407749     female         6    0.000000
          3  45407915     male           3    7.064516
          4  45421702     female         2    2.903226

                                probability
          0  [0.757037178035, 0.242962821965]
          1  [0.801899378504, 0.198100621496]
          2  [0.825324988849, 0.174675011151]
          3  [0.841050876501, 0.158949123499]
          4  [0.860268730997, 0.139731269003]
```

---

8 Schritt 6: Auswertung des Modells (Train vs. Test)

Die Auswertung muss nicht von angepasst werden, dieser Block sollte unverändert übernommen werden, damit die Auswertung für alle Gruppen gleich ist. Beachten Sie aber bitte den Erweiterungshinweis am Ende des Schritts, dieser sollte auf jeden Fall durchgeführt werden!

Der Kernansatz für die Bewertung ist die Performance des Test-Sets. Die Vorhersage-Performance des Test-Sets sollte nicht deutlich unter der des Trainings-Sets liegen. Ist dies der Fall, so hat der Algorithmus höchstwahrscheinlich keine generellen Zusammenhänge gelernt, sondern nur den speziellen Trainings-Datensatz. Dieses Problem nennt man Overfitting.

Dieses Problem ist von einem der herausragendsten Data-Science-Forscher der Welt, Andrew Ng in diesem Video erläutert: https://www.coursera.org/learn/machine-learning/lecture/ACpTQ/the-problem-of-overfitting (Kann als Preview des Kurses kostenfrei angehesen werden)

Die Metrik, die wir für diesen Kurs nutzen werden ist die so genannte ROC-Kurve. Diese Kurve zeigt an, wie gut das Modell in der Lage ist, die "guten" von den "schlechten" Kunden zu trennen.

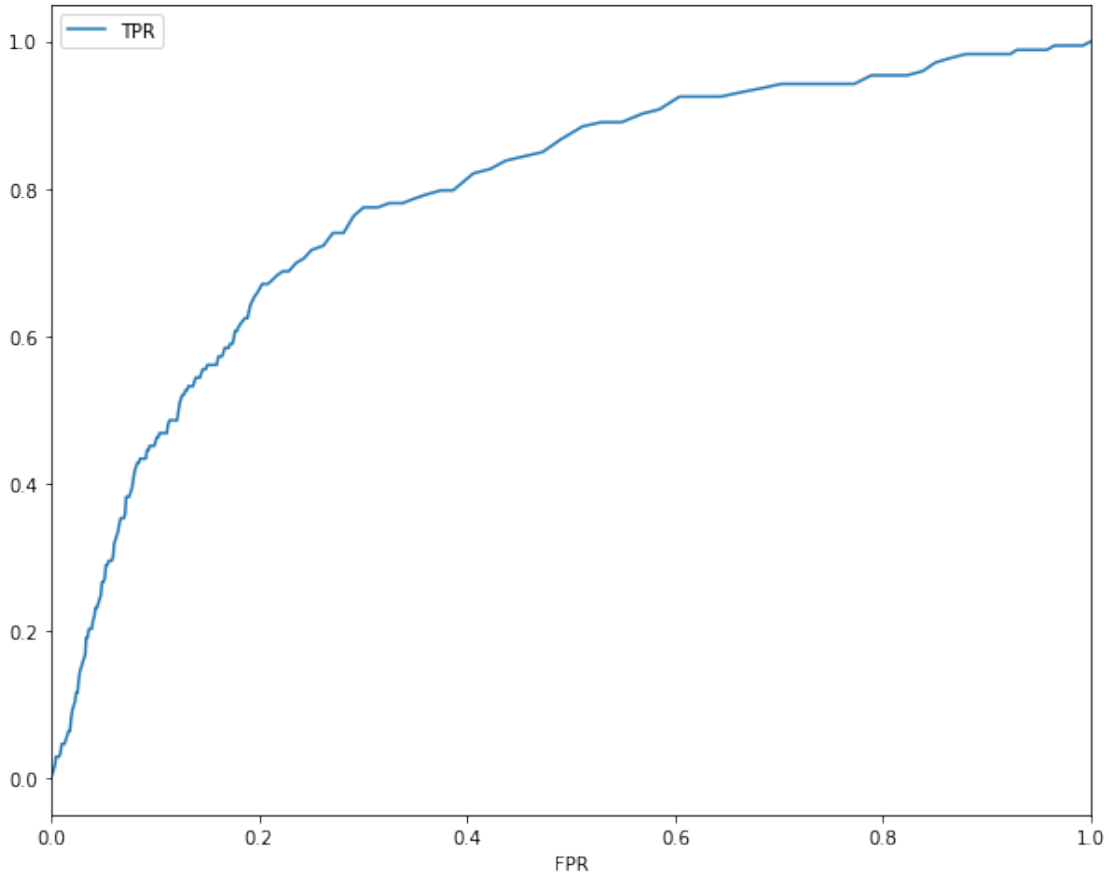
Je weiter die Kurve nach oben links wandert, um so besser ist die Prognose. Wenn die Kurve genau auf der Diagonalen liegt, so ist die Prognose nur genauso gut wie eine zufällige Entscheidung, z.B. ein Münzwurf und hat dann überhaupt keine Vorhersagekraft. Sollte die Kurve unterhalb der Diagonallinie liegen, so ist die Prognose sogar schlechter als der Zufall.

8.0.1 Dies ist die ROC-Kurve für den Trainingsdatensatz

```
In [148]: trainingSummary = lrmod.summary
```

```
In [108]: trainingSummary.roc.toPandas().plot(x="FPR", y="TPR", figsize=[10,8])
```

```
Out[108]: <matplotlib.axes._subplots.AxesSubplot at 0x7f180185d4e0>
```




```
In [109]: print("Area-under-the-ROC-Curve-Wert im Training: " + str(round(trainingSummary.areaUn

Area-under-the-ROC-Curve-Wert im Training: 0.7878
```

8.0.2 Evaluation des Testsets

8.0.3 Dies ist die Auswertung des Test-Sets

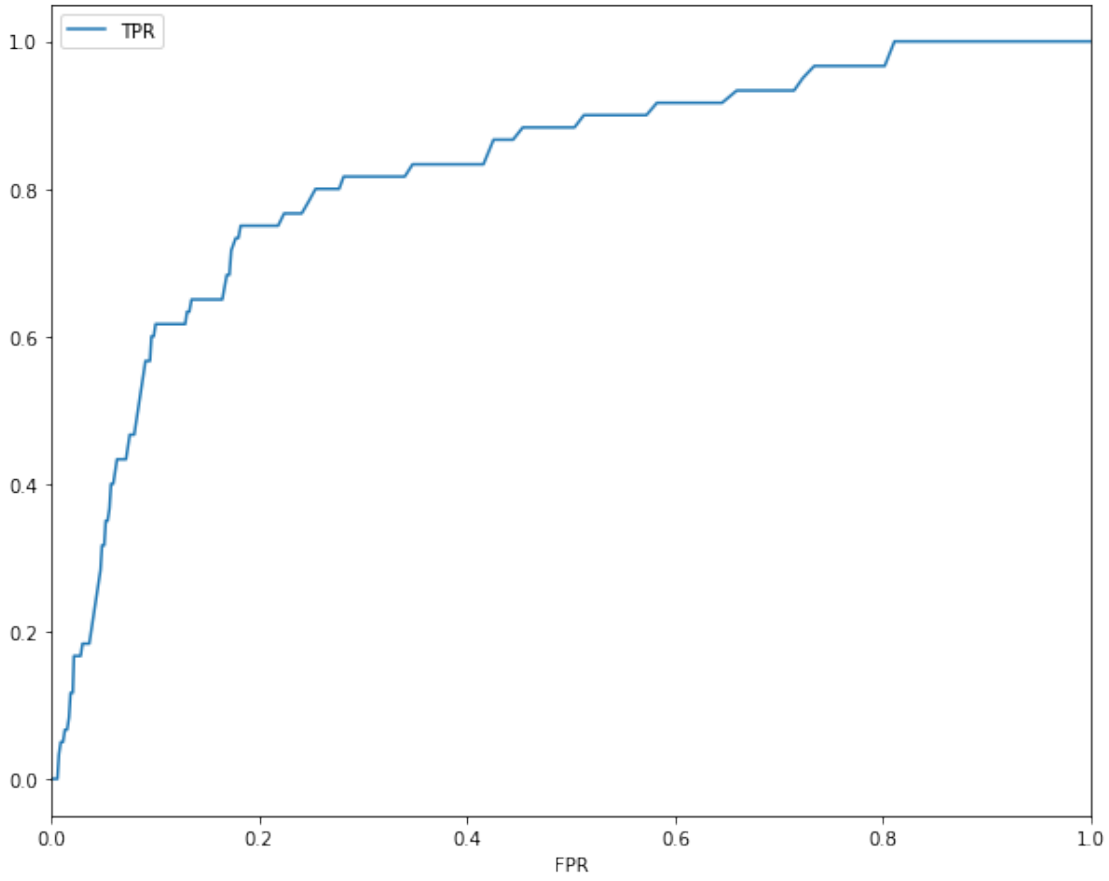
```
In [110]: # predicting test data set
          scores = model.transform(test)

In [112]: # Dies ist ein kleiner Workaround, um trotz der Pipeline auf die Evaluation-Funktionen
          testSummary = lrmod.evaluate(scores.drop("prediction", "rawPrediction", "probability")
```

8.0.4 Dies ist die ROC-Kurve für den Testdatensatz

```
In [115]: testSummary.roc.toPandas().plot(x="FPR", y="TPR", figsize=[10,8])
```

Out[115]: <matplotlib.axes._subplots.AxesSubplot at 0x7f1800101c18>




```
In [119]: print("Der 'Area under the curve'-Wert des Test-Sets ist: {}, verglichen mit dem AUC-W
              round(testSummary.areaUnderROC,4),
              round(trainingSummary.areaUnderROC,4)
                  )
               )
          print("Somit liegt der Wert im Test-Set um {}% unter der Wert im Trainings-Set.".forma
                  round(((trainingSummary.areaUnderROC-testSummary.areaUnderROC)/trainingSummary
                  )
               )
```

```
Der 'Area under the curve'-Wert des Test-Sets ist: 0.8233, verglichen mit dem AUC-Wert des Train
Somit liegt der Wert im Test-Set um -4.5% unter der Wert im Trainings-Set.
```

8.0.5 *Erweiterungshinweis 8*

Bitte ermittelt hier den besten und den schlechtesten Kunden im Test-Set nach Score und interpretiert dessen Attribute.