

BDA, Praktikumsbericht 1

Gruppe mi6xc: Alexander Kniesz, Maximilian Neudert, Oskar Rudolf

Aufgabe 1

Zuerst haben wir ein gemeinsames Notebook [bericht1](#) auf Zeppelin mit Ownern aller Gruppenmitgliedern erstellt, auf dem wir gemeinsam arbeiten können.

Wir haben als ApplicationID `app-20190425183601-0341` erhalten und uns auf `http://141.100.62.85:8080/` die Ressourcen angeschaut. Auffällig war, dass keine Kerne zugewiesen waren. Wenn man Testweise eine Endlosschleife mit PySpark ausgeführt hat, dann ging der Status auf Waiting. Wir sind von dem Monitor noch nicht ganz überzeugt. Der Status wirkt ziemlich träge. Aber man kann damit gut Applications abschießen die in Jobs festhängen.

Zuerst haben wir uns alle Million Song relevanten Tabellen ausgeben lassen:

```
%pyspark
spark.sql("show tables like 'msd10k*']").show(truncate=False)
```

```
+-----+-----+-----+
|database|tableName          |isTemporary|
+-----+-----+-----+
|default |msd10k_more_metadata|false      |
|default |msd10k_more_metadata_row|false     |
|default |msd10k_some_metadata |false      |
|default |msd10k_some_metadata_row|false     |
|default |msd10k_timbre        |false      |
|default |msd10k_timbre_row    |false      |
+-----+-----+-----+
```

Dann haben wir die Daten gesichtet:

```
%sql
select * from msd10k_timbre limit 100
select * from msd10k_some_metadata limit 100
select * from msd10k_more_metadata limit 100
```

| title | track_id | timbre_0 | timbre_1 | timbre_2 |
|--|------------------------|----------|----------|----------|
| Doppelgänger [Qliphothic Phantasmagoria] | TRABEFN128F92D92 5B | 21.613 | -136.784 | -105.838 |
| Doppelgänger [Qliphothic Phantasmagoria] | TRABEFN128F92D92 5B | 21.864 | -151.802 | -100.926 |
| Doppelgänger [Qliphothic Phantasmagoria] | TRABEFN128F92D92 5B | 20.972 | -156.087 | -94.076 |
| Doppelgänger [Qliphothic Phantasmagoria] | TRABEFN128F92D92 5B | 22.255 | -145.706 | -81.169 |

Wir haben vorerst geprüft, ob die Timbre überall gleich lang sind

```
%pyspark
s1 = 'TRAVHPV128F933E986'
s2 = 'TRAKXYJ128F42525ED'
def get_tdur(track_id):
    not_sql_df = spark.sql("select count(timbre_0) as val from msd10k_timbre where track_id = '{}'.format(track_id)")
    s_tcount = not_sql_df.collect()[0]['val']
    not_sql_df = spark.sql("select duration as dur from msd10k_more_metadata where track_id = '{}'.format(track_id)")
    s_duration = not_sql_df.collect()[0]['dur']
    timbre_duration = s_duration / s_tcount
    return timbre_duration

d1 = get_tdur(s1)
d2 = get_tdur(s2)

print(d2 - d1)
```

Wir haben **0.0299464126059322** als Ergebnis bekommen, was bedeutet, dass die Timbre nicht gleich lang sind.

Beispielhaft lassen wir uns für **duration** und **loudness** eine statistische Zusammenfassung mittels **describe()** geben:

```
%pyspark
df = spark.sql("select duration, loudness from msd10k_some_metadata")
df.describe().show()
```

```
+-----+-----+-----+
|summary|      duration|      loudness|
+-----+-----+-----+
|  count|      10000|      10000|
|   mean|238.50751842799997| -10.485668499999996|
|  stddev|114.13751356561322|   5.39978822917156|
|   min|       1.04444|      -51.643|
|   max|     1819.76771|       0.566|
+-----+-----+-----+
```

Beim Vergleich der Performance haben wir durch Sichtprüfung mehrerer runs einmal mit `describe` einmal mit Aggregationsfunktionen column based und row based verglichen und kamen zum Ergebnis, dass row based langsamer läuft.

```
%pyspark
from pyspark.sql import functions as F
df = spark.sql("select duration from msd1m_some_metadata")
df.describe().show()
```

```
+-----+-----+
|summary|      duration|
+-----+-----+
|  count|      1000053|
|   mean|249.5008840431487|
| stddev|126.2293871957265|
|   min|         0.31302|
|   max|      3034.90567|
+-----+-----+
```

Took 1 sec. Last updated by istmnneud at April 25 2019, 7:04:38 PM.

```
%pyspark
from pyspark.sql import functions as F
df = spark.sql("select duration from msd1m_some_metadata_row")
df.describe().show()
```

```
+-----+-----+
|summary|      duration|
+-----+-----+
|  count|      1000053|
|   mean|249.5008840431487|
| stddev|126.2293871957265|
|   min|         0.31302|
|   max|      3034.90567|
+-----+-----+
```

Took 4 sec. Last updated by istmnneud at April 25 2019, 7:04:33 PM.

```
%pyspark
from pyspark.sql import functions as F
df = spark.sql("select duration from msd1m_some_metadata")
df.agg(F.min(df.duration),F.max(df.duration),F.avg(df.duration),F.sum(df.duration)).show()
```

SPARK JOB FINISHED

```
+-----+-----+-----+-----+
|min(duration)|max(duration)|  avg(duration)|  sum(duration)|
+-----+-----+-----+-----+
|      0.31302|    3034.90567|249.5008840431487|2.4951410759000298E8|
+-----+-----+-----+-----+
```

Took 1 sec. Last updated by istmnneud at April 25 2019, 7:01:59 PM.

```
%pyspark
from pyspark.sql import functions as F
df = spark.sql("select duration from msd1m_some_metadata_row")
df.agg(F.min(df.duration),F.max(df.duration),F.avg(df.duration),F.sum(df.duration)).show()
```

SPARK JOB FINISHED

```
+-----+-----+-----+-----+
|min(duration)|max(duration)|  avg(duration)|  sum(duration)|
```

```

max(qdraccon), max(qdraccon), ... max(qdraccon), ... max(qdraccon), ...
+-----+-----+-----+-----+-----+-----+
|      0.31302|    3034.90567|249.5008840431487|2.4951410759000298E8|
+-----+-----+-----+-----+-----+

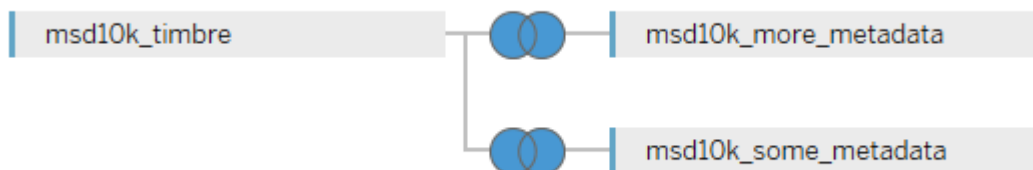
```

Took 5 sec Last updated by istmnneud at April 25 2019, 7:01:56 PM.





Aufgabe 2

a)

Wir haben die Joins nach folgendem Schema durchgeführt:



Beide Joins wurden über die Track Id durchgeführt:

| Join | | | ✕ |
|---|---|---|---|
|  |  |  |  |
| Inner | Left | Right | Full Outer |
| Data Source | | msd10k_more_metad... | |
| Track Id | = | Track Id (Msd10K ... | |
| Add new join clause | | | |

b)

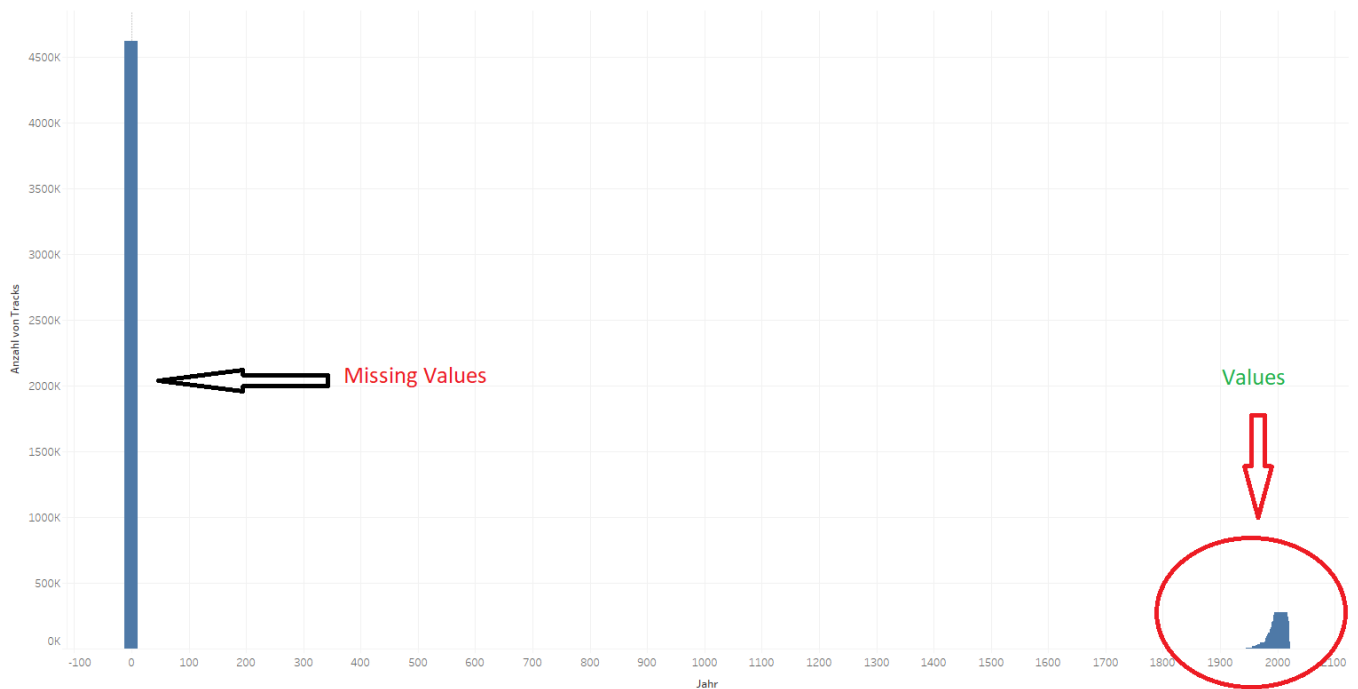
Skalierung

Die Daten werden von Tableau in zwei Kategorien eingeteilt: Dimensionen und Maßzahlen. Innerhalb der Variablen der Dimensions-Kategorie sind die kategorialen Merkmale (qualitativen), nach denen sich z.B. gut aggregieren lässt. Bei den Maßzahlen handelt es sich um metrisch Skalierte (quantitative) Variablen.

Missing Values

Obwohl es in Tableau möglich ist, sich fehlende Werte anzeigen zu lassen (z.B. über die folgende Darstellung), handelt es sich bei dem Tool eher um ein Visualisierungstool und die Analyse von Missings müsste für jede Variable einzeln mittels Grafik durchgeführt werden.

Fehlende Werte als 0 in Jahreszahlen



Für eine schnellere Analyse der Missing-Data haben wir uns mittels R einen schnellen Überblick verschafft:

```
require(data.table)

missing_names <- c("Variable", "NA_count", "empty_string_count", "0_count" )

setwd("C:\\Users\\rudol\\Documents\\AAA_Wichtig\\STUDIUM\\MSc. Data Science\\2.
Semester\\Big_Data_Analytics\\Datasets")

# Anzahl Missing Values im TimbreDatensatz:

timbres <- as.data.frame(fread("msd10k_timbre.tsv"))

timbre_missings <- data.frame (names(timbres))

timbre_missings <- cbind(timbre_missings, sapply(timbres, function(x)
sum(is.na(x))))
timbre_missings <- cbind(timbre_missings, sapply(timbres, function(x) sum(x=="")))
timbre_missings <- cbind(timbre_missings, sapply(timbres, function(x) sum(x==0)))

# Spaltennamen
names(timbre_missings) <- missing_names

# Anzahl Missing Values im MetaDatensatz (some + more enthalten viele
Redundanzen, daher hier nur "more"):

meta_data <- as.data.frame(fread("msd10k_more_metadata.tsv"))

# Anzahl Missing Values im TimbreDatensatz:

meta_data_missings <- data.frame (names(meta_data))
```

```

meta_data_missings <- cbind(meta_data_missings, sapply(meta_data, function(x)
sum(is.na(x))))
meta_data_missings <- cbind(meta_data_missings, sapply(meta_data, function(x)
sum(x=="")))
meta_data_missings <- cbind(meta_data_missings, sapply(meta_data, function(x)
sum(x==0)))

# Spaltennamen
names(meta_data_missings) <- missing_names

```

Hier ist das Ergebnis abgebildet:

| Variable | NA_count | empty_string_count | 0_count |
|----------|----------|--------------------|---------|
| V1 | 0 | 822 | 0 |
| V2 | 0 | 0 | 0 |
| V3 | 0 | 0 | 5010 |
| V4 | 0 | 0 | 51 |
| V5 | 0 | 0 | 64 |
| V6 | 0 | 0 | 106 |
| V7 | 0 | 0 | 105 |
| V8 | 0 | 0 | 91 |
| V9 | 0 | 0 | 112 |
| V10 | 0 | 0 | 172 |
| V11 | 0 | 0 | 167 |
| V12 | 0 | 0 | 197 |
| V13 | 0 | 0 | 255 |
| V14 | 0 | 0 | 178 |

Timbre_Daten:

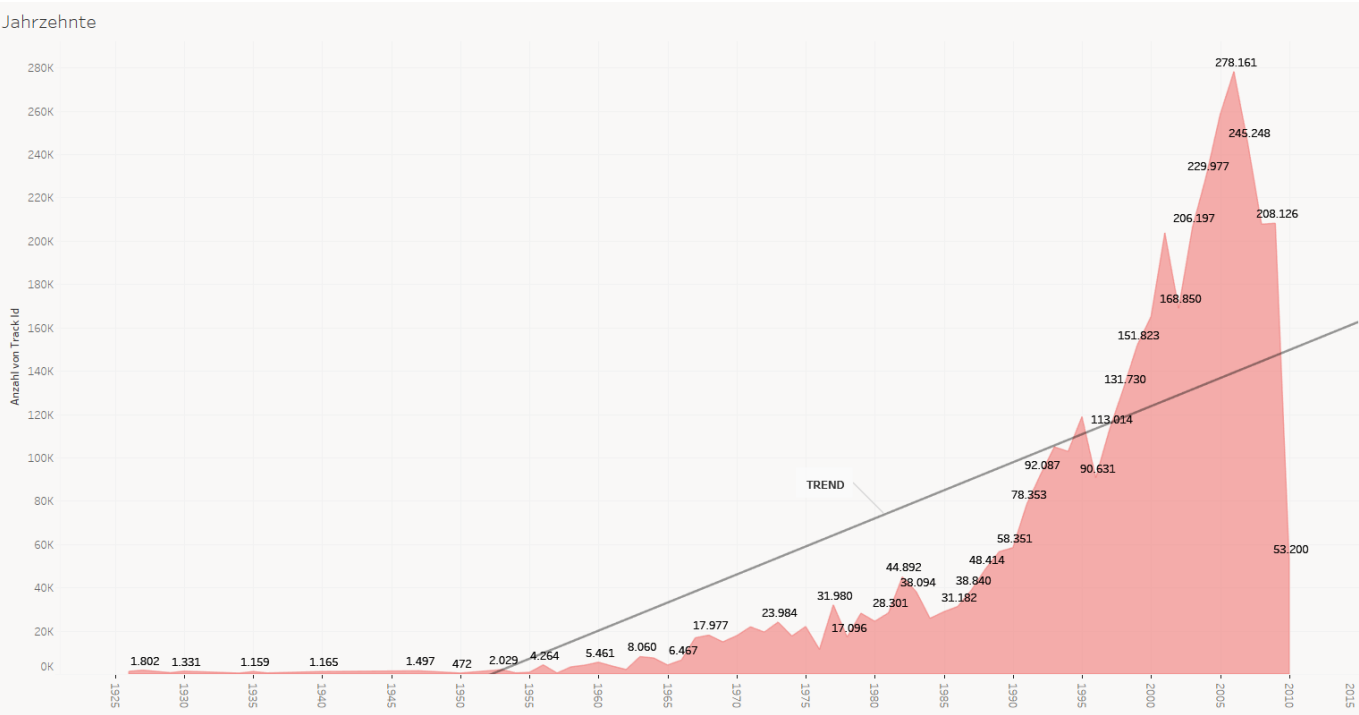
| Variable | NA_count | empty_string_count | 0_count |
|----------|----------|--------------------|---------|
| V1 | 0 | 0 | 0 |
| V2 | 0 | 1 | 0 |
| V3 | 0 | 0 | 0 |
| V4 | 0 | 0 | 0 |
| V5 | 0 | 0 | 25 |
| V6 | 0 | 0 | 5320 |
| V7 | 6258 | 0 | NA |
| V8 | 6258 | 0 | NA |
| V9 | 4 | 0 | NA |
| V10 | 4352 | 0 | NA |
| V11 | 0 | 0 | 0 |
| V12 | 0 | 0 | 0 |
| V13 | 0 | 0 | 0 |
| V14 | 0 | 0 | 1213 |
| V15 | 0 | 0 | 523 |
| V16 | 0 | 0 | 3089 |
| V17 | 0 | 0 | 277 |
| V18 | 0 | 0 | 3 |
| V19 | 0 | 0 | 2167 |

Meta_Daten:

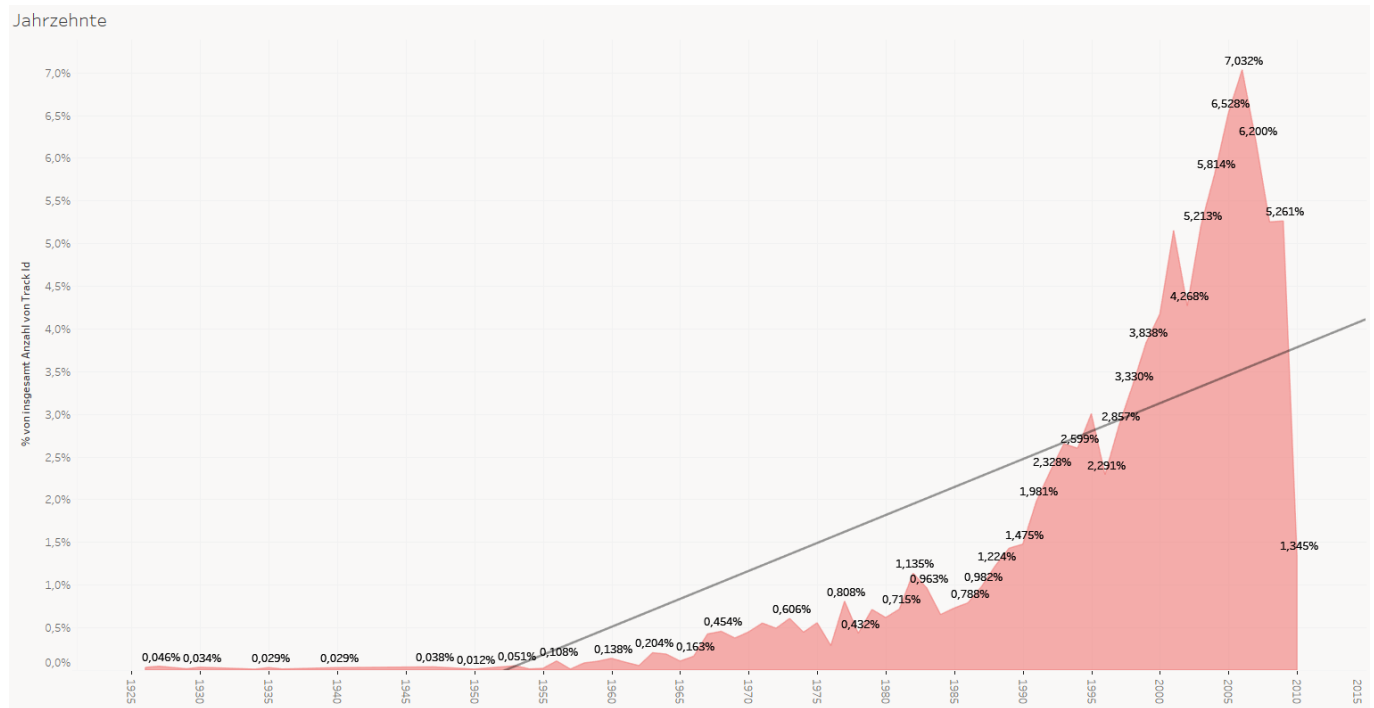
Diagramm mit Jahreszahlen

Nach herausfiltern der "überflüssigen" Nullwerte (Jahr==0) sind wir auf folgende Übersicht über die Jahrzehnte gekommen:

Insgesamt:



In Prozent:



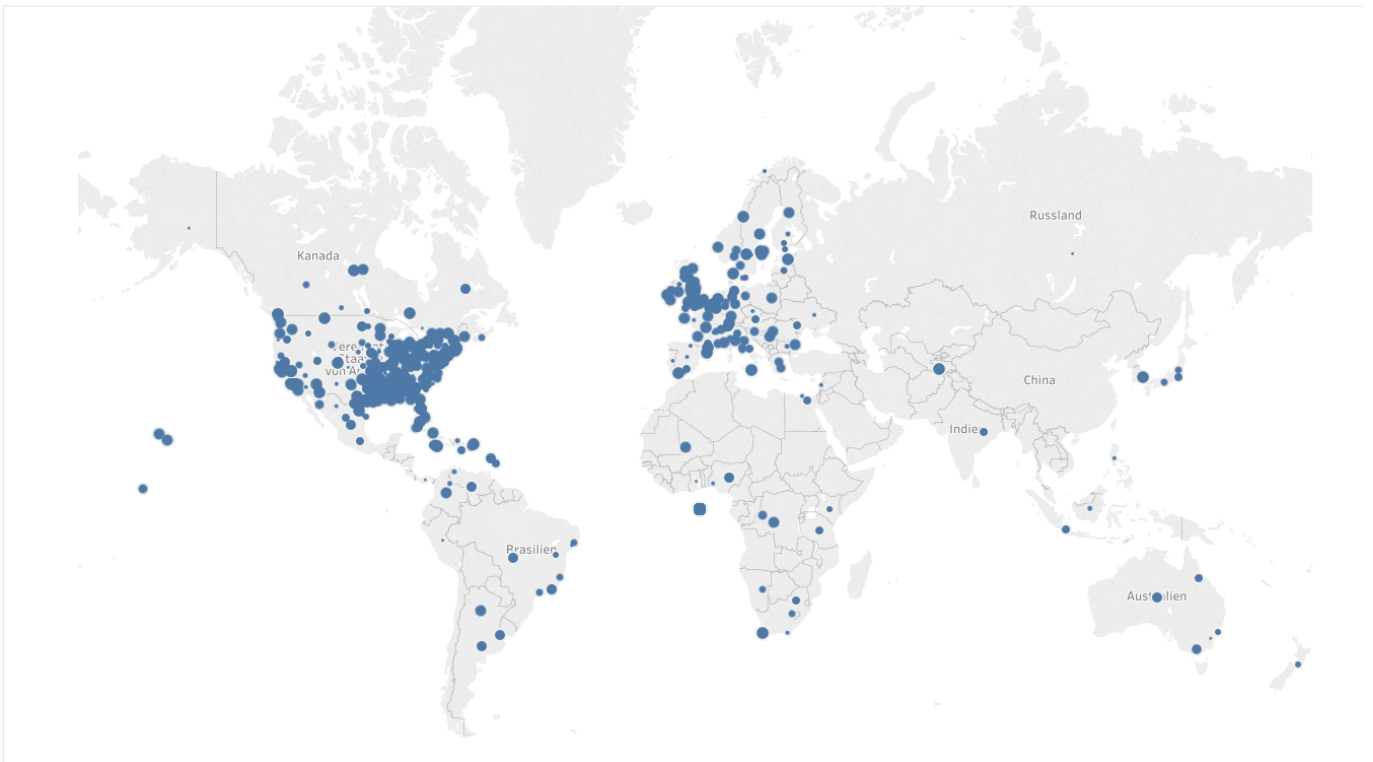
Weitere Fragestellungen

1. Woher kommen die meisten Künstler?
2. Sind die Lieder im Laufe der Zeit kürzer oder länger geworden?
3. Sind schnelle Songs beliebter als langsame Songs?

Zu 1.)

Wir sehen hier, dass viele Künstler aus den USA und Nord/West-Europa liegen. Auffällig ist, dass Asien (Russland, China, Indien) trotz hoher Bevölkerungszahl in diesen Daten fast gar nicht vertreten ist. Wurden hier eventuell nur englische Lieder in der Datenbank eingetragen?

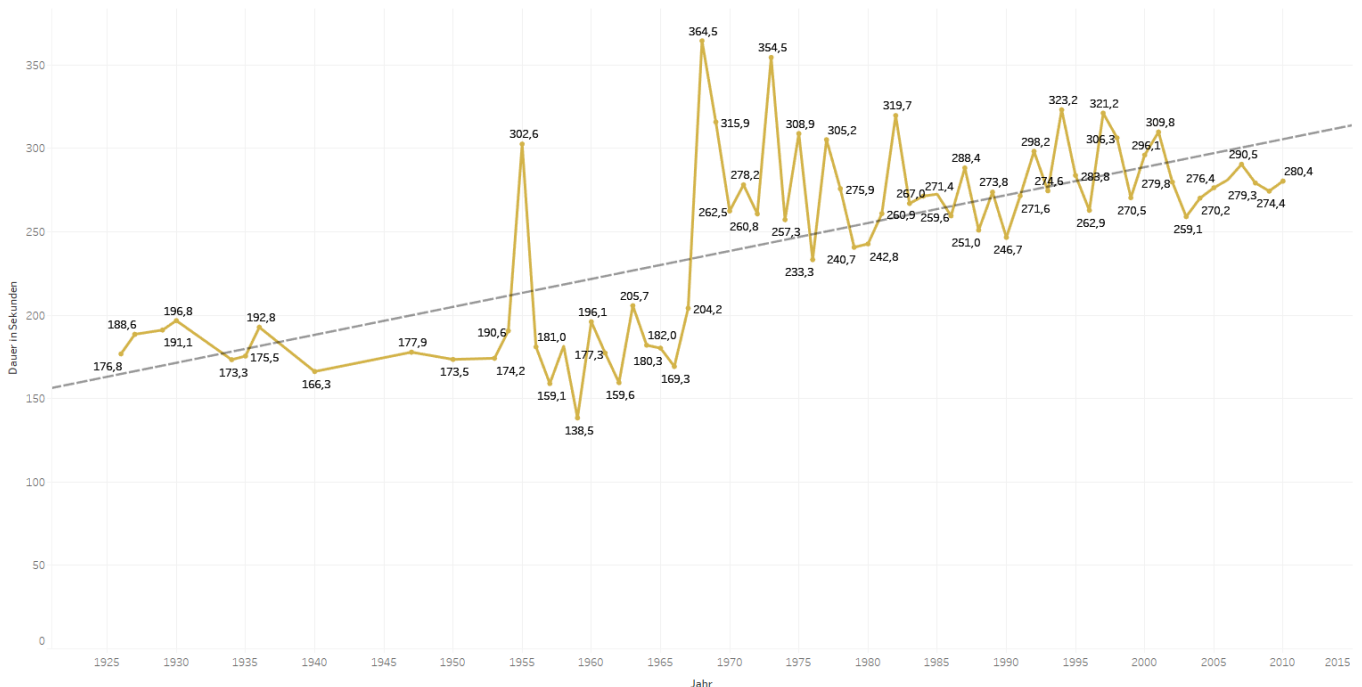
Künstler Herkunft



Zu 2.)

Tatsächlich scheint es, dass im Laufe der Zeit die (durchschnittliche) Länge der Lieder zugenommen hat, sich aber während der letzten Jahrzehnte etwas eingependelt hat bei ca. 280 Sekunden (4 Minuten, 40 Sekunden)

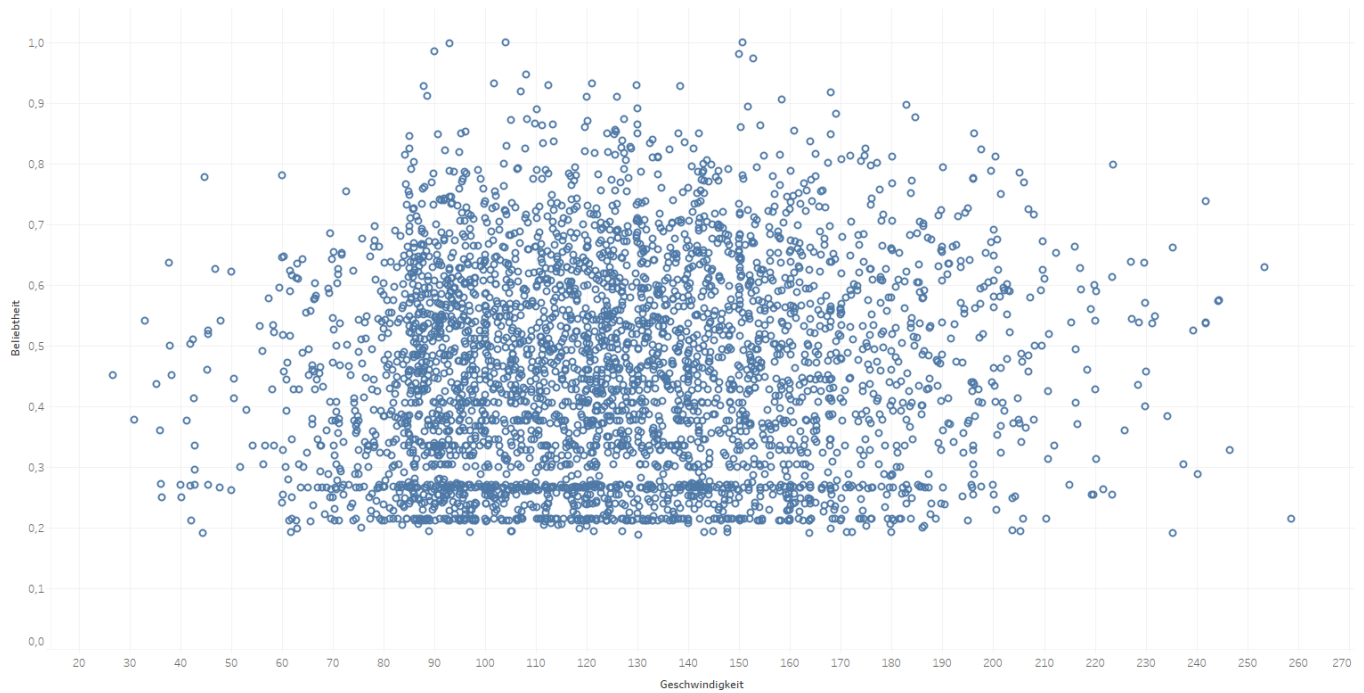
Liedlänge im Laufe der Jahre



Zu 3.)

Leider lässt sich die Frage nur schwer beantworten. Es scheint logisch, dass weder zu langsame, noch zu schnelle Songs zu den beliebtesten zählen. Eine Tendenz lässt sich eher nicht erkennen. Auf dieser Abbildung entspricht jeder Punkt einem Songtitel:

Schnell vs. Ruhig



Aufgabe 3

a)

Für das Binning haben wir 10 bins gewählt, diese mit PySpark erstellt und exemplarisch die Tabelle zeigen lassen. Wir haben festgestellt, dass der Bucketizer binning betreibt, indem dieser eine Spalte hinzufügt, in der die Zuordnung zu einem bin steht.

```
%pyspark

# Paketimport
from pyspark.ml.feature import Bucketizer as bt

# Übersicht über timbre_0 Werte
df = spark.sql("select track_id,timbre_0 from msd10k_timbre")
desc = df.select("timbre_0").describe()
desc.show()

# Parameter angeben
n_bins = 10

# Maximum/Minimum bestimmen
tmin = float(desc.collect()[3][1])
tmax = float(desc.collect()[4][1])

# Abstand zwischen Intervallen
delta = (tmax - tmin) / (n_bins - 1)

# Split-Intervalle festlegen
splits = [-float("inf")]
val = 0
for i in range(n_bins - 2):
    val += delta
    splits.append(val)
splits.append(float("inf"))

# Binning über alle Timbres
btz = bt(splits=splits, inputCol="timbre_0", outputCol="buckets")

# Bucketizer-Objekt in Tabelle konvertieren
df_b = btz.setHandleInvalid("keep").transform(df)

# Speichern als neue Tabelle
df_b.write.mode("overwrite").saveAsTable("mi6xc_bucketeddata")

# Erfolgskontrolle
df = spark.sql("SELECT * FROM mi6xc_bucketeddata LIMIT 5")
df.show()

# Alternative Speicherfunktionen
#df_b.createOrReplaceTempView("tmp_df_b")
#sqlContext.sql("create table mi6xc_bucketeddata as select * from tmp_df_b")
```

```
+-----+-----+
|summary|      timbre_0|
+-----+-----+
|  count|      8577406|
|   mean| 42.7280295816724|
| stddev|8.080462607982279|
|   min|           0.0|
|   max|          62.68|
+-----+-----+
```

```
+-----+-----+-----+
|      track_id|timbre_0|buckets|
+-----+-----+-----+
|TRABEFN128F92D925B| 21.613|    3.0|
|TRABEFN128F92D925B| 21.864|    3.0|
|TRABEFN128F92D925B| 20.972|    3.0|
|TRABEFN128F92D925B| 22.255|    3.0|
|TRABEFN128F92D925B| 21.871|    3.0|
```

Gespeichert haben wir die Tabelle dann als `mi6xc_bucketeddata` und zur Sicherheit den Speichervorgang überprüft.

```
%pyspark
df = spark.sql("select * from mi6xc_bucketeddata limit 5")
df.show()
```

```
+-----+-----+-----+-----+
|          title|          track_id|timbre_0|buckets|
+-----+-----+-----+-----+
|Doppelgoenger [Ql...|TRABEFN128F92D925B|  21.613|    2.0|
|Doppelgoenger [Ql...|TRABEFN128F92D925B|  21.864|    3.0|
|Doppelgoenger [Ql...|TRABEFN128F92D925B|  20.972|    2.0|
|Doppelgoenger [Ql...|TRABEFN128F92D925B|  22.255|    3.0|
|Doppelgoenger [Ql...|TRABEFN128F92D925B|   21.82|    3.0|
+-----+-----+-----+-----+
```

Anschließend haben wir die Pivotierung mittels Aggregation über die Anzahl durchgeführt und die Spalten danach in eine relative Maßzahl umgewandelt, was uns die Profil-Vektoren gibt.

```
%pyspark

# Paketimport
from pyspark.sql.functions import count, udf, struct

# Pivotieren
df = spark.sql("select * from mi6xc_bucketedata")
piv_df = df.groupBy('track_id').pivot('buckets').count()

# Absolute Häufigkeiten in Relative umrechnen // i==0 ist track_id
def recalc(x):
    a = [element if i==0 else 0 if element==None else int(element) for i,element in enumerate(x)]
    b = [element if i==0 else round(element/sum(a[1:len(a)]),2) for i,element in enumerate(a)]
    return b

# Spaltennamen
new_names = ['track_id', 'bin_0', 'bin_1', 'bin_2', 'bin_3', 'bin_4', 'bin_5', 'bin_6', 'bin_7', 'bin_8']

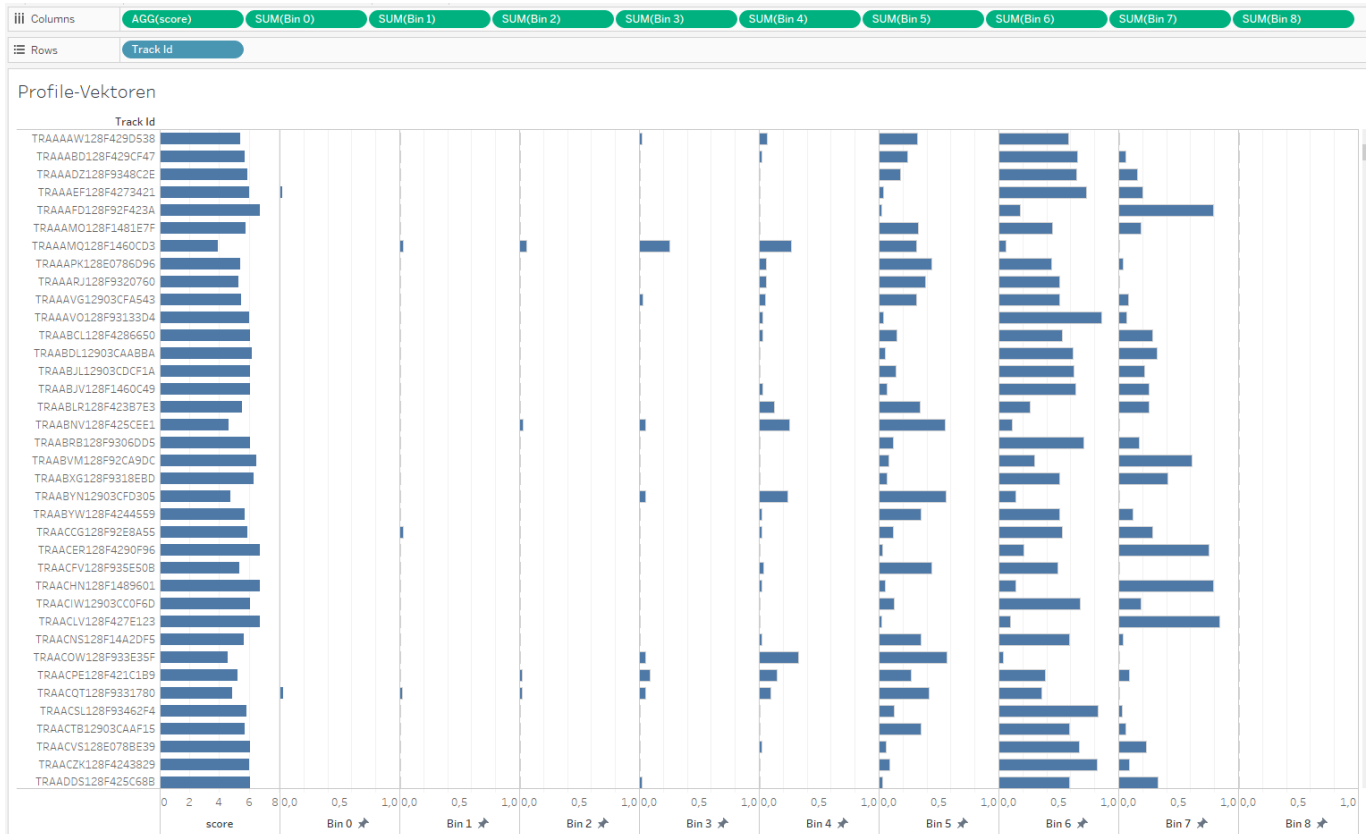
# Neuen DataFrame mit relativen Häufigkeiten erstellen: piv_df -> df_rel_freq
df_rel_freq = spark.createDataFrame(piv_df.rdd.map(recalc).collect(),new_names)
df_rel_freq.show()

df_rel_freq.write.mode("overwrite").saveAsTable("mi6xc_profilevectors")
```

| track_id | bin_0 | bin_1 | bin_2 | bin_3 | bin_4 | bin_5 | bin_6 | bin_7 | bin_8 |
|---------------------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| TRAZKDD12903CC8328 | 0.0 | 0.0 | 0.01 | 0.01 | 0.07 | 0.21 | 0.51 | 0.19 | 0.0 |
| TRALYIQ12903CEC83A | 0.0 | 0.0 | 0.0 | 0.0 | 0.03 | 0.32 | 0.6 | 0.04 | 0.0 |
| TRADTBA128F92CA48F | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.02 | 0.22 | 0.76 | 0.0 |
| TRAEDJW128F422B4CB | 0.0 | 0.01 | 0.02 | 0.07 | 0.24 | 0.56 | 0.11 | 0.0 | 0.0 |
| TRATPUS128F4271260 | 0.0 | 0.0 | 0.0 | 0.0 | 0.03 | 0.29 | 0.65 | 0.02 | 0.0 |
| TRABIOI12903CD889B | 0.0 | 0.0 | 0.0 | 0.0 | 0.01 | 0.02 | 0.12 | 0.84 | 0.0 |
| TRAZUKY128F426C8BC | 0.0 | 0.0 | 0.01 | 0.0 | 0.01 | 0.03 | 0.15 | 0.8 | 0.0 |
| TRBHBEB128F92CC0C6 | 0.0 | 0.0 | 0.0 | 0.01 | 0.15 | 0.54 | 0.29 | 0.0 | 0.0 |
| TRASOQC12903CFFED7 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.01 | 0.04 | 0.94 | 0.0 |
| TRAVHPV128F933E986 | 0.0 | 0.0 | 0.0 | 0.01 | 0.01 | 0.06 | 0.39 | 0.53 | 0.0 |
| TRALNOD128F4264AEB | 0.0 | 0.0 | 0.0 | 0.01 | 0.04 | 0.38 | 0.56 | 0.0 | 0.0 |
| TRAOVFR128F93275B4 | 0.01 | 0.0 | 0.02 | 0.04 | 0.16 | 0.32 | 0.42 | 0.03 | 0.0 |
| TRAXDLB128F423F8F8 | 0.0 | 0.0 | 0.01 | 0.0 | 0.01 | 0.12 | 0.76 | 0.09 | 0.0 |
| TRAJLOY128F92E4EAF | 0.01 | 0.01 | 0.01 | 0.02 | 0.16 | 0.41 | 0.13 | 0.23 | 0.0 |
| TRADNOD128F421A2FE6 | 0.0 | 0.0 | 0.16 | 0.38 | 0.14 | 0.18 | 0.12 | 0.0 | 0.0 |

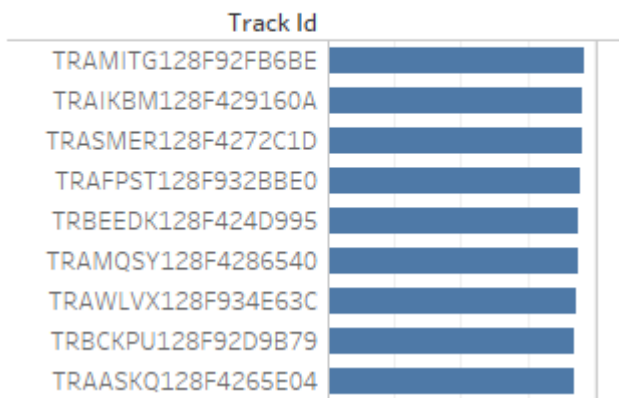
Aufgabe 4

Die einzelnen bins enthalten nun die relative Häufigkeit der einzelnen loudness Segmente. Zur Sichtung kann man diese als Spalte nehmen und über die `track_id` aufrufen. Wir haben eine Spalte `score` erstellt, die einen loudness score berechnet $\text{score} = \text{bin}_1 + 2 \cdot \text{bin}_2 + 3 \cdot \text{bin}_3 + \dots + 8 \cdot \text{bin}_8$, dieser gibt einen Wert wieder mit welchem sich der Anteil an lauten Segmenten vergleichen lässt.

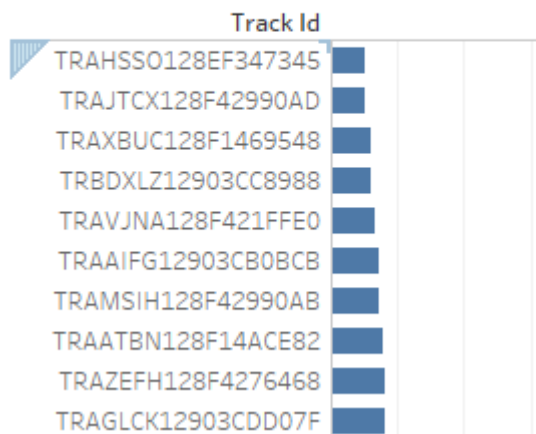


Wenn wir nun über den score absteigend sortieren finden wir laute songs, aufsteigend leise songs.

Profile-Vektoren



Profile-Vektoren



Da die Million Song Daten unter anderem von Spotify generiert sind, kann man nun die Titel und die Interpreten aus der Datenbank auslesen und die Songs mittels Spotify auf ihre Lautstärke akustisch überprüfen.

```
%pyspark

tid_loud = 'TRAMITG128F92FB6BE'
tid_soft = 'TRACFBD128F426CC34'

print('loud song')
df = spark.sql("select artist_name, title from msd10k_some_metadata where track_id = '{}'.format(tid_loud))
df.show()

print('soft song')
df = spark.sql("select artist_name, title from msd10k_some_metadata where track_id = '{}'.format(tid_soft))
df.show()

loud song
+-----+-----+
|artist_name| title|
+-----+-----+
|    Asure|Ugly MF|
+-----+-----+

soft song
+-----+-----+
| artist_name|          title|
+-----+-----+
|James Horner|The President's S...|
+-----+-----+
```

Loudness war sei dank war es schwieriger in Spotify eines der soften Songs zu finden. Verglichen haben wir dann akustisch:

Asure - 'Ugly MF' gegen James Horner - 'The President's Speech - Instrumental'

Das Ergebnis war deutlich. 'Ugly MF' hat so hohe Loudness, dass die Musik stark übersteuert klingt. Das andere Lied war sehr leise, ruhig und die loudness deutlich geringer, so dass es auch zu keiner digitalen Übersteuerung kam.

Aufgabe 5

Bins fester Breite lohnen sich, wenn man Informationen zur Verteilung gewinnen möchte. In diesem konkreten Fall für Aufgabe 4 war es durchaus sehr hilfreich, dass die bins in fester Breite erstellt wurden, um zwischen soften und lauten Songs zu unterscheiden. Bins fester Breite sind nicht unbedingt zur verteilten Berechnung gut geeignet. Zur verteilten Berechnung sind gleichstark gefüllte Bins zu bevorzugen, dies kann bei fester Breite bereits schon passieren, wäre aber nur Zufall.

Wenn wir uns die Verteilung der `timbre_0` anschauen, dann stellen wir fest, dass Bins fester Breite zur verteilten Berechnung nicht sinnvoll sind.

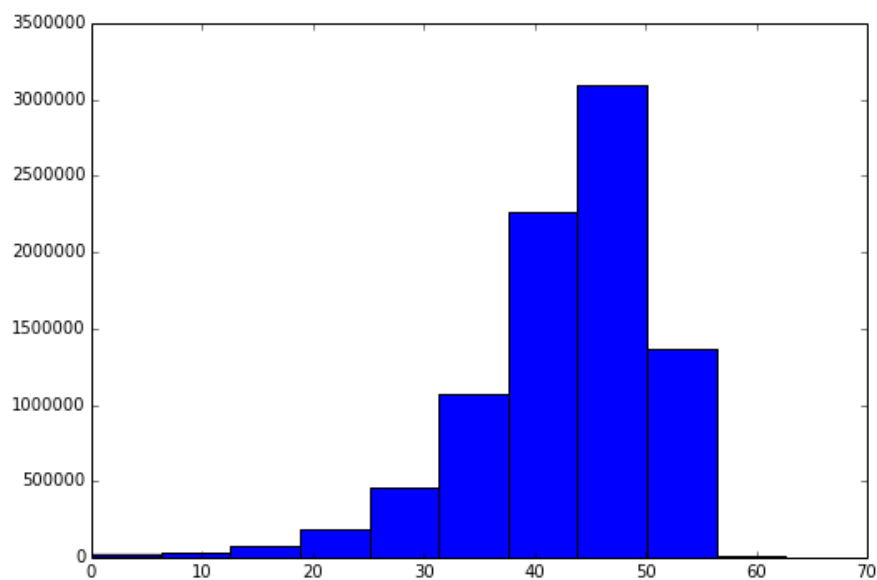

```
%pyspark
#bonus
from matplotlib import pyplot as plt

# get data frame
df = spark.sql("select timbre_0 from msd10k_timbre")

# create histogram
n_buckets = 10
bins, counts = df.select('timbre_0').rdd.flatMap(lambda x: x).histogram(n_buckets)

# plotten
plt.hist(bins[:-1], bins=bins, weights=counts)

(array([ 18174.,  30559.,  74463., 188216., 459180., 1069205.,
        2268378., 3097767., 1362795.,  8669.]), array([ 0.   ,  6.268, 12.536, 18.804, 25.072, 31.34 , 37.608, 43.876,
        50.144, 56.412, 62.68 ]), <a list of 10 Patch objects>)
```



Nutzen wir approximative Quantile, dann erhalten wir folgendes Binning (für 10% Quantile):

```
%pyspark
#bonus
from pyspark.ml.feature import Bucketizer as bt
from matplotlib import pyplot as plt

# help functions

def frange(start, stop, step):
    i = start
    out = []
    while i < stop:
        out.append(round(i,1))
        i += step
    return out

# get data frame
df = spark.sql("select track_id,timbre_0 from msd10k_timbre")

# calculate splits
splits = df.stat.approxQuantile("timbre_0",frange(0.1,1,0.1),0.05)
print('splits:')
print(splits[:-1])

# create histogram
bins, counts = df.select('timbre_0').rdd.flatMap(lambda x: x).histogram(splits)

# plotten
plt.clf()
plt.hist(bins[:-1], bins=bins, weights=counts)

# Binning über alle Timbres
btz = bt(splits=splits[:-1], inputCol="timbre_0", outputCol="buckets")

# Bucketizer-Objekt in Tabelle konvertieren
#df_b = btz.setHandleInvalid("keep").transform(df)
#df_b.show()
```

splits:

[36.092, 38.474, 40.91, 42.831, 44.547, 46.737, 48.01, 49.938, 52.05]

