

Big Data für Aktuare ein Praxisbeispiel

Dr. Axel Kaiser



BDO AG Wirtschaftsprüfungsgesellschaft

Hamburg 2017

BDO DEUTSCHLAND

Audit & Assurance | Tax & Legal | Advisory

DAV vor Ort

„Big Data für Aktuare -
ein Praxisbeispiel“

Dr. Axel Kaiser
2016 / 2017



BDO AUF EINEN BLICK

1.900
MITARBEITER



26
STANDORTE

€ 212 Mio.

UMSATZ IM GESCHÄFTSJAHR
2015



Seit 1920

GESELLSCHAFT MIT TRADITION



BDO NETZWERK



68.000

MITARBEITER



1.400+

MEHR ALS 1.400 BÜROS
WELTWEIT

158
IN 158 LÄNDERN
VERTRETEN



US\$ 7,6 Mrd.

UMSATZ IM GESCHÄFTSJAHR
2015/2016



BDO - BRANCHENCENTER VERSICHERUNGEN

Branchencenter Versicherungen

- Leitung: WP StB Thomas Volkmer
- Standorte: Köln, Hamburg, Frankfurt und München
- ca. 60 Mitarbeiter, darunter:
 - 8 Wirtschaftsprüfer (Versicherungsspezialisten)
 - 10 Steuerberater
 - Spezialisten (Aktuare, IT-Experten, Vertrieb)
- Betreuung von mehr als 100 Prüfungs- und Beratungsmandaten:
 - Versicherungen aller Sparten
 - Pensionskassen und Versorgungswerke
 - Versicherungsmakler, GKV
- Themenschwerpunkte
 - Jahresabschlussprüfungen
 - Revisionen
 - Projektunterstützung
 - (aktuarielle) Sonderthemen



Inhaltsverzeichnis

1. Einleitung	3
2. Ausgangspunkt sind die Daten der Krankenkasse.	4
3. Vorbereitung	5
4. Aufbereiten der Leistungsdaten	5
4.1. Verarbeiten der Rohdaten	6
4.2. Komprimierung und Umstrukturierung der Daten	7
5. Aufbereiten der Stammdaten	8
5.1. Einlesen und Aufbereiten der Stammdaten	8
5.2. Bereinigen der Stammdaten	11
5.3. Ermitteln der versicherten Jahresanteile	12
6. Aufbereiten der Wahltarifteilnehmer	14
6.1. Einlesen der Rohdaten	14
6.2. Aufbereiten der Daten	15
7. Kombination der Daten	17
7.1. Kombination der Stammdaten mit den Wahltarifteilnehmern	17
7.2. Kombination mit den Schadenaufwendungen	20
7.3. Durchschnittsbildung der normierten Leistungsdaten	21
7.4. Joinen von Stamm- und Leistungsdaten	22
7.5. Anreichern der Stammdaten	24
7.6. Verdichtung der Stammdaten	29
8. Durchführen der Logistischen Regression	31
8.1. Plausibilisierungen	32
8.2. Teilnahmewahrscheinlichkeiten	33
9. Durchführen des Matching	34
9.1 Zusammenfassung der Leistungsarten	34
9.2 Alters- und Kostenklassen	35
9.3 Gruppieren der Teilnehmer	35
10. Ergebnis	37
10.1 Kostenersparnis Wahltarif-freie Zeiten	38
10.2 Kostenersparnis PSM	38
10.3 Weitere Betrachtungen	38

[illegible]

1. Einleitung

Stattdessen möchte ich Ihnen heute ein Praxisbeispiel aus der aktuariellen Praxis zeigen, bei dem statistische Auswertungen auf (relativ) großen Datenmengen durchzuführen waren.

Methodisch kam hierfür das sog. "Propensity Score Matching" (PSM) zum Einsatz, technisch realisiert wurde die Aufgabe mit freier Software, nämlich der Programmiersprache Python mitsamt einiger Standardbibliotheken sowie der Browser-Oberfläche Jupyter, die sich zum gleichzeitigen Entwickeln und Dokumentieren eignet.

Einschub: PSM Die Teilnahme an dem betrachteten Wahltarif wird als abhängige Zufallsvariable betrachtet. Sie kann den Wert 0 oder 1 annehmen (keine Teilnahme oder Teilnahme), und hängt angenommenerweise von soziodemographischen, erklärenden Variablen X ab. Beispielsweise seien hier das Geschlecht, das Alter, Wohnort und Versicherungsleistungen genannt.

$$Y = \log(P/(1 - P))$$

3

$$Y \sim b * X$$

durchgeführt. Hierzu nutzen wir einerseits die Variablen von X aus der wahltariffreien Zeit und andererseits unser Wissen um die künftige Wahltarifteilnahme.

Auf der Basis der Regression werden dann für jeden (künftigen) Wahltarifteilnehmer Klassen von Mitgliedern gebildet, die eine nahezu gleiche Wahrscheinlichkeit zur Wahltarifteilnahme gemäß Modell haben. Abschließend vergleicht man die Schadenaufwendungen der Klassenmitglieder mit denen des Wahltarifteilnehmers und schließt hieraus auf die Einsparungen durch die Teilnahme.

Somit ist das Vorgehen dreigliedrig:

1. Aufbereiten der Daten, so dass die Variablen X und Y geeignet vorliegen (dies ist die Hauptarbeit und der Schwerpunkt des Vortrages)
2. Durchführen der Regression (i.W. lediglich ein Funktionsaufruf)
3. Bilden der Klassen und Auswerten der Kosteneffekte

Folgende Tools werden genutzt:

- Python: freie, objektorientierte Skriptsprache
- numpy: effiziente Implementierung mehrdimensionaler homogener Datenstrukturen (Arrays) für Python
- pandas: Implementierung von DataFrames (zweidimensionalen Tabellen analog SQL-Tabellen oder Excel oder R-Matrizen) auf der Basis von numpy
- matplotlib: zum Visualisieren von Sachverhalten
- sklearn: scipy-Bibliothek mit den benutzten statistischen Verfahren (hier: logistische Regression)
- Jupyter: Browser basierte Oberfläche mit sog. "Notebooks" zur Nutzung von Python und zur gleichzeitigen Dokumentation. Notebooks können auch als Präsentationen aufbereitet werden oder in PDF-Dateien umgewandelt werden (via $\LaTeX$)

Zum Ausprobieren bietet sich die [Anaconda-Distribution](#) an, die die benötigten Bibliotheken beinhaltet oder bereitstellt und auch unter Windows für einen lokalen Benutzer ohne Admin-Rechte installiert werden kann. Python selbst ist auf fast allen Systemen lauffähig, vom Handy bis zum Mainframe.

Alternativ wird für solche Zwecke auch gerne die Programmiersprache R genutzt, die als Programmiersprache nicht so universell ist wie Python, aber dafür im Bereich der statistischen Verfahren und Algorithmen "bleeding edge" ist.

2. Ausgangspunkt sind die Daten der Krankenkasse.

Geliefert wurden zum einen die jährlichen **Schadendaten** und zum anderen **Personen-Stammdaten**. Zusätzlich wurden noch Daten über die **Teilnahme** an den Wahltarifen geliefert.

- Die Schadendaten wurden für 6 Jahre als Textdatei mit 24 durch Semikolon getrennten Spalten geliefert. Pro Jahr waren etwa 16 Millionen Datensätze enthalten, d.h. das Gesamtvolumen betrug etwa 100 Millionen Datensätze. Für Versicherer geht das wahrscheinlich als groß durch. Die einzelnen Dateien waren unkomprimiert etwa 4 GB groß und konnten durch geeignete Komprimierung auf nahezu 10 % reduziert werden.
- Die Stammdaten enthielten etwa 2 Mio. Datensätze mit neun Spalten, die wieder als Textdateien vorlagen.

- Die Daten über die Teilnahme an den Wahlтарifen enthielten pro Jahr etwa 10.000 Datensätze mit 8 Spalten.

3. Vorbereitung

Zum Start sind einige Initialisierungen erforderlich. Einige spezifische Definitionen und Funktionen finden sich in einem separaten Modul namens `daten`.

```
In [1]: # startet einen Kommentar, der bis zum Zeilenende geht
# Definition einiger Module, Datenstrukturen und Funktionen
from daten import *

# die Grafiken sollen in das Notebook eingebettet werden
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
matplotlib.style.use('bmh')
#print(plt.style.available)

# wir nutzen das deutsche Zahlenformat (relevant für die Grafiken)
import locale
locale.setlocale(locale.LC_ALL, "")

# Formatieren von Fließkommazahlen in DataFrames
pandas.options.display.float_format = lambda x: locale.format("%1.2f", x, grouping=True)

# Funktion zum Formatieren der Achsen-Beschriftung
def formater(form="%d", faktor=1):
    return matplotlib.ticker.FuncFormatter(lambda i, pos: locale.format_string(form, i*faktor, grouping=True))

Leistungsjahre = range(2007, 2013) # betrachtete Leistungsjahre: 2007 - 2012
WT = "BR" # wir betrachten nur den Wahlтариф Beitragsrückgewähr
```

Die Rohdaten werden mit den in `daten` definierten Datenstrukturen eingelesen und für die weitere Verarbeitung als HDF-Dateien ("Hierarchical Data Format") gespeichert.

Zum Download und Abspeichern der Originaldaten bietet es sich an, diese zu komprimieren (z.B. mit dem Verfahren `xz`, welches 7-Zip beherrscht). Die Daten werden deutlich kleiner (bei `csv`-Dateien sind Kompressionsraten über 90 % realistisch) und das Einlesen kaum verlangsamt.

4. Aufbereiten der Leistungsdaten

Vor dem Einlesen großer Datenmengen lohnt eine Analyse der gelieferten Daten auf Inhalte und Struktur. Nicht benötigte Datenspalten brauchen nicht gelesen zu werden und manche Spalten müssen möglicherweise transformiert werden. Hierfür sind Leistungsdaten ein typisches Beispiel, bei dem man auch viel falsch machen kann.

So benötigt z.B. das Einlesen der Leistungsdaten mehrere Stunden, wenn man `pandas` die Datumsfelder automatisch analysieren und umwandeln lässt. Gibt man die Umwandlung explizit vor, dauert das Lesen nur noch wenige Minuten.

Darüber hinaus bietet es sich an, die Daten nach dem Einlesen und Transformieren für künftige Zugriffe abzuspeichern, z.B. als HDF-Dateien bzw. -Datenbanken.

Die Analyse der Leistungsdaten ergibt die folgende Datensatzstruktur:

```
leistungen = (("Leist_Kto_Auszug_KEY", False, None), # interner Schlüssel,
              ("Erheb_von_DTM", False, "Datum"), # stets der 01.01. des betrachteten Jahres,
              ("Erheb_bis_DTM", False, "Datum"), # stets der 31.12. des betrachteten Jahres,
              ("Erstell_DTM", False, "Datum"), # das Erstellungsdatum der Datei, für alle Sätze identisch,
              ("WT_CD", True, numpy.int_), # Wahlтариф-Teilnahme am Leistungstag? 1=ja, 0=nein)
```

```

("ID", True, numpy.int64),          # Versichertennummer
("IDMIT", True, numpy.int64),       # Versichertennummer des Mitversicherten
("GEBD", True, "Datum"),            # Geburtsdatum
("Vers_Typ_CD", False, None),       # Geschlecht "M" oder "F"
("Leist_Art_CD", False, numpy.str_), # Code für Leistungsart
("Leist_Art_Erw_CD", False, numpy.str_), # Code für erweiterte Leistungsart
("Leist_Von_DTM", True, "Datum"),    # Beginn Leistungserbringung
("Leist_Bis_DTM", True, "Datum"),     # Ende Leistungserbringung
("Vorsorge_CD", True, numpy.int_),   # Code Vorsorgeleistung (=1) oder nicht (=0)
("Geb_Pos_NR", False, None),         # EBM-Ziffer bei Arztabrechnungen bzw. PZN bei Arzneimitteln
("Geb_Pos_Erw_NR", False, None),     # ATC-Code bei Arzneimitteln
("Leist_Netto_BTR", True, numpy.float_), # Netto-Leistung
("Leist_Brutto_BTR", True, numpy.float_), # Brutto-Leistung
("Anteilig_CD", True, numpy.int_),   # Sind Ausgaben zeitanteilig (=1) oder nicht (=0) ermittelt worden
("LANR_NR", False, None),           # Lebenslange Arztnummer
("BSNR_NR", False, None),           # Betriebsstättennummer
("Leist_Ref_CD", False, None),       # Leistungs-Code
("Leist_Ref2_CD", False, None),      # Leistungs-Code 2
("Kontenart", True, numpy.str_),     # Kontenart der Leistung
)

```

wobei die erste Angabe den Spaltennamen angibt, die zweite, ob die betroffene Spalte gelesen werden soll, und die dritte den Datentyp.

4.1. Verarbeiten der Rohdaten

Einlesen der Leistungsdaten, verarbeiten und abspeichern in Tabellen LeistungenJJJJ in der Datei Leistungen.hdf.

```

In [2]: # Lesen der Leistungsdaten
#
# Leistungen.hdf ist die "Datenbank"
# und LeistungJJJJ hierin eine "Tabelle"
for jahr in Leistungsjahre:
    print("Lese Jahr %d ..." % jahr, end=" ")
    %time lies_leistungsdaten("Daten/Leistungsdaten%d.txt.xz" % jahr, "Leistungen.hdf", "Leistungen%d" % jahr)

pandas.HDFStore("Leistungen.hdf")

Lese Jahr 2007 ... Wall time: 6min
Lese Jahr 2008 ... Wall time: 4min 27s
Lese Jahr 2009 ... Wall time: 4min 26s
Lese Jahr 2010 ... Wall time: 4min 31s
Lese Jahr 2011 ... Wall time: 4min 41s
Lese Jahr 2012 ... Wall time: 4min 53s

Out[2]: <class 'pandas.io.pytables.HDFStore'>
File path: Leistungen.hdf
/Leistungen2007      frame      (shape->[20918412,10])
/Leistungen2008      frame      (shape->[15396012,10])
/Leistungen2009      frame      (shape->[15265502,10])
/Leistungen2010      frame      (shape->[15482841,10])
/Leistungen2011      frame      (shape->[15981457,10])
/Leistungen2012      frame      (shape->[16662650,10])

```

Die großen Datenmengen brauchen somit nur einmalig gelesen und interpretiert zu werden und sind in einem internen Format gespeichert deutlich schneller für die weitere Verwendung zugreifbar.

```

In [3]: %time lstg = pandas.HDFStore("Leistungen.hdf")["Leistungen2012"]
print("Anzahl Datensätze:", locale.format("%d", len(lstg), grouping=True))
lstg.head()

Wall time: 1.56 s
Anzahl Datensätze: 16.662.650

```

```

Out[3]:   WT_CD      ID      IDMIT      GEBD Leist_Von_DTM Leist_Bis_DTM \
0      0  1000000003  4621924273  2012-09-07  2012-10-01  2012-12-11
1      0  1000000003  4621924273  2012-09-07  2012-10-01  2012-10-01
2      0  1000000003  4621924273  2012-09-07  2012-10-15  2012-10-15
3      0  1000000003  4621924273  2012-09-07  2012-10-26  2012-10-26
4      0  1000000003  4621924273  2012-09-07  2012-10-26  2012-10-26

      Vorsorge_CD  Leist_Brutto_BTR  Anteilig_CD      Kostenart
0              1              56,00              0  Arztbehandlung
1              1              16,80              0  Arztbehandlung
2              1              30,45              0  Arztbehandlung
3              0               0,55              0  Arztbehandlung
4              0               6,42              0  Arztbehandlung

```

4.2. Komprimierung und Umstrukturierung der Daten

Wir verdichten die eben gewonnen Datensätze. Der Verdichtungsschlüssel ist durch die Felder ID, WT_CD und Kostenart gegeben. Das relevante Ergebnisfeld, das aufzusummieren ist, ist das Leistungsfeld Leist_Brutto_BTR.

Wir führen das hier exemplarisch für 2012 durch, umgesetzt ist es aber mit Hilfe einer Funktion aus dem Modul daten, wie wir später sehen werden.

```

In [4]: %time lstg = pandas.HDFStore("Leistungen.hdf")["Leistungen2012"]

Wall time: 1.59 s

```

Das Gruppieren geschieht quasi instantan, da lediglich Vorbereitungen getroffen aber keine weiteren Operationen durchgeführt werden. Das ergebnis der Operation ist ein DataFrameGroupBy-Objekt.

```

In [5]: %time lstg = lstg.groupby(["ID", "WT_CD", "Kostenart"])
        print(lstg)

Wall time: 0 ns
<pandas.core.groupby.DataFrameGroupBy object at 0x000000001C1C0A58>

```

Aber auch das Aufsummieren des verbliebenen Datenfeldes Leist_Brutto_BTR je Schlüsselprägung (immer noch knapp 1 Mio.) geht recht fix.

```

In [6]: %time lstg = lstg.Leist_Brutto_BTR.sum()
        print("Anzahl Datensätze:", locale.format("%d", len(lstg), grouping=True))
        lstg.head()

Wall time: 2.95 s
Anzahl Datensätze: 986.488

```

```

Out[6]: ID      WT_CD  Kostenart
6003    0      Arzneimittel    10,90
10105   0      Arzneimittel   131,14
        Arztbehandlung   244,22
92902   0      Arzneimittel    12,87
        Arztbehandlung    70,87
Name: Leist_Brutto_BTR, dtype: float64

```

Die Ausprägungen der Dimension Kostenart sollen als einzelne Spalten auftreten, wobei leere Felder mit Null zu belegen sind. Hierzu dient die Funktion unstack.

```

In [7]: %time lstg = lstg.unstack("Kostenart", fill_value=0)
        lstg.columns.name = ""
        print("Anzahl Datensätze:", locale.format("%d", len(lstg), grouping=True))
        lstg.head()

```

Wall time: 390 ms
Anzahl Datensätze: 377.285

```
Out[7]:
```

ID	WT_CD	Arzneimittel	Arztbehandlung	Heil_Hilfsmittel	Krankengeld	\
6003	0	10,90	0,00	0,00	0,00	
10105	0	131,14	244,22	0,00	0,00	
92902	0	12,87	70,87	0,00	0,00	
168602	0	0,00	11,61	0,00	0,00	
203012	0	15,59	186,89	0,00	1.945,57	

ID	WT_CD	Krankenhaus	Schwangerschaft	Sonstiges
6003	0	0,00	0,00	0,00
10105	0	0,00	0,00	0,00
92902	0	0,00	0,00	60,00
168602	0	0,00	0,00	30,00
203012	0	6.915,49	0,00	839,44

Vor dem Speichern sollen die beiden Schlüsselspalten ID und WT_CD wieder zu normalen Spalten "degradiert" werden.

```
In [8]: lstg = lstg.reset_index()
print("Anzahl Datensätze:", locale.format("%d", len(lstg), grouping=True))
lstg.head()
```

Anzahl Datensätze: 377.285

```
Out[8]:
```

	ID	WT_CD	Arzneimittel	Arztbehandlung	Heil_Hilfsmittel	Krankengeld	\
0	6003	0	10,90	0,00	0,00	0,00	
1	10105	0	131,14	244,22	0,00	0,00	
2	92902	0	12,87	70,87	0,00	0,00	
3	168602	0	0,00	11,61	0,00	0,00	
4	203012	0	15,59	186,89	0,00	1.945,57	

	Krankenhaus	Schwangerschaft	Sonstiges
0	0,00	0,00	0,00
1	0,00	0,00	0,00
2	0,00	0,00	60,00
3	0,00	0,00	30,00
4	6.915,49	0,00	839,44

Die oben dargestellten Schritte sind in der Funktion `verarbeite_leistungsdaten` definiert, die wir für alle Jahre aufrufen. Zusätzlich speichert die Funktion die Ergebnisse für die weitere Verwendung in einer HDF-Datei.

```
In [9]: %%time
for jahr in Leistungsjahre:
    print("Bearbeite Jahr %d ..." % jahr)
    verarbeite_leistungsdaten("Leistungen.hdf", "Leistungen_aggregiert.hdf", "Leistungen%d" % jahr)

Bearbeite Jahr 2007 ...
Bearbeite Jahr 2008 ...
Bearbeite Jahr 2009 ...
Bearbeite Jahr 2010 ...
Bearbeite Jahr 2011 ...
Bearbeite Jahr 2012 ...
Wall time: 30.2 s
```

5. Aufbereiten der Stammdaten

5.1. Einlesen und Aufbereiten der Stammdaten

Einlesen der Daten gemäß folgender Struktur:

```

stammdaten = (("ID", True, numpy.int64),      # Versichertennummer
              ("GES", True, numpy.str_),      # Geschlecht ("W" oder "M")
              ("GEBD", True, "Datum"),        # Geburtsdatum
              ("VERV", True, "Datum"),        # Beginn Versicherungszeit
              ("VERB", True, "Datum"),        # Ende Versicherungszeit
              ("VERST", True, None),          # Beitragsgruppe
              ("PLZ", True, numpy.str_),      # Postleitzahl
              ("EI", True, numpy.float_),     # Einkommensklasse
              ("IDMIT", True, numpy.int64)    # Versichertennummer des Mitversicherten
              )

```

Wir lesen die Stammdaten für die Jahre ab 2008 ein (für 2007 liegen keine Daten vor), für die sie vorliegen und bereinigen sie. Dazu gehört es insbesondere, Dupletten zu entfernen.

```

In [10]: %time stammsaetze = [lies_daten("Daten/Stammdaten%d.txt.bz2" % jahr, stammdaten) for jahr in Leistungsjahre[1:]]

```

Wall time: 29.5 s

```

In [11]: stamm = pandas.concat(stammsaetze, ignore_index=True)
print("Anzahl Datensätze:", locale.format("%d", len(stamm), grouping=True))
stamm.head()

```

Anzahl Datensätze: 2.358.657

```

Out[11]:
   ID  GES  GEBD  VERV  VERB  VERST  PLZ  EI  \
0  6060000021  W  1999-12-12  2005-10-01  2199-12-31    1111  89231  nan
1  7050010028  M  2003-08-15  2003-08-15  2199-12-31   700661  22549  nan
2  9000000005  W  2007-11-28  2007-11-28  2199-12-31   100000  88214  nan
3  9031000006  M  1988-04-23  2008-09-01  2199-12-31   986000  49080  nan
4  7092000005  W  1982-01-23  2008-07-01  2199-12-31    1111  16515  3,00

   IDMIT
0  9138910129
1  7746601005
2  1587348483
3           0
4           0

```

Zunächst entfernen wir doppelte Datensätze. Hierbei gehen die Felder VERST (Beitragsgruppe), PLZ und EI (Einkommensklasse) nicht in den Vergleich ein. Stattdessen nehmen wir hierfür die letzten, aktuellsten Informationen.

```

In [12]: stamm.drop_duplicates(subset=["ID", "GES", "GEBD", "VERV", "VERB", "IDMIT"], keep="last", inplace=True)
print("Anzahl Datensätze:", locale.format("%d", len(stamm), grouping=True))

```

Anzahl Datensätze: 678.086

Wir schauen uns den Fall mit der ID 9999215287 etwas genauer an, um die auftretenden Effekte zu demonstrieren. Der Beispielfall wurde in drei Stammdaten-Dateien geführt und wir müssen uns aus den Einzelinformationen eine sinnvolle Konstellation ableiten, die höchsten einen noch gültigen Datensatz enthalten. Zusätzlich sollten die verbleibenden Versicherungsintervalle eindeutig und überschneidungsfrei sein, was wir hier aber nicht überprüfen. Stattdessen wird in einem späteren Schritt der Schnitt der verbliebenen Versicherungsintervalle für die weitere Nutzung ermittelt.

```

In [13]: stamm.query("ID == 9999215287")

```

```

Out[13]:
   ID  GES  GEBD  VERV  VERB  VERST  PLZ  EI  \
1234262  9999215287  M  1963-11-20  2009-05-01  2199-12-31   500000  14542  2,00
1721313  9999215287  M  1963-11-20  2009-05-01  2012-07-22   410000  14542  0,00
2254663  9999215287  M  1963-11-20  2012-10-01  2199-12-31    1111  14542  3,00

   IDMIT
1234262    0
1721313    0
2254663    0

```

Sofern zu einer ID mehrere Datensätze mit VERB = 31.12.2199 (Vertrag bis) existieren, wird der Stand mit dem neuesten VERV genommen. Grundsätzlich müsste zu jedem der zu löschen den Sätze ein Satz mit gleichem VERV (Vertrag von) existieren, aber das prüfen wir nicht ab.

```
In [14]: # ursprüngliche Stammdaten
print("Anzahl ursprünglicher Stammdaten:", locale.format("%d", len(stamm), grouping=True))

# Alle Datensätze mit VERB=31.12.2199
inf = Stamm.query("VERB == '2199-12-31'")
print("Anzahl Datensätze mit VERB == '2199-12-31':", locale.format("%d", len(inf), grouping=True))

# Hiervon brauchen wir die Datensätze mit maximalem 'VERV' je ID
# dies liefert die Methode drop_duplicates mit dem Parameter keep='last'
infmax = inf.sort_values(by=["ID", "VERV"])
infmax = infmax.drop_duplicates(["ID"], keep="last")
print("Anzahl verbliebener Datensätze mit VERB == '2199-12-31':", locale.format("%d", len(infmax), grouping=True))

# bereinigte Stammdaten
stamm = Stamm.drop(set(inf.index) - set(infmax.index))
print("Anzahl Stammdaten nach Bereinigung:", locale.format("%d", len(stamm), grouping=True))
```

```
Anzahl ursprünglicher Stammdaten: 678.086
Anzahl Datensätze mit VERB == '2199-12-31': 531.759
Anzahl verbliebener Datensätze mit VERB == '2199-12-31': 472.584
Anzahl Stammdaten nach Bereinigung: 618.911
```

Wir konnten auf diese Weise knapp 60.000 Datensätze löschen. Unser Beispielfall sieht jetzt gut aus (aber das leider natürlich noch kein Beweis für irgendetwas).

```
In [15]: Stamm.query("ID == 9999215287")

Out[15]:
```

	ID	GES	GEBD	VERV	VERB	VERST	PLZ	EI	\
1721313	9999215287	M	1963-11-20	2009-05-01	2012-07-22	410000	14542	0,00	
2254663	9999215287	M	1963-11-20	2012-10-01	2199-12-31	1111	14542	3,00	


```

IDMIT
1721313    0
2254663    0
```

Im nächsten Schritt belegen wir bei Hauptversicherten das Feld IDMIT mit dem Wert von ID. Damit beinhaltet das Feld ID die Versicherungsnummer der versicherten Person und IDMIT stets die Versicherungsnummer des Mitglieds. Zusätzlich legen wir die Information über die Mitgliedschaft in der neuen Spalte Mitglied ab.

```
In [16]: HauptVers = Stamm.IDMIT == 0
HauptVers.head()

Out[16]:
```

	IDMIT
12	False
13	True
38	True
152	False
160	True

```
Name: IDMIT, dtype: bool
```



```
In [17]: # Hier erfolgt die Zuweisung der ID auf das Feld IDMIT sofern gleich 0 ist
Stamm.loc[Stamm.index[HauptVers], "IDMIT"] = Stamm[HauptVers].ID
Stamm["Mitglied"] = HauptVers
Stamm.head()

Out[17]:
```

	ID	GES	GEBD	VERV	VERB	VERST	PLZ	EI	\
12	9008000079	M	1999-10-11	2002-04-24	2199-12-31	700606	89522	nan	
13	2049000013	M	1973-05-29	2006-11-01	2199-12-31	420000	18069	nan	
38	1288000032	W	1981-06-11	2006-07-01	2199-12-31	986000	68307	nan	
152	2442020174	W	2001-11-15	2001-11-15	2199-12-31	1111	66119	nan	
160	7449010153	M	1989-01-31	2007-09-01	2199-12-31	1111	03048	1,00	


```

IDMIT Mitglied
12  9545902025  False
13  2049000013   True
38  1288000032   True
```

```
152 5297671118 False
160 7449010153  True
```

5.2. Bereinigen der Stammdaten

In diesem Schritt bereinigen wir die Daten. Dazu checken wir zunächst, welche Felder "ungültige" Werte enthalten. Hierzu dient die Funktion `isnull`.

```
In [18]: for i in stamm.columns:
         print("Spalte: ", i, "; Anzahl ungültiger Datensätze: ", locale.format("%d", stamm[i].isnull().sum(), grouping=

Spalte: ID; Anzahl ungültiger Datensätze: 0
Spalte: GES; Anzahl ungültiger Datensätze: 0
Spalte: GEBD; Anzahl ungültiger Datensätze: 0
Spalte: VERV; Anzahl ungültiger Datensätze: 0
Spalte: VERB; Anzahl ungültiger Datensätze: 0
Spalte: VERST; Anzahl ungültiger Datensätze: 0
Spalte: PLZ; Anzahl ungültiger Datensätze: 4
Spalte: EI; Anzahl ungültiger Datensätze: 260.013
Spalte: IDMIT; Anzahl ungültiger Datensätze: 0
Spalte: Mitglied; Anzahl ungültiger Datensätze: 0
```

Wir müssen also die Felder EI (Einkommensklasse) und PLZ (Postleitzahl) genauer ansehen. Zunächst betrachten wir die vorhandenen Ausprägungen der Einkommensklasse.

```
In [19]: print(sorted(stamm.EI.unique()))

[nan, 0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0, 13.0, 14.0]
```

Es bietet sich an, die ungültigen Werte (nan) mit -1 zu belegen.

```
In [20]: stamm.loc[:, "EI"] = stamm.EI.fillna(-1)
         print(sorted(stamm.EI.unique()))

[-1.0, 0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0, 13.0, 14.0]
```

Schwieriger wird es bei der Postleitzahl, da es hier deutlich mehr Ausprägungen gibt:

```
In [21]: print("Anzahl unterschiedlicher Postleitzahlen:", locale.format("%d", len(stamm.PLZ.unique()), grouping=True))

Anzahl unterschiedlicher Postleitzahlen: 9.947

In [22]: print(sorted(stamm.PLZ.str.len().unique()))

[1.0, 2.0, 3.0, 4.0, 5.0, nan]
```

Da es keine leeren Strings gibt, nehmen wir das für ungültige Werte. Vorher schauen wir uns noch die Fälle mit Null-Werten an.

```
In [23]: stamm[stamm.PLZ.isnull()]

Out[23]:
```

	ID	GES	GEBD	VERV	VERB	VERST	PLZ	EI	\
47518	3504570012	W	1943-05-05	1992-01-01	2008-05-31	1111	NaN	3,00	
505568	6347173006	W	1976-03-31	2004-03-01	2006-01-30	700606	NaN	-1,00	
1146722	2426502015	M	1962-04-25	2002-05-01	2199-12-31	700609	NaN	14,00	
2118674	9656172029	W	1969-06-21	1993-07-26	2007-12-31	1001	NaN	-1,00	

	IDMIT	Mitglied
47518	3504570012	True
505568	6347173006	True
1146722	2426502015	True
2118674	9656172029	True

```
In [24]: stamm.loc[:, "PLZ"] = stamm.PLZ.fillna("")
         print(sorted(stamm.PLZ.str.len().unique()))

[0, 1, 2, 3, 4, 5]
```

Den erreichten Stand sichern wir in einer HDF-Datei.

```
In [25]: stamm.to_hdf("Stammdaten.hdf", "Stammdaten")
```

5.3. Ermitteln der versicherten Jahresanteile

```
In [26]: %time stamm = pandas.HDFStore("Stammdaten.hdf")["Stammdaten"]
         stamm.head()
```

Wall time: 140 ms

```
Out[26]:
```

	ID	GES	GEBD	VERV	VERB	VERST	PLZ	EI	\
12	9008000079	M	1999-10-11	2002-04-24	2199-12-31	700606	89522	-1,00	
13	2049000013	M	1973-05-29	2006-11-01	2199-12-31	420000	18069	-1,00	
38	1288000032	W	1981-06-11	2006-07-01	2199-12-31	986000	68307	-1,00	
152	2442020174	W	2001-11-15	2001-11-15	2199-12-31	1111	66119	-1,00	
160	7449010153	M	1989-01-31	2007-09-01	2199-12-31	1111	03048	1,00	

	ID	IDMIT	Mitglied
12	9545902025	False	
13	2049000013	True	
38	1288000032	True	
152	5297671118	False	
160	7449010153	True	

Wir aggregieren die in den Jahren 2007 bis 2012 versicherten (anteiligen) Zeiträume über die Versicherten. Am einfachsten und am schnellsten wäre es, dies satzweise durchzuführen und die Anteile anschließend aufzusummieren. Leider gibt das die vorliegende Datenführung nicht her, da Überschneidungen in den einzelnen Sätzen vorhanden sind.

Aus diesem Grund erfolgt die Berechnung mittels der Funktion `apply` auf den gruppierten Daten. `apply` wird hierbei als Parameter eine Funktion übergeben, die als Parameter einen DataFrame bestehend aus den zum jeweiligen Schlüssel gehörenden Datensätzen übergeben bekommt. Im Beispiel ist das die Funktion `aggregiere_stamm` aus dem Modul `daten`. Dieses Verfahren ist äußerst langsam und grundsätzlich nicht zu empfehlen, aber manchmal halt unvermeidbar.

Anmerkung: Rechnen mit Datumswerten ist ziemlich kompliziert :-).

```
In [27]: %time stamm = stamm.groupby("ID", as_index=False).apply(aggregiere_stamm)
         stamm.reset_index(inplace=True, drop=True)
         stamm.head()
```

Wall time: 59min 46s

```
Out[27]:
```

	ID	IDMIT	GEBD	GES	Mitglied	EI	VERST	PLZ	\
0	6003	9742227275	2012-01-23	M	False	-1,00	1111	48346	
1	10105	8265802121	2007-08-15	M	False	-1,00	700606	80805	
2	92902	92902	1985-12-13	W	True	4,00	1111	35789	
3	117104	4054969608	1994-09-27	W	False	-1,00	700660	24589	
4	129213	129213	1977-07-19	M	True	-1,00	1111	12099	

	Vers_Anteil_2007	Vers_Anteil_2008	Vers_Anteil_2009	Vers_Anteil_2010	\
0	0,00	0,00	0,00	0,00	
1	0,38	1,00	1,00	1,00	
2	0,00	0,00	0,00	0,59	
3	0,34	0,58	0,00	0,00	

4	1,00	0,08	0,00	0,00
	Vers_Anteil_2011	Vers_Anteil_2012		
0	0,00	0,33		
1	1,00	1,00		
2	1,00	1,00		
3	0,00	0,00		
4	0,00	0,00		

Die so gewonnenen Daten können wir mit den jährlichen Schadenaufwendungen kombinieren, um die Schadenaufwände zu normieren. Daher abspeichern.

```
In [28]: stamm.to_hdf("Stammdaten.hdf", "Stammdaten_aggregiert")
```

Einschub: Map-Reduce Das behandelte Problem ist ein typisches "Map & Reduce"-Problem, wie es bei Big-Data häufig auftritt. Eine wichtige Eigenschaft solcher Probleme ist, dass sie typischerweise gut parallelisierbar sind, wie auch unser Problem:

- Wir haben Daten, deren Datensätzen zu einer speziellen Klasse gehören (hier: die eine gemeinsame ID haben).
- Auf diesen Klassen findet die gleiche Operation statt (hier: die Ermittlung der versicherten Zeiten).
- Abschließend sind die Ergebnisse zusammenzufassen.

Da eine Parallelisierung recht einfach umzusetzen ist, wollen wir im Folgenden den Effekt untersuchen. Es stellt sich heraus, dass auf dem genutzten System lediglich eine relativ geringe Einsparung von etwa 15 % erzielt werden kann, da der Rechner zwar mit vier Kernen ausgestattet ist, aber mit 16 Gigabyte einen relativ geringen Speicherausbau aufweist. Im Zuge der Berechnung wird ein virtueller Speicher von etwa 28 Gigabyte belegt, so dass hierdurch eine entsprechende Performanceeinbuße zu verzeichnen ist.

Daher vergleichen wir beide Verfahren auf einer geeigneten Teilmenge der Stammdaten (100.000 Datensätze), die kein Swappen erfordert. Es zeigt sich, dass immerhin eine Reduktion um 30 % erreicht werden kann.

Vorgehen zur Parallelisierung: Zusätzlich zum Python-Programm (bzw. zum Jupyter-Notebook) (und zur Installation von `ipyparallel`) ist eine Befehlskonsole zu öffnen und in das Verzeichnis mit den Programmen zu wechseln. Dort wird durch Eingabe von

```
ipcluster start -n 4
```

die benötigte Cluster-Infrastruktur gestartet. Der Parameter `-n` gibt hierbei die Anzahl der zu startenden Instanzen wieder und sollte der Anzahl der zu nutzenden Kerne entsprechen.

Zusätzlich sind lediglich die nachfolgenden Schritte für die Parallelisierung erforderlich.

```
In [29]: from ipyparallel import Client
         rc = Client()
         dview = rc[:]
         dview.block = True

         def aggregiere(df):
             import daten
             return daten.aggregiere_stamm(df)

In [30]: stamm = pandas.HDFStore("Stammdaten.hdf")["Stammdaten"]

In [31]: dflist = (group for name, group in stamm.groupby("ID", as_index=False))
         print("Aggregiere Daten ...", end=" ")
         %time res = dview.map(aggregiere, dflist)
         print("Ergebnis zusammenstellen ...", end=" ")
         %time res = pandas.concat(res, ignore_index=True)
```

```
Aggregiere Daten ... Wall time: 41min 32s
Ergebnis zusammenstellen ... Wall time: 8min 57s
```

Anmerkung Der Effekt durch die Parallelisierung ist geringer als erhofft. Das liegt zum Einen natürlich an dem Overhead für die Parallelisierung, im konkreten Fall aber auch daran, dass der Hauptspeicher zu klein ist. In dem Rechner befinden sich 16 Gigabyte, der tatsächlich genutzte (virtuelle) Speicher beträgt knapp 30 Gigabyte.

Wir exerzieren das Ganze daher auf einer Teilmenge der Daten durch, für die der physikalische Speicher ausreichend dimensioniert ist. In diesem Fall ergibt sich (immerhin / nur) eine Einsparung von bis zu 30 %. Die konkrete Einsparung ist abhängig von der sonstigen Belastung des Rechners..

```
In [32]: %%time
         dflist = (group for name, group in stamm.head(100000).groupby("ID", as_index=False))
         res = dview.map(aggregiere, dflist)
         res = pandas.concat(res, ignore_index=True)
```

```
Wall time: 8min 6s
```

```
In [33]: %%time
         stamm = stamm.head(100000).groupby("ID", as_index=False).apply(aggregiere_stamm)
         stamm.reset_index(inplace=True, drop=True)
```

```
Wall time: 9min 4s
```

```
In [34]: res.equals(stamm)
```

```
Out[34]: True
```

6. Aufbereiten der Wahltarifteilnehmer

6.1. Einlesen der Rohdaten

Einlesen der Daten gemäß der folgenden Struktur:

```
teilnahme = (("ID", True, numpy.int64),          # Versicherten-ID
              ("Art", True, numpy.str_),          # Tarifart (SB=Selbstbehalt, BR=Beitragsrückgewähr)
              ("Tarif", True, numpy.int_),        # Tarifstufe (numerisch)
              ("Tarifbeginn", True, "Datum"),     # Beginn Teilnahme am Wahltarif
              ("Tarifende", True, "Datum"),       # Ende Teilnahme am Wahltarif
              ("Storno", True, numpy.str_),       # nicht relevant ?!
              ("Abrechnungsjahr", True, numpy.float_), # Jahr für die Abrechnung
              ("Ergebnis", True, numpy.float),   # Ergebnis für Kunden im Abrechnungsjahr
              )
```

Die Daten enthalten Stornosätze, die zunächst gelöscht werden. Anschließend kann dann die gesamte Storno-Spalte gelöscht werden.

Als Ergebnis dieses Schrittes erhalten wir für jeden Wahltarifteilnehmer den Jahresanteil mit Wahltarifeinschluss für die betrachteten Jahre.

```
In [35]: %%time wt = lies_daten("Daten/Wahltarifteilnehmer.txt", teilnahme)
         wt.drop(wt.index[wt.Storno == "S"], inplace=True)
         del wt["Storno"]
         print("Anzahl Datensätze:", locale.format("%d", len(wt), grouping=True))
         wt.head()
```

```
Wall time: 125 ms
Anzahl Datensätze: 11.767
```

```
Out[35]:
```

	ID	Art	Tarif	Tarifbeginn	Tarifende	Abrechnungsjahr	Ergebnis
0	295000032	SB	120	2013-01-01	2015-12-31	nan	0,00
1	432000037	SB	500	2011-04-01	2199-12-31	nan	0,00
2	1372000036	BR	90	2009-04-01	2010-12-31	nan	0,00
4	5418000052	SB	120	2011-06-01	2199-12-31	nan	0,00
5	9010010104	BR	60	2009-04-01	2010-12-31	2.010,00	60,00

Wir betrachten hier lediglich den Wahltarif mit Beitragsrückgewähr (Art = "BR", definiert durch die Variable WT).

```
In [36]: wt = wt.query("Art == @WT")
print("Anzahl für Wahltarifart %s:" % WT, locale.format("%d", len(wt), grouping=True))
wt.head()
```

Anzahl für Wahltarifart BR: 9.882

```
Out[36]:
```

	ID	Art	Tarif	Tarifbeginn	Tarifende	Abrechnungsjahr	Ergebnis
2	1372000036	BR	90	2009-04-01	2010-12-31	nan	0,00
5	9010010104	BR	60	2009-04-01	2010-12-31	2.010,00	60,00
6	9010010104	BR	60	2009-04-01	2010-12-31	2.009,00	45,00
7	4048020241	BR	0	2007-07-01	2010-06-30	nan	0,00
8	2737020281	BR	90	2009-07-01	2010-12-31	2.009,00	45,00

6.2. Aufbereiten der Daten

Wir erzeugen für jedes Jahr eine Spalte mit dem Wahltarif-Anteil für dieses Jahr.

```
In [37]: %%time
for j in Leistungsjahre:
    start = pandas.tslib.Timestamp("01.01.%d" % j)
    ende = pandas.tslib.Timestamp("31.12.%d" % j)
    dauer = (ende - start).days + 1.0
    wt["Vers_Anteil_%d" % j] = wt.apply(lambda z, s=start, e=ende, d=dauer: max(0, min(1, ((min(z.Tarifende, e) - s) / d)))
```

Wall time: 10.8 s

```
In [38]: print("Anzahl Datensätze:", locale.format("%d", len(wt), grouping=True))
wt.head()
```

Anzahl Datensätze: 9.882

```
Out[38]:
```

	ID	Art	Tarif	Tarifbeginn	Tarifende	Abrechnungsjahr	Ergebnis	\
2	1372000036	BR	90	2009-04-01	2010-12-31	nan	0,00	
5	9010010104	BR	60	2009-04-01	2010-12-31	2.010,00	60,00	
6	9010010104	BR	60	2009-04-01	2010-12-31	2.009,00	45,00	
7	4048020241	BR	0	2007-07-01	2010-06-30	nan	0,00	
8	2737020281	BR	90	2009-07-01	2010-12-31	2.009,00	45,00	

	Vers_Anteil_2007	Vers_Anteil_2008	Vers_Anteil_2009	Vers_Anteil_2010	\
2	0,00	0,00	0,75	1,00	
5	0,00	0,00	0,75	1,00	
6	0,00	0,00	0,75	1,00	
7	0,50	1,00	1,00	0,50	
8	0,00	0,00	0,50	1,00	

	Vers_Anteil_2011	Vers_Anteil_2012
2	0,00	0,00
5	0,00	0,00
6	0,00	0,00
7	0,00	0,00
8	0,00	0,00

Die Spalten Abrechnungsjahr und Ergebnis interessieren uns für die Untersuchung nicht und werden gelöscht.

```
In [39]: del wt["Abrechnungsjahr"]
del wt["Ergebnis"]
```

Im nächsten Schritt fassen wir die doppelten Datensätze nach Verträgen (erkennbar durch Beginn- und Endetermin) zusammen. Als Werte verbleiben die eben erzeugten Anteil-Spalten.

Um sicherzustellen, dass es sich wirklich um Duplikate handelt, führen wir die Aggregation einmal mit einer Maximums- und einmal mit einer Minimumsbildung durch. Die Ergebnisse müssen übereinstimmen, sonst haben wir eine Inkonsistenz.

```
In [40]: schluessel = ["ID", "Art", "Tarif", "Tarifbeginn", "Tarifende"]
         minimum = wt.groupby(schluessel).min()
         maximum = wt.groupby(schluessel).max()
         assert minimum.equals(maximum)
```

Jetzt können wir die Gruppierung aufheben, Tarifart, Tarif, Tarifbeginn und Tarifende entfernen und nach der ID aggregieren. Wir enthalten dann je Mitglied einen Datensatz mit dem gewünschten Anteil der Wahltarifteilnahme pro Jahr.

```
In [41]: maximum.reset_index(inplace=True)
         del maximum["Art"]
         del maximum["Tarif"]
         del maximum["Tarifbeginn"]
         del maximum["Tarifende"]
         teilnehmer = maximum.groupby("ID").sum()
         teilnehmer.head()
```

```
Out[41]:
```

	Vers_Anteil_2007	Vers_Anteil_2008	Vers_Anteil_2009 \
ID			
320061	0,00	0,00	0,42
3827207	0,00	0,00	0,17
6500119	0,00	0,00	0,00
7526211	0,50	1,00	0,16
7772102	0,50	1,00	1,00

	Vers_Anteil_2010	Vers_Anteil_2011	Vers_Anteil_2012
ID			
320061	1,00	0,00	0,00
3827207	1,00	0,00	0,00
6500119	0,92	1,00	1,00
7526211	0,00	0,00	0,00
7772102	0,50	0,00	0,00

Interessanterweise gibt es für die Beitragsrückgewähr einen Fall, mit einem Anteil größer 1 in 2012, der daher rührt, dass ein Versicherter (mit der ID 4496772008) den Wahltarif gewechselt hat und zeitweise parallel in beiden Tarifen versichert war.

Dieser Fall wird manuell korrigiert.

```
In [42]: print(teilnehmer.max())
         print(teilnehmer.query("Vers_Anteil_2012 > 1"))
         teilnehmer.loc[teilnehmer.query("Vers_Anteil_2012 > 1").index, "Vers_Anteil_2012"] = 1
```

```
Vers_Anteil_2007    0,75
Vers_Anteil_2008    1,00
Vers_Anteil_2009    1,00
Vers_Anteil_2010    1,00
Vers_Anteil_2011    1,00
Vers_Anteil_2012    1,17
dtype: float64
```

	Vers_Anteil_2007	Vers_Anteil_2008	Vers_Anteil_2009 \
ID			
4496772008	0,00	0,00	0,00

	Vers_Anteil_2010	Vers_Anteil_2011	Vers_Anteil_2012
ID			
4496772008	0,92	1,00	1,17

```
In [43]: wt.query("ID == 4496772008")
```

```
Out[43]:
```

	ID	Art	Tarif	Tarifbeginn	Tarifende	Vers_Anteil_2007 \
7093	4496772008	BR	120	2010-02-01	2013-01-31	0,00

7094	4496772008	BR	150	2012-11-01	2199-12-31	0,00
------	------------	----	-----	------------	------------	------

	Vers_Anteil_2008	Vers_Anteil_2009	Vers_Anteil_2010	Vers_Anteil_2011	\
7093	0,00	0,00	0,92	1,00	
7094	0,00	0,00	0,00	0,00	

	Vers_Anteil_2012
7093	1,00
7094	0,17

Abspeichern der Teilnehmer an den Wahlтарифen.

```
In [44]: teilnehmer.to_hdf("Wahlтариф.hdf", "Wahlтариф")
```

7. Kombination der Daten

7.1. Kombination der Stammdaten mit den Wahlтарифteilnehmern

In diesem Schritt werden die Stammdaten um Informationen um die Wahlтарифteilnahme angereichert. Für jedes Mitglied erhalten wir einen Datensatz mit den versicherten Jahresanteilen, sofern keine Wahlтарифteilnahme vorliegt, bzw. zwei Datensätze mit den Jahresanteilen mit respektive ohne Wahlтарифteilnahme.

Einlesen von Stamm- und Wahlтарифdaten. Für die spätere Summation versehen wir hier die Wahlтариф-Anteile mit negativem Vorzeichen, weil wir für jeden Wahlтарифteilnehmer zwei Datensätze haben wollen: einen für die Zeit mit und einen für die Zeit ohne Wahlтарифteilnahme.

```
In [45]: %%time
stamm = pandas.HDFStore("Stammdaten.hdf")["Stammdaten_aggregiert"]
wt = - pandas.HDFStore("Wahlтариф.hdf")["Wahlтариф"]
```

Wall time: 359 ms

Wir vergleichen die ID, bzw. IDMIT zwischen den Dateien. Die unplausiblen Fälle sind für unsere Zwecke vernachlässigbar und werden ignoriert.

Anmerkung Es zeigt sich wieder einmal, dass vieles, was theoretisch denkbar ist, in der Praxis auch tatsächlich auftritt. Daher ist es eine sehr gute Idee, Daten stets auf merkwürdige Effekte zu untersuchen und nicht anzunehmen, dass sie so seien, wie man es vielleicht erwarten würde.

```
In [46]: # ID der Wahlтарифteilnehmer
i_wt = set(wt.index)
# ID der Mitglieder
i_mitg = set(stamm.IDMIT)
# ID der Versicherten
i_vers = set(stamm.ID)
# ID der Mitversicherten
i_mitv = i_vers - i_mitg

print("Anzahl der Mitglieder, die nicht versichert sind:", len(i_mitg - i_vers))
print("Anzahl der Wahlтарифteilnehmer, die keine Mitglieder sind:", len(i_wt - i_mitg))
print("Anzahl der Wahlтарифteilnehmer, die mitversichert sind:", len(i_wt & i_mitv))
print("Anzahl der Wahlтарифteilnehmer, die nicht versichert sind:", len(i_wt - i_vers))
```

```
Anzahl der Mitglieder, die nicht versichert sind: 0
Anzahl der Wahlтарифteilnehmer, die keine Mitglieder sind: 118
Anzahl der Wahlтарифteilnehmer, die mitversichert sind: 64
Anzahl der Wahlтарифteilnehmer, die nicht versichert sind: 54
```

Wir joinen die Stammdaten mit den Wahltarifteilnehmer und übernehmen deren Anteilwerte. Es wird ein "inner join" auf dem Feld IDMIT der Stammdaten (enthält die Mitgliedsnummer) und dem Feld ID der Wahltarife durchgeführt. Letzteres ist Schlüsselfeld der Tabelle und muss daher nicht explizit angegeben werden. Wir erhalten eine Tabelle der Stammdaten der Wahltarifteilnehmer.

```
In [47]: felder = ['ID', 'IDMIT', 'GEBD', 'GES', 'Mitglied', 'EI', 'VERST', 'PLZ']
wt_stamm = stamm[felder].join(wt, on="IDMIT", how="inner")
# wir indizieren neu; das ist aber eigentlich überflüssig, sieht aber "schöner" aus
wt_stamm.reset_index(drop=True, inplace=True)
wt_stamm.head()
```

```
Out[47]:
```

	ID	IDMIT	GEBD	GES	Mitglied	EI	VERST	PLZ	\
0	320061	320061	1967-02-01	W	True	4,00	1111	18182	
1	1052814	8476271013	2000-07-02	W	False	-1,00	1111	21395	
2	6149073116	8476271013	1996-11-30	W	False	-1,00	1111	21395	
3	8476271013	8476271013	1968-04-12	W	True	4,00	1111	21395	
4	3827207	3827207	1974-11-05	M	True	5,00	1111	38547	

	Vers_Anteil_2007	Vers_Anteil_2008	Vers_Anteil_2009	Vers_Anteil_2010	\
0	-0,00	-0,00	-0,42	-1,00	
1	-0,00	-0,00	-0,00	-0,75	
2	-0,00	-0,00	-0,00	-0,75	
3	-0,00	-0,00	-0,00	-0,75	
4	-0,00	-0,00	-0,17	-1,00	

	Vers_Anteil_2011	Vers_Anteil_2012
0	-0,00	-0,00
1	-1,00	-1,00
2	-1,00	-1,00
3	-1,00	-1,00
4	-0,00	-0,00

Die neue Tabelle wt_stamm enthält für jeden Versicherten, dessen zugehöriges Mitglied Wahltarifteilnehmer ist, neben den eigenen Stammdaten die Wahltarif-Versicherungsanteile des Mitglieds.

Anschließend fügen wir noch eine Wahltarifspalte ein, um die Datensätze später unterscheiden zu können.

```
In [48]: stamm["Wahltarif"] = False
wt_stamm["Wahltarif"] = True
```

Anhängen der gewonnen Daten an den ursprünglichen Stammdatensatz.

Das funktioniert, da beide Tabellen dieselbe Struktur haben.

```
In [49]: assert (stamm.columns == wt_stamm.columns).all()
stamm_mit_wt = stamm.append(wt_stamm, ignore_index=True)
```

Im nächsten Schritt müssen die versicherten Zeiten mit Wahltarif von denen ohne Berücksichtigung von Wahltarifzeiten abgezogen werden. Das Ergebnis wird in stamm_neu abgelegt. Die hier erzeugten Stammdaten enthalten also die Versicherungszeiten, zu denen keine Wahltarifteilnahme gegeben war.

Zunächst prüfen wir, dass jede ID höchstens zweimal auftritt (das erste Auftreten normal als Versicherter, das zweite durch die Wahltarifteilnahme); alles andere wäre ein Fehler.

```
In [50]: assert stamm_mit_wt.groupby(felder).ID.count().max() <= 2
%time stamm_neu = stamm_mit_wt.groupby(felder, as_index=False).sum()
stamm_neu["Wahltarif"] = False
stamm_neu.head()
```

Wall time: 952 ms

```
Out[50]:
```

	ID	IDMIT	GEBD	GES	Mitglied	EI	VERST	PLZ	\
0	6003	9742227275	2012-01-23	M	False	-1,00	1111	48346	

1	10105	8265802121	2007-08-15	M	False	-1,00	700606	80805
2	92902	92902	1985-12-13	W	True	4,00	1111	35789
3	117104	4054969608	1994-09-27	W	False	-1,00	700660	24589
4	129213	129213	1977-07-19	M	True	-1,00	1111	12099

	Vers_Anteil_2007	Vers_Anteil_2008	Vers_Anteil_2009	Vers_Anteil_2010	\
0	0,00	0,00	0,00	0,00	0,00
1	0,38	1,00	1,00	1,00	1,00
2	0,00	0,00	0,00	0,00	0,59
3	0,34	0,58	0,00	0,00	0,00
4	1,00	0,08	0,00	0,00	0,00

	Vers_Anteil_2011	Vers_Anteil_2012	Wahltarif
0	0,00	0,33	False
1	1,00	1,00	False
2	1,00	1,00	False
3	0,00	0,00	False
4	0,00	0,00	False

Bei den Mitversicherten kann es hierbei zu negativen Versicherungszeiten kommen (z.B. bei einem Eintritt in die Mitversicherung nach Einschluss eines Wahltarifes), bei den Mitgliedern sollte das eigentlich nicht passieren, aber wir sehen an dem Beispiel, dass auch diese Konstellation auftritt.

Das checken wir und setzen anschließend negative Zeiten für jedes Jahr auf 0.

Anschließend drehen wir in wt_stamm das Vorzeichen bei den Versicherungszeiten wieder um.

```
In [51]: stamm_neu.query("ID == 7636101004")
```

```
Out[51]:
```

	ID	IDMIT	GEBD	GES	Mitglied	EI	VERST	PLZ	\
415148	7636101004	7636101004	1967-06-11	M	True	5,00	410000	22607	

	Vers_Anteil_2007	Vers_Anteil_2008	Vers_Anteil_2009	\
415148	1,00	1,00	1,00	

	Vers_Anteil_2010	Vers_Anteil_2011	Vers_Anteil_2012	Wahltarif
415148	0,08	0,00	-0,42	False

```
In [52]: pandas.HDFStore("Stammdaten.hdf")["Stammdaten"].query("ID == 7636101004")
```

```
Out[52]:
```

	ID	GES	GEBD	VERV	VERB	VERST	PLZ	EI	\
469664	7636101004	M	1967-06-11	2006-02-21	2006-04-27	2111	22607	0,00	
1508291	7636101004	M	1967-06-11	2006-06-22	2012-01-11	410000	22607	0,00	
2027821	7636101004	M	1967-06-11	2012-06-14	2199-12-31	410000	22607	5,00	

	IDMIT	Mitglied
469664	7636101004	True
1508291	7636101004	True
2027821	7636101004	True

```
In [53]: lies_daten("Daten/Wahltarifteilnehmer.txt", teilnahme).query("ID == 7636101004")
```

```
Out[53]:
```

	ID	Art	Tarif	Tarifbeginn	Tarifende	Storno	Abrechnungsjahr	\
2769	7636101004	BR	150	2010-02-01	2013-01-31	NaN	nan	

	Ergebnis
2769	0,00

```
In [54]: for j in Leistungsjahre:
print("Mitglieder in %d mit negativem Versicherungsanteil: %d" % (j, len(stamm_neu.query("ID == IDMIT & Vers_Anteil_2007 < 0"))))
stamm_neu.loc[stamm_neu.index[stamm_neu["Vers_Anteil_%d" % j] < 0], "Vers_Anteil_%d" % j] = 0
wt_stamm["Vers_Anteil_%d" % j] = - wt_stamm["Vers_Anteil_%d" % j]
```

```
Mitglieder in 2007 mit negativem Versicherungsanteil: 1
Mitglieder in 2008 mit negativem Versicherungsanteil: 7
Mitglieder in 2009 mit negativem Versicherungsanteil: 144
Mitglieder in 2010 mit negativem Versicherungsanteil: 46
Mitglieder in 2011 mit negativem Versicherungsanteil: 13
Mitglieder in 2012 mit negativem Versicherungsanteil: 5
```

Alte und neue Stammdaten müssen grundsätzlich übereinstimmen, d.h. dieselbe Anzahl von Datensätzen und dieselben IDs enthalten.

```
In [55]: assert len(stamm) == len(stamm_neu)
         assert set(stamm.ID) == set(stamm_neu.ID)
```

Jetzt können wir an die neuen Stammdaten die Wahltarifstammdaten aus wt_stamm anhängen und nach ID, IDMIT und Wahltarif sortieren.

```
In [56]: stamm_mit_wt = stamm_neu.append(wt_stamm, ignore_index=True)
         stamm_mit_wt.sort_values(by=["ID", "IDMIT", "Wahltarif"], inplace=True)
```

Die Tabelle stamm_mit_wt enthält für alle Versicherten die versicherten Jahresanteile für die Jahre 2007 bis 2012.

Für Wahltarifteilnehmer enthält die Datei zwei Datensätze für die versicherten Jahresanteile bei Einschluss des Wahltarifes und für die Jahresanteile bei Ausschluss des Wahltarifes.

Wir sehen das an folgendem Ausschnitt der Daten.

```
In [57]: stamm_mit_wt.ix[12:14]
```

```
Out[57]:
```

	ID	IDMIT	GEBD	GES	Mitglied	EI	VERST	PLZ	\
12	292904	292904	1987-05-05	W	True	1,00	1111	01796	
13	320061	320061	1967-02-01	W	True	4,00	1111	18182	
544145	320061	320061	1967-02-01	W	True	4,00	1111	18182	
14	333303	2384055538	1967-04-11	W	False	-1,00	700606	40223	

	Vers_Anteil_2007	Vers_Anteil_2008	Vers_Anteil_2009	\
12	0,00	0,00	0,00	
13	1,00	1,00	0,58	
544145	0,00	0,00	0,42	
14	0,00	0,00	0,00	

	Vers_Anteil_2010	Vers_Anteil_2011	Vers_Anteil_2012	Wahltarif
12	0,00	0,17	1,00	False
13	0,00	1,00	1,00	False
544145	1,00	0,00	0,00	True
14	0,00	0,00	0,58	False

Zuletzt sichern wir das Ergebnis.

```
In [58]: stamm_mit_wt.to_hdf("Wahltarif_%s.hdf" % WT, "Stammdaten")
```

7.2. Kombination mit den Schadenaufwendungen

Für die Auswertung werden die Leistungen auf die versicherte Jahresdauer normiert. Somit ist eine Leistung von 1.000 € für ein halbes versichertes Jahr doppelt so "schlimm" wie 1.000 € Leistung für ein vollständig versichertes Jahr.

Das Ergebnis wird in der Datei Wahltarif_BR.hdf als Tabellen NormleistungenJJJJ gespeichert.

Folgendes passiert, obwohl es eigentlich nicht vorkommen sollte: * "verlorene" Leistungssätze: Leistungssätze ohne Stammsatz * Leistungssätze ohne Versicherung: es gibt zwar Stammdaten, aber diese weisen für das betroffene Jahr keinen Versicherungsschutz auf

Anmerkung: Die hier gezeigte Funktionalität wird erweitert durch die Funktion `normiere_leistung` im Modul `daten` implementiert.

```
In [59]: %%time
         felder = ['Arzneimittel', 'Arztbehandlung', 'Heil_Hilfsmittel', 'Krankengeld', 'Krankenhaus', 'Schwangerschaft', 'S
         stamm = pandas.HDFStore("Wahltarif_%s.hdf" % WT)["Stammdaten"]
         stamm["Wahltarif"] = 1 * stamm.Wahltarif # Die Spalte bekommt Typ Integer, damit es zu WT_CD aus den Leist
         stamm.set_index(["ID", "Wahltarif"], drop=True, inplace=True)
```

```

for jahr in Leistungsjahre:
    lstg = pandas.HDFStore("Leistungen_aggregiert.hdf")["Leistungen%d" % jahr]
    vers_ant = "Vers_Anteil_%d" % jahr
    lstg_join = lstg.join(stamm[vers_ant], on=["ID", "WT_CD"], how='inner')
    lstg_join.drop(lstg_join.index[lstg_join[vers_ant] <= 0], inplace=True)
    for feld in felder:
        lstg_join[feld] = lstg_join[feld] / lstg_join[vers_ant]
    del lstg_join[vers_ant]
    # Abspeichern des Ergebnisses
    lstg_join.to_hdf("Wahltarif_BR.hdf", "Normleistungen%d" % jahr)

Wall time: 3.42 s

```

Den Effekt der Normierung kann man im folgenden Beispiel im ersten Datensatz im Feld Arzneimittel sehen. Er resultiert daraus, dass der Versicherte in 2012 lediglich 4 Monate versichert war.

```

In [60]: lstg = pandas.HDFStore("Leistungen_aggregiert.hdf")["Leistungen2012"]
         lstg.head()

Out[60]:
```

	ID	WT_CD	Arzneimittel	Arztbehandlung	Heil_Hilfsmittel	Krankengeld	\
0	6003	0	10,90	0,00	0,00	0,00	
1	10105	0	131,14	244,22	0,00	0,00	
2	92902	0	12,87	70,87	0,00	0,00	
3	168602	0	0,00	11,61	0,00	0,00	
4	203012	0	15,59	186,89	0,00	1.945,57	

	Krankenhaus	Schwangerschaft	Sonstiges
0	0,00	0,00	0,00
1	0,00	0,00	0,00
2	0,00	0,00	60,00
3	0,00	0,00	30,00
4	6.915,49	0,00	839,44

```

In [61]: lstg_join = pandas.HDFStore("Wahltarif_%s.hdf" % WT) ["Normleistungen2012"]
         lstg_join.head()

Out[61]:
```

	ID	WT_CD	Arzneimittel	Arztbehandlung	Heil_Hilfsmittel	Krankengeld	\
0	6003	0	32,70	0,00	0,00	0,00	
1	10105	0	131,14	244,22	0,00	0,00	
2	92902	0	12,87	70,87	0,00	0,00	
3	168602	0	0,00	11,61	0,00	0,00	
4	203012	0	15,59	186,89	0,00	1.945,57	

	Krankenhaus	Schwangerschaft	Sonstiges
0	0,00	0,00	0,00
1	0,00	0,00	0,00
2	0,00	0,00	60,00
3	0,00	0,00	30,00
4	6.915,49	0,00	839,44

```

In [62]: pandas.HDFStore("Wahltarif_%s.hdf" % WT) ["Stammdaten"].query("ID == 6003")

Out[62]:
```

	ID	IDMIT	GEBO	GES	Mitglied	EI	VERST	PLZ	\
0	6003	9742227275	2012-01-23	M	False	-1,00	1111	48346	

	Vers_Anteil_2007	Vers_Anteil_2008	Vers_Anteil_2009	Vers_Anteil_2010	\
0	0,00	0,00	0,00	0,00	

	Vers_Anteil_2011	Vers_Anteil_2012	Wahltarif
0	0,00	0,33	False

7.3. Durchschnittsbildung der normierten Leistungsdaten

Wir haben die normierten Leistungen je Versichertem für die Jahre 2007 bis 2012. Wir erzeugen jetzt eine Tabelle mit den (normierten) Leistungsdaten aller Jahre, gruppieren sie nach ID und WT_CD und bilden den Mittelwert, d.h. wir berechneten die durchschnittlichen Leistungen pro Versicherungsjahr.

```
In [63]: lstg = pandas.concat((pandas.HDFStore("Wahltarif_%s.hdf" % WT)["Normleistungen%d" % j] for j in Leistungsjahre), ignore_index=True)

In [64]: print("Anzahl Datensätze:", locale.format("%d", len(lstg), grouping=True))
lstg.head()
```

Anzahl Datensätze: 2.135.361

```
Out[64]:
```

	ID	WT_CD	Arzneimittel	Arztbehandlung	Heil_Hilfsmittel	Krankengeld	\
0	10105	0	26,92	838,29	0,00	0,00	
1	129213	0	11,93	45,01	0,00	0,00	
2	216130	0	1.751,99	961,37	282,78	0,00	
3	287810	0	20,41	18,98	0,00	0,00	
4	320061	0	0,00	65,11	0,00	0,00	

	Krankenhaus	Schwangerschaft	Sonstiges
0	0,00	0,00	0,00
1	0,00	0,00	0,00
2	17.828,47	227,70	3.453,25
3	0,00	0,00	0,00
4	0,00	0,00	0,00

Wir können jetzt die Aggregation durchführen und anschließend das Ergebnis speichern.

```
In [65]: lstg = lstg.groupby(["ID", "WT_CD"]).mean()
lstg.to_hdf("Wahltarif_BR.hdf", "Durchschnittsleistung")
```

Da wir über 6 Leistungsjahre aggregieren, reduziert sich die Anzahl der Datensätze auf weniger als 25 %.

```
In [66]: print("Anzahl Datensätze:", locale.format("%d", len(lstg), grouping=True))
lstg.head()
```

Anzahl Datensätze: 488.584

```
Out[66]:
```

	ID	WT_CD	Arzneimittel	Arztbehandlung	Heil_Hilfsmittel	Krankengeld	\
6003	0		32,70	0,00	0,00	0,00	
10105	0		80,16	304,49	36,88	0,00	
92902	0		7,21	61,05	0,00	0,00	
129213	0		11,93	45,01	0,00	0,00	
168602	0		3,84	-1,17	0,00	0,00	

	ID	WT_CD	Krankenhaus	Schwangerschaft	Sonstiges
6003	0		0,00	0,00	0,00
10105	0		0,00	20,98	6,67
92902	0		0,00	0,00	44,70
129213	0		0,00	0,00	0,00
168602	0		0,00	0,00	10,00

7.4. Joinen von Stamm- und Leistungsdaten

In diesem Schritt erzeugen wir eine Tabelle `stamm_lstg_BR`, die zu den relevanten Stammdaten die jährlichen durchschnittlichen Leistungsausgaben enthält. Pro ID können maximal zwei Sätze existieren, nämlich einer mit und einer ohne Leistungen im Zeitpunkt eines Wahltarifeinschlusses des zugehörigen Mitglieds.

```
In [67]: lstg = pandas.HDFStore("Wahltarif_%s.hdf" % WT)["Durchschnittsleistung"]
stamm = pandas.HDFStore("Wahltarif_%s.hdf" % WT)["Stammdaten"]
```

Wir ermitteln noch den durchschnittlichen Versicherungsanteil in den betrachteten Jahren. Den benötigen wir, um einen Anteil der durchschnittlichen Mitversicherung zu ermitteln. Die jährlichen Versicherungsanteile benötigen wir nicht mehr.

```
In [68]: anteile = ["Vers_Anteil_%d" % j for j in Leistungsjahre]
stamm["Vers_Ant_Durchschnitt"] = stamm[anteile].sum(axis=1) / len(anteile)
```

Jetzt entfernen wir die Datensätze ohne Versicherungszeiten. Eine manuelle Überprüfung ergab viele Sätze mit Versicherungsbeginn in 2013, für das wir über keine Leistungsdaten verfügten.

```
In [69]: print("gelöschte Datensätze ohne Versicherungszeit:", locale.format("%d", len(stamm.query("Vers_Ant_Durchschnitt == 0")), inplace=True)
          stamm.drop(stamm.index[stamm.Vers_Ant_Durchschnitt == 0], inplace=True)
```

gelöschte Datensätze ohne Versicherungszeit: 42.925

Der Gruppierungsschlüssel setzt sich aus ID und WT_CD zusammen. Wir machen einen left-Join, damit keine Stammdatensätze verloren gehen. Für Fälle, zu denen keine Leistungsdaten vorliegen, werden die Felder ausgenullt. Das muss so sein, damit leistungsfreie Versicherte richtig behandelt werden.

Dies kann man schön an dem Stammsatz der ID 117104 sehen, für die es keine Leistungsdaten gibt.

```
In [70]: stamm.head()
```

```
Out[70]:
```

	ID	IDMIT	GEBD	GES	Mitglied	EI	VERST	PLZ	\
0	6003	9742227275	2012-01-23	M	False	-1,00	1111	48346	
1	10105	8265802121	2007-08-15	M	False	-1,00	700606	80805	
2	92902	92902	1985-12-13	W	True	4,00	1111	35789	
3	117104	4054969608	1994-09-27	W	False	-1,00	700660	24589	
4	129213	129213	1977-07-19	M	True	-1,00	1111	12099	

	Vers_Anteil_2007	Vers_Anteil_2008	Vers_Anteil_2009	Vers_Anteil_2010	\
0	0,00	0,00	0,00	0,00	
1	0,38	1,00	1,00	1,00	
2	0,00	0,00	0,00	0,59	
3	0,34	0,58	0,00	0,00	
4	1,00	0,08	0,00	0,00	

	Vers_Anteil_2011	Vers_Anteil_2012	Wahltarif	Vers_Ant_Durchschnitt
0	0,00	0,33	False	0,06
1	1,00	1,00	False	0,90
2	1,00	1,00	False	0,43
3	0,00	0,00	False	0,15
4	0,00	0,00	False	0,18

```
In [71]: lstg.head()
```

```
Out[71]:
```

	ID	WT_CD	Arzneimittel	Arztbehandlung	Heil_Hilfsmittel	Krankengeld	\
	6003	0	32,70	0,00	0,00	0,00	
	10105	0	80,16	304,49	36,88	0,00	
	92902	0	7,21	61,05	0,00	0,00	
	129213	0	11,93	45,01	0,00	0,00	
	168602	0	3,84	-1,17	0,00	0,00	

	ID	WT_CD	Krankenhaus	Schwangerschaft	Sonstiges
	6003	0	0,00	0,00	0,00
	10105	0	0,00	20,98	6,67
	92902	0	0,00	0,00	44,70
	129213	0	0,00	0,00	0,00
	168602	0	0,00	0,00	10,00

```
In [72]: schluessel = ["ID", "Wahltarif"]
felder = ['IDMIT', 'GEBD', 'GES', 'Mitglied', 'EI', 'VERST', 'PLZ', 'Vers_Ant_Durchschnitt']
stamm = stamm[schluessel+felder].join(lstg, on=schluessel, how="left")
stamm.fillna(0, inplace=True)
```

```
In [73]: stamm.head()
```

```
Out[73]:
```

	ID	Wahltarif	IDMIT	GEBD	GES	Mitglied	EI	VERST	PLZ	\
0	6003	False	9742227275	2012-01-23	M	False	-1,00	1111	48346	
1	10105	False	8265802121	2007-08-15	M	False	-1,00	700606	80805	
2	92902	False	92902	1985-12-13	W	True	4,00	1111	35789	
3	117104	False	4054969608	1994-09-27	W	False	-1,00	700660	24589	

4	129213	False	129213	1977-07-19	M	True	-1,00	1111	12099
---	--------	-------	--------	------------	---	------	-------	------	-------

	Vers_Ant_Durchschnitt	Arzneimittel	Arztbehandlung	Heil_Hilfsmittel	\
0	0,06	32,70	0,00		0,00
1	0,90	80,16	304,49		36,88
2	0,43	7,21	61,05		0,00
3	0,15	0,00	0,00		0,00
4	0,18	11,93	45,01		0,00

	Krankengeld	Krankenhaus	Schwangerschaft	Sonstiges
0	0,00	0,00	0,00	0,00
1	0,00	0,00	20,98	6,67
2	0,00	0,00	0,00	44,70
3	0,00	0,00	0,00	0,00
4	0,00	0,00	0,00	0,00

Zuletzt abspeichern.

```
In [74]: stamm.to_hdf("Wahltarif_%s.hdf" % WT, "Stamm_Leistung")
```

7.5. Anreichern der Stammdaten

Für unsere Auswertung müssen die Stammdaten noch angereichert bzw. bereinigt werden. Wir führen das Alter der versicherten Person ein und bereinigen die Informationen zu Postleitzahlen, Einkommens- und Beitragsklassen.

Für die nominalen Variablen führen wir sog. Dummy- oder Indikatorvariablen ein, die die Existenz einer Ausprägung anzeigen. Hierbei wird für jede mögliche Ausprägung der nominalen Variablen eine 0/1-Spalte angelegt (genauer gesagt für alle bis auf eine).

7.5.1 Einfügen einer Altersspalte

```
In [75]: stamm = pandas.HDFStore("Wahltarif_%s.hdf" % WT)["Stamm_Leistung"]
stamm.head()
```

```
Out[75]:
```

	ID	Wahltarif	IDMIT	GEBD	GES	Mitglied	EI	VERST	PLZ	\
0	6003	False	9742227275	2012-01-23	M	False	-1,00	1111	48346	
1	10105	False	8265802121	2007-08-15	M	False	-1,00	700606	80805	
2	92902	False	92902	1985-12-13	W	True	4,00	1111	35789	
3	117104	False	4054969608	1994-09-27	W	False	-1,00	700660	24589	
4	129213	False	129213	1977-07-19	M	True	-1,00	1111	12099	

	Vers_Ant_Durchschnitt	Arzneimittel	Arztbehandlung	Heil_Hilfsmittel	\
0	0,06	32,70	0,00		0,00
1	0,90	80,16	304,49		36,88
2	0,43	7,21	61,05		0,00
3	0,15	0,00	0,00		0,00
4	0,18	11,93	45,01		0,00

	Krankengeld	Krankenhaus	Schwangerschaft	Sonstiges
0	0,00	0,00	0,00	0,00
1	0,00	0,00	20,98	6,67
2	0,00	0,00	0,00	44,70
3	0,00	0,00	0,00	0,00
4	0,00	0,00	0,00	0,00

Einfügen einer Spalte mit dem Alter für Mitglieder. Da es nicht auf das konkrete Alter ankommt, nehmen wir als Bezugsjahr 2012 und die Geburtsjahresmethode.

```
In [76]: %time stamm["Alter"] = (2012 - stamm.GEBD.dt.year) * stamm.Mitglied
```

Wall time: 62.4 ms

7.5.2 Postleitzahlen Bei deutschen Postleitzahlen betrachten wir nur die erste Ziffer. Andere Postleitzahlen werden als "Sonstige" zusammengefasst. Hierbei interessieren uns nur die Daten der Mitglieder.

Zuerst checken wir die "ungültigen" (z.B. ausländischen) Sätze. 1. Fall: PLZ enthält nur Nullen
2. Fall: PLZ enthält Buchstaben 3. Fall: PLZ ist leer 4. Fall: PLZ enthält weniger als fünf Stellen

```
In [77]: null = stamm.PLZ.str.match("^0+$") & stamm["Mitglied"]
letter = stamm.PLZ.str.contains("[^0-9]") & stamm["Mitglied"]
leer = stamm.PLZ.str.len() == 0 & stamm["Mitglied"]
fünf = stamm.PLZ.str.len() < 5 & stamm["Mitglied"]
print("Anzahl PLZ nicht gültiger deutscher Postleitzahlen")
print(" - enthält nur Nullen: %d" % null.sum())
print(" - enthält Buchstaben: %d" % letter.sum())
print(" - ist leer: %d" % leer.sum())
print(" - enthält nicht fünf Ziffern: %d" % fünf.sum())
```

```
Anzahl PLZ nicht gültiger deutscher Postleitzahlen
- enthält nur Nullen: 45
- enthält Buchstaben: 48
- ist leer: 1
- enthält nicht fünf Ziffern: 1
```

Die Anzahl der "unsauberen" Fälle ist vernachlässigbar. Im nächsten Schritt erzeugen wir eine Hilfstabelle PLZ_dummies, die den Text "PLZ_" gefolgt von der ersten Ziffer der Postleitzahl bzw. bei unbekannten Postleitzahlen dem Wert "U" bei Mitgliedern und "M" bei Mitversicherten. Zusätzlich werden die ersten Ziffern zur Anonymisierung noch zufällig permutiert.

Im nächsten Schritt erzeugen wir hieraus sog. "Dummy"-Variablen, d.h. wir erzeugen zu jeder der Ausprägungen ("PLZ_0" - "PLZ_9", "PLZ_U" und "PLZ_M") eine Spalte mit Werten 0 oder 1. Die Spalte PLZ_M wird nicht benötigt und daher gelöscht.

```
In [78]: PLZ_dummies = stamm.PLZ.str.get(0)
PLZ_dummies[stamm.index[null]] = "U"
PLZ_dummies[stamm.index[letter]] = "U"
PLZ_dummies[stamm.index[leer]] = "U"
PLZ_dummies[stamm.index[fünf]] = "U"
PLZ_dummies[stamm.index[~ stamm.Mitglied]] = "M"

namen = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"]
namen_neu = numpy.random.permutation(namen)
namen_neu = ["PLZ_" + i for i in namen_neu]
plz_neu = dict(zip(namen, namen_neu))
plz_neu["U"] = "PLZ_U"
plz_neu["M"] = "PLZ_M"
PLZ_dummies.replace(plz_neu, inplace=True)

PLZ_dummies = pandas.get_dummies(PLZ_dummies)
del PLZ_dummies["PLZ_M"]
PLZ_dummies.head()
```

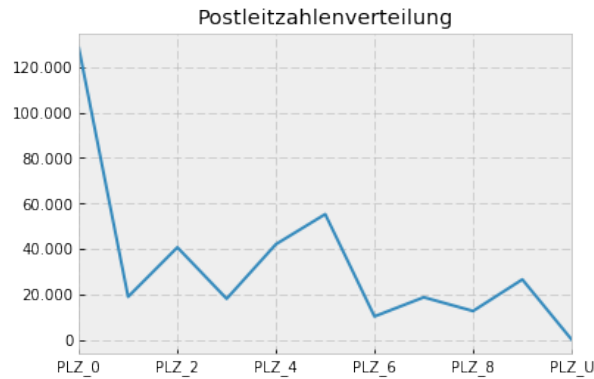
```
Out[78]:
```

	PLZ_0	PLZ_1	PLZ_2	PLZ_3	PLZ_4	PLZ_5	PLZ_6	PLZ_7	PLZ_8	PLZ_9	PLZ_U
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	1	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	1	0	0	0	0	0

Die neu erzeugten Variablen hängen wir den Stammdaten als neue Spalten an.

Vorher schauen wir uns aber noch die Postleitzahlen-Verteilung an und sehen eine starke örtliche Konzentration des Unternehmens.

```
In [79]: plot = PLZ_dummies.sum().plot()
_ = plot.yaxis.set_major_formatter(formatter())
_ = plot.set_title("Postleitzahlenverteilung")
```



```
In [80]: assert len(stamm) == len(PLZ_dummies)
stamm = pandas.concat([stamm, PLZ_dummies], axis=1)
stamm.head()
```

```
Out[80]:
```

	ID	Wahltarif	IDMIT	GEBD	GES	Mitglied	EI	VERST	PLZ	\
0	6003	False	9742227275	2012-01-23	M	False	-1,00	1111	48346	
1	10105	False	8265802121	2007-08-15	M	False	-1,00	700606	80805	
2	92902	False	92902	1985-12-13	W	True	4,00	1111	35789	
3	117104	False	4054969608	1994-09-27	W	False	-1,00	700660	24589	
4	129213	False	129213	1977-07-19	M	True	-1,00	1111	12099	

	Vers_Ant_Durchschnitt	...	PLZ_1	PLZ_2	PLZ_3	PLZ_4	PLZ_5	PLZ_6	\
0	0,06	...	0	0	0	0	0	0	
1	0,90	...	0	0	0	0	0	0	
2	0,43	...	0	0	1	0	0	0	
3	0,15	...	0	0	0	0	0	0	
4	0,18	...	0	0	0	0	1	0	

	PLZ_7	PLZ_8	PLZ_9	PLZ_U
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0

[5 rows x 29 columns]

7.5.3 Beitragsgruppe Für die Mitglieder setzen wir im nächsten Schritt den Code für die Beitragsgruppe. Für die Mitversicherten setzen wir den Wert "BG_M". Wir erzeugen erneut Indikatorvariablen und löschen die Spalte für die Mitversicherten.

Die Berechnung der Beitragsgruppen aus den gelieferten Daten erfolgt wieder mit Hilfe der Methode apply, die die übergebene Funktion für jede Zeile der Tabelle ausführt. Dies ist hier nötig, da das Feld VERST für unsere Zwecke unnötig viele Ausprägungen hat, die wir auf das Wesentliche reduzieren, nämlich 10 Beitragsklassen.

```
In [81]: %time BG_dummies = stamm.VERST.apply(beitragsgruppe)
BG_dummies[stamm.index[~ stamm.Mitglied]] = "BG_M"

BG_dummies = pandas.get_dummies(BG_dummies)
del BG_dummies["BG_M"]

print("Ausprägungen von VERST:")
print(sorted(stamm.VERST.unique()))
print("Anzahl der Ausprägungen von VERST:", locale.format("%d", len(stamm.VERST.unique()), grouping=True))
BG_dummies.head()
```

Wall time: 437 ms
Ausprägungen von VERST:

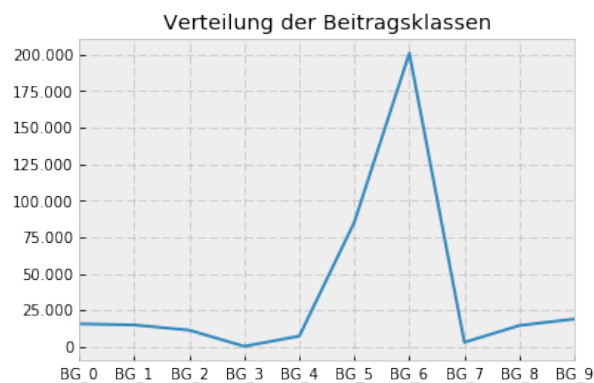
[0, 1, 100, 110, 1000, 1001, 1010, 1011, 1012, 1101, 1102, 1110, 1111, 1112, 1121, 1211, 1311, 1320, 1321, 2001, 2100, 2101, ...]
Anzahl der Ausprägungen von VERST: 107

```
Out[81]:
```

	BG_0	BG_1	BG_2	BG_3	BG_4	BG_5	BG_6	BG_7	BG_8	BG_9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	1	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	1	0	0	0

Auch hierzu sehen wir uns wieder die Verteilung an und hängen dann die Daten an die Stammdaten an.

```
In [82]: plot = BG_dummies.sum().plot()
_ = plot.yaxis.set_major_formatter(formatter())
_ = plot.set_title("Verteilung der Beitragsklassen")
```



Es handelt sich gemäß der Dokumentation der Beitragsklassen hauptsächlich um versicherungspflichtig Beschäftigte (BG_6) und Rentner (BG_5).

```
In [83]: stamm = pandas.concat([stamm, BG_dummies], axis=1)
stamm.head()
```

```
Out[83]:
```

	ID	Wahltarif	IDMIT	GEDB	GES	Mitglied	EI	VERST	PLZ	\
0	6003	False	9742227275	2012-01-23	M	False	-1,00	1111	48346	
1	10105	False	8265802121	2007-08-15	M	False	-1,00	700606	80805	
2	92902	False	92902	1985-12-13	W	True	4,00	1111	35789	
3	117104	False	4054969608	1994-09-27	W	False	-1,00	700660	24589	
4	129213	False	129213	1977-07-19	M	True	-1,00	1111	12099	

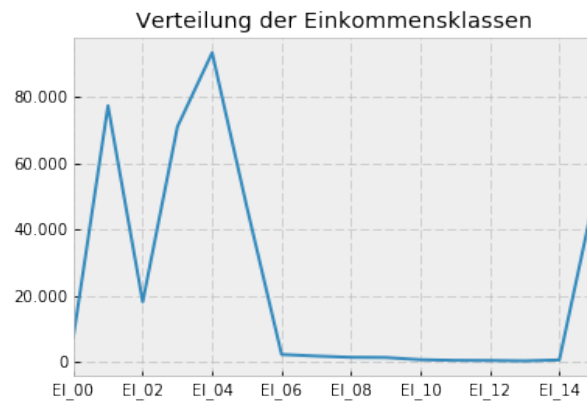
	Vers_Ant_Durchschnitt	...	BG_0	BG_1	BG_2	BG_3	BG_4	BG_5	BG_6	\
0	0,06	...	0	0	0	0	0	0	0	
1	0,90	...	0	0	0	0	0	0	0	
2	0,43	...	0	0	0	0	0	0	1	
3	0,15	...	0	0	0	0	0	0	0	
4	0,18	...	0	0	0	0	0	0	1	

	BG_7	BG_8	BG_9
0	0	0	0
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0

[5 rows x 39 columns]

7.5.4 Einkommensklasse Jetzt machen wir das Ganze noch für die Einkommensklasse.

```
In [84]: EI_dummies = "EI_" + stamm.EI.astype(int).astype(str).str.zfill(2)
EI_dummies[stamm.index[~ stamm.Mitglied]] = "EI_M"
EI_dummies[EI_dummies == "EI_-1"] = "EI_U"
EI_dummies = pandas.get_dummies(EI_dummies)
del EI_dummies["EI_M"]
plot = EI_dummies.sum().plot()
_ = plot.yaxis.set_major_formatter(formatter())
_ = plot.set_title("Verteilung der Einkommensklassen")
```



```
In [85]: stamm = pandas.concat([stamm, EI_dummies], axis=1)
stamm.head()
```

```
Out[85]:
```

	ID	Wahltarif	IDMIT	GEBD	GES	Mitglied	EI	VERST	PLZ	\
0	6003	False	9742227275	2012-01-23	M	False	-1,00	1111	48346	
1	10105	False	8265802121	2007-08-15	M	False	-1,00	700606	80805	
2	92902	False	92902	1985-12-13	W	True	4,00	1111	35789	
3	117104	False	4054969608	1994-09-27	W	False	-1,00	700660	24589	
4	129213	False	129213	1977-07-19	M	True	-1,00	1111	12099	

	Vers_Ant_Durchschnitt	...	EI_06	EI_07	EI_08	EI_09	EI_10	EI_11	\
0	0,06	...	0	0	0	0	0	0	
1	0,90	...	0	0	0	0	0	0	
2	0,43	...	0	0	0	0	0	0	
3	0,15	...	0	0	0	0	0	0	
4	0,18	...	0	0	0	0	0	0	

	EI_12	EI_13	EI_14	EI_U
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	1

[5 rows x 55 columns]

7.5.5 Geschlecht & Versichertenanteil Wir benötigen noch eine numerische Spalte für das Geschlecht des Mitgliedes und eine Spalte für die Anzahl der Mitversicherten von Mitgliedern (wobei jedes Mitglied bei sich selbst mitversichert ist, d.h. der Wert ist immer ≥ 1).

Erstere heißt Mann_M und ist genau dann 1, wenn der Versicherte ein männliches Mitglied ist, letztere Vers_Ant_Durchschnitt_M.

```
In [86]: stamm["Mann_M"] = (stamm.GES == "M") & stamm.Mitglied
stamm["Vers_Ant_Durchschnitt_M"] = stamm.Vers_Ant_Durchschnitt * stamm.Mitglied
```

7.6. Verdichtung der Stammdaten

Jetzt können wir die Sätze nach den Mitgliedern verdichten. Zuerst definieren wir die Felder, die wir behalten wollen, anschließend führen wir die Verdichtung durch. Aufgrund der Vorarbeiten sind alle beizubehaltenden Felder numerisch und der Wert nach Gruppierung kann durch Summation gebildet werden.

Wir bauen noch die Möglichkeit ein, die Leistungsfelder der Mitversicherten nicht zu berücksichtigen. Dies macht beispielsweise bei den Selbstbehaltstarifen Sinn.

```
In [87]: schluesssel = ["IDMIT", "Wahltarif"]
kosten = ['Arzneimittel', 'Arztbehandlung', 'Heil_Hilfsmittel',
          'Krankengeld', 'Krankenhaus', 'Schwangerschaft', 'Sonstiges']
felder = ['Alter', 'Mann_M', 'Vers_Ant_Durchschnitt', 'Vers_Ant_Durchschnitt_M']
felder += ["PLZ_%d" % i for i in range(0, 10)]
felder += ["PLZ_U"]
felder += ["BG_%d" % i for i in range(0, 10)]
felder += ["EI_%02d" % i for i in range(0, 15)]
felder += ["EI_U"]
felder += kosten

if WT == "SB" and False:
    # Für Selbstbehalttarife erfolgt keinerlei Berücksichtigung der Kosten von Mitversicherten
    # Bei Beitragsrückgewährtarifen könnte man beispielsweise lediglich die volljährigen Mitversicherten berücksichtigen
    for k in kosten:
        stamm[k] = stamm[k] * stamm.Mitglied
mitglieder = stamm[schluesssel + felder].groupby(schluesssel, as_index=False).sum()
```

Wir ermitteln nun den Mitversicherungsfaktor Mitversicherung je Mitglied. Dieser ist größer gleich 1, da das Mitglied mitgezählt wird. Es gibt einige Mitversicherte ohne Mitglied, die wir vorher entfernen.

```
In [88]: print("Mitversicherte ohne versichertes Mitglied:", locale.format("%d", numpy.sum(mitglieder.Vers_Ant_Durchschnitt_M == 0)))
print(u"Versicherungsleistungen für diese Fälle:")
print(mitglieder.query("Vers_Ant_Durchschnitt_M == 0")[kosten].sum())
mitglieder.drop(mitglieder.index[mitglieder.Vers_Ant_Durchschnitt_M == 0], inplace=True)
mitglieder["Mitversicherung"] = mitglieder.Vers_Ant_Durchschnitt / mitglieder.Vers_Ant_Durchschnitt_M
del mitglieder["Vers_Ant_Durchschnitt_M"]
del mitglieder["Vers_Ant_Durchschnitt"]
```

```
Mitversicherte ohne versichertes Mitglied: 166
Versicherungsleistungen für diese Fälle:
Arzneimittel      95.093,15
Arztbehandlung    55.841,15
Heil_Hilfsmittel  12.700,37
Krankengeld       31,96
Krankenhaus      1.410.433,89
Schwangerschaft   11.488,24
Sonstiges         53.082,85
dtype: float64
```

Spaßeshalber prüfen wir für die Wahltarife mit Beitragsrückgewähr noch den schlimmsten Ausreisser mit einem MV-Faktor über 100. Wir sehen, dass die Stammdaten hier nicht konsistent sind, da es einen Zeitraum der Mitversicherung gibt, in dem keine Versicherung vorliegt. Wir betrachten das Mitversicherungsverhältnis weiter unten noch einmal und bereinigen dort die offensichtlich unsinnigen Werte.

```
In [89]: if WT == "BR":
    %%time
    print(mitglieder.Mitversicherung.sort_values().tail())
    idmit = mitglieder.ix[12588].IDMIT
    print(idmit)
    print(pandas.HDFStore("Stammdaten.hdf").query("IDMIT == @idmit"))
```

```
Wall time: 0 ns
230711    73,00
133351    109,00
```

```

296874    126,23
107809    140,87
12588     335,92
Name: Mitversicherung, dtype: float64
277972130
      ID GES      GEBD      VERV      VERB  VERST  PLZ  EI  \
1858176  277972130  W  1970-12-29  2008-02-01  2008-02-01   1111  57350 -1,00
1868137  1437574751  M  2000-08-07  2006-07-19  2007-11-30   1111  57350 -1,00

      IDMIT Mitglied
1858176  277972130    True
1868137  277972130    False

```

Wir benennen noch die beiden Spalten IDMIT nach ID und Man_M nach Mann um und bringen anschließend noch die Spalte Mitversicherung nach vorne, so dass die Leistungsfelder hinten stehen.

```

In [90]: spalten = list(mitglieder.columns)
          spalten[0] = "ID"
          spalten[3] = "Mann"
          mitglieder.columns = spalten
          mitglieder.head()
          spalten[4:4] = ["Mitversicherung"]
          del spalten[-1]
          mitglieder = mitglieder.reindex(columns=spalten)
          mitglieder.head()

Out[90]:
      ID Wahltarif  Alter  Mann  Mitversicherung  PLZ_0  PLZ_1  PLZ_2  \
0   92902      False   27  False             1,00     0     0     0
1  129213      False   35   True             1,00     0     0     0
2  168602      False   22   True             1,00     0     0     0
3  203012      False   48   True             1,00     1     0     0
4  216130      False   27   True             1,00     0     0     0

      PLZ_3  PLZ_4  ...  EI_13  EI_14  EI_U  Arzneimittel  Arztbehandlung  \
0         1     0  ...      0      0     0           7,21           61,05
1         0     0  ...      0      0     1          11,93           45,01
2         0     0  ...      0      0     0           3,84           -1,17
3         0     0  ...      0      0     0          22,76          169,88
4         0     0  ...      0      0     0        1.323,71          794,67

      Heil_Hilfsmittel  Krankengeld  Krankenhaus  Schwangerschaft  Sonstiges
0                0,00         0,00         0,00             0,00         44,70
1                0,00         0,00         0,00             0,00         0,00
2                0,00         0,00         0,00             0,00        10,00
3                0,00        1.236,10        2.778,12             0,00        209,86
4               226,16         0,00       14.484,11             67,80       3.281,97

[5 rows x 49 columns]

```

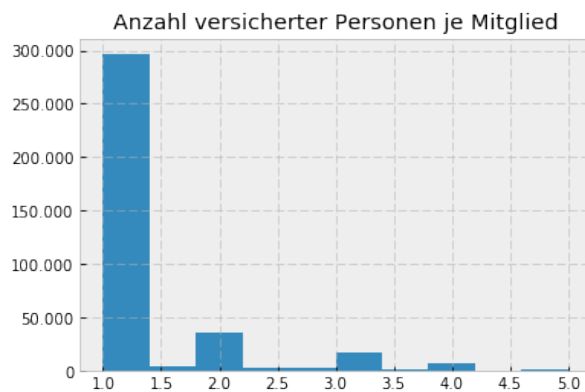
Leider gibt es immer noch einige Elemente mit unrealistisch hohen Faktoren, die zumindest teilweise aus der Datenführung resultieren. Für die Zwecke des PSM wird der Mitversicherungsfaktor auf den Wert 5 beschränkt. Werte größer als 5 werden als potenzielle Fehler dem Standardtopf 1 zugewiesen.

```

In [91]: print("Anzahl Sätze mit Mitversicherung > 5:", locale.format("%d", mitglieder.query("Mitversicherung > 5").Mitversicherung.
          mitglieder.loc[mitglieder.index[mitglieder.Mitversicherung > 5], "Mitversicherung"] = 1
          plot = mitglieder.Mitversicherung.hist()
          _ = plot.yaxis.set_major_formatter(formatter())
          _ = plot.set_title("Anzahl versicherter Personen je Mitglied")

```

```
Anzahl Sätze mit Mitversicherung > 5: 1.173
```



Und speichern des Ergebnisses.

```
In [92]: mitglieder.to_hdf("Wahltarif_%s.hdf" % WT, "Mitglieder")
```

8. Durchführen der Logistischen Regression

Die Stammdaten müssen zuerst aufgeteilt werden in die Mitglieder mit und die ohne Wahltarif. Hierbei ist zu beachten, dass bei den Mitgliedern mit Wahltarifen nicht unbedingt Versicherungszeiten ohne Wahltarifteilnahme vorliegen müssen. Für das PSM werden lediglich die Sätze ohne Wahltarif berücksichtigt, allerdings sind für die Mitglieder, für die ein Satz mit Wahltarif existiert, die entsprechenden Sätze für das PSM zu kennzeichnen.

```
In [93]: mitglieder = pandas.HDFStore("Wahltarif_%s.hdf" % WT) ["Mitglieder"]
mitglieder["Teilnahme"] = 0 # Teilnahme-Kennzeichen für PSM
wt_mit = mitglieder[mitglieder.Wahltarif]
wt_ohne = mitglieder[~ mitglieder.Wahltarif].copy()
wt_mit.set_index("ID", inplace=True)
wt_ohne.set_index("ID", inplace=True)
wt_mit_ohne = wt_ohne.index.join(wt_mit.index, how="inner") # diese ID gibt es sowohl mit als auch ohne Wahltarif
wt_ohne.loc[wt_mit_ohne, "Teilnahme"] = 1

print("Fälle ohne Wahltarif-freie Versicherungszeit:", len(wt_mit) - len(wt_mit_ohne))
```

Fälle ohne Wahltarif-freie Versicherungszeit: 148

Die logistische Regression wird auf den Datensätzen durchgeführt, die wir für die Mitglieder in den Wahltarif-freien Zeiten konstruiert haben. Die "erklärenden" Variablen werden in dem Array X gespeichert, die "abhängige" Variable, die angibt, ob das Mitglied grundsätzlich an dem Wahltarif teilnimmt, in Y.

Hierfür müssen wir das pandas-Umfeld verlassen und auf die zugrundeliegenden numpy-Arrays zurückgreifen, was über das Attribut values erfolgt.

```
In [94]: wt_ohne.reset_index(inplace=True)
wt_ohne["Mann"] = 1. * wt_ohne["Mann"] # Boolean Feld in numerisches Feld umwandeln
felder = wt_ohne.columns[2:-1] # für das PSM relevante Felder
X = wt_ohne[felder].values
Y = wt_ohne["Teilnahme"].values
```

Wir können jetzt die Regression durchführen. Zusätzlich fügen wir ein Feld mit der Wahrscheinlichkeit zur Wahltrifteilnahme auf Grund der ermittelten Regression ein.

```
In [95]: %%time
from sklearn import linear_model
```

```

logreg = linear_model.LogisticRegression(C=1e5)
logreg.fit(X, Y)
wt_ohne["WT_Propability"] = logreg.predict_proba(X)[: ,1]

Wall time: 12.1 s

```

Abspeichern des Ergebnisses.

```
In [96]: wt_ohne.to_hdf("Wahltarif_%s.hdf" % WT, "PSM")
```

8.1. Plausibilisierungen

Wir führen noch ein paar kleinere Plausibilisierungen zwecks Qualitätssicherung durch.

Zum einen betrachten wir die Regressionskoeffizienten auch um die relevanten Faktoren zu identifizieren, zum anderen ermitteln wir die Anpassungsgüte mittels der `score`-Funktion.

```
In [97]: for i, j in sorted(zip(wt_ohne.columns[2:-1], logreg.coef_[0])):
        print(i, ": ", locale.format("%.6f", j), sep="")
```

```

Alter: 0,016647
Arzneimittel: -0,000282
Arztbehandlung: -0,000505
BG_0: -0,564662
BG_1: -0,951788
BG_2: -0,235448
BG_3: 0,010728
BG_4: -0,712268
BG_5: -1,423062
BG_6: -0,206175
BG_7: 0,037697
BG_8: 0,274752
BG_9: 0,351331
EI_00: -0,590226
EI_01: -0,619400
EI_02: -0,539018
EI_03: -0,245211
EI_04: -0,021612
EI_05: -0,090662
EI_06: -0,100978
EI_07: 0,019379
EI_08: 0,039258
EI_09: 0,050829
EI_10: 0,097905
EI_11: 0,065104
EI_12: 0,007237
EI_13: 0,000447
EI_14: 0,012846
EI_U: -1,504793
Heil_Hilfsmittel: -0,000301
Krankengeld: -0,000005
Krankenhaus: -0,000003
Mann: -0,031654
Mitversicherung: 0,183663
PLZ_0: -0,499068
PLZ_1: -0,381068
PLZ_2: -0,575001
PLZ_3: -0,232964
PLZ_4: -0,673504
PLZ_5: -0,225707
PLZ_6: 0,193802
PLZ_7: -0,241088
PLZ_8: -0,208902
PLZ_9: -0,551490
PLZ_U: -0,023905
Schwangerschaft: 0,000038
Sonstiges: -0,000127

```

Das Ergebnis zeigt, dass einige Parameter wesentlich sind: Mitversicherung, Wohnsitz, gut verdienende Arbeitnehmer oder Selbständige, höheres Einkommen. Die angefallenen Gesundheitskosten üben quasi keinen Einfluss aus.

```
In [98]: print("Score: {:.2f} %".format(100 * logreg.score(X, Y)))
```

Score: 97.67 %

8.2. Teilnahmewahrscheinlichkeiten

Außerdem sehen wir uns die Wahrscheinlichkeitsverteilung für die Wahltarifteilnahme für Teilnehmer und Nicht-Teilnehmer an.

```
In [99]: wt_ohne.head()
```

```
Out[99]:
```

	ID	Wahltarif	Alter	Mann	Mitversicherung	PLZ_0	PLZ_1	PLZ_2	PLZ_3	\
0	92902	False	27	0,00	1,00	0	0	0	1	
1	129213	False	35	1,00	1,00	0	0	0	0	
2	168602	False	22	1,00	1,00	0	0	0	0	
3	203012	False	48	1,00	1,00	1	0	0	0	
4	216130	False	27	1,00	1,00	0	0	0	0	

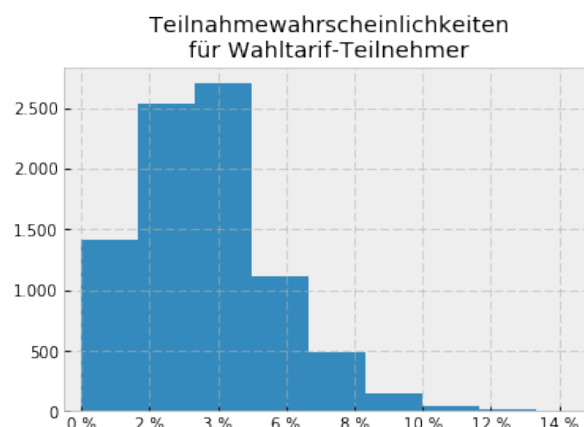
	PLZ_4	...	EI_U	Arzneimittel	Arztbehandlung	\
0	0	...	0	7,21	61,05	
1	0	...	1	11,93	45,01	
2	0	...	0	3,84	-1,17	
3	0	...	0	22,76	169,88	
4	0	...	0	1.323,71	794,67	

	Heil_Hilfsmittel	Krankengeld	Krankenhaus	Schwangerschaft	Sonstiges	\
0	0,00	0,00	0,00	0,00	44,70	
1	0,00	0,00	0,00	0,00	0,00	
2	0,00	0,00	0,00	0,00	10,00	
3	0,00	1.236,10	2.778,12	0,00	209,86	
4	226,16	0,00	14.484,11	67,80	3.281,97	

	Teilnahme	WT_Propability
0	0	0,04
1	0	0,01
2	0	0,01
3	0	0,02
4	0	0,01

[5 rows x 51 columns]

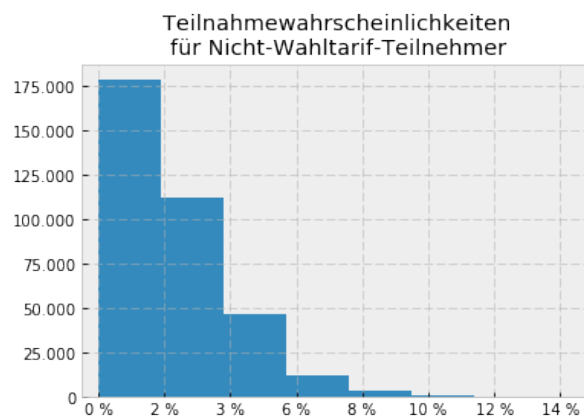
```
In [100]: plot = wt_ohne[wt_ohne.Teilnahme == 1].WT_Propability.hist()
_ = plot.set_xlim(-0.005, 0.15)
_ = plot.xaxis.set_major_formatter(formatter("%d %%", 100))
_ = plot.yaxis.set_major_formatter(formatter())
_ = plot.set_title("Teilnahmewahrscheinlichkeiten\nfür Wahltarif-Teilnehmer")
```



```
In [101]: wt_ohne[wt_ohne.Teilnahme == 1].WT_Propability.sort_values().tail()
```

```
Out[101]: 69044    0,14
          273741   0,15
          307306   0,16
          334072   0,16
          11631    0,17
          Name: WT_Propability, dtype: float64
```

```
In [102]: plot = wt_ohne[wt_ohne.Teilnahme == 0].WT_Propability.hist()
_ = plot.set_xlim(-0.005, 0.15)
_ = plot.xaxis.set_major_formatter(formatter("%d %%", 100))
_ = plot.yaxis.set_major_formatter(formatter())
_ = plot.set_title("Teilnahmewahrscheinlichkeiten\nfür Nicht-Wahltarif-Teilnehmer")
```



```
In [103]: wt_ohne[wt_ohne.Teilnahme == 0].WT_Propability.sort_values().tail()
```

```
Out[103]: 358699   0,16
          143860   0,17
          98010    0,18
          26898    0,19
          287333   0,19
          Name: WT_Propability, dtype: float64
```

Da insgesamt die Wahl eines Wahltarifes ein "seltenes" Ereignis ist, sind entsprechend auch die ermittelten Wahrscheinlichkeiten der Wahltarifteilnahme niedrig. Nichts desto trotz zeigt sich aber schon eine deutlich andere Wahrscheinlichkeitsstruktur zwischen Teilnehmern und Nicht-Teilnehmern.

9. Durchführen des Matching

In diesem Abschnitt führen wir das Matching des PSM durch. Hierzu definieren wir zunächst Klassen auf den beschreibenden Variablen. Anschließend suchen wir zu jedem Teilnehmer (mit "teilnahmslosen" Zeiten) die Elemente, die in die gleichen Klassen fallen und suchen dann maximal fünf Elemente mit dem geringsten Wahrscheinlichkeitsabstand.

9.1 Zusammenfassung der Leistungsarten

Zuerst fassen wir verschiedenen die Leistungsarten zusammen.

```
In [104]: psm = pandas.HDFStore("Wahltarif_%s.hdf" % WT)["PSM"]
         assert psm.Wahltarif.sum() == 0
```

```
In [105]: kosten = ['Arzneimittel', 'Arztbehandlung', 'Heil_Hilfsmittel', 'Krankengeld', 'Krankenhaus', 'Schwangerschaft', '
         psm["Kosten"] = psm[kosten].sum(axis=1)
         del psm["Wahltarif"]
         for k in kosten:
             del psm[k]
```

9.2 Alters- und Kostenklassen

Im zweiten Schritt bilden wir Alters- und Kostenklassen. Letztere bilden wir logarithmisch.

```
In [106]: %%time
         psm["Alter_CD"] = psm.Alter.apply(altersgruppe)
         psm["Kosten_CD"] = psm.Kosten.apply(kostengruppe)
         psm["Mitversicherung"] = psm.Mitversicherung.astype(numpy.int)
         del psm["Alter"]
         del psm["Kosten"]
```

Wall time: 421 ms

9.3 Gruppieren der Teilnehmer

Wir bilden nun die Gruppen der Teilnehmer und der Nicht-Teilnehmer.

```
In [107]: %%time
         psm.set_index("ID", inplace=True)
         felder = list(psm.columns)
         del felder[-4:-2]      # Teilnahme & WT_Propability
         gruppen = psm.query("Teilnahme == 0").groupby(felder).groups
         tn_gruppen = psm.query("Teilnahme == 1").groupby(felder).groups
```

Wall time: 11.4 s

```
In [108]: print("Anzahl betrachteter Mitglieder:", locale.format("%d", len(psm), grouping=True))
         print("Anzahl der insgesamt vorhandenen Gruppen:", locale.format("%d", len(gruppen), grouping=True))
         print("Anzahl der Gruppen von Wahltarifteilnehmern:", locale.format("%d", len(tn_gruppen), grouping=True))
```

```
Anzahl betrachteter Mitglieder: 363.140
Anzahl der insgesamt vorhandenen Gruppen: 72.568
Anzahl der Gruppen von Wahltarifteilnehmern: 5.847
```

Wir bauen jetzt ein "Dictionary" namens teilnehmer (oder auch eine "Hash-Tabelle") auf, das für jeden Teilnehmer die Nichtteilnehmer der gleichen Gruppe sowie den "Wahrscheinlichkeitsabstand" enthält, sofern deren Teilnahmewahrscheinlichkeit die Sensitivitätsschwelle von 1 % für den maximalen zulässigen Wahrscheinlichkeits-Abstand unterschreitet.

```
In [109]: %%time
         max_delta = 0.01
         teilnehmer = {}
         for gr, tn_l in tn_gruppen.items():
             for tn in tn_l:
                 g = gruppen.get(gr, [])[:]
                 #g.remove(tn)
                 prop = psm.at[tn, "WT_Propability"]
                 res = []
                 for x in g:
                     delta_prop = abs(prop - psm.at[x, "WT_Propability"])
                     if delta_prop <= max_delta:
                         res.append((x, delta_prop))
                 res.sort(key=lambda z: z[1])
                 teilnehmer[tn] = res[0:min(5, len(res))]
```

Wall time: 5.02 s

Wir betrachten die Teilnahmewahrscheinlichkeit und die gefundene Vergleichsgruppe für die ID 2416001024.

```
In [110]: print("PSM:", locale.format_string("%1.2f %%", 100 * psm.at[2416001024, "WT_Propability"]))
```

PSM: 5,24 %

```
In [111]: for id, p in teilnehmer[2416001024]:
           print("PSM-Unterschied für ID %s:" % id, locale.format_string("%1.2f %%", 100*p, grouping=True))
```

PSM-Unterschied für ID 5986321101: 0,37 %

PSM-Unterschied für ID 2586602015: 0,57 %

PSM-Unterschied für ID 9585301005: 0,96 %

Die meisten Vergleichsgruppen umfassen die Obergrenze von fünf Elementen, aber ca. tausend sind auch leer und somit für das PSM nicht unmittelbar nutzbar. Bei unserem Verfahren ignorieren wir diese Elemente.

```
In [112]: from collections import defaultdict
           hist = defaultdict(int)
           for x in teilnehmer.values():
               hist[len(x)] += 1
           plot = pandas.Series(hist).plot()
           _ = plot.yaxis.set_major_formatter(formatter())
           _ = plot.set_title("Vergleichsgruppen-Größe")
           print("Anzahlen der verschiedenen Gruppengrößen")
           for key, anz in hist.items():
               print("Größe %d:" % key, locale.format("%5d", anz, grouping=True))
           print("Anzahl Wahltarifteilnehmer mit nichtleerer Vergleichsgruppe:", locale.format("%d", sum(hist.values()) - hist[0]))
```

Anzahlen der verschiedenen Gruppengrößen

Größe 0: 1.106

Größe 3: 354

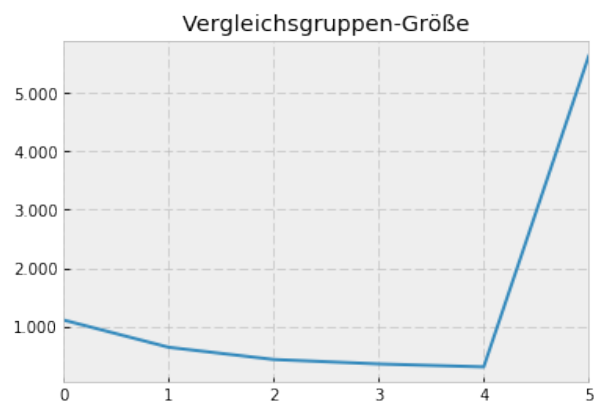
Größe 4: 303

Größe 1: 639

Größe 5: 5.635

Größe 2: 430

Anzahl Wahltarifteilnehmer mit nichtleerer Vergleichsgruppe: 7.361



10. Ergebnis

Wir können für die Teilnehmer und die Vergleichsgruppe jetzt die Leistungsdaten heranziehen. Für jeden Teilnehmer betrachten wir den gesamten durchschnittlichen Leistungsaufwand in den Jahren 2010 - 2012 (weil die Wahltarife erst 2010 eingeführt worden sind) jeweils mit und ohne Wahltarif, für die Vergleichsgruppe natürlich nur den ohne Wahltarif.

```
In [113]: lstg = pandas.concat((pandas.HDFStore("Wahltarif_%s.hdf" % WT)["Normleistungen%d" % j] for j in range(2010, 2013)))
          lstg = lstg.groupby(["ID", "WT_CD"]).sum()
```

```
In [114]: lstg.head()
```

```
Out[114]:
```

		Arzneimittel	Arztbehandlung	Heil_Hilfsmittel	Krankengeld \
ID	WT_CD				
6003	0	32,70	0,00	0,00	0,00
10105	0	285,94	490,54	0,00	0,00
92902	0	21,62	183,16	0,00	0,00
168602	0	11,52	-3,52	0,00	0,00
203012	0	43,83	198,70	0,00	1.945,57

		Krankenhaus	Schwangerschaft	Sonstiges
ID	WT_CD			
6003	0	0,00	0,00	0,00
10105	0	0,00	0,00	40,00
92902	0	0,00	0,00	134,11
168602	0	0,00	0,00	30,00
203012	0	6.915,49	0,00	839,44

Zunächst beschränken wir uns auf die Gesamtkosten, d.h. wir ignorieren die Verteilung auf einzelne Kostenarten.

```
In [115]: felder = ['Arzneimittel', 'Arztbehandlung', 'Heil_Hilfsmittel', 'Krankengeld', 'Krankenhaus', 'Schwangerschaft', 'Sonstiges']
          lstg["Kosten"] = lstg[felder].sum(axis=1)
          for f in felder:
              del lstg[f]
```

```
In [116]: lstg.head()
```

```
Out[116]:
```

		Kosten
ID	WT_CD	
6003	0	32,70
10105	0	816,48
92902	0	338,89
168602	0	38,01
203012	0	9.943,03

Für jeden Wahltarifteilnehmer ermitteln wir die folgenden drei Größen: - Kosten im Zeitraum der Wahltarifteilnahme - Kosten im Zeitraum ohne Wahltarifteilnahme - Durchschnittskosten der PSM-Referenzgruppe

Anschließend summieren wir die Ergebnisse über alle Teilnehmer auf und ermitteln die Ersparnis durch die Wahltarifteilnahme.

Hierbei nutzen wir aus, dass wir gar nicht alleine auf PSM angewiesen sind, sondern zu den Wahltarifteilnehmern die Zeiten mit und ohne Wahltarifteilnahme zu vergleichen. Dies ermöglicht einen Vergleich der beiden Verfahren.

```
In [117]: %%time
          for tn, ref_l in teilnehmer.items():
              res = {"Kosten_WT": 0,
                     "Kosten_ohne_WT": 0,
                     "Referenzgruppe": ref_l,
                     "Anzahl_Referenz": 0}
              if (tn, 1) in lstg.index:
                  res["Kosten_WT"] = lstg.at[(tn, 1), "Kosten"]
              if (tn, 0) in lstg.index:
                  res["Kosten_ohne_WT"] = lstg.at[(tn, 0), "Kosten"]
              kosten = 0
```

```

anzahl = 0
for ref, abstand in ref_l:
    if (ref, 0) in lstg.index:
        kosten += lstg.at[(ref, 0), "Kosten"]
        anzahl += 1
if anzahl > 0:
    kosten /= anzahl
res["Referenzgruppe"] = kosten
res["Anzahl_Referenz"] = anzahl
teilnehmer[tn] = res

```

Wall time: 2.06 s

10.1 Kostenersparnis Wahltarif-freie Zeiten

Kostenersparnis bei Vergleich der Zeiten mit und ohne Wahltarif-Teilnahme der Teilnehmer.

```

In [118]: # Wahltarif-freie Zeiten
anzahl = k_wt = k_owt = 0
for tn, val in teilnehmer.items():
    k_wt += val["Kosten_WT"]
    k_owt += val["Kosten_ohne_WT"]
    anzahl += 1
print("Durchschnittliche Kostenersparnis Wahltarif-freie Zeiten:", locale.format("%.2f", ((k_owt - k_wt) / anzahl)

```

Durchschnittliche Kostenersparnis Wahltarif-freie Zeiten: 763,56 €

10.2 Kostenersparnis PSM

Kostenersparnis bei Nutzung der PSM-Methode (ohne Berücksichtigung von Selbstselektion).

```

In [119]: # PSM-Methode
anzahl = k_wt = k_ref = 0
for tn, val in teilnehmer.items():
    if val["Anzahl_Referenz"] > 0:      # wir betrachten nur Teilnehmer mit nicht-leerer Referenzgruppe
        k_wt += val["Kosten_WT"]
        k_ref += val["Referenzgruppe"]
        anzahl += 1
print("Durchschnittliche Kostenersparnis PSM-Methode:", locale.format("%.2f", ((k_ref - k_wt) / anzahl), grouping=

```

Durchschnittliche Kostenersparnis PSM-Methode: 1.104,57 €

10.3 Weitere Betrachtungen

Insgesamt ist der Unterschied in den erbrachten Leistungen für Wahltarif-Teilnehmer und Nicht-Teilnehmer wie zu erwarten deutliche größer.

```

In [120]: print("Durchschnittliche Kosten für Nicht-Teilnehmer:",
    locale.format_string("%.2f €", lstg.query("WT_CD == 0").mean(), grouping=True))
print("Durchschnittliche Kosten für Teilnehmer:",
    locale.format_string("%.2f €", lstg.query("WT_CD == 1").mean(), grouping=True))
print("Kostenrelation:", locale.format("%.2f", lstg.query("WT_CD == 0").mean() / lstg.query("WT_CD == 1").mean()),
print("Kostenrelation nach PSM:", locale.format("%.2f", k_ref / k_wt, grouping=True))

```

Durchschnittliche Kosten für Nicht-Teilnehmer: 5.632,30 €

Durchschnittliche Kosten für Teilnehmer: 1.365,34 €

Kostenrelation: 4,13

Kostenrelation nach PSM: 1,90



© Thierry Gregorious CC-BY

Kontaktdaten



Dr. Axel Kaiser
Aktuar DAV
Senior Manager
Branchencenter
Versicherungen

Fuhlentwiete 12
20355 Hamburg
Tel: +49 40 30293-774
Fax: +49 40 30293-323
www.bdo.de

axel.kaiser@bdo.de

BDO AG Wirtschaftsprüfungsgesellschaft, eine Aktiengesellschaft deutschen Rechts, ist Mitglied von BDO International Limited, einer britischen Gesellschaft mit beschränkter Nachschusspflicht, und gehört zum internationalen BDO Netzwerk voneinander unabhängiger Mitgliedsfirmen.