Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

2024 FS CAS PML
1 Feature Engineering
1.3 Transformation

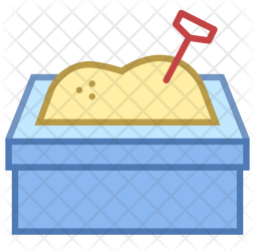
Werner Dähler 2024

1 Feature Engineering - AGENDA

11. Einführung
12. Exploration
13. Transformation
    131. Data Frame
    132. Kategoriale Variablen
    133. Numerische Variablen
    134. Bereinigen von Variablennamen
    135. Ändern von Datentypen
14. Konstruktion
15. Selektion
16. Implementation
17. Nachträge

1.3 Feature Engineering - Transformation

- unter Feature Exploration wurden die Daten gesichtet auch mit Blick auf mögliche Anomalien, welche zuhanden Machine Learning zu bereinigen sein werden (vgl. auch Kap. 1.1.6 Anforderungen an die Daten)
- in diesem Kapitel werden verschiedene Verfahren zur Datenbereinigung vorgestellt und mögliche Alternativen verglichen
- da sich einzelne Transformationsschritte aber auf andere auswirken können, wird nach dem Sandbox-Prinzip gearbeitet, d.h. die Untersuchungen werden jeweils auf einer Kopie der Ausgangsdaten (`ori_data`) durchgeführt, damit sie nicht jedes Mal neu geladen werden müssen



```
ori_data = pd.read_csv('bank_data.csv', sep=';')
```

- jeder Codeblock beginnt danach mit folgender Anweisung

```
data = ori_data.copy()
```

- eine konsolidierte Data Preparation erfolgt dann unter Kap. 1.6 Feature Engineering - Implementation

1.3 Feature Engineering - Transformation

Gliederung der folgenden Tätigkeiten mit Sicht auf

- Data Frame
- Kategoriale Variablen
- Numerische Variablen
- weitere Bereinigungen

1.3 Feature Engineering - Transformation

1.3.1 Data Frame

1.3.1.1 Entfernen von Beobachtungen

Nach Index

- Gezieltes Entfernen von Beobachtungen nach Index, z.B. Entfernen der Beobachtungen mit dem grössten Wert von "age" aus dem Data Frame
- dazu wird der Index der entsprechenden Beobachtung(en) ermittelt und in einer Liste hinterlegt, welche beim folgenden Aufruf von `.drop()` als Parameter mitgegeben wird

```
idx = (data.age[data.age == max(data.age)].index[0]).tolist()
data.drop(idx, inplace=True)
```

1.3 Feature Engineering - Transformation

1.3.1 Data Frame

1.3.1.1 Entfernen von Beobachtungen

Nach Bedingung

- Entfernen von Beobachtungen nach Bedingung(en), z.B. Entfernen aller Beobachtungen, für welche in der Variable "age" der Wert >= 100 ist

```
data.drop(data[data.age >= 100].index, inplace=True)
```

- **Entfernen von Duplikaten**

```
data.drop_duplicates(ignore_index=True, inplace=True)
```

1.3 Feature Engineering - Transformation

1.3.1 Data Frame

1.3.1.2 Entfernen von Variablen

Nach Index

- Entfernen von Variablen nach deren Spaltenindex im Data Frame (beginnend bei 0), z.B. entfernen der ersten drei Variablen (von links)

```
cols_to_drop = [0, 1, 2]
data.drop(data.columns[cols_to_drop], axis=1, inplace=True)
```

- aus Transparenzgründen wird erst eine Liste mit den Indices der zu entfernenden Variablen erstellt, welche der anschliessenden Methode .drop() als Parameter mitgegeben wird
- könnte aber auch kompakter formuliert werden:

1.3 Feature Engineering - Transformation

1.3.1 Data Frame

1.3.1.2 Entfernen von Variablen

Nach Name

- gezieltes Entfernen von Variablen nach deren Namen im Data Frame, z.B. entfernen von "marital" und "education"

```
cols_to_drop = ['marital', 'education']
data.drop(cols_to_drop, axis=1, inplace=True)
```

- auch hier wird mit zwei Schritten gearbeitet, welche kompakter codiert werden könnte (vgl. [ipynb])

1.3 Feature Engineering - Transformation

1.3.1 Data Frame

1.3.1.3. Einsetzen von Werten für Missing Values

- die "brachialen" Methoden, um NAs loszuwerden
  - entfernen aller Beobachtungen (rows) mit NAs
  - entfernen aller Variablen (columns) mit NAs

  können unter Umständen zu grossem Datenverlust führen, vgl. Ergebnisse von Feature Exploration

- Alternativen:
  - einsetzen eines willkürlichen Wertes
  - einsetzen eines errechneten Wertes (z.B. Modalwert bei Kategorialen Variablen, Median bei Numerischen Variablen), **was im Folgenden gezeigt werden soll**
  - einsetzen eines (mittels ML) geschätzten wahrscheinlichsten Wertes

1.3 Feature Engineering - Transformation

1.3.1 Data Frame

1.3.1.3. Einsetzen von Werten für Missing Values

Kategoriale Variablen

- Ersetzen von NAs bei "marital" durch den Modalwert aller nicht-NA Werte dieser Variablen

```
data.marital.fillna(data.marital.mode()[0], inplace=True)
```

- der Index `[0]` bei der Methode `.mode()` ist hier wichtig, da letztere mehrere Werte zurückgeben könnte und daher eine Liste ist

Numerische Variablen

- Ersetzen von NAs bei "age" durch den Median aller nicht-NA Werte dieser Variablen

```
data.age.fillna(data.age.median(), inplace=True)
```

1.3 Feature Engineering - Transformation

1.3.1 Data Frame

1.3.1.3. Einsetzen von Werten für Missing Values

Mehrere Variablen gleichzeitig

- mit einem Loop-Konstrukt können zwar mehrere Variablen desselben Datentyps entsprechend bearbeitet werden, dies bedingt aber etwas anspruchsvolleren Codieraufwand
- scikit-learn bietet mit der Klasse `sklearn.impute.SimpleImputer` eine Möglichkeit, dies mit wenigen Anweisungen zu erledigen
- da für kategoriale und numerische Variablen unterschiedliche Strategien einzusetzen sind, werden als Vorbereitung wiederum Listen mit den entsprechenden Variablennamen erstellt (vgl. Kap. 1.2)

```
cat_vars = data.select_dtypes(include='object').columns.tolist()
num_vars = data.select_dtypes(exclude='object').columns.tolist()
```

1.3 Feature Engineering - Transformation

1.3.1 Data Frame

1.3.1.3. Einsetzen von Werten für Missing Values

Mehrere Variablen gleichzeitig

- die danach folgenden Schritte folgen dem API-Konzept von scikit-learn (wie wir es dann unter Supervised Learning immer wieder sehen werden):
  - importieren einer Trainerklasse
  - instanziieren eines Trainerobjektes
  - trainieren und anwenden dieses Objektes auf die Daten

```
from sklearn.impute import SimpleImputer
imp_mode = SimpleImputer(missing_values=np.nan, strategy='most_frequent')
data[cat_vars] = pd.DataFrame(imp_mode.fit_transform(data[cat_vars]),
    columns=data[cat_vars].columns)
```

- "most_frequent" steht für den Modalwert
- analog kann mit den numerischen Variablen vorgegangen werden (`strategy='median'`)

1.3 Feature Engineering - Transformation

1.3.2 Kategoriale Variablen

1.3.2.1 Reduzieren der Kardinalität

- unter Feature Exploration wurde festgestellt, dass z.B. für die Variable "education" eine Kategorie (Level) mit vergleichsweise wenigen Werten vorliegt: "illiterate"
- falls dies vom Fach so akzeptiert wird, kann diese Kategorie mit "basic.4y" zusammengelegt werden

```
data.education = np.where(
    data.education == 'illiterate', ## condition
    'basic.4y',                     ## if true
    data.education)                 ## if false
```

- dazu besonders geeignet ist die Funktion `where` aus dem Package `numpy(np)`
- bedarfsweise können auch mehrere Kategorien zu einer zusammenkombiniert werden, indem mehrere Bedingungen verknüpft werden (vgl. [ipynb])

1.3 Feature Engineering - Transformation

1.3.2 Kategoriale Variablen

1.3.2.2 Numerisieren

- um Informationen aus Kategorialen Variablen in Machine Learning verwenden zu können, muss diese numerisch dargestellt werden können (vgl. 1.1.6)
- dazu stehen verschiedene Möglichkeiten zur Verfügung
  - Faktorisieren
  - Ordinal Encodieren
  - Nominal Encodieren,
    welche im Folgenden vorgestellt werden

1.3 Feature Engineering - Transformation

1.3.2 Kategoriale Variablen

1.3.2.2 Numerisieren - 1.3.2.2.1 Faktorisieren

- jeder Kategorie einer Kategorialen Variablen wird ein Integer-Wert zugeordnet (beginnend bei 0)
- z.B. für die Variable "job":

```
data.job = pd.factorize(data.job)[0]
```

- tatsächlich gibt `.factorize()` ein Tupple mit folgende Komponenten zurück
  - `numpy.ndarray`: faktorisierte Werte, beginnend bei 0
  - `pandas.core.indexes.base.Index`: Zuordnungen der obigen Werte zu den Ausgangswerten (erstes Element für 0)
- mit dem oben stehenden Code können die Komponenten gleich beim Aufruf aufgetrennt werden (`[0]` nach `.factorize()`)
- per Default werden die Faktorwerte gemäss ihrem sequentiellen Auftreten vergeben
- mit dem Parameter sort=True werden die Faktorwerte lexikografisch in Bezug auf die Ausgangswerte vergeben

1.3 Feature Engineering - Transformation

1.3.2 Kategoriale Variablen

1.3.2.2 Numerisieren - 1.3.2.2.2 Ordinal Encodieren

- eine Schwäche von Faktorisieren: die numerischen Werte werden scheinbar willkürlich zugeordnet (nach deren sequenziellen Auftreten im Dataset)
- um also eine als ordinal skaliert erkannte Variable korrekt zu numerisieren, müssen die gezielt den einzelnen Kategorien zugeordnet werden können
- am Beispiel der Variable "education" könnte eine solche Zuordnung wie folgt aussehen:

illiterate	→	0
unknown	→	0
basic.4y	→	1
basic.6y	→	2
basic.9y	→	3
professional.course	→	4
high.school	→	5
university.degree	→	6

1.3 Feature Engineering - Transformation

1.3.2 Kategoriale Variablen

1.3.2.2 Numerisieren - 1.3.2.2.2 Ordinal Encodieren

- als dazu geeignete Funktion wird hier `.replace()` eingesetzt, welche als Parameter ein Python-Dictionary entgegennimmt, welches idealerweise in einem vorherigen Schritt definiert wird

```
replace_nums = {    ## a dictionary of dictionaries
    'education': {
        'illiterate': 0,
        'unknown': 0,
        'basic.4y': 1,
        'basic.6y': 2,
        'basic.9y': 3,
        'professional.course': 4,
        'high.school': 5,
        'university.degree': 6
    }
}
```

1.3 Feature Engineering - Transformation

1.3.2 Kategoriale Variablen

1.3.2.2 Numerisieren - 1.3.2.2.2 Ordinal Encodieren

- der effektive Aufruf:

```
data.replace(replace_nums, inplace=True)
```

- es können auch gleich mehrere Variablen mit einem Aufruf encodiert werden (vgl. [ipynb])

- es gibt zwar folgenden Encoder von scikit-learn:
  `sklearn.preprocessing.OrdinalEncoder`
- macht aber tatsächlich eine Faktorisierung, für effaktiv ordinales Encodieren ist wesentlich anspruchsvollere Parametrisierung notwendig

1.3 Feature Engineering - Transformation

1.3.2 Kategoriale Variablen

1.3.2.2 Numerisieren - 1.3.2.2.2 Ordinal Encodieren

Spezialfall: 0-1 Encodieren

- falls Kategoriale Variablen nur zwei Kategorien aufweisen, können diese auch einfach mit `numpy.where` encodiert werden
- so weist z.B. "contact" nur die Kategorien "cellular" und "telephone" auf

```
data['contact'] = np.where(data.contact == 'cellular', 1, 0)
```

- der obenstehende Aufruf weist dem Wert "cellular" neu den Wert 1 zu, allen anderen (!) den Wert 0

- aus Gründen der Transparenz kann es Sinn machen, die umcodierte Variable anschliessend umzubenennen, z.B. wie folgt:

```
data.rename(columns = {'contact' : 'contact_cellular'}, inplace=True)
```

1.3 Feature Engineering - Transformation

1.3.2 Kategoriale Variablen

1.3.2.2 Numerisieren - 1.3.2.2.3 Nominal Encodieren (aka One-Hot Encoding)

oder: Erstellen von Dummy-Variablen

- für die folgenden Darstellungen wird aus praktischen Gründen eine Zufallsstichprobe der Daten erstellt

```
data = ori_data.sample(6, random_state=1234)
print(data.marital)
```

```
9056      single
9483    divorced
788       single
9554    divorced
809     divorced
4822     married
```

1.3 Feature Engineering - Transformation

1.3.2 Kategoriale Variablen

1.3.2.2 Numerisieren - 1.3.2.2.3 Nominal Encodieren (aka One-Hot Encoding)

- die Pandas-Funktion .get_dummies() erstellt für jede auftretende Kategorie eine neue Variable (Dummy Variable) **und entfernt die Ausgangsvariable**
- dabei stehen verschiedene Parametrisierungsmöglichkeiten zur Verfügung
- für die folgenden Darstellungen wird das Ergebnis jeweils den Ausgangsdaten gegenübergestellt

```
new_data = pd.get_dummies(data.marital)
print(pd.merge(data.marital, new_data, left_index=True, right_index=True))
```

```
       marital  divorced  married  single
9056    single         0        0       1
9483  divorced         1        0       0
788     single         0        0       1
9554  divorced         1        0       0
809   divorced         1        0       0
4822   married         0        1       0
```

1.3 Feature Engineering - Transformation

1.3.2 Kategoriale Variablen

1.3.2.2 Numerisieren - 1.3.2.2.3 Nominal Encodieren (aka One-Hot Encoding)

- mit Parameter `drop_first`:

```
new_data = pd.get_dummies(data.marital, drop_first=True)
:
```

```
       marital  married  single
9056    single        0       1
9483  divorced        0       0
788     single        0       1
9554  divorced        0       0
809   divorced        0       0
4822   married        1       0
```

- es wird eine Dummy-Variable weniger erstellt, als ursprünglich Kategorien vorhanden sind (weshalb wohl?)

1.3 Feature Engineering - Transformation

1.3.2 Kategoriale Variablen

1.3.2.2 Numerisieren - 1.3.2.2.3 Nominal Encodieren (aka One-Hot Encoding)

- mit Parameter `prefix`:

```
new_data = pd.get_dummies(data.marital, prefix='marital')
:
```

	marital	marital_divorced	marital_married	marital_single
9056	single	0	0	1
9483	divorced	1	0	0
788	single	0	0	1
9554	divorced	1	0	0
809	divorced	1	0	0
4822	married	0	1	0

- dem Namen der Dummy-Variablen wird jeweils der Name der Ausgangsvariable als Präfix mitgegeben - ist vor allem dann angebracht, wenn mehrere Variablen auf diese Weise transformiert werden sollen

1.3 Feature Engineering - Transformation

1.3.2 Kategoriale Variablen

1.3.2.2 Numerisieren - 1.3.2.2.3 Nominal Encodieren (aka One-Hot Encoding)

- `.get_dummies()` kann auch gleich auf mehrere Variablen gleichzeitig angewendet werden

```
data = pd.get_dummies(
    data,
    columns=['job', 'marital', 'loan'],
    drop_first=True)
```

- da im obigen Aufruf der Parameter columns eine Liste der zu berücksichtigenden Variablen enthält, ist es auch möglich, für alle zum Zeitpunkt nicht kategorialen Variablen (ausser vielleicht dem Target) mit einem Aufruf Dummy-Variablen zu erstellen

```
target = 'y'
sel_vars = data.select_dtypes(include=['object']).columns.drop(target)
data = pd.get_dummies(data, columns=sel_vars, drop_first=True)
```

1.3 Feature Engineering - Transformation

1.3.2 Kategoriale Variablen

1.3.2.2 Numerisieren - 1.3.2.2.3 Nominal Encodieren (aka One-Hot Encoding)

ein Nachtrag zu scikit-learn

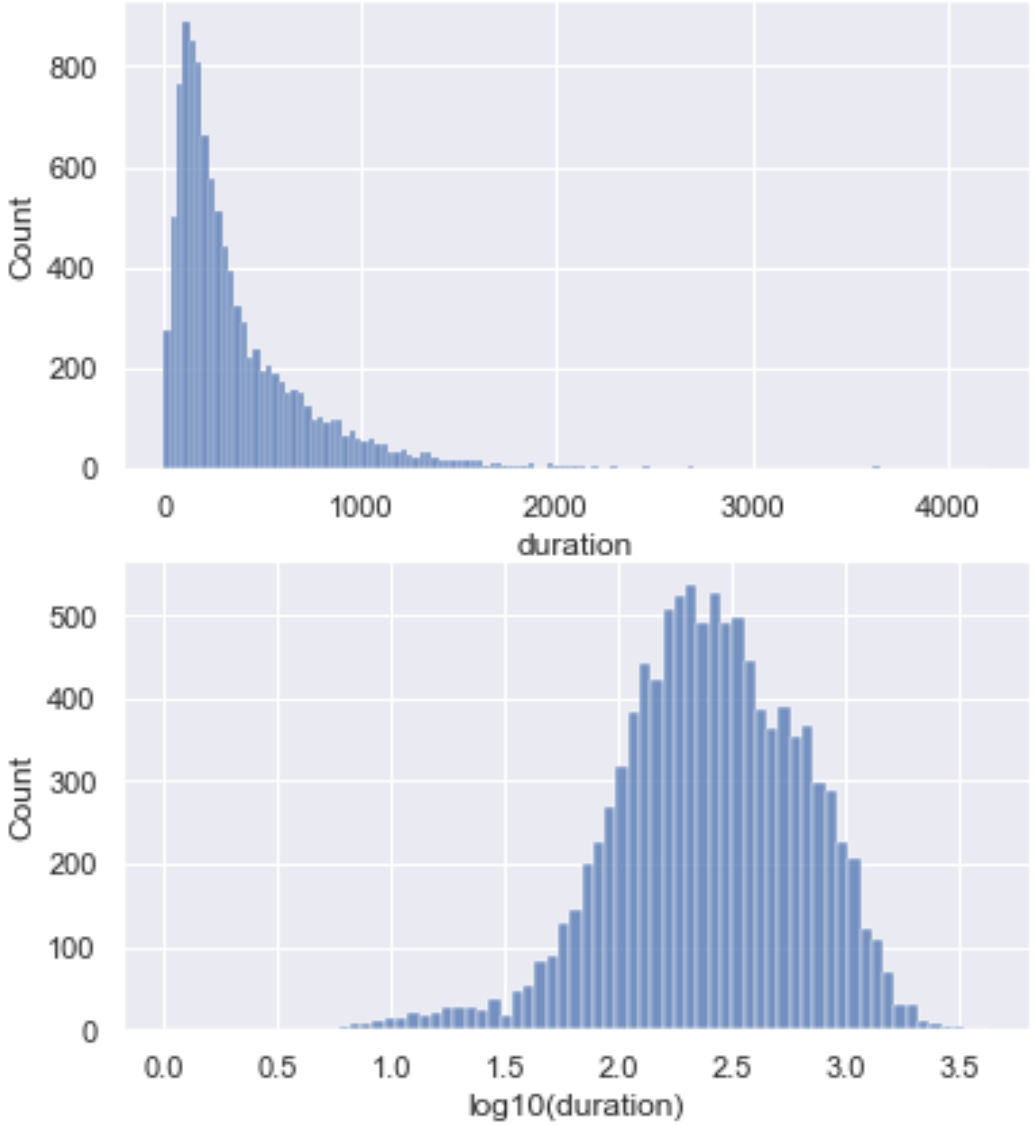
- `sklearn.preprocessing.OneHotEncoder`
- wesentlich aufwändiger zu parametrisieren
- Herausforderungen:
  - bringt einen np.ndarray zurück
  - keine Featuernamen
  - insbesondere schwierig in der Anwendung, wenn nur ein Subset der Variablen bearbeitet werden soll
- Empfehlung daher: lieber nicht!

1.3 Feature Engineering - Transformation

1.3.3 Numerische Variablen

1.3.3.1 Logarithmieren

- wie wir gesehen haben, ist z.B. "duration" extrem schief verteilt (oben)
- durch Logarithmieren lässt sich die Verteilung u.U. symmetrischer darstellen (unten)
- dabei sind folgende Punkte zu beachten:
  - es stehen von `numpy` Logarithmus-Funktionen zu unterschiedlichen Basen zur Verfügung (e, 2, 10), hier wird log10 verwendet, um den Ausgangswert einfacher abschätzen zu können
  - vor dem Logarithmieren wird zusätzlich jeder Ausgangswert um 1 erhöht



1.3 Feature Engineering - Transformation

1.3.3 Numerische Variablen

1.3.3.1 Logarithmieren

Begründung

- logarithmieren von Werten ≤ 0 führt zu undefinierten Werten
- logarithmieren von Werten > 0 und < 1 führt zu negativen Werten, je näher der Ausgangswert bei 0 liegt, umso grösser wird der negative Betrag des Logarithmus
- wenn also negative Werte ausgeschlossen werden können, erfolgt die Berechnung nach der Form $x' = log(x + 1)$

- andernfalls $x' = log(x - \min(x) + 1)$

  was sicherstellt, dass der kleinste Ausgangswert 1 und der daraus ermittelte Logarithmus 0 wird

1.3 Feature Engineering - Transformation

1.3.3 Numerische Variablen

1.3.3.1 Logarithmieren

```
data.duration = np.log10(data.duration + 1)
```

- bei der Kontrolle der Ergebnisse (vgl. [ipynb]) nähern sich Mittelwert und Median einander relativ an, da sich die beiden Werte voneinander entfernen, je unsymmetrischer eine Verteilung ist
- falls das Minimum der zu logarithmierenden Werte < 0 ist, kann der Code wie folgt ergänzt werden

```
data.duration = np.log10(data.duration - data.duration.min() + 1)
```

1.3 Feature Engineering - Transformation

1.3.3 Numerische Variablen

1.3.3.1 Logarithmieren

ein Nachtrag zu Deskriptiven Kennzahlen

- neben den in Kap. 1.2.3.1 vorgestellten Kennzahlen gibt es noch weiterer Kennzahlen
- zwei davon sind insbesondere zur quantitativen Beurteilung von Verteilungen nützlich:
  - skew()
  - kurtosis()
- während Kurtosis hier keine Rolle spielt, kann mit Skewness de Einfluss auf die Symmetrie der Verteilung durch Logarithmieren auch quantitativ beurteilt werden
- Skewness (Schiefe) berechnet sich wie folgt:

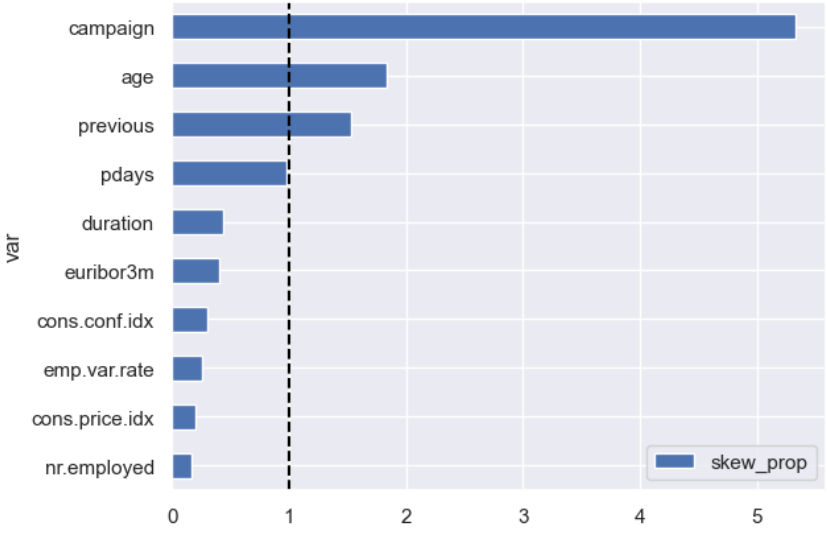
$$x = \frac{1}{n}\sum\left(\frac{x_{i-\bar{x}}}{s}\right)^3$$

1.3 Feature Engineering - Transformation

1.3.3 Numerische Variablen

1.3.3.1 Logarithmieren

- das Ergebnis wird wie folgt interpretiert:
  - negativer Wert: linksschiefe Verteilung
  - Wert nahe bei 0: symmetrische Verteilung
  - positiver Wert: rechtsschiefe Verteilung
- im untenstehenden Beispiel wird für alle numerischen Variablen des Data Frame abs(skew()) vor und nach dem Logarithmieren ermittelt und danach der Quotient berechnet (vertikale Linie: kein Einfluss, vgl. [ipynb])
- Fazit:
  - für die folgenden Variablen verbessert sich die Symmetrie der Verteilungen durch Logarithmieren
    - stark: "campaign", "duration"
    - schwach: "previous", "age"
  - bei den übrigen verbessert sich die Situation dagegen nicht oder verschlechtert sich sogar



1.3 Feature Engineering - Transformation

1.3.3 Numerische Variablen

1.3.3.2 Skalieren

- bei gleichzeitiger Berücksichtigung mehrerer Variablen ist das Risiko hoch, dass unterschiedliche Masseinheiten die Resultate verfälschen können
  - ist insbesondere von Bedeutung bei Methoden des Unüberwachten Lernens sowie einzelnen Methoden des Überwachten Lernens
- grosse Einheiten (km, Mio. CHF) führen zu kleinen Zahlenwerten und umgekehrt
- die Masseinheit einer numerischen Variablen darf auf eine Auswertung resp. Training von Modellen keinen Einfluss haben
- die beiden im Folgenden vorgestellten Methoden umgehen derartige Probleme

- Normalisieren und Standardisieren sind Lineartransformationen, welche Einflüsse von Masseinheiten neutralisieren

1.3 Feature Engineering - Transformation

1.3.3 Numerische Variablen

1.3.3.2 Skalieren - 1.3.3.2.1 Normalisieren

- auch Min-Max-Skalierung genannt
- transformiert alle Einzelwerte derart linear, dass der kleinste Wert = 0, der grösste = 1 wird

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

- bedarfsweise kann auch auf einen anderen Ziel-Range transformiert werden, z.B. 0-100

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)} * 100$$

1.3 Feature Engineering - Transformation

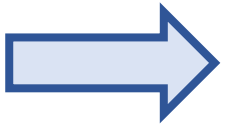
1.3.3 Numerische Variablen

1.3.3.2 Skalieren - 1.3.3.2.1 Normalisieren

- Vorgehen Normalisieren am Beispiel von "age"

```
data.age = (data.age - data.age.min()) / \
    (data.age.max() - data.age.min())
```

```
min     17.0
max    116.0
```



```
min    0.0
max    1.0
```

1.3 Feature Engineering - Transformation

1.3.3 Numerische Variablen

1.3.3.2 Skalieren - 1.3.3.2.2 Standardisieren

- Standardisieren wird auch als Kombination von Zentrieren und Skalieren bezeichnet
  - Zentrieren: von jedem Einzelwert wird der Mittelwert (aller Werte) subtrahiert:
    das hat zur Folge, dass der neue Mittelwert = 0 wird

$$x' = x - \bar{x}$$

    - wobei:

$$\bar{x} = \frac{1}{n}\sum x$$

  - Skalieren: jeder Einzelwert wird durch die Standardabweichung aller Werte dividiert:
    das hat zur Folge, dass die Standardabweichung des transformierten Wertes = 1 wird

$$x' = \frac{x}{s_x}$$

    (der Begriff "Skalieren" wird an dieser Stelle als im Engeren Sinn verstanden, nicht zu verwechseln mit Skalieren als Überbegriff von Normalisieren und Standardisieren)

1.3 Feature Engineering - Transformation

1.3.3 Numerische Variablen

1.3.3.2 Skalieren - 1.3.3.2.2 Standardisieren

- wobei

$$s_x = \sqrt{\frac{1}{n-1}\sum(x-\bar{x})^2}$$

- zentrieren und skalieren kombiniert führt zu folgender Formulierung

$$x' = \frac{x-\bar{x}}{s_x}$$

- diese Art von Transformation wird in der Literatur (zu Statistik) auch als z-Transformation bezeichnet

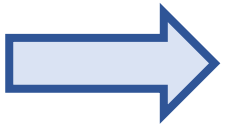
1.3 Feature Engineering - Transformation

1.3.3 Numerische Variablen

1.3.3.2 Skalieren - 1.3.3.2.2 Standardisieren

- Vorgehen Standardisieren am Beispiel von "age"

```
data.age = (data.age - data.age.mean()) / \
    data.age.std()
```

mean	40.423191		mean	1.795497e-17
std	11.915715		std	1.000000e+00

- (mean ist zwar nicht genau 0, aber immerhin nahezu, weshalb wohl?)

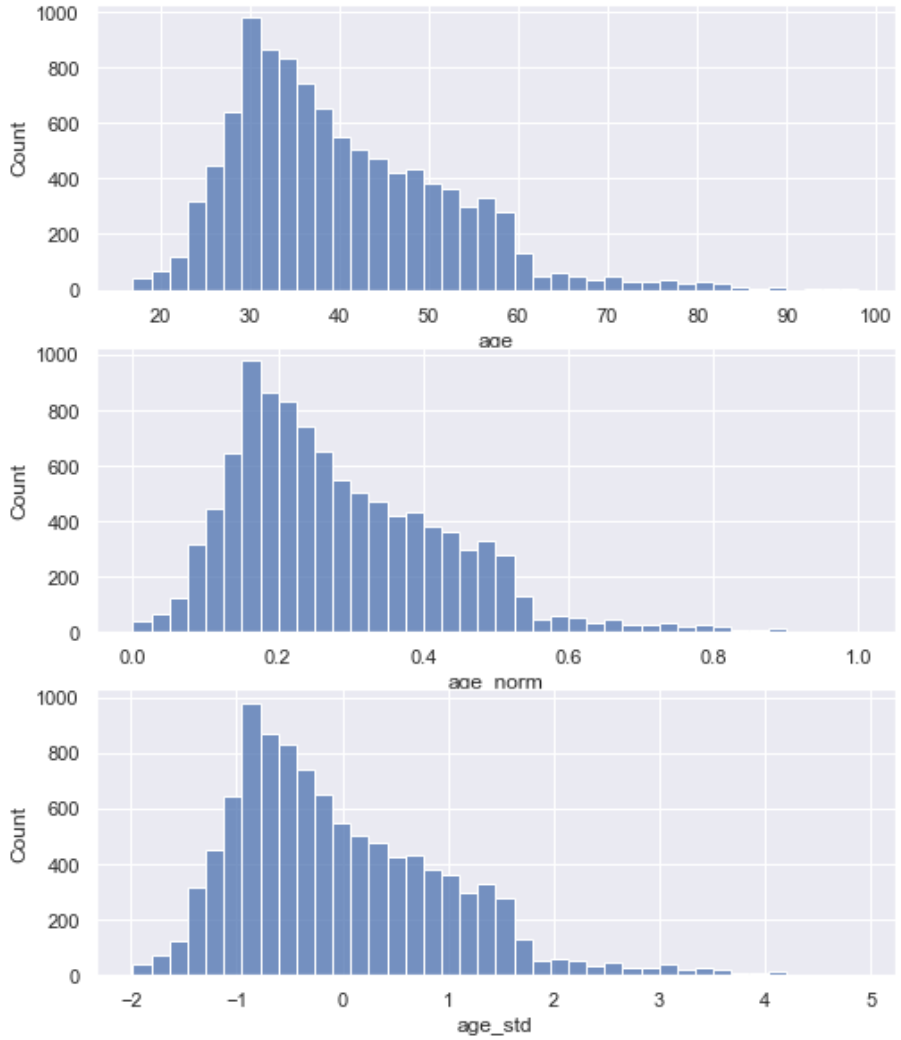
1.3 Feature Engineering - Transformation

1.3.3 Numerische Variablen

1.3.3.2 Skalieren

- eine visuelle Gegenüberstellung der beiden Verfahren ([ipynb])
  - Ausgangswerte (oben)
  - normalisiert (mitte)
  - standardisiert (unten)

- an der Verteilung ändert sich nichts, einziger Unterschied: Skala der x-Achse



1.3 Feature Engineering - Transformation

1.3.3 Numerische Variablen

1.3.3.2 Skalieren - 1.3.3.2.3 Skalieren mit `sklearn.preprocessing`

- das Ganze macht nur Sinn, wenn die jeweilige Transformation gleich auf **alle** interessierenden Variablen angewendet wird
- ausserdem muss unter Umständen die Transformation für späteren Gebrauch mit neuen Daten hinterlegt werden können
- die Library scikit-learn (sklearn) bietet dazu im Modul sklearn.preprocessing (unter anderen) die folgenden beiden Funktionen an:
  - `MinMaxScaler`
  - `StandardScaler`

1.3 Feature Engineering - Transformation

1.3.3 Numerische Variablen

1.3.3.2 Skalieren - 1.3.3.2.3 Skalieren mit `sklearn.preprocessing`

- Vorgehen Skalieren aller numerischen Variablen des Data Frame

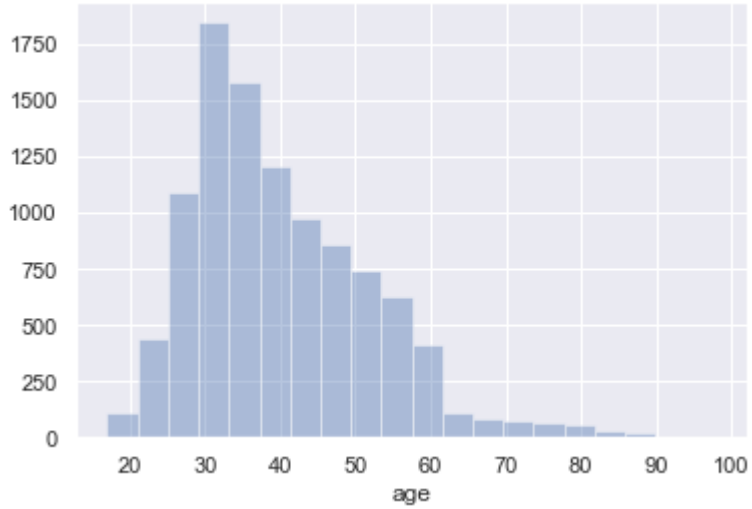
```
data = ori_data.select_dtypes(exclude=['object'])
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler().set_output(transform="pandas")
data = scaler.fit_transform(data)
```

- hier exemplarisch mit MinMaxScaler, welcher bei Standard-Parametrisierung eine Normalisierung (0-1 Transformation) durchführt
- die verschiedenen Funktionen sind im [ipynb] ausführlich kommentiert
- in den meisten Situationen werden diese Transformationen erst unmittelbar beim Trainieren eingesetzt, weshalb bei der Implementierung vorerst darauf verzichtet wird
- für spätere Verwendung können die Skalierungsmodelle (scaler) auch abgespeichert und wieder geladen werden (vgl. Kap. 2.1.1.4, 5.3)

1.3 Feature Engineering - Transformation

1.3.3 Numerische Variablen

1.3.3.3 Binning



- dieser Thematik sind wir bereits bei den Histogrammen begegnet
- für graphische Darstellung der Häufigkeitsverteilung wird intern ein Binning vorgenommen, um die Werte in Klassen einzuteilen
- anschliessend werden die absoluten bzw. relativee Klassenhäufigkeiten in einem Balkendiagramm dargestellt, (vgl. dazu Binning)
- für die Klassen werden (normalerweise) gleich grosse Intervalle gewählt

- Binning spielt auch bei einzelnen ML Methoden intern einer Rolle, da dadurch die Zeiten für Training und Prediction massiv verkürzt werden können (vgl. `HistGradientBoostingClassifier`, Kap. 2.2.4.2)

1.3 Feature Engineering - Transformation

1.3.3 Numerische Variablen

1.3.3.3 Binning

- für die Bestimmung der optimalen Anzahl Klassen bei der Erstellung von Histogrammen liegen allerdings unterschiedliche Vorstellungen vor:
  - seaborn.histplot: Binwidth gemäss Freedman–Diaconis rule
  - pandas.DataFrame.hist: 10 (dasselbe wie plt.hist), dabei wird der Bereich zwischen min und max konsequent in 10 gleich grosse Bereiche unterteilt, was wie bei seaborn.histplot zu unschönen Klassengrenzen führen kann
  - R hist(): Sturges Regel, ausserdem werden mit pretty() die Grenzen so gesetzt, dass sie durch 1, 2, oder 5 teilbar sind
  - R ggplot2::geom_histogram(): 30
- viele Autoren sind sich immerhin einig, dass es keinen "vernünftigen" Defaultwert gibt, dass also unterschiedliche Anzahlen Bins, resp. Binwidth experimentell ermittelt werden sollten

1.3 Feature Engineering - Transformation

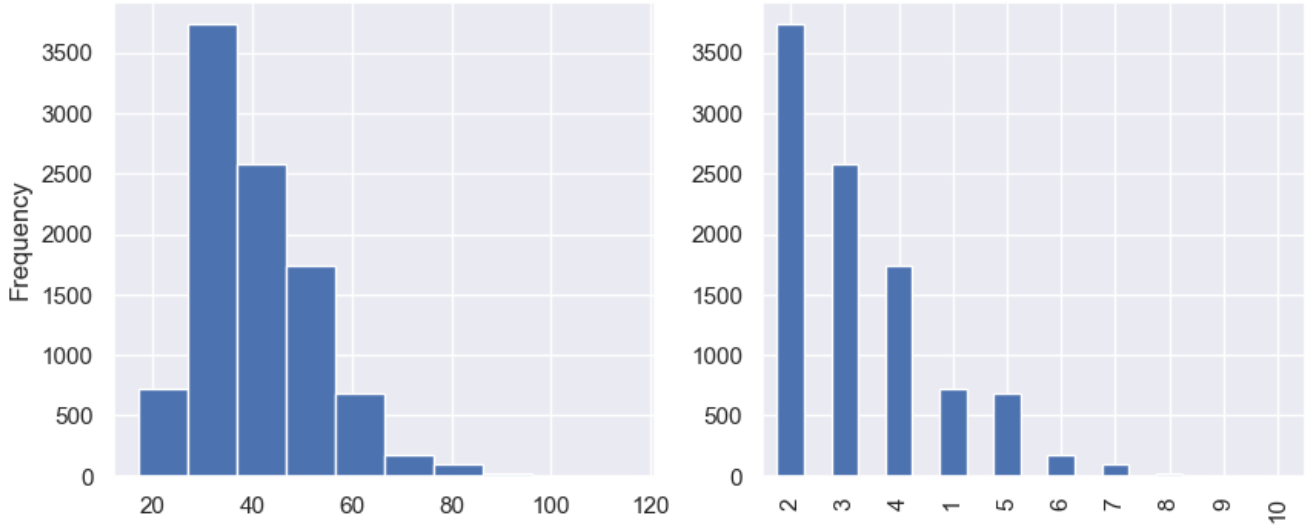
1.3.3 Numerische Variablen

1.3.3.3 Binning - 1.3.3.3.1 Equal Binning

- Vorgehen Equal Binning mit pd.cut am Beispiel von "age"

```
bins = 10
data.age = (pd.cut(
    data.age,
    bins = bins,
    labels = list(range(1, bins + 1))))
```

- links: Histogramm von "age" vor Binning
- rechts: Barplot nach Binning



1.3 Feature Engineering - Transformation

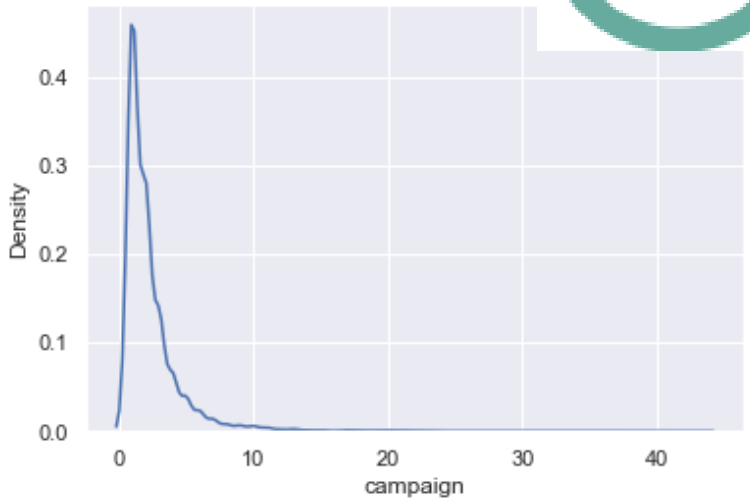
1.3.3 Numerische Variablen

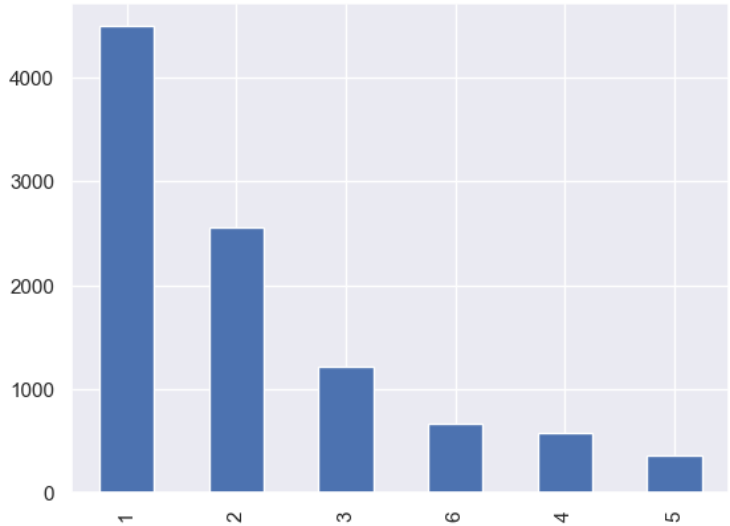
1.3.3.3 Binning - 1.3.3.3.2 Custom Binning

- "campaign" zeigt eine extrem (rechts-) schiefe Verteilung
- eine Analyse mit value_counts() zeigt darüber hinaus, dass z.B. Werte ab einer bestimmten Grenze auf deren Wert zurückgeschnitten werden könnten (hier Werte > 6 $\rightarrow$ 6
- dies kann mit einer einfachen Umformung erreicht werden:

```
data.campaign = np.where(
    data.campaign <= 5, data.campaign, 6)
```

- anschliessend visualisieren mit Barplot





1.3 Feature Engineering - Transformation

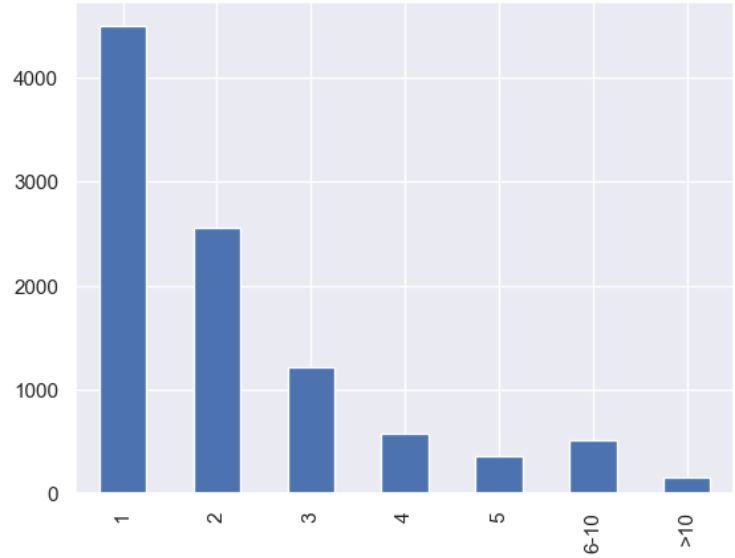
1.3.3 Numerische Variablen

1.3.3.3 Binning - 1.3.3.3.2 Custom Binning

- für unterschiedliche Bin-Bereiche können wiederum mit pandas.cut() unterschiedliche Bins definiert werden, z.B.
  - Werte von 1 bis 5 so übernehmen
  - zusätzlich je eine Klasse für
    - Werte von 6 - 10
    - Werte > 10

```
data.campaign = pd.cut(
    data.campaign,
    bins = [0, 1, 2, 3, 4, 5, 10, 1000],
    labels = [1, 2, 3, 4, 5, '6-10', '>10'])
```

- anschliessend visualisieren mit Barplot



1.3 Feature Engineering - Transformation

1.3.4 Bereinigen von Variablennamen

- durch Nominal Encodieren (One-Hot Encoding) können seltsame Variablennamen entstehen
- die Namen der entstehenden Dummy-Variablen setzen sich zusammen aus dem Namen der Ausgangsvariable sowie der Bezeichnung der jeweiligen Kategorie
- am Beispiel von "job" z.B. wie folgt:

```
data = pd.get_dummies(ori_data)
print(data.columns[data.columns.str.contains('job_')].tolist())
```

```
['job_admin.', 'job_blue collar', 'job_entrepreneur', 'job_housemaid',
'job_management', 'job_retired', 'job_self-employed', 'job_services',
'job_student', 'job_technician', 'job_unemployed']
```

- "job_blue collar": enthält ein Leerzeichen
- "job_self-employed" enthält einen Bindestrich

1.3 Feature Engineering - Transformation

1.3.4 Bereinigen von Variablennamen

- ausserdem enthalten die sozioökonomischen Variablen ("emp.var.rate", etc.) einen Punkt im Namen
- dies ist normalerweise kein Problem
- allerdings behandeln verschiedene Methoden des Machine Learning die Variablen (Features) als eigenständige Python-Objekte, und dort sind solche Zeichen in Bezeichnern nicht erlaubt
- mittels der Funktion `.str.contains()` und eines Regulären Ausdrucks (Regex) kann untersucht werden, welche Variablen ungünstige Zeichen enthalten, dabei werden die Namen angezeigt, welche andere als die erlaubten Zeichen (`a-z, A-Z, 0-9, _`) enthalten

```
old_names = data.columns
old_names = old_names[old_names.str.contains('[^a-zA-Z0-9_]')]
print(old_names.tolist()) ## check
```

```
['emp.var.rate', 'cons.price.idx', 'cons.conf.idx', 'nr.employed', 'job_admin.',
'job_blue collar', 'job_self-employed', 'education_basic.4y',
'education_basic.6y', 'education_basic.9y', 'education_high.school',...
```

1.3 Feature Engineering - Transformation

1.3.4 Bereinigen von Variablennamen

- mit der Funktion `.str.replace()` kann ebenfalls mit Unterstützung von Regex eine Liste erzeugt werden, in welcher alle unerlaubten Zeichen z.B. durch "_" ersetzt werden

```
new_names = old_names.str.replace('[^a-zA-Z0-9_]', '_', regex=True)
```

- die beiden oben erstellten Listen (old_names und new_names) können anschliessend verwendet werden, um die fraglichen Variablen im Data Frame umzubenennen

```
for i in range(len(old_names)):
    data.rename(columns={old_names[i]:new_names[i]}, inplace=True)
```

- im [ipynb] findet sich auch noch eine etwas elegantere Variante des Obigen mit `.dict(zip())`

1.3.5 Ändern von Datentypen

- z.B. ändern des Datentyps von "age" auf int32

```
data['age'] = np.int32(data['age'])
```