Neuronale Netze mit zensierten Daten

Marc Karic

26. März 2024

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Im Rahmen meiner Bachelorarbeit im Studiengang Data Science habe ich zu dem Thema Neuronale Netze mit zensierten Daten recherchiert und untersucht, wie man dies umsetzen könnte. Nachdem die Umsetzung klar war, wurde dies konkret an einem konkreten Beispiel mit realen Daten umgesetzt.
Um die zwei Themen Neuronale Netze und zensierte Daten zusammenzuführen ist es natürlich wichtig, sich für die jeweiligen Themen ein Verständnis aufzubauen, was in folgenden Kapiteln eingeführt wird. Um den Umfang der Komplexität des Lernens eines Neuronalen Netzes mit einer gewichteten Lossfunktion verstehen, wird dies außerdem an einem Beispiel dargestellt. Die Umsetzung wurde letztendlich an den Patientendaten der UK Biobank versucht. Dabei war es insbesondere interessant verschiedene Zensierungsraten auszuprobieren und zu schauen, welchen Effekt die Zensierung auf das Lernen des Neuronalen Netzes hat. Hierbei wurde versucht, die Überlebensdauer vorherzusagen, also wie lang ein Patient nach Einstieg in die Studie ohne Herz- oder Hirntumor überleben wird.

2 Rechts-zensierte Daten

In klinischen Studien werden häufig viele Patienten untersucht, wobei man sich dabei häufig für die Überlebensdauer der Patienten interessiert. Diese könnten während der Studie beispielsweise an einer Krankheit versterben (hier in rot). Damit wären sie vollständig beobachtet und man könnte die Überlebensdauer vom Eintritt der Studie bis zum Todeszeitpunkt berechnen. Sollte man nun aber beispielsweise den Kontakt zu dem Patienten im Laufe der Studie verlieren (hier in blau) oder der Patient überlebt länger, als die Beobachtungszeit der Studie (hier in orange), sind diese rechts-zensiert. Wir können also keine konkrete Überlebensdauer errechnen.

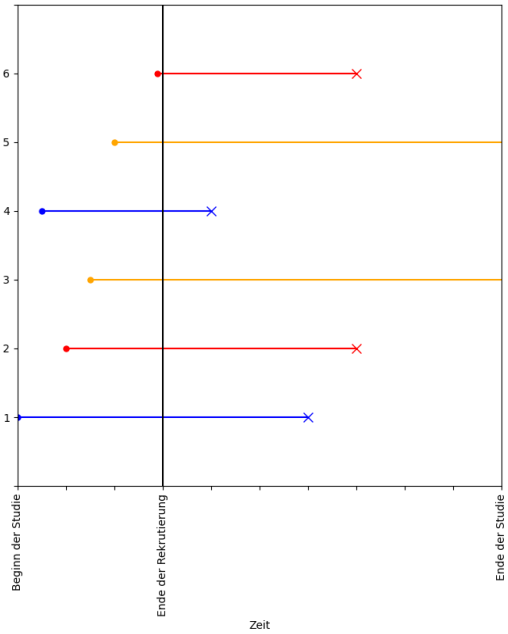

Abbildung 1: Möglicher Verlauf einer Survival-Studie
Quelle: Eigene Darstellung

Nun kann man die Beobachtungszeiträume verschieben, sodass diese für jeden Patienten zeitgleich starten und erhält somit die *beobachtete Lebensdauer*:

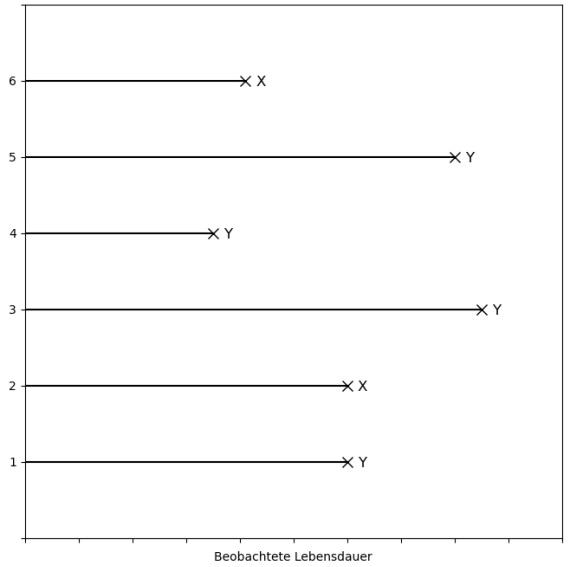

Abbildung 2: Möglicher Verlauf einer Survival-Studie
Quelle: Eigene Darstellung

In Abbildung 2 sind die Patienten 1, 3, 4 und 5 mit einem Y versehen, da sie rechts-zensiert sind. Ihre tatsächliche Lebensdauer kann also nicht beobachtet werden. Hingegen kann für die Patienten 2 und 6 die tatsächliche beobachtete Lebensdauer errechnet werden, da diese während der Studie an einer Krankheit verstorben sind. Diese sind in mit einem X versehen.

Mathematisch könnte man solch ein Szenario wie folgt formulieren:
Es existieren unabhängige, identische und nach einer Verteilungsfunktion F verteilte Folge von Zufallsvariablen $X_1, \ldots, X_n$. Diese enthalten die wahren Lebensdauern und die X_i könnten eventuell rechts-zensiert sein. Außerdem sind $Y_1, \ldots, Y_n$ von X_i unabhängige und identische Zufallsvariablen, die nach einer Verteilungsfunktion G verteilt sind. Diese enthalten die zensierten Lebensdauern. Dabei existiert dann

$$\begin{aligned} Z_i &= min\{X_i, Y_i\} \\ \delta_i &= 1_{\{X_i \leq Y_i\}} \\ &= \begin{cases} 1 \Leftrightarrow X_i \leq Y_i \\ 0 \Leftrightarrow X_i > Y_i \end{cases} \end{aligned}$$

wobei Z_i die beobachtete Lebensdauer ist und δ_i der zu dem jeweiligen Z_i zugehörige Zensierungsindikator. Wertet $\delta_i = 1$ aus, so wurde eine wahre Lebensdauer beobachtet, also $Z_i = X_i$. Sollte $\delta_i = 0$ auswerten, dann wurde eine zensierte Lebensdauer beobachtet, also $Z_i = Y_i < X_i$. Dementsprechend ist nur bekannt, dass ein Patient mindestens so lang überlebt hat, wie Y_i angibt. Wir versuchen dann eine Aussage über die Verteilungsfunktion F treffen zu können. Dazu gibt es die

sogenannte *Survival-Funktion*:

$$1 - F(x) = \mathbb{P}(X > x)$$

Diese beschreibt die Wahrscheinlichkeit nach dem Zeitpunkt x zu überleben [4, S.64-70].

3 Kaplan-Meier-Schätzer

Um Näheres über die Verteilungsfunktion F zu erfahren, müssen wir einen Schätzer finden, der diese möglichst gut beschreibt. Hier für gibt es unter anderem den sogenannten *Kaplan-Meier-Schätzer*, der die *Survival-Funktion* schätzen kann. Falls alle Daten vollständig beobachtet sind, liefert die Verteilungsfunktion für alle Z_i dieselbe Gewichtung, nämlich $\frac{1}{n}$. Damit sind die Sprünge in der Stufenfunktion gleich groß. Falls jedoch die Daten zum Teil zensiert sind, erhalten sie die Gewichtung durch den *Kaplan-Meier-Schätzer* für aufsteigend sortierte Z_i auf folgende Weise:

$$W_{ni} = \frac{\delta_{[i:n]}}{n-i+1} \prod_{j=1}^{i-1} \left(\frac{n-j}{n-j+1} \right)^{\delta_{[j:n]}}$$

Somit lässt sich die Verteilungsfunktion F_n wie folgt schätzen:

$$F_n(t) = \sum_{i=1}^{n} W_{ni} 1_{\{Z_{i:n} \leq t\}}$$

Wenn für ein $Z_{i:n}\delta_{[i:n]} = 0$ gilt, erhält das zugehörige $Z_{i:n}$ die Gewichtung 0. Die verlorene Gewichtung wird dann auf die nachfolgenden $Z_{i:n}$ verteilt. Um besser zu verstehen, wie die Masse verteilt wird, wird dies anhand von einem Beispiel im nachfolgendem Abschnitt gezeigt.

Wie bereits erwähnt, werden die $Z_{i:n}$ aufsteigend sortiert. Dann erhält jedes $Z_{i:n}$, wie im unzensiertem Fall, die Masse $\frac{1}{n}$. Nun wird das erste Z, also $Z_{1:n}$ mitsamt Zensierungsindikator beobachtet. Ist nun $\delta_{[1:n]} = 1$, dann ist $W_{n1} = \frac{1}{n}$. Sollte jedoch $\delta_{[1:n]} = 0$ so erhält W_{n1} den Wert 0. Die nun verlorene Masse würde indem Fall auf die restlichen $Z_{i:n}$ verteilt werden. Somit hätten dann $Z_{2:n}, \ldots, Z_{n:n}$ die Masse $\frac{1}{n} + \frac{1}{n} \cdot \frac{1}{n-1}$. Sollte nun der Fall eintreten, dass $Z_{n:n}$ den Zensierungsindikator $\delta_{[n:n]} = 0$ hat, so wäre an dieser Stelle die Verteilungsfunktion $F_n \neq 1$. Das auf diese Weise die Masse verloren geht, muss man für den Schätzer in kauf nehmen.
Würde man dies für die Daten aus Abbildung 2 anwenden, würde man diese zunächst aufsteigend sortieren.

	Z	delta
3	3.500000	0
5	4.100000	0
0	6.000000	0
1	6.000000	1
4	8.000000	1
2	8.500000	0

Tabelle 1: Aufsteigend sortierte $Z_{i:n}$
Quelle: Eigene Darstellung

Und daraus kann dann die Verteilungsfunktion F_n geschätzt werden:

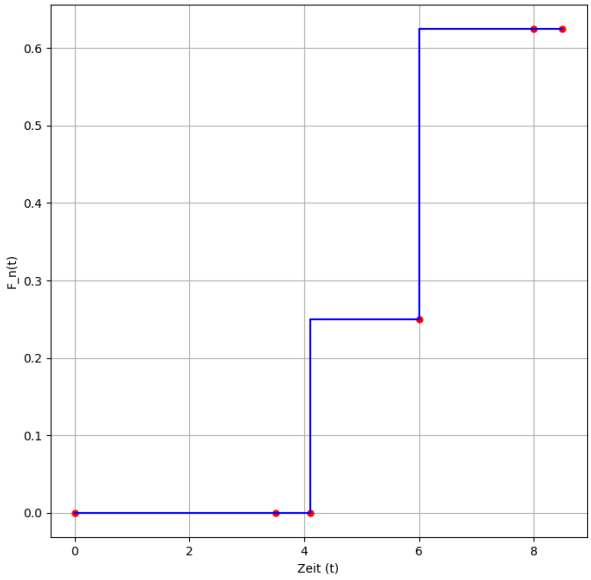

Abbildung 3: Verteilungsfunktion F_n nach Schätzung des Kaplan-Meier-Schätzers
Quelle: Eigene Darstellung

Hier kann man wunderbar sehen, dass Z_1, Z_2 und Z_3 den Wert 0 erhalten, da sie jeweils den Zensierungsindikator $\delta = 0$ haben. Somit wird die Masse auf das Z_4 um verteilt, wodurch dieses den Wert 0.25 erhält. Außerdem erhält in dem Fall das Z_6, also allgemein gesprochen das $Z_{n:n}$, den gleichen Wert, wie das Z_5, da es zensiert ist. Hier tritt also das oben genannte Problem auf, dass die Verteilungsfunktion F_n nicht auf 1 springt, sondern auf dem Wert des vorherigen, nicht-zensierten Z bleibt [4, S.92-104].

4 Neuronale Netze

Ein Neuronales Netz erzeugt für einen Input $x_1, \ldots, x_n$ einen Output $y_1, \ldots, y_n$. Für ein Netz mit einem Outputs und drei Inputs, wobei F das Netz von $\mathbb{R}^3$ nach $\mathbb{R}$ repräsentiert, kann man folgende Vorschrift formulieren:

$$y = F(x_{i1}, x_{i2}, x_{i3})$$

Hier meint i den i-ten Datenpunkt in dem Datensatz. Ein solches Netz könnte dann so aussehen:

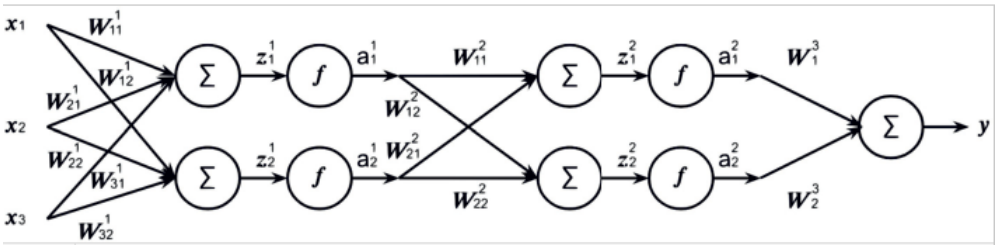

Abbildung 4: Beispiel eines Fully Connected Neural Networks
Quelle: Drori (2023:12)

Ein Neuronales Netz besteht unter anderem aus künstlichen Neuronen und Gewichten. Die Gewichte enthalten skalare Werte. Ein künstliches Neuron wertet den erhaltenen Input zu einem Output aus und gibt es an nachfolgende Neuronen weiter. Auf diese Weise ist das Neuronale Netz vernetzt und erzeugt mittels mathematischer Operationen einen Output. Hierbei wird jeweils das Skalarprodukt von Input und den sogenannten Gewichten erzeugt, abhängig davon, wie der Input mit den Folgeknoten verbunden ist. In der, in Abbildung 4 gegebenen, Grafik würde der Wert a_1^1 so entstehen [1, S.9-14]:

$$a_1^1 = f(x_1 \cdot W_{11}^1 + x_2 \cdot W_{21}^1 + x_3 \cdot W_{31}^1)$$

Dabei ist f in dem Kontext eine sogenannte Aktivierungsfunktion. Sie ist dafür verantwortlich, welche Ausgabe das Neuron erzeugt. Dafür gibt es zum Beispiel die *sigmoide Funktion*, welche den erhaltenen Input auf einen Wert zwischen 0 und 1 skaliert.
Das Netz ist in der Lage zu lernen und vergleicht dabei den Output y mit sogenannten *Targets*, die das richtige Ergebnis enthalten. Dieser Vergleich erfolgt über eine *Lossfunktion*, die den Fehler berechnet. Ein Beispiel für eine Lossfunktion könnte zum Beispiel der mittlere quadratische Fehler (*mean-squared-error*) sein:

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}(y_i^{target} - y_i^{predicted})^2$$

Mit dem *Backpropagation-Algorithmus* kann das Netz seine Gewichte anpassen, wodurch es im Optimalfall in der nächsten Iteration mit seiner Ausgabe richtig liegt. Das bedeutet, dass die Gewichte die Stellschrauben des Netzes sind, um sich zu verbessern [1, S.16-33] [2, S.138-143][3, S.313-316].

5 Gewichtete Lossfunktionen

Wie man die beiden Themen *zensierte Daten* und *Neuronale Netze* vereinen kann, wird in diesem Kapitel genauer erklärt. Die Lossfunktion die bereits im vorherigen Kapitel beschrieben wurde ist eine ungewichtete Lossfunktion. Das bedeutet, das alle Werte gleich bewertet werden. Wenn wir nun aber die berechneten Kaplan-Meier Gewichte mit in die Lossfunktion einbauen, erhalten wir eine *gewichtete Lossfunktion*, in diesem Fall den gewichteten *mean-squared-error*:

$$\mathcal{L}_{weighted} = \frac{1}{n}\sum_{i=1}^{n} W_{ni}(y_i^{target} - y_i^{predicted})^2$$

Auf diese Weise erhält jedes y_i die Gewichtung des Kaplan-Meier-Schätzers und dadurch kann das Neuronale Netz mit zensierten Daten trainiert werden.

6 Backpropagation mit gewichteter Lossfunktion

Über den Backpropagation-Algorithmus berechnet das Neuronale Netz die Gradienten des Outputs in Bezug auf den Input mittels der Kettenregel. Diese werden sukzessive rückwärts, beginnend mit dem letzten Layer, berechnet. Durch die Berechnung der Gradienten werden dann die Gewichte aktualisiert:

$$w^l = w^l - \alpha \frac{\partial \mathcal{L}}{\partial w^l}$$

Dabei ist l das l-te Layer, α die Lernrate (also wie stark die Aktualisierung der Gewichte erfolgen soll) und $\mathcal{L}$ der Loss bezüglich der Gewichte. Wie sich die Gradienten berechnen lassen, wird im folgenden Abschnitt anhand von einem Gewicht pro Layer des Netzes aus Abbildung 4 gezeigt [1, S.28-33] [2, S.138-143] [3, S.313-316]:

Dritter Layer:

$$\begin{aligned}
&\frac{\partial}{\partial w_1^3} \frac{1}{n} \sum_{i=1}^{n} W_{ni}(y_i^{target} - F(x_{i1}, x_{i2}, x_{i3}))^2 \\
&= \frac{1}{n} \sum_{i=1}^{n} W_{ni} \frac{\partial}{\partial w_1^3} (y_i^{target} - F(x_{i1}, x_{i2}, x_{i3}))^2 \\
&= -\frac{1}{n} \sum_{i=1}^{n} 2 \cdot W_{ni}(y_i^{target} - F(x_{i1}, x_{i2}, x_{i3})) \cdot \frac{\partial}{\partial w_1^3} \cdot (a_{i1}^2 \cdot w_1^3 + a_{i2}^2 \cdot w_2^3) \\
&= -\frac{1}{n} \sum_{i=1}^{n} 2 \cdot W_{ni}(y_i^{target} - F(x_{i1}, x_{i2}, x_{i3})) \cdot a_{i1}^2
\end{aligned}$$

Zweiter Layer:

$$\begin{aligned}
&\frac{\partial}{\partial w_{11}^2} \frac{1}{n} \sum_{i=1}^{n} W_{ni}(y_i^{target} - F(x_{i1}, x_{i2}, x_{i3}))^2 \\
&= -\frac{1}{n} \sum_{i=1}^{n} 2 \cdot W_{ni}(y_i^{target} - F(x_{i1}, x_{i2}, x_{i3})) \cdot \frac{\partial}{\partial w_{11}^2} \cdot (a_{i1}^2 \cdot w_1^3 + a_{i2}^2 \cdot w_2^3) \\
&= -\frac{1}{n} \sum_{i=1}^{n} 2 \cdot W_{ni}(y_i^{target} - F(x_{i1}, x_{i2}, x_{i3})) \cdot \frac{\partial}{\partial w_{11}^2} (f(z_{i1}^2) \cdot w_1^3 + f(z_{i2}^2) \cdot w_2^3) \\
&= -\frac{1}{n} \sum_{i=1}^{n} 2 \cdot W_{ni}(y_i^{target} - F(x_{i1}, x_{i2}, x_{i3})) \cdot \frac{\partial}{\partial w_{11}^2} (f(a_{i1}^1 \cdot w_{11}^2 + a_{i2}^1 \cdot w_{21}^2) \cdot w_1^3 + f(a_{i1}^1 \cdot w_{12}^2 + a_{i2}^1 \cdot w_{22}^2) \cdot w_2^3) \\
&= -\frac{1}{n} \sum_{i=1}^{n} 2 \cdot W_{ni}(y_i^{target} - F(x_{i1}, x_{i2}, x_{i3})) \cdot (f'(a_{i1}^1 \cdot w_{11}^2 + a_{i2}^1 \cdot w_{21}^2) \cdot w_1^3 \cdot a_{i1}^1)
\end{aligned}$$

Erster Layer:

$$
\begin{aligned}
&\frac{\partial}{\partial w_{11}^1}\frac{1}{n}\sum_{i=1}^{n} W_{ni}(y_i^{target} - F(x_{i1}, x_{i2}, x_{i3}))^2 \\
&= -\frac{1}{n}\sum_{i=1}^{n} 2 \cdot W_{ni}(y_i^{target} - F(x_{i1}, x_{i2}, x_{i3})) \cdot \frac{\partial}{\partial w_{11}^1}(a_{i1}^2 \cdot w_1^3 + a_{i2}^2 \cdot w_2^3) \\
&= -\frac{1}{n}\sum_{i=1}^{n} 2 \cdot W_{ni}(y_i^{target} - F(x_{i_1}, x_{i2}, x_{i3})) \cdot \frac{\partial}{\partial w_{11}^1}(f(z_{i1}^2) \cdot w_1^3 + f(z_{i2}^2) \cdot w_2^3) \\
&= -\frac{1}{n}\sum_{i=1}^{n} 2 \cdot W_{ni}(y_i^{target} - F(x_{i1}, x_{i2}, x_{i3})) \cdot \frac{\partial}{\partial w_{11}^1}(f(a_{i1}^1 \cdot w_{11}^2 + a_{i2}^1 \cdot w_{21}^2) \cdot w_1^3 \\
&+ f(a_{i1}^1 \cdot w_{12}^2 + a_{i2}^1 \cdot w_{22}^2) \cdot w_2^3) \\
&= -\frac{1}{n}\sum_{i=1}^{n} 2 \cdot W_{ni}(y_i^{target} - F(x_{i1}, x_{i2}, x_{i3})) \\
&\cdot \frac{\partial}{\partial w_{11}^1}(\{f[f(x_{i1} \cdot w_{11}^1 + x_{i2} \cdot w_{21}^1 + x_{i3} \cdot w_{31}^1) \cdot w_{11}^2 \\
&+ f(x_{i1} \cdot w_{12}^1 + x_{i2} \cdot w_{22}^1 + x_{i3} \cdot w_{32}^1) \cdot w_{21}^2]\} \cdot w_1^3 \\
&+ \{f[f(x_{i1} \cdot w_{11}^1 + x_{i2} \cdot w_{21}^1 + x_{i3} \cdot w_{31}^1) \cdot w_{12}^2 \\
&+ f(x_{i1} \cdot w_{12}^1 + x_{i2} \cdot w_{22}^1 + x_{i3} \cdot w_{32}^2) \cdot w_{22}^2]\} \cdot w_2^3) \\
&= -\frac{1}{n}\sum_{i=1}^{n} 2 \cdot W_{ni}(y_i^{target} - F(x_{i1}, x_{i2}, x_{i3})) \\
&\cdot f'\{f(x_{i1} \cdot w_{11}^1 + x_{i2} \cdot w_{21}^1 + x_{i3} \cdot w_{31}^1) \cdot w_{11}^2 \\
&+ f(x_{i1} \cdot w_{12}^1 + x_{i2} \cdot w_{22}^1 + x_{i3} \cdot w_{32}^1) \cdot w_{21}^1\} \cdot w_1^3 \\
&\cdot [f'(x_{i1} \cdot w_{11}^1 + x_{i2} \cdot w_{21}^1 + x_{i3} \cdot w_{31}^1) \cdot w_{11}^2 \\
&+ f'(x_{i1} \cdot w_{12}^1 + x_{i2} \cdot w_{22}^1 + x_{i3} \cdot w_{32}^1) \cdot w_{21}^1] \cdot (x_{i1}) \\
&+ f'\{f(x_{i1} \cdot w_{11}^1 + x_{i2} \cdot w_{21}^1 + x_{i3} \cdot w_{31}^1) \cdot w_{12}^2 \\
&+ f(x_{i1} \cdot w_{12}^1 + x_{i2} \cdot w_{22}^1 + x_{i3} \cdot w_{32}^1) \cdot w_{22}^2\} \cdot w_2^3 \\
&\cdot [f'(x_{i1} \cdot w_{11}^1 + x_{i2} \cdot w_{21}^1 + x_{i3} \cdot w_{31}^1) \cdot w_{12}^2 \\
&+ f'(x_{i1} \cdot w_{12}^1 + x_{i2} \cdot w_{22}^1 + x_{i3} \cdot w_{32}^1) \cdot w_{22}^2] \cdot (x_{i1})
\end{aligned}
$$

7 Anwendung an die Daten der UK Biobank

7.1 Datenlage

Die UK Biobank verfügt über Patientendaten mit einem Stichprobenumfang von 477.574. Über diese Patienten sind folgende *Features* bekannt:

- Alter,
- Geschlecht,
- Systolischer Blutdruck,
- Gesamt-Cholesterol,
- HDL-Cholesterol,
- LDL-Cholesterol,
- Raucher(Ja/Nein)
- Datum der Baseline-Untersuchung
- Datum des Herzinfarkts (falls vorhanden)
- Datum des Hirninfarkts (falls vorhanden)

Nun ist interessant zu wissen, wie lange ein Patient mit den gegebenen *Features* ohne Herz- oder Hirninfarkt überleben kann. Hierbei soll konkret die Anzahl der Jahre vorausgesagt werden, bis einer der genannten Fälle eintritt.

7.2 Präprozessierung der Daten

Vorab müssen die Daten in das richtige Format gebracht und bereinigt werden, bevor diese verwendet werden können. Falls beispielsweise ein Patient sowohl Herz- als auch Hirninfarkt erlitten hat, so wird das Datum des zuerst eintretenden Infarkts gewählt. Außerdem, sollen nur Patienten beobachtet werden, deren Infarkt nach der Baseline-Untersuchung stattgefunden haben.

7.3 Berechnung der Zensierungsindikatoren δ

Um zu wissen, welche Patienten zensiert sind, werden vorher die Zensierungsindikatoren berechnet und als zusätzliche Spalte an das *DataFrame* Objekt angehängt. Ein *DataFrame* ist eine Art Tabelle der Python Library *pandas*, mit der es schnell und einfach möglich ist, Operationen und Berechnung durchzuführen. Hierbei sind konkret alle Patienten, welche ein kardiovasukläres Ereignis hatten (also Herz- oder Hirntumor) vollständig beobachtet. Andersherum sind alle Patienten, die kein kardiovaskuläres Ereignis während der Studienzeit hatten, zensiert. Diese erhalten auf diese Weise ihren jeweiligen Zensierungsindikator.

	Age	Sex	SBP	Cholesterol	HDL	LDL	SmokingStatus	$\text{diff}_{in_{days}}$	delta
0	65	F	187.000000	6.515000	1.546000	4.115000	0.000000	16.772603	0
1	64	M	138.000000	6.633000	1.239000	4.336000	1.000000	6.041096	1
2	45	F	105.000000	5.034000	2.292000	2.655000	0.000000	16.772603	0
3	57	M	126.000000	5.678000	1.125000	3.912000	1.000000	16.772603	0
4	66	F	115.000000	5.095000	2.296000	2.359000	1.000000	16.772603	0

Tabelle 2: Ausschnitt der Daten nach der Präprozessierung
Quelle: Eigene Darstellung, Datengrundlage: UK Biobank (2023)

Notiz: Die Spalte *diff_in_days* enthält die Zeitdifferenzen in Jahre, nicht in Tage, da dies nachträglich geändert wurde.
Die Berechnung der Zensierungsindikatoren in Python:

```
cardiovascular_data["delta"] = 10
for i in range(len(cardiovascular_data)):
        if(pd.isnull(cardiovascular_data.loc[i, "DateOfCardiovascularEvent"])):
                cardiovascular_data.loc[i, "delta"] = 0
        else:
                cardiovascular_data.loc[i, "delta"] = 1
```

7.4 Simulation verschiedener Zensierungsraten

Da die Daten stark zensiert sind ist es natürlich interessant herauszufinden, welchen Einfluss die Zensierungsraten auf die Leistungsfähigkeit des Netzes haben. Deshalb wurden abgesehen von den Basisdaten folgende Zensierungsraten ausprobiert, indem das Zensierungsverhältnis künstlich und zufällig ausgeglichen wird, sodass die besagte Zensierungsrate zutrifft. Die Erstellung des jeweiligen Datensatzes mit gegebener Zeniserungsrate kann so erreicht werden:

1. Zähle die Anzahl der beiden Zensierungsindikatoren in den Daten
2. Berechne die benötigte Anzahl an Zensierungsindikatoren für die größere Klasse, indem die Anzahl der kleineren Klasse mit der gewünschten Zensierungsrate multipliziert wird ($N = count(\delta_{lower}) \cdot Zensierungsrate$)
3. Ziehe eine zufällige Stichprobe aus der größeren Klasse mit der Stichprobe N aus Punkt 2
4. Füge die Stichprobe mit allen Daten der kleineren Klasse zusammen

Die Zensierungsrate der Basisdaten beträgt 93,8%. Dann wurden die Daten der Zensierungsraten 10%, 25%, 50% und 75% ausprobiert, indem diese alle auf dem selben Netz jeweils 50 mal trainiert wurden. Das Ergebnis dieser Untersuchung folgt in einem späteren Kapitel.

In Python kann man dies wiefolgt umsetzen:

```
def censor_distribution(data, distribution, RS):
        lower_class = data[data["delta"] == 1]
        upper_class = data[data["delta"] == 0]
        lower_count = len(lower_class)
        N = int(lower_count * distribution)

        class_balance = upper_class.sample(n=N, random_state=RS)

        new_data = pd.concat([lower_class, class_balance])
        return new_data
```

7.5 One-Hot Encoding und Normalisierung der Daten

Ein Neuronales Netz kann keine kategorischen *Features* verarbeiten, daher ist es notwendig diese umzukodieren. Jedoch wäre es nicht korrekt, den Daten eine künstliche Rangordnung zu verschaffen, indem wir beispielsweise nominale Daten einen numerischen Wert 0 und 1 zuordnen. Aus diesem Grund können wir diese mittels One-Hot Encoding umkodieren. Diese Methode teilt das jeweilige Feature in so viele Features auf, wie es Kategorien innerhalb des Features gibt. So existiert in dem Datensatz das nominale Feature *Sex*, welches die Kategorie M für männlich und F für weiblich besitzt. Dementsprechend entstehen dann die zwei neuen Features *Sex_M* und *Sex_F*, welche durch das One-Hot Encoding binär gemacht werden. Nun ist das Feature nicht mehr binär und kann dem Netz übergeben werden.
Das Normalisieren der Daten kann außerdem helfen, die Konvergenz des Netzes zu fördern und zu vermeiden, das die Gradienten zu groß werden und deshalb die Gewichtsänderung während des Trainings zu stark ist. Hierbei wird konkret ein sogenannter *MinMaxScaler* von *scikit-learn* verwendet, welcher die Daten auf wie folgt skaliert:

$$X_{scaled} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

Der *MinMaxScaler* nimmt sich also den Datenpunkt X, welchen er skalieren will und den jeweils größten (X_{max}) und kleinsten (X_{min}) Datenpunkt innerhalb des Features und skaliert ihn auf den Wertebreich 0 bis 1.

	Age	SBP	Cholesterol	HDL	LDL	SmokingStatus	$\text{diff}_{in_{days}}$	delta	Sex_F	Sex_M
0	0.771429	0.606796	0.356891	0.316168	0.403214	0.000000	16.772603	0	1.000000	0.000000
1	0.742857	0.368932	0.365375	0.242635	0.426426	1.000000	6.041096	1	0.000000	1.000000
2	0.200000	0.208738	0.250413	0.494850	0.249869	0.000000	16.772603	0	1.000000	0.000000
3	0.542857	0.310680	0.296714	0.215329	0.381893	1.000000	16.772603	0	0.000000	1.000000
4	0.800000	0.257282	0.254799	0.495808	0.218780	1.000000	16.772603	0	1.000000	0.000000

Tabelle 3: Ausschnitt der Daten nach dem One-Hot Encoding und Normalisierung
Quelle: Eigene Darstellung, Datengrundlage: UK Biobank (2023)

Das One-Hot-Encoding und die Normalisierung in Python:

```
def do_onehot_and_scale(df):
        # OneHot Encoding fuer das Geschlechtsfeature
        dummy_df = pd.get_dummies(df["Sex"], prefix="Sex", dtype="float")
        df = pd.concat([df, dummy_df], axis=1)
        df.drop(["Sex"], axis=1, inplace=True)

        # Normalisierung
        scaler = MinMaxScaler()
        df[["Age", "SBP", "Cholesterol", "HDL", "LDL"]]
        = scaler.fit_transform(df[["Age", "SBP", "Cholesterol", "HDL", "LDL"]])
        return df
```

7.6 Aufteilung in Trainings- und Testdaten

Es ist wichtig, die Daten in Trainings- und Testdaten aufzuteilen, um feststellen zu können, ob das Netz in der Lage ist zu generalisieren. Sollte beispielsweise der *Loss* bezüglich der Trainingsdaten deutlich geringer sein, als der der Testdaten, so spricht man von *Overfitting*, also einer Überanpassung des Neuronalen Netzes an die Trainingsdaten. Um dies verhindern zu können, gibt es einige Regularisierungsmethoden, wie zum Beispiel dem *Weight Decay* oder *Dropout* auf die aber nicht weiter eingegangen werden.
Von den gesamten Umfang der Daten, sind in der Regel 70-90% die Trainingsdaten und die restlichen 10-30% die Testdaten. Es ist zu dem in unserem Fall wichtig, die Daten zuerst aufzuteilen, bevor die Kaplan-Meier Gewichtsberechnung durchgeführt wird, da Trainings- und Testdaten die Gewichte getrennt voneinander berechnet bekommen sollen, um eine richtige Schätzung zu ermöglichen. Hier war es notwendig eine eigene Funktion für das Aufteilen zu schreiben, da die häufig benutzte *train_test_split* Methode aus der Python Library *scikit-learn* die Daten direkt in Input- und Targetdaten aufteilt, was hier noch nicht gewünscht ist.

```
def split_data(df, test_size=0.2, seed=1):
        shuffled = df.loc[np.random.RandomState(seed=seed)
                                    .permutation(df.index)]
                                    .reset_index(drop=True)
        test_df_size = int(len(shuffled) * test_size)
        train_df_size = int(len(shuffled) - test_df_size)
        df1 = shuffled.iloc[:test_df_size,:]
        df2 = shuffled.iloc[test_df_size:train_df_size+test_df_size,:]
        return df1, df2
```

7.7 Gewichtsberechnung

In diesem Abschnitt geht es um die konkrete Kaplan-Meier Gewichtsberechnung in Python. Hier kommt konkret die Python Library *lifelines* zum Einsatz, welche den Kaplan-Meier-Schätzer bereits implementiert. Dieser berechnet jedoch nur die konkrete Wahrscheinlichkeit nach einem Zeitpunkt t zu überleben. Da die KM-Gewichte die Sprunghöhen der Verteilungsfunktion F sind, kann man diese dann berechnen:

$$W_{ni} = F_n(Z_{i:n}) - F_n(Z_{i-1:n})$$

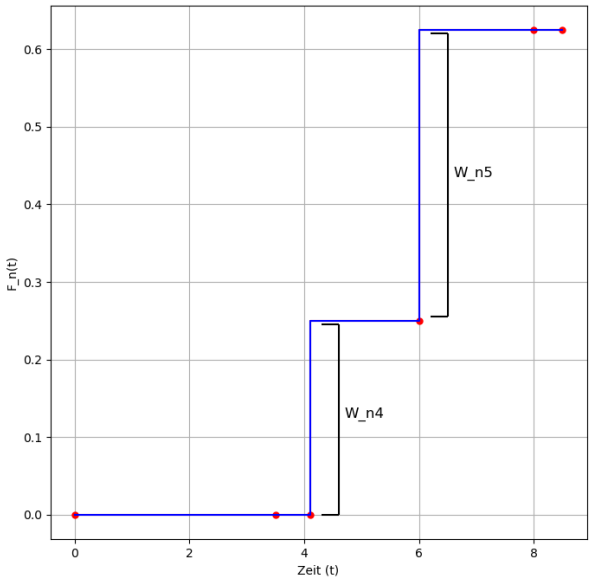

Abbildung 5: Sprunghöhen aus dem Beispiel aus Kapitel 3
Quelle: Eigene Darstellung

Die Umsetzung in Python sieht dann folgendermaßen aus:

```
def weight_calc(df):
        df = df.sort_values(by="diff_in_days", ascending=True)
                .reset_index(drop=True)

        # Weight Berechnung ueber die Surival function des
        # KaplanMeierFitters von lifelines
        kmf = KaplanMeierFitter()
        kmf.fit(durations=df["diff_in_days"],
                        event_observed=df["delta"],
                        timeline=df["diff_in_days"].unique())

        # Berechnung der timeline fuer die Gewichte
        unique_diffs = pd.Series(df["diff_in_days"].unique())
```

```
    unique_diffs = pd.DataFrame(unique_diffs, columns=["timeline"])

    f_n = (kmf.survival_function_ - 1) * (-1)
    merged_df = pd.merge(f_n,
                         unique_diffs,
                         left_index=True,
                         right_on=["timeline"],
                         how="inner")

    # Gewichtsberechnung
    merged_df["weights"] = 0
    for i in range(len(merged_df)-1):
    merged_df.loc[i, "weights"] =
    merged_df.loc[i+1, "KM_estimate"] - merged_df.loc[i, "KM_estimate"]

    # Uebertragung in das richtige DataFrame
    df = pd.merge(df,
                  merged_df,
                  left_on="diff_in_days",
                  right_on="timeline")
            .drop(labels = ["KM_estimate","timeline"], axis=1)

    return df
```

7.8 Aufteilung in Input-, Target- und Gewichtsdaten

Nun erfolgt die besagte Aufteilung in Input-, Target- und Gewichtsdaten, da dies erst nach der Gewichtsberechnung geht. Hier wird nun auch die Spalte mit den Zensierungsindikatoren δ entfernt, da diese für das Training nicht benötigt wird. Als Inputdaten haben wir nun die Features *Age, SBP, Cholesterol, HDL, LDL, Sex_F, Sex_M* und *SmokingStatus* unser Target ist dabei die Spalte *diff_in_days*, welche die beobachtete Lebensdauern beinhaltet.

```
def split_input_target(df):
    df.drop("delta", inplace=True, axis=1)
    X = df[["Age", "SBP", "Cholesterol", "HDL", "LDL", "Sex_F",
            "Sex_M", "SmokingStatus"]]
    y = tf.constant(df["diff_in_days"], dtype="float32")
    weights = tf.constant(df["weights"], dtype="float32")
    return X, y, weights
```

7.9 Erstellung eines ersten Neuronalen Netzes

Nun kann ein erstes Neuronales Netz erstellt werden. Hierbei kommen die Python Libraries *Tensorflow* und *Keras* zum Einsatz. Dafür wird einfach das *Sequentialmodell* aus Tensorflow geladen, wobei man dann jeden Layer einzeln hinzufügen kann und dabei unteranderem die Anzahl an Neuronen, die Aktivierungsfunktion und den Datentyp übergeben kann. Mittels dem *compile*-Attribut

kann dann das Neuronale Netz erstellt werden. Hier kann der gewünschte Optimierungsalgorithmus, die Lossfunktion und wenn gewünscht, eine oder mehrere Metriken übergeben werden, um das Netz abgesehen von dem Loss bewerten zu können, hinzugefügt werden. Schlussendlich wird mit dem *fit*-Attribut das Training des Neuronalen Netzes gestartet, wobei hier die Trainingsinputdaten, Trainingstargetdaten, die Anzahl der Epochen, die Größe der Batchsize und in unserem Falle, auch die KM-Gewichte übergeben werden können. Die Epochen ist die Anzahl, wie oft der gesamte Trainingsdatensatz trainiert wird und die Batchsize, wie viele Datenpunkte trainiert werden soll, bis eine Aktualisierung der Gewichte geschehen soll.

```
model = tf.keras.models.Sequential([
        tf.keras.layers.Dense(8, input_shape=(8,), activation='relu',
                              dtype="float32"),
        tf.keras.layers.Dense(128, activation="relu", dtype="float32"),
        tf.keras.layers.Dense(128, activation="relu", dtype="float32"),
        tf.keras.layers.Dense(1, activation='linear', dtype="float32")
])
model.compile(optimizer='adam', loss="mean_squared_error",
              weighted_metrics=[])
model.fit(x=X_train, y=y_train, epochs=5, batch_size=1,
          sample_weight=weights_train)
```

Hier konnte nach nur 5 Epochen ein Trainingsloss von $8,4032e^{-6}$ erreicht werden.

7.10 Hyperparametersuche

Ein Neuronales Netz hat einige Stellschrauben, daher ist es sinnvoll, die Werte dieser Stellschrauben, den sogenannten *Hyperparametern*, zu variieren. So kann eventuell eine bessere Leistungsfähigkeit des Netzes erzielt werden, da es besser während des Trainings konvergiert. Hierbei kann zum Beispiel die Anzahl der Neuronen, die Anzahl der Layer oder die jeweilige Aktivierungsfunktion verändert werden. Auf diese Weise kann man einen Suchraum angeben, der bestimmt, welche Hyperparameter welche Werte austestet. Dafür gibt es verschiedene Methoden um den gegebenen Suchraum zu explorieren. Im Folgenden wurde *GridSearch* verwendet, welche, sofern es nicht zu viele sind, alle Hyperparameterkombinationen ausprobiert. Außerdem wurde *RandomSearch* verwendet, was zufällige Hyperparameterkombinationen aus dem Suchraum auswählt.
In der Praxis wurde dies in Python mittels sogenannten *Keras Tunern* erzielt, wobei man hier einfach ein Neuronales Netz erstellen kann, aber dabei verschiedene Optionen, wie das Netz aufgebaut sein soll, offen lässt und die Wahl der Hyperparametern dem Tuner übergibt. Diesem Tuner kann man dann ein *Objective*, also ein Ziel übergeben, was es erreichen soll. In unserem konkreten Fall ist es sinnvoll, den Trainingsloss minimieren zu wollen. Wenn man dann das Training startet, so kann man dem Tuner außerdem sogenannte *Callbacks* übergeben. Diese können helfen, die Trainingszeit zu optimieren, da das Training unter bestimmten Bedingungen abgebrochen werden kann und mit einer anderen Hyperparameterkombination fortgesetzt wird. In diesem konkreten Fall wird das Training abgebrochen, wenn der Trainingsloss sich nicht mehr signifikant verbessert.

Suchraum der Hyperparametersuche

- num_layers [1;5] *Anzahl der Layer*
- units [32;512; step=32] *Anzahl der Neuronen pro Layer*
- activations [relu, sigmoid, leaky_relu, swish] *Aktivierungsfunktionen*
- use_dropout [1;0] *Benutzung von Dropout Ja/Nein*
- dropout_rate [0;0.5; step=0.1] *Dropoutwahrscheinlichkeit, falls use_dropout = 1*
- optimizer [adam, sgd, rmsprop, adagrad] *Optimierungsverfahren*
- learning_rate [$1e^{-4}$;$1e^{-1}$] *Lernrate*

Ergebnis der Hyperparametersuche
Das *GridSearch* Modell konnte einen Trainingsloss von $8.315e^{-6}$ erreichen mit einem Testloss von $9.685e^{-5}$. Hierbei ist wichtig zu erwähnen, dass nicht der gesamte Suchraum exploriert werden konnte, da die Anzahl an Hyperparameterkombinationen zu groß war. Hingegen konnte das *RandomSearch* Modell einen Trainingsloss von $8.289e^{-6}$ mit einem Testloss von $1.034e^{-4}$ erreichen.
Das beste Netz des *GridSearch* Tuners hatte dann folgende Hyperparameter:

- num_layers=3 (1 Input-, 1 Hidden- und 1 Outputlayer)
- units=[9, 32, 1]
- activation=relu
- use_dropout=False
- dropout_rate=None
- optimizer=adam
- learning_rate=0.0028

Das Netz ist also für die Menge an Daten relativ klein und erzielt dennoch ein gutes Ergebnis.

```
def create_base_model(hp):

        model = Sequential()

        model.add(Dense(8, input_dim=8, activation=tf.nn.relu,
                kernel_initializer=
                tf.keras.initializers.he_uniform(seed=42)))
        num_layers = hp.Int("num_layers", min_value=1, max_value=5,
                            step=1)

        # Layer, activation, units und Dropoutchoice
        for i in range(num_layers):
        model.add(keras.layers.Dense(units=hp.Int(f'units_{i}',
                                     min_value=32, max_value=512,
                                     step=32),
```

```
        activation=hp.Choice("activation",
                             ["relu", "sigmoid", "leaky_relu", "swish"])))

        use_dropout = hp.Boolean(f'use_dropout_{i}', default=False)
        if use_dropout:
        # Dropout-Layer mit Dropout-Rate als Hyperparameter
        dropout_rate = hp.Float(f'dropout_rate_{i}', min_value=0.0,
                                max_value=0.5, step=0.1, default=0.2)
        model.add(keras.layers.Dropout(rate=dropout_rate))

        # Optimizer und Learningrate Choice
        optimizer_choice = hp.Choice('optimizer',
                               values=['adam', 'sgd', 'rmsprop', 'adagrad'])

        if optimizer_choice == 'adam':
        optimizer = keras.optimizers.Adam(learning_rate=
        hp.Float('learning_rate_adam', min_value=1e-4, max_value=1e-1,
        sampling='LOG', default=1e-3))

        elif optimizer_choice == 'sgd':
        optimizer = keras.optimizers.SGD(learning_rate=
        hp.Float('learning_rate_sgd', min_value=1e-4, max_value=1e-1,
         sampling='LOG', default=1e-2))

        elif optimizer_choice == "adagrad":
        optimizer = keras.optimizers.Adagrad(learning_rate=
        hp.Float('learning_rate_adagrad', min_value=1e-4, max_value=1e-1,
         sampling='LOG', default=1e-2))

        else:
        optimizer = keras.optimizers.RMSprop(learning_rate=
        hp.Float('learning_rate_rmsprop', min_value=1e-4, max_value=1e-1,
         sampling='LOG', default=1e-3))


        model.add(keras.layers.Dense(units=1, activation='linear'))

        model.compile(loss="mean_squared_error", optimizer = optimizer,
                      weighted_metrics=[])
        return model

tuner = GridSearch(
        create_base_model,
        objective=Objective("loss", direction="min"),
        max_trials=100,
        directory="/nn_survival_analysis/",
        project_name='my_tuning_project')
```

```
tuner.search(x=X_train,
        y=y_train,
        epochs=10,
        validation_data=(X_test, y_test),
        batch_size=16,
        callbacks=[stop_early, stop_early_nan],
        sample_weight=weights_train)
```

7.11 Leistungsfähigkeit nach Zensierungsrate

Wie bereits in Abschnitt **7.4** erwähnt, werden verschiedene Zensierungraten ausprobiert, um den Einfluss auf die Leistungsfähigkeit zu beobachten. Dafür wird jeder Datensatz mit fester Zensierungsrate auf dem gleichen Netz 50 mal trainiert, um die Robustheit dieser Beobachtung zu gewährleisten. Wichtig zu erwähnen ist auch, dass die Generierung der Daten mit fester Zensierungsrate in jedem der jeweiligen 50 Datensätzen pro Zensierungsrate zufällig ist. Das heißt, dass nicht immer die gleichen Daten gewählt werden, um das Netz zu trainieren.

```
def test_censor_performance(df, censorrate, train_weights, test_weights):
        loss_df = pd.DataFrame(columns=["train_loss", "test_loss"])
        pack = 0
        packs = 50
        model = tf.keras.models.Sequential([
        tf.keras.layers.Dense(8, input_shape=(8,), activation='relu',
                              dtype="float32"),
        tf.keras.layers.Dense(128, activation="relu", dtype="float32"),
        tf.keras.layers.Dense(128, activation="relu", dtype="float32"),
        tf.keras.layers.Dense(1, activation='linear', dtype="float32")
        ])

        model.compile(optimizer='adam', loss="mean_squared_error",
                      weighted_metrics=[])
        while pack < packs:
        data = df.copy()
        data = censor_distribution(data, censorrate, pack)
        data = do_onehot_and_scale(data)
        df1, df2 = split_data(data, seed=pack)
        df1 = weight_calc(df1)
        df2 = weight_calc(df2)
        train_x, train_y, train_weights = split_input_target(df1)
        test_x, test_y, test_weights = split_input_target(df2)
        history = model.fit(train_x, train_y, epochs=5, batch_size=8,
                            verbose=0, sample_weight=train_weights)
        test_loss = model.evaluate(test_x, test_y, batch_size=8, verbose=0,
                                   sample_weight=test_weights)
        loss_df.loc[pack] = (history.history["loss"][-1], test_loss)
        pack+=1
        print("Pack ", pack , "/" , packs)
        clear_output(wait=True)
```

```
    return loss_df
```

Daraus ergibt sich folgender Boxplot, wobei jeweils der Trainings- und Testloss für die jeweilige Zensierungsrate abgebildet ist.

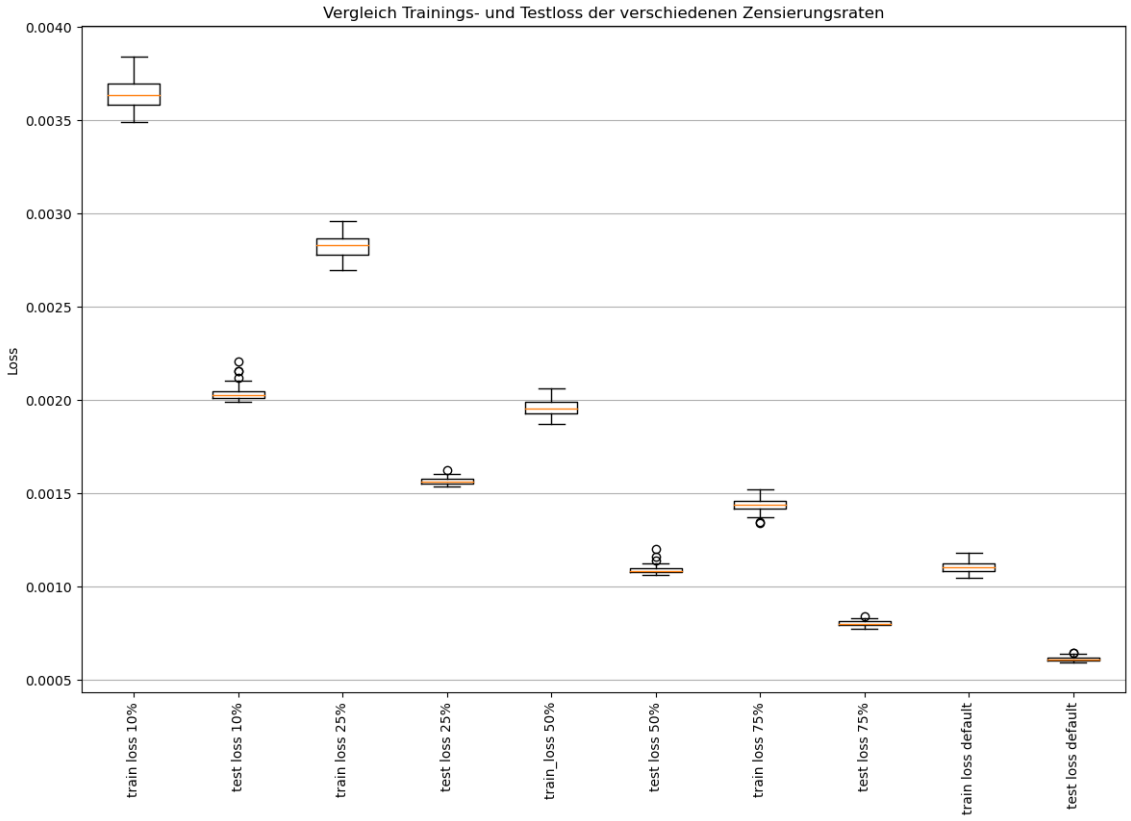

Abbildung 6: Leistungsfähigkeit des Netzes nach Zensierungsraten
Quelle: Eigene Darstellung; Datengrundlage: UK Biobank (2023)

Hieraus geht überraschenderweise hervor, dass sowohl Trainings- als auch Testloss besser wird, umso **höher** die Zensierungsrate wird. Zusätzlich kann man beobachten, dass der Testloss immer besser, als der Trainingsloss ist. Dies könnte eventuell damit zusammen hängen, dass viele zensierte Patienten die gleiche beobachtete Zeitdauer haben und das Netz sich eher diesen Zeitdauern anpasst, da diese im Vergleich zu den unzensierten Zeitdauern enorm überwiegen.

8 Fazit

9 Literatur- und Quellenverzeichnis

Literatur

[1] Drori, I. (2023). *The Science of Deep Learning.* Cambridge: Cambridge University Press.

[2] Deisenroth, M., Faisal, A., Ong, C. (2020). *Mathematics for machine learning.*, Cambridge: Cambridge University Press.

[3] Ertel, W. (2021). *Grundkurs Künstliche Intelligenz: Eine praxisorientierte Einführung.* 5.Aufl., Wiesbaden: Springer Fachmedien Wiesbaden

[4] Klein J. P., Moeschberger, M. L. (2005) *Survival Analysis: Techniques for Censored and Truncated Data.*, New York: Springer.