INSTITUT FÜR INFORMATIK
Machine Learning

Universitätsstr. 1 D–40225 Düsseldorf

Heinrich Heine
Universität
Düsseldorf

Graph-based semi-supervised learning: Welchen Vorteil bringt die Verteilung der nicht gegebenen Label?

Mick Potzkai

Bachelorarbeit

Beginn der Arbeit:	26. Juni 2020
Abgabe der Arbeit:	28. September 2020
Gutachter:	Prof. Dr. Stefan Harmeling
	Prof. Dr. Gunnar W. Klau

Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Düsseldorf, den 28. September 2020

Mick Potzkai

Zusammenfassung

In dieser Arbeit behandele ich Semi-Supervised Learning Probleme auf Graphen. Insbesondere gehe ich auf einen Algorithmus (*GNM*) von Zhou et al., 2019 ein, der als erster in dieser Art die Verteilung der unbekannten Label mit einbezieht. Dabei stelle ich diesen Algorithmus zunächst ausführlich vor und zeige, wie die einzelnen Schritte motiviert sind. Zudem biete ich eine Toolbox mit einer Implementierung des Algorithmus, welche kurz vorgestellt wird.

Der Algorithmus ist in zwei Schritte aufgeteilt. Im ersten Schritt wird die Verteilung der Knoten mit bekanntem Label approximiert, woraufhin im zweiten Schritt die Werte der unbekannten Label geschätzt werden. Jeder Schritt zieht dabei die Schätzung des anderen mit ein. Durch Wiederholung dieser zwei Schritte wird sich eine stets besser werdende Approximation der Label erhofft.

Im Anschluss untersuche ich die einzelnen Schritte auf die Qualität ihrer Schätzung. Es stellt sich hierbei heraus, dass die Schätzung der Verteilung der unbekannten Label zwei Qualitätsklassen hat und sich zudem mit besser werdenden Schätzungen der unbekannten Label nicht signifikant verbessert.

Dadurch motiviert schlage ich einen weiteren Algorithmus (*GNMn*) vor, von dem erwartet wird, dass er die potenziellen Schwachstellen von *GNM* verbessert.

In einer abschließenden experimentellen Analyse wird *GNM* und *GNMn* verglichen mit einem Basismodel *SM*, welches die Verteilung der Knoten mit bekanntem Label nicht beachtet. Sollte die Verteilung der Knoten mit bekanntem Label stark von einer Gleichverteilung abweichen, so zeigt sich *GNM* effizient. *GNMn* beweist sich als ein guter Kompromiss zwischen *GNM* und *SM*.

Inhaltsverzeichnis

1 Einführung

Der Bereich des maschinellen Lernens wird mit der Zeit immer mächtiger und findet damit auch immer mehr praktische Anwendung. Eines der Probleme, die mithilfe von maschinellem Lernen gelöst werden können, ist Semi-Supervised Classification und Regression auf Graphen.

Gegeben sei ein Graph, in dem zu jedem Knoten ein Label existiert. Es ist aber nur eine kleine Teilmenge der Label bekannt. Die Aufgabe des Programms ist es, alle unbekannten Label zu ermitteln und möglichst präzise anzugeben. Für diese Aufgabe existieren bereits validierte Methoden wie Graph Attention Networks von Veličković et al., 2017 oder Graph Convolutional Networks von Kipf und Welling, 2016.

Aufgrund der Art, wie die Daten erhoben werden, kann es aber zu einem nicht zu ignorierenden Fehlen von Daten kommen (kurz: MNAR, engl.: „Missing not at Random“). Dies bedeutet, dass die Knoten mit bekanntem Label nicht gleichverteilt in der Menge der Knoten liegen, sondern einer anderen Verteilung unterliegen. Diese Verteilung kann sowohl von den wahren Label als auch von den Features eines Knotens abhängen. Dazu zum Verständnis:

Beispiel 1. *Man stelle sich einen Graphen vor, in dem Knoten Kreuzungen und Kanten Straßen darstellen. Die Label der Knoten stehen für das an dieser Kreuzung existierende Verkehrsaufkommen. Es sollen die unbekannten Label ermittelt werden. Es ist aber so, dass das Verkehrsaufkommen vermehrt dort gemessen wird, wo viel Verkehr herrscht. Deshalb hängt die Verteilung der MNAR von den wahren Label ab.*

Beispiel 2. *In einem zweiten Graphen stellen Knoten Accounts auf Facebook und Kanten die Freundschaften dieser Accounts dar. Zu jedem Knoten existieren Features wie das Alter und der Beruf der Person. Das Ziel ist es zu ermitteln, welche politische Partei von einer Person bei einer kommenden Wahl gewählt wird. Eine Reihe von Personen geben bereits in ihrem Profil an, welcher Partei sie zugehören. Daraus wird entnommen, dass sie diese Partei auch wählen werden. Ob eine Person nun aber angibt, ob und welcher Partei sie angehört, hängt sowohl von ihrer Partei, als auch von ihrem Beruf und ihrem Alter ab. Ist eine Person z.B. Politiker, so ist die Wahrscheinlichkeit sehr hoch, dass sie dies preisgibt. Ist sie jedoch Journalist für eine neutrale Zeitung, so scheint es unwahrscheinlicher. Daher hängt die Verteilung der MNAR in diesem Beispiel von den Features und den Labels der Knoten ab.*

Liegt eine MNAR vor, so ist es für eine gute Schätzung der unbekannten Label essentiell, diese ebenfalls unbekannte Verteilung zu ermitteln und bei der Schätzung mit einzubeziehen. Das erste Paper, welches den MNAR Mechanismus nutzt, ist „Graph-based Semi-Supervised Learning with Nonignorable Missingness“ von Zhou et al., 2019. In diesem Paper wird ein Algorithmus vorgeschlagen, welcher die Verteilung der bekannten Label mit einbezieht und so nach eigenen Angaben State-of-the-Art Ergebnisse erzielt. Dieser Algorithmus nutzt einen Ansatz, in dem abwechselnd die Parameter von zwei Funktionen verbessert werden. Die erste parametrisierte Funktion schätzt die unbekannten Label, während die zweite parametrisierte Funktion die Verteilung der bekannten Label schätzt. In das Training dieser Funktionen wird die aktuelle Schätzung der jeweils anderen Funktion mit einbezogen. Auf diese Weise wird die unbekannte Verteilung der bekannten Label mit in das Training einbezogen.

In der hier präsentierten Arbeit wird dieses Paper von Zhou et al., 2019 ausführlich erklärt und im Anschluss der Algorithmus in Kapitel 6 auf experimentelle Art und Weise auf seine Leistungsfähigkeit untersucht.

2 Modell

In diesem Abschnitt führe ich kurz die genutzte Notation ein und stelle anschließend das im Algorithmus genutzte Model vor.

Gegeben sei ein Graph $G = (V, E, A)$. Hierbei sei $V = (v_1, ..., v_N)$ die Knotenmenge, E die Kantenmenge und $A \in \mathbb{R}^{N \times N}$ die Adjazenzmatrix. Es kann sich bei G sowohl um einen gerichteten als auch um einen ungerichteten Graphen handeln. Handelt es sich bei G um einen ungerichteten Graphen, so ist A symmetrisch. Sollte es keine Kantengewichte geben, so ist $A \in \{0, 1\}^{N \times N}$ mit $A_{ij} = \begin{cases} 1 & (v_i, v_j) \in E \\ 0 & \text{sonst} \end{cases}$. Anderenfalls beschreibt A_{ij} das Kantengewicht der Kante von v_i nach v_j.

Zu jedem Knoten v_i sei $x_i \in \mathbb{R}^p$ ein Feature Vektor. Alle Feature Vektoren werden zusammengefasst zu $X = [x_1, ..., x_N] \in \mathbb{R}^{N \times p}$. y_i sei das Label zu v_i. Bezeichne mit $Y = (y_1, ..., y_N)$ den Vektor der Labels. Es kann entweder gelten $Y \in \mathbb{R}^N$ oder $Y \in \mathbb{K}^N$ mit $\mathbb{K} = \{1, 2, ..., K\}$. Im ersten Fall handelt es sich bei Y um eine kontinuierliche Zufallsvariable, im zweiten Fall um eine diskrete Zufallsvariable mit K Klassen. Es sei der Großteil der Labels unbekannt. $r_i \in \{0, 1\}$ gibt an, ob y_i gegeben ist. In diesem Fall gilt $r_i = 1$. Ist y_i unbekannt, so gilt $r_i = 0$.

Weiter seien zwei parametrisierte Funktionen $h(x_i; \theta_h)$ und $\mathcal{G}^A(X; \theta_g)$ gegeben, welche mithilfe des Gradientenabstiegsverfahren optimiert werden können. Für beliebige $s, q \in \mathbb{N}$ sei

$$\mathcal{G}^A(\cdot\,; \theta_g) : \mathbb{R}^{N \times p} \to \mathbb{R}^{N \times q}$$

und

$$h(\cdot\,; \theta_h) : \mathbb{R}^p \to \mathbb{R}^s.$$

In dieser Darstellung stehen θ_g und θ_h für die Parameter der Funktionen.

Die Outputs von $\mathcal{G}^A$ bzw. h haben keine direkte Interpretation. Sie sind Zwischenergebnisse bei der Berechnung der Verteilung von Y bzw. der Verteilung $P(r_i = 1 | y_i, h(x_i))$.

$\mathcal{G}^A$ stellt die Grundlage zur Approximation von Y dar. Bei dieser Funktion kann es sich zum Beispiel um GCNs (Kipf und Welling, 2016) oder GATs (Veličković et al., 2017) handeln.

h stellt die Grundlage zur Approximation der r_i dar. Dabei kann es sich ebenfalls um ein neuronales Netz handeln. In Kapitel 6 wird h beispielsweise als ein mehrlagiges Perzeptron gewählt.

Es wird angenommen, dass r_i einer Bernoulli-Verteilung folgt, sollte das wahre Label y_i und $h(x_i; \theta_h)$ bekannt sein. Also:

$$r_i | y_i, h(x_i; \theta_h) \sim \text{Bernoulli}(\pi_i)$$

2.1 Modell im Fall: Klassifizierung

Handelt es sich bei einem Problem um ein Klassifizierungsproblem, so gibt es zwei Möglichkeiten ein Label darzustellen. Gehört ein Knoten v der Klasse $k \in \{1, ..., K\}$ an, dann schreibe ich

- one-hot-kodiert: $y \in \{0,1\}^K$ mit $\forall j \in \{1, ..., K\}: \; y_j = \begin{cases} 1 & , j = k \\ 0 & , j \neq k \end{cases}$
- bzw. als Klassenindex: $y = k$.

Die Verteilung $P(r_i | y_i, h(x_i); \theta_h)$ modelliere ich durch das Anhängen einer weiteren linearen Schicht und der Sigmoid-Aktivierungsfunktion an $h(\cdot\,; \theta_h)$. Die neu entstehende Funktion in x_i und y_i wird mit $\pi(\cdot\,, h(\cdot\,; \theta_h))$ bezeichnet. Für $i \in \{1, ..., N\}$, ein $x_i \in X$ und ein beliebiges one-hot-kodiertes Label y_i (nicht notwendigerweise das wahre Label) ist dann:

$$\begin{aligned} \pi(y_i, h(x_i; \theta_h)) &= \pi(y_i, h(x_i; \theta_h); \alpha_r, \gamma, \phi) = P(r_i = 1 | y_i, h(x_i; \theta_h); \alpha_r, \gamma, \phi) \\ &= \frac{\exp(\alpha_r + \gamma^T h(x_i; \theta_h) + \phi^T y_i)}{1 + \exp(\alpha_r + \gamma^T h(x_i; \theta_h) + \phi^T y_i)} \end{aligned} \tag{1}$$

Hierbei ist $\alpha_r \in \mathbb{R}$ ein Skalar. $\gamma \in \mathbb{R}^s$ und $\phi \in \mathbb{R}^K$ sind Vektoren. Zusammengefasst schreibe ich die Parameter als $\theta_r = (\alpha_r, \gamma, \phi, \theta_h)$.

Es ist essentiell, dass y_i one-hot-kodiert ist, da das Modell anderenfalls auf natürliche Art und Weise Klassen mit höherem Index höhere Wahrscheinlichkeiten zuordnen würde.

Das primäre Ziel ist, die unbekannten y_i zu bestimmen. Dazu wird folgendes Modell genutzt:

$$P(y_i = k | \mathcal{G}^A(X; \theta_g)_i; \alpha_k, \beta_k) = \frac{\exp(\alpha_k + \beta_k^T \mathcal{G}^A(X; \theta_g)_i)}{\sum_{j=1}^K \exp(\alpha_j + \beta_j^T \mathcal{G}^A(X; \theta_g)_i)} \tag{2}$$

Hier wird der Output von $\mathcal{G}^A(X; \theta_g)$ als Input einer linearen Schicht betrachtet und abschließend eine Soft-Max-Aktivierungsfunktion angewandt, um eine Wahrscheinlichkeitsverteilung zu erhalten. In diesem Modell sind $\alpha \in \mathbb{R}^K$, $\beta = [\beta_1, ..., \beta_K] \in \mathbb{R}^{q \times K}$ und θ_g die trainierbaren Parameter.

Ich fasse die Parameter zusammen zu $\theta_y = (\alpha, \beta, \theta_g)$.

Sind die Parameter trainiert, lassen sich die Vorhersagen bestimmen, indem für jedes $i \in \{1, ..., N\}$ die Klasse $\hat{y}_i \in \{1, ..., K\}$ gewählt wird, sodass $P(y_i = \hat{y}_i | \mathcal{G}^A(X; \theta_g)_i; \alpha_k, \beta_k)$ maximal ist.

2.2 Modell im Fall: Regression

Die Verteilung $P(r_i|y_i, h(x_i); \theta_h)$ wird im kontinuierlichem Fall analog zum diskreten Fall (1) modelliert. Für $i \in \{1, ..., N\}$, $x_i \in X$ und ein beliebiges Label $y_i \in \mathbb{R}$ gilt:

$$\begin{aligned} \pi(y_i, h(x_i; \theta_h)) &= \pi(y_i, h(x_i; \theta_h); \alpha_r, \gamma, \phi) = P(r_i = 1|y_i, h(x_i; \theta_h); \alpha_r, \gamma, \phi) \\ &= \frac{\exp(\alpha_r + \gamma^T h(x_i; \theta_h) + \phi y_i)}{1 + \exp(\alpha_r + \gamma^T h(x_i; \theta_h) + \phi y_i)} \end{aligned} \tag{3}$$

Der einzige Unterschied zu (1) ist, dass $\phi \in \mathbb{R}$ kein Vektor ist und y nicht one-hot-kodiert ist.

Das folgende Modell schätzt die unbekannten y_i:

$$Y = \alpha 1_N + \mathcal{G}^A(X; \theta_g)\beta + \varepsilon \tag{4}$$

Dabei ist $\beta \in \mathbb{R}^q$ ein Vektor, $\alpha \in \mathbb{R}$ ein Skalar, $1_N = (1, ..., 1)^T \in \mathbb{R}^N$ der 1-Vektor und $\varepsilon \sim N(0, \sigma^2 I)$ mit $\varepsilon \in \mathbb{R}^N$ eine Störung. In diesem Modell sind α, β und θ_g trainierbare Parameter. Ich fasse die Parameter wieder zusammen zu $\theta_r = (\alpha_r, \gamma, \phi, \theta_h)$ und $\theta_y = (\alpha, \beta, \theta_g)$.

Sobald die Parameter θ_y trainiert sind, lassen sich die Vorhersagen für die unbekannten y_i als Output des Modells direkt ablesen.

Für Rechnungen in Kapitel 3 ist es sowohl im kontinuierlichen als auch im diskreten Fall nötig, dass $\mathcal{G}^A(X; \theta_g)$ und $h(x_i; \theta_h)$ so gewählt sind, dass folgende Bedingungen für $i \neq j$ erfüllt sind:

$$\begin{aligned} &1. \quad y_i \perp y_j \mid \mathcal{G}^A(X; \theta_g) \\ &2. \quad r_i \perp r_j \mid h(x_i; \theta_h),\ y_i \end{aligned} \tag{5}$$

In Worten bedeutet dies, dass die Label zweier Knoten unabhängig sein müssen, sollte $\mathcal{G}^A(X; \theta_g)$ bekannt sein. Sollte außerdem für ein $i \in \{1, ..., N\}$ $h(x_i; \theta_h)$ und y_i bekannt sein, so müssen für alle $j \in \{1, ..., i-1, i+1, ..., N\}$ r_i und r_j unabhängig sein.

3 Annahmen zur Identifizierbarkeit

Die Funktionen $h(x_i; \theta_h)$ und $\mathcal{G}^A(X; \theta_g)$ dürfen nicht beliebig gewählt werden. Damit der Algorithmus funktioniert, muss sichergestellt sein, dass Änderungen in den Parametern tatsächlich Änderungen in den Outputs hervorrufen.

Betrachte dazu zunächst die gemeinsame Verteilung von $y_{\text{obs}} = \{y_i|r_i = 1\}$ und $R = (r_1, ..., r_N)$. O.B.d.A. sei $r_1, ..., r_n = 1$ und $r_{n+1}, ..., r_N = 0$.

$$f(y_{\text{obs}}, R|X, \theta_r, \theta_y) = f(y_1, ..., y_n, r_1, ..., r_N|X, \theta_r, \theta_y)$$
$$\overset{(5)}{=} \prod_{i=1}^{n} f(y_i, r_i|X, \theta_r, \theta_y) \prod_{i=n+1}^{N} \int f(y_i, r_i|X, \theta_r, \theta_y) dy_i$$

Mit $f(y_i, r_i|X, \theta_r, \theta_y) = P(r_i|y_i, h(x_i); \theta_r) f(y_i|\mathcal{G}^A(X)_i; \theta_y)$ erhalten wir

$$= \prod_i \left(P(r_i = 1|y_i, h(x_i); \theta_r) f(y_i|\mathcal{G}^A(X)_i; \theta_y) \right)^{r_i} \left(1 - \int P(r_i = 1|y, h(x_i); \theta_r) f(y|\mathcal{G}^A(X)_i; \theta_y) dy \right)^{1-r_i} \tag{6}$$

Das allgemeine Ziel mit $f(y_{\text{obs}}, R|X, \theta_r, \theta_y)$ ist es die Parameter θ_r und θ_y so zu wählen, dass f als Funktion in θ_r und θ_y maximiert wird.

Definition 3 (Identifizierbarkeit der Parameter)**.** *Es sei ein Modell P_θ gegeben, welches von Parametern $\theta \in \Theta$ abhängt. Das Model P heißt identifizierbar auf dem Parameterraum Θ, falls für $\theta_1, \theta_2 \in \Theta$ gilt:*

$$P_{\theta_1} = P_{\theta_2} \Rightarrow \theta_1 = \theta_2 \tag{7}$$

In unserem Fall ist diese Definition äquivalent dazu, dass aus

$$\left(\forall X, y, i : P(r_i|y, h(x_i); \theta_r) f(y|\mathcal{G}^A(X)_i; \theta_y) = P(r_i|y, h(x_i); \theta'_r) f(y|\mathcal{G}^A(X)_i; \theta'_y) \right)$$

folgt, dass $(\theta_y, \theta_r) = (\theta'_y, \theta'_r)$. Ist das Modell identifizierbar, so ist das globale Maximum eindeutig.

Jedoch gilt die Identifizierbarkeit auf dem Parameterraum für viele neuronale Netze nicht. Dazu ein Beispiel:

Beispiel 4 (Identifizierbarkeit schlägt fehl)**.** *Notwendig für die Identifizierbarkeit des gesamten Modells ist die Identifizierbarkeit der Parameter von (1). Wähle*

$$h(x_i; \beta_r) := Relu(x_i \odot \beta_r) \tag{8}$$

wobei $\odot$ die elementweise Multiplikation von zwei Vektoren derselben Länge darstellt. Damit erhalten wir

$$P(r_i = 1|y_i, h(x_i; \beta_r); \alpha_r, \gamma, \phi) \overset{(1)}{=} \frac{\exp(\alpha_r + \gamma^T Relu(x_i \odot \beta_r) + \phi^T y)}{1 + \exp(\alpha_r + \gamma^T Relu(x_i \odot \beta_r) + \phi^T y)}$$
$$= \frac{\exp(\alpha_r + \frac{1}{2}\gamma^T Relu(x_i \odot 2\beta_r) + \phi^T y)}{1 + \exp(\alpha_r + \frac{1}{2}\gamma^T Relu(x_i \odot 2\beta_r) + \phi^T y)}$$
$$\overset{(1)}{=} P(r_i = 1|y_i, h(x_i; 2\beta_r); \alpha_r, \frac{1}{2}\gamma, \phi)$$

Damit haben wir zwei verschiedene Mengen von Parametern θ_r und θ'_r mit $P(r_i = 1|y_i, h(x_i); \theta_r) = P(r_i = 1|y_i, h(x_i); \theta'_r)$. Somit erfüllt das Modell mit dem gewählten h nicht die Identifizierbarkeit auf dem Parameterraum.

Zu unserem Glück ist die Identifizierbarkeit auf dem Parameterraum eine Eigenschaft, die für uns nicht notwendig ist. Angenommen wir haben unser Modell P passend gewählt und $\theta_0 \in \Theta$ ist die wahre Parametermenge. Dann genügt es uns, wenn wir eine Parametermenge $\theta_1 \in \Theta$ finden, sodass $P_{\theta_1} = P_{\theta_0}$. Das motiviert zu folgender Definition:

Definition 5 (PEQ Raum)**.** *Gegeben seien $\theta_y = (\alpha, \beta, \theta_g)$, $\theta_r = (\alpha_r, \gamma, \phi, \theta_h)$, $\theta'_y = (\alpha', \beta', \theta'_g)$ und $\theta'_r = (\alpha'_r, \gamma', \phi', \theta'_h)$.*

Wir sagen (θ_y, θ_r) und (θ'_y, θ'_r) sind äquivalent, in Zeichen $(\theta_y, \theta_r) \sim (\theta'_y, \theta'_r)$, falls folgende Bedingungen wahr sind:

1. $\alpha_r = \alpha'_r$
2. $\alpha = \alpha'$
3. $\phi = \phi'$
4. $\forall x \in X : \gamma^T h(x; \theta_h) = \gamma'^T h(x; \theta'_h)$
5. $\mathcal{G}^A(X; \theta_g)\beta = \mathcal{G}^A(X; \theta'_g)\beta'$

Bezeichne die Klasse aller Tupel von Parametermengen (θ_y, θ_r) mit $\mathcal{D}(\theta_y) \times \mathcal{D}(\theta_r)$. Mit der Äquivalenzrelation $\sim$ auf $\mathcal{D}(\theta_y) \times \mathcal{D}(\theta_r)$ können wir nun den Quotientenraum $\mathcal{D}(\theta_y) \times \mathcal{D}(\theta_r)/\sim$ bilden. Diesen nennen wir ***Prediction Equivalent Quotient*** *(PEQ) Raum.*

Auf diesem Raum wollen wir nun die Identifizierbarkeit definieren

Definition 6 (Identifizierbarkeit auf dem PEQ Raum)**.** *Ein GNM Modell heißt identifizierbar auf dem PEQ Raum, falls:*

$$\Big(\forall X, y, i : f(y|\mathcal{G}^A(X)_i; \theta_y) P(r = 1|y, h(x_i); \theta_r) = f(y|\mathcal{G}^A(X)_i; \theta'_y) P(r = 1|y, h(x_i); \theta'_r)\Big)$$
$$\Rightarrow (\theta_y, \theta_r) \sim (\theta'_y, \theta'_r)$$

Identifizierbarkeit auf dem PEQ Raum folgt offensichtlich aus der Identifizierbarkeit auf dem Parameterraum. Die Rückrichtung gilt aber, wie an Beispiel 4 zu sehen ist, nicht. Während Identifizierbarkeit der Parameter wichtig ist, wenn man an den wahren Parametern selbst interessiert ist, ist Identifizierbarkeit auf dem PEQ-Raum praktisch, wenn man bloß an der wahren Vorhersage interessiert ist. Nach Definition 5 der Äquivalenz von Parametermengen ist es klar, dass wenn zwei Parametermengen in der selben Äquivalenzklasse sind, ihre Vorhersage auch konstant bleibt. Außerdem ist es nach Definition der Identifizierbarkeit auf dem PEQ-Raum gegeben, dass wenn diese gilt, aus der Gleichheit der Vorhersage, die Äquivalenz der Parametermengen folgt.

Es halte nun für ein Modell die Identifizierbarkeit auf dem PEQ-Raum. Ändert man an diesem Modell die Parameter, aber die Vorhersage bleibt gleich, so ist klar, dass man die

Äquivalenzklasse nicht verlassen hat. Also kann man nur θ_h und γ in einer Form geändert haben, dass Punkt 4. aus Definition 5 hält, oder β und θ_g geändert haben, sodass Punkt 5. aus Definition 5 hält oder beides.

Nun gibt es aber auch Fälle, in denen die Identifizierbarkeit auf dem PEQ-Raum nicht hält. Also gibt es Modelle, sodass zwei Parametermengen exisitieren, die zwar die gleiche Vorhersage machen, aber nicht äquivalent sind.

Folgender Satz bietet zwei leichte Möglichkeiten, die Identifizierbarkeit auf dem PEQ-Raum zu zeigen:

Satz 7. *Ein diskretes GNM Modell ist identifizierbar auf dem PEQ-Raum falls:*

(A1) Für alle θ_g existieren X_1 und X_2, sodass für alle i gilt $\mathcal{G}^A(X_1;\theta_g)_i \neq \mathcal{G}^A(X_2;\theta_g)_i$. Außerdem gilt $\beta \neq 0$.

Zu einem kontinuierlichen GNM Modell sei $X = [Z, U]$, sodass $f(y_i|\mathcal{G}^A(X)_i)$ von U abhängt, während $P(r_i = 1|y_i, h(x_i))$ nicht von U abhängt. Dann ist das Modell unter folgenden zwei Kriterien identifizierbar auf dem PEQ-Raum:

(A2) Für alle θ_g und Z existieren U_1 und U_2, sodass für alle i gilt $\mathcal{G}^A([Z,U_1];\theta_g)_i \neq \mathcal{G}^A([Z,U_2];\theta_g)_i$. Außerdem gilt $\beta \neq 0$.

(A3) Für alle θ_h gibt es z_1 und z_2, sodass $h(z_1;\theta_h) \neq h(z_2;\theta_h)$ gilt. Außerdem gilt $\gamma \neq 0$.

4 Loss Funktionen

Da die Wahl von $h(x_i;\theta_h)$ und $\mathcal{G}^A(X;\theta_g)$ weitestgehend beliebig ist, kann nicht garantiert werden, dass die gemeinsame Verteilung von y_{obs} und R in (6) direkt maximiert werden kann.

Um diese gemeinsame Verteilung möglichst gut zu schätzen, wird ein doppelt robuster Schätzer vorgeschlagen. Ein doppelt robuster Schätzer (engl.: *doubly robust estimator*, Bang und J. M. Robins, 2005) zeichnet sich dadurch aus, dass dieser konsistent ist, sollte eine der beiden unbekannten Verteilungen korrekt spezifiziert sein.

In unserem Fall handhaben wir dies, indem wir unsere beiden Modelle abwechselnd verbessern und in diese Verbesserung die Schätzung des jeweils anderen Modells mit einfließen lassen. Wir verbessern ein Modell, indem wir es mit einer Loss Funktion bewerten und anschließend mit dem Gradientenabstiegsverfahren (GD) trainieren. Als erstes schauen wir uns die Loss Funktion zum Training von θ_y an.

4.1 Inverse Probability Weighted Estimator für θ_y

Gegeben sei eine Schätzung $\hat{\theta}_r$ vom wahren θ_r. Mit dieser Schätzung lässt sich $\hat{\pi}$ von $\pi = (\pi_1, ..., \pi_N)^T$ mit $\pi_i = P(r_i = 1|y_i, h(x_i);\theta_r)$ schätzen.

Diese Schätzung soll genutzt werden, um θ_y zu verbessern. Im Fall von Klassifizierung ist in dieser Situation eine gut geeignete Loss Funktion der Cross Entropy Loss.

$$\mathscr{L}(\theta_y) = -\sum_{i:r_i=1} \log\left(P(y_i|\mathscr{G}^A(X)_i;\theta_y)\right) \tag{9}$$

$P(\cdot\,|\mathscr{G}^A(X)_i;\theta_y)$ ist die Wahrscheinlichkeitsverteilung über die verschiedenen Klassen $k \in \{1,...,K\}$.

Diese Loss Funktion verbessern wir, motiviert durch den Inverse Probability Weighted Estimator von J. Robins et al., 1994 bzw. den verwandten Horvitz-Thompson Schätzer (Horvitz und Thompson, 1952), indem wir jeden Knoten mit seiner invertierten Wahrscheinlichkeit, gegeben zu sein, gewichten.

Damit ergibt sich:

$$\mathscr{L}_1(\theta_y|\hat{\theta}_r) = -\sum_{i:r_i=1} \frac{1}{\hat{\pi}_i} \log\left(P(y_i|\mathscr{G}^A(X)_i;\theta_y)\right) \tag{10}$$

Bei dieser Formel ist zu beachten, dass die Abhängigkeit $\hat{\pi}_i = \hat{\pi}_i(\hat{\theta}_r)$ gegeben ist.

Befinden wir uns im Fall von einer kontinuierlichen Zufallsvariable Y, wählen wir anstelle des Cross Entropy Loss den Mean Squared Error Loss und erhalten insgesamt:

$$\mathscr{L}_1(\theta_y|\hat{\theta}_r) = -\sum_{i:r_i=1} \frac{1}{\hat{\pi}_i} \left(y_i - \alpha + \mathscr{G}^A(X)_i\beta\right)^2 \tag{11}$$

Knoten, die aufgrund von ihrem Feature und ihres Labels, selten gesehen werden, werden in (10) und (11) stark gewichtet und Knoten, die oft gesehen werden, werden schwach gewichtet. Das ist eine sinnvolle Gewichtung, da so alle Knoten entsprechend der Häufigkeit ihres Vorkommens gleich gewichtet sind.

$\hat{\theta}_y := \arg\min_{\theta_y} L_1(\theta_y|\theta_r)$ ist im kontinuierlichen und im diskreten Fall ein konsistenter Schätzer, wenn θ_r korrekt gewählt ist.

4.2 Schätzer für θ_r

Gegeben sei eine Schätzung $\hat{\theta}_y$ von der wahren Parametermenge θ_y.

Definiere $r := (r_1,...,r_N)$. Die Stichprobe r von der Verteilung $P(r_i = 1|y_i, h(x_i))$ ist die einzige Information, die wir zur Schätzung von π haben.

Um die optimale Schätzung $\hat{\pi}_{opt}$ zu erhalten, müssen wir für eine Zufallsvariable $X \sim \text{Ber}(\hat{\pi}_{opt})$ die Wahrscheinlichkeit, dass $X = r$ gilt maximieren. Das ergibt $\hat{\pi}_{opt} = \arg\max_\pi P(X = r|\pi)$.

Das motiviert zu folgendem Schätzer. Für eine Schätzung $\hat{\pi} = (\hat{\pi}_1,...,\hat{\pi}_N)$ mit $\hat{\pi}_i = \pi(y_i, h(x_i);\theta_r)$ sei $X \sim \text{Ber}(\hat{\pi})$. Dann definiere einen Schätzer wie folgt:

$$
\begin{aligned}
\hat{\theta}_r &= \arg\max_{\theta_r} P(X = r|\theta_r) \\
&\overset{(5)}{=} \arg\max_{\theta_r} \prod_{i=0}^{N} P(X_i = r_i|\theta_r) \\
&= \arg\max_{\theta_r} \prod_{i:r_i=1} P(X_i = 1|\theta_r) \prod_{i:r_i=0} P(X_i = 0|\theta_r) \\
&= \arg\max_{\theta_r} \prod_{i:r_i=1} \hat{\pi}_i \prod_{i:r_i=0} (1 - \hat{\pi}_i) \\
&= \arg\max_{\theta_r} \log\left(\prod_{i:r_i=1} \hat{\pi}_i \prod_{i:r_i=0} (1 - \hat{\pi}_i) \right) \\
&= \arg\max_{\theta_r} \sum_{i:r_i=1} \log\left(\hat{\pi}_i\right) + \sum_{i:r_i=0} \log\left(1 - \hat{\pi}_i\right)
\end{aligned}
$$

Für $r_i = 0$ ist das wahre y_i unbekannt. Integriere dieses daher raus.

$$
= \arg\max_{\theta_r} \sum_{i:r_i=1} \log\left(\hat{\pi}_i\right) + \sum_{i:r_i=0} \log\left(1 - \mathbf{E}_{y_i}\left[\pi(y_i, h(x_i); \theta_r)\right]\right)
$$

Damit ergibt sich folgende Loss Funktion:

$$
\mathscr{L}_2(\theta_r|\hat{\theta}_y) = -\sum_{i:r_i=1} \log(\pi(y_i, h(x_i); \theta_r)) - \sum_{i:r_i=0} \log\left(1 - \mathbf{E}_{y_i}\left[\pi(y_i, h(x_i); \theta_r)\right]\right) \tag{12}
$$

Im Fall von Klassifizierung ist die Verteilung von Y aus (2) bekannt. Daher kann der Erwartungswert im rechten Teil der Gleichung direkt berechnet werden:

$$
\mathbf{E}_{y_i}\left[\pi(y_i, h(x_i); \theta_r)\right] = \sum_{k=1}^{K} P(y_i = k|\mathscr{G}^A(X); \hat{\theta}_y)\, \pi(k, h(x_i); \theta_r) \tag{13}
$$

Außerdem besteht die Möglichkeit Stichproben $\{y_{ib}\}_{b=1}^{B} \sim P(y_i|\mathscr{G}^A(X)_i; \hat{\theta}_y)$ zu nehmen und mit diesen den Erwartungswert zu approximieren:

$$
\mathbf{E}_{y_i}\left[\pi(y_i, h(x_i); \theta_r)\right] \approx \frac{1}{B} \sum_{b=1}^{B} \pi(y_{ib}, h(x_i); \theta_r) \tag{14}
$$

Diese Methode wird von Zhou et al., 2019 genutzt.

Bei einer hohen Zahl an Klassen, kann die Berechnung von (13) lange dauern. In diesem Fall ist (14) eine interessante Alternative, da eine hohe Geschwindigkeit von Loss Funktionen wünschenswert ist.

Im Fall von einer kontinuierlichen Zufallsvariable Y haben wir die Dichtefunktion der Verteilung nicht gegeben und können den Erwartungswert daher nicht direkt ausrechnen

bzw. können auch keine Stichproben aus der Verteilung nehmen. Daher wählen wir $B = 1$ in (14) und nehmen als einzige Wert $y_{i1} = \mathbf{E}[y_i] = \alpha + \mathcal{G}^A(X)_i\beta$. Damit haben wir:

$$\mathbf{E}_{y_i}\left[\pi(y_i, h(x_i); \theta_r)\right] \approx \pi(\alpha + \mathcal{G}^A(X)_i\beta, h(x_i); \theta_r) \tag{15}$$

Ist θ_y korrekt gewählt, dann ist $\hat{\theta}_r := \arg\min_{\theta_r} L_2(\theta_r|\theta_y)$ ein kontinuierlicher Schätzer.

5 Algorithmus

In diesem Kapitel stelle ich den Algorithmus vor, der von Zhou et al., 2019 vorgeschlagen wird. Die Idee des Algorithmus ist, dass abwechselnd θ_y und θ_r bzw. $\hat{Y}$ und $\hat{\pi}$ verbessert werden. Das wird getan, indem die zugehörigen Loss-Funktionen $\mathcal{L}_1(\theta_y|\hat{\theta}_r)$ bzw. $\mathcal{L}_2(\theta_r|\hat{\theta}_y)$ minimiert werden und mit dem Gradientenabstiegsverfahren (GD) die Parameter θ_y bzw. θ_r trainiert werden. Da in $\mathcal{L}_1(\theta_y|\hat{\theta}_r)$ die bisherige Schätzung $\hat{\theta}_r$ von θ_r eingeht, bzw. in $\mathcal{L}_2(\theta_r|\hat{\theta}_y)$ die bisherige Schätzung $\hat{\theta}_y$ von θ_y eingeht, ist zu erwarten, dass die Schätzungen mit fortlaufenden Epochen immer besser werden.

In den folgenden Schritten sind die Details des Algorithmus zu sehen:

1. Wähle initiale Werte für $\pi_i^{(0)}$. Es kann z.B. $\forall i: \pi_i^{(0)} = 1$ gewählt werden. Das ist eine sinnvolle Wahl, da sich dann $\mathcal{L}_1(\theta_y|\hat{\theta}_r)$ im diskreten Fall zum Negativ-Log-Likelihood-Loss bzw. im kontinuierlichen Fall zum Mean-Squared-Error-Loss vereinfacht. Also in beiden Fällen sinnvolle Loss-Funktionen. Wähle außerdem $\theta_y^{(0)}$ und $\theta_r^{(0)}$.

2. Setze $e = 1$. e gibt die Epoche an, in der man sich befindet.

3. Mit $\pi^{(e-1)}$ ist $\mathcal{L}_1(\theta_y|\hat{\theta}_r^{(e-1)})$ bestimmt. Setze $\theta_y^{(e,0)} = \theta_y^{(e-1)}$. Verbessere $\theta_y^{(e,0)}$ für $M^{(e)}$ viele Iterationen mit GD. In Iteration i haben wir:

   $$\theta_y^{(e,i+1)} \quad \leftarrow \quad \theta_y^{(e,i)} - \gamma_0 \nabla_{\theta_y} \mathcal{L}_1(\theta_y|\hat{\theta}_r^{(e-1)})$$

   γ_0 ist dabei die Lernrate. Schreibe $\theta_y^{(e)} := \theta_y^{(e,M^{(e)})}$ nach der letzten Iteration.

4. Bestimme $\mathcal{L}_2(\theta_r|\hat{\theta}_y)$. Bestimme dazu im kontinuierlichen Fall für alle unbekannten Labels die Approximation durch $y_i^{(e)} := \alpha^{(e)} + \mathcal{G}^A(X; \theta_g^{(e)})_i\beta^{(e)}$, wobei $\theta_y^{(e)} = (\alpha^{(e)}, \beta^{(e)}, \theta_g^{(e)})$.

   Wähle im diskreten Fall für alle unbekannten Labels B viele Sample aus der Verteilung $P(y_i|\mathcal{G}^A(X)_i; \theta_y^{(e)})$.

5. Setze $\theta_r^{(e,0)} = \theta_r^{(e-1)}$. Verbessere $\theta_r^{(e,0)}$ für $N^{(e)}$ viele Iterationen mit GD. In Iteration j haben wir:

   $$\theta_r^{(e,j+1)} \quad \leftarrow \quad \theta_r^{(e,j)} - \gamma_1 \nabla_{\theta_r} \mathcal{L}_2(\theta_r|\hat{\theta}_y^{(e-1)})$$

   γ_1 ist die Lernrate. Schreibe $\theta_r^{(e)} := \theta_r^{(e,N^{(e)})}$ nach der letzten Iteration.

6. Aktualisiere $\pi^{(e)}$ mithilfe der im letzten Schritt erhaltenen Verteilung $P(r_i = 1|y_i, h(x_i); \theta_r^{(e)})$.
7. Stoppe, falls Konvergenzkriterium erfüllt ist oder die maximale Zahl an Epochen erreicht ist. Gehe anderenfalls zu Schritt 3.

Das Konvergenzkriterium ist das folgende:

$$\left(\sum_{i\colon r_i=0} |y_i^{(e)} - y_i^{(e-1)}| \Big/ \sum_i 1_{r_i=0} \right) \leq \varepsilon \tag{16}$$

In Worten bedeutet (16), dass im kontinuierlichen Fall gestoppt wird, falls die Summe der Änderungen in der letzten Epoche nicht mehr signifikant groß ist. Im diskreten Fall wird gestoppt, falls die Anzahl der Labels, die sich in der letzten Epoche geändert haben, ausreichend klein ist.

5.1 Implementierung

Den Algorithmus habe ich mit Python implementiert. Als Basis habe ich die Pytorch Geometric Bibliothek von Fey und Lenssen, 2019 genutzt. Der gesamte Code dazu liegt im Anhang vor.

In diesem Abschnitt werde ich als wesentlichste Teile der Implementierung die Implementierung der Loss Funktionen vorstellen.

Die erste Loss-Funktion, die wir betrachten, ist $\mathscr{L}_1$:

$$\mathscr{L}_1(\theta_y|\hat{\theta}_r) = -\sum_{i\colon r_i=1} \frac{1}{\hat{\pi}_i} \log\left(P(y_i|\mathscr{G}^A(X)_i; \theta_y)\right) \tag{10 Wdhg}$$

In der Implementierung unterscheiden wir zusätzlich zur eigentlichen Definition zwei verschiedene Arten der Reduktion. Diese sind „Mean“ und „Sum“. Im Fall von „Sum“ entspricht die Implementierung gerade (10). Im Fall von „Mean“ wird im Anschluss noch mit $1/\#\{r_i=1\}$ multipliziert, um die Abhängigkeit des Losses von der Anzahl der Knoten zu entfernen. In der Anwendung stellte sich dies als überlegen heraus.

Die Implementierung beruht auf dem Cross Entropy Loss aus dem Paket *torch.nn.functional*. Es wird angenommen, dass die Maske, auf der die Loss-Funktion ausgewertet werden soll, bereits vor dem Aufruf auf pi, out und target angewandt wurde. Diese drei Parameter müssen also den gleichen *shape* haben.

```
1 def inverse_weighted_categorial_crossentropy_loss(pi, reduction='sum'):
2     length = pi.shape[0]
3
4     def loss_mean(out, target):
5         tmp = torch.nn.functional.cross_entropy(out, target, reduction='none')
6         return torch.mean((1./pi).view((length)) * tmp.view((length)))
7
8
9     def loss_sum(out, target):
```

```
        tmp = torch.nn.functional.cross_entropy(out, target, reduction='none')
        return torch.sum((1/pi).view((length)) * tmp.view((length)))

    if reduction == 'mean':
        return loss_mean
    else:
        return loss_sum
```

Sehr ähnlich zur Implementierung von Gleichung (10) ist die Implementierung von $\mathscr{L}_1$ im kontinuierlichen Fall.

$$\mathscr{L}_1(\theta_y|\hat{\theta}_r) = -\sum_{i:r_i=1} \frac{1}{\hat{\pi}_i} \left(y_i - \alpha + \mathscr{G}^A(X)_i\beta\right)^2 \tag{11 Wdhg}$$

Es ist zu beachten, dass angenommen wird, dass out $= \alpha + \mathscr{G}^A(X)_i\beta$ gilt.

```
def inverse_weighted_mean_squared_error(weight, reduction='sum'):
    length = weight.shape[0]

    def loss_sum(out, target):
        return torch.sum((1/weight).view(length) * (out.view(length) -
    ↳ target.view(length)) ** 2, dim=0)

    def loss_mean(out, target):
        return torch.mean((1/weight).view(length) * (out.view(length) -
    ↳ target.view(length)) ** 2, dim=0)

    if reduction == 'mean':
        return loss_mean
    else:
        return loss_sum
```

Zuletzt bleibt noch die Implementierung von $\mathscr{L}_2$.

$$\mathscr{L}_2(\theta_r|\hat{\theta}_y) = -\sum_{i:r_i=1} \log(\pi(y_i, h(x_i); \theta_r)) - \sum_{i:r_i=0} \log\left(1 - \mathbf{E}_{y_i}\left[\pi(y_i, h(x_i); \theta_r)\right]\right) \tag{12 Wdhg}$$

Da es für die Berechnung des Erwartungswert $\mathbf{E}_y\left[\pi(y, h(x_i); \theta_r)\right]$ drei Methoden gibt, sind auch drei Implementierungen vorhanden. Diese werden im Folgenden vorgestellt. Die Implementierungen sind jeweils aufgeteilt in die Berechnung der ersten und der zweiten Summe. Die ersten beiden Methoden funktionieren nur, wenn die Verteilung von Y bekannt ist. Daher kann man diese nur nutzen, wenn Y eine diskrete Zufallsvariable ist.

Direkte Berechnung des Erwartungswerts

Die direkte Berechnung des Erwartungswerts geht wie folgt:

$$\mathbf{E}_{y_i}\left[\pi(y_i, h(x_i); \theta_r)\right] = \sum_{k=1}^{K} P(y_i = k|\mathscr{G}^A(X); \hat{\theta}_y)\, \pi(k, h(x_i); \theta_r) \tag{13 Wdhg}$$

In der Implementierung wird $P(y_i = k|\mathscr{G}^A(X); \hat{\theta}_y)$ für alle $i \in \{i : r_i = 0\}$ und für alle $k \in \{1, ..., K\}$ im Parameter *y_dist* übergeben. Berechnet werden muss im Wesentlichen $\pi(k, h(x_i); \theta_r)$ für alle $i \in \{i : r_i = 0\}$ und für alle $k \in \{1, ..., K\}$.

```python
def evaluate_loss2_exact(x, y, mask, modelR, y_dist, num_classes, classification):
    N0 = sum(mask.logical_not())

    # Berechne pi(y_i, h(x_i)) für bekannte i
    y0 = y[mask].view(sum(mask), )
    y_one_hot = F.one_hot(y0.type(torch.int64), num_classes=num_classes)
    out1 = modelR(x[mask], y_one_hot.type(x.dtype))

    # Berechne expectation of pi(y, h(x_i)) für unbekannte i
    yh = torch.zeros((N0, num_classes))
    for k in range(num_classes):
        yb = torch.ones((N0,)) * k
        yb = F.one_hot(yb.type(torch.int64), num_classes)
        yb = yb.type(x.dtype)
        yh[:, k] = modelR(x[mask.logical_not()], yb).view(N0)
    out2 = torch.sum(yh * y_dist[mask.logical_not()], dim=1)

    if len(out2.shape) == 1:
        out2 = out2.view((len(out2), 1))

    return l2(out1, out2, reduction='sum')

def l2(piyi, piyib):
    return torch.sum(-torch.log(piyi)) + torch.sum(-torch.log(1-piyib))
```

Berechnung des Erwartungswert mit Stichproben

Es seien Stichproben $\{y_{ib}\}_{b=1}^{B} \sim P(y_i|\mathcal{G}^A(X)_i;\hat{\theta}_y)$ gegeben. Dann lässt sich der Erwartungswert wie folgt berechnen:

$$\mathbf{E}_{y_i}\left[\pi(y_i, h(x_i);\theta_r)\right] \approx \frac{1}{B}\sum_{b=1}^{B}\pi(y_{ib}, h(x_i);\theta_r) \qquad \text{(14 Wdhg)}$$

Für die Implementierung relevant ist vor allem die Berechnung von $\pi(y_{ib}, h(x_i);\theta_r)$ für alle $i \in \{i : r_i = 0\}$ und alle $b \in \{1, ..., B\}$.

Die Verteilung $P(y_i|\mathcal{G}^A(X)_i;\hat{\theta}_y)$, von welcher die Stichproben $\{y_{ib}\}_{b=1}^{B}$ genommen wurden, ändert sich im Laufe einer Trainingsepoche nicht. Daher ist es aus Effizienzgründen wichtig, dass diese bereits vor einer Trainingsepoche bestimmt wurden. Das ist auch der Grund, warum in der Implementierung diese als Parameter in Form einer Matrix $YIB \in \mathbb{R}^{N_0 \times B}$ entgegengenommen werden. Dabei sei $N_0 = \#\{i : r_i = 0\}$.

```python
def evaluate_loss2_sampling(x, y, mask, modelR, YIB, num_classes, B):
    N0 = sum(mask.logical_not())

    # Berechne pi(y_i, h(x_i)) für bekannte i
    y0 = y[mask].view(sum(mask), )
    y_one_hot = F.one_hot(y0.type(torch.int64), num_classes=num_classes)
    out1 = modelR(x[mask], y_one_hot.type(x.dtype)) # output für bekannte i; TODO:
    ↳ Check typechange this is ok

    # Berechne expectation of pi(y, h(x_i)) für unbekannte i
    out2 = torch.zeros((N0,B))
    for b in range(B):
        yb = YIB[:,b]
        yb = yb.view(yb.shape[0],)
        yb = F.one_hot(yb.type(torch.int64), num_classes)
        yb = yb.type(x.dtype) # TODO: Check if this is ok
        out2[:, b] = modelR(x[mask.logical_not()], yb).view(N0)
```

```
17
18      if len(out2.shape) == 1:
19          out2 = out2.view((len(out2), 1))
20
21      return l2(out1, out2)
22
23  def l2(piyi, piyib):
24      return torch.sum(-torch.log(piyi)) + torch.sum(-torch.log(1-piyib))
```

Berechnung des Erwartungswert mit $\mathbf{E}[y_i]$

Approximiere den Erwartungswert wie folgt:

$$\mathbf{E}_{y_i}\left[\pi(y_i, h(x_i); \theta_r)\right] \approx \pi(\mathbf{E}[y_i], h(x_i); \theta_r) \tag{17}$$

Aus einer mathematischen Sichtweise ist diese Methode den Erwartungswert zu berechnen mit Sicherheit die unpräziseste. Befindet man sich im kontinuierlichen Fall, hat man allerdings keine Alternative, da man über für unbekannte Knoten nur $\mathbf{E}[y_i]$ kennt und nicht, wie im diskreten Fall, die gesamte Verteilung. Für den diskreten Fall ist diese Methode dennoch interessant, da sie mit Abstand die schnellste ist. Man muss $\pi(\cdot, h(\cdot); \theta_r)$ nämlich nur einmal auswerten und zudem keine Stichprobe von P_Y nehmen.

Im diskreten Fall ist für ein $i \in \{1, ..., N\}$ der Erwartungswert von y_i definiert als

$$\mathbf{E}[y_i] := \underset{k=1,...,K}{\arg\max}\, P(y_i = k | \mathcal{G}^A(X)_i; \theta_y) \tag{18}$$

Die Implementierung sieht wie folgt aus:

```
1  def evaluate_loss2_fast(x, y_exp, mask, modelR, num_classes):
2      pi_est = modelR(x, F.one_hot(y_exp, num_classes).type(x.dtype))
3      return l2(pi_est[mask], pi_est[mask.logical_not()])
4
5  def l2(piyi, piyib):
6      return torch.sum(-torch.log(piyi)) + torch.sum(-torch.log(1-piyib))
```

6 Experimentelle Untersuchung des Algorithmus

Der in Kapitel 5 aufgeführte Algorithmus basiert auf zwei Schritten. Im ersten Schritt wird eine Schätzung für π gesucht und im zweiten Schritt wird eine Schätzung für Y gesucht. Diese beiden Schritte werden in den folgenden Abschnitten isoliert betrachtet und auf ihre Leistungsfähigkeit untersucht.

6.1 Schätzung von Y

Die Schätzung von Y basiert auf der Minimierung der $\mathscr{L}_1$ Loss Funktion:

$$\mathscr{L}_1(\theta_y | \hat{\theta}_r) = -\sum_{i:r_i=1} \frac{1}{\hat{\pi}_i} \log\left(P(y_i | \mathcal{G}^A(X)_i; \theta_y)\right) \tag{10 Wdhg}$$

bzw. im kontinuierlichen Fall

$$\mathscr{L}_1(\theta_y|\hat{\theta}_r) = -\sum_{i:r_i=1} \frac{1}{\hat{\pi}_i} \left(y_i - \alpha + \mathscr{G}^A(X)_i\beta\right)^2 \tag{11 Wdhg}$$

In diesem Abschnitt werden wir experimentell überprüfen, wie gut die Wahl dieser Loss Funktion ist. Dazu betrachten wir das Setting aus der Real-Data-Analysis von Zhou et al., 2019. Wir modifizieren den Cora Datensatz zu einem binären Datensatz, indem wir die Klasse 'Neural Networks' als Klasse 1 und die restlichen Klassen als Klasse 0 setzen. Damit haben wir

$$N_0 = \#\{y = 0\} = 1890 \quad \text{und} \quad N_1 = \#\{y = 1\} = 818$$

Des Weiteren wählen wir die Verteilung $P(r_i = 1|y_i, h(x_i))$, welche angibt, mit welcher Wahrscheinlichkeit das Label eines Knotens gegeben ist. Das ist nötig, da der konstruierte Algorithmus annimmt, dass die gegeben Label nicht zufällig sind (Missing not at Random; MNAR).

$$P(r_i = 1|y_i, h(x_i)) = \frac{\exp(\alpha_r + \gamma^T h(x_i) + \phi^T y_i)}{1 + \exp(\alpha_r + \gamma^T h(x_i) + \phi^T y_i)} \tag{19}$$

wobei

$$h(x_i) = \exp\left(\frac{\sum_j x_{ij}}{13} - 4\right) - \frac{\left(\sum_j x_{ij} - 15\right)}{15}, \tag{20}$$
$$\alpha_r = -\log(35), \quad \gamma = 1, \quad \phi = (0, 1.6)^T$$

Bei der Rechnung ist zu beachten, dass y_i one-hot-kodiert vorliegt. Wir definieren $\pi := (\pi_1, .., .\pi_N)^T$ mit $\pi_i := P(r_i = 1|y_i, h(x_i))$. Mit π ist der wahre Vektor für die MNAR Wahrscheinlichkeiten gegeben.

Setze die Anzahl der Klassenzugehörigkeit als $n_k := \#\{(y_i = k) \wedge (r_i = 1)\}$ und das Verhältnis von Klasse 1 zu Klasse 0 als $\lambda = {}^{n_1}/_{n_0}$. Anhand von π können wir uns Masken generieren. Um eine gute Übersicht zu gewährleisten, betrachten wir nur Masken, für die $\lambda \in \{1.2, 1.7, 2.2\}$ gilt. Zu jedem λ generieren wir 40 verschiedene Masken.

Weitergehend stören wir den wahren Vektor π verschieden stark. Dazu definiere

$$\pi_\sigma := \pi + \varepsilon, \quad \varepsilon \sim N(\mathbf{0}, \sigma^2 \boldsymbol{I})$$

Wir wollen untersuchen wie sich die Störungen auf die Vorhersagen eines Neuronalen Netzes auswirken. Wir betrachten ein 2-schichtiges Graph Convolutional Network (Kipf und Welling, 2016). Dieses Netzwerk wird mit jeder Maske und jeder Loss Funktion 4 mal trainiert. Als Referenzmodell trainieren wir es außerdem mit dem Negativ Log Likelihood Loss (NLL).

Die Ergebnisse dieses Experiments sind in Abbildung 1 und Tabelle 1 dargestellt.

λ	Loss	Mittelwert	Varianz
1.2	**NLL**	**0.878**	**2.18e-04**
	$\mathscr{L}_1$	**0.888**	**1.95e-04**
	$\mathscr{L}_1 + \sigma = 0.0025$	0.890	1.68e-04
	$\mathscr{L}_1 + \sigma = 0.005$	0.887	2.5e-04
	$\mathscr{L}_1 + \sigma = 0.0075$	0.883	6.74e-04
	$\mathscr{L}_1 + \sigma = 0.01$	0.866	1.53e-03
	$\mathscr{L}_1 + \sigma = 0.0125$	0.842	2.03e-03
	$\mathscr{L}_1 + \sigma = 0.015$	0.820	2.32e-03
	$\mathscr{L}_1 + \sigma = 0.0175$	0.802	2.87e-03
	$\mathscr{L}_1 + \sigma = 0.02$	0.770	4.43e-03
1.7	**NLL**	**0.852**	**4.14e-04**
	$\mathscr{L}_1$	**0.871**	**4.21e-04**
	$\mathscr{L}_1 + \sigma = 0.0025$	0.871	3.92e-04
	$\mathscr{L}_1 + \sigma = 0.005$	0.873	3.60e-04
	$\mathscr{L}_1 + \sigma = 0.0075$	0.868	5.79e-04
	$\mathscr{L}_1 + \sigma = 0.01$	0.858	8.75e-04
	$\mathscr{L}_1 + \sigma = 0.0125$	0.831	2.85e-03
	$\mathscr{L}_1 + \sigma = 0.015$	0.808	3.95e-03
	$\mathscr{L}_1 + \sigma = 0.0175$	0.790	2.86e-03
	$\mathscr{L}_1 + \sigma = 0.02$	0.753	6.02e-03
2.2	**NLL**	**0.816**	**1.5e-03**
	$\mathscr{L}_1$	**0.848**	**8.89e-04**
	$\mathscr{L}_1 + \sigma = 0.0025$	0.846	9.94e-04
	$\mathscr{L}_1 + \sigma = 0.005$	0.848	8.78e-04
	$\mathscr{L}_1 + \sigma = 0.0075$	0.846	9.73e-04
	$\mathscr{L}_1 + \sigma = 0.01$	0.830	1.58e-03
	$\mathscr{L}_1 + \sigma = 0.0125$	0.812	2.48e-03
	$\mathscr{L}_1 + \sigma = 0.015$	0.776	3.37e-03
	$\mathscr{L}_1 + \sigma = 0.0175$	0.749	5.33e-03
	$\mathscr{L}_1 + \sigma = 0.02$	0.723	6.34e-03

Tabelle 1: Mittelwerte (arithmetisches Mittel) und Varianzen (Cora)

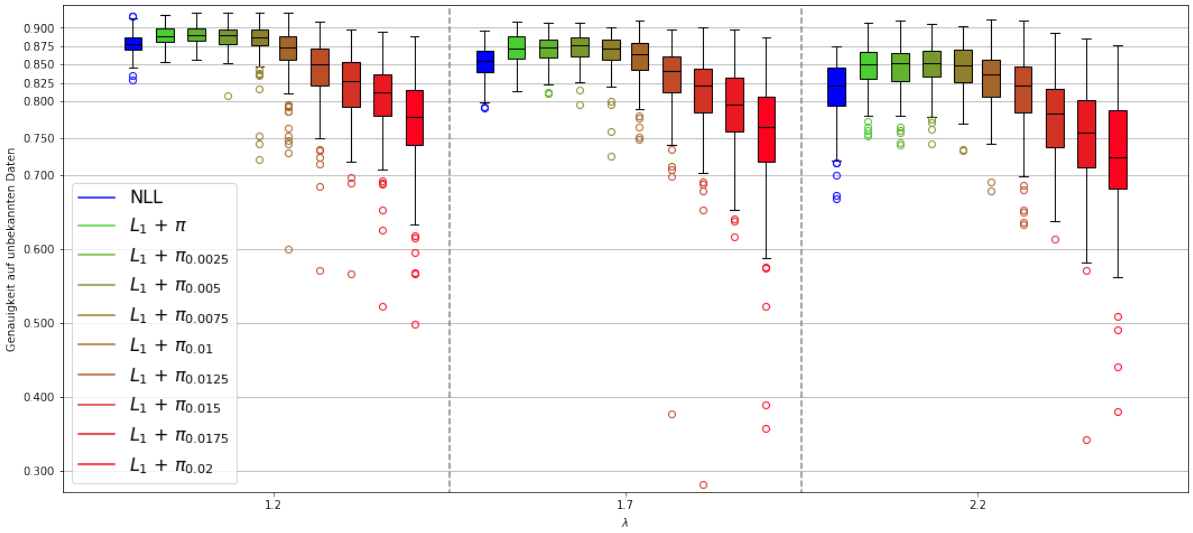


Abbildung 1: Stabilität der Schätzung von Y (Cora)

Die erste Erkenntnis, die aus dem Experiment gewonnen werden kann, ist, dass $\mathscr{L}_1$ in diesem Setup eine deutlich besser geeignete Loss Funktion ist als NLL, solange π richtig spezifiziert ist. Des Weiteren ist zu sehen, dass kleine Störungen an der Genauigkeit des resultierenden Modells keinen Einfluss nehmen. So kann man bei diesem Experiment auf dem Cora Datensatz für $\lambda = 1.2$ bei Störungen mit $\sigma \leq 0.005$ keinen Unterschied zur ungestörten Loss Funktion erkennen. Desto größer λ wird, desto mehr setzt sich $\mathscr{L}_1$ von NLL ab und desto größer darf die Störung werden, ohne dass sie schlechter als NLL ist.

Ein analoges Experiment wurde auf dem Citeseer Datensatz durchgeführt. Der Citeseer Datensatz unterscheidet sich vom Cora Datensatz unter anderem dadurch, dass in diesem deutlich mehr disjunkte Teilgraphen enthalten sind. Im Cora Datensatz sind die 2708 Knoten auf 78 disjunkte Untergraphen aufgeteilt, während im Citeseer Datensatz 3327 Knoten auf 438 disjunkte Untergraphen aufgeteilt sind. Die Struktur der Graphen wird von Rossi und Ahmed, 2015 dargestellt. Die Ergebnisse zum Experiment auf dem Citeseer Datensatz sind in Abbildung 2 zu sehen.

Hier setzt sich $\mathscr{L}_1$ nicht so stark von NLL ab, wie im ersten Experiment. Unverändert bleibt aber die Tatsache, dass Störungen bis zu einer gewissen Stärke keinen Einfluss auf die Genauigkeit nehmen.

Als Fazit können wir also annehmen, dass solange π ausreichend gut approximiert wird, sich unser Modell mit $\mathscr{L}_1$ besser trainieren lassen sollte als mit einer Loss Funktion, die nicht von π abhängt, wie z.B. NLL.

6.2 Schätzung von π

Essentiell für unseren Algorithmus ist es, dass die Schätzung von π besser wird, wenn die Schätzung von Y besser wird. D.h., dass, wenn wir unsere Schätzung von Y verbessern, unsere Schätzung von π sich ebenfalls verbessern sollte. Anderenfalls würde es sonst

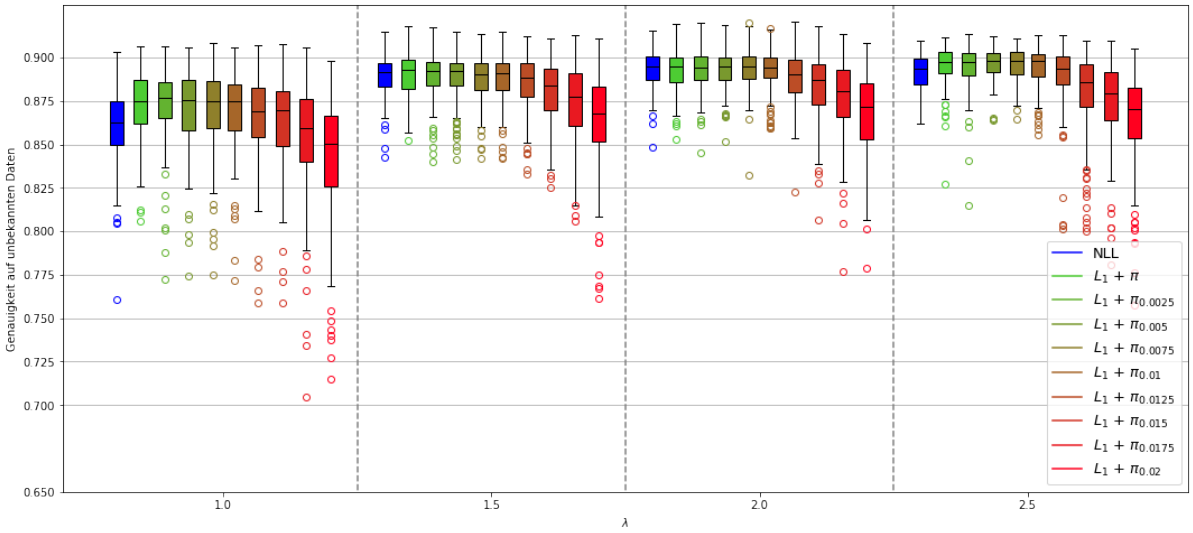


Abbildung 2: Stabilität der Schätzung von Y (Citeseer)

keinen Sinn ergeben, ein mehrfach das Model zur Schätzung von π zu trainieren.

Die Schätzung von π basiert auf $\mathscr{L}_2$:

$$\mathscr{L}_2(\theta_r|\hat{\theta}_y) = -\sum_{i:r_i=1} \log(\pi(y_i, h(x_i); \theta_r) - \sum_{i:r_i=0} \log\left(1 - \mathbf{E}_y\left[\pi(y, h(x_i); \theta_r)\right]\right) \qquad \text{(12 Wdhg)}$$

In drei Experiment werde ich dazu zeigen, dass - entgegen der Intuition - Rauschen auf dem übergebenen Y keinen Einfluss auf die Genauigkeit der Schätzung von π hat.

Für das erste Experiment betrachten wir dazu wieder einen ähnlich modifizierten Cora Datensatz wie schon in Abschnitt (6.1). π ist dadurch bereits eindeutig festgelegt. Für das Training von Model (1) benötigen wir die bisherige Schätzung von Y. Es hätte auch die Schätzung von der Verteilung von Y genutzt werden können. Da wir aber den Erwartungswert in Gleichung (12) mit jeweils nur einer Stichprobe ($B = 1$) berechnen, genügt hier direkt die Schätzung von Y, welche als Stichprobe aufgefasst werden kann.

In diesem Experiment werden wir diese Schätzung verschieden stark stören. Die Störung sieht dabei so aus, dass mit einer Wahrscheinlichkeit $p \in \{0, 0.1, 0.2, 0.4, 0.8, 1\}$ ein korrektes Klassenlabel durch ein zufälliges Klassenlabel ersetzt wird. Bei $p = 1$ haben wir somit zufällige Klassenlabels, während bei $p = 0$ die wahren Klassenlabels übergeben werden. Wir trainieren jeweils 2 Netzwerke auf 40 verschiedenen Masken. Also pro Störung werden 80 Netzwerke trainiert. Eine Schätzung $\hat{\pi}$ bewerten wir mit $\mathscr{L}_2$ wie in (12). Die Ergebnisse sind in Abbildung 3 dargestellt.

Die erste interessante Erkenntnis ist, dass der $\mathscr{L}_2$ Loss der Schätzung von π nur minimal schlechter wird, wenn Y gestört wird. Selbst, wenn Y komplett zufällig gewählt ist, bleibt der Loss ähnlich gut. Das heißt, in welcher Epoche wir π schätzen, scheint keinen wesentlichen Unterschied zu machen. Wir können es dementsprechend auch direkt zu Beginn schätzen.

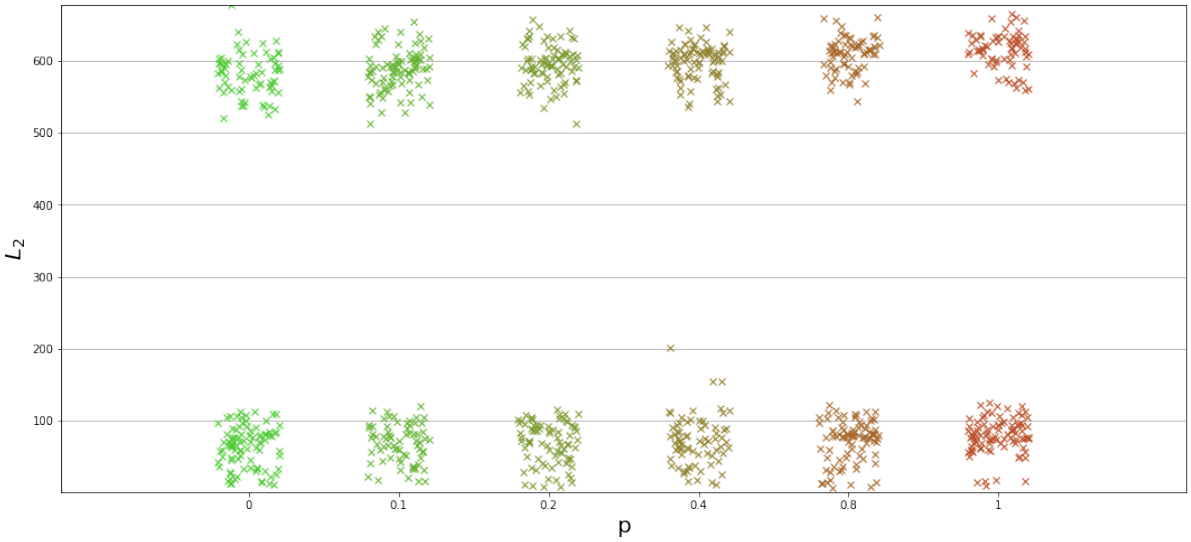


Abbildung 3: Genauigkeit der Schätzung von π (Cora)

Die zweite interessante Erkenntnis ist, dass die Ergebnisse für jedes p in zwei Klassen eingeteilt sind. $\mathscr{L}_2$ scheint zwei lokale Minima zu haben, auf die die Parameter etwa gleich häufig optimiert werden. Bezeichne im Folgenden Ergebnisse im kleinere lokalen Minimum als *Klasse 1* und Ergebnisse im größeren lokalen Minimum als *Klasse 2*. Im Experiment fielen etwa 50% der Ergebnisse in *Klasse 1* und 50% in *Klasse 2*.

Um das Ergebnis dieses Experiments zu untermauern, habe ich zwei weitere Experimente auf dem Citeseer Datensatz ausgeführt. Einmal war dieser zu einem 2-Klassen Datensatz modifiziert und einmal als 6-Klassen Datensatz belassen. In beiden Experimenten habe ich manuell eine Verteilung $P(r_i = 1|y_i, h(x_i))$ spezifiziert. Die Ergebnisse dazu sind in Abbildung 4 zu sehen.

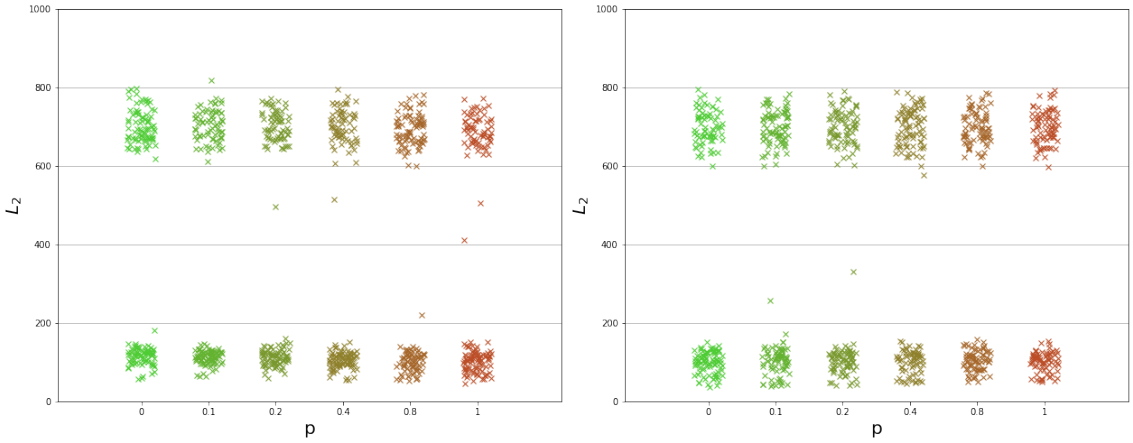
6.3 Verbesserung der Schätzung von π

Für das Training unseres Modells zur Approximation von Y ist es erstrebenswert, eine Schätzung für π zu haben, die in *Klasse 1* (Bezeichnung aus Abschnitt 6.2) liegt.

Dadurch motiviert versuche ich in diesem Abschnitt die Schätzung von π zu stabilisieren. Das Ziel ist es sich mit einer Wahrscheinlichkeit von über 99% in *Klasse 1* zu befinden. Die Ergebnisse aus 6.2 geben an, dass beim Training eines Modells die Wahrscheinlichkeit, in *Klasse 2* zu sein, bei ca. 50% liegt. Da $(1/2)^7 < 0.01$ schlage ich folgenden Algorithmus vor:

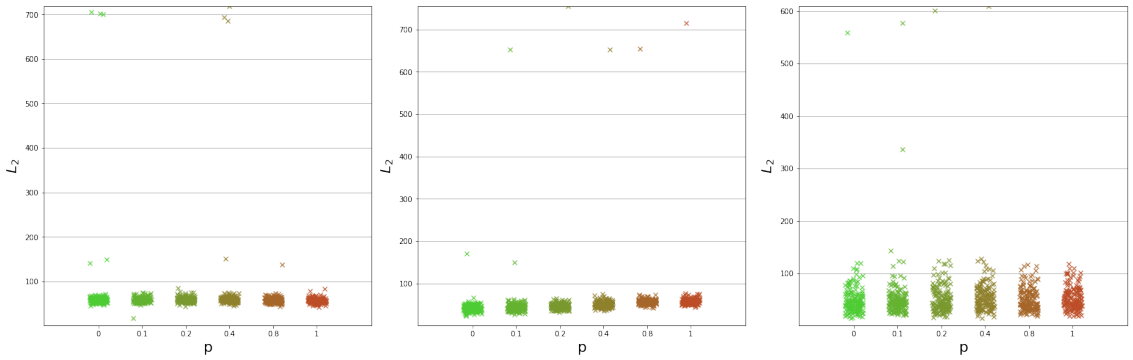
1. Trainiere das Modell 7 mal mit wenigen Trainingsiterationen.
2. Wähle nun das Modell mit geringstem $\mathscr{L}_2$ Loss und trainiere dieses mit gewohnter Zahl an Trainingsiterationen weiter.

Um die Effektivität dieses Algorithmus zu bestätigen, wiederhole ich die drei Experimente aus Abschnitt 6.2 mit dem modifizierten Trainingsalgorithmus.



(a) modifiziertes Citeseer mit 2 Klassen (b) Citeseer mit 6 Klassen

Abbildung 4: Genauigkeit der Schätzung von π (Citeseer)



(a) Citeseer 2 Klassen (b) Citeseer 6 Klassen (c) Cora 2 Klassen

Abbildung 5: Verbesserung der Schätzung von π

In Abschnitt 6.2 wurden die Modelle jeweils bis zu einer maximalen Anzahl von 100 Iterationen trainiert. Jetzt werden die Modelle im ersten Schritt mit bis zu 30 Iterationen trainiert. Im zweiten Schritt wird das beste Modell dann noch weitere bis zu 100 Iterationen trainiert. Damit werden maximal 310 Trainingsiterationen durchgeführt.

Die Ergebnisse dazu sind wie erwartet. Bis auf einige wenige Ausnahmen befindet sich das Ergebnis immer *Klasse 1*. In Abbildung 5 sind die Ergebnisse dargestellt.

7 Angepasster Algorithmus

In Kapitel 6 wurden zwei Erkenntnisse gezeigt:

1. Durch ein besser geschätztes $\hat{\theta}_y$ verbessert sich die Schätzung von θ_r nicht signifikant oder gar nicht.
2. Die Loss Funktion $\mathscr{L}_2$ zur Schätzung von θ_r hat zwei lokale Minima, die etwa gleich

oft eintreten. Dabei ist eines der beiden signifikant niedriger als das andere. Mit dem in Kapitel 6.3 vorgeschlagenen Algorithmus, erhält man mit einer Wahrscheinlichkeit von über 99% eine Schätzung von θ_r, welche sich in *Klasse 1*.

Diese Erkenntnisse motivieren dazu den in Kapitel 5 vorgeschlagenen Algorithmus anzupassen. Wir wollen das Konzept der Epochen verwerfen und stattdessen π nur noch einmal schätzen. Da die mathematische Grundlage zu $\mathscr{L}_2$ sehr solide scheint, trainieren wir zunächst die Parameter θ_y, um eine Schätzung für Y zu erhalten, bevor wir mit dieser Schätzung θ_r trainieren und dadurch π schätzen. Abschließend trainieren wir unsere eigentliche Parametermenge θ_y mit $\mathscr{L}_1$ unter der Nutzung unserer Schätzung für π.

Im Folgenden ist der neue Algorithmus detailliert aufgeschrieben:

1. Wähle eine beliebige passende Loss Funktion $\mathscr{L}$. Beispielsweise kann diese der Negativ Log Likelihood Loss sein. Wähle $\theta_y^{(0)}$ beliebig. Verbessere $\theta_y^{(0)}$ für $M^{(0)}$ viele Iterationen mit GD. In Iteration i haben wir:

   $$\theta_y^{(i+1)} \quad \leftarrow \quad \theta_y^{(i)} - \gamma_0 \nabla_{\theta_y} \mathscr{L}(\theta_y)$$

   γ_0 ist dabei die Lernrate. Schreibe $\hat{\theta}_y := \theta_y^{(M^{(0)})}$ nach der letzten Iteration.

2. Für $j = 0, ..., 6$:

   Wähle $\theta_r^{(j,0)}$ beliebig. Verbessere $\theta_r^{(j,0)}$ für $M^{(1)}$ viele Iterationen mit GD. In Iteration i haben wir:

   $$\theta_r^{(j,i+1)} \quad \leftarrow \quad \theta_r^{(j,i)} - \gamma_1 \nabla_{\theta_r} \mathscr{L}_2(\theta_r | \hat{\theta}_y)$$

   γ_1 ist dabei die Lernrate. Schreibe $\theta_r^{(j)} := \theta_r^{(j,M^{(1)})}$ nach der letzten Iteration.

3. Wähle nun $j_{\min} = \arg\min_j \mathscr{L}_2(\theta_r^{(j)} | \hat{\theta}_y)$, sodass $\theta_r^{(j_{\min})}$ die beste Parametermenge ist.

4. Verbessere $\theta_r^{(j_{\min},0)} := \theta_r^{(j_{\min})}$ für $M^{(2)}$ viele Iterationen mit GD. In Iteration i haben wir:

   $$\theta_r^{(j_{\min},i+1)} \quad \leftarrow \quad \theta_r^{(j_{\min},i)} - \gamma_2 \nabla_{\theta_r} \mathscr{L}_2(\theta_r | \hat{\theta}_y)$$

   γ_2 ist dabei die Lernrate. Schreibe $\hat{\theta}_r := \theta_r^{(j_{\min},M^{(2)})}$ nach der letzten Iteration.

5. Setze nun $\theta_y^{(0)} := \hat{\theta}_y$ und verbessere es für $M^{(3)}$ viele Iterationen mit GD unter Nutzung von $\mathscr{L}_1$. In Iteration i haben wir:

   $$\theta_y^{(i+1)} \quad \leftarrow \quad \theta_y^{(i)} - \gamma_0 \nabla_{\theta_y} \mathscr{L}_1(\theta_y | \hat{\theta}_r)$$

   Setze $\hat{\theta}_y := \theta_y^{(M^{(3)})}$.

6. $\hat{\theta}_y$ ist das Ergebnis des Algorithmus.

Zu beachten bei der Ausführung ist, dass $M^{(1)}$ deutlich niedriger als $M^{(2)}$ gewählt werden sollte. Abhängig davon wieviele Epochen der Algorithmus aus Kapitel 5 laufen würde, ist bei diesem Algorithmus eine wesentlich schnellere Ausführungszeit zu erwarten.

Damit sich die verbesserte Version des Algorithmus von der ursprünglichen in einer besseren Genauigkeit absetzt, ist wesentlich, dass Erkenntnis 1 von Beginn des Kapitels wahr ist. Die im folgenden Kapitel durchgeführten Experimente finden auf denselben Datensätzen wie in Kapitel 6 statt. Daher ist hier anzunehmen, dass Erkenntnis 1 gilt.

8 Vergleich der Algorithmen

In Kapitel 5 und Kapitel 7 wurden jeweils ein Algorithmus vorgestellt. In diesem Kapitel werden wir diese beiden in Experimenten vergleichen und auf ihre Stärken und Schwächen untersuchen.

Bezeichne den Algorithmus aus Kapitel 5 als *GNM* und den Algorithmus aus 7 als *GNMn*. Bezeichne ein Basis Model, welches die Verteilung $P(r_i = 1|y_i, h(x_i); \theta_r)$ ignoriert, als *SM*.

Wir untersuchen die Genauigkeit der Schätzung von Y in Abhängigkeit zum Verhältnis der Klassen in einer Trainingsmaske. Definiere dazu für die k-te Klasse von K Klassen $n_k := \#\{(r_i = 1) \wedge (y_i = k)\}$. Gilt $K = 2$ so definiere weiter $\lambda := {}^{n_1}/_{n_0}$. Ist $K > 2$ so definiere

$$\lambda := \frac{1}{K} \sum_{k=1}^{K} \left(n_k - \frac{1}{K} \sum_{k=1}^{K} n_k \right)^2$$

als die Varianz von $n_0, ..., n_K$.

Die Experimente werden auf dem Cora und dem Citeseer Datensatz durchgeführt. Je Datensatz findet ein Experiment mit voller Klassenzahl (6 bei Citeseer, 7 bei Cora) und ein Experiment, indem die Klassenzahl auf 2 reduziert wurde (siehe Kapitel 6), statt.

Für jedes Experiment ist eine Verteilung $P(r_i = 1|y_i, h(x_i))$ spezifiziert, von welcher Trainings- und Validationmasken berechnet werden. Dies ist nötig, da eine MNAR-Mechanismus vorausgesetzt wird. Die Verteilung ist wie folgt gesetzt:

$$P(r_i = 1|y_i, h(x_i)) = \frac{\exp(\alpha_r + \gamma^T h(x_i) + \phi^T y_i)}{1 + \exp(\alpha_r + \gamma^T h(x_i) + \phi^T y_i)} \tag{21}$$

wobei

$$h(x_i) = \exp\left(\frac{\sum_j x_{ij}}{a_0} - a_1 \right) - \frac{\left(\sum_j x_{ij} - a_2 \right)}{a_3} \tag{22}$$

Die Variablen $a_0, ..., a_3, \alpha_r, \gamma$ und ϕ sind in jedem Experiment individuell so gesetzt, dass sich die Verteilung möglichst von einer Gleichverteilung unterscheidet und zudem $\sum_i {}^{\pi_i}/_N \approx 0.05$ ergibt. Das bedeutet, dass stets etwa 5% der Labels bekannt sind.

$\mathcal{G}^A$ wird in allen 4 Experimenten als ein 2-lagiges Graph Convolutional Network (Kipf und Welling, 2016) gewählt. h wird als ein dreischichtiges Perzeptron gewählt.

Die Ergebnisse sind in Abbildung 6 zu sehen. Die Hauptlinien geben eine Approximation des Erwartungswertes an. Die Nebenlinien stellen eine Approximation der Standardabweichung dar.

In den Ergebnissen erkennt man folgenden Dinge:

- *GNM* ist für hohe λ besser geeignet als *SM* bzw. *GNMn*. Besonders im Experiment auf Citeseer mit 2 Klassen ist zu sehen, dass die Genauigkeit der Schätzung von *GNM* bei $\lambda = 6$ mehr als einen Prozentpunkt über der Genauigkeit von *SM* liegt.
- *GNMn* ist in kaum einem Fall der beste Algorithmus. Allerdings auch fast nie der Schlechteste. In den Experimenten auf Cora mit 2 und mit 7 Klassen sowie im Experiment auf Citeseer mit 6 Klassen, ist die Genauigkeit stets sehr ähnlich zur Genauigkeit von *SM*. Das Experiment auf Citeseer mit 2 Klassen ist das einzige auf dem sich *GNMn* gegenüber *SM* signifikant absetzt.
- Zuletzt ist zu bemerken, dass die Standardabweichung von allen drei Algorithmen meistens sehr ähnlich ist. Nur im Experiment auf Cora mit 7 Klassen hat *SM* eine etwas höhere Standardabweichung als *GNM* und *GNMn*.

Zu beachten bei diesen Ergebnissen ist, dass etwa 5% der Labels gegeben sind. Das bedeutet, dass es nur wenige $i \in \{1, ..., N\}$ geben kann, für die π_i signifikant größer als 0.05 ist. Also ist der Einfluss des MNAR-Mechanismus nicht sehr stark.

9 Fazit

Aus den Ergebnissen von Kapitel 8 wird klar, dass der *GNM* Algorithmus durchaus Anwendung finden kann. Sollte für einen Datensatz bekannt sein, dass ein starker MNAR-Mechanismus vorliegt, so ist es empfehlenswert diesen Algorithmus zu verwenden.

Im Paper „Graph-based Semi-Supervised Learning with Nonignorable Missingness“ von Zhou et al., 2019 überragte *GNM* dem Standard Model *SM* teilweise mit bis zu 30 Prozentpunkten. Diese herausragenden Ergebnisse konnte ich nicht rekonstruieren.

Ob es Situationen gibt, in denen der *GNMn* Algorithmus aus Kapitel 7 Anwendung finden könnte, ist unklar. Liegt ein starker MNAR-Mechanismus vor, übertrumpft *GNMn* zwar *SM*, scheint aber *GNM* zu unterliegen. Andererseits scheint *GNMn* in Situationen, in denen *GNM* schlechter als *SM* schätzt, ebenfalls schlechter als *SM* zu schätzen.

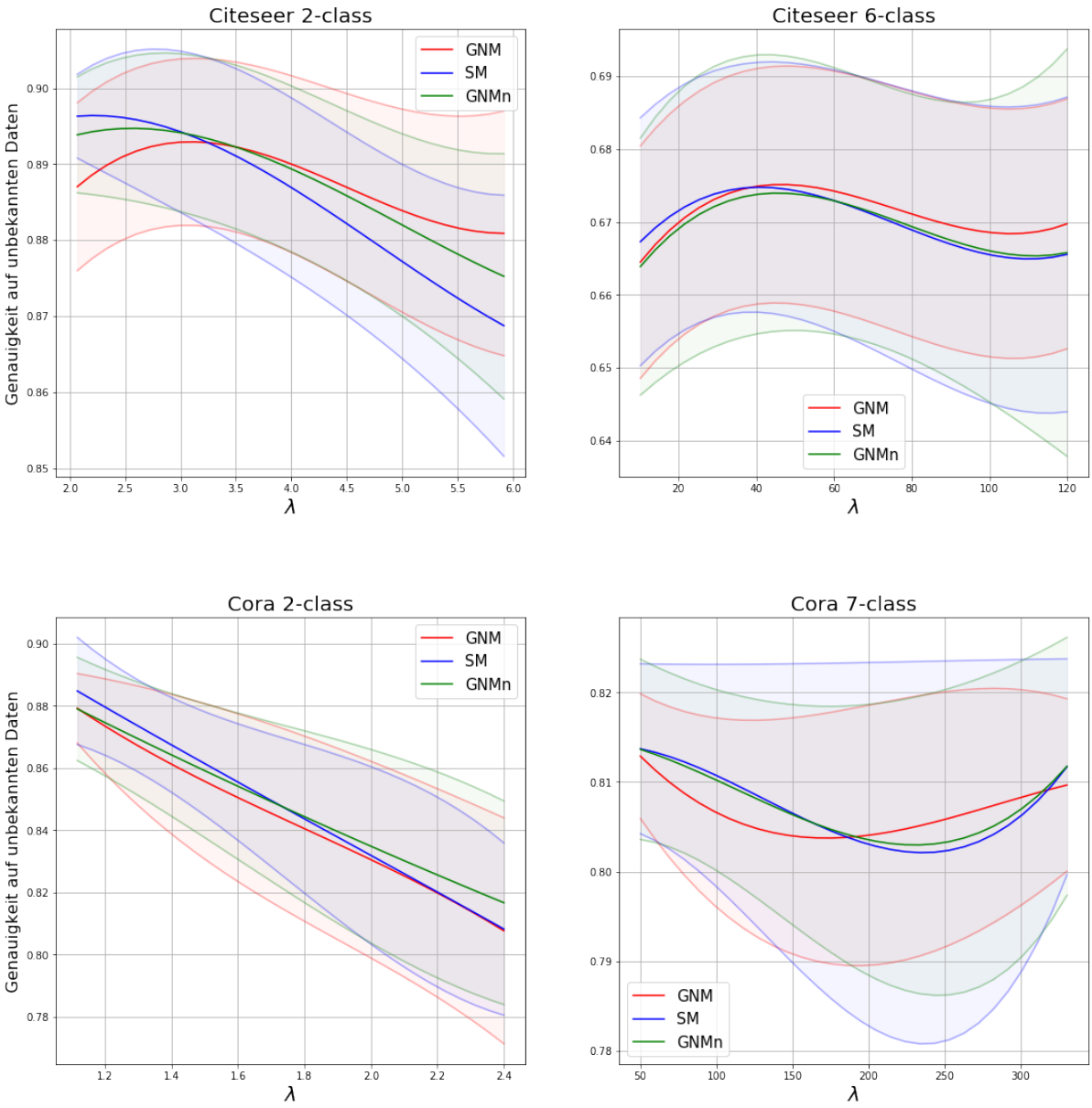


Abbildung 6: Experimente zum Vergleich der Algorithmen

Literatur

Heejung Bang und James M. Robins (2005). „Doubly Robust Estimation in Missing Data and Causal Inference Models“. In: *Biometrics* 61.4, S. 962–973. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1541-0420.2005.00377.x.

Matthias Fey und Jan E. Lenssen (2019). „Fast Graph Representation Learning with PyTorch Geometric“. In: *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

D. G. Horvitz und D. J. Thompson (1952). „A Generalization of Sampling Without Replacement From a Finite Universe“. In: *Journal of the American Statistical Association* 47.260, S. 663–685.

Thomas N. Kipf und Max Welling (2016). *Semi-Supervised Classification with Graph Convolutional Networks*. arXiv: 1609.02907 [cs.LG].

James Robins, A G Rotnitzky und Lue Zhao (Sep. 1994). „Estimation of Regression Coefficients When Some Regressors Are Not Always Observed“. In: *Journal of The American Statistical Association - J AMER STATIST ASSN* 89, S. 846–866.

Ryan A. Rossi und Nesreen K. Ahmed (2015). „The Network Data Repository with Interactive Graph Analytics and Visualization“. In: *AAAI*.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò und Yoshua Bengio (2017). *Graph Attention Networks*. arXiv: 1710.10903 [stat.ML].

Fan Zhou, Tengfei Li, Haibo Zhou, Hongtu Zhu und Ye Jieping (2019). „Graph-Based Semi-Supervised Learning with Non-ignorable Non-response“. In: *Advances in Neural Information Processing Systems 32*. Hrsg. von H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox und R. Garnett. Curran Associates, Inc., S. 7015–7025.

Abbildungsverzeichnis

Tabellenverzeichnis