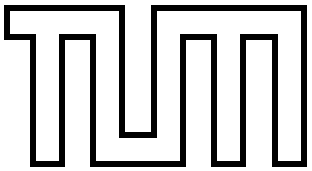


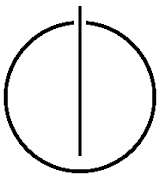
FAKULTÄT FÜR INFORMATIK

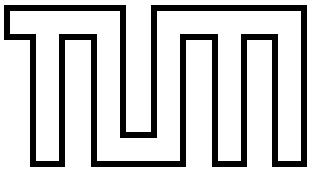
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Abschlussarbeit in Informatik

Effiziente statistische Methoden für Datenbanksysteme

Thomas Heyenbrock





FAKULTÄT FÜR INFORMATIK

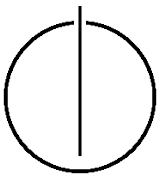
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Abschlussarbeit in Informatik

Effiziente statistische Methoden für Datenbanksysteme

Efficient statistical methods for database systems

Autor:	Thomas Heyenbrock
Aufgabensteller:	Prof. Alfons Kemper, Ph.D.
Betreuer:	Maximilian E. Schüle, M.Sc.
Datum:	15.01.2017



Ich versichere, dass ich diese Abschlussarbeit selbständig verfasst und nur die angegebenen
Quellen und Hilfsmittel verwendet habe.

München, den 10. Januar 2018 Thomas Heyenbrock

Abstract

An abstracts abstracts the thesis!

Contents

Outline of the Thesis

Teil I: Introduction and Theory

CHAPTER 1: INTRODUCTION

This chapter presents an overview of the thesis and it purpose. Furthermore, it will discuss the sense of life in a very general approach.

CHAPTER 2: THEORY

No thesis without theory.

Teil II: The Real Work

CHAPTER 3: OVERVIEW

This chapter presents the requirements for the process.

1. Einführung und typische statistische Problemstellungen

Here starts the thesis with an introduction. Please use nice latex and bibtex entries [2]. Do not spend time on formating your thesis, but on its content.

1.1. Latex Introduction

There is no need for a latex introduction since there is plenty of literature out there.

2. Grundlagen statistischer Methoden

Bei der Regressionsanalyse geht es im Allgemeinen darum, das Verhalten einer Größe Y in Abhängigkeit einer oder mehrerer anderer Größen $X_1, X_2, \ldots, X_n$ zu modellieren. Die Größe Y wird abhängig genannt, die Größen X_i nennt man unabhängig. Für diese Arbeit wollen wir zunächst einige Annahnmen über diese voraussetzen. Diese Punkte gelten immer, falls nicht explizit etwas anderes festgelegt wird.

- Die genannten Größen sind Zufallsvariablen. Das sind Funktionen deren Werte die Ergebnisse eines Zufallsvorgangs darstellen.
- Die Zufallsvariablen sind auf der Menge $M = \{1, \ldots, m\}$ definiert und bilden in die reellen Zahlen ab:

  $$Y : M \to \mathbb{R}, \;\; X_1 : M \to \mathbb{R}, \;\; \ldots, \;\; X_n : M \to \mathbb{R}$$

  Das bedeutet die Zufallsvariablen sind metrisch skaliert. Die m Zahlen in der Menge M entsprechen den m Datenpunkten, die wir als Datenbasis für die Regressionsanalyse besitzen.
- Wir verwenden die folgenden Abkürzungen für die Werte der Zufallsvariablen:

  $$\begin{aligned} y_i &:= Y(i) \quad \text{für alle } i \in M, \\ x_{i,j} &:= X_j(i) \quad \text{für alle } i \in M \;\; \text{und } 1 \leq j \leq n \end{aligned}$$

- Einen Datenpunkt aus unserer Datenbasis fassen wir als Vektor der Länge $(n+1)$ auf. Damit lässt sich die Datenbasis schreiben als:

  $$(y_1, x_{1,1}, \ldots, x_{1,n}), \ldots, (y_m, x_{m,1}, \ldots, x_{m,n})$$

Das Modell definieren wir anhand einer Funktion f, welche für Werte der unabhängigen Variablen einen geschätzten Wert für die abhängige Variable liefert. Idealerweise existiert eine Funktion, die zum Einen eine einfache Darstellung (z.B. durch eine arithmetische Formel) besitzt und zum Anderen alle unabhängigen Werte der Datenmenge exakt prognostiziert. Das bedeutet:

$$y_i = f(x_{i,1}, \ldots, x_{i,n}) \quad \text{für alle } \; 1 \leq i \leq m$$

Falls eine Formel wie hier für alle Datenpunkte gelten soll, verwenden wir als Abkürzung auch die Zufallsvariablen selbst, also:

$$Y = f(X_1, \ldots, X_N)$$

Im Allgemeinen ist es nicht möglich eine Funktion f zu finden, die beide Eigenschaften erfüllt. Man versucht also eine Funktion mit einer möglichst einfachen Form zu finden, die die Datenmenge möglichst gut approximiert. Wir definieren für jeden Datenpunkt den Fehler e_i, der sich durch die nicht exakte Modellfunktion f ergibt:

$$e_i = y_i - f(x_{i,1}, \ldots, x_{i,n})$$

Ziel der Regressionsanalyse ist es nun eine Funktion f zu finden, die diese Fehlerterme minimiert. Diese Optimierung geschieht global, also für die gesamte Datenmenge und nicht nur für einzelne Datenpunkte.

2.1. Lineare Regression

Bei der linearen Regression geht man von einem linearen Zusammenhang zwischen der abhängigen und den unabhängigen Variablen aus. Die Funktion f ist also von folgender Form:

$$f(x_1, \ldots, x_n) = \alpha + \sum_{i=1}^{n} \beta_i \cdot x_i \quad \text{mit} \quad \beta_i \in \mathbb{R}$$

Das Maß für die Qualität einer Funktion f definiert durch die Parameter $\alpha, \beta_1, \ldots, \beta_n$ ist die Summe der quadrierten Fehlerterme:

$$E(\alpha, \beta_1, \ldots, \beta_n) = \sum_{j=1}^{m} e_j^2 = \sum_{j=1}^{m} (y_j - f(x_{j,1}, \ldots, x_{j,n}))^2 = \sum_{j=1}^{m} \left(y_j - \alpha - \sum_{i=1}^{n} \beta_i \cdot x_{i,j} \right)^2$$

Wir suchen also die Parameter $\hat{\alpha}, \hat{\beta}_1, \ldots, \hat{\beta}_n$ für die gilt:

$$E(\hat{\alpha}, \hat{\beta}_1, \ldots, \hat{\beta}_n) = min\left\{ E(\alpha, \beta_1, \ldots, \beta_n) \mid \alpha \in \mathbb{R}, \beta_1 \in \mathbb{R}, \ldots, \beta_n \in \mathbb{R} \right\}$$

Um dieses Minimierungsproblem zu lösen berechnen wir die partiellen Ableitungen von E.

$$\begin{aligned}
\frac{\partial E}{\partial \alpha} &= -2 \cdot \sum_{j=1}^{m} (y_j - f(x_{j,1}, \ldots, x_{j,n})) = -2 \cdot \sum_{j=1}^{m} \left(y_j - \alpha - \sum_{i=1}^{n} \beta_i \cdot x_{i,j} \right) \\
\frac{\partial E}{\partial \beta_k} &= -2 \cdot \sum_{j=1}^{m} x_{k,j} \cdot (y_j - f(x_{j,1}, \ldots, x_{j,n})) \\
&= -2 \cdot \sum_{j=1}^{m} x_{k,j} \cdot \left(y_j - \alpha - \sum_{i=1}^{n} \beta_i \cdot x_{i,j} \right) \quad \text{für } 1 \leq k \leq n
\end{aligned}$$

Durch Nullsetzen der partiellen Ableitungen erhält man ein lineares Gleichungssystem mit $(n+1)$ Gleichungen und ebensovielen Unbekannten.

$$\frac{\partial E}{\partial \alpha} = 0, \quad \frac{\partial E}{\partial \beta_1} = 0, \quad \ldots, \quad \frac{\partial E}{\partial \beta_n} = 0$$

Die Lösung dieses Gleichungssystems (falls eine existent) ist das gesuchte Minimum.

2.1.1. Einfache lineare Regression

Man spricht von einfacher linearer Regression, wenn man mit nur eine unabhängige Variable arbeitet. Anschaulich möchte mann hier die bestmögliche Schätzgerade durch eine gegebene Punktwolke legen.

Wir nennen die unabhängige Variable in diesem Kapitel statt X_1 einfach nur X. Ebenso schreiben wir $\beta_1 = \beta$ und $x_{1,j} = x_j$. Dann können wir das lineare Gleichungssystem zum Auffinden des Minimums explizit aufschreiben:

$$0 = -2 \cdot \sum_{j=1}^{m} (y_i - \alpha - \beta \cdot x_j)$$
$$0 = -2 \cdot \sum_{j=1}^{m} x_j \cdot (y_i - \alpha - \beta \cdot x_j)$$

Für dieses Gleichungssystem kann die Lösung explizit angegeben werden, wobei wir hier nicht näher auf die Herleitung dieses Ergebnisses eingehen wollen:

$$\hat{\beta} = \frac{\sum_{j=1}^{m} (x_j - \bar{x})(y_j - \bar{y})}{\sum_{j=1}^{m} (x_j - \bar{x})^2}$$
$$\hat{\alpha} = \bar{y} - \hat{\beta}\bar{x}$$

Dabei bezeichnen $\bar{x}$ und $\bar{y}$ die Mittelwerte von X respektive Y.

2.1.2. Multible lineare Regression

Bei multibler linearer Regression existieren mindestens zwei unabhängige Variablen. Hier ist es nicht mehr zweckmäßig eine explizite Lösung anzugeben. Hier sind alternative Methoden zur Berechnung der Parameter nötig.

Neben einer Vielzahl von Algorithmen, die ein Optimierungsproblem iterativ lösen, gibt es auch die Möglichkeit die Parameter durch Matrizenmultiplikation zu berechnen. Definieren wir dazu die folgenden Matrizen und Vektoren:

$$X = \begin{pmatrix} 1 & x_{1,1} & \dots & x_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{m,1} & \dots & x_{m,n} \end{pmatrix} \in \mathbb{R}^{m \times (n+1)}$$

$$y = \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix} \in \mathbb{R}^{m \times 1}, \quad b = \begin{pmatrix} \hat{\alpha} \\ \hat{\beta}_1 \\ \vdots \\ \hat{\beta}_n \end{pmatrix} \in \mathbb{R}^{(n+1) \times 1}$$

Dabei ist b der Vektor mit gesuchten Parametern für die Minimierung der kleinsten Quadrate. Falls die Matrix $X^T X$ invertiertbar ist, gilt die folgende Formel für die Berechnung der gesuchten Parameter:

$$b = (X^T X)^{-1} X^T y$$

2.2. Logistische Regression

Die logistische Regression findet Anwendung im Falle, dass die abhängige Variable eine binäre Variable ist, also eine Variable, die nur zwei Werte annehmen kann. Oft handelt es sich um eine Eigenschaft, die ein bestimmter Datensatz besitzt oder nicht, wie zum Beispiel ein Premium-Abonnement für eine Web-Service oder der Besitz eines Auto. Auch das Geschlecht einer Person ist ein Beispiel für eine binäre Variable. Wir bezeichnen die beiden möglichen Werte einer solchen Variablen hier immer mit 0 und 1. Die Zuordnung vom Merkmal zur Zahl ist frei wählbar.

Lineare Regression eignet sich oft nicht zur Modellierung einer binären Variablen, da eine lineare Funktion in der Regel unbeschränkt ist, also insbesondere Werte größer als 1 und kleiner als 0 annimmt. Um diesem Problem abzuhelfen wählen wir eine Funktion, die beliebige Zahlen auf das Interval $[0, 1]$ abbildet. Im Falle der logistischen Regression verwendet man die gleichnamige logistische Funktion:

$$l : \mathbb{R} \to (0, 1), \quad x \mapsto \frac{1}{1 + e^{-x}}$$

Diese Funktion wendet man nun auf die Linearkombination aller unabhängigen Variablen mit Parametern $\beta_1, \ldots, \beta_n$ und konstantem Term α an. Zur Vereinfachung definieren wir für das restliche Kapitel die Variable c wiefolgt:

$$c := \alpha + \sum_{i=1}^{n} \beta_i \cdot x_{i,j}$$

Das Ergebnis der Funktion l für den i-ten Datensatz bezeichnen wir mit π_i

$$\pi_i = \pi_i(\alpha, \beta_1, \ldots, \beta_n) := l\left(\alpha + \sum_{j=1}^{n} \beta_j \cdot x_{i,j}\right) = \frac{1}{1 + e^{-c}}$$

Wir stellen hierbei fest, dass folgende Identität gilt:

$$\begin{aligned}
\pi(-\alpha, -\beta_1, \ldots, -\beta_n) &= \frac{1}{1 + e^{c}} = \frac{1 + e^{c} - e^{c}}{1 + e^{c}} \\
&= 1 - \frac{e^{c}}{1 + e^{c}} = 1 - \frac{1}{e^{-c} + 1} \\
&= 1 - \pi(\alpha, \beta_1, \ldots, \beta_n)
\end{aligned}$$

Anschaulich repräsentiert π_i die Wahrscheinlichkeit dafür, dass die abhängige Variable eines Datensatzes mit unabhängigen Variablen $x_{i,1}, \ldots, x_{i,n}$ gleich 1 ist, also:

$$\pi_i = P(Y_i = 1 | X_1 = x_{i,1}, \ldots, X_n = x_{i,n})$$

Man möchte die Parameter $\alpha, \beta_1, \ldots, \beta_n$ nun so schätzen, dass die Wahrscheinlichkeit für das Auftreten der vorhandenen Datenbasis maximiert wird. Diese Wahrscheinlichkeit ist

gegeben durch:

$$\begin{aligned} L(\alpha, \beta_1, \ldots, \beta_n) &= \prod_{i=1}^{m} P(Y_i = y_i | X_1 = x_{i,1}, \ldots, X_n = x_{i,n}) \\ &= \prod_{i=1}^{m} y_i \cdot \pi_i(\alpha, \beta_1, \ldots, \beta_n) + (1 - y_i) \cdot (1 - \pi_i(\alpha, \beta_1, \ldots, \beta_n)) \\ &= \prod_{i=1}^{m} y_i \cdot \pi_i(\alpha, \beta_1, \ldots, \beta_n) + (1 - y_i) \cdot \pi_i(-\alpha, -\beta_1, \ldots, -\beta_n)) \end{aligned}$$

Da alle y_i im Fall der logistischen Regression entweder gleich 0 oder gleich 1 sind, ist immer nur einer der beidem Summanden in jedem Faktor nicht null. Diese Fallunterscheidung kann man auch in das Vorzeichen der Parameter verschieben, da sich die beiden möglichen Faktoren nur darin unterscheiden. Dann erhält man:

$$\begin{aligned} L(\alpha, \beta_1, \ldots, \beta_n) = \prod_{i=1}^{m} \pi((&2 \cdot y_i - 1) \cdot \alpha, \\ &(2 \cdot y_i - 1) \cdot \beta_1, \\ &\ldots, \\ &(2 \cdot y_i - 1) \cdot \beta_n) \end{aligned}$$

Das Verfahren der Maximierung dieser Wahrscheinlichkeit bezeichnet man auch als Maximum-Likelihood-Methode. Die Funktion L nennt man dementsprechend auch Likelihoodfunktion. Oft maximiert man nicht L direkt, sondern eher $\ln(L)$. Der Sinn ist, dass man das Produkt damit in eine Summe einzelner Logarithmen umwandeln kann, welche wiederum einfacher abzuleiten ist. Das darf man machen, da der Logarithmus eine stetig wachsende Funktion ist und die Werte von L stets zwischen 0 und 1 liegen.

Wir suchen also die Parameter $\hat{\alpha}, \hat{\beta}_1, \ldots, \hat{\beta}_n$ mit:

$$L(\hat{\alpha}, \hat{\beta}_1, \ldots, \hat{\beta}_n) = \max \{L(\alpha, \beta_1, \ldots, \beta_n) \mid \alpha \in \mathbb{R}, \beta_1 \in \mathbb{R}, \ldots, \beta_n \in \mathbb{R}\}$$

In diesem Fall kommt man leider nicht mehr an einer iterativen Lösung vorbei, da die partiellen Ableitungen und das entstehende lineare Gleichungssystem nicht mehr exakt lösbar sind. Eine der einfachsten Methoden zur Lösung von Optimierungsproblemen ist das Gradientenverfahren, welches im kommenden Teilkapitel kurz eingeführt wird.

2.2.1. Gradientenverfahren

Das Gradientenverfahren ist ein iterativer Algorithmus zur Lösung von Optimierungsproblemen. Nachdem wir hier bei der logistischen Regression eine Funktion maximieren wollen führen wir das Gradientenverfahren dementsprechend ein. Man kann dasselbe Verfahren aber auch zur Lösung von Minimierungsproblem einsetzen. Gegeben sei also eine Funktion der folgenden Form, die maximiert werden soll:

$$L : \mathbb{R}^{n+1} \to \mathbb{R}, \;\; (\alpha, \beta_1, \ldots, \beta_n) \mapsto L(\alpha, \beta_1, \ldots, \beta_n)$$

Beim Gradientenverfahren beginnt man mit beliebigen Startwerten $\alpha_0, \beta_{0,1}, \ldots, \beta_{0,n}$ und einer Schrittweite $s \in \mathbb{R}^+$. Vom Startpunkt aus geht man nun in die Richtung des steilsten

Anstieges der Funktion und erhält dadurch neue Werte. Diese Richtung ist gerade der sogenannte Gradient der Funktion L.

Der Gradient ist ein Vektor, der sich aus den partiellen Ableitungen von L nach jeweils einer Variablen zusammensetzt und wird wiefolgt notiert:

$$\text{grad}(L) = \begin{pmatrix} \partial L/\partial\alpha \\ \partial L/\partial\beta_1 \\ \vdots \\ \partial L/\partial\beta_n \end{pmatrix}$$

Der Gradient von L ist also wiederum eine Funktion, die Werte $\alpha, \beta_1, \ldots, \beta_n$ auf einen Vektor der Länge $n+1$ abbildet. Der iterative Schritt des Verfahrens definiert sich wiefolgt:

$$\begin{pmatrix} \alpha_{i+1} \\ \beta_{i+1,1} \\ \vdots \\ \beta_{i+1,n} \end{pmatrix} = \begin{pmatrix} \alpha_i \\ \beta_{i,1} \\ \vdots \\ \beta_{i,n} \end{pmatrix} + s \cdot \text{grad}(L)(\alpha_i, \beta_{i,0}, \ldots, \beta_{i,n})$$

Danach muss noch getestet werden, dass L für die neuen Parameter auch wirklich einen größeren Wert annimmt also zuvor. Falls nicht, muss die Schrittweite s verkleintert werden, zum Beispiel um einen festen zuvor definierten Faktor.

Das Verfahren konvergiert nicht zwingend, falls die Funktion nach oben unbeschränkt ist. In unserem Fall ist die Likelihoodfunktion L aber durch 1 nach oben beschränkt. Trotzdem konvergiert das Gradientenverfahren nur mit Sicherheit gegen ein lokales Maximum von L, welches nicht zwingend auch ein globales Maximum sein muss.

2.2.2. Gradient bei logistischer Regression

Um das Gradientenverfahren bei logistischer Regression einsetzen zu können, muss der Gradient für den Logarithmus der Likelihoodfunktion bekannt sein. In diesem Kapitel bilden aso wir die partiellen Ableitungen nach allen Parametern.

Um $L_{log} = \ln(L)$ partiell ableiten zu können, berechnen wir zuerst die partiellen Ableitungen aller π_i. Für die partielle Ableitung nach α ergibt sich mit der Kettenregel folgende Funktion:

$$\begin{aligned} \frac{\partial \pi_i}{\partial \alpha} &= -\left(1 + \exp\left(-\alpha - \sum_{j=1}^{n} \beta_j \cdot x_{i,j}\right)\right)^{-2} \cdot \exp\left(-\alpha - \sum_{j=1}^{n} \beta_j \cdot x_{i,j}\right) \cdot (-1) \\ &= \left(1 + \exp\left(-\alpha - \sum_{j=1}^{n} \beta_j \cdot x_{i,j}\right)\right)^{-1} \cdot \left(1 + \exp\left(\alpha + \sum_{j=1}^{n} \beta_j \cdot x_{i,j}\right)\right)^{-1} \\ &= \pi_i(\alpha, \beta_1, \ldots, \beta_n) \cdot \pi_i(-\alpha, -\beta_1, \ldots, -\beta_n) \end{aligned}$$

Die partiellen Ableitungen einem der β_k für $k = 1, \ldots, n$ kann fast analog gebildet werden. Bei der Anwendung der Kettenregel auf die innerste lineare Funktion bleibt jedoch noch der konstanter Faktor $x_{i,k}$ übrig.

$$\frac{\partial \pi_i}{\partial \beta_k} = x_{i,k} \cdot \pi_i(\alpha, \beta_1, \ldots, \beta_n) \cdot \pi_i(-\alpha, -\beta_1, \ldots, -\beta_n)$$

Nun zur den eigentlichen partiellen Ableitunge. Zuerst noch einmal zu der Funktion, die wir nun ableiten wollen. Definieren wir $\tilde{\alpha} := (2y_i - 1)\alpha$ und $\tilde{\beta}_i := (2y_i - 1)\beta_i$ für $i = 1, \ldots, n$. Dann erhält man:

$$\begin{aligned} L_{log}(\alpha, \beta_1, \ldots, \beta_n) &= \ln(L(\alpha, \beta_1, \ldots, \beta_n)) \\ &= \sum_{i=1}^{m} \ln\left(y_i \cdot \pi(\tilde{\alpha}, \tilde{\beta}_1, \ldots, \tilde{\beta}_n)\right) \end{aligned}$$

Leitet man nach α ab, so erhält man:

$$\begin{aligned} \frac{\partial L_{log}}{\partial \alpha} &= \sum_{i=1}^{m} \frac{\partial}{\partial \alpha} \left(\ln\left(\pi_i(\tilde{\alpha}, \tilde{\beta}_1, \ldots, \tilde{\beta}_n)\right)\right) \\ &= \sum_{i=1}^{m} \left(\pi_i(\tilde{\alpha}, \tilde{\beta}_1, \ldots, \tilde{\beta}_n)\right)^{-1} \cdot \frac{\partial \pi_i}{\partial \tilde{\alpha}} \cdot \frac{\partial \tilde{\alpha}}{\partial \alpha} \\ &= \sum_{i=1}^{m} \left(\pi_i(\tilde{\alpha}, \tilde{\beta}_1, \ldots, \tilde{\beta}_n)\right)^{-1} \cdot \pi_i(\tilde{\alpha}, \tilde{\beta}_1, \ldots, \tilde{\beta}_n) \cdot (1 - \pi_i(\tilde{\alpha}, \tilde{\beta}_1, \ldots, \tilde{\beta}_n)) \cdot (2y_i - 1) \\ &= \sum_{i=1}^{m} (1 - \pi_i(\tilde{\alpha}, \tilde{\beta}_1, \ldots, \tilde{\beta}_n)) \cdot (2y_i - 1) \end{aligned}$$

Für die partielle Ableitung nach β_k erhält man analog:

$$\frac{\partial L_{log}}{\partial \beta_k} = \sum_{i=1}^{m} x_{i,k} \cdot (1 - \pi_i(\tilde{\alpha}, \tilde{\beta}_1, \ldots, \tilde{\beta}_n)) \cdot (2y_i - 1)$$

Betrachten wir die Summanden der partiellen Ableitungen nun getrennt für die beiden möglichen Werten von y_i. Ist $y_i = 0$ dann gilt:

$$\begin{aligned} (1 - \pi_i(\tilde{\alpha}, \tilde{\beta}_1, \ldots, \tilde{\beta}_n)) \cdot (2y_i - 1) &= (1 - \pi_i(-\alpha, -\beta_1, \ldots, -\beta_n)) \cdot (-1) \\ &= -1 + (1 - \pi_i(\alpha, \beta_1, \ldots, \beta_n)) \\ &= -\pi_i(\alpha, \beta_1, \ldots, \beta_n) \end{aligned}$$

Für $y_i = 1$ ergibt sich folgendes:

$$\begin{aligned} (1 - \pi_i(\tilde{\alpha}, \tilde{\beta}_1, \ldots, \tilde{\beta}_n)) \cdot (2y_i - 1) &= (1 - \pi_i(\alpha, \beta_1, \ldots, \beta_n)) \cdot (2 - 1) \\ &= 1 - \pi_i(\alpha, \beta_1, \ldots, \beta_n) \end{aligned}$$

Damit können wir die partiellen Ableitungen weiter vereinfachen:

$$\begin{aligned} \frac{\partial L_{log}}{\partial \alpha} &= \sum_{i=1}^{m} y_i - \pi_i(\alpha, \beta_1, \ldots, \beta_n) \\ \frac{\partial L_{log}}{\partial \beta_k} &= \sum_{i=1}^{m} x_{i,k} \cdot (y_i - \pi_i(\alpha, \beta_1, \ldots, \beta_n)) \end{aligned}$$

Diese Darstellung der partiellen Ableitungen erlaubt es uns später in SQL den Gradienten zu berechnen. Der Term innerhalb der Summe wird einfach für jeden Datenpunkt berechnet, danach wird die resultierende Spalte zusammen mit einer Gruppierung summiert.

3. Anwendung statistischer Methoden

In diesem Kapitel werden nun mehrere Programmiersprachen vorgestellt, die sich für Regressionsanalyse eignen. Im letzten Teilkapitel wird demonstriert, wie man solche Methoden mit vorhandener SQL-Syntax umsetzen und durchführen kann.

3.1. Beispieldaten

Um Regressionsanalyse auch praktisch betreiben zu können, arbeiten wir in dieser Arbeit mit einem Satz an Beispieldaten. Diese Daten wurden mit einem Python-Skript erstellt, welches im als Ganzes im Anhang zu finden ist. Dabei werden die einzelnen Merkmale eines Datensatzes mit Absicht so erstellt, dass eine Korrelation zwischen diesen bewusst erzeugt oder nicht erzeugt wird. Diese Beispieldaten liegen in Form einer csv-Datei vor, welche in jeder Sprache einfach eingelesen werden kann.

Wir betrachten hier fiktive Kunden von Amazon. Für jeden Kunden wissen wir das Alter, die Anzahl seiner Käufe, die Summe des ausgegebenen Geldes und ob der Kunde Amazon-Prime Mitglied ist oder nicht. Der ausgegebene Betrag wird in Cent angegeben, um mit ganzen Zahlen rechnen zu können. Die Prime-Mitgliedschaft wird mit einer 1 symbolisiert, während eine 0 das Gegenteil bedeutet.

Insgesamt wurden für diese Arbeit 100.000 solcher Datensätze erzeugt. In der folgenden Tabelle sind die ersten 10 Datensätze beispielhaft dargestellt.

age	**purchases**	**money**	**prime**
30	1	4421	0
30	11	23346	1
33	1	4010	0
31	19	52517	1
29	3	8046	0
28	12	25295	0
41	16	38236	1
23	3	7098	1
25	1	2707	0
38	20	50976	1

Wir definieren uns außerdem drei Fragestellungen, welche wir jeweils mit einer Art der in Kapitel 2 vorgestellten Regressionen beantworten werden:

1. Zuerst wollen wir wissen, ob das ausgegebene Geld mit der Anzahl der Käufe in linearem Zusammenhang steht. Diese Fragen können wir mit einfacher linearer Regression beantworten. $money$ ist hierbei die abhängige Variable und $purchases$ ist die unabhängige Variable.

2. Die zweite Frage ist ähnlich der ersten, nur wollen wir hier wissen, ob neben der Anzahl der Käufe auch das Alter des Kunden einen linearen Einfluss auf das ausgegebene Geld hat. Hier haben wir nun zwei unabhängige Variablen, nämhlich age und $purchases$. Die abhängige Variable bleibt $money$. Diese Frage beantworten wir also mit multipler linearer Regression.

3. Als letztes interessiert uns, ob eine Prime-Mitgliedschaft von der Summe des ausgegebenen Geldes zusammenhängt. $money$ ist also nun die unabhängige Variable, während $prime$ die abhängige Variable ist. Außerdem ist $prime$ eine binäre Variable. Deshalb nutzen wir hier also logistische Regression.

3.2. R-Projekt

Das R-Projekt oder einfach nur R ist eine Sprache für statistische Berechnungen und graphische Darstellung. Damit ist R wie geschaffen für Regressionsanalyse. Von allen hier behandelten Sprachen ist R damit auch die einfachste und direkteste für Regression.

3.2.1. Grundprinzip

In R sind einfache Datenstrukturen wie Vektoren, Matrizen und Listen als Datentypen vorhanden. Darauf aufbauend existieren sogenannten Dataframes. Ein Dataframe ist eine Liste von Vektoren der gleichen Länge und wird in R zur Repräsentation von Datentabellen verwendet. Die Vektoren der Liste entsprechen dann den Spalten der Tabelle. In einen solchen Dataframe importieren wir unsere Daten.

In R lassen sich außerdem sehr einfach sogenannte Modelle definieren, welche als Eingabe nur die Daten und eine Formel benötigen. Eine Formel ist von der Form $y \sim modell$ und enthält den funktionalen Zusammenhang zwischen der abhängigen und den unabhängigen Variablen.

Hat man ein Modell erstellt, so bietet R einfache Funktionen, um die Parameter für das gegebene Modell mittels Regressionsanalyse zu berechnen. Wir werden im Folgenden von den Funktionen lm (für "linear model") und glm für ("generalized linear model") Gebrauch machen.

3.2.2. Einfache lineare Regression

Betrachten wir also Frage Nummer 1 aus dem vorherigen Teilkapitel. Die Formel lautet dann einfach $money \sim purchases$. Man liest also die Daten aus der csv-Datei, erstellt das Modell mit der genannten Formel und berechnet die Parameter mit der Funktion lm. Insgesamt braucht man also nur drei Zeilen Code. Die print-Funktion dient hierbei nur zur Ausgabe des Ergebnisses.

```
data <- read.csv2("sample.csv", sep = ",", header = TRUE)
modell <- as.formula("money ~ purchases")
slr <- lm(modell, data = data)
print(slr)
```

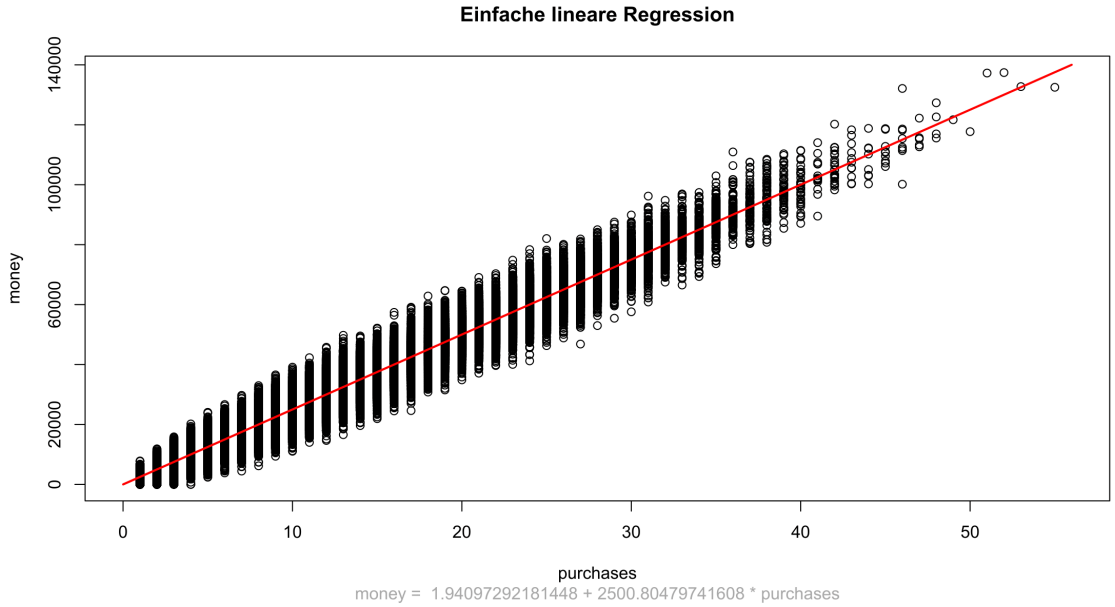
Das Ergebnis des obigen Codes ist folgendes:

```
Coefficients:
(Intercept)    purchases
      1.941     2500.805
```

Der Wert unter $(Intercept)$ entspricht dabei dem Paramter α in unserer Notation, der Wert unter $purchases$ entspricht β.

Wir wollen dieses Ergebnis kurz interpretieren. Der kleine Wert für α entspricht der Intuition, dass ein Kunde ohne Käufe auch kein Geld ausgegeben hat. Der relativ große Wert von ca. 2500 für β zeigt, dass die Anzahl der gekauften Artikel sehr einen großen Einfluss auf das ausgegebene Geld hat. Die Kunden geben pro gekauftem Artikel etwa 2500 Cent, also 25 Euro aus.

R verfügt auch über Möglichkeiten zur graphischen Darstellung. Lässt man die die Datenpunkte und die lineare Ausgleichsfunktion mit den berechneten Parameter plotten, erhält man dieses Ergebnis:



3.2.3. Multiple lineare Regression

Bei multipler lineare Regression unterscheidet sich der R-Code nur in der Wahl der Formel. Hier wollen wir $money$ durch eine lineare Summe von $purchases$ und age modellieren, deshalb lautet die Formel hier $money \sim purchases + age$. Man erhält das folgende Ergebnis.

```
Coefficients:
(Intercept)    purchases          age
   -16.4842    2500.8042       0.5318
```

Auch hier eine kurze Interpretation dieses Ergebnisses: Der Wert für α ist wieder relativ klein, der Wert für das β zu $purchases$ ist fast exakt derselbe wie bei einfacher linerarer Regression, was bei denselben Daten auch zu erwarten war. Der Wert für das β zu age ist dagegen nahe bei null. Das bedeutet, dass das Alter neben der Anzahl der Käufe keinen signifikanten Einfluss auf das ausgegebene Geld hat.

3.2.4. Logistische Regression

Bei logistischer Regression nutzen wir nun nicht mehr ein lineares Modell wie bisher, sondern ein generalisiertes lineares Modell. Logistische Regression ist im Wesentlichen ein Spezialfall dieses Modelles. Hier nutzen wir also die glm-Funktion. Um logistische Regression damit betreiben zu können, wählt man den Parameter $family$ dieser Funktion als $binomial$.

Man braucht wie auch bei linearer Regression ein Formel für das Modell. Diese bildet man analog wie bisher, indem mal die abhängige Variable mit den unabhängigen Variablen über eine Tilde verbindet. Im Fall der dritten Fragestellung aus Kapitel 3.1 wählen die also die Formel $prime \sim money$.

Der gesamte R-Code für die logistische Regression lautet also wiefolgt:

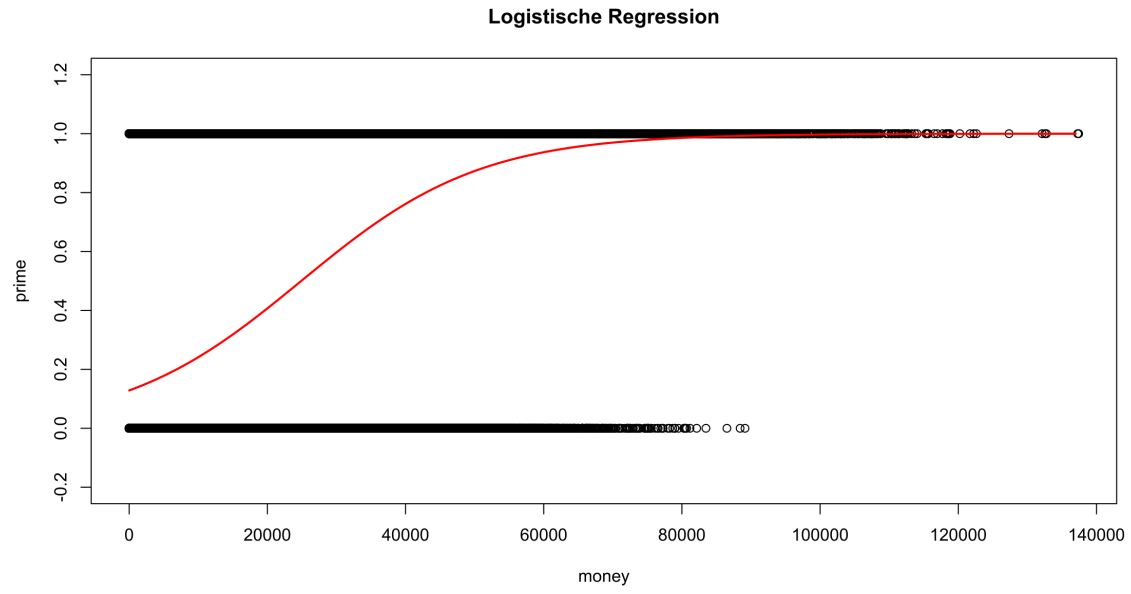
```
data <- read.csv2("sample.csv", sep = ",", header = TRUE)
modell <- as.formula("prime ~ money")
logit <- glm(modell, family = binomial, data = data)
print(logit)
```

Nach der Ausführung erhält man das folgende Ergebnis:

```
Coefficients:
(Intercept)        money
-1.9145608    0.0000769

Degrees of Freedom: 99999 Total (i.e. Null);  99998 Residual
Null Deviance:              138600
Residual Deviance: 100500     AIC: 100500
```

Eine anschauliche Interpretation der zurückgegebenen Parameter ist nicht mehr so einfach. Wir lassen uns das Ergebnis daher wieder als Plot visualisieren:



Für Kunden, die weniger als 100 Euro ausgegeben haben ist die Wahrscheinlichkeit Prime-Mitglied zu sein mit etwa 25% relativ gering. Je höher die Summe aber wird, desto größer wird auch diese Wahrscheinlichkeit. So ist ein Kunde mit mehr als 800 Euro Ausgaben so gut wie immer ein Prime-Mitglied.

3.3. TensorFlow

TensorFlow ist eine Software-Bibliothek, die unter der Haube von Google für die Umsetzung von Algorithmen für maschinelles Lernen entwickelt wurde. Das umfasst insbesondere auch die Möglichkeit zur iterativen Optimierung von Kostenfunktionen, was wir nun für Regressionsanalyse nutzen wollen.

TensorFlow bietet APIs für verschiedene Programmiersprachen an. Die Skripte, welche für dieser Arbeit erstellt wurden, sind in Python geschrieben. Im Unterschied zu den Ergebnissen der R-Skripte werden wir hier nicht dieselben Ergebnisse erhalten. Wir verwenden TensorFlows Implementierung eines Gradientenabstiegsverfahrens, für welches man die Anzahl der Schritte und die Abstiegsgeschwindigkeit selbst wählen muss. Die Genauigkeit der berechneten Parameter hängt also zusätzlich von einer angemessenen Auswahl dieser Werte ab.

Die Python-Skripte umfassen nun zwischen 70 und 100 Codezeilen, daher findet man diese Skripte im Ganzen nur im Anhang. Die wichsten Ausschnitte sollen im folgenden aber einen Einblick in die Funktioneweise des Codes geben.

3.3.1. Grundprinzip

TensorFlow arbeitet auf dem untersten Level mit Tensoren. Das sind im Wesentlichen Matrizen mit festen Dimensionen. Diese Tensoren können dann mit Hilfe aller möglicher Operatoren weiterverarbeitet werden.

Es gibt drei Möglichkeiten, Tensoren zu definieren: Als Konstante, als Variable oder als Platzhalter. Während Konstanten ihren Wert nicht mehr ändern können, sind die beiden letztgenannten veränderbar. Der Unterschied besteht darin, dass Variablen mit einem Startwert initiiert werden und Platzhalter ohne Wert. Wir werden Variablen nutzen, um die Parameter über die Iterationen zu speichern. Die Daten, mit denen wir das Modell trainieren werden, übergeben wir an Platzhalter.

Wie das Modell exakt definiert wird zeigen die folgenden Teilkapitel. Allgemein gesamt wird eine Kostenfunktion aufgestellt, die dann mit Hilfe eines Gradientenabstiegsverfahrens iterativ minimiert wird. Die Definition dieses sogenannten Trainingsschrittes sieht immer gleich aus:

```
train_step = tf.train
                .GradientDescentOptimizer(learn_rate)
                .minimize(cost)
```

Dabei ist tf die importierte TensorFlow-Bibliothek, $learn\_rate$ ist die Geschwindigkeit bzw. Schrittweite des Verfahrens und $cost$ ist die zuvor definierte Kostenfunktion.

Um dann auch wirklich Berechnungen durchführen zu können, muss in TensorFlow eine Session erzeugt werden. In dieser Session wird dann die Iteration gestartet, die

$train_step$ immer wieder mit echten Daten füttert. Je mehr Iterationen, desto exakter werden die Parameter.

3.3.2. Einfache lineare Regression

Hier definieren wir unsere Platzhalter und Variable wiefolgt:

```
x = tf.placeholder(tf.float32, [None, 1])
y = tf.placeholder(tf.float32, [None, 1])
alpha = tf.Variable(tf.zeros([1]))
beta = tf.Variable(tf.zeros([1, 1]))
```

x und y sind die Platzhalter der wahren Werte für das folgende Training. $alpha$ und $beta$ sind die Parameter-Variablen, welche mit null-Werten initiiert werden. Daraus berechnen wir über den linearen funktionalen Zusammenhang die geschätzten y Werte:

```
y_calc = tf.matmul(x, beta) + alpha
```

Die Funktion $matmul$ führt Matrizenmultiplikation durch. Nun definieren wir noch die Kostenfunktion als Mittelwert der Quadrate zwischen den wahren und berechneten y-Werten:

```
cost = tf.reduce_mean(tf.square(y - y_calc))
```

Damit können wir die Session starten und unsere Parameter berechnen lassen. Mit einer Schrittweite von 0.0054 und 2000 Iterationen erhält man folgendes Ergebnis:

```
alpha:  1.976688
beta:   2500.802979
cost:   13818006.000000
```

Die Werte für $alpha$ und $beta$ sind damit schon sehr nahe an den exakten Werten. Zusätzlich wird hier auch der aktuelle Wert für die Kostenfunktion ausgegeben.

3.3.3. Multiple lineare Regression

Nachdem in diesem Fall nun mehr unabhängige Variablen vorhanden sind, vergrößern wir die Größe der Tensoren x und $beta$ um eins.

```
x = tf.placeholder(tf.float32, [None, 2])
beta = tf.Variable(tf.zeros([2, 1]))
```

Die restlichen Variablen werden wie bei einfacher linearer Regression definiert. Das Gradientenverfahren ist bei zwei abhängigen Variablen ineffizienter und komplexer, daher muss die Schrittweite verringert werden. Eine Folge davon ist, dass man mehr Schritte für dieselbe Präzision des Ergebnisses durchführen muss. Bei einer Schrittweite von 0.00071 und 50000 Schritten erhält man folgendes Ergebnis:

```
alpha:              -16.035418
beta_purchases:     2500.799316
beta_age:           0.521137
cost:               13817990.000000
```

3.3.4. Logistische Regression

Wir definieren nun unsere Tensoren wieder exakt wie bei der einfachen linearen Regression, da wie hier wieder mit je einer unabhängigen und einer abhängigen Variablen arbeiten. Die bisher verwendete Berechnung der y-Werte fügen wir nun zusätzlich in die logistische Funktion ein:

```
y_calc = 1 / (1 + tf.exp(- tf.matmul(x, beta) - alpha))
```

Die Kostenfunktion ist nun nicht mehr die Summe der Quadrate sondern im Wesentlichen die Likelihoodfunktion. TensorFlow bietet nur eine API für Minimierung, während wir hier eine Funktion maximieren wollen. Deshalb verwenden wir das Inverse der Likelihoodfunktion als Kostenfunktion. Zusätzlich wenden wir wieder einen Logarithmus auf die Funktion an:

```
cost = - tf.reduce_sum(
  tf.log(
    y * y_calc +
    (1 - y) * (1 - y_calc)
  )
)
```

Wir wählen eine Schrittweite von 0.0001 und iterieren 1000 Schritte, um das folgende Ergebnis zu erhalten:

```
alpha:  -1.914557
beta:   0.000077
cost:   50272.593750
```

3.4. SQL

Die "Structured Query Language" alias "SQL" ist eine Sprache zur Definition und Verarbeitung von Datenstrukturen in Datenbanksystemen und wird in nahezu allen Implementierungen relationaler Datenbanken unterstützt. Of liegen die Daten, welche man für Regressionsanalyse verwenden möchte in einer solchen Datenbank.

Natürliche Methoden für Regression gehören nicht zum Portfolie von SQL, da Statistik und Datenanalyse nicht der primäre Einsatzzweck für SQL ist. Doch auch wenn jede Datenbank ihren eigenen SQL-Dialekt anbietet, ist es in nahezu allen Systemen möglch, mit standardisierten SQL-Methoden Regression direkt in der Datenbank zu betreiben.

In diesem Kapitel soll das nun für zwei der beliebtesten Open-Source-Datenbanksystemen umgesetzt werden, nämlich MySQL und PostgreSQL. Der vollständige SQL-Code befindet sich wegen der Länge wieder komplett im Anhang.

Im Gegensatz zu den beiden bisher vorgestellten Sprachen verfolgen wir hier kein Grundprinzip, in dem sich alle Regressionen ähnlich sind. Die einzige Gemeinsamkeit ist, dass wir in allen SQL-Skripten Prozeduren bzw. Funktionen definieren, welche bei Aufruf die Regressionsanalyse durchführen. Wir nehmen dazu an, dass die Daten für die Regression in einer Tabelle namens $sample$ liegen.

Bei einfacher linearer Regression berechnen wir die Parameter exakt über die Formeln aus Kapitel 2.1.1. Bei multipler Regression verwenden wir die Matrixformel aus Kapitel 2.1.2. Hier müssen wir zusätzlich Algorithmen zur Multiplikation und Invertierung von Matrizen implementieren. Für logistische implementieren wir dann ein Gradientenverfahren.

3.4.1. Einfache lineare Regression

Einfache lineare Regression stellt uns in SQL noch vor wenig Herausforderungen. Wir berechnen zuerst die Mittelwerte über die Spalten $purchases$ und $money$, dann die Summen in Zähler und Nenner der Formel für β und können dann mit einfachen Rechenoperationen die beiden Paramter bestimmen.

Diese Berechnung kann man sogar in einer einzelnen Abfrage umsetzen. Das Skript für PostgreSQL tut das auch und definiert die genannten Berechnungsschritte als einzelne Views. In MySQL existiert die VIEW-Syntax nicht. Deshalb wird die Berechnung der Übersicht halber auf mehrere Abfragen aufgeteilt.

Führen wir die Prozeduren im jeweiligen Datenbanksystem aus, erhalten wir folgende Ergebnisse:

Table 3.1.: Einfache lineare Regression in MySQL

variable	value
alpha	1.94097291465000744000
beta	2500.80479741553200000000

Table 3.2.: Einfache lineare Regression in PostgreSQL

variable	value
alpha	1.940972912129943338303950912446192000000000000
beta	2500.80479741577058414342859876

3.4.2. Multiple lineare Regression

Wir wollen zur Lösung dieses Regressionsproblems die Matrix-Formel aus Kapitel 2.1.2 anwenden. Dazu müssen Methoden für das Transponieren, Multiplizieren und Invertieren von Matrizen implementiert werden, da weder MySQL noch PostgreSQL über diese Funktionen verfügt. Dazu müssen wir außerdem einen Weg finden, Matrizen im jeweiligen Datenbanksystem zu repräsentieren.

Diese Repräsentation lösen wir unterschiedlich in den beiden Datenbanksystemen. Beginnen wir mit MySQL. Hier definieren wir uns temporäre Tabellen, welche jeweils eine Matrix repräsentieren. Jede solche Tabelle besteht aus drei Spalten, nämlich row, $column$ und $value$. Die ersten beiden enthalten die Indizes des Matrixelements, letztere enthält den Wert des jeweiligen Elements.

Wir definieren insgesamt sieben solcher Tabellen:

- $matrix\_X$: Entspricht der Matrix X aus der Berechnungsformel.
- $matrix\_y$: Entspricht der Matrix y aus der Berechnungsformel.

- $matrix\_transposed$: Entspricht der Matrix X^T.
- $matrix\_product\_1$: Entspricht dem Matrixprodukt X^tX.
- $matrix\_inverse$: Entspricht dem Inversen des obigen Matrixprodukts $(X^TX)^{-1}$.
- $matrix\_product\_2$: Entspricht dem Matrixprodukt X^Ty.
- $matrix\_result$: Entspricht dem Endergebnis der Berechnungsformel $(X^TX)^{-1}X^Ty$.

Die ersten beiden dieser Tabellen werden einfach mit den vorhandenen Werten aus der Tabelle $sample$ befüllt. Die Tabelle $matrix\_transposed$ wird mit einem einfachen Query aus der Tabelle $matrix_X$ berechnet, indem die Indizes für Zeile und Spalte vertauscht werden.

Die Tabellen $matrix\_product\_1$, $matrix\_product\_2$ und $matrix\_result$ sind Ergebnisse von Matrizenmultiplikationen. Diese Produkte werden mit zwei Schleifen berechnet, die über die Zeilen und Spalten der Ergebnismatrix iterieren. Jeden Element der zu berechnenden Matrix wird dann über eine Abfrage berechnet, welche die jeweilige Zeile und Spalte der beiden Faktor-Tabellen zusammenjoint und über das Ergebnis summiert. Die Abfrage fügt das Ergebnis der Summierung in die Zieltabelle ein.

Bei den Tabellen $matrix\_product\_2$ und $matrix\_result$ ist die eine Dimension der zu berechnenden Tabelle gleich eins ist. Die zweite Schleife besitzt also eine Iteration, weswegen diese in dem SQL-Skript auch einfach weggelassen wird.

Für die Berechnung der inversen Matrix, also für die Tabelle $matrix\_inverse$ wird ein einfacher iterativer Algorithmus verwendet, welcher über alle Zeilen der zu invertierenden Matrix läuft. In jedem Schritt werden alle Elemente der Matrix nach einem bestimmten Schema angepasst. Details zu dem verwendeten Algorithmus findet man in [1].

Die komplette Berechnung wurde in eine Prozedur verpackt. Führt man diese aus, erhält man das folgende Ergebnis:

Table 3.3.: Multiple lineare Regression in MySQL

variable	**value**
alpha	-16.48419722318438785193
beta_purchases	2500.80422198935324307395
beta_age	0.53177717855855479093

Betrachten wir nun die Implementierung in PostgreSQL. Gegenüber MySQL hat man hier den Vorteil, dass mehrdimensionale Listen als Datentyp existieren. Wir brauchen nun also keine temporären Tabellen für die Matrizen mehr, sondern speichern diese einfach als zweidimensionales Array vom Datentyp $NUMERIC$.

Neben der eigentlichen Prozedur zum Ausführen der Regression definieren wir drei weitere Funktionen:

- $matrix\_transpose$: Diese Funktion nimmt ein zweidimensionales Array von Typ $NUMERIC$ als Input und gibt die transponierte Matrix zurück. Die Transponierte wird mit zwei Schleifen über die Zeilen und Spalten gebildet, welche im Wesentlichen die Zeilen- und Spaltenindizes vertauschen.

- $matrix\_multiplication$: Diese Funktion nimmt zwei zweidimensionale Arrays vom Typ $NUMERIC$ als Input und gibt das Produkt der beiden Matrizen zurück. Hier werden drei Schleifen zur Berechnung verwendet. Die ersten beiden iterieren über die Zeilen und Spalten der Ausgabematrix. Die dritte iteriert über die Zeile bzw. Spalte, welche zur Berechnung der aktuellen Elements verwendet werden und addiert die Produkte der Elemente dieser Zeile und Spalte auf.
- $matrix\_inversion$: Diese Funktion nimmt ein zweidimensionales Array vom Typ $NUMERIC$ als Input und gibt das Inverse dieser Matrix zurück. Dabei wird derselbe Algorithmus verwendet, der auch in MySQL implementiert wurde.

Mit diesen drei Funktionen lässt sich die Formel aus Kapitel 2.1.2 mit drei Abfragen umsetzen. Dazu erzeugen wir zuerst zwei Matrizen x und y mit den unabhängigen und der abhängigen Variable als Elemente aus der $sample$-Tabelle. Im dritten Query nutzen wir die genannten Funktionen, um die Matrix mit den gesuchten Parametern zu berechnen.

Führt man die multiple lineare Regressionsanalyse in PostgreSQL durch, erhält man das folgende Ergebnis:

Table 3.4.: Multiple lineare Regression in PostgreSQL

variable	**value**
alpha	-16.484197223184387851 93
beta_purchases	2500.804221989353243073 95
beta_age	0.531777178558554790 93

Da wir dieselben Algorithmen und dieselbe Präzision für Kommazahlen in MySQL und PostgreSQL verwendet haben, stimmen die beiden Ergebnisse sogar exakt überein.

3.4.3. Logistische Regression

Um logistische Regression in SQL betreiben zu können, möchten wir ein Gradientenverfahren zur Lösung implementieren. Wir implementieren denselben Algorithmus in MySQL und PostgreSQL, das heißt die Skripte unterscheiden sich lediglich in der datenbankspezifischen Syntax.

Wir verwenden für das Verfahren mehrere Tabellen, in denen wir gewissen Informationen speichern und verarbeiten werden:

- Die Tabelle $datapoints$ besteht aus drei Spalten id, $variable$ und $value$. Die id-Spalte enthält einen hier generierten Wert, der einen Datensatz aus der $sample$-Tabelle identifiziert. Die $variable$-Spalte enthält den Namen der gespeicherten unabhängigen Variable. In unserem Fall steht hier immer $beta_money$, da wir nur eine unabhängige Variable betrachten. Gegebenenfalls können hier mehrere Variablen betrachtet werden. (Die Werte für einen Datenpunkt werden dann also auf mehrere Zeilen aufgeteilt.) Die Spalte $value$ enthält dann den Wert der entsprechenden Variable des jeweiligen Datensatzes.
- Die Tabelle $binary\_values$ besteht aus zwei Spalten id und $value$. Hier werden für jeden Datenpunkt die Werte der abhängigen binären Variable gespeichert, in unserem Fall also die Werte aus der Spalte $prime$.

- Die Tabelle $parameters$ besteht aus drei Spalten $variable$, old und new. Die erste Spalte enthält den Namen des Parameters, also $alpha$ oder $beta_money$. Die Spalten old und new enthalten die Werte der Parameter vor bzw. nach dem aktuellen Schritt im Gradientenverfahren. Wir benötigen beide Werte, um überprüfen zu können, ob die neuen Parameter ein besseres Ergebnis liefern als die alten.
- Die Tabelle $logits$ besteht aus drei Spalten id, old und new. Hier werden für jeden Datenpunkt die Werte der logistischen Funktion π_i berechnet. Dabei werden einmal die alten und einmal die neuen Parameter zur Berechnung verwendet.
- Die Tabelle $gradient$ besteht aus zwei Spalten $variable$ und $value$. Hier wird der Gradient bzw. die Werte der partiellen Ableitungen nach jedem Parameter gespeichert.

Wir definieren außerdem einige Hilfsfunktionen:

- $calculate\_logits$: Diese Funktion berechnet die Werte für die logistische Funktion für alle Datenpunkte und trägt diese in die Tabelle $logit$ ein. Dabei wird die Funktion für beide Parameterwerte (old und new) aus der Tabelle $parameters$ durchgeführt.
- $calculate\_gradient$: Diese Funktion berechnet den Gradienten aus Basis der alten Parameterwerte und der Werte der logistischen Funktion aus der Tabelle $logits$.
- $calculate\_parameters$: Diese Funktion nimmt eine Schrittweite als Argument, berechnet daraus und aus den Werten der $gradient$-Tabelle die neuen Parameter und schreibt diese zurück in die Spalte new der Tabelle $parameters$.
- $are\_new\_parameters\_better$: Diese Funktion berechnet den Logarithmus der Likelihoodfunktion für die neuen und alten Parameter aus der Tabelle $parameters$ und gibt einen Wahrheitswert zurück. Die Rückgabe ist wahr, wenn die Likelihoodfunktion für die neuen Parameter einen größeren Wert annimmt als für die alten Parameter.

Wir führen wie schon bei TensorFlow zuerst eine Lineartransformation für die Werte aus $money$ durch und bilden diese linear auf das Interval $[0, 1]$ ab. Das dient erneut dazu, dass die Werte der logistischen Funktion nicht zu nahe an null geraten. Die transformierten Werte fügen wir in die $datapoints$-Tabelle ein. Auch die anderen Tabellen erzeugen wir und fügen initiale Werte ein. Als Anfangswert für Schrittweite wählen wir eins.

Es folgt eine $while$-Schleife, die solange läuft, bis entweder die vorgegebene Anzahl an Schritten erreicht wurde, oder die Schrittweite zu klein für die gewählte Präzision der Kommazahlen wird. Wir berechnen zuerst den Gradienten, dann die neuen Parameter. Dann ist ein Aufruf von $calculate\_logits$ nötig, um die logistische Funktion für die neuen Parameter zu berechnen.

Wir überprüfen, ob die neuen Parameter wirklich besser sind als die alten. Falls nicht, wird die Schrittweite halbiert, die Parameter und die Werte der logistischen Funktion werden erneut berechnet. Das wird solange wiederholt, bis die neuen Parameter besser sind oder die Schrittweite unter die Präzisionsgrenze fällt.

Haben wir die neuen Parameter erfolgreich berechnet, werden die Werte der old-Spalten in den Tabellen $parameters$ und $logits$ mit den neuen Werten überschrieben und der Iterationsschritt ist beendet. Nachdem die Schleife beendet wurde, werden die Parame-

ter wieder entsprechend linear transformiert, um den tatsächlichen Werten für $money$ zu entsprechen.

Wir entscheiden und für 1000 Iterationen und führen die Prozeduren im jeweiligen Datenbanksystemen aus. Das Ergebnis ist folgendes:

Table 3.5.: Logistische Regression in MySQL

variable	**value**
alpha	-1.914557182274162638251164513436
beta_money	0.000076896120535027827177415497

Table 3.6.: Logistische Regression in PostgreSQL

variable	**value**
alpha	
beta_money	

4. Vergleich der verschiedenen Implementierungen

Wir haben im vorherigen Kapitel drei verschiedene Programmiersprachen bzw. Softwarebibliotheken vorgestellt und gezeigt, wie man dort Regressionsanalyse betreiben kann. In diesem Kapitel wollen wir zum Einen konkret auf die Unterschiede eingehen und zum anderen einen Benchmark bezüglich der Laufzeit abhängig von der Anzahl der zu verarbeitenden Datensätze erstellen.

5. Erweiterungspotenzial in Datenbanksystemen

Here starts the thesis with an introduction. Please use nice latex and bibtex entries [2]. Do not spend time on formating your thesis, but on its content.

5.1. Latex Introduction

There is no need for a latex introduction since there is plenty of literature out there.

6. Fazit

Here starts the thesis with an introduction. Please use nice latex and bibtex entries [2]. Do not spend time on formating your thesis, but on its content.

6.1. Latex Introduction

There is no need for a latex introduction since there is plenty of literature out there.

Appendix

A. Detailed Descriptions

Here come the details that are not supposed to be in the regular text.

Bibliography

[1] Khan Hamid Ahmad Farooq. An efficient and simple algorithm for matrix inversion. 2010.

[2] Leslie Lamport. *LaTeX : A Documentation Preparation System User's Guide and Reference Manual*. Addison-Wesley Professional, 1994.