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:	15.02.2017

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

München, den 12. Februar 2018 Thomas Heyenbrock

Zusammenfassung

Das Ziel der vorliegenden Arbeit ist es, die Durchführung von statistischen Methoden in relationalen Datenbanksystemen zu demonstrieren. Das statistische Konzept, dass dazu verwendet wird, ist die lineare und die logistische Regressionsanalyse. Dazu werden zuerst kurz die mathematischen Grundlagen erklärt. Danach wird die Umsetzung der Regressionsanalyse mit verschiedenen Programmiersprachen demonstriert, insbesondere in zwei relationalen Datenbanken. Diese Implementierungen werden daraufhin miteinander verglichen. Dabei stellt man fest, dass die lineare Regressionsanalyse bei kleinen Datenmengen in Datenbanken sehr gut funktioniert. Logistische Regression mit Gradientenverfahren kann dagegen nicht mehr mit anderen Implementierungen mithalten. Zum Abschluss wird ein mögliches Erweiterungspotenzial für relationale Datenbanksysteme erläutert, insbesondere eine Erweiterung um Matrix-Operationen.

Abstract

The goal of this thesis is to demonstrate the implementation of statistical methods in relational database systems. The statistical concepts used are linear and logistic regression analysis. First, the mathematical basics are explained briefly. Then the implementation of the regression analysis will be demonstrated with different programming languages, especially in two relational databases. These implementations are then compared. It is found that the linear regression analysis works very well for small amounts of data in relational databases. On the other hand, logistic regression with gradient descent can no longer compete with other implementations. Finally, an expansion potential for relational database systems is explained, in particular an extension to matrix operations.

Inhaltsverzeichnis

1. Einführung und typische statistische Problemstellungen

Die Statistik ist ein Teilgebiet der Mathematik, in welchem Methoden zum Umgang und zur Verarbeitung von Daten behandelt werden. Dabei wird oft ein vorhandener Satz an Daten, auch Stichprobe genannt, betrachtet und analysiert, um daraus Vorhersagen für die Gesamtheit aller Daten zu treffen. Den Teilbereich der Statistik, welcher sich mit solchen Problemen befasst, nennt man induktive oder schließende Statistik.

Betrachtet man beispielsweise die Körpergröße und das Gewicht von 100 Testpersonen, dann kann man sich fragen, ob diese beiden Merkmale in Zusammenhang stehen. Insbesondere ist interessant, wie man einen möglichen Zusammenhang quantitativ darstellen kann und ob man neben der Körpergröße auch andere Faktoren für das Gewicht einer Person betrachten sollte. Das sind beispielhafte Typen von Fragen, welchen man in der Statistik oft begegnet.

Die Statistik zeigt Methoden und Vorgehensweisen auf, wie man solche Fragen angehen und beantworten kann. Ein oft verwendetes Verfahren ist die Regression bzw. die Regressionsanalyse. Hier sucht und analysiert man Beziehungen zwischen mehreren Variablen und versucht diese quantitativ zu beschreiben.

Greifen wir das obige Beispiel wieder auf: Bei der Regression sucht man nach einer Formel, welche für gegebene Körpergröße das Gewicht einer Person möglichst gut schätzt. Oft trifft man Annahmen über die Art der Beziehung zwischen den Dimensionen, um die Suche a priori einzugrenzen. Man beschränkt sich in vielen Fällen auf lineare Funktionen, da solche leicht zu behandeln sind. In der Praxis sind aber auch allgemeine Potenzfunktionen, exponentielle Funktionen oder logistische Funktionen bzw. Beziehungen oft anzutreffen.

Es gibt speziell für Statistik entwickelte Software, seien es einfache Programmiersprachen wie das R-Projekt oder komplexere Programme mit grafischem Interface wie SPSS von IBM. Die Daten, welche man als Basis für Analysen verwendet, müssen aber an einer anderen Stelle gespeichert und verwaltet werden. Oft liegen diese in einer Datenbank und müssen zuerst in das Analysetool importiert werden.

Konzeptuell ist eine Datenbank nicht für Regressionsanalyse geschaffen. Dennoch ist es in relationalen Datenbanksystemen mit SQL möglich, Regression direkt in der Datenbank durchzuführen. Damit übergeht man den eben genannten Schritt des Importierens. Man kann außerdem die Ergebnisse der Analyse direkt aus der Datenbank abfragen oder dort weiterverwenden.

Diese Arbeit wird zuerst das Konzept der Regression konkreter einführen und die mathematischen Grundlagen darlegen. Darauf aufbauend betrachten wir Implementierungen für Regressionsanalyse in verschiedenen Programmiersprachen. Hier soll insbesondere die Anwendung von Regressionsanalyse mit Hilfe von SQL demonstriert werden. Danach wollen wir die Sprachen bezüglich der Laufzeit miteinander vergleichen und noch kurz auf das Erweiterungspotenzial für relationale Datenbanksysteme eingehen.

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_j nennt man unabhängig. Für diese Arbeit wollen wir zunächst einige Annahmen treffen. Diese sollen immer gelten, falls nicht explizit etwas anderes festgelegt wird.

- Die genannten Größen sind Zufallsvariablen. Eine solche Zufallsvariable ist eine Funktion, deren Werte die Ergebnisse eines Zufallsvorgangs darstellt.
- Die Zufallsvariablen sind auf der Menge $M = \{1, \ldots, m\}$ definiert und bilden in die reellen Zahlen ab:

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

  Die Zufallsvariablen sind also 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:

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

Nun definieren wir ein Modell, mit dem der Zusammenhang zwischen der abhängigen und den unabhängigen Variablen dargestellt werden soll. Dazu verwenden wir eine Funktion f, welche für Werte von X_1 bis X_n einen geschätzten Wert für Y liefert. Idealerweise existiert eine Funktion, die zum einen eine einfache Darstellung (etwa durch eine arithmetische Formel) besitzt und zum anderen alle unabhängigen Werte der Datenmenge exakt prognostiziert. Das bedeutet:

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

Im Allgemeinen ist es nicht möglich, eine solche Funktion zu finden. Man verwirft also die Anforderung der Exaktheit für alle Datenpunkte und versucht stattdessen eine einfache Funktion zu finden, die die Datenmenge möglichst gut approximiert. Wir definieren für jeden Datenpunkt den Fehler e_i, der sich durch die Ungenauigkeit der Modellfunktion f ergibt:

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

Je näher ein Fehlerterm bei null liegt, desto besser ist die Annäherung für den jeweiligen Datenpunkt. Um eine gute Approximation für die gesamte Datenmenge zu erhalten, sollten die Fehlerterme global betrachtet möglichst klein bleiben. Das Ziel der Regressionsanalyse ist nun die Bestimmung der Funktion f. Dazu nimmt man an, dass f eine bestimmte Form hat. Diese Annahme kann sich je nach der Problemstellung unterscheiden. Oft arbeitet man mit linearen Funktionen f, aber auch quadratische, exponentielle oder logistische Funktionen sind nicht unüblich.

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:

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

Dabei sind α und β_k für $k = 1, \ldots, n$ reelle Zahlen, die sogenannten Parameter der Funktion. Das Maß für die Qualität von f ist die Summe der quadrierten Fehlerterme. Diese Summe kann wiederum als Funktion in Abhängigkeit der Parameter definiert werden:

$$E\left(\alpha, \beta_1, \ldots, \beta_n\right) := \sum_{i=1}^{m} e_i^2 = \sum_{i=1}^{m} \left(y_i - f(x_{i,1}, \ldots, x_{i,n})\right)^2 = \sum_{i=1}^{m} \left(y_i - \alpha - \sum_{j=1}^{n} \beta_j \cdot x_{i,j}\right)^2$$

Bei der linearen Regression sucht man die Parameter $\hat{\alpha}$ und $\hat{\beta}_k$ für die E einen minimalen Wert annimmt:

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

Um dieses Minimierungsproblem zu lösen, berechnen wir die partiellen Ableitungen von E nach allen Parametern:

$$\begin{aligned}
\frac{\partial E}{\partial \alpha} &= \sum_{i=1}^{m} 2 \cdot \left(y_i - \alpha - \sum_{j=1}^{n} \beta_j \cdot x_{i,j}\right) \cdot (-1) \\
&= -2 \cdot \sum_{i=1}^{m} \left(y_i - \alpha - \sum_{j=1}^{n} \beta_j \cdot x_{i,j}\right) \\
\frac{\partial E}{\partial \beta_k} &= \sum_{i=1}^{m} 2 \cdot \left(y_i - \alpha - \sum_{j=1}^{n} \beta_j \cdot x_{i,j}\right) \cdot (-x_{i,k}) \\
&= -2 \cdot \sum_{i=1}^{m} x_{i,k} \cdot \left(y_i - \alpha - \sum_{j=1}^{n} \beta_j \cdot x_{i,j}\right)
\end{aligned}$$

Durch Nullsetzen der partiellen Ableitungen erhält man ein lineares Gleichungssystem mit $(n+1)$ Gleichungen und ebenso vielen 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 diese existiert) ist das gesuchte Minimum. Damit findet man die gesuchten Parameter für die lineare Funktion f.

2.1.1. Einfache lineare Regression

Man spricht von einfacher linearer Regression, wenn man mit nur einer unabhängigen Variable arbeitet. Anschaulich möchte man 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 $\beta := \beta_1$ und $x_i := x_{i,1}$. Dann können wir das lineare Gleichungssystem zum Auffinden des Minimums wie folgt aufschreiben:

$$\begin{aligned} 0 &= -2 \cdot \sum_{i=1}^{m} (y_i - \alpha - \beta \cdot x_i) \\ 0 &= -2 \cdot \sum_{i=1}^{m} x_i \cdot (y_i - \alpha - \beta \cdot x_i) \end{aligned}$$

Dieses Gleichungssystem kann explizit gelöst werden. Man erhält das folgende Ergebnis für die Lösung $\hat{\alpha}$ und $\hat{\beta}$:

$$\begin{aligned} \hat{\beta} &= \frac{\sum_{i=1}^{m} (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{m} (x_i - \bar{x})^2} \\ \hat{\alpha} &= \bar{y} - \hat{\beta}\bar{x} \end{aligned}$$

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

$$\bar{x} = \frac{1}{m} \sum_{i=1}^{m} x_i \quad , \quad \bar{y} = \frac{1}{m} \sum_{i=1}^{m} y_i$$

Eine Herleitung dieser Lösung findet sich in Kapitel 3.6.2 in [2].

2.1.2. Multiple lineare Regression

Bei multipler linearer Regression existieren mindestens zwei unabhängige Variablen. Anstatt wieder explizite Formeln für jeden einzelnen Parameter anzugeben, berechnen wir alle gesuchten Parameter gleichzeitig mit Hilfe von Matrizenrechnung. 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 den gesuchten Parametern für die Minimierung der kleinsten Quadrate bzw. der Funktion E. Falls die Matrix $X^T X$ invertierbar ist, gilt die folgende Formel für die Berechnung der gesuchten Parameter:

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

Mehr dazu findet man auch in Kapitel 12.2.3 in [2].

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 oder einen Gegenstand, den man entweder besitzt oder nicht, wie zum Beispiel das Geschlecht einer Person, ein Premium-Abonnement für einen Web-Service oder der Besitz eines Autos. Wir bezeichnen die beiden möglichen Werte einer solchen Variablen mit 0 und 1. Die Zuordnung vom Merkmal zur Zahl ist frei wählbar.

Lineare Regression eignet sich nicht zur Modellierung einer binären Variablen, da eine lineare Funktion entweder konstant oder unbeschränkt ist, in zweiten Fall also insbesondere Werte größer als 1 und kleiner als 0 annimmt. Um diesem Problem abzuhelfen, wählen wir zusätzlich zu der linearen Funktion eine weitere 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 β_1 bis β_n und konstantem Term α an. Zur Vereinfachung definieren wir für das restliche Kapitel die Variable c_i wie folgt:

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

Das Ergebnis der Funktion l für den i-ten Datensatz bezeichnen wir mit π_i. Dieses ist wieder eine Funktion in Abhängigkeit der Parameter:

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

Wir stellen hierbei fest, dass folgende Identität für die Funktionen π_i gilt:

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

Wir interpretieren π_i als die Wahrscheinlichkeit dafür, dass der Wert der abhängigen Variable eines Datensatzes mit Werten $x_{i,1}, \ldots, x_{i,n}$ der unabhängigen Variablen 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 α und β_k nun so schätzen, dass die Wahrscheinlichkeit für das Auftreten der vorhandenen Datenbasis maximiert wird. Diese Wahrscheinlichkeit ist gegeben durch:

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

Da alle y_i entweder gleich 0 oder gleich 1 sind, ist immer nur einer der beiden Summanden in obigem Produkt nicht null. Diese Fallunterscheidung kann man auch in das Vorzeichen der Parameter verschieben, da sich die beiden möglichen Faktoren nur darin unterscheiden. Es gilt also:

$$L(\alpha, \beta_1, \ldots, \beta_n) = \prod_{i=1}^{m} \pi_i\big((2y_i - 1)\alpha, (2y_i - 1)\beta_1, \ldots, (2y_i - 1)\beta_n\big)$$

Das Verfahren der Maximierung dieser Wahrscheinlichkeit bezeichnet man auch als Maximum-Likelihood-Methode. Die Funktion L nennt man Likelihoodfunktion. Oft maximiert man nicht L direkt, sondern eher den natürlichen Logarithmus von L:

$$\begin{aligned}
L_{log}(\alpha, \beta_1, \ldots, \beta_n) &:= \ln(L(\alpha, \beta_1, \ldots, \beta_n)) \\
&= \sum_{i=1}^{m} \ln\Big(\pi_i\big((2y_i - 1)\alpha, (2y_i - 1)\beta_1, \ldots, (2y_i - 1)\beta_n\big)\Big)
\end{aligned}$$

Der Sinn ist, dass man das Produkt damit in eine Summe einzelner Logarithmen umwandeln kann, welche wiederum einfacher abzuleiten ist. Die Maximierung von L ist äquivalent mit der von L_{log}, da der Logarithmus eine stetig wachsende Funktion ist. Die Werte von L liegen stets zwischen 0 und 1, also ist L_{log} wohldefiniert.

Um dieses Regressionsproblem zu lösen, muss man also die Parameter $\hat{\alpha}$ und $\hat{\beta}_k$ finden, für die gilt:

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

In diesem Fall kommt man nicht mehr an einer iterativen Lösung vorbei, da das lineare Gleichungssystem aus den partiellen Ableitungen nicht mehr exakt lösbar ist. Eine der einfachsten Methoden zur Lösung von Optimierungsproblemen ist das Gradientenverfahren, welches im kommenden Teilkapitel kurz eingeführt wird. Danach wird gezeigt, wie man das Gradientenverfahren für logistische Regression anwendet.

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 als Maximierungsalgorithmus ein. Man kann dasselbe Verfahren aber auch zur Lösung von Minimierungsproblemen einsetzen. Gegeben sei eine Funktion der folgenden Form, die maximiert werden soll:

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

Beim Gradientenverfahren beginnt man mit beliebigen Startwerten α^0 und β_k^0 und einer Schrittweite $s \in \mathbb{R}^+$. Vom Startpunkt aus geht man nun etwas in die Richtung des steilsten Anstieges der Funktion und erhält dadurch neue Werte α^1 und β_k^1. Diese Richtung ist der Gradient der Funktion f.

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

$$\nabla(f) = \begin{pmatrix} \partial f / \partial \alpha \\ \partial f / \partial \beta_1 \\ \vdots \\ \partial f / \partial \beta_n \end{pmatrix}$$

Der Gradient von f ist wieder eine Funktion, die Werte α und β_k auf einen Vektor der Länge $n + 1$ abbildet. Der iterative Schritt des Verfahrens definiert sich wie folgt:

$$\begin{pmatrix} \alpha^{i+1} \\ \beta_1^{i+1} \\ \vdots \\ \beta_n^{i+1} \end{pmatrix} = \begin{pmatrix} \alpha^i \\ \beta_1^i \\ \vdots \\ \beta_n^i \end{pmatrix} + s \cdot \nabla(f)(\alpha^i, \beta_1^i, \ldots, \beta_n^i)$$

Will man ein Minimierungsproblem lösen, muss man nur das Vorzeichen des Gradienten vertauschen, also ein Minus statt einem Plus in der obigen Formel verwenden. Man geht also entgegengesetzt der Richtung des steilsten Anstiegs und damit in die Richtung des steilsten Abstiegs von f.

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 verkleinert werden, zum Beispiel um einen festen, zuvor definierten Faktor.

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 berechnen 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. Dazu notieren wir die natürliche Exponentialfunktion als exp. 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 nach 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 konstante Faktor $x_{i,k}$ übrig.

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

Bevor wir L_{log} ableiten, definieren wir Hilfsvariablen $\tilde{\alpha} := (2y_i-1)\alpha$ und $\tilde{\beta}_k := (2y_i-1)\beta_k$ für $k=1,\ldots,n$. Damit erhält man:

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

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:

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

Wir betrachten nun die einzelnen Summanden der partiellen Ableitungen 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:

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

Damit können wir die partiellen Ableitungen weiter vereinfachen:

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

Diese Darstellung der partiellen Ableitungen verwenden wir später in SQL zur Berechnung des Gradienten. Der Term innerhalb der Summe wird für jedes Tupel der Relation der Datenpunkte berechnet. Danach wird das resultierende Attribut mit einer Gruppierung summiert.

3. Anwendung statistischer Methoden

In diesem Kapitel werden mehrere Programmiersprachen vorgestellt, die sich für Regressionsanalyse eignen. Dabei betrachten wir für Statistik entwickelte Programmiersprachen ebenso wie Softwarebibliotheken für maschinelles Lernen. Im letzten Teil dieses Kapitels wird demonstriert, wie man Regression mit vorhandener SQL-Syntax umsetzen und durchführen kann.

3.1. Beispieldaten

Wir arbeiten in dieser Arbeit mit Beispieldaten, welche mit einem Python-Skript erstellt wurden. Das Skript ist als Ganzes im Anhang A zu finden. Die Daten liegen in Form einer csv-Datei vor, welche in so gut wie jeder Sprache einfach eingelesen werden kann. Außerdem wird der Datensatz auch als INSERT-Abfrage in eine sql-Datei geschrieben, mit der die Daten in einem Datenbanksystem in eine Relation geschrieben werden können.

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

Insgesamt wurden für diese Arbeit 100000 solcher Datensätze erzeugt. Die später dargestellten Ergebnisse wurden mit den ersten 1000 Datenpunkten berechnet. In der folgenden Tabelle sind die ersten 10 dieser Datensätze dargestellt.

Tabelle 3.1.: Auszug aus den Beispieldaten

age	**purchases**	**money**	**premium**
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 wählen außerdem drei Fragestellungen, welche wir mit den in Kapitel 2 vorgestellten Arten von Regressionen beantworten werden:

1. Als Erstes wollen wir wissen, ob das ausgegebene Geld mit der Anzahl der Käufe in linearem Zusammenhang steht. Diese Frage 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. Hier wollen wir wissen, ob neben der Anzahl der Käufe auch das Alter des Kunden einen Einfluss auf das ausgegebene Geld hat. Hier haben wir nun zwei unabhängige Variablen, nämlich *age* und *purchases*. Die abhängige Variable bleibt *money*. Diese Frage beantworten wir mit multipler linearer Regression.

3. Zuletzt interessiert uns, ob eine Premium-Mitgliedschaft mit der Summe des ausgegebenen Geldes zusammenhängt. *money* ist also nun die unabhängige Variable und *premium* ist die abhängige Variable. Nachdem *premium* eine binäre Variable ist, nutzen wir hier logistische Regression.

3.2. R-Projekt

Das R-Projekt oder einfach nur R ist eine Sprache für statistische Berechnungen und auch grafische Darstellung. Damit ist R wie geschaffen für Regressionsanalyse. Die für diese Arbeit erstellten R-Skripte sind im Anhang B.

3.2.1. Grundprinzip

In R sind Datenstrukturen wie Vektoren, Matrizen und Listen als Datentypen vorhanden. Darauf aufbauend existieren sogenannte Dataframes. Ein Dataframe ist eine Liste von Vektoren gleicher Länge und wird in R zur Repräsentation von Datentabellen verwendet. Die Vektoren der Liste entsprechen den Attributen einer Relation. Wir importieren zuerst immer die Beispieldaten aus der csv-Datei in einen solchen Dataframe.

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

Hat man ein Modell erstellt, so bietet R Funktionen, um die Parameter für das definierte Modell mittels Regressionsanalyse zu berechnen. Wir werden im Folgenden von den Funktionen *lm* (für "linear model") und *glm* (für "generalized linear model") Gebrauch machen. Die Implementierungen der beiden Funktionen wurden an die gleichnamigen Funktionen in der verwandten Programmiersprache S angelehnt. Mehr dazu findet man in [3].

3.2.2. Einfache lineare Regression

Betrachten wir also die erste Frage aus dem vorherigen Teilkapitel. Die Formel lautet hier "$money \sim purchases$". Man liest also die Daten aus der csv-Datei, erstellt das Modell mit der genannten Formel und berechnet die Parameter mit der Funktion *lm*. Mit nur drei

Zeilen Code lässt sich also die komplette Regressionsanalyse implementieren. Die *print*-Funktion dient zur Ausgabe des Ergebnisses.

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

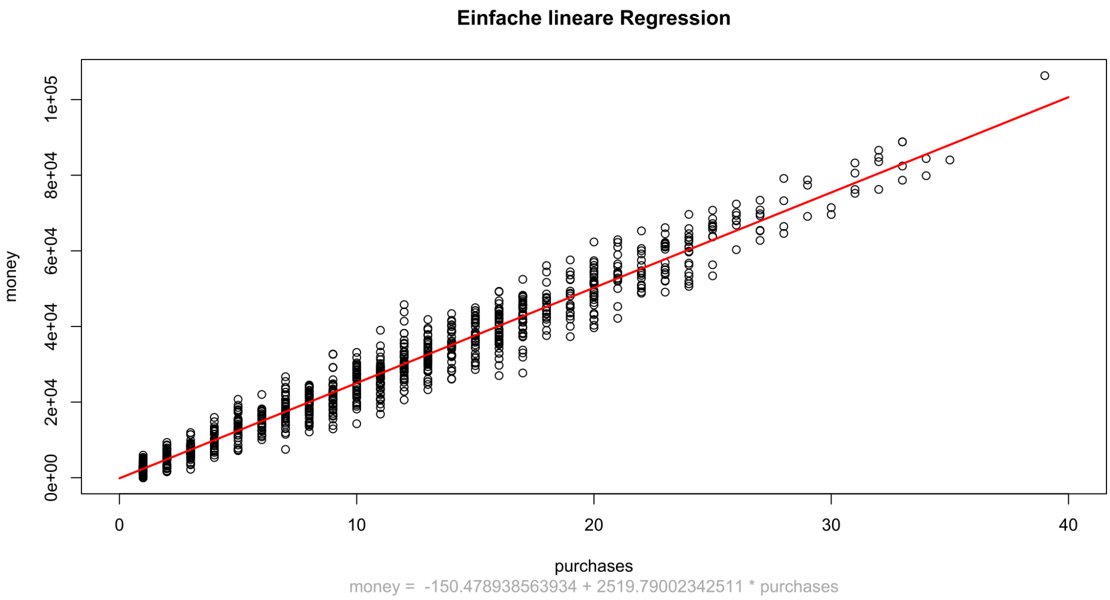
Das Ergebnis des obigen Codes ist Folgendes:

```
Coefficients:
(Intercept)    purchases
     -150.5       2519.8
```

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

Der relativ kleine Wert für α entspricht der Intuition, dass ein Kunde ohne Käufe auch kein Geld ausgegeben hat. Der relativ große Wert von ca. 2520 für β zeigt, dass die Anzahl der gekauften Artikel sehr einen großen Einfluss auf das ausgegebene Geld hat. Ein Kunde gibt pro gekauftem Artikel etwa 2520 Cent, also 25.20 Euro aus.

R verfügt auch über Möglichkeiten zur grafischen Darstellung. Lässt man die Datenpunkte und die lineare Ausgleichsfunktion mit den berechneten Parametern plotten, erhält man dieses Diagramm:



3.2.3. Multiple lineare Regression

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

```
Coefficients:
(Intercept)    purchases         age
     -766.7       2520.3        17.5
```

Der Wert für das β zu *purchases* ist fast exakt derselbe wie bei einfacher linearer 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 im Vergleich zu der Anzahl der Käufe keinen signifikanten Einfluss auf das ausgegebene Geld hat.

3.2.4. Logistische Regression

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

Man braucht wie bei linearer Regression eine 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 wählt man die Formel als "$premium \sim money$".

Der gesamte R-Code für die logistische Regression ist wieder ähnlich zu dem Code aus 3.2.2 und lautet wie folgt:

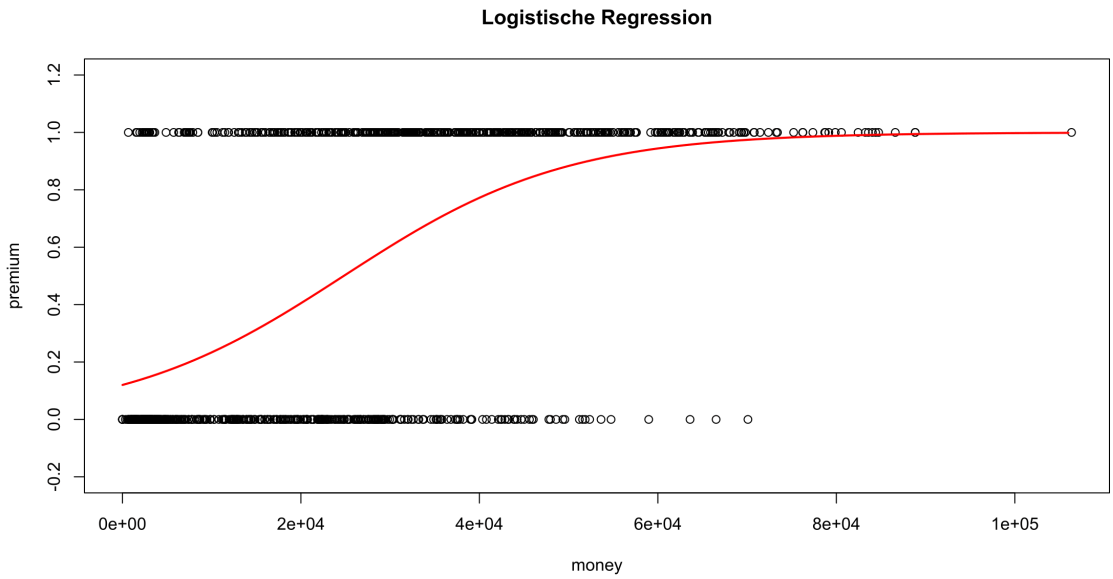
```r
data <- read.csv2("sample.csv", sep = ",", header = TRUE)
modell <- as.formula("premium ~ money")
logit <- glm(modell, family = binomial, data = data)
print(logit)
```

Nach der Ausführung erhält man das folgende Ergebnis:

```
Coefficients:
(Intercept)        money
 -1.9910911    0.0000803

Degrees of Freedom: 999 Total (i.e. Null);  998 Residual
Null Deviance:             1386
Residual Deviance: 1006          AIC: 1010
```

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 Premium-Mitglied zu sein mit etwa 25% relativ gering. Je höher der Wert für *money* aber wird, desto größer wird auch die genannte Wahrscheinlichkeit. So ist ein Kunde mit mehr als 800 Euro Ausgaben so gut wie immer ein Premium-Mitglied.

3.3. TensorFlow

TensorFlow ist eine Softwarebibliothek, die 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 zur Regressionsanalyse nutzen wollen.

TensorFlow bietet APIs für verschiedene Programmiersprachen an. Die Skripte, welche für diese Arbeit erstellt wurden, sind in Python geschrieben. Wir verwenden TensorFlows Implementierung eines Gradientenabstiegsverfahrens, für welches man die Anzahl der Schritte und die Abstiegsgeschwindigkeit selbst wählen muss. Die Genauigkeit des Ergebnisses hängt folglich von einer angemessenen Auswahl dieser Werte ab.

Die Python-Skripte umfassen nun zwischen 70 und 100 Codezeilen, daher findet man diese Skripte als Ganzes nur im Anhang C. Die wichtigsten Ausschnitte aus dem Code sollen in den folgenden Teilkapiteln aber einen Einblick in die Funktionsweise des Codes geben.

3.3.1. Grundprinzip

TensorFlow arbeitet mit Tensoren als grundliegende Datenstruktur. Solche sind im Wesentlichen mehrdimensionale Matrizen mit festen Dimensionen. Tensoren können mit Hilfe verschiedenster Operatoren weiterverarbeitet werden.

Es gibt drei Möglichkeiten, Tensoren zu definieren, nämlich als Konstante, als Variable oder als Platzhalter. Während Konstanten ihren Wert nach der Initialisierung nicht mehr ändern können, sind die Werte der beiden Letztgenannten veränderbar. Der Unterschied besteht darin, dass Variablen mit einem Startwert initiiert werden, Platzhalter besitzen dagegen anfangs keinen 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. Wir stellen immer eine Kostenfunktion auf, 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 wirklich Berechnungen durchführen zu können, muss in TensorFlow eine Session erzeugt werden. In dieser Session wird dann eine Schleife gestartet, die *train_step* immer wieder mit der Datenbasis füttert, um den aktuellen Wert der Parameter zu verbessern. Je mehr Iterationen durchlaufen werden, desto exakter werden die Parameter approximiert.

3.3.2. Einfache lineare Regression

Hier definieren wir unsere Platzhalter und Variablen wie folgt:

```
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 für die Datensätze für die später folgende Optimierung. *alpha* und *beta* sind die Variablen für die Parameter, welche mit Werten null initiiert werden. Daraus berechnen wir über den linearen funktionalen Zusammenhang die geschätzten y Werte:

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

Die Funktion *tf.matmul* führt Matrizenmultiplikation durch. *y_calc* entspricht π_i aus dem Grundlagenkapitel 2.2. Nun definieren wir die Kostenfunktion als Mittelwert der Quadrate zwischen den wahren und geschätzten 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:  -150.377670
beta:   2519.784424
cost:   15031988.000000
```

Die hier berechneten Werte für *alpha* und *beta* sind schon sehr nahe an den exakten Werten. Zusätzlich geben wir hier auch den aktuellen Wert der Kostenfunktion aus.

3.3.3. Multiple lineare Regression

In diesem Fall sind nun mehr unabhängige Variablen vorhanden, daher vergrößern wir die Dimension 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 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:           -754.910095
beta_purchases:  2520.172363
beta_age:        17.224550
cost:            15004541.000000
```

3.3.4. Logistische Regression

Wir definieren unsere Tensoren exakt wie bei einfacher linearer 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))
```

Bei der verwendeten Exponentialfunktion besteht die Gefahr, dass der Wert so nahe an null gerät, dass Python auf exakt null rundet und damit weiterrechnet. Um dieses Problem auszuschließen, werden die Werte der unabhängigen Variable zuvor linear in das Intervall $[0, 1]$ transformiert. Die am Ende berechneten Parameter werden entsprechend linear zurücktransformiert, um den ursprünglichen Daten zu entsprechen.

Hier wollen wir die Likelihoodfunktion maximieren. TensorFlow bietet allerdings nur eine API für Minimierung, weshalb wir das Inverse der Likelihoodfunktion als Kostenfunktion verwenden. Zusätzlich wenden wir wieder den natürlichen Logarithmus auf die Funktion an. Die Kostenfunktion sieht also wie folgt aus:

```
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:  -1.991089
beta:   0.000080
cost:   502.972290
```

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 Datenbanksysteme unterstützt. Oft liegen die Daten, welche man für Regressionsanalyse verwenden möchte in einem solchen Datenbanksystem.

SQL als Programmiersprache ist turingvollständig. Es ist also mit standardisierten SQL-Methoden möglich, Regression direkt in der Datenbank zu betreiben. Die konkrete Umsetzung hängt von der Art der Regression und dem spezifischen Datenbanksystem ab. In diesem Kapitel soll das nun in zwei Open-Source-Datenbanksystemen demonstriert werden, nämlich MySQL und PostgreSQL. Der vollständige SQL-Code befindet sich wegen der Länge wieder komplett im Anhang. Die MySQL-Skripte sind unter D zu finden, die Skripte für PostgreSQL liegen in E.

Im Gegensatz zu den beiden bisher vorgestellten Sprachen verfolgen wir hier kein einheitliches Grundprinzip, in dem sich alle Implementierungen 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 Relation *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 Transponierung, Multiplikation und Invertierung von Matrizen implementieren. Für logistische Regression steht uns keine explizite Formel zur Verfügung, weshalb wir ein Gradientenverfahren zur Maximierung der Likelihoodfunktion implementieren.

3.4.1. Einfache lineare Regression

Einfache lineare Regression kann ohne größeren Aufwand mit SQL umgesetzt werden. Wir berechnen zuerst die Mittelwerte über die Spalten *purchases* und *money*, dann die Summen in Zähler und Nenner der Formel für β. Daraus können wir mit einfacher Arithmetik die beiden Parameter bestimmen.

Diese Berechnung kann man leicht 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:

Tabelle 3.2.: Einfache lineare Regression in MySQL

variable	**value**
alpha	-150.478938563892081700000000000000
beta	2519.790023425114700000000000000000

Tabelle 3.3.: Einfache lineare Regression in PostgreSQL

variable	**value**
alpha	-150.47893856381151146861740350360640000000000000
beta	2519.7900234251071069143923667424

3.4.2. Multiple lineare Regression

Wir wollen zur Lösung dieses Regressionsproblems die Matrixformel 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 solche Funktionen verfügen. Wir müssen außerdem einen Weg finden, Matrizen im jeweiligen Datenbanksystem zu repräsentieren.

Die Matrizenrepräsentierung lösen wir unterschiedlich in den beiden Datenbanksystemen. In MySQL definieren wir uns temporäre Relationen, welche jeweils eine Matrix repräsentieren. Jede solche Relation besitzt dasselbe Schema und besteht aus den Attributen *row*, *column* und *value*. Die Ersten beiden enthalten die Indizes des Matrixelements, Letzteres enthält den Wert des jeweiligen Matrixelements.

Wir definieren insgesamt sieben solcher Relationen:

Tabelle 3.4.: Relationen für multiple lineare Regression in MySQL

Schema	**entspricht folgender Matrix** (vergleiche Berechnungsformel)
$matrix\_X([row, column, value])$	X
$matrix\_y([row, column, value])$	y
$matrix\_transposed([row, column, value])$	X^T
$matrix\_product\_1([row, column, value])$	$X^T X$
$matrix\_inverse([row, column, value])$	$(X^T X)^{-1}$
$matrix\_product\_2([row, column, value])$	$X^T y$
$matrix\_results([row, column, value])$	$(X^T X)^{-1} X^T y$

Die beiden erstgenannten Relationen werden mit den vorhandenen Werten aus der Relation *sample* befüllt. Die Relation *matrix_transposed* wird mit einer einfachen Abfrage über der Relation *matrix_X* berechnet, welche die Indizes für Zeile und Spalte vertauscht.

Die Relationen *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, also über die Tupel der Ergebnisrelation. Der jeweilige Wert zusammen mit dem Zeilen- und Spaltenindex der zu berechnenden Matrix wird über eine Abfrage eingefügt. Diese Abfrage nimmt das Kreuzprodukt beider zu multiplizierenden Relationen und betrachtet die entsprechende Zeile der ersten Relation und die entsprechende Spalte der zweiten Relation. Außerdem sollen das Attribut *row* der ersten Relation gleich dem Attribut *colum* der zweiten Relation sein. Der Wert für die zu berechnende Matrix ergibt sich als Summe über die Produkte der Werte der verbleibenden Tupel.

Für die Berechnung der inversen Matrix, also für die Relation *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 so angepasst, sodass man am Ende das Inverse erhält. 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:

Tabelle 3.5.: Multiple lineare Regression in MySQL

variable	**value**
alpha	-766.736173784421688472047506524797
beta_purchases	2520.301741791306785710069856997055
beta_age	17.503722143360901662105040790569

Betrachten wir nun die Implementierung in PostgreSQL. Gegenüber MySQL hat man hier den Vorteil, dass mehrdimensionale Felder als Datentyp existieren. Wir verwenden also keine temporären Relationen für die Matrizen mehr, sondern speichern diese als zweidimensionales Array.

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

Tabelle 3.6.: Funktionen für multiple lineare Regression in PostgreSQL

Name	**Eingabe**	**Ausgabe**	**Beschreibung**
matrix_transpose	*a numeric* $(65, 30)[\,][\,]$	*numeric* $(65, 30)[\,][\,]$	transponiert die Matrix a
matrix_multiplication	*a numeric*, $(65, 30)[\,][\,]$, *b numeric* $(65, 30)[\,][\,]$	*numeric* $(65, 30)[\,][\,]$	multipliziert die Matrizen a und b
matrix_inversion	*a numeric* $(65, 30)[\,][\,]$	*numeric* $(65, 30)[\,][\,]$	invertiert die Matrix a

Mit diesen drei Funktionen lässt sich die Formel aus Kapitel 2.1.2 einfach umsetzen. Mit zwei auf der *sample*-Relation erzeugten Matrizen x und y 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:

Tabelle 3.7.: Multiple lineare Regression in PostgreSQL

variable	**value**
alpha	-766.736173784421688472047506524797
beta_purchases	2520.301741791306785710069856997055
beta_age	17.503722143360901662105040790569

3.4.3. Logistische Regression

Zur Lösung des logistischen Regressionsproblems möchten wir in SQL ein Gradientenverfahren implementieren. Wir verwenden denselben Algorithmus in MySQL und PostgreSQL. Die Skripte unterscheiden sich lediglich in der datenbankspezifischen Syntax.

Wir verwenden für das Verfahren mehrere Relationen, in denen gewisse Informationen gespeichert und verarbeiten werden. Dabei weisen wir jedem Datenpunkt eine *id* zu, um verschiedene Relationen sinnvoll verknüpfen zu können.

Tabelle 3.8.: Tabellen für logistische Regression

Schema	**Beschreibung**
$datapoints([id, variable, value])$	Ein Tupel enthält den Wert einer bestimmten unabhängigen Variable für einen bestimmten Datenpunkt.
$binary\_values([id, value])$	Ein Tupel enthält den Wert für die abhängige Variable eines bestimmten Datenpunktes.
$parameters([variable, old, new])$	Ein Tupel enthält den alten und den neuen Wert des Parameters einer bestimmten Variable.
$logits([id, old, new])$	Ein Tupel enthält den Wert der logistischen Funktion (also π_i) berechnet mit den alten oder neuen Parameterwerten für einen bestimmten Datenpunkt.
$gradient([variable, value])$	Ein Tupel enthält den Wert der partiellen Ableitung nach einer bestimmten Variable.

Wir definieren außerdem einige Hilfsfunktionen. Diese Funktionen besitzen keinen Rückgabewert. Stattdessen bearbeiten sie die oben genannten Tabellen.

Tabelle 3.9.: Funktionen für logistische Regression

Name	Eingabe	Beschreibung
calculate_logits		berechnet den Wert der logistischen Funktion bzw. π_i mit den alten und neuen Parameterwerten für jeden Datenpunkt
calculate_gradient		berechnet den Wert der partiellen Ableitungen nach allen Variablen
calculate_new_parameters	*step* $numeric(65, 30)$	berechnet die neuen Parameterwerte aus dem Gradienten und der übergebenen Schrittweite

Wir führen wie schon bei TensorFlow zuerst eine Lineartransformation für die Werte aus *money* durch und bilden diese linear auf das Intervall $[0, 1]$ ab. Das dient erneut dazu, dass die Werte der Exponentialfunktion nicht zu nahe an null geraten. Die transformierten Werte fügen wir in die Relation *datapoints* ein. Danach werden die anderen Relationen erzeugen und initiale Werte eingefügt.

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, danach werden die Parameter und die Werte der logistischen Funktion 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*-Attribute in den Relationen *parameters* und *logits* mit den neuen Werten überschrieben und der Iterationsschritt ist beendet. Nachdem die Schleife beendet wurde, werden die Parameter wieder entsprechend linear transformiert, um den tatsächlichen Werten für *money* zu entsprechen.

Wir entscheiden uns für 1000 Iterationen und wählen eine Schrittweite von 0.008. Führt man die Prozeduren im jeweiligen Datenbanksystem aus, erhält man das folgende Ergebnis:

Tabelle 3.10.: Logistische Regression in MySQL

variable	value
alpha	-1.991090115311453143480846099933
beta_money	0.000080298536993602280846099933

Tabelle 3.11.: Logistische Regression in PostgreSQL

variable	**value**
beta_money	0.000080298565326972737807102042
alpha	-1.991090799730035106155294219916

4. Vergleich der verschiedenen Implementierungen

Nun wollen wir die implementierten Algorithmen im Bezug auf ihre Laufzeit miteinander vergleichen. Die Berechnung der Laufzeiten erfolgt erneut mit einem Python-Skript. Dieses Skript führt die jeweilige Regression über einen Kommandozeilenbefehl aus und misst die Zeit dieser Operation. Die Anzahl der Wiederholungen und die Anzahl der verwendeten Datenpunkte können als Parameter übergeben werden. Das Skript findet man im Anhang unter F.1.

Die Laufzeiten der Berechnungen werden als csv-Datei gespeichert. Pro Berechnung wird eine Zeile in die Ergebnisdatei geschrieben. Diese Zeile enthält die Programmiersprache, die Art der Regression, die Anzahl der verwendeten Datenpunkte und die Dauer der Berechnung.

Für jede Kombination der ersten drei Werte wurden 100 Berechnungen durchgeführt, falls die Dauer einer Berechnung höchstens 100 Sekunden betrug. Für Berechnungen, die länger als 100 Sekunden dauerten, wurde die Anzahl auf 50 Berechnungen reduziert. Skripte mit einer Laufzeit von länger als 1000 Sekunden wurden nur zehn mal ausgeführt. Für das Benchmarking wurde ein MacBook Pro (Mitte 2012, Betriebssystem macOS High Sierra 10.13.2) mit einem 2,9 GHz Intel Core i7 Prozessor und 8 GB 1600 MHz DDR3 Arbeitsspeicher verwendet.

Auch für das Auswerten der berechneten Benchmarks wurde ein Python-Skript erstellt, welches im Anhang F.2 abgedruckt ist. Dieses Skript liest die csv-Datei mit Benchmarks ein und gibt eine Tabelle mit den durchschnittlichen Laufzeiten pro Art der Regression, Programmiersprache und Anzahl verwendeter Datenpunkte aus. Außerdem wird pro Typ der Regression ein Plot zum Vergleich der Laufzeiten erzeugt.

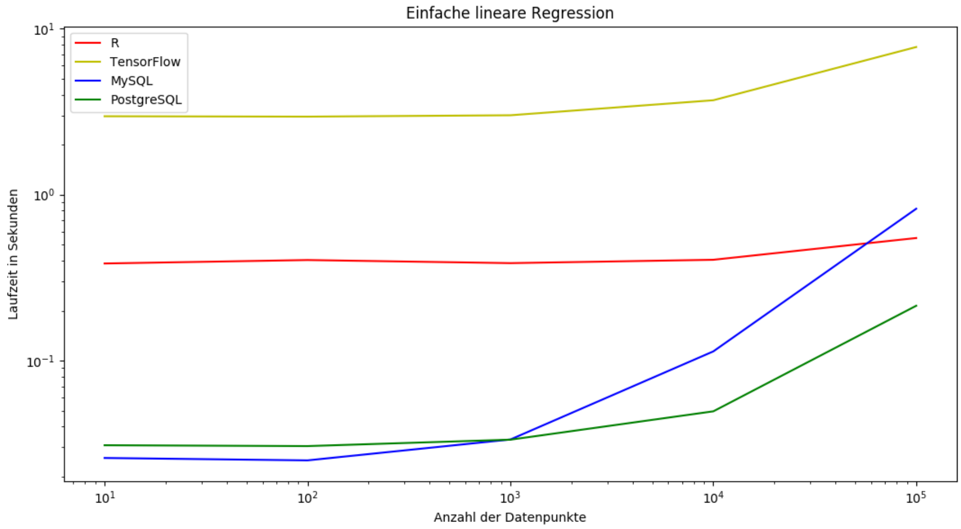
4.1. Einfache lineare Regression

Folgende Ergebnisse erhalten wir bei einfacher linearer Regression:

Tabelle 4.1.: Laufzeiten in Sekunden für einfache lineare Regression

	10	**100**	**1000**	**10000**	**100000**
r	0.38453477	0.40336191	0.38638819	0.40475887	0.54730985
tensorflow	2.96190362	2.94665535	3.00290149	3.69807061	7.73616447
mysql	0.02591379	0.02506811	0.03348756	0.11362195	0.82149320
postgresql	0.03092405	0.03056195	0.03340294	0.04948901	0.21394655

Der zugehörige Graph sieht folgendermaßen aus:



TensorFlow und R haben eine relativ konstante Lautzeit, auch bei größeren Datenmengen. Dabei ist TensorFlow mit iterativer Berechnung wie erwartet mit Abstand am langsamsten. Die SQL-Implementierungen sind bei geringer Anzahl an Datenpunkten sogar die schnellsten Skripte. Für größer werdende Datenmengen erkennt man aber einen rapiden Anstieg in der Laufzeit.

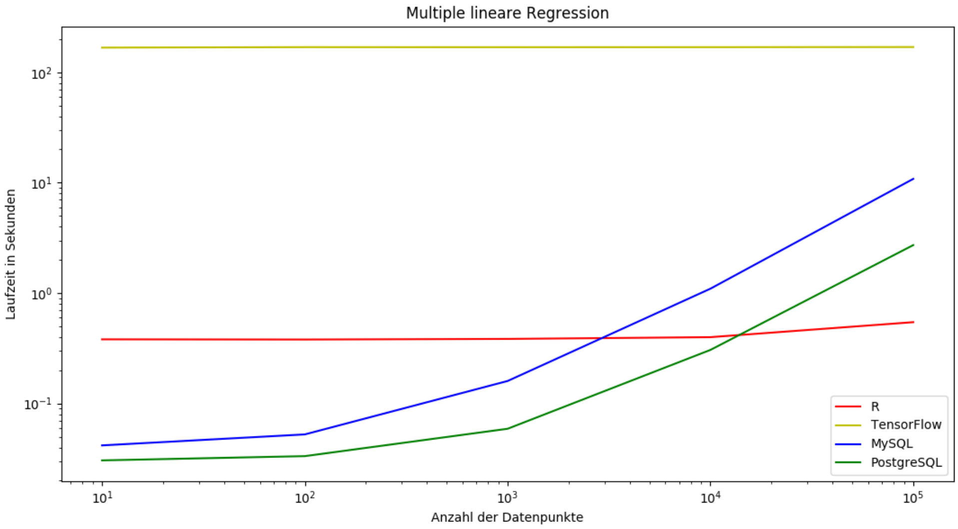
4.2. Multiple lineare Regression

Die Skripte für multiple lineare Regression besitzen folgende Laufzeiten:

Tabelle 4.2.: Laufzeiten in Sekunden für multiple lineare Regression

	10	**100**	**1000**	**10000**	**100000**
r	0.38108478	0.37967851	0.38452695	0.39888490	0.54517789
tensorflow	167.220486	168.885465	168.668689	168.821214	169.254182
mysql	0.04174929	0.05248516	0.15990342	1.09634777	10.8214030
postgresql	0.03053954	0.03336087	0.05904620	0.30572490	2.71992954

Das geplottete Ergebnis sieht wie folgt aus:



Hier zeigt sich ein ähnliches Bild wie schon bei einfacher linearer Regression. Wieder liefern R und TensorFlow relativ konstante Laufzeiten, wobei die Laufzeit des TensorFlow-Skriptes wegen der 50000 durchgeführten Iterationen dieses Mal extrem langsam ist. Wieder sind die SQL-Skripte bei kleinen Datenmengen am schnellsten. Bei größeren Datenmengen werden sie allerdings von R geschlagen. Interessant ist außerdem, dass die PostgreSQL-Implementierung noch schneller als die Variante in MySQL. Die Arrays in PostgreSQL arbeiten also effizienter als die temporären Relationen in MySQL.

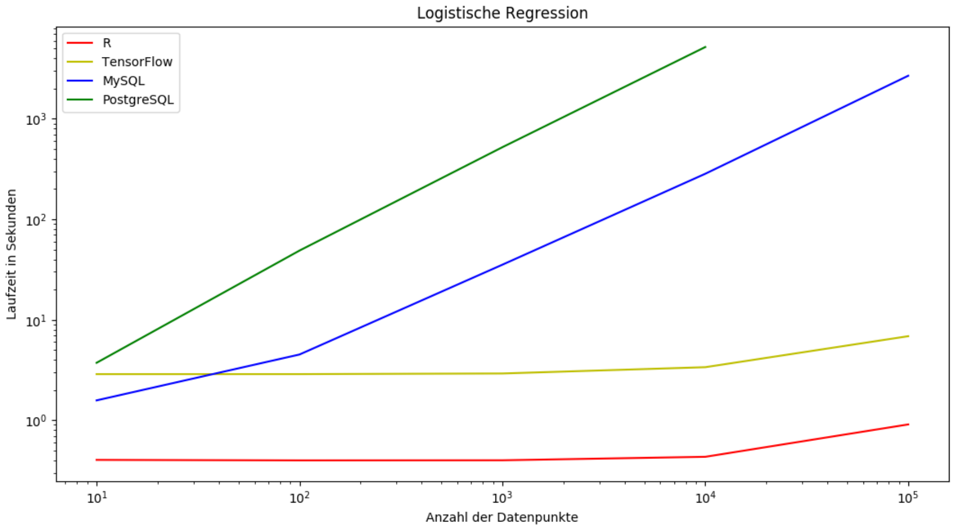
4.3. Logistische Regression

Betrachten wir zuletzt noch die Laufzeiten für logistische Regression:

Tabelle 4.3.: Laufzeiten in Sekunden für logistische Regression

	10	**100**	**1000**	**10000**	**100000**
r	0.40403676	0.39953323	0.40047518	0.43356463	0.91014730
tensorflow	2.88191271	2.88278436	2.92470042	3.37794654	6.87024382
mysql	1.57908838	4.51631200	35.3472805	283.849979	2680.43056
postgresql	3.73521209	48.8452754	521.625671	5175.87997	

Die Visualisierung der Tabelle sieht so aus:



Wieder erkennt man eine Ähnlichkeit zu den vorherigen Diagrammen. Die SQL-Implementierungen sind nun aber von Anfang an deutlich langsamer als die Skripte in R und TensorFlow. Die Laufzeit steigt außerdem sehr schnell weiter an. So wurden für 100000 Datenpunkte in PostgreSQL gar keine Benchmarks mehr berechnet, da die erwartete Laufzeit etwa 50000 Sekunden, also knapp 14 Stunden beträgt. Klarer Gewinner ist hier R, wo auch 100000 Datenpunkte in weniger als einer Sekunde verarbeitet werden können.

5. Erweiterungspotenzial in Datenbanksystemen

In Kapitel 3.4 wurde gezeigt, wie Regressionsanalyse in SQL durchgeführt werden kann. Die dazu implementierten Funktionen können als Erweiterungspotenzial für relationale Datenbanksysteme gesehen werden.

In diesen Funktionen sind die zu verwendende Relation und deren Attribute aktuell noch fest implementiert. Um diese Funktionen in der Praxis nutzbar zu machen, müsste man weitere Parameter einfügen, mit denen man Relation und Attribute zum Zeitpunkt der Ausführung bestimmen kann.

Wir wollen nun die Abfragen der in Kapitel 3.4 implementierten Funktionen mit Hilfe von Operatorbäumen darstellen und beschreiben.

5.1. Einfache lineare Regression

Die einfache lineare Regression besteht aus mehreren kleinen Teilabfragen. Zuerst berechnet man die Mittelwerte der beiden Attribute für die Regression. Das Ergebnis ist die Relation *means*:

$$\begin{array}{c} \rho_{mean\_purchases \leftarrow avg(purchases),\ mean\_money \leftarrow avg(money)} \\ | \\ \gamma_{avg(purchases),\ avg(money)} \\ | \\ datapoints \end{array}$$

Mit diesen Mittelwerten berechnet man die Summen im Nenner und Zähler der Lösungsformel für β aus 2.1.1. Die Relation *sums* ist das Ergebnis dieser Abfrage:

$$\begin{array}{c} \rho_{denominator \leftarrow sum(power(purchases-mean\_purchases,2))} \\ | \\ \rho_{nominator \leftarrow sum((purchases-mean\_purchases)\cdot(money-mean\_money))} \\ | \\ \gamma_{sum((purchases-mean\_purchases)\cdot(money-mean\_money)),\ sum(power(purchases-mean\_purchases,2))} \\ | \\ \times \\ \diagup \quad \diagdown \\ datapoints \quad means \end{array}$$

Damit berechnet man einen Wert für *beta*:

$$\begin{array}{c} \rho_{variable \leftarrow \text{"}beta\text{"},\ value \leftarrow nominator/denominator} \\ | \\ \pi_{\text{"}beta\text{"},\ nominator/denominator} \\ | \\ sums \end{array}$$

Mit Hilfe der Relation *beta* kann man nun auch *alpha* berechnen:

$$\begin{array}{c} \rho_{variable \leftarrow \text{"}alpha\text{"},\ value \leftarrow mean\_money - value \cdot mean\_purchases} \\ | \\ \pi_{\text{"}alpha\text{"},\ mean\_money - value \cdot mean\_purchases} \\ | \\ \times \\ / \quad \backslash \\ means \quad beta \end{array}$$

5.2. Multiple lineare Regression

Die hier gezeichneten Bäume entsprechen der Implementierung in MySQL. Dort verwenden wir Relationen und verarbeiten diese mit den Abfragen der hier dargestellten Bäume. In PostgreSQL verwenden wir dagegen Arrays und Schleifen zur Berechnung.

Wir beginnen mit der Berechnung der transponierten Matrix von X. Die zugehörige Relation wird *matrix_transposed* genannt:

$$\begin{array}{c} \rho_{row \leftarrow column,\ column \leftarrow row} \\ | \\ matrix\_X \end{array}$$

Die Matrixprodukte werden auch in MySQL mit Schleifen berechnet. Dabei wird ein Element der zu berechnenden Matrix mit folgender Abfrage bestimmt. Die Iteratoren *counter*_1 und *counter*_2 sind durch die Schleifen gegeben.

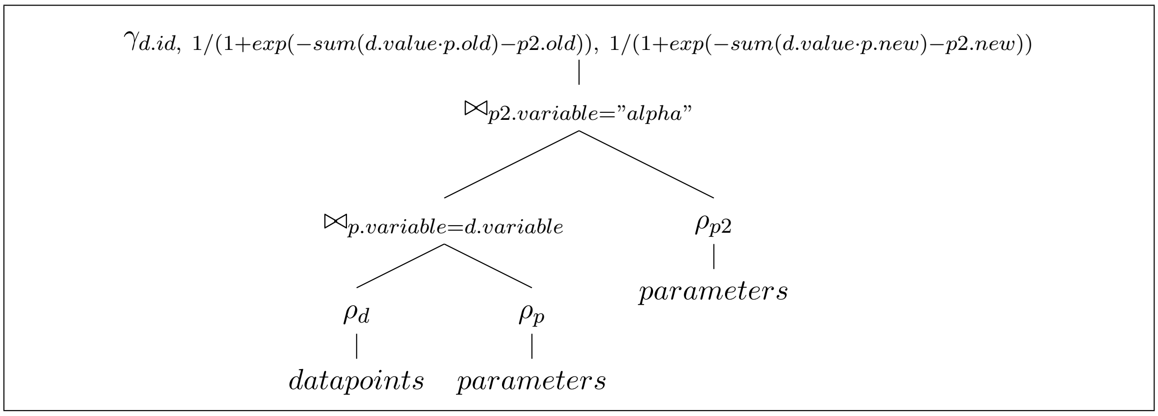
$$\begin{array}{c} \gamma_{sum(m1.value \cdot m2.value)} \\ | \\ \sigma_{m1.row = counter\_1,\ m2.column = counter\_2,\ m1.column = m2.row} \\ | \\ \times \\ / \qquad \backslash \\ \begin{array}{cc} \rho_{m1} & \rho_{m2} \\ | & | \\ matrix\_transposed & matrix\_X \end{array} \end{array}$$

Die beiden Matrixprodukte zur Berechnung von *matrix_product_2* und *matrix_result* besitzen denselben Operatorbaum, nur dass die beiden Relationen *matrix_transposed* und *matrix_y* bzw. *matrix_inverse* und *matrix_product_2* verwendet werden.

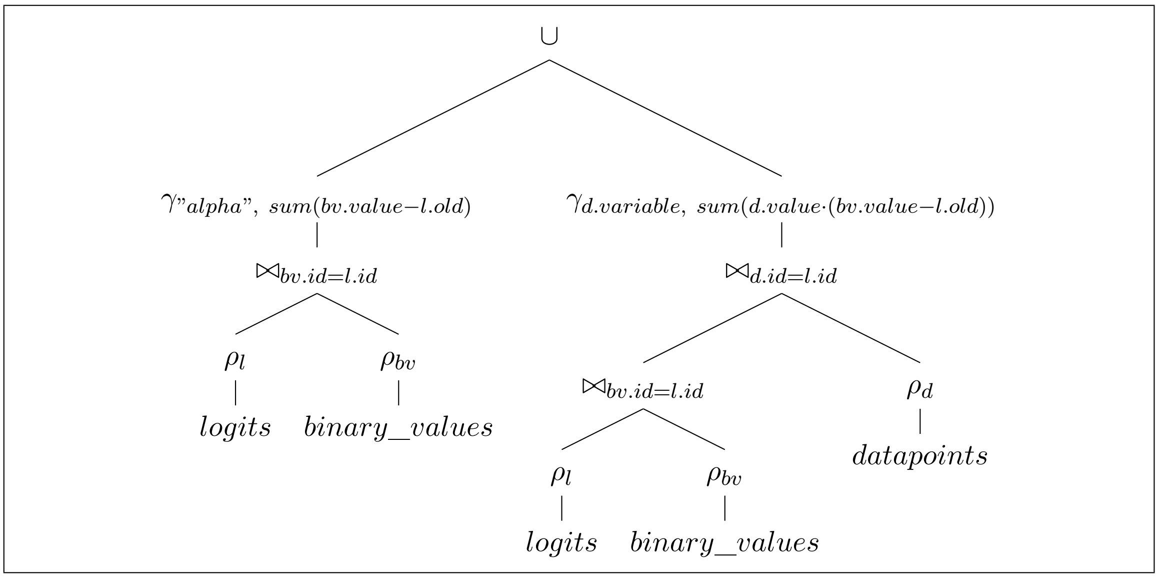
Zur Berechnung der inversen Matrix für die Relation *matrix_inverse* wird in PostgreSQL und MySQL ein iterativer Algorithmus verwendet, dessen Äquivalent wir hier auf Grund der Komplexität nicht als Operatorbaum darstellen wollen.

5.3. Logistische Regression

Die Prozedur für die logistische Regression ist in verschiedene Teilprozeduren aufgeteilt, die jeweils eine Abfrage durchführen. In *calculate_logit* wird die Relation *logits* mit folgender Abfrage befüllt:



Die Prozedur *calculate_gradient*, die den Gradienten in die Relation *gradient* schreibt, führt diese Abfrage aus:



Die Prozedur *calculate_new_parameters* führt eine Update-Abfrage durch. Hierfür zeichnen wir keinen Operatorbaum.

In der Hauptprozedur *logistic_regression* wird in jeder Iteration überprüft, ob die mit der aktuellen Schrittweite berechneten neuen Parameter ein besseres Ergebnis liefern als die alten Parameter. Das geschieht mit folgender Abfrage:

$$
\begin{array}{c}
\pi_{old>new} \\
| \\
\rho_{new \leftarrow sum(log(bv.value \cdot l.new+(1-bv.value)\cdot(1-l.new)))} \\
| \\
\rho_{old \leftarrow sum(log(bv.value \cdot l.old+(1-bv.value)\cdot(1-l.old)))} \\
| \\
\gamma_{sum(log(bv.value \cdot l.new+(1-bv.value)\cdot(1-l.new))),\ sum(log(bv.value \cdot l.old+(1-bv.value)\cdot(1-l.old)))} \\
| \\
\bowtie_{bv.id=l.id} \\
\begin{array}{cc}
/ & \backslash \\
\rho_{l} & \rho_{bv} \\
| & | \\
logits & binary\_values
\end{array}
\end{array}
$$

Ansonsten werden in *logistic_regression* nur die anderen Prozeduren aufgerufen und einfache Update-Abfragen gestellt.

6. Fazit

Wir haben in dieser Arbeit mit der Motivation für Regressionsanalyse begonnen. Danach haben wir die mathematischen Grundlagen der Regressionsanalyse erläutert. Darauf aufbauend haben wir die Umsetzung in R, TensorFlow und SQL demonstriert.

Hierbei haben zwei verschiedene Arten zur Berechnung der Parameter verwendet. Bei expliziter Berechnung wurden verschiedene Berechnungsformeln ausgewertet. Bei iterativer Berechnung wurden Optimierungsverfahren verwendet, um die Werte der gesuchten Parameter Schritt für Schritt besser zu approximieren.

In R haben wir ausschließlich explizite Berechnungen durchgeführt. Dazu haben wir die vorhandenen Funktionen *lm* für lineare Regression und *glm* für logistische Regression verwendet. Das Kürzel *lm* steht für "linear model", *glm* steht für "generalized linear model".

In TensorFlow kamen dagegen ausschließlich iterative Berechnungen zum Einsatz. Wir haben ein von TensorFlow implementiertes Gradientenverfahren genutzt, um eine von uns definierte Kostenfunktion zu minimieren. Die Kostenfunktionen waren dabei die Summe der kleinsten Quadrate bei linearer Regression und die inverse Likelihoodfunktion bei logistischer Regression.

In SQL haben wir beide Berechnungsarten umgesetzt. Bei linearer Regression haben wir die expliziten Formeln aus den Kapiteln 2.1.1 und 2.1.2 verwendet. Bei logistischer Regression wurde wiederum ein Gradientenverfahren angewandt. Dieses Mal wurde das Verfahren eigens implementiert.

Danach haben wir die Implementierungen bezüglich der Laufzeit miteinander verglichen. Dabei haben wir festgestellt, dass die SQL-Skripte bei kleinen Datenmengen und bei Verwendung von expliziten Berechnungsformeln durchaus mit R mithalten können. Besonders bei logistischer Regression erkannte man aber, dass die Implementierung in R von der Laufzeit her deutlich überlegen ist. TensorFlow war durch die strikte Verwendung iterativer Berechnungsarten immer langsamer als R.

Eine Implementierung der hier gezeigten Funktionen für Regression direkt in Datenbanksystemen und die Verwendung von effizienteren Algorithmen könnte die Performanz noch steigern und würde Regressionsanalyse in SQL auch praktisch nutzbar machen. Diese Arbeit ist ein erster "Proof of Concept" zu diesem Thema.

Anhang

A. Python-Skript zum Generieren der Beispieldaten

```python
import sys
import random
import math

# Erzeuge Funktion, die die Daten in eine csv-Datei schreibt.
def outputCsv(data):
  # Erzeuge ein Array mit den zu schreibenden Zeilen und übergebe die Spaltennamen.
  output = ["%s,%s,%s,%s" % (
    "age",
    "purchases",
    "money",
    "premium"
  )]

  # Iteriere über alle Datenpukte.
  for datapoint in data:
    # Hänge eine Zeile an das output-Array an.
    output.append("%s,%s,%s,%a" % (
      datapoint["age"],
      datapoint["purchases"],
      datapoint["money"],
      datapoint["premium"]
    ))

  # Öffne eine csv-Datei.
  f = open("sample.csv", "w")

  # Schreibe das output-Array als mit Zeilenumbrüchen gejointen String in die Datei.
  f.write("\n".join(output))

  return

# Erzeuge Funktion, die die Daten in eine sql-Datei schreibt.
def outputSql(data):
  # Erzeuge ein Array mit den zu schreibenden Zeilen.
```

```python
  # Füge SQL-Abfragen ein, die eine eventuell bestehende Tabelle löscht und neu erstellet.
  # Beginne mit der INSERT-Abfrage.
  output = [
    "DROP TABLE IF EXISTS sample;",
    "",
    "CREATE TABLE sample (",
    "  age INTEGER,",
    "  purchases INTEGER,",
    "  money INTEGER,",
    "  premium INTEGER",
    ");",
    "",
    "INSERT INTO sample (%s,%s,%s,%s) VALUES" % (
      "age",
      "purchases",
      "money",
      "premium"
    )
  ]

  # Iteriere über alle Datenpunkte.
  for datapoint in data:
    # Füge eine Zeile an die INSERT-Abfrage an.
    output.append("(%s,%s,%s,%s)," % (
      datapoint["age"],
      datapoint["purchases"],
      datapoint["money"],
      datapoint["premium"]
    ))

  # Ersetze das letzte Komma durch ein Semikolon, um die INSERT-Abfrage zu beenden.
  output[-1] = output[-1][:-1] + ";"

  # Öffne eine sql-Datei.
  f = open("sample.sql", "w")

  # Schreibe das output-Array als mit Zeilenumbrüchen gejointen String in die Datei.
  f.write("\n".join(output))

  return

def main(argv):
```

```
  # Beende die Ausführung, wenn die Anzahl der zu erzeugenden Datenpunkte nicht übergeben wurde.
  if len(argv) < 2:
    print("Please provide number of datapoints that shall be generated.")
    return

  # Erzeuge ein leeres Array für die Daten.
  data = []

  # Iteriere über die Anzahl der zu erzeugenden Datenpunkte.
  for i in range(0, int(argv[1])):
    # Bestimme pseudo-zufällige Werte für den Datenpunkt.
    age = int(max(random.normalvariate(25, 10) + 10, 18))
    purchases = int(max(random.normalvariate(10, 10), 1))
    money = int(max(purchases * 25 + random.normalvariate(0, (math.log(purchases) + 1) * 12), 0.01) * 100)
    if random.uniform(0, 1) > math.exp(0.2 * purchases - 2) / (1 + math.exp(0.2 * purchases - 2)):
      premium = 0
    else:
      premium = 1

    # Hänge den Datenpunkt an das Array an.
    data.append({
      "age": age,
      "purchases": purchases,
      "money": money,
      "premium": premium
    })

  # Schreibe die generierten Daten in eine .csv und eine .sql Datei.
  outputCsv(data)
  outputSql(data)
  return

# Führe die main-Funktion aus.
if __name__ == "__main__":
  main(sys.argv)
```

B. R-Skripte

B.1. Einfache lineare Regression

```r
# Lese die übergebenen Argumente ein.
args = commandArgs(trailingOnly = TRUE)

# Setzte default-Werte für die Anzahl der Datenpunkte und ob geplottet werden soll.
n <- 1000
plot <- TRUE

# Ändere die default-Werte, falls entsprechende Argumente übergeben wurden.
if (length(args) == 1) {
  if (substr(args[1], 1, 1) == "-") {
    plot <- FALSE
  } else {
    n = strtoi(args[1])
  }
}
if (length(args) == 2) {
  if (substr(args[1], 1, 1) == "-") {
    n = strtoi(args[2])
  } else {
    n = strtoi(args[1])
  }
  plot <- FALSE
}

# Speichere die aktuelle Zeit zur Zeitmessung.
start_time <- Sys.time()

# Lies die Daten aus der csv-Datei ein.
data <- head(read.csv2("./data/sample.csv", sep = ",", header = TRUE), n)

# Definiere das Modell.
modell <- as.formula("money ~ purchases")

# Führe die Regression durch.
slr <- lm(modell, data = data)
```

```
# Speichere die aktuelle Zeit zur Zeitmessung.
end_time <- Sys.time()

# Drucke die Ergebnisse der Regressionsanalyse und die Laufzeit.
print(slr)
print(end_time - start_time)

# Erstelle einen Plot.
if (plot) {
  # Bestimme die Grenzen für die unabhängige Variable.
  xmin <- min(data$purchases)
  xmax <- max(data$purchases)

  # Bestimme die Grenzen für die abhängige Variable.
  ymin <- min(data$money)
  ymax <- max(data$money)

  # Bestimme die Koeffizienten aus der Regressionsanalyse.
  b0 <- coef(slr["coefficients"])[1]
  b1 <- coef(slr["coefficients"])[2]

  # Erzeuge Vektoren zum Plot der linearen Funktion.
  xplot <- c(xmin - 1, xmax + 1)
  yplot <- c(b0 + (xmin - 1) * b1, b0 + (xmax + 1) * b1)

  # Erstelle den Plot.
  plot(
    c(xmin - 1, xmax + 1),
    c(ymin - 1, ymax + 1),
    type = "n",
    xlab = "purchases",
    ylab = "money",
    main = "Einfache lineare Regression",
    sub = paste("money = ", b0, "+", b1, "* purchases"),
    col.sub = "darkgray"
  )
  # Füge die Datenpunkte ein.
  lines(
    data$purchases,
    data$money,
    type="p"
  )
  # Füge die Ausgleichsgerade ein.
  lines(
    xplot,
```

```
    yplot,
    col = "red",
    lwd = 2
  )
}
```

B.2. Multiple lineare Regression

```
# Lese die übergebenen Argumente ein.
args = commandArgs(trailingOnly = TRUE)

# Setzte default-Wert für die Anzahl der Datenpunkte.
n = 1000

# Ändere den default-Wert, falls ein entsprechendes Argumente übergeben wurde.
if (length(args) > 0) {
  n = strtoi(args[1])
}

# Speichere die aktuelle Zeit zur Zeitmessung.
start_time <- Sys.time()

# Lies die Daten aus der csv-Datei ein.
data <- head(read.csv2("./data/sample.csv", sep = ",", header = TRUE), n)

# Definiere das Modell.
modell <- as.formula("money ~ purchases + age")

# Führe die Regression durch.
mlr <- lm(modell, data = data)

# Speichere die aktuelle Zeit zur Zeitmessung.
end_time <- Sys.time()

# Drucke die Ergebnisse der Regressionsanalyse und die Laufzeit.
print(mlr)
print(end_time - start_time)
```

B.3. Logistische Regression

```
# Lese die übergebenen Argumente ein.
args = commandArgs(trailingOnly = TRUE)

```

```r
# Setzte default-Werte für die Anzahl der Datenpunkte und ob geplottet
↪  werden soll.
n <- 1000
plot <- TRUE

# Ändere die default-Werte, falls entsprechende Argumente übergeben
↪  wurden.
if (length(args) == 1) {
  if (substr(args[1], 1, 1) == "-") {
    plot <- FALSE
  } else {
    n = strtoi(args[1])
  }
}
if (length(args) == 2) {
  if (substr(args[1], 1, 1) == "-") {
    n = strtoi(args[2])
  } else {
    n = strtoi(args[1])
  }
  plot <- FALSE
}

# Speichere die aktuelle Zeit zur Zeitmessung.
start_time <- Sys.time()

# Lies die Daten aus der csv-Datei ein.
data <- head(read.csv2("./data/sample.csv", sep = ",", header = TRUE), n)

# Definiere das Modell.
modell <- as.formula("premium ~ money")

# Führe die Regression durch.
logit <- glm(modell, family = binomial, data = data)

# Speichere die aktuelle Zeit zur Zeitmessung.
end_time <- Sys.time()

# Drucke die Ergebnisse der Regressionsanalyse und die Laufzeit.
print(logit)
print(end_time - start_time)

# Erstelle einen Plot.
if (plot) {
  # Bestimme die Grenzen für die unabhängige Variable.
  xmin <- min(data$money)
```

```
  xmax <- max(data$money)

  # Bestimme eine Funktion, die den Wert der logistischen Funktion berechnet.
  logitFunction <- function(x){
    # Verwende die Parameter aus der Regressionsanalyse.
    b0 <- coef(logit["coefficients"])[1]
    b1 <- coef(logit["coefficients"])[2]
    c <- b0 + x * b1
    return(exp(c) / (1 + exp(c)))
  }

  # Erzeuge Vektoren zum Plot der logistischen Funktion.
  xplot <- seq(xmin - 1, xmax + 1, 1000)
  yplot <- logitFunction(xplot)

  # Erstelle den Plot.
  plot(
    c(xmin - 1, xmax + 1),
    c(-0.2, 1.2),
    type = "n",
    xlab = "money",
    ylab = "premium",
    main = "Logistische Regression"
  )
  # Füge die Datenpunkte ein.
  lines(
    data$money,
    data$premium,
    type="p"
  )
  # Füge die logistische Funktion ein.
  lines(
    xplot,
    yplot,
    col = "red",
    lwd = 2
  )
}
```

C. TensorFlow-Skripte

C.1. Einfache lineare Regression

```python
import numpy as np
import tensorflow as tf
import os.path as p
import csv
import sys
import matplotlib.pyplot as plt
from time import time

# Erstelle eine Funktion zum einlesen der Daten aus der csv-Daten.
def get_data(n_samples):
  # Bestimme den Pfad der csv-Datei und öffne die Datei.
  filename = p.abspath(p.join(p.dirname(p.realpath(__file__)), "..", "data", "sample.csv"))
  csvfile = open(filename, newline="")
  csvreader = csv.reader(csvfile, delimiter=",", quotechar="|")

  # Erstelle leere Arrays für die Daten.
  x = []
  x_plot = []
  y = []

  # Iteriere über die Zeilen der csv-Datei.
  for row in csvreader:
    # Überspringe die erste Zeile.
    if not row[0] == "age":
      # Füge die Daten der Zeile zum jeweiligen Array hinzu.
      x.append([int(row[1])])
      x_plot.append(int(row[1]))
      y.append(int(row[2]))

  # Gib die Arrays bis zu der gewünschten Menge an Datenpunkten zurück.
  return (np.array(x[:n_samples]), x_plot[:n_samples], np.transpose([y[:n_samples]]))

def main(argv):
  # Bestimme default-Werte für die Anzahl der Datenpunkte und ob geplottet werden soll.
```

```
datapoint_size = 1000
plot = True

# Ändere die default-Werte, wenn entsprechende Argumente übergeben wurden.
if len(argv) == 2:
  if argv[1] == "-":
    plot = False
  else:
    datapoint_size = int(argv[1])
elif len(argv) == 3:
  plot = False
  if argv[1] == "-":
    datapoint_size = int(argv[2])
  else:
    datapoint_size = int(argv[1])

# Bestimme die aktuelle Zeit zur Zeitmessung.
start_time = time()

# Bestimme die Anzahl der Iterationen und die Schrittweite (anhängig von der Anzahl der Datenpunkte.)
steps = 2000
if datapoint_size <= 10:
  learn_rate = 0.0076
elif datapoint_size <= 100:
  learn_rate = 0.0064
elif datapoint_size <= 1000:
  learn_rate = 0.0056
elif datapoint_size <= 10000:
  learn_rate = 0.0054
elif datapoint_size <= 100000:
  learn_rate = 0.0054

# Deklariere die Platzhalter und Variablen.
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]))
y_calc = tf.matmul(x, beta) + alpha

# Definiere die Kostenfunktion und die Minimierungsoperation.
cost = tf.reduce_mean(tf.square(y - y_calc))
train_step = tf.train.GradientDescentOptimizer(learn_rate).minimize(cost)
```

```
  # Importiere die Daten.
  (all_xs, plot_xs, all_ys) = get_data(datapoint_size)

  # Starte eine Session in TensorFlow.
  sess = tf.Session()
  init = tf.global_variables_initializer()
  sess.run(init)

  # Iteriere und trainiere.
  for i in range(steps):
    feed = { x: all_xs, y: all_ys }
    sess.run(train_step, feed_dict=feed)

  # Bestimme die aktuellen Parameterwerte nach der Anzahl der Iterationen.
  (curr_alpha, curr_beta, curr_cost) = sess.run([alpha, beta, cost], feed_dict=feed)

  # Bestimme die aktuelle Zeit zur Zeitmessung.
  end_time = time()

  # Drucke die Ergebnisse.
  print("alpha:  %f" % curr_alpha)
  print("beta:   %f" % curr_beta)
  print("cost:   %f" % curr_cost)
  print("")
  print("time elapsed:  %f sec" % (end_time - start_time))

  # Erstelle einen Plot (falls gewünscht).
  if plot:
    plt.plot(plot_xs, all_ys, "ro", label="Original data")
    plt.plot(plot_xs, curr_beta * all_xs + curr_alpha , label="Fitted line")
    plt.legend()
    plt.show()

# Führe die main-Funktion aus.
if __name__ == "__main__":
  main(sys.argv)
```

C.2. Multiple lineare Regression

```
import numpy as np
import tensorflow as tf
import os.path as p
```

```
import csv
import sys
from time import time

# Erstelle eine Funktion zum einlesen der Daten aus der csv-Daten.
def get_data(n_samples):
  # Bestimme den Pfad der csv-Datei und öffne die Datei.
  filename = p.abspath(p.join(p.dirname(p.realpath(__file__)), "..",
  ↪  "data", "sample.csv"))
  csvfile = open(filename, newline="")
  csvreader = csv.reader(csvfile, delimiter=",", quotechar="|")

  # Erstelle leere Arrays für die Daten.
  x = []
  y = []

  # Iteriere über die Zeilen der csv-Datei.
  for row in csvreader:
    # Überspringe die erste Zeile.
    if not row[0] == "age":
      # Füge die Daten der Zeile zum jeweiligen Array hinzu.
      x.append([int(row[1]), int(row[0])])
      y.append(int(row[2]))

  # Gib die Arrays bis zu der gewünschten Menge an Datenpunkten zurück.
  return (np.array(x[:n_samples]), np.transpose([y[:n_samples]]))

def main(argv):
  # Bestimme default-Wert für die Anzahl der Datenpunkte.
  datapoint_size = 1000

  # Ändere den default-Wert, wenn ein entsprechendes Argument übergeben
  ↪  wurde.
  if len(argv) == 2:
    datapoint_size = int(argv[1])

  # Bestimme die aktuelle Zeit zur Zeitmessung.
  start_time = time()

  # Bestimme die Anzahl der Iterationen und die Schrittweite (anhängig von
  ↪  der Anzahl der Datenpunkte.)
  steps = 50000
  if datapoint_size <= 10:
    learn_rate = 0.00093
  elif datapoint_size <= 100:
    learn_rate = 0.00078
```

```python
  elif datapoint_size <= 1000:
    learn_rate = 0.0007
  elif datapoint_size <= 10000:
    learn_rate = 0.00071
  elif datapoint_size <= 100000:
    learn_rate = 0.00071

  # Deklariere die Platzhalter und Variablen.
  x = tf.placeholder(tf.float32, [None, 2])
  y = tf.placeholder(tf.float32, [None, 1])
  alpha = tf.Variable(tf.zeros([1]))
  beta = tf.Variable(tf.zeros([2, 1]))
  y_calc = tf.matmul(x, beta) + alpha

  # Definiere die Kostenfunktion und die Minimierungsoperation.
  cost = tf.reduce_mean(tf.square(y - y_calc))
  train_step = tf.train.GradientDescentOptimizer(learn_rate).minimize(cost)

  # Importiere die Daten.
  (all_xs, all_ys) = get_data(datapoint_size)

  # Starte eine Session in TensorFlow.
  sess = tf.Session()
  init = tf.global_variables_initializer()
  sess.run(init)

  # Iteriere und trainiere.
  for i in range(steps):
    feed = { x: all_xs, y: all_ys }
    sess.run(train_step, feed_dict=feed)

  # Bestimme die aktuellen Parameterwerte nach der Anzahl der Iterationen.
  (curr_alpha, curr_beta, curr_cost) = sess.run([alpha, beta, cost], feed_dict=feed)

  # Bestimme die aktuelle Zeit zur Zeitmessung.
  end_time = time()

  # Drucke die Ergebnisse.
  print("alpha:           %f" % curr_alpha)
  print("beta_purchases:  %f" % curr_beta[0])
  print("beta_age:        %f" % curr_beta[1])
  print("cost:            %f" % curr_cost)
  print("")
```

```python
  print("time elapsed: %f sec" % (end_time - start_time))

# Führe die main-Funktion aus.
if __name__ == "__main__":
  main(sys.argv)
```

C.3. Logistische Regression

```python
import numpy as np
import tensorflow as tf
import os.path as p
import csv
import sys
import matplotlib.pyplot as plt
from time import time

# Erstelle eine Funktion zum einlesen der Daten aus der csv-Daten.
def get_data(n_samples):
  # Bestimme den Pfad der csv-Datei und öffne die Datei.
  filename = p.abspath(p.join(p.dirname(p.realpath(__file__)), "..",
  ↪ "data", "sample.csv"))
  csvfile = open(filename, newline="")
  csvreader = csv.reader(csvfile, delimiter=",", quotechar="|")

  # Erstelle leere Arrays für die Daten.
  x = []
  x_plot = []
  y = []

  # Iteriere über die Zeilen der csv-Datei.
  for row in csvreader:
    # Überspringe die erste Zeile.
    if not row[0] == "age":
      # Füge die Daten der Zeile zum jeweiligen Array hinzu.
      x.append([int(row[2])])
      x_plot.append(int(row[2]))
      y.append(int(row[3]))

  # Gib die Arrays bis zu der gewünschten Menge an Datenpunkten zurück.
  return (np.array(x[:n_samples]), x_plot[:n_samples],
  ↪ np.transpose([y[:n_samples]]))

def main(argv):
  # Bestimme default-Werte für die Anzahl der Datenpunkte und ob geplottet
  ↪ werden soll.
```

```
datapoint_size = 1000
plot = True

# Ändere die default-Werte, wenn entsprechende Argumente übergeben
↪  wurden.
if len(argv) == 2:
  if argv[1] == "-":
    plot = False
  else:
    datapoint_size = int(argv[1])
elif len(argv) == 3:
  plot = False
  if argv[1] == "-":
    datapoint_size = int(argv[2])
  else:
    datapoint_size = int(argv[1])

# Bestimme die aktuelle Zeit zur Zeitmessung.
start_time = time()

# Bestimme die Anzahl der Iterationen und die Schrittweite (anhängig von
↪  der Anzahl der Datenpunkte.)
steps = 1000
if datapoint_size == 10:
  learn_rate = 1
elif datapoint_size == 100:
  learn_rate = 0.1
elif datapoint_size == 1000:
  learn_rate = 0.01
elif datapoint_size == 10000:
  learn_rate = 0.001
elif datapoint_size == 100000:
  learn_rate = 0.0001

# Deklariere die Platzhalter und Variablen.
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]))
y_calc = 1 / (1 + tf.exp(- tf.matmul(x, beta) - alpha))

# Definiere die Kostenfunktion und die Minimierungsoperation.
cost = - tf.reduce_sum(
  tf.log(
    y * y_calc +
    (1 - y) * (1 - y_calc)
```

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

# Importiere die Daten.
(all_xs, plot_xs, all_ys) = get_data(datapoint_size)

# Transformiere die unabhägige Variable linear ins Interval zwischen 0 und 1.
min_x = min(all_xs)
max_x = max(all_xs)
all_xs = (all_xs - min_x) / (max_x - min_x)

# Starte eine Session in TensorFlow.
sess = tf.Session()
init = tf.global_variables_initializer()
sess.run(init)

# Iteriere und trainiere.
for i in range(steps):
  feed = { x: all_xs, y: all_ys }
  sess.run(train_step, feed_dict=feed)

# Bestimme die aktuellen Parameterwerte nach der Anzahl der Iterationen.
(curr_alpha, curr_beta, curr_cost) = sess.run([alpha, beta, cost], feed_dict=feed)

# Transformiere die Parameter linear, um den originalen Daten zu entsprechen.
curr_beta = curr_beta / (max_x - min_x)
curr_alpha = curr_alpha - curr_beta * min_x

# Bestimme die aktuelle Zeit zur Zeitmessung.
end_time = time()

# Drucke die Ergebnisse.
print("alpha:  %f" % curr_alpha)
print("beta:   %f" % curr_beta)
print("cost:   %f" % curr_cost)
print("")
print("time elapsed: %f sec" % (end_time - start_time))

# Erstelle einen Plot (falls gewünscht).
if plot:
```

```
    all_xs = all_xs * (max_x - min_x) + min_x
    plot_ys = 1 / (1 + np.exp(- curr_beta * all_xs - curr_alpha))
    plot_order = np.argsort(plot_xs)
    plt.plot(plot_xs, all_ys, "ro", label="Original data")
    plt.plot(np.array(plot_xs)[plot_order], np.array(plot_ys)[plot_order],
     ↪  label="Fitted line")
    plt.legend()
    plt.show()

# Führe die main-Funktion aus.
if __name__ == "__main__":
  main(sys.argv)
```

D. MySQL-Skripte

D.1. Einfache lineare Regression

```sql
-- Lösche die bestehende Prozedur, falls vorhanden.
DROP PROCEDURE IF EXISTS simple_linear_regression;

DELIMITER ;;

-- Erstelle die Prozedur für einfache lineare Regression.
CREATE PROCEDURE `simple_linear_regression`(IN number_datapoints INT(11))
BEGIN

-- Deklariere die verwendeten Variablen.
DECLARE purchases_mean DECIMAL(65, 30);
DECLARE money_mean DECIMAL(65, 30);
DECLARE alpha DECIMAL(65, 30);
DECLARE beta DECIMAL(65, 30);

-- Erstelle eine temporäre Relation für die zu verwendenden Datenpunkte.
DROP TEMPORARY TABLE IF EXISTS datapoints;
CREATE TEMPORARY TABLE datapoints (
  purchases INT(11),
  money INT(11)
);

-- Füge die gewünschte Anzahl der Datenpunkte in die temporäre Relation ein.
INSERT INTO datapoints
SELECT purchases, money
FROM sample
LIMIT number_datapoints;

-- Berechne die Mittelwerte der abhängigen und unabhängigen Variable.
SET purchases_mean = (
  SELECT AVG(purchases)
  FROM datapoints
);
SET money_mean = (
  SELECT AVG(money)
  FROM datapoints
```

```
);

-- Berechne beta.
SET beta = (
  SELECT SUM((purchases - purchases_mean) * (money - money_mean))
  FROM datapoints
);
SET beta = beta / (
  SELECT SUM(POWER(purchases - purchases_mean, 2))
  FROM datapoints
);

-- Berechne alpha.
SET alpha = money_mean - (beta * purchases_mean);

-- Gib eine Relation mit Parametername und zugehörigem Wert zurück.
SELECT 'alpha' AS `variable`, alpha AS `value`
UNION
SELECT 'beta' AS `variable`, beta AS `value`;

-- Lösche die temporäre Relation mit den Datenpunkten wieder.
DROP TEMPORARY TABLE IF EXISTS datapoints;

END;;

DELIMITER ;
```

D.2. Multiple lineare Regression

```
-- Lösche die bestehende Prozedur, falls vorhanden.
DROP PROCEDURE IF EXISTS multiple_linear_regression;

DELIMITER ;;

-- Erstelle die Prozedur für multiple lineare Regression.
CREATE PROCEDURE multiple_linear_regression(IN number_datapoints INT(11))
BEGIN

-- Deklariere die verwendeten Variablen.
DECLARE m INT(11);
DECLARE n INT(11);
DECLARE counter_1 INT(11);
DECLARE counter_2 INT(11);
DECLARE counter_3 INT(11);
DECLARE pivot DECIMAL(65, 30);
```

```sql

-- Bestimme die Dimensionsn für die Matrix X.
SET m = number_datapoints;
SET n = 3;

-- Lösche vorhandene temporäre Relationen.
DROP TEMPORARY TABLE IF EXISTS matrix_X;
DROP TEMPORARY TABLE IF EXISTS matrix_transposed;
DROP TEMPORARY TABLE IF EXISTS matrix_product_1;
DROP TEMPORARY TABLE IF EXISTS matrix_inverse;
DROP TEMPORARY TABLE IF EXISTS matrix_product_2;
DROP TEMPORARY TABLE IF EXISTS matrix_y;
DROP TEMPORARY TABLE IF EXISTS matrix_result;

-- Erstelle temporäre Relationen für die zu berechnenden Matrizen.
CREATE TEMPORARY TABLE matrix_X (
  `row` INT(11),
  `column` INT(11),
  `value` DECIMAL(65, 30)
);
CREATE TEMPORARY TABLE matrix_transposed (
  `row` INT(11),
  `column` INT(11),
  `value` DECIMAL(65, 30)
);
CREATE TEMPORARY TABLE matrix_product_1 (
  `row` INT(11),
  `column` INT(11),
  `value` DECIMAL(65, 30)
);
CREATE TEMPORARY TABLE matrix_inverse (
  `row` INT(11),
  `column` INT(11),
  `value` DECIMAL(65, 30)
);
CREATE TEMPORARY TABLE matrix_product_2 (
  `row` INT(11),
  `column` INT(11),
  `value` DECIMAL(65, 30)
);
CREATE TEMPORARY TABLE matrix_y (
  `row` INT(11),
  `column` INT(11),
  `value` DECIMAL(65, 30)
);
CREATE TEMPORARY TABLE matrix_result (
```

```sql
  `row` INT(11),
  `column` INT(11),
  `value` DECIMAL(65, 30)
);

-- Füge Werte der unabhängigen Variablen in die Relation matrix_X ein.
SET @id = 0;

INSERT INTO matrix_X
SELECT
  @id := (@id + 1) AS `row`,
  1 AS `column`,
  1 AS `value`
FROM sample
LIMIT number_datapoints;

SET @id = 0;

INSERT INTO matrix_X
SELECT
  @id := (@id + 1) AS `row`,
  2 AS `column`,
  purchases AS `value`
FROM sample
LIMIT number_datapoints;

SET @id = 0;

INSERT INTO matrix_X
SELECT
  @id := (@id + 1) AS `row`,
  3 AS `column`,
  age AS `value`
FROM sample
LIMIT number_datapoints;

-- Füge Werte der abhängigen Variable in die Relation matrix_y ein.
SET @id = 0;

INSERT INTO matrix_y
SELECT
  @id := (@id + 1) AS `row`,
  1 AS `column`,
  money AS `value`
FROM sample
LIMIT number_datapoints;
```

```
-- Berechne matrix_transposed.
INSERT INTO matrix_transposed
SELECT
  `column` AS `row`,
  `row` AS `column`,
  `value` AS `value`
FROM matrix_X;

-- Berechne matrix_product_1. Iteriere dazu über alle Zeilen und Spalten
 ↪  der Ergebnismatrix.
SET counter_1 = 1;

WHILE counter_1 <= n DO

  SET counter_2 = 1;

  WHILE counter_2 <= n DO

    -- Berechne den Wert des aktuellen Matrixelements.
    INSERT INTO matrix_product_1 VALUES (
      counter_1,
      counter_2,
      (
        SELECT SUM(matrix_X.`value` * matrix_transposed.`value`)
        FROM matrix_X, matrix_transposed
        WHERE matrix_X.`column` = counter_2
          AND matrix_transposed.`row` = counter_1
          AND matrix_transposed.`column` = matrix_X.`row`
      )
    );

    SET counter_2 = counter_2 + 1;

  END WHILE;

  SET counter_1 = counter_1 + 1;

END WHILE;

-- Berechne matrix_inverse. Verwende dazu den in der Arbeit referenzierten
 ↪  Algorithmus.
INSERT INTO matrix_inverse
SELECT *
FROM matrix_product_1;
```

```
SET counter_1 = 0;

WHILE counter_1 < n DO

  SET counter_1 = counter_1 + 1;

  DROP TEMPORARY TABLE IF EXISTS pivot_row;
  CREATE TEMPORARY TABLE pivot_row (
      `column` INT(11),
      `value` DECIMAL(65, 30)
  );

  INSERT INTO pivot_row
  SELECT `column`, `value`
  FROM matrix_inverse
  WHERE `row` = counter_1;

  SET pivot = (
      SELECT `value`
      FROM matrix_inverse
      WHERE `row` = counter_1 AND `column` = counter_1
  );

  UPDATE matrix_inverse
  SET `value` = `value` / pivot
  WHERE `row` = counter_1 AND `column` <> counter_1;

  UPDATE matrix_inverse
  SET `value` = - `value` / pivot
  WHERE `row` <> counter_1 AND `column` = counter_1;

  SET counter_2 = 1;

  WHILE counter_2 <= n DO

    IF counter_2 <> counter_1 THEN

      SET counter_3 = 1;

      WHILE counter_3 <= n DO

        IF counter_3 <> counter_1 THEN

          SET pivot = (
            SELECT `value`
            FROM pivot_row
```

```
          WHERE `column` = counter_3
        ) * (
          SELECT `value`
          FROM matrix_inverse
          WHERE `row` = counter_2 AND `column` = counter_1
        );

        UPDATE matrix_inverse
        SET `value` = `value` + pivot
        WHERE `row` = counter_2 AND `column` = counter_3;

      END IF;

      SET counter_3 = counter_3 + 1;

    END WHILE;

  END IF;

  SET counter_2 = counter_2 + 1;

END WHILE;

UPDATE matrix_inverse
SET `value` = 1 / `value`
WHERE `row` = counter_1 AND `column` = counter_1;

END WHILE;

-- Berechne matrix_product_2. Iteriere dazu über alle Zeilen der
↪ Ergebnismatrix.
SET counter_1 = 1;

WHILE counter_1 <= n DO

  -- Berechne den Wert des aktuellen Matrixelements.
  INSERT INTO matrix_product_2 VALUES (
    counter_1,
    1,
    (
      SELECT SUM(matrix_y.`value` * matrix_transposed.`value`)
      FROM matrix_y, matrix_transposed
      WHERE matrix_transposed.`row` = counter_1
        AND matrix_transposed.`column` = matrix_y.`row`
    )
  );
```

```

  SET counter_1 = counter_1 + 1;

END WHILE;

-- Berechne matrix_result. Iteriere dazu über alle Zeilen der
↪  Ergebnismatrix.
SET counter_1 = 1;

WHILE counter_1 <= n DO

  -- Berechne den Wert des aktuellen Matrixelements.
  INSERT INTO matrix_result VALUES (
    counter_1,
    1,
    (
      SELECT SUM(matrix_product_2.`value` * matrix_inverse.`value`)
      FROM matrix_product_2, matrix_inverse
      WHERE matrix_inverse.`row` = counter_1
        AND matrix_inverse.`column` = matrix_product_2.`row`
    )
  );

  SET counter_1 = counter_1 + 1;

END WHILE;

-- Gib eine Relation mit Parameternamen und zugehörigen Werten zurück.
SELECT
  CASE row
    WHEN 1 THEN 'alpha'
    WHEN 2 THEN 'beta_purchases'
    WHEN 3 THEN 'beta_age'
  END AS `variable`,
  value
FROM matrix_result;

-- Lösche die temporären Relationen wieder.
DROP TEMPORARY TABLE IF EXISTS matrix_X;
DROP TEMPORARY TABLE IF EXISTS matrix_transposed;
DROP TEMPORARY TABLE IF EXISTS matrix_product_1;
DROP TEMPORARY TABLE IF EXISTS matrix_inverse;
DROP TEMPORARY TABLE IF EXISTS matrix_product_2;
DROP TEMPORARY TABLE IF EXISTS matrix_y;
DROP TEMPORARY TABLE IF EXISTS matrix_result;

```

```
END;;

DELIMITER ;
```

D.3. Logistische Regression

```
-- Lösche die bestehenden Prozeduren, falls vorhanden.
DROP PROCEDURE IF EXISTS calculate_gradient;
DROP PROCEDURE IF EXISTS calculate_new_parameters;
DROP PROCEDURE IF EXISTS calculate_logit;
DROP PROCEDURE IF EXISTS logistic_regression;

DELIMITER ;;
-- Erstelle eine Prozedur zur Berechnung der Werte der logistischen Funktion.
CREATE PROCEDURE `calculate_logit`()
BEGIN

-- Deklariere die benötigten Variablen.
DECLARE alpha_old DECIMAL(65, 30);
DECLARE alpha_new DECIMAL(65, 30);

-- Bestimme den alten und neuen Wert von alpha.
SET alpha_old = (
  SELECT old
  FROM parameters
  WHERE variable = 'alpha'
);
SET alpha_new = (
  SELECT new
  FROM parameters
  WHERE variable = 'alpha'
);

-- Berechne die Werte der logistischen Funktion für alle Datenpunkte.
DELETE FROM logits;
INSERT INTO logits
  SELECT
    d.id,
    1 / (1 + exp(- SUM(d.value * p.old) - alpha_old)) AS `old`,
    1 / (1 + exp(- SUM(d.value * p.new) - alpha_new)) AS `new`
  FROM datapoints d
  JOIN parameters p ON p.variable = d.variable
  GROUP BY d.id;

```

```
END;;

-- Erstelle eine Prozedur zur Berechnung des Gradienten.
CREATE PROCEDURE `calculate_gradient`()
BEGIN

DELETE FROM gradient;

-- Berechne die partielle Ableitung nach alpha.
INSERT INTO gradient
SELECT 'alpha' AS `variable`, SUM(bv.value - l.old) AS `value`
FROM logits l
JOIN binary_values bv ON bv.id = l.id;

-- Berechne die partielle Ableitung nach allen beta-Parametern.
INSERT INTO gradient
SELECT d.variable, SUM(d.value * (bv.value - l.old)) AS `value`
FROM logits l
JOIN binary_values bv ON bv.id = l.id
JOIN datapoints d ON d.id = l.id
GROUP BY d.variable;

END;;

-- Erzeuge eine Prozedur zur Berechnung der neuen Parameter abhängig von
↪  der aktuellen Schrittweite.
CREATE PROCEDURE `calculate_new_parameters`(IN step DECIMAL(65, 30))
BEGIN

UPDATE parameters
JOIN gradient ON gradient.variable = parameters.variable
SET parameters.new = parameters.old + step * gradient.value;

END;;

-- Erstelle die Prozedur für logistische Regression.
CREATE PROCEDURE `logistic_regression`(IN number_datapoints INT(11), IN
↪  rounds INT(11), step DECIMAL(65, 30))
BEGIN

-- Deklariere die verwendeten Variablen.
DECLARE min INT(11);
DECLARE max INT(11);
DECLARE transform DECIMAL(65, 30);
DECLARE better INT(1);
DECLARE counter INT(11);
```

```

-- Erstelle eine temporäre Relation für die Werte der unabhängigen Variablen.
DROP TEMPORARY TABLE IF EXISTS datapoints;
CREATE TEMPORARY TABLE datapoints (
  id INT(11),
  variable VARCHAR(32),
  value DECIMAL(65, 30),
  PRIMARY KEY (id, variable)
);

-- Berechne Minimum und Maximum der unabhängigen Variable.
SET min = (SELECT MIN(money) FROM sample);
SET max = (SELECT MAX(money) FROM sample);

-- Füge die linear transformierten Werte der unabhängigen Variablen in die Relation datapoints ein.
SET @counter = 0;
INSERT INTO datapoints
SELECT
  @counter := @counter + 1 AS `id`,
  'beta_money' AS `variable`,
  (money - min) / (max - min) AS `value`
FROM sample
LIMIT number_datapoints;

-- Erstelle eine temporäre Relation für die (binären) Werte der abhängigen Variablen.
DROP TEMPORARY TABLE IF EXISTS binary_values;
CREATE TEMPORARY TABLE binary_values (
  id INT(11),
  value INT(1),
  PRIMARY KEY (id)
);

-- Füge die Werte der abhängingen Variable ein.
SET @counter = 0;
INSERT INTO binary_values
SELECT
  @counter := @counter + 1 AS `id`,
  premium AS `value`
FROM sample
LIMIT number_datapoints;

-- Erstelle eine temporäre Relation für die alten und neuen Parameterwerte.
```

```sql
DROP TEMPORARY TABLE IF EXISTS parameters;
CREATE TEMPORARY TABLE parameters (
  variable VARCHAR(32),
  old DECIMAL(65, 30),
  new DECIMAL(65, 30),
  PRIMARY KEY (variable)
);

-- Füge die Initialwerte der Parameter ein.
INSERT INTO parameters VALUES
  ('alpha', 0, 0),
  ('beta_money', 0, 0);

-- Erstelle eine temporäre Relation für die Werte der logistischen Funktion für alle Datenpunkte.
DROP TEMPORARY TABLE IF EXISTS logits;
CREATE TEMPORARY TABLE logits (
  id INT(11),
  old DECIMAL(65, 30),
  new DECIMAL(65, 30),
  PRIMARY KEY (id)
);

-- Befülle die Relation für die Werte der logistischen Funktion.
CALL calculate_logit();

-- Erstelle eine teporäre Relation für den Gradienten.
DROP TEMPORARY TABLE IF EXISTS gradient;
CREATE TEMPORARY TABLE gradient (
  variable VARCHAR(32),
  value DECIMAL(65, 30),
  PRIMARY KEY (variable)
);

-- Iteriere über die Anzahl der gewünschten Iterationen.
SET counter = 0;
WHILE counter < rounds AND step > 0.000000000000000000000000000001 DO

  -- Berechne den Gradienten und die neuen Parameter mit der aktuellen Schrittweite.
  CALL calculate_gradient();
  CALL calculate_new_parameters(step);
  CALL calculate_logit();

  -- Verringere die Schrittweite solange, bis die neuen Parameter ein besseres Ergebnis liefern als die alten.
```

```
  WHILE (
    SELECT
      SUM(LOG(bv.value * l.new + (1 - bv.value) * (1 - l.new))) >
      SUM(LOG(bv.value * l.old + (1 - bv.value) * (1 - l.old)))
    FROM logits l
    JOIN binary_values bv ON bv.id = l.id
  ) = 0 AND step > 0.000000000000000000000000000001 DO

    SET step = step / 2;
    CALL calculate_new_parameters(step);
    CALL calculate_logit();

  END WHILE;

  -- Ersetze die alten Werte durch die neuen Werte.
  UPDATE parameters
  SET parameters.old = parameters.new;

  UPDATE logits
  SET logits.old = logits.new;

  SET counter = counter + 1;

END WHILE;

-- Transformiere die Parameter linear, um den originalen Daten zu
↪  entsprechen.
UPDATE parameters
SET old = old / (max - min)
WHERE variable = 'beta_money';

SET transform = (SELECT old FROM parameters WHERE variable =
↪  'beta_money');

UPDATE parameters
SET old = old - transform * min
WHERE variable = 'alpha';

-- Gib eine Relation mit Parametername und zugehörigem Wert zurück.
SELECT variable, old AS `value`
FROM parameters;

-- Lösche die temporären Relationen wieder.
DROP TEMPORARY TABLE IF EXISTS datapoints;
DROP TEMPORARY TABLE IF EXISTS binary_values;
DROP TEMPORARY TABLE IF EXISTS parameters;
```

```
DROP TEMPORARY TABLE IF EXISTS logits;
DROP TEMPORARY TABLE IF EXISTS gradient;

END;;
DELIMITER ;
```

E. PostgreSQL-Skripte

E.1. Einfache lineare Regression

```sql
-- Erstelle (oder ersetze falls vorhanden) die Prozedur für einfache lineare Regression.
CREATE OR REPLACE FUNCTION simple_linear_regression(number_datapoints INTEGER)
RETURNS TABLE (
  variable VARCHAR(50),
  value NUMERIC(65, 30)
) AS $$
BEGIN

-- Erstelle eine Relation für die zu verwendenden Datenpunkte.
DROP TABLE IF EXISTS datapoints;
CREATE TEMPORARY TABLE datapoints (
  purchases INTEGER,
  money INTEGER
);

-- Füge die gewünschte Anzahl der Datenpunkte in die temporäre Relation ein.
INSERT INTO datapoints
SELECT purchases, money
FROM sample
LIMIT number_datapoints;

RETURN QUERY
WITH
  -- Berechne die Mittelwerte der abhängigen und unabhängigen Variable.
  means AS (
    SELECT
      AVG(purchases) AS mean_purchases,
      AVG(money) AS mean_money
    FROM datapoints
  ),
  -- Berechne die Summen im Nenner und Zähler der Formel für beta.
  sums AS (
    SELECT
```

```
      SUM((purchases - mean_purchases) * (money - mean_money)) AS
       ↪  nominator,
      SUM(POWER(purchases - mean_purchases, 2)) AS denominator
    FROM datapoints, means
  ),
  -- Berechne beta.
  beta AS (
    SELECT
      'beta'::VARCHAR(50) AS variable,
      nominator / denominator AS value
    FROM sums
  ),
  -- Berechne alpha.
  alpha AS (
    SELECT
      'alpha'::VARCHAR(50) AS variable,
      mean_money - beta.value * mean_purchases AS value
    FROM means, beta
  )
-- Gib eine Relation mit Parametername und zugehörigem Wert zurück.
SELECT *
FROM alpha
UNION
SELECT *
FROM beta;

-- Lösche die temporäre Relation mit den Datenpunkten wieder.
DROP TABLE IF EXISTS datapoints;

END;
$$ LANGUAGE plpgsql;
```

E.2. Multiple lineare Regression

```
-- Erstelle Funktion für die Berechnung der transponierten Matrix.
CREATE OR REPLACE FUNCTION matrix_transpose(a NUMERIC(65, 30)[][])
RETURNS NUMERIC(65, 30)[][] AS $$
DECLARE
  rows_a INTEGER := array_length(a, 1);
  columns_a INTEGER := array_length(a, 2);
  i INTEGER;
  j INTEGER;
  c NUMERIC(65, 30)[][];
  new_row NUMERIC(65, 30)[];
```

```
BEGIN

-- Iteriere über alle Zeilen und Spalten der ursprüglichen Matrix.
i := 1;
WHILE i <= columns_a LOOP

  j := 1;
  WHILE j <= rows_a LOOP
    -- Erzeuge ein Array mit der neuen Zeile der transponierten Matrix aus
    ↪  der Spalte der ursprünglichen Matrix.
    new_row[j] := a[j][i];
    j := j + 1;
  END LOOP;

  -- Füge die Zeile in die Ergebnismatrix ein.
  c := array_cat(c, array[new_row]);
  i := i + 1;
END LOOP;

RETURN c;

END;
$$ LANGUAGE plpgsql;

-- Erstelle Funktion für die Berechnung des Produktes zweier Matrizen.
CREATE OR REPLACE FUNCTION matrix_multiplication(a NUMERIC(65, 30)[][], b
↪  NUMERIC(65, 30)[][])
RETURNS NUMERIC(65, 30)[][] AS $$
DECLARE
  rows_a INTEGER := array_length(a, 1);
  columns_a INTEGER := array_length(a, 2);
  columns_b INTEGER := array_length(b, 2);
  new_row NUMERIC(65, 30)[];
  c NUMERIC(65, 30)[][];
  counter_1 INTEGER;
  counter_2 INTEGER;
  counter_3 INTEGER;
BEGIN

-- Iteriere über die Zeilen und Spalten der Ergebnismatrix.
counter_1 := 1;
WHILE counter_1 <= rows_a LOOP

  counter_2 := 1;
  WHILE counter_2 <= columns_b LOOP
```

```

    -- Initiiere den Wert des aktuellen Elementes der Ergebnismatrix mit 0.
    new_row[counter_2] := 0;

    -- Iteriere über die Summanden zur Berechnung des aktuellen Matrixelements.
    counter_3 := 1;
    WHILE counter_3 <= columns_a LOOP
      -- Addiere den aktuellen Summanden zum Wert des aktuellen Elementes.
      new_row[counter_2] := new_row[counter_2] + a[counter_1][counter_3] * b[counter_3][counter_2];
      counter_3 := counter_3 + 1;
    END LOOP;

    counter_2 := counter_2 + 1;
  END LOOP;

  c := array_cat(c, array[new_row]);
  counter_1 := counter_1 + 1;
END LOOP;

RETURN c;

END;
$$ LANGUAGE plpgsql;

-- Erstelle Funktion für die Berechnung der inversen Matrix.
CREATE OR REPLACE FUNCTION matrix_inversion(a NUMERIC(65, 30)[][])
RETURNS NUMERIC(65, 30)[][] AS $$
DECLARE
  n INTEGER := array_length(a, 1);
  p INTEGER := 0;
  i INTEGER;
  j INTEGER;
  c NUMERIC(65, 30)[][] := a;
  o NUMERIC(65, 30)[][];
BEGIN

-- Verwende den in der Arbeit referenzierten Algorithmus.
WHILE p < n LOOP

  p := p + 1;

```

```
  o := c;

  j := 1;
  WHILE j <= n LOOP
    IF j <> p THEN
      c[p][j] := c[p][j] / c[p][p];
    END IF;
    j := j + 1;
  END LOOP;

  i := 1;
  WHILE i <= n LOOP
    IF i <> p THEN
      c[i][p] := - c[i][p] / c[p][p];
    END IF;
    i := i + 1;
  END LOOP;

  i := 1;
  WHILE i <= n LOOP
    IF i <> p THEN
      j := 1;
      WHILE j <= n LOOP
        IF j <> p THEN
          c[i][j] := c[i][j] + o[p][j] * c[i][p];
        END IF;
        j := j + 1;
      END LOOP;
    END IF;
    i := i + 1;
  END LOOP;

  c[p][p] := 1 / c[p][p];

END LOOP;

RETURN c;

END;
$$ LANGUAGE plpgsql;

-- Erstelle die Funktion für multiple lineare Regression.
CREATE OR REPLACE FUNCTION multiple_linear_regression(number_datapoints
↪ INTEGER)
RETURNS TABLE (
```

```sql
  variable VARCHAR(50),
  value NUMERIC(65, 30)
) AS $$
DECLARE
  -- Erzeuge die Matrix X mit der gewünschten Anzahl an Datenpunkten.
  x INTEGER[][]:= (
    SELECT ARRAY(
      SELECT ARRAY[1, purchases, age]
      FROM sample
      LIMIT number_datapoints
    )
  );
  -- Erzeuge die Matrix y mit der gewünschten Anzahl an Datenpunkten.
  y INTEGER[]:= (
    SELECT ARRAY(
      SELECT ARRAY[money]
      FROM sample
      LIMIT number_datapoints
    )
  );
  b NUMERIC(65, 30)[][];
BEGIN

-- Berechne die Lösungsformel unter Verwendung der zuvor definierten
↪  Funktionen.
b := matrix_multiplication(
  matrix_inversion(
    matrix_multiplication(
      matrix_transpose(x),
      x
    )
  ),
  matrix_multiplication(
    matrix_transpose(x),
    y
  )
);

-- Gib eine Relation mit Parameternamen und zugehörigen Werten zurück.
RETURN QUERY
SELECT 'alpha'::VARCHAR(50) AS variable, b[1][1] AS value
UNION
SELECT 'beta_purchases'::VARCHAR(50) AS variable, b[2][1] AS value
UNION
SELECT 'beta_age'::VARCHAR(50) AS variable, b[3][1] AS value;
```

```
END;
$$ LANGUAGE plpgsql;
```

E.3. Logistische Regression

```
-- Erstelle (oder ersetzte falls vorhanden) eine Prozedur zur Berechnung
 ↪ der Werte der logistischen Funktion.
CREATE OR REPLACE FUNCTION calculate_logit()
RETURNS void AS $$
BEGIN

DELETE FROM logits;

WITH
  -- Bestimme den alten und neuen Wert von alpha.
  alpha_old AS (
    SELECT old
    FROM parameters
    WHERE variable = 'alpha'
  ),
  alpha_new AS (
    SELECT new
    FROM parameters
    WHERE variable = 'alpha'
  )
-- Berechne die Werte der logistischen Funktion für alle Datenpunkte.
INSERT INTO logits
  SELECT
    d.id,
    1 / (1 + EXP(- SUM(d.value * p.old) - (SELECT old FROM alpha_old))) AS
     ↪ old,
    1 / (1 + EXP(- SUM(d.value * p.new) - (SELECT new FROM alpha_new))) AS
     ↪ new
  FROM datapoints d
  JOIN parameters p ON p.variable = d.variable
  GROUP BY d.id;

RETURN;

END;
$$ LANGUAGE plpgsql;
```

```sql
-- Erstelle (oder ersetze falls vorhanden) eine Prozedur zur Berechnung
↪  des Gradienten.
CREATE OR REPLACE FUNCTION calculate_gradient()
RETURNS void AS $$
BEGIN

DELETE FROM gradient;

-- Berechne die partielle Ableitung nach alpha.
INSERT INTO gradient
SELECT 'alpha' AS variable, SUM(bv.value - l.old) AS value
FROM logits l
JOIN binary_values bv ON bv.id = l.id;

-- Berechne die partielle Ableitung nach allen beta-Parametern.
INSERT INTO gradient
SELECT d.variable, SUM(d.value * (bv.value - l.old)) AS value
FROM logits l
JOIN binary_values bv ON bv.id = l.id
JOIN datapoints d ON d.id = l.id
GROUP BY d.variable;

RETURN;

END;
$$ LANGUAGE plpgsql;

-- Erzeuge (oder ersetze falls vorhanden) eine Prozedur zur Berechnung der
↪  neuen Parameter abhängig von der aktuellen Schrittweite.
CREATE OR REPLACE FUNCTION calculate_new_parameters(step NUMERIC(65, 30))
RETURNS void AS $$
BEGIN

UPDATE parameters
SET new = old + step * gradient.value
FROM gradient
WHERE gradient.variable = parameters.variable;

RETURN;

END;
$$ LANGUAGE plpgsql;

-- Erstelle die Prozedur für logistische Regression.
```

```
CREATE OR REPLACE FUNCTION logistic_regression(number_datapoints INTEGER,
↪  rounds INTEGER, step NUMERIC(65, 30))
RETURNS TABLE (
  variable VARCHAR(50),
  value NUMERIC(65, 30)
) AS $$
DECLARE
  counter INTEGER;
BEGIN

-- Erstelle eine Relation für die Werte der unabhängigen Variablen.
DROP TABLE IF EXISTS datapoints;
CREATE TEMPORARY TABLE datapoints (
  id INTEGER,
  variable VARCHAR(50),
  value NUMERIC(65, 30)
);

-- Füge die linear transformierten Werte der unabhängigen Variablen in die
↪  Relation datapoints ein.
INSERT INTO datapoints
SELECT
  row_number() OVER () AS id,
  'beta_money' AS variable,
  (money - (
    SELECT MIN(money) FROM sample
  ))::NUMERIC(65, 30) / ((
    SELECT MAX(money) FROM sample
  ) - (
    SELECT MIN(money) FROM sample
  ))::NUMERIC(65, 30) AS value
FROM sample
LIMIT number_datapoints;

-- Erstelle eine Relation für die (binären) Werte der abhängigen
↪  Variablen.
DROP TABLE IF EXISTS binary_values;
CREATE TEMPORARY TABLE binary_values (
  id INTEGER,
  value INTEGER
);

-- Füge die Werte der abhängingen Variable ein.
INSERT INTO binary_values
SELECT
```

```
  row_number() OVER () AS id,
  premium AS value
FROM sample
LIMIT number_datapoints;

-- Erstelle eine Relation für die alten und neuen Parameterwerte.
DROP TABLE IF EXISTS parameters;
CREATE TEMPORARY TABLE parameters (
  variable VARCHAR(50),
  old NUMERIC(65, 30),
  new NUMERIC(65, 30)
);

-- Füge die Initialwerte der Parameter ein.
INSERT INTO parameters VALUES
  ('alpha', 0, 0),
  ('beta_money', 0, 0);

-- Erstelle eine Relation für die Werte der logistischen Funktion für alle
↪ Datenpunkte.
DROP TABLE IF EXISTS logits;
CREATE TEMPORARY TABLE logits (
  id INTEGER,
  old NUMERIC(65, 30),
  new NUMERIC(65, 30)
);

-- Befülle die Relation für die Werte der logistischen Funktion.
PERFORM calculate_logit();

-- Erstelle eine Relation für den Gradienten.
DROP TABLE IF EXISTS gradient;
CREATE TEMPORARY TABLE gradient (
  variable VARCHAR(50),
  value NUMERIC(65, 30)
);

-- Iteriere über die Anzahl der gewünschten Iterationen.
counter := 0;
WHILE counter < rounds AND step > 0.000000000000000000000000000001 LOOP

  -- Berechne den Gradienten und die neuen Parameter mit der aktuellen
  ↪ Schrittweite.
  PERFORM calculate_gradient();
  PERFORM calculate_new_parameters(step);
  PERFORM calculate_logit();
```

```sql

    -- Verringere die Schrittweite solange, bis die neuen Parameter ein besseres Ergebnis liefern als die alten.
    WHILE NOT (
      SELECT
        SUM(LOG(bv.value * l.new + (1 - bv.value) * (1 - l.new))) >
        SUM(LOG(bv.value * l.old + (1 - bv.value) * (1 - l.old)))
      FROM logits l
      JOIN binary_values bv ON bv.id = l.id
    ) AND step > 0.000000000000000000000000000001 LOOP

      step := step / 2;
      PERFORM calculate_new_parameters(step);
      PERFORM calculate_logit();

    END LOOP;

    -- Ersetze die alten Werte durch die neuen Werte.
    UPDATE parameters
    SET old = new;

    UPDATE logits
    SET old = new;

    counter := counter + 1;

  END LOOP;

  -- Transformiere die Parameter linear, um den originalen Daten zu entsprechen.
  UPDATE parameters
  SET old = old / ((SELECT MAX(money) FROM sample) - (SELECT MIN(money) FROM sample))
  WHERE parameters.variable = 'beta_money';

  UPDATE parameters
  SET old = old - (SELECT old FROM parameters p2 WHERE p2.variable = 'beta_money') * (SELECT MIN(money) FROM sample)
  WHERE parameters.variable = 'alpha';

  -- Gib eine Relation mit Parametername und zugehörigem Wert zurück.
  RETURN QUERY
  SELECT parameters.variable::VARCHAR(50), old AS value
  FROM parameters;

  -- Lösche die Relationen wieder.
```

```
DROP TABLE IF EXISTS datapoints;
DROP TABLE IF EXISTS binary_values;
DROP TABLE IF EXISTS parameters;
DROP TABLE IF EXISTS logits;
DROP TABLE IF EXISTS gradient;

END;
$$ LANGUAGE plpgsql;
```

F. Python-Skripte für das Benchmarking

F.1. Berechnung der Benchmarks

```
from time import time
from subprocess import call
import os
import sys
import csv
import json

# Erzeuge Funktion, um einen Fortschrittsbalken zu drucken.
def print_progessbar(iterations, progress):
  sys.stdout.write("\033[100D")
  for i in range(progress * 50 // iterations): sys.stdout.write("#")
  for i in range(50 - (progress * 50 // iterations)): sys.stdout.write(".")
  sys.stdout.write("  ||  %i%% / 100%%" % (progress * 100 // iterations))
  sys.stdout.flush()

# Erzeuge Funktion, die eine bestimme Anzahl an Laufzeiten für eine bestimmte Anzahl an Datenpunkten für eine bestimme Art der Regression in einer bestimmten Sprache berechnet.
def benchmark(type_regression, language, command, set_number_datapoints, file, iterations):
  print("Evaluating %s in %s..." % (type_regression, language))

  # Iteriere über ein übergenenes Array mit den Anzahlen der zu verwendenden Datenpunkte.
  for number_datapoints in set_number_datapoints:
    print("Use %s datapoints:" % (number_datapoints))

    # Erzeuge eine Datei, in die die Ausgaben aus stdout und stderr geschrieben werden.
    logfile = open("logs/%s-%s-%i.txt" % (language, type_regression.replace(" ", "-"), number_datapoints), "a")

    # Füge die Anzahl der Datenpunkte in den Kommandozeilen-Befehl ein.
    if command.count("%") == 2:
      exec_command = command % (number_datapoints, 8 / number_datapoints)
    else:
```

```
      exec_command = command % number_datapoints

    # Iteriere über die Anzahl der gewünschten Iterationen.
    for i in range(iterations):
      # Drucke den Fortschrittsbalken
      print_progessbar(iterations, i)

      # Speichere die Startzeit.
      start_time = time()

      # Führe den Kommandozeilen-Befehl aus.
      call(exec_command, stdout = logfile, stderr = logfile, shell = True)

      # Speichere die Endzeit.
      end_time = time()

      # Füge eine neue Zeile in der Benchmark-Datei ein.
      file.write("%s,%s,%i,%s\n" % (
        language,
        type_regression,
        number_datapoints,
        (end_time - start_time)
      ))

    # Schließe die Log-Datei.
    logfile.close()

    # Drucke 100% im Forschrittbalken und eine leere Zeile.
    print_progessbar(iterations, iterations)
    print("")
  return

def main(argv):
  # Bestimme default-Werte.
  iterations = 100
  set_number_datapoints = [10, 100, 1000, 10000, 100000]
  run_r = False
  run_tensorflow = False
  run_mysql = False
  run_postgresql = False
  run_simple_linear_regression = False
  run_multiple_linear_regression = False
  run_logistic_regression = False

  # Durchlaufe die übergebenen Argumente.
  for arg in argv:
```

```python
    # Fixiere die Anzahl der Datenpunkte.
    if "--datapoints=" in arg: set_number_datapoints = [int(arg.split("=")[1])]

    # Setzte die Anzahl der Iterationen.
    if "--iterations=" in arg: iterations = int(arg.split("=")[1])

    # Berechne Benchmarks für R.
    if arg in ["--r", "-r"]: run_r = True

    # Verwende TensorFlow.
    if arg in ["--tensorflow", "-t"]: run_tensorflow = True

    # Berechne Benchmarks für MySQL.
    if arg in ["--mysql", "-m"]: run_mysql = True

    # Berechne Benchmarks für PostgreSQL.
    if arg in ["--postgresql", "-p"]: run_postgresql = True

    # Berechne Benchmarks für einfache lineare Regression.
    if arg in ["--simple-linear", "-slr"]: run_simple_linear_regression = True

    # Berechne Benchmarks für multiple lineare Regression.
    if arg in ["--multiple-linear", "-mlr"]: run_multiple_linear_regression = True

    # Berechne Benchmarks für logistische Regression.
    if arg in ["--logistic", "-lr"]: run_logistic_regression = True

  # Wenn keine Art der Regression und keine Sprache spezifiziert wurde, berechne alles.
  if not ((
    run_r or
    run_tensorflow or
    run_mysql or
    run_postgresql
  ) and (
    run_simple_linear_regression or
    run_multiple_linear_regression or
    run_logistic_regression
  )):
    run_r = True
    run_tensorflow = True
    run_mysql = True
    run_postgresql = True
```

```python
    run_simple_linear_regression = True
    run_multiple_linear_regression = True
    run_logistic_regression = True

  # Erzeuge eine Datei für die berechneten Laufzeiten und schreibe die erste Zeile.
  file = open("benchmarks-%i.csv" % (time()), "w")
  file.write("language,type,datapoints,time\n")

  # Berechne einfache lineare Regression in R.
  if run_r and run_simple_linear_regression:
    benchmark(
      "simple linear regression",
      "r",
      "Rscript r/simpleLinearRegression.R %i -",
      set_number_datapoints,
      file,
      iterations
    )

  # Berechne multiple lineare Regression in R.
  if run_r and run_multiple_linear_regression:
    benchmark(
      "multiple linear regression",
      "r",
      "Rscript r/multipleLinearRegression.R %i -",
      set_number_datapoints,
      file,
      iterations
    )

  # Berechne logistische Regression in R.
  if run_r and run_logistic_regression:
    benchmark(
      "logistic regression",
      "r",
      "Rscript r/logisticRegression.R %i -",
      set_number_datapoints,
      file,
      iterations
    )

  # Berechne einfache lineare Regression in TensorFlow.
  if run_tensorflow and run_simple_linear_regression:
    benchmark(
      "simple linear regression",
```

```
      "tensorflow",
      "python3 tensorflow/simpleLinearRegression.py %i -",
      set_number_datapoints,
      file,
      iterations
    )

  # Berechne multiple lineare Regression in TensorFlow.
  if run_tensorflow and run_multiple_linear_regression:
    benchmark(
      "multiple linear regression",
      "tensorflow",
      "python3 tensorflow/multipleLinearRegression.py %i -",
      set_number_datapoints,
      file,
      iterations
    )

  # Berechne logistische Regression in TensorFlow.
  if run_tensorflow and run_logistic_regression:
    benchmark(
      "logistic regression",
      "tensorflow",
      "python3 tensorflow/logisticRegression.py %i -",
      set_number_datapoints,
      file,
      iterations
    )

  # Berechne einfache lineare Regression in MySQL.
  if run_mysql and run_simple_linear_regression:
    benchmark(
      "simple linear regression",
      "mysql",
      "echo \"CALL regression.simple_linear_regression(%i)\" | " + "mysql
      ↪  -u %s -p%s" % (
        json.loads(os.environ["MYSQL_CONFIG"])["user"],
        json.loads(os.environ["MYSQL_CONFIG"])["password"]
      ),
      set_number_datapoints,
      file,
      iterations
    )

  # Berechne multiple lineare Regression in MySQL.
  if run_mysql and run_multiple_linear_regression:
```

```
    benchmark(
      "multiple linear regression",
      "mysql",
      "echo \"CALL regression.multiple_linear_regression(%i)\" | " +
       ↪  "mysql -u %s -p%s" % (
        json.loads(os.environ["MYSQL_CONFIG"])["user"],
        json.loads(os.environ["MYSQL_CONFIG"])["password"]
      ),
      set_number_datapoints,
      file,
      iterations
    )

  # Berechne logistische Regression in MySQL.
  if run_mysql and run_logistic_regression:
    benchmark(
      "logistic regression",
      "mysql",
      "echo \"CALL regression.logistic_regression(%i, 1000, %f)\" | " +
       ↪  "mysql -u %s -p%s" % (
        json.loads(os.environ["MYSQL_CONFIG"])["user"],
        json.loads(os.environ["MYSQL_CONFIG"])["password"]
      ),
      set_number_datapoints,
      file,
      iterations
    )

  # Berechne einfache lineare Regression in PostgreSQL.
  if run_postgresql and run_simple_linear_regression:
    benchmark(
      "simple linear regression",
      "postgresql",
      "echo \"SELECT simple_linear_regression(%i)\" | psql regression",
      set_number_datapoints,
      file,
      iterations
    )

  # Berechne multiple lineare Regression in PostgreSQL.
  if run_postgresql and run_multiple_linear_regression:
    benchmark(
      "multiple linear regression",
      "postgresql",
      "echo \"SELECT multiple_linear_regression(%i)\" | psql regression",
      set_number_datapoints,
```

```
      file,
      iterations
    )

  # Berechne logistische Regression in PostgreSQL.
  if run_postgresql and run_logistic_regression:
    benchmark(
      "logistic regression",
      "postgresql",
      "echo \"SELECT logistic_regression(%i, 1000, %f)\" | psql regression",
      set_number_datapoints,
      file,
      iterations
    )

  # Schließe die Benchmark-Datei.
  file.close()
  return

# Führe die main-Funktion aus.
if __name__ == "__main__":
  main(sys.argv)
```

F.2. Auswertung der Benchmarks

```
import sys
import os.path as p
import csv
import matplotlib.pyplot as plt

# Erzeuge Funktion, die die Benchmarks aus der csv-Datei einliest un gruppiert.
def calculate_benchmarks():
  # Bestimme den Dateipfad der benchmark-Datei und öffne diese.
  filename = p.abspath(p.join(p.dirname(p.realpath(__file__)), "benchmarks.csv"))
  csvfile = open(filename, newline="")
  csvreader = csv.reader(csvfile, delimiter=",", quotechar="|")

  # Erstelle ein dictionary, in dem die Laufzeiten aggregiert werden sollen.
  count = {
    "simple linear regression": {
      "10": {
```

```
        "r": {"count": 0, "time": 0},
        "tensorflow": {"count": 0, "time": 0},
        "mysql": {"count": 0, "time": 0},
        "postgresql": {"count": 0, "time": 0}
      },
      "100": {
        "r": {"count": 0, "time": 0},
        "tensorflow": {"count": 0, "time": 0},
        "mysql": {"count": 0, "time": 0},
        "postgresql": {"count": 0, "time": 0}
      },
      "1000": {
        "r": {"count": 0, "time": 0},
        "tensorflow": {"count": 0, "time": 0},
        "mysql": {"count": 0, "time": 0},
        "postgresql": {"count": 0, "time": 0}
      },
      "10000": {
        "r": {"count": 0, "time": 0},
        "tensorflow": {"count": 0, "time": 0},
        "mysql": {"count": 0, "time": 0},
        "postgresql": {"count": 0, "time": 0}
      },
      "100000": {
        "r": {"count": 0, "time": 0},
        "tensorflow": {"count": 0, "time": 0},
        "mysql": {"count": 0, "time": 0},
        "postgresql": {"count": 0, "time": 0}
      }
    },
    "multiple linear regression": {
      "10": {
        "r": {"count": 0, "time": 0},
        "tensorflow": {"count": 0, "time": 0},
        "mysql": {"count": 0, "time": 0},
        "postgresql": {"count": 0, "time": 0}
      },
      "100": {
        "r": {"count": 0, "time": 0},
        "tensorflow": {"count": 0, "time": 0},
        "mysql": {"count": 0, "time": 0},
        "postgresql": {"count": 0, "time": 0}
      },
      "1000": {
        "r": {"count": 0, "time": 0},
        "tensorflow": {"count": 0, "time": 0},
```

```
        "mysql": {"count": 0, "time": 0},
        "postgresql": {"count": 0, "time": 0}
      },
      "10000": {
        "r": {"count": 0, "time": 0},
        "tensorflow": {"count": 0, "time": 0},
        "mysql": {"count": 0, "time": 0},
        "postgresql": {"count": 0, "time": 0}
      },
      "100000": {
        "r": {"count": 0, "time": 0},
        "tensorflow": {"count": 0, "time": 0},
        "mysql": {"count": 0, "time": 0},
        "postgresql": {"count": 0, "time": 0}
      }
    },
    "logistic regression": {
      "10": {
        "r": {"count": 0, "time": 0},
        "tensorflow": {"count": 0, "time": 0},
        "mysql": {"count": 0, "time": 0},
        "postgresql": {"count": 0, "time": 0}
      },
      "100": {
        "r": {"count": 0, "time": 0},
        "tensorflow": {"count": 0, "time": 0},
        "mysql": {"count": 0, "time": 0},
        "postgresql": {"count": 0, "time": 0}
      },
      "1000": {
        "r": {"count": 0, "time": 0},
        "tensorflow": {"count": 0, "time": 0},
        "mysql": {"count": 0, "time": 0},
        "postgresql": {"count": 0, "time": 0}
      },
      "10000": {
        "r": {"count": 0, "time": 0},
        "tensorflow": {"count": 0, "time": 0},
        "mysql": {"count": 0, "time": 0},
        "postgresql": {"count": 0, "time": 0}
      },
      "100000": {
        "r": {"count": 0, "time": 0},
        "tensorflow": {"count": 0, "time": 0},
        "mysql": {"count": 0, "time": 0},
        "postgresql": {"count": 0, "time": 0}
```

```
      }
    }
  }

  # Iteriere über alle Zeilen der csv-Datei.
  for row in csvreader:
    # Überspringe die erste Zeile.
    if not row[0] == "language":
      # Füge die Laufzeit in das dictionary ein.
      count[row[1]][row[2]][row[0]]["count"] += 1
      count[row[1]][row[2]][row[0]]["time"] += float(row[3])

  # Durchlaufe das dictionary (Typ der Regression).
  for regression_type, obj1 in count.items():
    print("benchmarks for %s:\n" % regression_type)

    # Erzeuge ein neues dictionary, um eine Tabelle zu drucken.
    table = {"header": [""], "r": ["r"], "tensorflow": ["tensorflow"], "mysql": ["mysql"], "postgresql": ["postgresql"]}

    # Iteriere über die im ersten dictionary enthalten dictionaries (Anzahl Datenpunkte).
    for number_datapoints, obj2 in obj1.items():
      # Ergänze die Anzahl der Datenpunkte als Spaltenbeschriftung.
      table["header"].append(str(number_datapoints))

      # Iteriere über die im zweiten dictionary enthalten dictionaries (Sprache).
      for language, data in obj2.items():
        # Füge die durchschnittliche Laufzeit in die Tabelle ein (falls Lafzeiten vorhanden sind).
        if data["count"] > 0:
          table[language].append(str(data["time"] / data["count"])[:10])
        else:
          table[language].append("          ")

    # Erzeuge die zu druckende Tabelle zeilenweise.
    print_table = [
      "|" + "-" * 89 + "|",
      "|" + "  %s            |  %s          |  %s         |  %s        |  %s       |  %s      " % tuple(table["header"]) + "|",
      "|" + "-" * 89 + "|",
      "|" + "  %s           |  %s  |  %s  |  %s  |  %s  |  %s   " % tuple(table["r"]) + "|",
      "|" + "-" * 89 + "|",
```

```
      "|" + "  %s  |  %s  |  %s  |  %s  |  %s  |  %s  " % tuple(table["tensorflow"]) + "|",
      "|" + "-" * 89 + "|",
      "|" + "  %s      |  %s  |  %s  |  %s  |  %s  |  %s  " % tuple(table["mysql"]) + "|",
      "|" + "-" * 89 + "|",
      "|" + "  %s  |  %s  |  %s  |  %s  |  %s  |  %s  " % tuple(table["postgresql"]) + "|",
      "|" + "-" * 89 + "|"
    ]

    # Drucke die Tabelle und eine Leerzeile danach.
    print(str.join("\n", print_table))
    print("\n")

  # Gib das dictionary mit allen aggregierten Laufzeiten zurück.
  return count

# Erzeuge Funktion, um die Laufzeiten für eine bestimmte Art der Regression zu plotten.
def plot(benchmarks, regression_type, plot_title):
  # Definiere ein Array mit den Anzahlen der Datenpunkte für den Plot.
  x = [10, 100, 1000, 10000, 100000]

  # Erzeuge ein dictionary mit leeren Arrays für die durchschnittlichen Laufzeiten.
  values = {
    "r": [],
    "tensorflow": [],
    "mysql": [],
    "postgresql": []
  }

  # Durchlaufe alle Sprachen.
  for language in ["r", "tensorflow", "mysql", "postgresql"]:
    # Durchlaufe die Anzahl der Datenpunkte.
    for number_datapoints in x:
      # Füge die durchschnittliche Laufzeit in das Array ein (falls Laufzeiten vorhanden sind).
      if benchmarks[regression_type][str(number_datapoints)][language]["count"] > 0:
        values[language].append(
          benchmarks[regression_type][str(number_datapoints)][language]["time"] /
```

```
          benchmarks[regression_type][str(number_datapoints)][language]["count"]
        )
      else:
        values[language].append(None)

  # Erzeuge den Plot.
  plt.loglog(x, values["r"], "r-", label="R")
  plt.loglog(x, values["tensorflow"], "y-", label="TensorFlow")
  plt.loglog(x, values["mysql"], "b-", label="MySQL")
  plt.loglog(x, values["postgresql"], "g-", label="PostgreSQL")
  plt.title(plot_title)
  plt.legend()
  plt.xlabel("Anzahl der Datenpunkte")
  plt.ylabel("Laufzeit in Sekunden")
  plt.show()

def main(argv):
  # Erzeuge default-Wert, ob Plots erstellt werden sollen.
  print_plots = True

  # Überschreibe default-Wert, falls ein entsprechendes Argument übergeben wurde.
  if len(argv) == 2:
    if argv[1] == "-":
      print_plots = False

  # Berechne und drucke die Benchmarks.
  benchmarks = calculate_benchmarks()

  # Plote die Benchmarks (falls gewüscht).
  if print_plots:
    plot(benchmarks, "simple linear regression", "Einfache lineare Regression")
    plot(benchmarks, "multiple linear regression", "Multiple lineare Regression")
    plot(benchmarks, "logistic regression", "Logistische Regression")

# Führe die main-Funktion aus.
if __name__ == "__main__":
  main(sys.argv)
```

Bibliografie

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

[2] Ludwig Fahrmeir. *Statistik*. Springer Spektrum, 2016.

[3] Trevor John Hastie John Mckinley Chambers. *Statistical models in S*. Wadsworth & Brooks/Cole, 1992.