MARTIN-LUTHER
UNIVERSITÄT
HALLE-WITTENBERG

Masterarbeit zum Thema

SQL2Stan: automatische Inferenz von Machine-Learning-Modellen für Datenbanken

Dmitry Trofimov
Matrikelnummer: 213222170

Erstgutachter: Dr. Alexander Hinneburg
Zweitgutachter: Prof. Dr. Matthias Hagen

Sommersemester 2019
Abgabetermin: 25.04.2019

Kurzfassung

Eine der Herausforderungen der Analyse von Big Data liegt in der Verknüpfung der Bereiche Datenmanagement und Machine-Learning-Algorithmen. Für eine Disziplin des maschinellen Lernens – Bayes'sche Inferenz – lässt sich diese Verknüpfung mit Hilfe von relationalen Schemata und SQL bewerkstelligen. Diese aus den Datenbanken bekannten Konzepte können dazu genutzt werden, die Bayes'sche Inferenz deklarativer, abstrakter und einfacher zu gestalten. Durch SQL werden fachübergreifend viele Domänenspezialisten angesprochen, die keine zusätzliche Sprache lernen müssen, um anhand eines gegebenen Bayes'schen Modells ihrer Wahl etwas Unbekanntes aus ihren Daten zu inferieren. Die „high-level"-Abstraktion, die SQL und relationale Schemata bieten, steigert das Potenzial für automatische Optimierung beim Ausrechnen der Inferenzprobleme. Außerdem stellt sie mehr technische Flexibilität in Aussicht, indem sie Komponenten und Algorithmen, die vor dem Programmierer dadurch verborgen bleiben, austauschbar macht.

Die bisher vorgestellten Projekte, die die Bayes'sche Inferenz in den Datenbankkontext einbinden, weisen unterschiedliche verbesserungsbedürftige Aspekte auf. Einer dieser Aspekte ist die nutzerseitige Spezifikation von beliebigen Bayes'schen Modellen. Einige Systeme erlauben bei der Modellspezifikation keine hierarchischen Modelle (z.B. LDA), andere Systeme bieten gar keine nutzerseitige Spezifikation Bayes'scher Modelle an (ersetzt durch vorimplementierte Modelle oder automatische Modellgenerierung). Derartige Einschränkungen sind nicht optimal. Wissenschaftliche Veröffentlichungen stellen eine große Menge an domänenspezifischen Bayes'schen Modellen bereit. Es ist von Vorteil für Domänenspezialisten, wenn sie relevante Bayes'sche Modelle unaufwändig nachbauen und damit Bayes'sche Inferenz durchführen können. Das wissenschaftlich dokumentierte Überführen von Bayes'schen Modellen in den Datenbankkontext kann dabei hilfreich sein; an diesen Erkenntnissen knüpft dennoch keines der bisher vorgestellten Projekte an. Außerdem beseitigt die erwähnte Übersetzung von Bayes'schen Modellen in den Datenbankkontext einen weiteren Kritikpunkt: die nicht immer gegebene Entkopplung vom Verständnis der Inferenz-Ein- und Ausgabe vom Verständnis der Modellstruktur und der Inferenzalgorithmen. Der SQL-Programmierer, der ein Bayes'sches Modell formuliert und statistische Inferenz durchführen lässt, ist nicht immer die gleiche Person, die die Inferenzergebnisse über eine Datenbankschnittstelle in eine Anwendung einbindet. Ganz gleich, welche der beiden Rollen der Nutzer einnimmt – seine Interaktion mit dem System soll für ihn möglichst verständlich sein. Das ist in den bisher vorgestellten Systemen nicht immer der Fall.

Ein weiterer kritischer Aspekt der bisher vorgestellten Projekte für Bayes'sche Inferenz im Datenbankkontext ist Austauschbarkeit von den beteiligten Komponen-

ten (sowohl an der Seite des Datenmanagements als auch beim Inferenz-Backend). Eine derartige technologische Flexibilität ist bei Systemen mit deklarativen Programmiersprachen (v.a. SQL) möglich, und im Angesicht der voranschreitenden akademischen Entwicklungen im Datenmanagement- und im Machine-Learning-Bereich sogar wünschenswert. Dennoch wird sie von den existierenden Projekten nicht adressiert, und ist somit entweder nicht möglich oder nur schwer zu bewältigen. Außerdem, der technologischen Flexibilität sowie der Verständlichkeit der Programmiersprache zugute, soll eine deklarative Programmiersprache zum Spezifizieren Bayes'scher Modelle (benötigt für die Inferenz) nicht von den Details konkreter Inferenzalgorithmenklassen (z.B. Markov-Chain-Monte-Carlo) abhängen. Manche vielversprechende Softwareprojekte sind leider sowohl technologisch als auch sprachlich in einer einzigen Klasse von Inferenzalgorithmen verankert.

Einer der Beiträge dieser Arbeit ist ein Prototyp namens SQL2Stan, der die oben genannten Schmerzpunkte beseitigt. In dessen Rahmen wurde ein SQL-Dialekt entworfen, mit dem Bayes'sche Modelle (auch hierarchischer Art) beschrieben werden können. Der neue SQL-Dialekt knüpft, anders als alle bisher vorgestellten Systeme, an die Überführung von Bayes'schen Netzen (grafische Darstellung von Bayes'schen Modellen) in relationale Schemata an. Diese theoretische Grundlage vereinfacht die statistische Modellierung in SQL und konzipiert zugleich eine klare DBMS-seitige Schnittstelle für die Ein- und Ausgabe der Inferenz. Der SQL2Stan-Prototyp präsentiert an drei Modellbeispielen, dass SQL als Programmiersprache genügen kann, um die statistische Inferenz mit nutzerdefinierten Bayes'schen Modellen in das Datenmanagement einzubinden und zu automatisieren. SQL2Stan bietet eine relational-orientierte SQL-Abstraktion sowohl für das Datenmanagement als auch für Bayes'sche Inferenz, und erzielt dadurch folgende Vorteile:

1. Bayes'sche Inferenz wird durch SQL zugänglicher für Domänenspezialisten mit wenig Erfahrung im Programmieren und statistischer Analyse.

2. Die Spezifikation eigener Bayes'scher Modelle wird erleichtert, aber nicht vollständig wegabstrahiert. Das ermöglicht unaufwändiges Nachbauen von den bisher veröffentlichten Bayes'scher Modellen für unterschiedlichste Anwendungsbereiche. Hierarchische Modelle werden unterstützt.

3. Die Interfacesprache ist reines SQL. Sie benötigt – anders als bei allen anderen ML-Projekten mit SQL – keinen speziellen Parser.

4. Der Compiler verfügt über viel Freiheit zur automatischen Optimierung, weil der nutzergeschriebene SQL-Code abstrakt und deklarativ gehalten wird. Der nutzergeschriebene Code wird in zwei Stufen übersetzt (SQL nach Stan nach C++), und kann an jeder Stufe automatisch optimiert werden.

5. Die Inferenzsoftware und die Inferenzalgorithmen sind austauschbar, ohne dass die nutzergeschriebene Spezifikation geändert werden muss. Konkret: anstatt No-U-Turn-Sampler kann in SQL2Stan leicht ein anderer, schnellerer Inferenzalgorithmus namens ADVI (Automatic Differentiation Variational Inference) eingesetzt werden, ohne dass der SQL-Code geändert werden muss.

6. Das Datenmanagementsystem ist austauschbar. Da die Tabellen und Spalten schließlich nur eine Abstraktion sind, muss im Hintergrund keineswegs ein klassisches zeilenorientiertes relationales Datenbankmanagementsystem (RDBMS) genutzt werden. Eine in dieser Arbeit vorgestellte Idee zum Abbilden von Tabellenspaltenbeschreibungen auf multidimensionale Arrays ermöglicht den Einsatz von Array-DBMS zwecks physischer Datenverwaltung.

Abgesehen vom Entwurf eines SQL-Dialekts zur Bayes'schen Modellierung gehört zum Beitrag dieser Arbeit ein Konzept zur automatischen Übersetzung vom SQL-Code in eine probabilistische Programmiersprache Stan. Durch die Übersetzung von SQL nach Stan wird eine Implementierung für Bayes'sche Inferenz generiert. Die Effizienz des automatisch generierten Stan-Codes ist vergleichbar mit der des handgeschriebenen Stan-Codes (experimentelle Vergleiche vorhanden), der von Stan-Entwicklern für drei ausgewählte Beispielmodelle veröffentlicht wurde.

Related Work (ausführliche Version im Anhang) mit einer Zusammenfassung über Veröffentlichungen zum Gegenstand der Zusammenführung vom maschinellen Lernen und Datenmanagement ist der letzte Beitrag dieser Arbeit. Diese Übersicht thematisiert dabei insbesondere die Abstraktion als eine erkennbare Tendenz in den Programmiersprachen relevanter Projekte.

Zum Abschluss der Zusammenfassung und als Vorwort zum Hauptteil sollte erwähnt werden, dass diese Arbeit (ohne Anhänge) über 100 Seiten hinausläuft. Das liegt teils am Seitenspiegel, jedoch vor allem am Thema der Arbeit: es geht um den Entwurf eines SQL-Dialekts sowie um seine Übersetzung in eine andere Sprache. Diese Aspekte müssen unabdinglich mit vielen erklärenden Diagrammen und Code-Beispielen belegt werden, die den Hauptteil der Arbeit sehr groß werden lassen.

Inhaltsverzeichnis

1. Motivation und Einleitung

In vier Sätzen: Lücke in der Forschung Datenmanagement sowie Bayes'sche Inferenz (ein Bereich von Machine Learning) lassen sich mit Hilfe von SQL sprachlich auf der gleichen Abstraktionsebene zusammenführen. Das bringt zwei Vorteile: Bedienungseinfachheit für SQL-Programmierer sowie bessere technologische Flexibilität (Backend-Komponenten austauschbar). Die bisher vorgestellten Systeme weisen inhärenten Verbesserungsbedarf im Hinblick auf diese Aspekte auf. Diese Arbeit präsentiert einen Verbesserungsansatz: darin wird ein Entwurf eines neuen SQL-Dialekts zur Bayes'schen Modellspezifikation vorgestellt, bekräftigt durch einen Prototypen zur Einbindung diesen Dialekts in ein Datenbanksystem.

Big Data und maschinelles Lernen Die Zusammenführung von Machine Learning (ML) und dem skalierbaren Datenmanagement wurde in den letzten Jahren immer wieder angesprochen ([Jan+19; Tam+05; Fen+12; Hel+12; Kra+13; KNP15; KBY17; BBM18; Kar+18a] und viele weitere Veröffentlichungen). Einen starken Ansporn dazu gibt die zunehmende Wichtigkeit, immense Mengen an Daten – Big Data – zu analysieren. Maschinelles Lernen profitiert von großen Massen an Daten: je mehr verwertbare Daten, umso nützlicher die Ergebnisse ihrer Verarbeitung. Zum Beispiel, ML-Modelle, die mit mehr Daten trainiert wurden, sind besser als Modelle, die auf kleineren Datensätzen trainiert wurden [1]. Oftmals passen die vielen Daten, die man in ML-Workflows verarbeiten möchte, nicht mehr in den Hauptspeicher eines Rechners und müssen daher verteilt in einem Rechnercluster untergebracht werden. Mit dem Wunsch, mehr Daten in Analyseprozesse einzupflegen, kommt der Bedarf an neuen, robusten Methoden und Werkzeugen, die die statistische bzw. ML-gestützte Datenanalyse auch auf großen Datenmengen ermöglichen [LM12, S. 1]. Datenmanagementsysteme (kurz: DMS) kommen als Infrastruktur zum Unterbringen von Big Data infrage, weil sie sich auf die skalierbare Verwaltung von großen Datenmengen spezialisieren. Somit erscheint es nachvollziehbar, dass maschinelles Lernen eine Anknüpfung an DMS braucht. Direkt hervorzuheben ist, dass in dieser Arbeit um Bayes'sche Inferenz gehen wird, als ein relevanter Teil von ML, der mit Hilfe eines neuen Ansatzes mit DMS verknüpft wird.

Ansätze zur Verknüpfung von DMS und ML Es gibt zwei mögliche Ansätze zum verknüpfen von DMS und ML: entweder die Datenanalytik in das Datenmanagement einbetten, oder alternativ die Datenanalyse-Software um Datenmanagement--Funktionalität erweitern. Beide Ansätze sind nicht vielversprechend [LM12, S. 2].

[1] Vorausgesetzt, die Güte der Datensätze ist ausreichend und homogen.

Einerseits, ein deklarativ zu bedienendes DBMS mit eingebauten flexiblen und leicht zugänglichen Machine-Learning-Werkzeugen wäre aus der Sicht der Benutzerfreundlichkeit ein Segen, denn viele DBMS-Nutzer würden davon profitieren. Andererseits ist es eher unrealistisch zu erwarten, dass eine herkömmliche Datenbank (z.B. MySQL, PostgreSQL oder Oracle DB) so weit erweitert wird, dass seine interne Umsetzung der Datenanalytik ernsthaft mit statistischen Schwergewichten wie MATLAB konkurrieren könnte; laut [Wan+18, S. 1] sei dies fast unmöglich. Solch eine Datenbank müsste außerdem stets mit dem neuesten Know-How aus dem maschinellen Lernen mitziehen. Die Wissenschaftler verbessern unaufhörlich alte Verfahren bzw. entwerfen neue ML-Algorithmen (oft komplex konzipiert und nicht trivial implementierbar), die man in diesem Falle ins eigene Datenmanagementsystem integrieren sollte, um es möglichst attraktiv und innovativ zu gestalten.

Von der anderen Seite gesehen handelt es sich um die Anknüpfung von Datenmanagement an ausgewachsene ML-Systeme. Diese Anknüpfung kann ebenso nicht ohne viel Aufwand umgesetzt werden. Von beispielsweise MATLAB oder Stan dürfte man kaum erwarten, dass diese datenanalytischen Softwareprodukte ein konkurrenzfähiges, robustes und skalierbares Datenmanagement von Haus aus anbieten würden [LM12, S. 2]. Darüber hinaus müsste der Softwareanbieter stets die neuen, sinnvollen Datenbankkonzepte nachimplementieren, die die Forschung mit sich bringt (beispielsweise verteilte Array-DBMS oder Lösungen zur verteilten physikalischen Verwaltung für tensorbasierte Anwendungen). Man könnte darauf spekulieren, dass Systeme, welche nur ML-Algorithmen ohne Datenmanagement, Datenaufarbeitung und Modellserving anbieten, in kommender Zeit nach und nach von anderen Systemen verdrängt werden, die sich besser für Big-Data-Analyse eignen.

Eine umfangreiche Erweiterung von DMS- oder ML-Systemen erscheint also unrealistisch. Das größte Potenzial besitzen vielmehr Systeme, die die ML- und Datenmanagementaspekte nicht *ineinander*, sondern *miteinander* integrieren, damit die Stärken und Fortschritte beider Spezialbereiche ausgenutzt werden können [LM12]. Aus der technischen Sicht ist diese Art der Integration nicht einfach zu bewerkstelligen. Von dieser Komplexität, die hinter der Zusammenarbeit der Komponenten steckt, soll der Programmierer so wenig wie möglich wahrnehmen. Nur so bleibt das Benutzerinterface einfach und zugänglich. Es läuft somit darauf hinaus, dass ML- und DMS-Aspekte in einer gemeinsamen deklarativen Programmiersprache unter einen Hut gebracht werden sollen. Diese Benutzerinterface-Sprache soll nur das Notwendige – z.B. die Spezifikation einer ML-Rechenaufgabe – offenbaren und vor dem Programmierer möglichst viele andere, komplexe Details wegabstrahieren.

Das Finden einer gemeinsamen Sprache für beide ML und DMS ist somit eine der Fragestellungen bei ihrer integrativen Zusammenführung. Zwischen den Sprachen dieser beiden Bereiche besteht eine klare Diskrepanz. ML und DMS bedienen sich unterschiedlicher Programmiersprachen aus unterschiedlichen Programmierparadigmen, blenden verschiedenartige Details aus und konzentrieren sich wiederum auf ungleiche Aspekte. Ein möglicher Ansatz zur Überwindung dieser programmiersprachlichen Uneinigkeit besteht darin, eine high-level-Sprache im Benutzerinterface anzubieten, die sowohl ML- als auch DMS-Aspekte berücksichtigt. Die

genauen technischen Details zur Zusammenführung der Komponenten werden vor dem Nutzer wegabstrahiert. Sowohl DMS- als auch ML-Systeme bieten ihrerseits high-level-Interfacesprachen an, wovon SQL am attraktivsten erscheint.

Probabilistische Programmierung als Programmierparadigma Diese Arbeit beschäftigt sich mit einer Gattung von Machine Learning – der Bayes'schen Modellierung und der damit verbundenen statistischen Inferenz. Dieser Paragraph bietet einen kleinen Einblick in die high-level-Sprachen dieser ML-Sparte.

Bei der Bayes'schen Modellierung handelt es sich darum, dass man die Realität, repräsentiert durch gegebene Daten, auf ein Modell der Bayes'schen Stastistik abbildet. Dazu werden statistische Mittel (hauptsächlich Wahrscheinlichkeitsverteilungen und Zufallsvariablen) genutzt. Statistischen Modelle inkorporieren das statistische Wissen nicht nur über die beobachtbare Realität, sondern auch über die latenten (versteckten) Erkenntnisse, die den beobachtbaren Daten zugrunde liegen. Die Bayes'sche statistische Modellierung ist, um es einfacher auszudrücken, das Modellieren von dem Bekannten und dem Unbekannten und das Kombinieren ihrer mit Wahrscheinlichkeiten. Sie findet Anwendung in vielen wissenschaftlichen und industriellen Bereichen. Statistische (Bayes'sche) Inferenz ist eine Methode, um von einem Modell sowie einem Datensatz ausgehend an unbekannte Werte von latenten (unbeobachteten) Modellvariablen ranzukommen. Die Werkzeuge zur automatischen Inferenz bilden eine Klasse von algorithmischen Mitteln, um konkrete Werte für unbekannte Modellparameter zu schätzen. Die Entwicklung der statistischen Inferenz gebahr die probabilistische Programmierung. Dieses relativ neue Programmierparadigma soll die Wahrscheinlichkeitstheorie, Statistik und Programmiersprachen miteinander verheiraten [Roy11]. In den probabilistischen Programmiersprachen (kurz: PPL) wird ein ML-Modell als Programmcode beschrieben, diesem Programm werden Daten zur Verfügung gestellt und die ML-Software kümmert sich automatisch um die Lösung eines modellbezogenen Rechenproblems. Im Zusammenhang mit vielen PPL wird der Abstraktivität viel Wichtigkeit beigemessen, damit die statistische Modellierung sprachlich möglichst attraktiv und leistungstechnisch (durch automatische Optimierungen) möglichst effizient gestaltet werden kann [Rei18; Bür16; PyM19c]. Die typische Herangehensweise eines Programmierers bei der Arbeit mit modernen PPL ist deklarativ: der Programmierer gibt nur an, *was* ausgerechnet werden soll, und nicht *wie* das passieren soll. Der Nutzer einer PPL implementiert keinen (Inferenz-)Algorithmus; das macht die ML-Software automatisch. Vielmehr gibt der Nutzer nur ein ML-Modell sowie dazugehörige Modelldaten an und inferiert daraus das Unbekannte. Die Deklarativität in der probabilistischen Programmierung erweist sich als vorteilhaft. Erstens entfällt dadurch die Notwendigkeit zur manuellen Implementierung und Bereitstellung von konkreten Inferenzalgorithmen durch den Nutzer, zweitens erlaubt sie schnelles Prototyping, und drittens – ein ML-Modell wird von den Inferenz-Algorithmen klar getrennt [Kat18], viertens – Skalierbarkeit und Parallelisierbarkeit können dadurch verbessert werden [Jan+19]. Diese Vorteile einer high-level-PPL treffen insbesondere beim

Spezifizieren statistischer Modelle in SQL zu.

Probabilistische Programmierung in SQL DMS-Benutzerinterfaces bieten high-level-Sprachen an, die speziell dazu entworfen wurden, komplexe technische Details hinter der Kulisse der Abstraktion zu verstecken. Sie zeichnen sich durch Deklarativität aus: der Nutzer gibt nur ein Rechenproblem an. Die Implementierung der Rechenproblemlösung wird ausgeblendet, ebenso wie die automatische Optimierung dieser Implementierung. Deklarative Sprachen zeichnen sich durch Einfachheit der Bedienung und Leichtigkeit des Erlernens aus, und genießen daher eine weite Verbreitung. Vermutlich jeder Informatiker und viele Spezialisten aus anderen Domänen kennen die deklarative Datenbanksprache SQL. Die Überträgung probabilistischer Programmierung auf SQL-Konzepte soll viele mit SQL vertraute Spezialisten dazu ermutigen, einen frischen Blick auf Bayes'sche Inferenz zu werfen. Es sei angemerkt, dass Bayes'sche Modelle in Domänen wie Umweltforschung [Cla05], Astrophysik [HSI17] [Kag12], Medizin und Bioinformatik [HDR10][Bro+03], Investment und Banken [SS01] [SS00] und sogar „Hacking" [Dav18] Einzug fanden. Vereinfachte probabilistische Programmierung in SQL macht die Vorteile von PPL für viele Domänenexperten abseits der Statistik und ML zugänglich, besonders in Fächern, die wenig mit probabilistischer Programmierung zu tun haben. SQL ist also eine legitime Wahl der Sprache zur Bayes'schen Modellierung, u.a. auch aufgrund der gut erforschten semantischen Verbindung zwischen Bayes'schen Modellen (grafisch repräsentierbar) und Entity-Relationship-Datenmanagementabstraktionen (übersetzbar in relationale Schemata, ausdrückbar in SQL) [RH16; SG13]. Mit SQL als Sprache zur high-level-Spezifikation von Inferenzaufgaben können die einzelnen Komponenten und TEchnologien – welches konkrete DMS eingesetzt wird, welches ML-Backend, welche ML-Algorithmen, welche automatischen Optimierungsmethoden usw. – stets „unter der Haube" bleiben, verborgen vor dem Benutzerinterface. Die beteiligten Softwarekomponenten sind somit leichter austauschbar gegen modernere, bessere, leistungsfähigere Komponenten, ohne dass eine Änderung vom SQL-Nutzerinterface benötigt wird.

Forschungslücken für Inferenzsysteme mit SQL-Interface Mehrere Projekte versuchen, mit herkömmlichen Mitteln der Datenmodellierung (relationale Schemata, SQL oder sogar Spreadsheets) ein deklaratives Nutzerinterface für Bayes'sche Inferenz bereitzustellen. In all diesen Projekten arbeiten die ML- und DMS-Komponenten verknüpft im Hintergrund. Die Lücke in der Forschung besteht darin, dass alle diese Projekte unterschiedliche Schwachstellen besitzen. Manche Systeme schränken ein (z.B. BayesDB [Man+15], Tabular [Gor+14], MCDB [Jam+08]) oder verbieten gar das explizite Bauen eigener Bayes'scher Modelle (MADlib [Hel+12]). Zu einer wesentlichen Einschränkung gehört beispielsweise die fehlende Möglichkeit, hierarchische Bayes'sche Modelle wie LDA [BNJ03] zu beschreiben. Inferenzsysteme mit derartigen verwehren den unkomplizierten Nachbau von vielfältigen bisher veröffentlichten Bayes'schen Modellen.

Der Mangel an Systemen mit Inferenzfunktionalität und SQL-Interface, die ihre Nutzer im Spezifizieren eigener Bayes'schen Modelle nicht einschränken, soltle vom Projekt SimSQL [Cai+13] beseitigt werden. In diesem System wird bei der DMS-Abstraktion auf klassische Datenbankmittel zurückgegriffen: für Modelldaten wird mit relationalen Schemata eine Struktur vorgegeben, und zur Spezifikation beliebiger Bayes'scher Modelle wird abgewandeltes SQL benutzt. Dies hat den Vorteil, dass SimSQL viele potenzielle Nutzer anspricht, die mit herkömmlichen Datenbanken und SQL vertraut sind. Darüber hinaus bedient sich SimSQL der deklarativen Mittel von typischen PPL, z.B. die Nutzung von Wahrscheinlichkeitsverteilungen über vordefinierte SQL-Funktionen. Das Potenzial der durch SQL gegebenen Deklarativität wird im Falle von SimSQL dennoch nicht vollends ausgenutzt. Die Austauschbarkeit der Systemkomponenten (DMS- bzw. ML-seitig) wird nicht diskutiert und ist nicht vorgesehen, obwohl das einer der Vorteile des deklarativen SQL-Interfaces ist. Ganz im Gegenteil: SimSQL setzt ausschließlich auf Markov-Chain-Monte-Carlo-Verfahren (MCMC) zur automatischen Inferenz, und ignoriert dabei – als eine Big-Data-Inferenzplattform sogar zu Unrecht – die deutlich schnelleren, aber etwas weniger genauen [SKW15] Variational-Inference-Algorithmen (VI). Das deklarative SQL-Interface von SimSQL ist leider zugeschnitten auf die Details des MCMC-gestützten Inferenzalgorithmus. Ohne diese MCMC-spezifischen Details wäre das SQL-Interface einfacher zu bedienen und offener für andere Inferenzalgorithmen. Letzter Kritikpunkt: das SimSQL-Benutzerinterface berücksichtigt nicht die in diesem Zusammenhang vorteilhafte Übersetzung Bayes'scher Modelle in relationale Schemata [RH16]. Diese Forschungsergebnisse bilden aber die Grundlage dazu, den Programmierer im gesamten Inferenz-Workflow zu begleiten, angefangen beim statistischen Modellentwurf, über DMS-seitige Modellierung hinweg bis hin zum Einbinden der Inferenzergebnisse in weitere Anwendungen. Diese Paper konzentriert sich auf ein SQL-Interface zur Abfrage der Modelldaten, entkoppelt vom Verständnis des statistischen Modells und des Inferenzalgorithmus. Diese Entkopplung der Inferenz-Ein- und Ausgabe von der Modellspezifikation ist bei SimSQL nicht gegeben. Die probabilistische Programmierung ereignet sich direkt im relationalen Schema, innerhalb der Create-Table-Statements. Daraus ergibt sich eine komplexere, schwieriger durchzublickende SimSQL-Benutzerschnittstelle, die dazu tendiert, extra Joins zur Interpretation der Inferenzergebnisse zu benötigen. Fazit: aus der Sicht der Benutzerfreundlichkeit ist es besser, wenn Übersetzung Bayes'scher Modelle in den Datenbankkontext nach [RH16] berücksichtigt wird.

Um also ein besseres SQL-Interface zur Bayes'schen Inferenz anzubieten als die bisher vorgestellten Projekte, muss ein System folgende Anforderungen erfüllen:

1. Übersetzbarkeit von Bayes'schen Netzen in relationale Datenbankschemata wird explizit eingeplant, um die Benutzerschnittstelle zu vereinfachen.

2. Das Spezifizieren eigener Bayes'scher Modelle wird nicht eingeschränkt und darüber hinaus unkompliziert gestaltet.

3. Austauschbarkeit von den beteiligten Komponenten seitens ML und DMS.

Beiträge dieser Arbeit Diese Arbeit beschäftigt sich mit dem Entwurf eines solchen SQL-Interfaces, der die obigen Anforderungen erfüllt, sowie mit dessen prototypischer Umsetzung in Form einer Middleware [2] namens SQL2Stan. Zu Beiträgen dieser Arbeit gehört ein Konzept für einen SQL-Dialekt, mit dem Bayes'sche Modelle ausgedrückt werden können und der für Nichtstatistiker möglichst zugänglich erscheinen soll. Dieser SQL-Dialekt entspricht dem SQL-Sprachstandard, braucht keinen eigenen Parser (ein PostgreSQL-Parser reicht aus) und bietet eine inhärent hohe Expressivität, weil dieser Dialekt z.T. ähnliche Sprachstrukturen wie bei einer probabilistischen Programmiersprache namens Stan auf SQL abbildet. In diesem SQL-Dialekt werden technische DMS- und Inferenz-Details, darunter physische Datenverwaltung oder Implementierung der Inferenzprozesse, abstrakt gehandhabt. Der SQL2Stan-Ansatz baut darauf auf, dass man mit einem aus dem Bayes'schen Netz nach [RH16] übersetzten relationalen Schema arbeitet.

Abgesehen vom Entwurf eines SQL-Dialekts zur Bayes'schen Modellierung gehört zum Beitrag dieser Arbeit ein Konzept zur automatischen Übersetzung vom SQL-Code in eine probabilistische Programmiersprache Stan. Durch die Übersetzung von SQL nach Stan wird eine Implementierung für automatische Inferenz automatisch generiert. SQL-Code als Basis für weitere Codeübersetzung hebt die Stärke der Deklarativität hervor, darunter die technologische Flexibilität. Beim SQL2Stan-Ansatz sind in der Eigenschaft eines Datenverwaltungsbackends nicht nur relationale Datenbankmanagementsysteme wie PostgreSQL einsetzbar, sondern auch andere DMS (Array-DBMS wie SciDB oder relationale DBMS wie SimSQL). Das relationale Schema ist dabei nur eine Abstraktion: auf der Ebene, wo die Inferenz implementiert wird, werden die Tabellenspalten als Arrays nach einem speziell dazu entworfenen Prinzip gehandhabt. Die Inferenzalgorithmen sind austauschbar. Es bedarf einer Änderung eines einzigen Wortes, um den Inferenzalgorithmus von MCMC auf das schnellere VI-Verfahren umzustellen; dabei bleibt die in SQL verfasste Inferenzaufgaben-Spezifikation gleich. Es steht außerdem nichts im Wege, im Inferenz-Backend eine Kombination beider Verfahren [SKW15] zu nutzen. Die Tatsache, dass der SQL-Ansatz alle nötigen Informationen zur Generierung einer Inferenzimplementierung liefern kann, lässt begründet hoffen, dass man SQL nicht nur nach Stan übersetzen könnte, sondern auch in andere PPL. Somit sollte auch das Inferenz-Backend austauschbar sein.

Der letzte Beitrag dieser Arbeit ist der Related-Work-Teil (Kurzfassung im Kap. 2, ausführlich im Anhangskapitel E) mit einer Zusammenfassung über Veröffentlichungen zum Gegenstand der Zusammenführung vom maschinellen Lernen und Datenmanagement. Diese Übersicht thematisiert dabei insbesondere die Abstraktion als eine erkennbare Tendenz in den Programmiersprachen relevanter Projekte. Ein vergleichbares Résumé war bisher in keiner der wissenschaftlichen Veröffentlichungen zu finden.

[2] Der Begriff „Middleware“ steht hier für eine über SQL bedienbare Softwareschicht zwischen einem PostgreSQL-Ddatenbankmanagementsystem und der Stan-Inferenzsoftware.

2. Verwandte Arbeiten (Related Work)

Die Einordnung dieser Arbeit in das wissenschaftliche Gesamtbild ist mit dem Vergleich von SQL2Stan mit anderen ML-Systemen hinsichtlich zweierlei Aspekte verbunden. Zum Einen sind nur diejenigen Projekte vergleichbar, die über eine Anbindung eines Datenmanagementsystems verfügen. Zum Anderen sollen die vergleichbaren Projekte dem Nutzer erlauben, eigene ML-Modelle zu definieren. Die Tabelle E.1) bietet eine Übersicht zu den wissenschaftlich verwandten Projekten.

Dieses Kapitel stellt die kurze Version der Related-Work-Zusammenfassung dar. Das Kapitel E im Anhang beinhaltet eine umfänglichere Version vom Related Work – mit mehr Systemen, Konzepten sowie Büchern und Veröffentlichungen, deren Zusammenhang mit dem SQL2Stan-Ansatz etwas detaillierter geschildert wird.

Relevanz von integrierten Datenmanagement-Lösungen ML-Systeme ohne angebundene DMS-Funktionalität büßen Potenzial ein: verschiedene Vorteile von DMS – vor allem Skalierbarkeit – kommen dadurch nicht zum Tragen. Obwohl solche Systeme, die sich rein mit den ML-Algorithmen beschäftigen, interessante Einsichten in die ML-Programmiersprachen bieten, stellen sie nicht auf der gleichen Argumentationslinie mit dem SQL2Stan-Konzept, wo ein DMS als angebunden betrachtet wird. Relevant ist demzufolge die Zeile „Datenmanagement-Funktionalität integriert“ der Tabelle E.1).

Modellspezifikation anstatt Blackbox-ML Der zentrale Punkt von SQL2Stan ist, dass der Programmierer eigenhändig und unaufwändig eigene Modelle spezifizieren kann. Dadurch erhält man den Vorteil, dass eigene Modellentwürfe beziehungsweise bereits veröffentlichte Bayes'sche Modelle genutzt werden können. Demnach sollte der SQL2Stan-Ansatz nur mit anderen Konzepten verglichen werden, die ihre Modelle explizit nicht als Blackboxes handhaben bzw. die ML-Funktionalität nicht auf Bibliotheksfunktionen beschränken. Die für Vergleiche besonders relevante Spalte der Tabelle E.1 nennt sich daher „Nutzerschnittstelle für ML-Modelle: Modellspezifikation“.

Konkurrierende Ansätze Das Tabellenfeld, das sich durch die Kreuzung der Zeile „Datenmanagement-Funktionalität integriert“ und Spalte „Nutzerschnittstelle für ML-Modelle: Modellspezifikation“ in der Tabelle E.1 ergibt, enthält Projekte mit besonderer Relevanz für diese Arbeit, darunter SimSQL, Tabular oder PyMC4. Aus diesem Tabellenfeld werden nur Projekte, die die statistische Inferenz mit nutzerdefinierten Modellen (d.h. kein Deep Learning o.ä. und kein Blackbox-ML) mit dem Datenmanagement verknüpfen, zum Vergleich mit SQL2Stan herangezogen.

Tabelle 2.1.: Übersicht von Machine-Learning-Systemen, geteilt nach Anbindung der Datenmanagementfunktionalität sowie nach Möglichkeit des eigenständigen Modellebauens durch den Programmierer. Aufgeführte Projekte mit einem Stern-Präfix (*) vor ihrem Namen legen viel Wert auf Deklarativität.

Datenmanagement-funktionalität \ Schnittstelle für ML-Modelle:	Bibliothek/ Blackbox	Modellspezifikation
DM-Integration nicht vorgesehen	R (MASS [Rip+19]) Python (Scikit-learn [Bui+13]) Stata [Sta19e; Ham12] Keras [Cho+15]	R (brms [Bür19; Bür16]) Python (PyMC3 [PyM19a], pyro [Ube19], Bambi [YW16]) Stata [Sta19e; Ham12] Keras [Cho+15] Infer.net [NET19a; WW11] WebPPL [Com19; Möb18b] Stan [Sta19b; Car+17] Church [Goo+12] IBAL [Pfe07] Figaro [Pfe09] * Tabular [Gor+14]
DM integriert / integrierbar	* MADlib [MAD19] * BayesDB [MIT19] * ease.ml [DS319; Li+17a] Rafiki [Chu19; Wan+18] Spark MLlib [The19g; Men+16]	* SimSQL [Cai+19; Cai+13] * **SQL2Stan** (diese Arbeit) * MLog [DS317; Li+17b] TensorFlow Probability [Ten19a; Ten19b] Python (PyMC4 [PyM19b; PyM18], to be released) Agrios [LM12] SystemML [The19a; Boe+16]

Kontextpositionierung von SQL2Stan SQL2Stan ist ein prototypisch umgesetztes System für automatisierte statistische Inferenz, welches auf eine deklarative Weise das Datenmanagement mit der Spezifikation Bayes'scher Modelle verbindet. Die wichtigsten Aspekte von SQL2Stan sind: 1) ein Nutzerinterface in einer deklarativen Datenbanksprache SQL, zugänglich auch für weniger programmierfachkundige Domänenspezialisten, 2) Abstraktion der Details sowohl vom Datenmanagement als auch von den Algorithmen zur Bayes'schen Inferenz, 3) das Spezifizieren von eigenen (auch hierarchischen) generativen Bayes'schen Modellen, 4) ein einfaches relationales Datenbankschema als Schnittstelle für Inferenz-Ein- und Ausgabe.

SimSQL Am nähesten zu SQL2Stan steht SimSQL [Cai+13] (ausführlich im Kap. E.2.16) – ein Prototyp eines relationalen Big-Data-Datenmanagementsystems zur statistischen Datenanalyse. Sowohl SimSQL als auch SQL2Stan nutzen relationale Datenbankschemata und bieten ein deklaratives und flexibles SQL-Interface zur statistischen Inferenz an. Die Deklarativität von SQL verhilft SimSQL zu einer guten Skalierbarkeit und Parallelisierbarkeit von ML-Datenanalyseprozessen [Jan+19]. SimSQL wird über einen speziellen SQL-Dialekt bedient, der erlernt werden muss und mit herkömmlichen SQL-Parsern inkompatibel ist. Im Gegensatz dazu bedient sich SQL2Stan des nicht-erweiterten SQL-Sprachstandards, ist somit zugänglicher für SQL-Programmierer und benötigt keinen extra SQL-Parser. Ähnlich wie in SQL2Stan formuliert der Programmierer einen modellspezifischen generativen Algorithmus für seine Daten; das (Nach-)Bauen von beliebigen Bayes'schen Modellen ist somit problemlos möglich.

In SimSQL beinhaltet das relationale Datenbankschema nicht nur die Schnittstelle für die Ein- und Ausgabe der Inferenz (wie bei SQL2Stan), sondern auch die probabilistischen Details der Modellspezifikation (SQL2Stan: relationales Schema und Modellspezifikation textuell getrennt). Als SimSQL-Nutzer muss man zum Abrufen der Inferenzausgabe SimSQL-spezifische Compute-Queries nutzen und dabei die Anzahl der Monte-Carlo-Iterationen sowie die Anzahl von Sampling-Pfaden angeben, über die gemittelt wird [Cai+13, S. 641]. Solche komplexen Details in den SQL-Abfragen sind einem SQL-Programmierer nicht immer zuzumuten; daher erfolgt das Abrufen der Inferenzausgabe in SQL2Stan über ganz normale Standard-SQL-Queries. Durch algorithmusspezifische Details beschränkt sich SimSQL auf einen MCMC-Inferenzalgorithmus. SQL2Stan unterstützt hingegen sowohl MCMC als auch VI; auch eine Kombination dieser beiden Techniken [SKW15] wäre bei SQL2Stan denkbar.

Die Übersetzung Bayes'scher Modelle in den Datenbankkontext nach [RH16] kann mit SimSQL kaum in Verbindung gebracht werden, was bei SQL2Stan sogar erwünscht ist. Dadurch können in SQL2Stan die SimSQL-typischen Komplikationen (von der Modellspezifikation untrennbare versionierte Relationsdefinitionen, durch zusätzliche Joins potenziell umständlichere Interpretation der Inferenzausgabe) vermieden werden. Beim SQL2Stan-Ansatz wird der SQL-Programmierer auf dem Weg vom statistischen Modellentwurf bis zur Konzipierung der Datenmanagement-

Abstraktion begleitet (siehe [RH16]) und von dort aus bei der probabilistischen Programmierung in SQL durch Inhalte dieser Masterarbeit unterstützt. Bei SimSQL liegt es hingegen am Programmierer selbst, wie er die Entitäten eines Bayes'schen Modells in sein relationales Schema hineinbringt – der statistische Modellentwurf und die damit verbundene Konzipierung der Datenmanagement-Abstraktion werden im SimSQL-Konzept nicht angesprochen. Ein probabilistische Programm in SQL2Stan besteht aus einer einzigen großen SQL-Summenabfrage. Ein probabilistisches Programm in SimSQL ist viel komplizierter – mehrere Evolutionsschleifen für versionierte stochastische Tabellen und dazu eine Compute-Abfrage mit speziellen Inferenzdetails (Anzahl der Monte-Carlo-Iterationen, Anzahl von den zu zu mittelnden Sampling-Pfaden).

SimSQL nutzt einen Hadoop-Cluster als Dateninfrastruktur und eignet sich im aktuellen Zustand für die Analyse großer Datenmengen. Der SQL2Stan-Prototyp ist keine Big-Data-Plattform, weil die Daten zwischen den Softwarekomponenten (PostgreSQL und Stan) bewegt werden müssen. Das SQL2Stan-Konzept lässt allerdings viel Spielraum für Softwarekomponenten offen (z.B. ein Array-DBMS wie SciDB, Apache Spark oder SimSQL selbst als DMS-Backend).

Tabular Das Projekt Tabular [Gor+14] (ausführlich im Kap. E.2.15) visiert eine ähnliche Zielgruppe wie SimSQL und SQL2Stan an – nämlich Nutzer, die nicht zwangsläufig Mathematiker oder Programmierer sind, und mit Hilfe statistischer Modellen Fragen zu ihren Daten beantworten wollen. Dieses Projekt verfolgt auch ein ähnliches Interface-Konzept: das sogenannte „schema-driven probabilistic programming", wo der Programmierer mit einem relationalen Schema anfängt und es um inferenzspezifische Angaben erweitert. Tabular macht die statistische Inferenz über ein Spreadsheet-Interface (sichtbar ähnlich zu Microsoft Excel) zugänglich, und blendet die Implementierungsdetails zur Bayes'schen Inferenz dabei vollständig aus. Die Tabular-Sprache bietet allerdings eine im Vergleich zu SimSQL und SQL2Stan eingeschränkte Expressivität: hierarchische Modelle wie beispielsweise LDA [BNJ03] können in Tabular nicht genutzt werden. Über DMS scheint Tabular zwar nicht zu verfügen, doch dessen Spreadsheet-Interface würde eine DMS-Anbindung zulassen: Spreadsheet-Datenanalysesysteme lassen sich beispielsweise mit Cloud-Technologien skalierbar machen (siehe `keikai.io`).

Agrios Agrios [LM12] (ausführlich im Kap. E.2.8) ist ein Projekt, in dem eine imperative, datenanalyse-freundliche Programmiersprache R mit einem skalierbaren Datenmanagement auf der sprachlichen Ebene verbunden wird. Als DBMS wurde SciDB [Sto+13] gewählt: ein Array-DBMS für wissenschaftliche Big-Data-Use-Cases, wo klassische relationale DBMS ineffizient arbeiten. Die SciDB-Anfragesprache für Arrays ist AQL (Array Query Language), die viele Ähnlichkeiten zu SQL aufweist. SciDB verfügt außerdem noch über eine weitere, funktionale Sprache namens AFL, welche einige Ähnlichkeiten mit relationaler Algebra aufweist [LM12, S. 7]. Das Agrios-Projekt stellt Ideen vor, wie die Sprache R in die SciDB-Sprache AFL

übersetzt werden kann, um die Datenanalyse-Lösungen in R für Big-Data-Use-Cases verfügbar zu machen.

MLog Das Projekt MLog [Li+17b] (ausführlich im Kap. E.2.10) ist ein Ansatz zur Integration vom deklarativen Machine Learning (ähnlich zu Keras) ins ebenfalls relationale deklarative Datenmanagement (SciDB und PostgreSQL) mit Hilfe von einer high-level-Sprache. Im Groben geht es dabei darum, die relationale Algebra und SQL um lineare Algebra und Tensoren zu erweitern. ML soll trotz Deklarativität explizit nicht als Blackbox gehandhabt werden und ausgerechnet die Klassen von ML-Modellen unterstützen, die von Datenanalysesystemen wie MADlib [Hel+12], MLlib [Men+16] und dem Array-DBMS SciDB [Sto+13] nicht unterstützt werden, was über die Funktionalität zur Bayes'sche Inferenz deutlich hinausgeht. Der nutzergeschriebene MLog-Code ähnelt Datalog – einer der bekanntesten deklarativen Datenbanksprachen nebst SQL – und wird zwecks automatischer Codeoptimierung nach Datalog übersetzt.

SystemML SystemML [Boe+16] (ausführlich im Kap. E.2.18) beschäftigt sich mit dem deklarativen Machine Learning auf verteilten Systemen (Apache Spark). Der Schwerpunkt liegt hierbei nicht bei der Bayes'schen Inferenz, sondern vielmehr auf Deep Learning und neuronalen Netzen. SystemML orientiert sich an Data Scientists (Zielgruppe von SQL2Stan – mit SQL vertraute Datenenthusiasten) und benutzt daher bewusst andere Sprachkonzepte als SQL2Stan. Dennoch legt SystemML ebenso wie SQL2Stan viel Wert auf Verknüpfung mit dem Datenmanagement, auf Skalierbarkeit und Deklarativität sowie auf Möglichkeiten zur automatischen Optimierung. Die genutzte Programmiersprache nennt sich DML (Akronym für Declarative ML) mit einer R- bzw. Python-ähnliche Syntax. Interessanterweise lassen sich DML-Skripte mit Spark-SQL-Abfragen kombinieren.

PyMC4 PyMC (aktueller Release: PyMC3 [PyM19a], ausführlich im Kap. E.2.3) ist eine Bibliothek für Python zur probabilistischen Programmierung. Dieses Framework erlaubt den Nutzern, mit Hilfe von unterschiedlichen numerischen und statistischen Methoden automatische Modellanpassung (bzw. automatische Inferenz) für Bayes'sche Modelle durchzuführen. Als eine imperativ aufgebaute PPL ist sie ungeeignet für SQL-Benutzer und ist eher mit Stan zu vergleichen – einer PPL, in die SQL nach dem SQL2Stan-Ansatz übersetzt werden kann. Stan ist für Nutzer gedacht, die aus dem statistiklastigen Umfeld kommen, PyMC orientiert sich an Python-Nutzer [Abe17]. PyMC4 [PyM19b; PyM18] ist noch in einem frühen Entwicklungsstadium, doch dieser Release wird eine grundlegende Neuerung mit sich bringen. Anstatt des im PyMC3 genutzten und mittlerweile eingestellten ML-Frameworks Theano [The16] wird in PyMC4 TensorFlow (kurz: TF) als low-level-ML-Backend genutzt. Bei Theano fehlt die deklarative Datenmanagement-Schicht, bei TF hingegen kann sie über Apache Spark (siehe TensorFlowOnSpark [Yah19]) oder Hadoop (siehe TonY [Lin19]) angebaut werden. Konkret wird PyMC4 auf

TensorFlow Probability [Ten19a; Ten19b] zurückgreifen, was PyMC4-Projekte skalierbar machen und das Bauen von generativen Deep-Learning-Modellen erleichtern wird. TF Probability ist eine Python-Bibliothek für TF mit dem Schwerpunkt auf Deep Learning und mit eigener PPL Edward2.

Methoden relevanter Systeme Viele Datenverarbeitungssysteme führen deklarative Aspekte in ihre high-level-Sprachen und -API ein (oft SQL-basiert), nutzen automatische Abfrageoptimierer [Kle17] und genießen Vorteile von deklarativen Abfragesprachen (u.a. bessere Skalierbarkeit und Parallelisierbarkeit [Jan+19]). Für manche Bereiche der Datenanalyse (z.B. Bayes'sche Inferenz, vertreten durch SQL-Systeme wie SimSQL, SQL2Stan und MADlib) reichen deklarative Abfragen aus. Manche ML-Workflows (z.B. für Empfehlungsdiente) kann man allerdings nicht ausschließlich über deklarative SQL-Abfragen umsetzen; in solchen Fällen bedarf es imperativen Codes. Daher bieten einige Systeme (z.B. Apache Hive [The19d]) neben dem deklarativen Verarbeitungsmodell (SQL) zusätzlich flexiblere Programmierschnittstellen (MapReduce o.a.) zum Ausführen vom imperativen Code an.

SQL ist also oft, aber nicht immer eine high-level-Sprache der Wahl bei Datenanalysesystemen. Die meisten Projekte orientieren sich nach einer bestimmten Zielgruppe, welcher eine Klasse von Aufgaben einfacher gemacht werden soll. Die Analyseaufgaben, die durch high-level-Sprachen formulierbar sind, lassen sich oft dem maschinellen Lernen zuordnen und reichen über statistische Inferenz bis zu neuronalen Netzen und Deep Learning. Manche Projekte (SimSQL, SQL2Stan, MADlib, MLog, Rafiki) sprechen Datenbankprogrammierer an, die mit relationalen Datenbanken und SQL arbeiten. Andere Projekte wenden sich an R-Programmierer (Agrios; brms, übersetzt R nach Stan), Python-Nutzer (scikit-learn, PyMC) und JavaScript-Anwender (WebPPL). Sogar Spreadsheets-Anwendern wird Datenanalyse zugänglicher gemacht (Tabular). Doch auch Data-Scientists, die Datenanalyse beruflich betreiben und dazu mehr Fertigkeiten besitzen, werden mit immer stärker deklarativ geprägten Programmiersprachen versorgt (u.a. Edward2 in TF Probability, ease.ml, SystemML, Keras). Mit Hilfe von Ende-zu-Ende-Softwarestacks wird versucht, die Indienststellung von Machine-Learning-Verfahren zu demokratisieren und möglichst zugänglich zu machen (TensorFlow Extended [Bay+17], DAWN [Bai+17]).

Wichtig für die Indienststellung von ML erscheint das Zusammenstellen und Betreiben von Datenanalyse-Workflows, welche regelmäßig genutzt werden und flexibel anpassbar sind. Die Spezifikation einer analytischer Aufgabe ist bei solchen Workflows nur einer der Schritte. Zu den Workflows gehören meist auch Datenvorverarbeitung (u.a. Datensäuberung) und das Serving der Analyseergebnisse (die Ergebnisse des Workflows sollen zu einem bestimmten Zweck in einer Anwendung dienen). Bei diesen Schritten setzt SQL2Stan auf die Stärken von Datenmanagementsystemen. DMS eignen sich gut als Infrastrukturen für Datenvorverarbeitung [BBM18], und sind zwecks ML-Serving inhärent gut darin, gelagerte Informationen effizient und anpassbar über vielfältige Schnittstellen zur Verfügung zu stellen.

3. Das Konzept von SQL2Stan

Das SQL2Stan-Konzept ist einfach: das System soll wie eine gewöhnliche Datenbank wirken, die via SQL bedient wird und anhand benutzerdefinierter statistischer Modelle Bayes'sche Inferenz durchführen kann. Der typische Nutzer dieses Systems sei ein sogenannter Datenenthusiast („data enthusiast" nach [Han12]) – eine Person, die Daten aus seiner Domäne besitzt, mit dem Glauben, mit Hilfe statistischer Modelle und Inferenzalgorithmen einige Fragen zu den Daten beantworten zu können. Beruflich sei ein Datenenthusiast weder Statistiker noch Programmierer. Mit relationalen Datenschemata und SQL kenne sich diese Person etwas aus; statistische Verteilungen und Bayes'sche Modelle seien für sie zumindest ein Begriff. Die meisten Inferenz-Tools sind für ihn also nicht besonders zugänglich, und ihm soll es ermöglicht werden, eigene Aufgaben zur Bayes'schen Inferenz in SQL zu definieren. In dieser Arbeit wird ein Ansatz präsentiert, wo der Nutzer mit SQL sowohl die Datenmanagement- als auch die Inferenz-Softwareschichten deklarativ bedient.

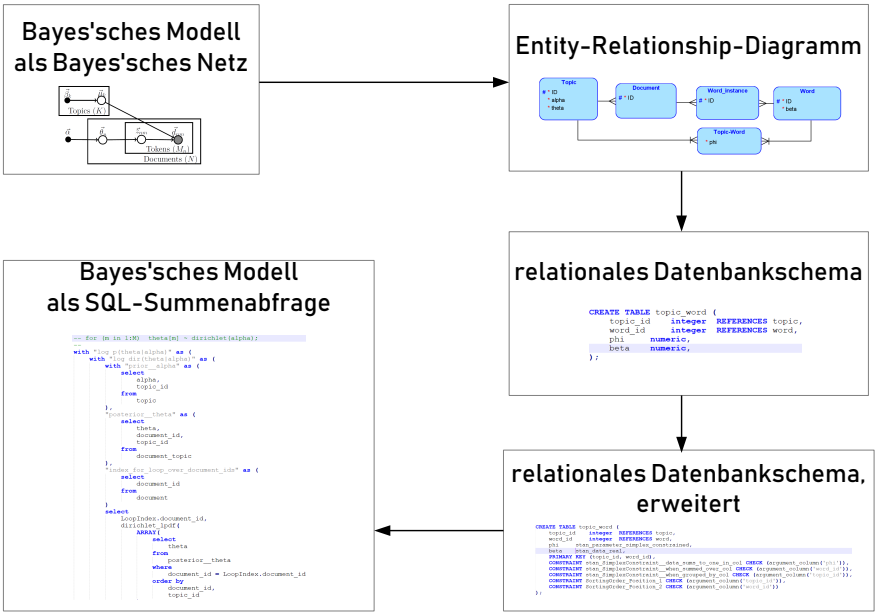


Abbildung 3.1.: Die reale Welt wird im SQL2Stan-Inferenzworkflow mit unterschiedlichen Methoden abgebildet, die sowohl Datenmanagement- als auch Bayes'sch-statistische Aspekte hervorheben. Bis auf die Ausgangssituation (Bayes'sches Netz) werden ausschließlich die gängigsten Abstraktionsmittel der Datenbanken genutzt.

Die wichtigste Voraussetzung für den SQL-Programmierer im SQL2Stan-Inferenzworkflow (Abb. 3.1)) ist das Verständnis des statistischen Bayes'schen Modells, welches zur automatischen Bayes'schen Inferenz verwendewerden soll. Das Modellverständnis wird benötigt, um die modellspezifischen Details im Workflow nach und nach explizit zu machen. Beim Explizitmachen der Modelldetails arbeitet der Programmierer mit einfachen und zugänglichen Mitteln – mit grafischen Darstellungen und mit SQL. Er beginnt mit einem Bayes'schen Modell, gegeben als eine grafische Darstellung in Form eines Bayes'schen Netzes (kurz: BN). Er übersetzt das Bayes'sche Netz manuell nach [RH16] in ein Entity-Relationship-Diagramm (ER-Diagramm). Das ER-Diagramm ist eine grafische Datenbankschema-Darstellung, die zu einem relationalen Datenbankschema führt (über automatische [Ora19] oder manuelle [KE06, S. 71–83] Übersetzung). Das relationale Schema (kurz: RS) wird in der Create-Table-Notation niedergeschrieben. Durch die Überführung des Bayes'schen Netzes in ein relationales Datenbankschema werden alle nötigen Details explizit gemacht, die zur Verwaltung aller Daten im Bayes'schen Modell integriertes (d.h. sowohl die Ein- als auch Ausgabe der Bayes'schen Inferenz) notwendig sind. Um die Bayes'sche Inferenz zu implementieren, bedarf es darüber hinaus der Explizitmachung anderer modellspezifischer Details, die mit dem Datenmanagementaspekt nichts zu tun haben. Dafür wird das relationale Datenbankschema um zusätzliche Informationen via Spaltentypen und Constraints erweitert. Dabei wird explizit angegeben, welche Schema-Entitäten als Inferenzeingabe und welche als -Ausgabe dienen, und außerdem welchen modellspezifischen Einschränkungen sie unterliegen (z.B. welche Spaltenwerte müssen sich unter welchen Bedingungen zu einer Eins aufsummieren lassen). Zuletzt braucht der Programmierer nur noch das Bayes'sche Modell beschreiben (mit Entitätennamen aus dem relationalen Datenbankschema), welches er am Anfang des Workflows als Bayes'sches Netz parat hielt. Das Beschreiben des statistischen Modells in SQL erfolgt im Stil einer probabilistischen Programmiersprache. D.h., das Modell wird in Form einer Wahrscheinlichkeitsfunktion spezifiziert. Dabei werden die restlichen impliziten Modelldetails (z.B. durch die Modellstruktur implizierten Zufallsverteilungen und ihre Abhängigkeiten) explizit gemacht. Sobald man ein um modellspezifische Details erweitertes relationales Schema durch ein SQL-Statement mit der Modellbeschreibung ergänzt hat, genügt es, auf „Start“ zu drücken, um die statistische Inferenz zu starten. Die Ergebnisse der Inferenz sind nach ihrem Abschluss über das relationale Schema mit gängigen SQL-Abfragen abrufbar.

In drei Sätzen: Anwendbarkeit vom SQL2Stan-Ansatz Die Use-Cases für SQL2Stan betreffen folgende Ausgangslage: verfügt der Nutzer über strukturierbare Daten sowie ein statistisches Modell (gegeben als ein Bayes'sches Netz), so kann er einen SQL-basierten Ansatz zur Einbindung von Bayes'schen Inferenz an das Datenmanagement nutzen. Durch die Überführung des Bayes'schen Netzes in ein relationales Datenbankschema nach [RH16] werden alle nötigen Details explizit gemacht, die zur Verwaltung aller Daten im Bayes'schen Modell integriertes (d.h. sowohl die Ein- als auch Ausgabe der Bayes'schen Inferenz) notwendig sind. Die Explizitmachung anderer wichtiger Details, die zur Implementierung der Bayes'schen Inferenz vonnöten sind, geschieht durch die Erweiterung des relationalen Schemas sowie durch die Angabe eines SQL-Statements mit der Beschreibung des Bayes'schen Modells.

3.1. Bayes'sche Modellierung

Im Mittelpunkt von Datenanalyse-Projekten stehen Daten sowie Annahmen zu den diesen Daten. Daten sind etwas, was einem bekannt ist. Meist sind Daten Beobachtungen der realen Welt. Jede Beobachtung umfasst gemessene Werte für einen bestimmten Satz an beobachteten Merkmalen. Man beobachtet z.B. ein Wort in einem Dokument und schließt auf dessen Merkmale wie 1) welches Wort im Vokabular das sein soll, 2) in welchem Dokument und 3) an welcher Stelle im Dokument es steht. Beobachtungen kann man modellieren (die beobachtete Wirklichkeit vereinfacht abbilden). Die Wahl eines Modells hängt davon ab, was man mit dem Modell bezwecken will.

Will man nur abbilden, in welcher Struktur die Daten abgespeichert werden sollen, so reicht dafür auch ein relationales Modell. Relationale Datenschemata und Entity-Relationship-Modelle (automatisch [Ora19] oder manuell [KE06, S. 71–83] ineinander überführbar) stammen aus dem Datenbankbereich, und präsentieren Strukturen und Beziehungen von Datenbanktabellen. Wenn man die Daten jedoch tiefer analysieren will, muss man annehmen, dass den Daten etwas Nichtbeobachtbares zugrunde liegt, was ihre Beschaffenheit beeinflusst. Das unbekannte, noch versteckte Wissen zu den Daten kann ebenfalls modelliert werden. Dazu werden Data-Mining-Modelle genutzt (nicht zu verwechseln mit zuvor genannten Datenbankmodellen), welche die Daten mit statistischen Methoden modellieren. Bekannte Daten können in einem statistisch geprägten Data-Mining-Modell auf beobachtbare Variablen abgebildet werden, das versteckte Wissen über die Daten wird durch nicht-beobachtbare Zufallsvariablen ausgedrückt; die Unsicherheiten werden durch Wahrscheinlichkeiten repräsentiert. Ein Data-Mining-Modell kann Annahmen dazu darstellen, wie die Daten entstanden (stochastisch gesehen: „erwürfelt“) sein könnten. Statistische Modelle, die mit derartigen Datenentstehungsannahmen arbeiten, werden aufgrund der Fragestellung „wie wurden die Daten möglicherweise generiert?“ als generativ bezeichnet.

3.1.1. „Bayes'sch" als statistisches Paradigma

Solche datenanalytischen Grundkonzepte wie die oben genannten *generativen* Modelle haben ihre Hintergründe in der Wahrscheinlichkeitstheorie. Mit der Einführung von Wahrscheinlichkeiten und Wahrscheinlichkeitsverteilungen werden die Unsicherheiten in den analytisch zu untersuchenden statistischen Effekten handhabbar. Bayes'sche Statistik ist ein Bereich der Wahrscheinlichkeitstheorie, in dem man mit den Maßen von Unsicherheiten arbeitet. Bayes'sche Statistik verwendet den Satz von Bayes, den Bayes'schen Wahrscheinlichkeitsbegriff und u.a. generative Modelle, um stochastische Fragen zu beantworten. Eine Wahrscheinlichkeit nach Bayes ist keine Häufigkeit oder Verwirklichungstendenz von bestimmten Phänomenen, sondern ein durch die bisher bekannte Information begründeter Grad an Glaubwürdigkeit, dass ein Ereignis in den aktuellen Wissenskontext reinpasst.

Der Satz von Bayes ($P(A \mid B) = \frac{P(B|A)\,P(A)}{P(B)}$ mit $P(B) \neq 0$) beschreibt die Wahrscheinlichkeit von einem Ereignis ahnand der bisherigen Erkenntnisse, die mit diesem Ereignis im Zusammenhang stehen können. Dabei sei A eine Behauptung/Hypothese (z.B. ein hergestelltes Bauteil ist mangelhaft) und B sei die Evidenz (ein Test zeigt an, dass ein Bauteil mangelhaft sein könnte). $P(A)$ ist der sogenannte Prior – der Grad der Glaubwürdigkeit der Behauptung A. Mit anderen Worten ist der Prior die Wahrscheinlichkeit für die Hypothese A vor dem Beobachten der Evidenz B (wie z.B. die Wahrscheinlichkeit, dass ein beliebig ausgewähltes Bauteil mangelhaft ist). $P(B)$ ist der Grad für Glaubwürdigkeit des Eintritts der Evidenz B (inwieweit ist es glaubwürdig, dass das Testergebnis korrekt ist?). Die Wahrscheinlichkeit der Evidenz $P(B)$ ist unabhängig von den aufgestellten Hypothesen. $P(B \mid A)$ ist die sogenannte Likelihood bzw. Plausibilität. Als Wahrscheinlichkeit für B gegeben A gibt sie an, wie kompatibel die Evidenz B zur aufgestellten Hypothese A ist (angenommen, das Bauteil ist definitiv defekt, wie hoch wäre die Glaubwürdigkeit vom Test?). $P(A \mid B)$ sei der sogenannte Posterior – der Grad der Glaubwürdigkeit für die Hypothese A gegeben Evidenz B, nachdem die Evidenz B beobachtet wurde (wenn der Test anzeigt, ein Bauteil sei mangelhaft, wie glaubwürdig ist es, dass das Bauteil tatsächlich defekt ist?). Und der Quotient $\frac{P(B|A)}{P(B)}$ drückt schließlich aus, inwiefern die Evidenz B die Hypothese A befürwortet. Der Satz von Bayes verknüpft den Grad der Glaubwürdigkeit einer Behauptung vor und nach dem Einbeziehen der Evidenz. Daher wird er in den Verfahren der Bayes'schen Statistik dazu verwendet, die Bayes'sche Wahrscheinlichkeit für eine Hypothese zu aktualisieren, sobald neue Informationen verfügbar werden. Diese durch den Eingang vom neuen Wissen bedingte Aktualisierung von Wahrscheinlichkeiten nach dem Bayes'schen Satz ist das Hauptunterscheidungsmerkmal von Bayes'scher Inferenz als einem Paradigma der statistischer Inferenz.

Das Hauptwerkzeug der Bayes'schen Statistik sind Wahrscheinlichkeitsverteilungen. Im bereits gesehenen Ausdruck $P(A \mid B)$ mit Zufallsvariablen A und B kann anstelle des Buchstaben P auch ein Name einer Wahrscheinlichkeitsverteilung stehen, z.B. *Dirichlet*. In der vor kurzem erwähnten statistischen Inferenz geht es darum, die

Eigenschaften von Wahrscheinlichkeitsverteilungen abzuleiten. Dazu benötigt man Annahmen. Eine typische Annahme hat man bereits im Zusammenhang mit den generativen Modellen gesehen: man nimmt an, dass die beobachteten Daten einer Grundgesamtheit entstammen, und dass diese Grundgesamtheit einer bestimmten Zufallsverteilung folgt. Kennt man diese Zufallsverteilung, so kann man mit ihr neue Daten generieren, welche in ihren Merkmalen den bereits bekannten Daten gleichen. Die Zufallsverteilung, die der Grundgesamtheit zugrunde liegen soll, besteht selbst aus Verteilungen und den darin involvierten Zufallsvariablen. Die Annahmen zur Verteilung der Grundgesamtheit fasst man in einem generativen statistischen Modell zusammen, das man auch als *generatives Bayes'sches Modell* bezeichnen kann. In solch einem Modell sind die Zufallsvariablen beobachtbar (die Werte liegen auf der Hand) oder latent (Werte vor der Inferenz unbekannt). Die Methoden Bayes'scher Inferenz schätzen konkrete Werte für versteckte Modellparameter. Diese Schätzung ist genau das, was ein Datenenthusiast von einem Inferenzsystem erwartet.

3.1.2. Bayes'sche Netze – eine Darstellungsform Bayes'scher Modelle

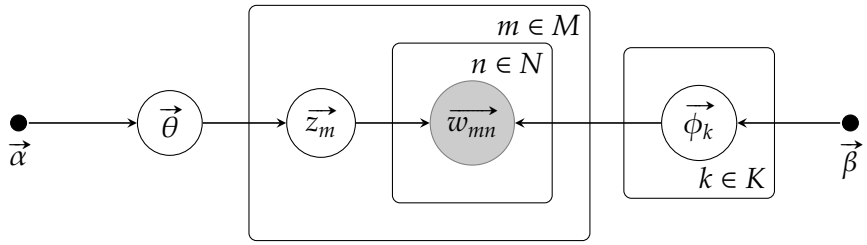


Abbildung 3.2.: Ein Bayes'sches Modell für den unsupervised-naïve-Bayes-Klassifikator, grafisch dargestellt als ein Bayes'sches Netz in der Plate-Notation.

Eine Art, ein generatives Modell zu beschreiben, ist visuell – die grafisch spezifizierten Modelle nennt man PGM (Probabilistic Graphical Model). Ein Beispiel dafür sieht man in der Abbildung 3.2 : es ist ein Bayes'sches Netz (mehr zu Bayes'schen Netzen in [Bis06, S. 360–371]) für den unsupervised-naïve-Bayes-Klassifikator.

Bayes'sche Netze sind gerichtete, azyklische Graphen – eine der grafischen Notationen für probabilistische Machine-Learning-Modelle. Sie beschreiben die Daten visuell mit Hilfe von Zufallsvariablen. Ein Knoten im Graphen ist eine Zufallsvariable. Es gibt drei Arten von Knoten, die sich visuell voneinander unterscheiden. Grau ausgefüllt sind die Knoten für beobachtete Variablen (deren Werte wir wissen), unausgefüllt/weiss sind die Knoten der latenten Variablen (deren Werte wir nicht wissen), und zuletzt die schwarzen kleinen Knoten, die für Modell-Hyperparameter stehen (eine Möglichkeit für Datenforscher, Feintuning am Modell vorzunehmen). Die in der Abbildung gewählte Notation ist die sogenannte Plate Notation, d.h. die sich wiederholenden Unterstrukturen werden auf eine Ebene (einen mit einer Kar-

dinalität beschrieben „Teller“, engl. Plate) gesetzt. Steht beispielsweise ein Knoten z auf einer abgegrenzten, mit einer Zahl M beschrifteten Ebene, dann gibt es insgesamt M viele z-Knoten, wovon jeder einen eigenen Index m besitzt. Die gerichteten Kanten beschreiben eine Abhängigkeit: wenn z.B. im Graphen eine gepfeilte Kante von *alpha* nach *theta* verläuft, so bedingt *alpha* die Zufallsvariable *theta*. Solch eine Abhängigkeit, ausgedrückt mit einer gerichteten Kante, entspricht somit einer Zufallsverteilung. Wichtig ist auch zu wissen, dass die Abwesenheit einer Kante zwischen zwei Zufallsvariablen-Knoten auf ihre Unabhängigkeit hindeutet.

Bayes'sche Netze lassen sich in Entity-Relationship-Diagramme für Datenbanken übersetzen [RH16], die sich wiederum leicht (da automatisierbar) in relationale Schemata überführen lassen. Das aus dem Bayes'schen Netz übersetzte DB-Schema ist das Interface zwischen der Machine-Learning- und DBMS-Datenschnittstellen. Dieses relationale Interface zwischen ML und DMS kann in den Einklang mit den bereits gegebenen Daten gebracht werden, indem es mit dem relationalen Schema für gegebene Daten verschmolzen wird. Dabei ist der Begriff „integriertes Datenschema“ von Belang, welcher im auf diesen Abschnitt folgenden Kapitel erklärt wird.

Zuletzt ist es dennoch interessant und erwähnenswert, dass man nicht nur aus einem Bayes'schen Netz ein ER-Diagramm für Datenbanken erzeugen kann. Auch die umgekehrte Richtung – aus einem relationalen DB-Schema zum Bayes'sches Modell – würde gelten [SG13]. Ein relationales Schema ist zwar kein Entity-Relationship-Modell, aber ER- und relationale Schemata lassen sich ineinander automatisiert (z.B. mit Oracle SQL Developer Data Modeler [Ora19]) oder manuell (siehe [KE06, S. 71–83]) übersetzen.

3.2. Integriertes relationales Schema als zentrale Schnittstelle

Anhand der Daten und eines statistischen Modells kann approximativ auf konkrete Werte für die versteckte Zufallsvariablen geschlossen werden, was mit speziellen Inferenzalgorithmen bewerkstelligt werden kann. Der Ausgangspunkt sind die in einer Datenbank gelagerten Daten. Die inferierten (d.h. die durch Inferenz abgeleiteten neuen) Werte vervollständigen das Gesamtbild in der nutzerverwalteten Datenbank – sie ergänzen die Datenbankeinträge um zusätzlich gewonnene Informationen. Das somit ergänzte bzw. trainierte Modell kann vielfältig von einer an die Datenbank gebundenen Anwendung eingesetzt werden, u.a. für Zwecke der Klassifikation oder Vorhersage.

Ein mit Daten bestücktes probabilistisches Modell kann also Einblicke in die unter den Daten verstecken Strukturen liefern. Der typische Datenenthusiast ist an diesen Einblicken natürlich interessiert. Um sie zu bekommen, würde er jedoch ungern neue Programmiersprachen erlernen bzw. in den ihm bekannten Sprachen viel programmieren wollen. Probabilistische Sprachen für Inferenzsysteme sind nicht immer einfach (was man in späteren Kapiteln an der Übersetzung von SQL nach

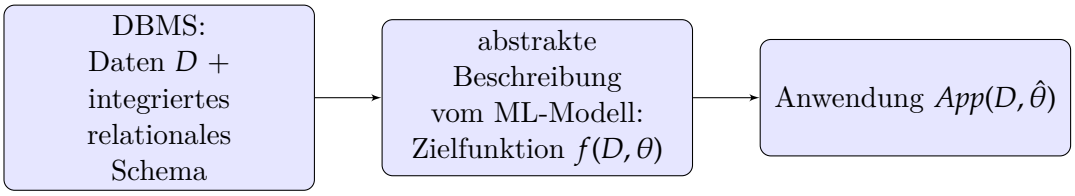


Abbildung 3.3.: Workflow zur Bayes'schen Inferenz mit SQL2Stan, komprimiert auf die wesentlichsten Nutzerinteraktionen.

Stan merken wird), und darüber hinaus kann das Implementieren und Optimieren zeitaufwändig sein. Diese Komplikationen können einen Datenenthusiasten davon abhalten, neue (Modell-)Ideen schnell zu prototypisieren. In SQL durch die Deklarativität dieser Sprache muss der Programmierer nichts selbst implementieren, demzufolge kann der Datenenthusiast zwecks Inferenz von Bayes'schen Modellen zu SQL2Stan greifen. Dabei werden folgende Schritte durchgegangen (Workflow aus der Abb. 3.3, aber detaillierter):

1. Der Datenenthusiast (DE) formuliert zu seinen Daten ein relationales DB-Schema.

2. Dann wählt DE ein generatives Data-Mining-Modell (Bayes'sches Netz) aus.

3. DE übersetzt das ausgewählte statistische Modell in ein ER-Modell (nach [RH16]).
   - Nun besitzt er zwei Datenbankschemata: zum Einen – das ursprüngliche DB-Schema für seine Daten, zum Anderen – das durch Übersetzung aus dem statistischen Modell entstandene DB-Schema für das Modell.

4. DE lässt diese zwei Schemata verschmelzen.
   - Im Grunde erweitert er dabei das gegebene DB-Schema für Domänendaten um neue, modellspezifische Spalten für versteckte Modellparameter (θ in Abb. 3.3 seien ungeschätzte latente Modellparameter, $\hat{\theta}$ seien inferierte Werte für latente Modellparameter). Das Ergebnis der Schemaerweiterung ist das **integrierte relationale Schema**.

5. DE abstrahiert die Zusammensetzung vom generativen Modell über eine Wahrscheinlichkeitsfunktion ($f(D, \theta)$ in Abb. 3.3).
   - Der Nutzer beschreibt diese Funktion deklarativ in SQL und nutzt dabei die Spaltennamen aus dem integrierten DB-Schema.

Das zentrale Element in diesem „data model driven"-Workflow [RH16, S. 13] ist das **integrierte Datenbankschema**. Integriertes Datenschema bedeutet: das DB-Schema für Daten wird mit dem DB-Schema für das statistische Modell verschmolzen. Die Entitäten im statistischen Modell werden beim Integrieren an die

Entitäten in den gegebenen Daten gematched. Das Modell- und Daten-Entitäten-Matching kann man sich ahnads eines Beispiels wie folgt vorstellen. In einem statistischen Beispielmodell zur Modellierung von der Struktur einer tokenisierten Textsammlung sei eine Index-Kardinalität N erwähnt, und damit sei die Anzahl der Wortinstanzen gemeint. N wird mit der Menge der Spaltenwerte *word_instance_id* in der DB-Tabelle *word_instance* gematcht. Dann wird eine Modellvariable w_n – ein Wort, das im Token n steht – in der Datenbank mit der Spalte *word_id* aus der Tabelle *word_instance* gematcht. Das Entitätenmatching möchte ich an der Stelle sogar nochmals verdeutlichen, anhand eines anderen Beispiels (Abb. 3.4) – man nehme anstatt einer tokenisierten Textsammlung einfach eine Onlineshop-Datenbank und ein Bayes'sches Modell dazu, gegeben als ein Bayes'sches Netz. Das Matching ihrer Entitäten kann man sich im Groben so vorstellen:

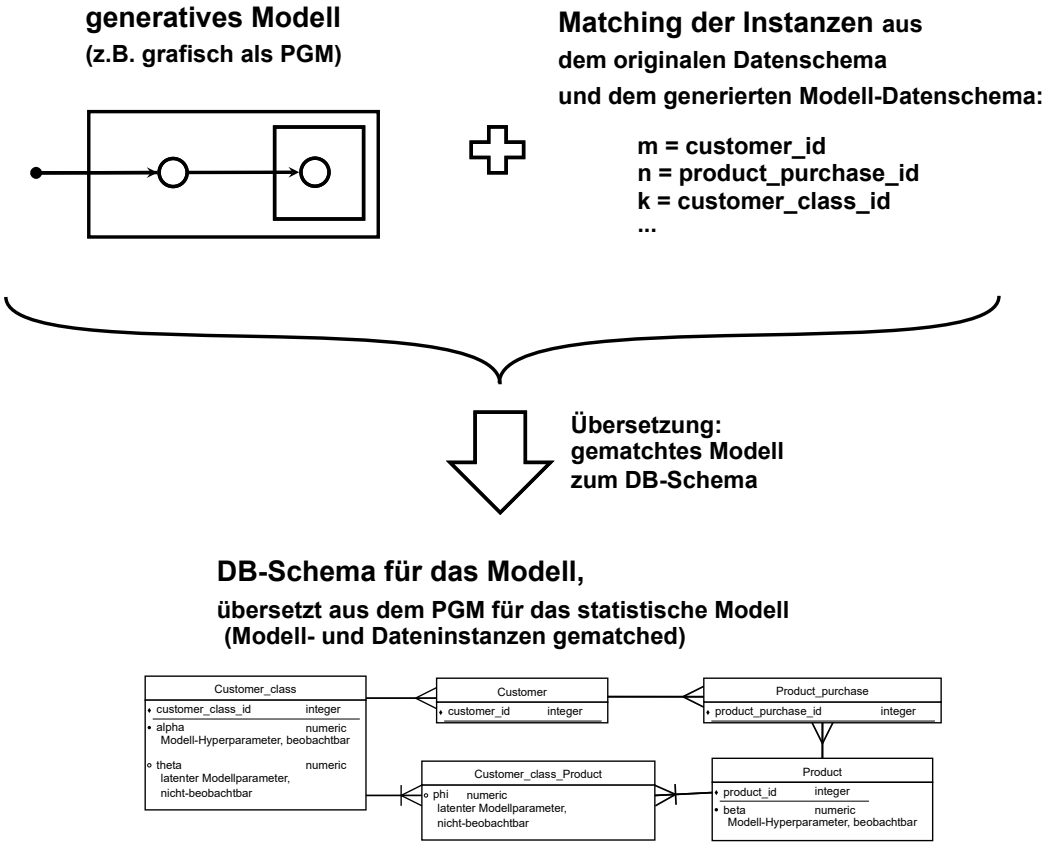


Abbildung 3.4.: Entity-Relationship-Diagramm, übersetzt aus einem Bayes'schen Netz (Entitätennamen gematched)

Das Entitäten-Matching – zu erfolgen durch den Nutzer – definiert eine Schnittstelle zwischen den Machine-Learning-Algorithmen und den konkreten Eingabedaten für diese Algorithmen. Dadurch weiss das System, mit welchen DB-Spalteneinträgen die ML-Inferenzalgorithmen versorgt werden sollen, und in welche DB-Spalten die

algorithmisch geschätzten Ergebnisse reingeschrieben werden.

Durch das Integrieren vom Daten- mit dem Modell-Schema werden die Ein- und Ausgabe der Parameterschätzung in einem gemeinsamen Datenbank-Schema aneinandergeknüpft, und dieses integrierte Schema dient als API nach Außen für datenbankgebundene Anwendungen. Alles, was zu dieser Integration im Normalfall nötig ist, ist das Hinzufügen von zusätzlichen modellspezifischen Tabellen und Spalten in das bereits existierende relationale Modell. Ein Beispiel: eine Tabelle mit Kundendaten in Abb. 3.5 erhält eine neue Spalte, weil der Datenbankbetreiber die Kunden mit Hilfe von Bayes'scher Inferenz klassifizieren möchte und die Inferenzergebnisse (u.a. Zuweisung Kunde–Kundenklasse) in der Datenbank landen sollen. Das Ergebnis der Erweiterung vom relationalen Datenschema auf der Ebene einer einzigen Tabelle ist in der Abbildung 3.6 zu sehen.

customer_id	first_name	surname	city
1	Max	Mustermann	Berlin
2	Erika	Mustermann	Hannover

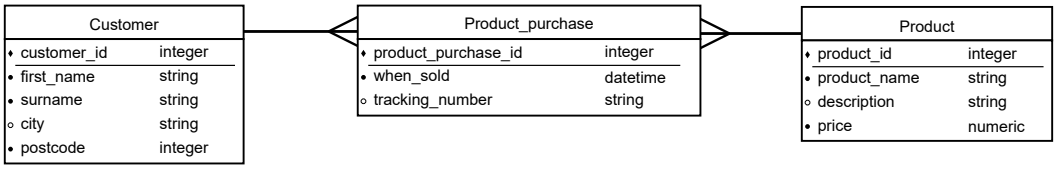
Abbildung 3.5.: Tabelle aus dem relationalen Schema einer Beispiel-Kundendatenbank.

customer_id	first_name	surname	city	customer_class
1	Max	Mustermann	Berlin	NULL
2	Erika	Mustermann	Hannover	NULL

Abbildung 3.6.: Eine Beispiel-Tabelle, erweitert um eine modellspezifische Spalte (Kundenklassen-ID). Die neu hinzugekommene Spalte ist noch leer.

Auf der Schema-Ebene ist es sehr wahrscheinlich, dass auch neue modellspezifische Tabelle hinzukommen. Die Verschmelzung von den Schemata zu einem integrierten Schema kann auf die in der Abb. 3.7 dargestellte Weise präsentiert werden.

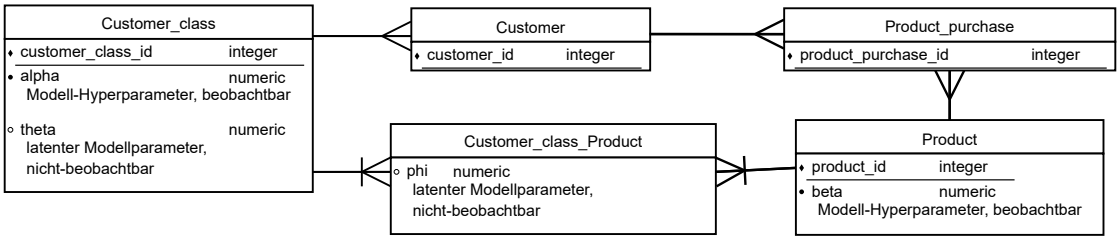
Grober Leitfaden zur aktuellen Orientierung Der SQL2Stan-Anstz verlässt sich auf das Verfahren von [RH16] bis zu dem Punkt, wo das integrierte Entity-Relationship-Diagramm aus einem Bayes'schen Modell erstellt wird. Der Leser ist gerade genau an dieser Stelle. Dieses ER-Diagramm lässt sich mit automatischen [Ora19] oder manuellen [KE06, S. 71–83] Mitteln in ein relationales Schema übersetzen. SQL2Stan geht über diesen Stand hinaus, indem es dazu eine Art probabilistische Programmierung in SQL entwirft. Das integrierte relationale Schema wird demzufolge um inferenzspezifische Einschränkungen erweitert, und das Bayes'sche Modell wird in einem zusätzlichen SQL-Statement außerhalb des relationalen Schemas beschrieben. Die entsprechende Vorgehensweise wird in folgenden Kapiteln thematisiert.



Originales DB-Schema für gegebene Daten

DB-Schema für das Modell,

übersetzt aus dem PGM für das statistische Modell (Modell- und Dateninstanzen gematched)



integriertes Schema:

beinhaltet Spalten sowohl für Eingabe- als auch für Inferenz-Ausgabe-Daten

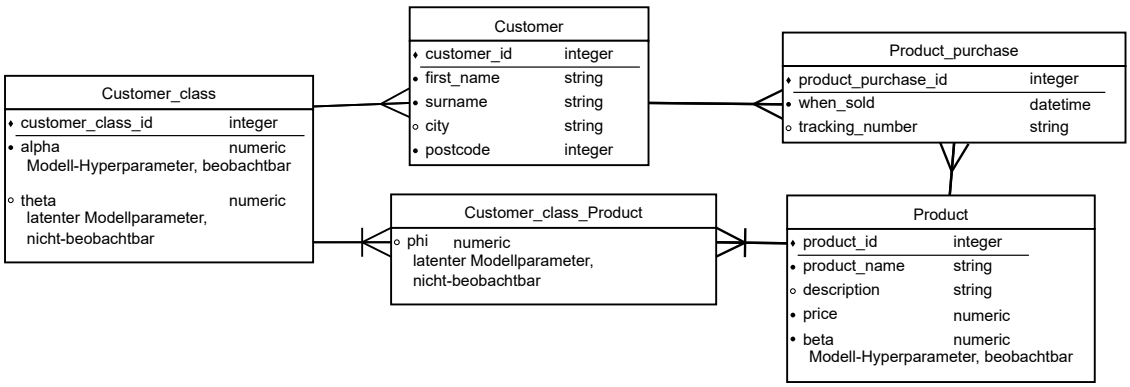


Abbildung 3.7.: Erstellung eines integrierten Schemas, welches die Entitäten des Naïve-Bayes-Modells mit den Entitäten einer Kunden-DB zusammenbringt.

Die vom Datenenthusiasten begehrten errechneten Schätzungsergebnisse ($\hat{\theta}$ in Abb. 3.3) landen aus seiner Sicht automatisch in den dafür vorgesehenen Tabellenspalten (wie z.B. die Spalte *customer_class_id* in Abb. 3.6) im integrierten relationalen Datenschema. Es ist also Aufgabe des Systems, statistische Inferenzalgorithmen automatisch zu implementieren, mit diesen Algorithmen konkrete Werte für die im statistischen Modell enthaltenen unbekannten Parameter zu schätzen und die geschätzten Werte in das Datenmanagement-System einzutragen (als Ergänzung zu den Daten, die sich dort schon zu Beginn des Workflows befanden).

Die statistischen Modelle kamen im Workflow an zwei grundlegenden Stellen vor: beim Erstellen von einem integrierten Datenschema sowie bei der Modellspezifikation. Gleich wird der Leser erfahren, wie das integrierte Schema im SQL-Dialekt von SQL2Stan beschrieben, erweitert und in eine PPL übersetzt werden kann. Um etwas vorzugreifen: die Modellspezifikation wird direkt im Anschluss eine Rolle spielen. Ein statistisches (Bayes'sches) Modell kann nämlich als eine Wahrscheinlichkeitsfunktion (Zielfunktion) beschrieben werden – sogar in SQL.

3.2.1. Inferenz-Ein- und Ausgabe im integrierten Schema

In einem Satz: integriertes Schema Integriertes Schema ist ein relationales Datenbankschema mit folgenden Merkmalen: 1) als eine datenmanagement-seitige Abstraktion bildet es in einer gemeinsamen logischen Struktur sowohl gegebene Daten als auch die Modellentitäten ab, und 2) man kann darin anhand der Spaltentypen leicht erkennen, was vor der automatischen Inferenz an Daten gegeben sein soll – und was als Inferenzausgabe gilt.

Create-Table-Anweisungen – ein sprachliches Mittel der DMS-Abstraktion Das integrierte Datenschema lässt sich in SQL mit Hilfe von Create-Table-Anweisungen ausdrücken. Die Klarheit darüber, was zu gegebenen Daten gehört und was durch das System automatisch ausgerechnet werden muss, wird via Spaltentypen vermittelt. Im Listing 3.1 sieht man ein Beispiel mit Spaltendefinitionen:

Listing 3.1: Festlegung von Spaltennamen und -Typen einer Tabelle im relationalen Schema (notiert in SQL als Create-Table-Anweisung). Constraints ausgeblendet.

```
CREATE TABLE customer_class (
        customer_class_id   integer not null,
        alpha   stan_data_real,
        theta   stan_parameter_simplex_constrained,
        ... -- Nur Spalten und deren Typen;
        -- Constraints weggelassen.
);
```

Eine Tabelle aus dem integrierten Schema wird in SQL durch eine Create-Table-Anweisung (SQL-Notation des relationalen Schemas) beschrieben, welches seinerseits aus zwei teilen besteht: Definition von Tabellenspalten sowie Spezifikation der

Constraints (Spezialbedingungen, die für die Spalten dieser Tabelle gelten). Abb. 3.8 bietet eine Übersicht zur SQL2Stan-typischen Tabellenspaltendefinition:

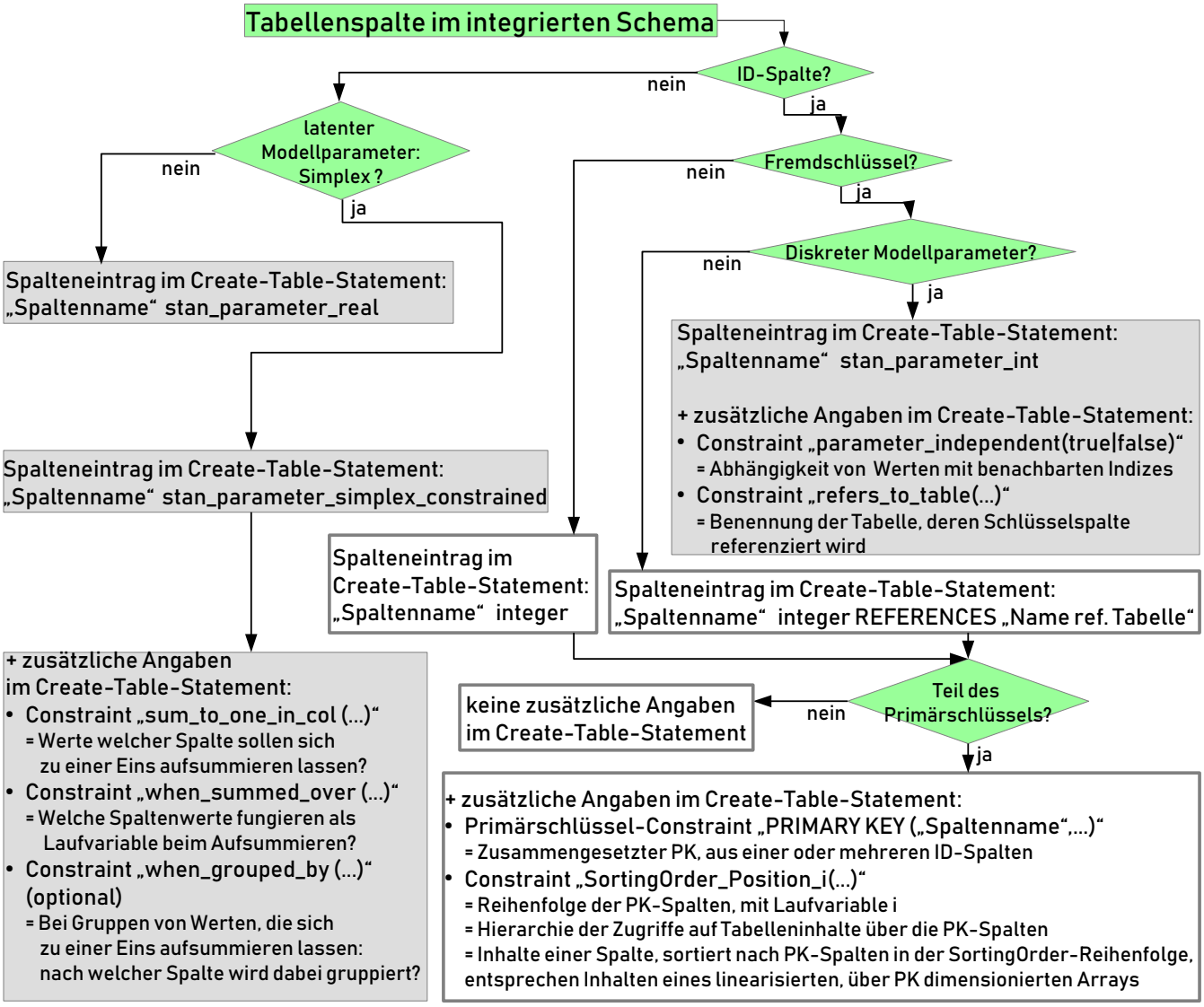


Abbildung 3.8.: Allgemeines Vorgehen beim Spezifizieren der Tabelle aus dem integrierten Schema in SQL, spaltenweise. Zur Semantik der Abbildung: grüne Rauten implizieren Ja/Nein-Entscheidungen, hellgraue Rechtecke betreffen unbeobachtete Daten (latente Modellparameter), weiß ausgefüllte Rechtecke beziehen sich auf beobachtete Daten (nicht-latente Modellvariablen).

Die kommenden Paragraphen legen die Typen von Spalten in SQL2Stan dar. Im Anschluss werden die (aktuell noch ausgeblendeten) Constraints aus dem SQL2Stan-

Dialekt erklärt, und darauf folgend – die Übersetzung des durch Create-Table-Statements beschriebenen integrierten relationalen Schemas in Vektoren und Arrays im automatisch durch SQL2Stan generierten Stan-Programm.

ID-Spalten Die Schlüsselspalten sind immer ganzzahlig (integer) und darf nicht den Wert NULL annehmen. Es kann eine oder mehrere Schlüsselspalten sein, d.h. der *primary key* kann auch zusammengesetzt sein. Solche Spalten zur eindeutigen Zeilenidentifizierung (ID) spielen bei SQL2Stan eine sehr große Rolle: anhand der Tabellenschlüssel können aus Tabellenspalten Arrays gebildet werden. Arrays sind im maschinellen Lernen durchaus gewollt: die meisten Algorithmen zur Bayes'schen Inferenz laufen auf Arrays, Vektoren und Matrizen, und nicht auf Tabellen und Spalten Außerdem wurden sogar spezielle Array-DBMS ins Leben gerufen, aus dem Grund, dass sich Arrays für wissenschaftliche Big-Data-Analysezwecke besser eignen sollen (siehe SciDB [Sto+13]). SQL2Stan schlägt eine Möglichkeit vor, die im integrierten Datenschema codierten Spaltendaten auf Arrays zu übertragen. Die allgemeine Idee dazu lautet: jede Tabellenspalte entspricht einem Array, dessen Dimensionalität vom Tabellenschlüssel diktiert wird. Für die Erklärung, wie genau die Übersetzung vom relationalen Schema in Arrays geschieht, muss der Leser sich noch ein wenig gedulden – dafür müssen noch die Constraints im integrierten Schema erklärt werden, die im obigen Beispiel nicht zu sehen sind.

Spalten mit Inferenz-Eingabe-Daten Prinzipiell gehören die integer-typisierten IDSpalten zur Inferenzeingabe – d.h. zu den bekannten Daten. Falls die bekannten Modelldaten jedoch nicht integer-getypt sind, gibt es eine andere Möglichkeit. Die Spalten mit reellen numerischen Werten für beobachtete Modellvariablen sind vom Spaltentyp „stan_data_real“ (*Stan* wie der Name der Inferenzsoftware, *data* stünde für „gegebene Daten“, *real* impliziert den numerischen Typ der reellen Zahlen). Spalten von diesem Typ sind beispielsweise Modell-Hyperparameter, die zum nutzerseitigen Modelltuning dienen können.

Spalten mit Inferenz-Ausgabe-Daten Die Spalten, in denen nach der automatischen Inferenz die geschätzten Werte für Modellparameter landen sollen, besitzen den Spaltentyp „stan_parameter_real“. Falls sie besonderen Einschränkungen unterliegen, lautet der Typname vielmehr „stan_parameter_simplex_constrained“ und impliziert somit das Vorhandensein spezieller Constraints, welche diese besonderen Einschränkungen zum Ausdruck bringen. Eine sehr typische (und aktuell die einzig ausdrückbare) Sondereinschänkung für Modellparameter ist die sogenannte Simplex-Bedingung: die Werte müssen sich auf eine bestimmte Art zu einer 1 aufsummieren lassen.

Spalten mit diskreten Modellparametern Einen Moment später wird man auch den Spaltentyp „stan_parameter_int“ kennenlernen. Er ist den Spalten mit diskreten Modellparametern vorbehalten. Statt diskret kann man einfach sagen: ganzzah-

lig, inhärent vom Typ integer. Solche ganzzahligen Modellvariablen-Werte werden nicht geschätzt, sie sind weder Ein- noch Ausgabe der Inferenz. Sie dienen vielmehr dazu, die Referenzen zwischen den Tabellen (Fremdschlüssel) als unbekannt zu markieren. Ein Beispiel: die Klassen der Onlineshop-Kunden. Wenn man den Kunden vor dem Modelltraining keine Klassen manuell zuweist, dann ist die Zuweisung Kundenklasse–Kunde (die Spalte customer_class_id in der Tabelle customer) ein diskreter Modellparameter.

3.2.2. Inferenzspezifische SQL-Constraints

Mit Sorting-Order-Constraints zur Array-Ansicht Das Datenbankschema braucht zusätzliche Information. Aus dem integrierten Schema allein lässt sich die automatische Inferenz an einem konkret gewählten Bayes'schen Modell nicht automatisch implementieren. Die Lösung dafür liegt daher darin, das integrierte Schema um inferenzspezifische Details (als SQL-Constraints in den Create-Table-Statements) zu erweitern, und später die Modellstruktur durch eine zusätzlich SQL-Abfrage zu beschreiben.

Modellspezifische SQL-Constraints im integrierten Schema

In einem Satz: modellspezifische Constraints im integrierten Schema Alle Angaben zum generativen Modell (ob explizit oder implizit), die aus dem Bayes'schen Netz herauszulesen sind, müssen explizit gemacht und als Constraints in das integrierte Schema eingetragen werden.

Einschänkungen im SQL2Stan-Prototyp und ihre Behebung Der SQL2Stan-Prototyp besitzt aktuell eine Einschränkung: jede Art von inferenzspezifischen Constraints darf pro Tabelle nur einmal definiert werden. Z.B. zwei Simplex-Bedingungen in einer Tabelle dürfte man (noch) nicht definieren. Sprachlich gesehen wäre das aber Problem, die Art der Formulierung von Constraints ein wenig zu ändern, dass man pro Tabelle mehrere inferenzspezifische Constraints gleicher Art nutzen kann. Es würde dann einfach darauf hinauslaufen, dass der Name oder der Rumpf des Constraints zusätzlich einen Bezeichner erhält, der die Eindeutigkeit beim Interpretieren mehrerer gleichartiger Constraints gewährleistet. An relevanten Stellen werden dafür entsprechende Verbesserungs- und Weiterentwicklungsvorschläge beschrieben.

Simplex-Constraints Eine Art solcher Zusatzangaben sind die Simplex-Bedingungen, d.h. bestimmte Werte müssen zu einer 1 aufsummiert werden können. Die Gründe, warum man sowas braucht, ruhen aus der Statistik: die Wahrscheinlichkeiten für alle möglichen Ausgänge eines Zufallsexperimentes summieren sich zu einer 1. Diese Information über die Simplex-Bedingungen im statistischen Modell werden durch folgende drei SQL-Constraints ausgedrückt:

1. stan_SimplexConstraint___**data_sums_to_one_in_col**
   - CHECK-Argument: argument_column('Spalte aus dem Tabellenschlüssel')
2. stan_SimplexConstraint___**when_summed_over_col**
   - CHECK-Argument: argument_column('Spalte aus dem Tabellenschlüssel')
3. stan_SimplexConstraint___**when_grouped_by_col** (optional)
   - CHECK-Argument: argument_column('Spalte aus dem Tabellenschlüssel')

Im einfachsten Falle braucht man nur die ersten zwei Constraints. Z.B. Werte von *theta* aus der Tabelle Customer, aufsummiert über alle Kundenklassen $k \in K$, ergeben eine Eins: $\theta \in (0,1)^K,\ \sum_K \theta_k = 1$. Für SQL-Notation siehe Listing 3.2:

Listing 3.2: Beispiel-Anwendung von Simplex-Constraints auf Spalte theta

```
theta stan_parameter_simplex_constrained,

CONSTRAINT
   stan_SimplexConstraint__data_sums_to_one_in_col
   CHECK (argument_column('theta')),

CONSTRAINT
   stan_SimplexConstraint__when_summed_over_col
   CHECK (argument_column('customer_class_id'))
```

Man braucht den optionalen dritten Constraint **when_grouped_by_col**, um SimplexBedingungen pro Wertegruppe ausdrücken zu können, in denen es heißt „Pro Wert in Spalte C summieren sich die Werte der Spalte A zu einer 1 über Spalte B auf“. Ein Beispiel: die jeweiligen Wahrscheinlichkeiten für den Kauf eines konkreten Artikels $v \in V$ summieren sich pro Kundenklasse $k \in K$ zu einer Eins auf: $\phi \in (0,1)^{K \times V},\ \forall k \in K : \sum_V \phi_{kv} = 1$. D.h. Summe aller *phi*-Werte, die zur selben Kundenklasse gehören, ergibt eine 1. Für SQL-Notation siehe Listing 3.3:

Listing 3.3: Beispiel-Anwendung von Simplex-Constraints auf Spalte phi

```
phi stan_parameter_simplex_constrained,

CONSTRAINT
   stan_SimplexConstraint__data_sums_to_one_in_col
   CHECK (argument_column('phi')),

CONSTRAINT
   stan_SimplexConstraint__when_summed_over_col
```

```
    CHECK (argument_column('product_id')),

CONSTRAINT
    stan_SimplexConstraint__when_grouped_by_col
    CHECK (argument_column('customer_class_id'))
```

Ein möglicher, im SQL2Stan-Prototypen nicht implementierter **Verbesserungsvorschlag**: mehrere Simplex-Constraints in ein und derselben Tabelle ließen sich ausdrücken, indem man jeder Constraint-Gruppe (*data_sums_to_one* + *when_summed_over_col* + eventuell *when_grouped_by_col*) einen eindeutigen Bezeichner im Präfix des Constraint-Namens vergibt, z.B. *stan_SimplexConstraint___1*.

Constraints für diskrete Modellparameter Der Spaltentypen *stan_parameter_int* soll – wie bereits erwähnt – dazu dienen, die Referenzen zwischen den Tabellen (Fremdschlüssel) als unbekannt zu markieren. Zu klären wäre nur noch, welche Schlüsselspalte in welcher Tabelle referenziert werden soll. Zum Beispiel, bei beobachteten Fremdschlüssel-Referenzen wird schlichtweg der Spaltentyp *integer REFERENCES tablename* genommen – d.h. darin steckt die Information darüber, welche Schlüsselspalte aus welcher Tabelle referenziert wird. Bei unbeobachteten Referenzen wird der Spaltentyp *stan_parameter_int* eingesetzt, d.h. syntaktisch fallen in diesem Falle das Wort *REFERENCES* sowie der Name der referenzierten Tabelle weg. Die Information zur referenzierten Tabelle wird daher in einem **refers_to_table**-Constraint mitgegeben (siehe SQL-Auschnitt weiter unten). Zum Schluss wird man bei den diskreten Modellparametern noch mitteilen müssen, ob sie unabhängig sind oder abhängig von benachbarten Spaltenwerten (**independent**-Constraint). Zur Übersicht – die zwei Constraints für diskrete Modellparameter:

1. stan_Constraint___**int_parameter_independent**
   - CHECK-Argument: true | false
2. stan_Constraint___**int_parameter_refers_to_table**
   - CHECK-Argument: argument_table('Name der referenzierten Tabelle')

An dieser Stelle sollte man direkt anmerken, dass der SQL2Stan-Prototyp noch keine umgesetzten Lösungen für Fälle parat hält, wo diskrete Modellparameter von benachbarten Spaltenwerten abhängen (d.h. *int_parameter_independent* = false). Die Constraint-Spezifikation sowie die Übersetzung von SQL nach Stan wurden im Prototypen für diese Möglichkeit nicht ausgebaut. Es fällt mir außerdem schwer, ein Beispiel dafür zu finden, wo eine solche Art der Modellparameterabhängigkeit zum Tragen kommen könnte; ich würde solche Fälle dennoch nicht ausschließen.

Als Beispiel für Constraints zu diskreten Modellparametern – ein Ausschnitt aus der Definition der Tabelle Customer (integriertes Datenschema mit dem unsupervised-naïve-Bayes-Klassifikator). Die Kundenklasse ist unbeobachtet/latent (daher auch unsupervised – den Kunden werden im Vorfeld keine Klassen zugewiesen), die entsprechende Modellvariable ist unabhängig von den benachbarten Werten. Im SQL2Stan-Dialekt wird dieses Beispiel wie im Listing 3.4 formuliert:

Listing 3.4: Beispiel: SQL-Beschreibung eines diskreten Modellparameters

```
customer_class_id stan_parameter_int,

CONSTRAINT
   stan_Constraint__int_parameter_independent
   CHECK (true),
CONSTRAINT
   stan_Constraint__int_parameter_refers_to_table
   CHECK (argument_table('customer_class'))
```

Ein nichtimplementierter **Verbesserungsvorschlag**: sollte es sich herausstellen, dass bei manchen Bayes'sche Modelle mehrere diskrete Modellparameter in derselben Tabelle untergebracht werden, so ist folgende mögliche Änderung des Constraint-Dialekts nicht abwegig. Die Constraint-Argumente könnten um eindeutige Bezeichner (Spaltennamen) erweitert werden, wie im Listing 3.5 geschildert:

Listing 3.5: Vorschlag: mehrere diskrete Modellparameter pro Relation

```
-- NUR EIN VERBESSERUNGSVORSCHLAG für mehrere...
-- ...diskrete Modellparameter in einer Tabelle!
-- Diese Ausdrucksweise wurde nicht implementiert!

id_1 stan_parameter_int,
id_2 stan_parameter_int,

CONSTRAINT
   stan_Constraint__int_parameter_independent
   CHECK (args_column_boolean('id_1, true')),
CONSTRAINT
   stan_Constraint__int_parameter_refers_to_table
   CHECK (args_column_table('id_1, table_one')),
CONSTRAINT
   stan_Constraint__int_parameter_independent
   CHECK (args_column_boolean('id_2, false')),
CONSTRAINT
   stan_Constraint__int_parameter_refers_to_table
   CHECK (args_column_table('id_2, table_two'))
```

Arrayspezifische SQL-Constraints im integrierten Schema

In einem Satz: arrayspezifische Constraints im integrierten Schema Aus jeder Spalte im integrierten Schema wird ein Array (da Inferenzalgorithmen vorzugsweise mit Arrays arbeiten und weil Array-DBMS als Infrastruktur viel Potenzial haben), und die Dimensionalität der Arrays wird über Tabellenschlüssel festgelegt; mit extra Constraints (Reihenfolge der Schlüsselspalten) wird die Hierarchie der aus den Schlüsselspalten entstehenden Dimensionen pro Tabelle festgelegt.

Mit Sorting-Order-Constraints zur Array-Ansicht Diese Art von Constraints findet man in jedem Create-Table-Statement aus den SQL2Stan-Beispielmodellen. Diese Constraints dienen zum Abbilden von Daten aus der Datenbank in die Array-Ansicht. Ohne sie ist es auf der syntaktischen Ebene leider nicht möglich, Tabellenspalten als eine Abstraktion für Arrays zu verwenden. Nach dem aktuellen Stand ist die automatische Inferenz mit flexibler Zielfunktions-Spezifikation kein Teil eines DBMS [1]. In diesem Kontext kann man PostgreSQL – das DBMS im SQL2Stan-Prototyp – nicht einfach so als Datenquelle an eine Inferenzsoftware anbinden. Die Software Stan als Backend zur automatischen Inferenz verdaut keine Tabellenspalten, sondern nur Vektoren, Arrays oder linearisierte Matrizen (inkl. Angabe von Matrizendimensionen). Unser relationales DBMS kann diese Vektoren und Matrizen in linearisierter Form bereitstellen (eine Tabelle sortieren und eine Spalte rauskopieren); ein Array-DBMS könnte die Arrays sogar direkt zur Verfügung stellen. Dementsprechend lautet die allgemeine Abstraktionsidee: aus jeder Tabellenspalte im integrierten Schema entsteht ein Vektor oder ein Array in Stan. Die Dimensionsangaben für die Arrays werden den Tabellenschlüsseln entnommen. Doch bei mehreren Dimensionen (bei zusammengesetzten Schlüsseln) muss geklärt werden, in welcher Reihenfolge sie angeordnet werden müssen, um einen sinnvollen Datenzugriff auf die Vektoreninhalte zu gewährleisten.

Ein Beispiel: Spalte *phi* aus der Tabelle customer_class_product wird zu einem Vektor $\overrightarrow{phi}$. Der zusammengesetzte Schlüssel der Tabelle customer_class_product besteht aus zwei Spalten: *customer_class_id* und *product_id*. Die Anzahl der Dimensionen von $\overrightarrow{phi}$ – offensichtlich zwei – wäre damit schon mal geklärt. Doch welche Reihenfolge von den Tabellenschlüsseln soll man zwecks Vektorwerte-Zugriff nehmen: *phi*[customer_class_id,product_id] oder *phi*[product_id,customer_class_id]?

Diese Entscheidung soll der Nutzer treffen. Sie hängt davon ab, wie und wofür *phi* eingesetzt wird. Ein kleines Brainstorming für die erste Variante (*phi*[customer_class_id,product_id]): man greift auf den zweidimensionalen *phi*-Vektor über seine Dimensionen „von vorn nach hinten“ zu, d.h. angefangen mit einer Kunden-ID. So erhält man über diesen Zugriff einen Vektor mit *phi*-Werten, der so groß ist wie die Anzahl von Produkt-IDs. Die Dimensionalität [customer_class_id,product_id]

[1] Man kann „nur so tun, als ob“: eine deklarative Spezifikation der Inferenzaufgabe schreiben, Daten in die DB reinladen, abwarten und sich über die automatisch in die DB eingelesenen Schätzergebnisse freuen.

würde damit also nahelegen, die *phi*-Einträge zu unterschiedlichen Produkt-IDs nach Kunden-IDs zu gruppieren. Bei der anderen, ebenfalls möglichen Dimensionalität [product_id,customer_class_id] sähe die Gruppierung der *phi*-Einträge genau umgekehrt aus: die *phi*-Einträge zu unterschiedlichen Customer-IDs wären nach Produkt-IDs gruppiert. Für welche Variante soll man sich nun entscheiden, für die erste oder für die zweite? Ein Detail kann sich an der Stelle als hilfreich erweisen: für *phi* soll ein Simplex-Constraint gewährleistet werden, sodass die Summe aller *phi*-Werte, die zur selben Kundenklasse gehören, eine 1 ergibt. Demnach werden die *phi*-Einträge erst nach Kundenklasse gruppiert, um dann aufsummiert zu werden. „Erst nach Kundenklasse gruppieren“ – das klingt ganz nach einer Reihenfolge! Daher sollte die Wahl auf den ersten Vorschlag fallen (*phi*[customer_class_id,product_id]). Das Listing 3.6 ist ein SQL-Codeausschnitt aus der Customer_class_Product-Tabellendefinition; darin wird die Sortierreihenfolge aus Schlüsselspalten der genannten Tabelle festgelegt, sodass die Spalten dieser Tabelle eindeutig auf mutlidimensionale Arrays abgebildet werden können.

Listing 3.6: Beispiel: Sorting-Order-Constraint für PK-Spalten

```
CONSTRAINT SortingOrder_Position_1
   CHECK (argument_column('customer_class_id')),
CONSTRAINT SortingOrder_Position_2
   CHECK (argument_column('product_id'))
```

Im obigen Beispiel hatte man das Glück, eine hilfreiche Nebenbedingung (einen Simplex-Constraint) gehabt zu haben. Falls man als SQL2Stan-Programmierer solch eine Tabellenbedingung einfach nicht hat, muss man via SortingOrder-Constraint dennoch irgendwie eine PK-Spalten-Reihenfolge festlegen. In diesem Falle kann man einfach folgende Faustregel nutzen: die Dimensionen bzw. Schlüsselspalten im SortingOrder sollen aufsteigend nach ihrer Größe (Anzahl von den umfassten Entitäten) sortiert werden. Mit anderen Worten: die erste Dimension, über die auf Array zugegriffen wird, soll die kleinste sein, die zweite Dimension – die zweitkleinste und so weiter.

Die Tabellenspalten werden also zu Vektoren und Arrays. Ihre Dimensionen werden durch Tabellenschlüssel festgelegt. Die Reihenfolge der Zugriffe über diese Dimensionen soll vom Nutzer mit Hilfe von Sorting-Order-Constraints festgelegt werden. Die Sorting-Order-Constraints legen eine Reihenfolge der Schlüsselspalten fest, damit die Tabelleninhalte danach sortiert werden können. Die Sortierreihenfolge entspricht in unserem Falle der Dimensionenreihenfolge beim Vektor-/Array-Zugriff. Die Syntax der Sortierreihenfolge-SQL-Constraints schaut wie folgt aus:

- **SortingOrder**_Position_i
  - i = Platznummer in der Reihenfolge
  - CHECK-Argument: argument_column('Tabellenschlüssel-Spalte')

3.2.3. Übersetzung des integrierten Schemas nach Stan

Die Inferenzsoftware Stan [2] wird im SQL2Stan-Prototypen als Backend für die Bayes'sche Inferenz genutzt. Die Tatsache, dass der SQL-Code genug Informationen liefern kann, um automatisch die Bayes'sche Inferenz zu implementieren, ermutigt dazu, weitere Compiler (nicht nur für Stan) zu bauen, um aus den nutzergeschriebenen SQL-Spezifikationen evtl. andere Inferenzimplementierungen zu generieren. Somit gilt das Ziel, in SQL sowohl die Datenmanagementdetails als auch die Inferenzimplementierung zu abstrahieren, erreicht. In diesem Kapitel wird erklärt, wie das integrierte Datenschema in ein Teil eines generierten probabilistischen Stan-Programm übersetzt wird.

Verwendete Tools Das integrierte Schema besteht aus Create-Table-Statements (ein SQL-Statement pro Tabelle). Jedes von diesen Statements steht in einer einzelnen Textdatei (*.sql) und wird vom Parser namens **libpg_query** [Fit19] geparst. Das Parsing-Ergebnis sind Parserbäume im JSON-Format. Sie werden in einer JSON-Abfragesprache Jsonata (implementiert im Projekt **jfq** [Blu19]) ausgewertet. Zuerst werden die Informationen über das integrierte Schema gesammelt [3] und als JSON-formatierte „Dictionaries" gespeichert. „Dictionaries" dienen zum leichteren Nachschlagen von dem aus den Parserbäumen bereits extrahierten Wissen, ohne dass man es jedes Mal nachschlagen muss. Nach dem Füllen von den „Dictionaries" wird mit Hilfe von Jsonata/jfq sowie den Mitteln des bash-Terminals Stan-Code erzeugt.

Programmstruktur von Stan In der PPL Stan formuliert der Programmierer eine Zielfunktion – genauer genommen, eine für das jeweilige Modell spezifische logarithmierte Wahrscheinlichkeitsdichtefunktion, die an ihrem Maximum die wahrscheinlichste Wertebelegung für die versteckten Modellparameter besitzt. Stan ermittelt das Maximum der Zielfunktion und gibt die geschätzten Wertebelegungen für unbeobachtete Modellvariablen zurück. Die Stan-Sprache selbst basiert auf dem Prinzip der Blöcke. Die drei wichtigsten Programmblöcke in Stan sind:

- **data**-Block: für die Inferenz benötigte beobachtete Eingabedaten
- **parameters**-Block: durch Inferenz zu schätzende Modellparameter
- **model**-Block: Angabe der statistischen Modellstruktur, d.h. hier hat die Zielfunktion – die logarithmierte Wahrscheinlichkeitsdichtefunktion – zu stehen (dazu kommen wir noch später)

Die ersten zwei Stan-Blöcke – **data** und **parameters** – können durch Übersetzung des integrierten Datenschemas generiert werden.

[2] Die probabilistische Programmiersprache Stan trägt den Namen der ML-Software Stan.

[3] Informationen wie: welche Spalten gibt es, stehen darin gegebene Daten oder latente Parameter, in welchen Tabellen stehen sie, wie sind sie dimensioniert, wie sieht der Stan-Code für sie aus usw.

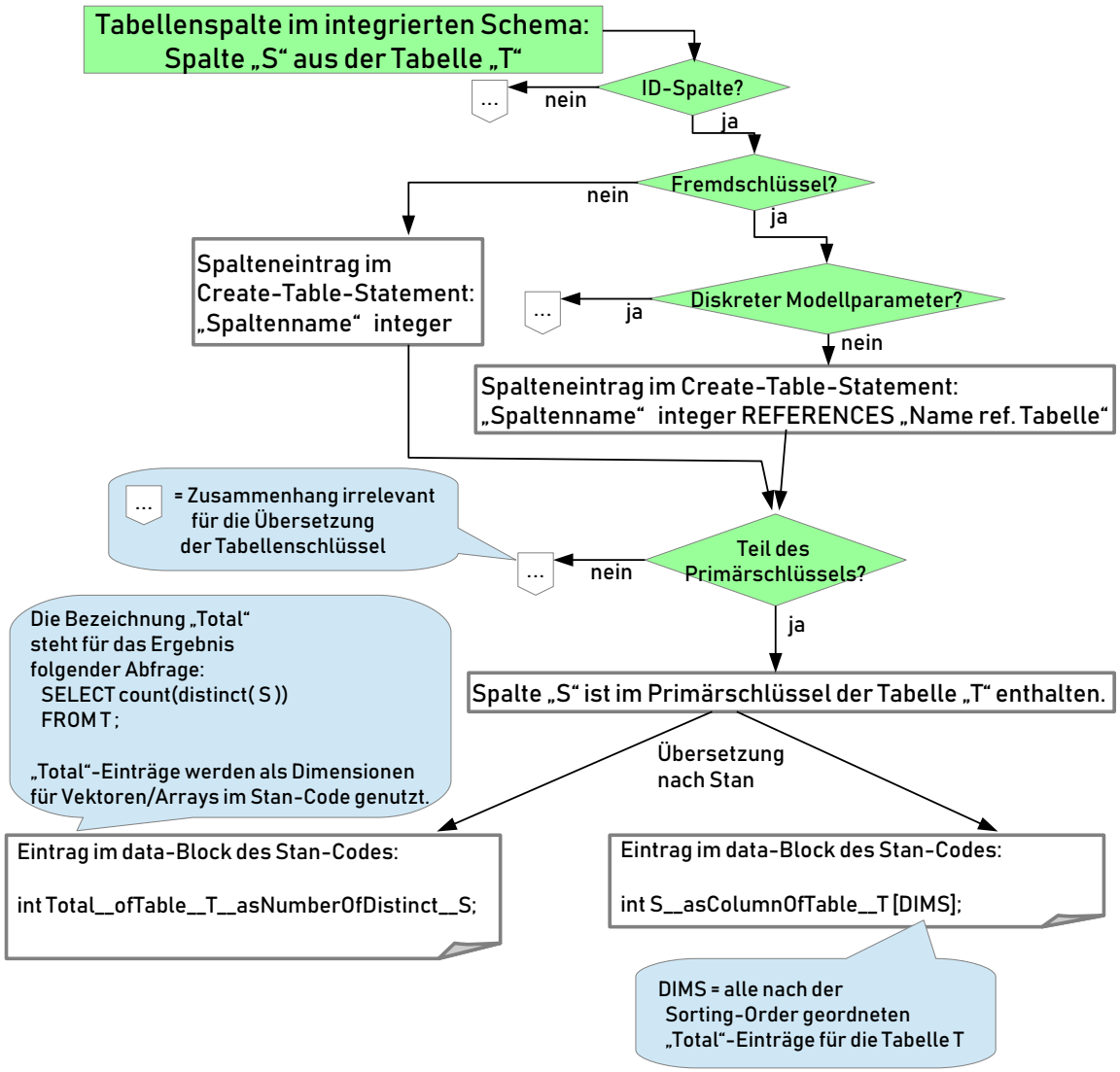


Abbildung 3.9.: Aus einer Tabellenschlüsselspalte S aus dem Create-Table-Statement der Tabelle T werden im **data**-Block des Stan-Code zwei Einträge erzeugt.
Das Diagramm hat folgende Semantik: grüne Rauten sind Verzweigungen, weiße Rechtecke (ohne Knickkante unten) – Beschreibungen der beobachteten Modellvariablen, weiße Rechtecke mit geknickter unterer Kante – generierter Stan-Code, blaue „Sprechblasen“ mit abgerundeten Ecken enthalten Kommentare. Irrelevante Zweige wurden weggelassen.

Übersetzung der Tabellenschlüssel (Abb. 3.9) Aus den ganzzahligen ID-Spalten der Tabellen (Tabelleschlüssel) entstehen zweierlei Arten von Einträgen. Die erste Art von Einträgen sind Dimensionen – ganz im Einklang mit der Idee, dass aus Tabellenspalten Arrays entsprechen, und dass die Schlüsselspalten der Tabellen als Dimensionen von Arrays verwendet werden. Es ist also erwünscht, dass aus einer Tabelleschlüssel-Spalte eine Dimension wird. Eine Dimension wird beschrieben durch die Anzahl von unterschiedlichen Werten, die in einer PK-Spalte vorkommen – eine integer-Zahl, die die Dimensionsgröße vorgibt. Der Name solcher integer-Einträge fängt mit dem Präfix „Total“ (deutsch: Gesamtmenge) an. Diese *Total*-Einträge erscheinen im **data**-Block des generierten Stan-Programms, und werden nach folgendem Muster gebildet:

```
int Total__ofTable__TABLENAME__asNumberOfDistinct__COLNAME;
```

TABLENAME sei Platzhalter für den Tabellennamen, und *COLNAME* eine Veränderliche für den Spaltennamen. Die zweite Art von Einträgen kommt vom Ansatz, jede Tabellenspalte in ein Array zu übersetzen. Diese aus den PK-Spalten übersetzten Arrays kommen als integer-Arrays in den **data**-Block des generierten Stan-Programms – d.h. dorthin, wo Stan-Einträge für beobachtete Modellvariablen platziert sind. Anbei folgt das syntaktische Gerüst für solche Stan-Einträge:

```
int COLNAME__asColumnOfTable__TABLENAME[DIMS];
```

DIMS seien die nach SortingOrder geordneten *Total*-Einträge für *TABLENAME*.

Übersetzung der Spalten vom Typ *stan_data_real* Die nicht-ganzzahligen Eingabedaten für Bayes'sche Inferenz (z.B. Werte für Modell-Hyperparameter) werden im **data**-Block des übersetzten Stan-Programms untergebracht, als Vektoren aus reellen Zahlen. So schaut das syntaktische Muster solcher Stan-Vektoren aus:

```
vector[LASTDIM] COLNAME__asColumnOfTable__TABLENAME[OTHERDIMS];
```

An der Stelle spielen die durch die Tabellenschlüssel entstehenden Dimensionen eine Rolle, ebenso wie die Reihenfolge der Schlüsselspalten, die durch den Sorting-Order-Constraint festgelegt wurde. Im Sorting-Order-Constraint werden alle Spalten des primären Tabellenschlüssels in einer bestimmten Reihenfolge aufgezählt. Die Mustervariable *LASTDIM* steht für den *Total*-Eintrag für die letzte Schlüsselspalte in dieser Reihenfolge. Der Platzhalter *OTHERDIMS* im Übersetzungsmuster repräsentiert alle übrigen geordneten *Total*-Einträge für die Tabelle *TABLENAME* in der durch den Sorting-Order-Constraint angegebenen Sortierreihenfolge.

Übersetzung der Modellparameter-Spalten (Abb. 3.10) Inferenzausgabe-Spalten sind im integrierten Schema entweder vom Typ „stan_parameter_real“ oder „stan_parameter_simplex_constrained“. Sie werden in Einträge im **parameters**-Block des generierten Stan-Programms übersetzt. Eine Spalte vom Typ „stan_parameter_real“ wird nach folgendem Muster in Stan übersetzt:

```
vector[LASTDIM] COLNAME__asColumnOfTable__TABLENAME[OTHERDIMS];
```

Das Übersetzungsmuster für „stan_parameter_simplex_constrained“-Spalten sieht fast genauso aus („simplex“ statt „vector“; ebenfalls **parameters** als Zielblock):

```
simplex[LASTDIM] COLNAME__asColumnOfTable__TABLENAME[OTHERDIMS];
```

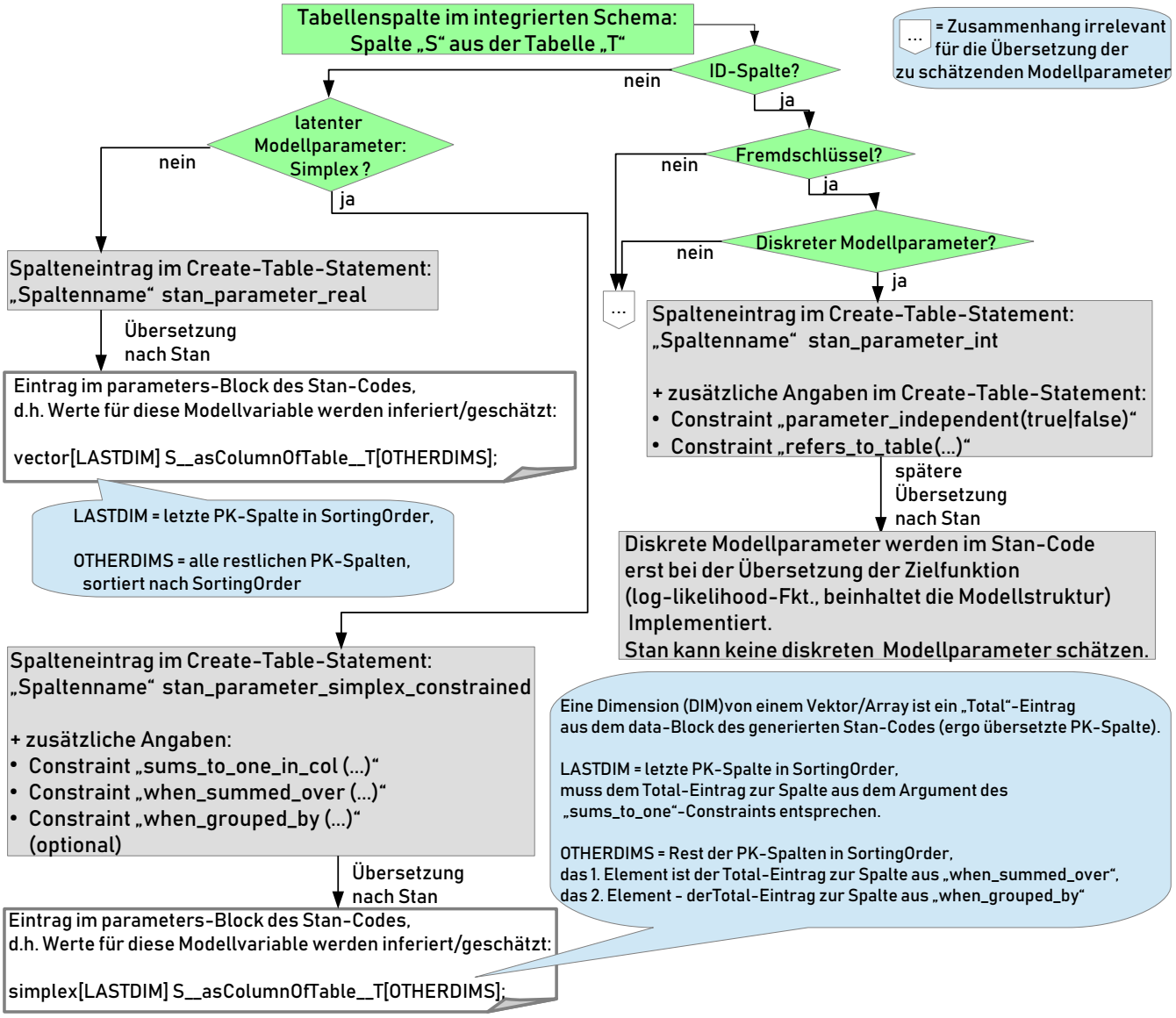


Abbildung 3.10.: Übersetzung von Modellparameter-Spalten nach Stan. Grüne Rauten sind Verzweigungen, graue Rechtecke – Beschreibungen latenter Modellparameter, weiße Rechtecke mit geknickter unterer Kante – generierter Stan-Code, blaue „Sprechblasen“ mit abgerundeten Ecken enthalten Kommentare. Irrelevante Zweige wurden weggelassen.

Spaltenimport und Sorting-Order-Constraints Im SQL2Stan-Prototypen werden Daten zwischen Stan und PostgreSQL bewegt; dazu muss man sie leider kopieren (prototypische Einschränkung). Man kann im SQL2Stan-Prototypen also nur vorgeben, die Backends würden nahtlos miteinander arbeiten – wichtig beim Prototypen war nur der Entwurf eines SQL-Dialekts. Die durch Sorting-Order-Constraints festgelegte Tabellen-Sortierreihenfolge ist u.a. wichtig beim Exportieren der Daten aus Spalten. Aus der deklarativen Beschreibung wird eine imperative Implementierung mit Vektoren und Arrays erzeugt, die noch gefüllt werden müssen. Die Syntax für Stan-Vektoren/-Arrays schaut (laut Abb. 3.10 und 3.9) vereinfacht wie folgt aus:

- vector[Dimensionen[letzte_Dimension]] *stan_beispielvektor*[Dimensionen[alle anderen bis auf die letzte]]
  - „Dimensionen“ entsprechen Sorting-Order aus den Tabellenconstraints
- int *stan_beispielarray*[Dimensionen]
  - „Dimensionen“ entsprechen Sorting-Order aus den Tabellenconstraints

Nun zum eigentlichen Datenimport im SQL2Stan-Prototyp. Die benutzerdefinierte Tabellenschlüssel-Sortierreihenfolge soll uns helfen, die Daten aus der relationalen PostgreSQL-Datenbank herauszuholen und sie in Vektoren/Arrays zu packen. Im SQL2Stan-Prototyp werden die Sorting-Order-Constraints auch dazu genutzt, die Vektor-/Array-Einträge mit Daten aus der Tabelle zu füllen. Dazu werden die entsprechenden Spaltendaten über SQL abgefragt (wie im Listing 3.7), in eine CSV-Datei ausgelagert, der CSV-formatierte Inhalt wird anschließend in eine textuelle Inferenz-Eingabedatei für die Inferenz-Software Stan eingelesen.

Listing 3.7: Beispiel: Bereitstellung einer Spalte zum CSV-Export

```
SELECT beobachtete-modellvariable FROM tabelle ORDER BY
   SortingOrder;
```

Der CSV-formatierte, linearisierte Spalteninhalt wird anschließend in eine textuelle Inferenz-Eingabedatei (Dateinamenendung *.data.R) für Stan eingebunden, siehe Listing 3.8:

Listing 3.8: Bereistellung der beobachteten Daten zum Stan-Programm

```
stan-eintrag_zu_einer_gefüllten_tabellenspalte <- structure(
  c( linearisierter Spalteninhalt ),
  .Dim=c( SortingOrder )
)
```

Das Listing 3.8 verfügt über folgende Legende:

- „stan-eintrag_zu_einer_gefüllten_tabellenspalte“ = Vektor/Array/Matrix aus dem **data**-Block des generierten Stan-Codes, korrespondiert zu einer beobachteten Modellvariable.

- „linearisierter Inhalt“ = linearisierter Inhalt des Vektor/-Array/-Matrix, geschöpft aus dem CSV-formatierten Tabellenspalten-Dump. D.h. das, was in einer Tabellenspalte als beobachtete Modelldaten in der Datenbank lag, wurde in eine CSV-Datei exportiert. Der Inhalt dieser CSV-Datei wird im Stan-kompatiblen linearisierten Format (Werte kommagetrennt, ohne Zeilenumbrüche) als Inferenzeingabe bereitsgestellt.

- „SortingOrder“ = Namen der Stan-Einträge, die als Dimensionen für den Vektor/Array/Matrix namens „stan_eintrag“ fungieren. Sie sind sortiert nach der Reihenfolge, die durch SortingOrder-Constraints im jeweiligen Create-Table-Statement vorgegeben wurde.

Übersetzungsbeispiel Zu Verdeutlichung übersetzen wir einfach eine Beispiel-Tabelle, die aus einem integrierten Schema stammt. Das Create-Table-Statement für eine Beispiel-Tabelle sei im Listing 3.9 aufgeführt:

Listing 3.9: Beispiel einer Tabellendefinition im SQL2Stan-Dialekt

```
CREATE TABLE customer_class_product (
        customer_class_id   integer
                REFERENCES customer_class,
        product_id   integer
                REFERENCES product,
        phi   stan_parameter_simplex_constrained,

        PRIMARY KEY (customer_class_id, product_id),

        -- modellspezifische Constraints:
        CONSTRAINT
           stan_SimplexConstraint__data_sums_to_one_in_col
           CHECK (argument_column('phi')),
        CONSTRAINT
           stan_SimplexConstraint__when_summed_over_col
           CHECK (argument_column('product_id')),
        CONSTRAINT
           stan_SimplexConstraint__when_grouped_by_col
           CHECK (argument_column('customer_class_id')),

        -- arrayspezifische Constraints:
        CONSTRAINT SortingOrder_Position_1
           CHECK (argument_column('customer_class_id')),
        CONSTRAINT SortingOrder_Position_2
           CHECK (argument_column('product_id'))
);
```

Der SQL-Code für eine Tabelle aus dem relationalen Schema kann vom Stan-Compiler nach Stan übersetzt werden. Die aus den Tabellenspalten erzeugten Stan-Einträge finden im generierten Stan-Programm ihren Platz sowohl im **data**- (beobachtete Daten) als auch im **parameters**-Block (zu schätzende Modellparameter).

SQL2Stan übersetzt den SQL-Code einer Tabellendefinition aus dem Listing 3.9 in den Stan-Code, der im Listing 3.10 zu sehen ist.

Listing 3.10: Von SQL2Stan aus Tabellendefinition 3.9 generierter Stan-Code

```
data {
int Total__ofTable__customer_class_product__asNumberOfDistinct__customer_class_id;
int Total__ofTable__customer_class_product__asNumberOfDistinct__product_id;

int customer_class_id__asColumnOfTable__customer_class_product[
    Total__ofTable__customer_class_product__asNumberOfDistinct__customer_class_id,
    Total__ofTable__customer_class_product__asNumberOfDistinct__product_id];
int product_id__asColumnOfTable__customer_class_product[
    Total__ofTable__customer_class_product__asNumberOfDistinct__customer_class_id,
    Total__ofTable__customer_class_product__asNumberOfDistinct__product_id];
}

parameters {
simplex[Total__ofTable__customer_class_product__asNumberOfDistinct__product_id]
    phi__asColumnOfTable__customer_class_product[
    Total__ofTable__customer_class_product__asNumberOfDistinct__customer_class_id];
}
```

3.3. Data model driven probabilistic programming

Die Aspekte rund um das relationale DMS-Schema wurden somit geklärt. Dennoch: zur Implementierung der Bayes'schen Inferenz reicht ein relationales Schema allein nicht aus; dazu muss ein probabilistisches Programm in SQL geschrieben werden. Der Leser wird im diesem Kapitel erfahren, wie das in SQL2Stan geht.

In drei Sätzen: probabilistische Programmierung in SQL Beim Explizitmachen von Informationen, die im modellspezifischen Bayes'schen Netz enthalten sind, lässt sich ein statistisches Bayes'sches Modell durch eine logarithmierte Wahrscheinlichkeits(dichte)funktion (auch: zu optimierende Zielfunktion, log-likelihood-Funktion) beschreiben. Diese Funktion lässt sich in SQL als Summenabfrage ausdrücken, so können in diese SQL-Abfrage die Entitäten des integrierten relationalen Schemas einbezogen werden. Aus dieser SQL-Summenabfrage lässt sich automatisch ein probabilistisches Programm generieren, welches die automatische Bayes'sche Inferenz an dem in SQL beschriebenen Modell implementiert.

3.3.1. Wahrscheinlichkeitsfunktionen zur Modellspezifikation

Bayes'sche Modelle kann man grafisch als Bayes'sches Netz oder aber auch über mathematische Zielfunktionen beschreiben. Eine Möglichkeit wäre die Modellspezifikation über eine Wahrscheinlichkeitsfunktion (eine statistische Formel mit Zufallsvariablen). Sie ist so aufgebaut, dass man in diese Formel für Variablen konkrete Werte einsetzen kann, und ihr Funktionswert gibt an, wie wahrscheinlich (bzw. glaubwürdig) es ist, dass die eingesetzten konkreten Werte der Grundgesamtheit entstammen. Eine derartige modellspezifische Zielfunktion ist ein Produkt aus Wahrscheinlichkeiten und kann z.B. wie die Formel 3.1 (LDA-Zielfunktion, nicht logarithmiert, nicht vereinfacht) aussehen.

$$p(d, z, \mu, \theta|\alpha, \beta) = \prod_{n \in N} Dirichlet(\theta_n|\alpha) \cdot \prod_{k \in K} Dirichlet(\mu_k|\beta_k) \cdot \prod_{n \in N} \prod_{m \in M_n} Categorical(z_{n_m}|\theta_n) \cdot \prod_{n \in N} \prod_{m \in M_n} Categorical(d_{n_m}|\mu_{z_{n_m}}) \tag{3.1}$$

Man wird eine solche Zielfunktion logarithmieren (dann wird sie zu einer Summe aus logarithmierten Wahrscheinlichkeiten) und in SQL aufschreiben wollen. Der springende Punkt dabei ist, dass man zuallererst eine solche Formel haben muss, um sie anschließend zu logarithmieren und in SQL zu formulieren. Folgende Paragraphen werden den Leser durch die Ideen dazu begleiten, wie man anhand eines Bayes'schen Modells (gegeben als ein Bayes'sches Netz, z.B. wie in Abb. 3.11) an eine solche Formel kommt.

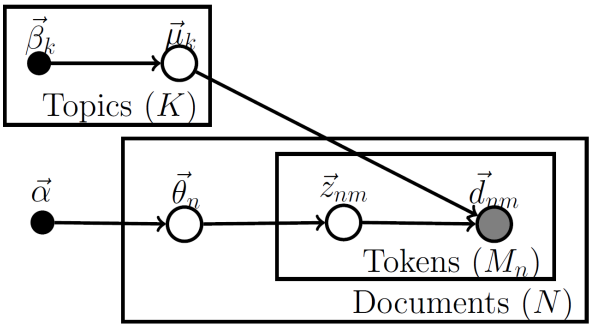


Abbildung 3.11.: Bayes'sches Netz (BN) für Latent Dirichlet Allocation (LDA [BNJ03]) [RH16, S. 10], entspricht der ZF-Formel 3.1.

Wie kommt man an eine in SQL zu beschreibende Zielfunktion?

Bisher wurde dargelegt, dass Bayes'sche Modelle (grafisch dargestellt in Form von Bayes'schen Netzen) in relationale Schemata nach [RH16] übersetzt werden können. Diese Veröffentlichung legt dar, wie man die im Bayes'schen Netz implizit enthaltenen datenmanagementspezifischen Details explizit macht. Es wird allerdings nicht präsentiert, wie man von einem Bayes'schen Netz (wie z.B. in Abb. 3.11 dargestellt) zu einer mathematischen Beschreibung des Bayes'schen Modells kommt. Das ist dennoch wichtig, wenn man wie bei SQL2Stan die probabilistische Programmierung in SQL im Sinn hat, um automatische Inferenz zu implementieren. Für probabilistische Programmierung (zumindest in der PPL Stan) wird eine Wahrscheinlichkeitsfunktion gebraucht, die die Modellstruktur explizit beschreibt und die der mathematischen Beschreibung des Modells entspricht. Demzufolge müssen die im Bayes'schen Netz implizit enthaltenen Informationen, die für die Bayes'sche Inferenz relevant sind, explizit gemacht werden.

Das wichtigste, was ein SQL2Stan-Nutzer dazu braucht, ist das Verständnis von dem Bayes'schen Modell, welches anfangs grafisch als Bayes'sches Netz vorliegt. Ohne dieses Verständnis kann man das Bayes'sche Netz sowieso nicht in das relationale Schema nach [RH16] übersetzen. Diese Übersetzung (u.a. durch ihre Zwischenergebnisse in Form des Atomic Plate Model) kann hilfreich sein, wenn man anhand eines Bayes'schen Netzes (BN) eine mathematische Zielfunktion für das entsprechende Bayes'sche Modell formulieren möchte.

In einem Bayes'schen Netz – einem gerichteten azyklischen Graphen – sind Knoten Zufallsvariablen und Kanten Wahrscheinlichkeitsverteilungen. Darin gibt es außerdem mit Indexbereichen beschriftete „Plates" zum Zusammenfassen mehrerer gleichverteilter Knoten. Liegt, zum Beispiel, der Knoten der Zufallsvariable z auf einer mit M beschrifteten Plate, dann ist so zu interpretieren, dass es M-viele Zufallsvariablen z_m gibt.

Ein Bayes'sches Netz kann in eine Zielfunktion überführt werden, indem jeder BN-Knoten mit eingehenden Kanten mathematisch als Teil einer Wahrscheinlichkeitsfunktion beschrieben wird. Sobald alle bedingten BN-Knoten in die mathematische Darstellung überführt worden sind, fügt man sie in einer Produktformel via Multiplikation zusammen und integriert dieses Produkt mit Hilfe des natürlichen Logarithmus. So erhält man aus einem Produkt eine Summe. Diese Summe drückt die Bayes'sche Modellstruktur genauso gut aus wie das Produkt (alle bedingten Zufallsverteilungen des Modells liegen ebenfalls vor). Die die Maxima der modellspezifischen Wahrscheinlichkeitsfunktion werden bei einer Summe genauso gut erfasst wie bei einem Produkt (Maximastellen von logarithmierten und nichtlogarithmierten Funktionen sind gleich aufgrund von Monotonie des natürlichen Logarithmus). Ferner ist eine Summe numerisch unproblematisch in der probabilistischen Auswertung und lässt sich in SQL auf eine einfache Weise als ein Summenstatement beschreiben.

Bedingte BN-Knoten werden zu expliziten Zufallsverteilungen Eine Verbindung im Bayes'schen Netz (eine gerichtete Kante geht vom Knoten A zum Knoten B) entspricht einer Zufallsverteilung, und ist somit im einfachsten Falle mathematisch als $p(\mathrm{B}|\mathrm{A})$ definiert. p ist ein Platzhalter für eine Zufallsverteilung (wie Dirichlet- oder Multinomial-), und durch das Modellverständnis soll man wissen, welche Art der Verteilung das ist. p muss dementprechend durch einen konkreten Verteilungsnamen ersetzt werden. Die Zufallsverteilung p ist parametrisiert, und die Parameter dieser Verteilung sind zwischen den runden Klammern nach dem p im Ausdruck $p(\mathrm{B}|\mathrm{A})$ untergebracht. Der Teil nach dem |-Strich (Notation für eine Verteilungsbedingung) – in diesem Falle nur A – nennt sich Prior; der Teil vor dem |-Strich (hier: B) nennt sich Posterior. Prior bedingt Posterior: $p(\mathrm{B}|\mathrm{A})$ ist zu interpretieren als Glaubwürdigkeit/Wahrscheinlichkeit für B gegeben A. Wenn der Prior selbst eine Zufallsverteilung ist, lässt sich $p(\mathrm{B}|\mathrm{Prior})$ interpretieren wie ein Wert für Glaubwürdigkeit/Wahrscheinlichkeit dafür, dass B aus der Prior-Verteilung durch ein Zufallsexperiment gezogen wird.

BN-Plates werden zu Produktzeichen und Indizes Aus den BN-Plates, auf denen der Posterior-Knoten einer Zufallsverteilung p liegt, sollen indexierte Produktzeichen vor dem mathematischen p-Verteilungsausdruck entstehen.

Liegt der Posterior einer Verteilung p auf einer oder mehreren Plates, so werden vor den p-Ausdruck Produktzeichen geschrieben – ein indexiertes Produktzeichen pro Plate, auf der der Posterior-Knoten im BN liegt. Die Laufinzides der Produktzeichen entsprechend den Dimensionen der BN-Plates, auf denen der Posterior-Knoten liegt, beginnend von der untersten/äußersten Plate (erstes Produktzeichen, am weitesten vor dem p-Ausdruck) bis zur obersten/innersten BN-Plate, auf der der Posterior-Knoten von p direkt draufliegt (Produktzeichen mit der Dimension der innersten Plate steht direkt vor dem p-Ausdruck). Sei A der Prior und B der Posterior von p (d.h. im BN gibt es eine Kante von A nach B), und der BN-Knoten von B liege auf einer mit N beschrifteten Plate, so wird daraus $\prod_{m \in M} p(\mathrm{B}_{\mathrm{Index}}|\mathrm{A}_{\mathrm{Index}})$. Es kann natürlich sein, dass die M-Plate selbst auf einer anderen Plate liegt, z.B. auf der N-Plate (wie in Abb. 3.11), sodass der Posterior-Knoten praktisch auf zwei aufeinanderliegenden Plates gleichzeitig positioniert ist. In diesem Falle kommt vor den mathematischen Ausdruck für die Verteilung p noch ein Produktzeichen hinzu (Vorgehensweise analog): $\prod_{n \in N} \prod_{m \in M_n} p(\mathrm{B}_{\mathrm{Index}}|\mathrm{A}_{\mathrm{Index}})$. Der Index vom inneren Produktzeichen wurde mit dem Index des äußeren Produktzeichens versehen, um die Hierarchie der den Posterior beherbergenden Plates wiederzugeben (Hierarchie: M-Plate liegt auf der N-Plate).

Somit wurde geklärt, wie aus den Posterior-Plates indexierte Produktzeichen vor dem Verteilungsausdruck entstehen. Erklärungsbedürftig bleibt somit nur noch das, was in dem Verteilungsausdruck selbst geschieht. Der Leser hat bereits im Ausdruck $\prod_{n \in N} \prod_{m \in M_n} p(\mathrm{B}_{\mathrm{Index}}|\mathrm{A}_{\mathrm{Index}})$ gesehen, dass die Indizes vom Prior und Posterior (ersetzt durch den Platzhalter „Index“) Fragen aufwerfen. Wie sind diese Posterior- und Prior-Indizes aufzubauen? Die Antwort hängt ebenfalls mit Plates zusammen;

kurz gesagt: diese Indizes entsprechen der Plates-Beschriftung von dem jeweiligen BN-Knoten, geschachtelt in der Hierarchie der Plates. Ich erkläre es etwas ausführlicher. Ob Posterior oder Prior: es ist immer ein BN-Knoten. Liegt ein solcher Knoten nicht auf einer Plate (wie z.B. α in der Abb. 3.11), dann gibt es für diesen Knoten keine Indexierung – der oben verwendete Platzhalter „Index“ im Index von A bzw. B bleibt in solchen Fällen leer. Liegt ein Posterior- oder Prior-Knoten auf einer Plate (wie z.B. β_k auf der K-Plate in der Abb. 3.11), dann übernimmt er zwischen den runden Klammern seiner p-Verteilung die Beschriftung dieser Plate als Index. Liegt ein Posterior- oder Prior-Knoten auf mehreren, aufeinanderliegenden Plates (wie z.B. z_{n_m} auf der M_n-Plate in der Abb. 3.11), dann übernimmt er die Beschriftungen dieser Plates als Index, und zwar geschachtelt und in der Reihenfolge der Hierarchie von diesen Plates (von „unten“/außen nach „oben“/innen).

Was lässt sich vereinfachen? In manchen Fällen, wo die Beschriftung einer inneren Plate auch ohne die Einbeziehung der äußeren Plate eindeutig ist, ist das indexierte Produktzeichen der äußeren Plate nicht notwendig. Ein Beispiel dafür (siehe Abb. 3.11)): die Zufallsvariable z liege im Bayes'schen Netz auf einer M_n-Plate, die wiederum selbst auf einer N-Plate liegt. Man nehme an, der Wert von z entspricht einer Thema, $n \in N$ stehe für die Dokument-IDs im Textkorpus, und $m \in M_n$ stehe für Wortinstanz-IDs im Dokument n aus der Menge der Dokumente N. In diesem Falle muss z in der mathematischen ZF-Beschreibung mit zusätzlichen, hierarchisch geschachtelten Indizes für $n \in N$ und $m \in M_n$ versehen werden, mit dem Ergebnis z_{n_m}. Der Aufruf einer Multinomialverteilung mit z_{n_m} als Posterior schaut in diesem Falle so aus: $\prod_{n \in N} \prod_{m \in M_n} \mathit{Categorical}(z_{n_m}|\theta_n)$ (siehe Formel 3.1 bezogen auf die BN-Abbildung 3.11). Was passiert, wenn die Wortinstanz-IDs nicht nur pro Dokument $n \in N$ eindeutig sind, sondern eindeutig global, im ganzen Textkorpus? Dann entledigt man sich der Formelteile, die man durch idese Eindeutigkeit nicht mehr braucht; der im vorletzten Satz gezeigte Aufruf einer Multinomialverteilung wird vereinfacht und sieht anschließend so aus: $\prod_{m \in M} \mathit{Categorical}(z_m|\theta_{m_n})$. Dem aufmerksamen Leser wird nicht entgangen sein, dass im vereinfachten Falle der Index von θ von n auf m_n gewechselt hat. Das liegt daran, dass θ nach wie vor auf der N-Plate steht und über den Index $n \in N$ zu erreichen ist. Durch die angenommene Eindeutigkeit von $m \in M$ konnte das indexierte Produktzeichen $\prod_{n \in N}$ aus dem Aufruf der oben geschilderten Multinomialverteilung $\prod_{n \in N} \prod_{m \in M_n} \mathit{Categorical}(z_{n_m}|\theta_n)$ zwar verbannt werden; die Produktschleife läuft über den Index $m \in M$. Die zum Zugriff auf die θ-Werte benötigten n-Indexwerte kriegt man einfach, indem man vom Produktschleifenindex m (innere M-Plate) auf den zu ihm korrespondierenden n-Wert schließt (d.h. man schlägt nach, in welchem Dokument eine Wortinstanz steht). So erklärt sich die Indexangabe n_m für θ.

Sonderfall: BN-Knoten bedingt durch mehrere BN-Kanten Es ist möglich, dass auf einen Knoten im Bayes'schen Netz mehrere Kanten gerichtet sind. Ein Beispiel dafür ist der BN-Knoten d_{n_m} in der Abb. 3.11, worin die Kanten aus μ_k und z_{n_m} eingehen. Um diese Gegebenheit in einer modellspezifischen Zielfunktion detailliert mathematisch zu beschreiben, kann man sie einfach auf das Modellverständnis stützend über Unterzugriffe formulieren („Wie bringe ich die Modellentitäten über Plate-Indizes unter ein Dach?").

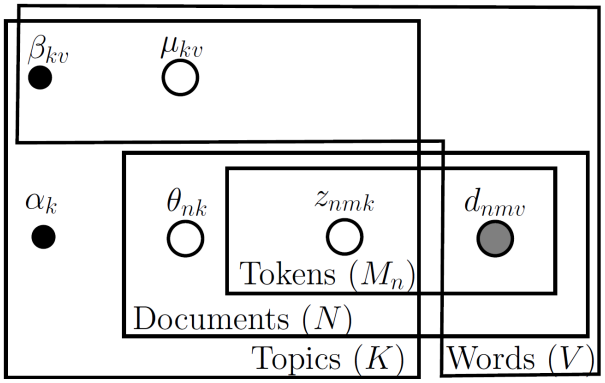


Abbildung 3.12.: Atomic Plate Model (APM) für Latent Dirichlet Allocation (LDA [BNJ03]) [RH16, S. 10], abgeleitet aus dem BN in der Abb. 3.11.

Man könnte in diesem Falle aber auch sowohl das Bayes'sche Netz als auch das Atomic Plate Model (APM; ein Zwischenergebnis der Übersetzung eines BN in ein relationales Modell nach [RH16]) zurate ziehen. Ein Beispiel für ein APM sieht man in der Abb. 3.12. Im APM werden die im BN implizit enthaltenen Datentransformationen und -Relationen explizit gemacht, und die multidimensionalen Modellvariablen in ihre Komponenten aufgesplittet. Ein APM ist kein probabilistisches Modell mehr, da es keine Zufallsverteilungen mehr beinhaltet; dennoch ist ein APM nützlich, wenn man implizite Modellstrukturen explizit machen möchte.

Man überführt einen durch mehrere Priors bedingten Posterior (wie z.B. den Posterior d im Falle von $p(d|\mu, z)$ in der Abb. 3.11) in eine detaillierte mathematische Beschreibung wie folgt. Zuerst merke man sich, aus welchen BN-Knoten sich der Prior zusammengesetzt ist (in unserem Falle seien es die BN-Knoten μ und z). Diese BN-Knoten sind auch im APM (siehe Abb. 3.12) vorhanden.

Im APM sucht man nun nach einem Pfad, sodass:

1. der gesuchte APM-Pfad beim Posterior-Knoten (d in Abb. 3.12) anfängt und ausschließlich über alle zuvor gemerkten Prior-Knoten verläuft (μ und z in Abb. 3.12)

2. man im APM von einem Pfadknoten zum nächsten nur dann laufen darf, wenn beide Knoten im APM einen gemeinsamen Indexbuchstaben besitzen

3. das letzte Element eines solchen APM-Pfades der BN-Knoten einer möglichst atomaren/nichtaggregierten Modellentität sein soll (beispielsweise Wortinstanzen wie z aus Abb. 3.12, und nicht Dokumente, Themen oder Wörter)

Man entferne nun das erste Element (den Posterior) aus dem gefundenen Pfad, der diesen Anforderungen entspricht. Dieser APM-Pfad besteht aus Prior-Knoten in der Reihenfolge ihrer Schachtelung im Prior-Teil der detaillierten mathematischen Beschreibung (z.B. $p(d|\mu_z)$). So kann man damit weiter verfahren, und anhand der bisher erläurterten Prinzipien beispielsweise den Verteilungsausdruck $\prod_{n \in N} \prod_{m \in M_n} Categorical(d_{n_m}|\mu_{z_{n_m}})$ formulieren (wie in der Formel 3.1) bzw. diesen zum Ausdruck $\prod_{m \in M} Categorical(d_m|\mu_{z_{n_m}})$ vereinfachen (bei letzterer Variante ist die globale Eindeutigkeit von $m \in M$ vorausgesetzt).

Fazit zum Formulieren einer Zielfunktion anhand von BN und APM Hoffentlich konnte dem Leser verständlich genug nahegelegt werden, wie man jeden durch eingehende Kanten bedingten Knoten des Bayes'schen Netzes in in die mathematische Form überführt. Man macht das für jeden bedingten Knoten eines BN, und anschließend verknüpft man sie via Multiplikation. Dadurch entsteht ein Produkt, welches die Struktur eines konkreten Bayes'schen Modells in einer mathematischen Notation ausdrückt. Das ist eine modellspezifische Zielfunktion. Man sollte sie logarithmieren (mit dem natürlichen Logarithmus), um daraus eine leichter zu handhabende Summenfunktion zu erhalten – die *log-likelihood*. Die log-likelihood-Funktion wird man in einem SQL-Dialekt beschreiben. Auf diese Weise betreibt man in SQL2Stan probabilistische Programmierung – d.h. man beschreibt ein statistisches Modell mit Hilfe vom Programmcode.

Bayes'sche Inferenz als Optimierung einer Wahrscheinlichkeitsfunktion

Die Beispielformel 3.1 ist die Zielfunktion von einem Themenmodellierungsansatz namens LDA, und sie nimmt in dieser Formel die Form einer Wahrscheinlichkeitsfunktion (likelihood-Funktion) ein. Im Kontext dieser Arbeit ist unter dem Begriff Zielfunktion stets eine Wahrscheinlichkeits(dichte)funktion (WDF bzw. WF) [4] gemeint, die von Zufallsvariablen abhängt. Die approximative Suche nach den Extrema solcher Zielfunktionen ist ein Teil Bayes'scher Inferenz (Suche nach den Werten für versteckte Modellparameter). Je höher der Wahrscheinlichkeitswert für konkrete Parameterwerte gegeben beobachtbare Daten, desto näher ist man an der Lösung dran. Mit anderen Worten: setzt man in die Formel der Zielfunktion anstatt Variablen konkrete Werte ein, so ergibt der Funktionswert die Glaubwürdigkeit davon, dass die eingesetzten konkreten Werte der Grundgesamtheit entstammen. Je mehr Glaubwürdigkeit, desto besser: Inferenzalgorithmen versuchen,

[4] Unterschied (unter anderem): Wahrscheinlichkeitsdichtefunktion nutzt nicht-ganzzahlige (kontinuierliche) Zufallsvariablen; Wahrscheinlichkeitsfunktion nutzt ganzzahlige (kategoriale/diskrete) Variablen.

die likelihood-Funktion zu maximieren, und geben die Variablenbelegung am gefundenen Funktionsextremum zurück. Der Lösungsalgorithmus besteht also in der automatischen Optimierung der Zielfunktion. Die automatische Optimierung von Zielfunktionen führt zur Ermittlung von den Maxima dieser Zielfunktionen – der Stellen, an denen die jeweilige likelihood-Funktion (Wahrscheinlichkeitsfunktion) der höchsten Funktionswert besitzt. Das, was einen Datenenthusiasten interessiert, ist die Inferenzausgabe – die Belegung von den nicht-beobachtbaren Modellvariablen. Diese Inferenzausgabe stammt aus der Variablenbelegung am Maximum der jeweiligen Wahrscheinlichkeitsfunktion. Daher stellen modellspezifische Zielfunktionen eine legitime Möglichkeit dar, Inferenzalgorithmen zu abstrahieren.

3.3.2. Wahrscheinlichkeitsfunktion in SQL: sprachliche Mittel

Der Nutzer von SQL2Stan kann (genauso wie bei Stan) eine Zielfunktion selbst definieren. So kann er ein Bayes'sches Modell definieren, und dementsprechend ein probabilistisches Programm schreiben. Die Spezifikation eigener statistischer Modelle (ergo probabilistische Programmierung) ist notwendig, damit der Nutzer (hier: der SQL-Programmierer, der SQL2Stan zwecks Inferenz bedient) die bereits veröffentlichten Bayes'schen Modelle eigens nachimplementieren zu können.

Sprachliche Bestandteile von Wahrscheinlichkeitsfunktionen Was benötigt die probabilistische Programmierung Bayes'scher Modelle an sprachlichen Mitteln? Ein Blick auf die bereits gesehene Formel 3.1 hilft bei dieser Fragestellung weiter. Die durch diese Formel beschriebene Wahrscheinlichkeitsfunktion ist stark vereinfacht: im Normalfall werden die p-Buchstaben, die in der likelihood-Funktion als Platzhalter für Wahrscheinlichkeitsverteilungen fungieren, durch konkrete Zufallsverteilungen (z.B. kategorische oder Dirichlet-Verteilung) ersetzt. Der Baukasten für Zielfunktionen beinhaltet daher spezielle Wahrscheinlichkeitsverteilungen – man benötigt also Funktionen, die für diese Wahrscheinlichkeitsverteilungen stehen.

Was braucht man sonst? Typische mathematische Operatoren (vor allem Addition) sind vonnöten; darüber hinaus auch Summenzeichen (inkl. Laufindex) und die Funktion für den natürlichen Logarithmus (log) sowie die Eulersche Zahl (e). Hat man alle nötigen Sprachmittel beisammen, kann man eine Zielfunktion in den Datenbankkontext hineinbringen, und zwar mit Hilfe von SQL. Mathematische Grundoperationen unterstützt SQL von Haus aus, ebenso wie Summen (sum), natürlichen Logarithmus (log) und die Eulersche Zahl (exp). Will man solch eine Zielfunktion (likelihood-Funktion) in SQL zusammenfassen, so fügt man dem SQL-Syntax nur noch die Wahrscheinlichkeitsverteilungen aus unserem Zielfunktions-Baukasten hinzu, und schon kann man mit der Bayes'schen Modellspezifikation loslegen.

Sprachliche Bestandteile von logarithmierten Wahrscheinlichkeitsfunktionen Im Falle von SQL2Stan braucht der SQL-Dialekt (abgesehen von den Verteilungsfunktionen) nur Summen und Logarithmen – ergo keine mathematischen Produkte und sehr viel weniger Multiplikationen. Die Produkte und Multiplikationen, die man

erst vorhin in der Formel 3.1 gesehen hatte, bleiben somit außen; es soll sich nicht um die likelihood-Funktion handeln, sondern um ihre logarithmierte Variante – die log-likelihood-Funktion.

Der Grund zum Logarithmieren ist der folgende: man möchte keine Produkte in den Wahrscheinlichkeitsrechnungen haben. Produkte sind in ihrer reinen Form, unabhängig von der Programmiersprache, numerisch schwer handhabbar. Wahrscheinlichkeitswerte sind reelle Zahlen zwischen 0 und 1, und wenn man sie mehrmals miteinander multipliziert, entstehen sehr kleine Zahlen, die das Zahlensystem eines Rechners zum Überlaufen bringen können. Daher setzt man gern den sogenannten Log-Sum-Exp-Trick (siehe [Eis16]) ein. Dabei logarithmiert man einfach die Wahrscheinlichkeitsfunktion, und schon werden aus Multiplikationen Additionen, und aus Produkten Summen. Die Funktionsextrema – das, wonach die Inferenzalgorithmen suchen werden – bleiben nach dem Anwenden vom Log-Sum-Exp-Trick praktischerweise unberührt. Also, um nochmal zusammenzufassen: Inferenzaufgaben lassen sich durch modellspezifische Zielfunktionen – (log-)likelihood-Funktionen – beschreiben, und diese modellspezifischen Zielfunktionen (formuliert als zu maximierende Wahrscheinlichkeitsfunktionen) lassen sich in SQL ausdrücken, insofern man SQL um Verteilungsfunktionen erweitert.

Der SQL-Code für die modellspezifische Zielfunktion stützt sich auf die Namen von Tabellen und Spalten aus dem integrierten Datenbankschema (daher auch „data model driven“), welches das ursprüngliche Datenschema um modellspezifischen Kontext erweitert. Am integrierten Schema lässt sich ablesen, was die Inferenz-Eingabe (also Daten) und was die Inferenz-Ausgabe (geschätzte Werte für versteckte Modellparameter) ist. In dem SQL-Code für die Zielfunktion steckt sehr ein wesentlicher Teil vom SQL2Stan-Projekt: das ist die Stelle, an der die die deklarative probabilistische Programmierung in einer Datenbanksprache stattfindet. Der Nutzer beschreibt zwar mit seinem SQL-Code nur, wie sich die Zielfunktion für sein Modell zusammenstellt. Das System übersetzt den nutzergeschriebenen SQL-Code in ein probabilistisches Programm, welches eine Implementierung für den Inferenzalgorithmus liefert.

3.3.3. Struktur der Zielfunktion in SQL

Zur besseren numerischen Handhabbarkeit von Wahrscheinlichkeitsrechnungen setzt SQL2Stan bei der modellspezifischen Zielfunktion bewusst auf Rechenoperationen in der *log*-Domäne, weil dort Produkte durch Summen ersetzt werden. In der Formel 3.2 findet man ein Beispiel für eine log. Wahrscheinlichkeitsfunktion:

$$\begin{aligned} \lg p(w,z,\phi,\theta|\alpha,\beta) = \lg \mathit{Dirichlet}(\theta|\alpha) \; + \; \sum_{m\in M} \lg \mathit{Categorical}(z_m|\theta) \; + \\ \sum_{k\in K} \lg \mathit{Dirichlet}(\phi_k|\beta) \; + \; \sum_{n\in N} \lg \mathit{Categorical}(w_n|\phi_{z_{m_n}}) \end{aligned} \tag{3.2}$$

Die Formel 3.2 birgt die Zielfunktion (log-likelihood) des Naïve-Bayes-Klassifikators. Eine log-likelihood-Funktion ist, wie man sieht, eine Summenfunktion, die aus mehreren Summanden bestehen kann. In SQL ist die Zielfunktion (ZF) somit ein Summenstatement, mit Summanden-Views im WITH-Teil (siehe Abb. 3.13). Im Rumpf der Summanden-Views liegen die Verteilungsfunktionsaufrufe (*Dirichlet* oder *Categorical*). Der SQL-Code einer Zielfunktion (ZF) wird nicht ausgeführt, sondern nur geparst und nach Stan übersetzt.

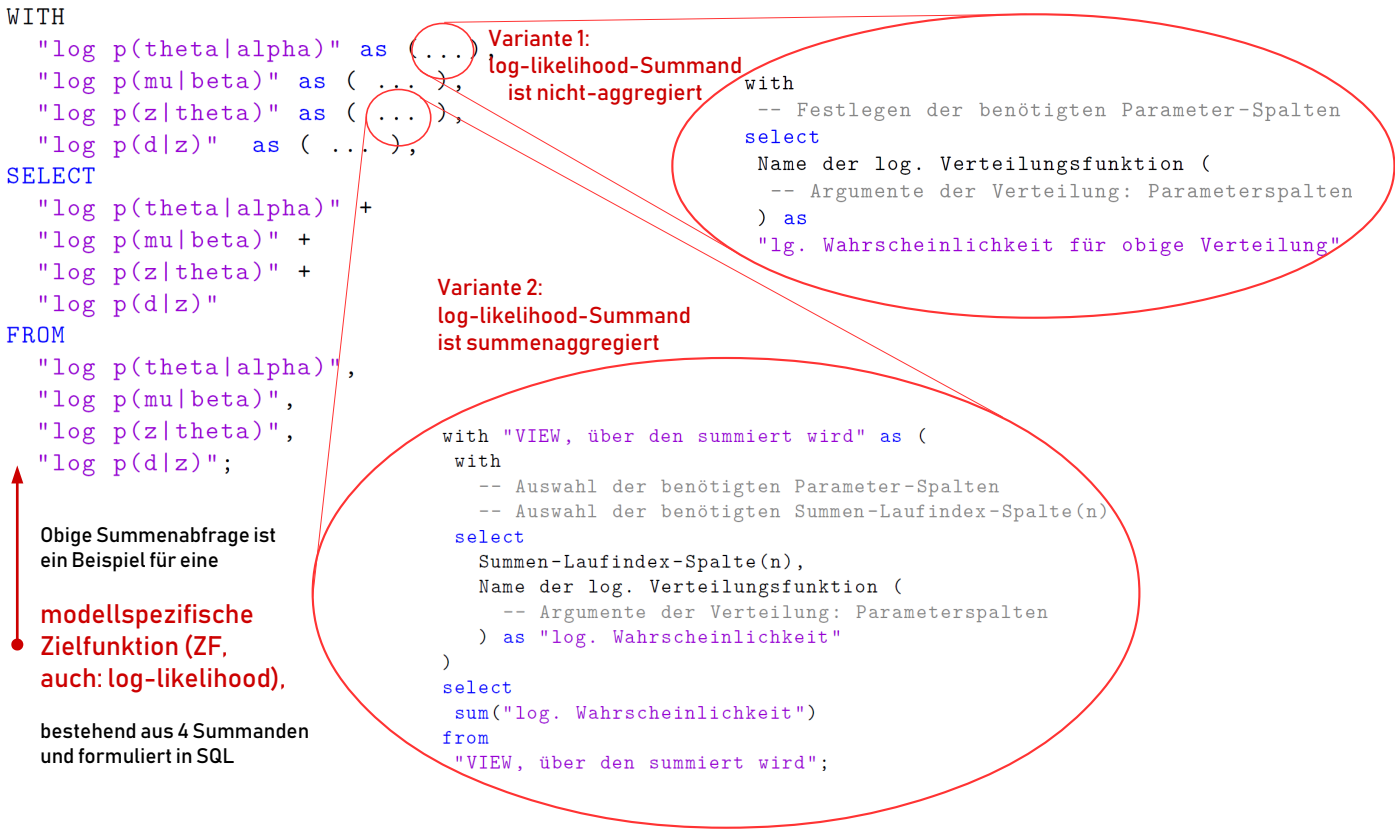


Abbildung 3.13.: Beispiel einer Zielfunktion (ZF) mit vier Summanden. Ihre SQL-Spezifikation besteht aus den Teilen WITH, SELECT und FROM. Einzelne Zielfunktionssummanden werden als Views im WITH-Teil formuliert; davon gibt es zwei Arten (summenaggregiert und nicht-aggregiert). Text- und Linienfarben sind unwichtig.

3.3.4. Struktur von Zielfunktions-Summanden

Ein Summand der Zielfunktion (ergo ein View im WITH-Teil der log-likelihood-Summenabfrage) kann zweierlei Gestalten annehmen – er kann eine oder keine summierende Aggregation beinhalten (entspricht den Varianten 2 und 1 in der Abb. 3.13). Das kann man sich wie folgt vorstellen:

- nicht-aggregierter ZF-Summand: lg *Verteilungsname* (*Parameter*)
- summenaggregierter ZF-Summand: $\sum_{\text{Laufindex}}$ lg *Verteilungsname* (*Parameter*)

Vom SQL2Stan-Prototypen unterstützte *Verteilungsnamen* sind ***dirichlet_lpdf*** (log. Wahrscheinlichkeitsdichtefunktion für Dirichlet-Verteilung) oder ***categorical_lpmf*** (log. Wahrscheinlichkeitsfunktion für Multinomialverteilung bzw. kategorische Verteilung). Diese Liste lässt sich um weitere Verteilungen aus der PPL Stan erweitern.

Zielfunktions-Summand unaggregiert (ohne Summe) in SQL

Das SQL-Gerüst eines unaggregierten (ohne Summe) ZF-Summanden der Form „lg *Verteilungsname* (*Parameter*)“ beinhaltet nur den Aufruf einer Verteilungsfunktion, siehe Listing 3.11. Die Ausgabe der Verteilungsfunktion wird in einem solchen ZF-Summanden nicht aggregiert.

Listing 3.11: Silhouette vom SQL-Code eines unaggregierten ZF-Summanden.

```
with
 -- Festlegen der benötigten Parameter-Spalten für diesen
    Zielfunktions-Summanden  als Views. Nebenbedingungen (
    Slicing, Filter usw.) im WHERE-Teil der Views möglich.
select
 Name der log. Verteilungsfunktion (
    -- Argumente der Verteilung: Parameterspalten als Arrays
          (+ ORDER-BY-Angaben zur Bewahrung der
       Arraystruktur)
 ) as "lg. Wahrscheinlichkeit für oben beschriebene
    Verteilung"
```

Als Beispiel für einen einfachen, unaggregierten Zielfunktions-Summanden eignet sich gut der mathematische Ausdruck lg $Dirichlet(\theta|\alpha)$, entnommen aus der obigen ZF-Formel 3.2 für Naïve Bayes. Ein SQL-View für diesen Dirichlet-Summanden (siehe Abb. 3.14) muss einen WITH-Teil haben, wo alle darin involvierten DB-Tabellenspalten (*theta* und *alpha*) als Views gesammelt werden. Die Festlegung eines Views für eine benötigte Datenbankspalte *theta* beschränkt sich auf eine SELECT-Abrage für 1) die Spalte, die man benötigt (*theta* aus der Tabelle *customer_class* in Abb. 3.14) sowie 2) Spalten des Primärschlüssels der Tabelle, die im FROM-Teil steht (in Abb. 3.14 ist *customer_class_id* die einzige PK-Spalte der Tabelle *customer_class*, aus der die benötigte Spalte *theta* kommt).

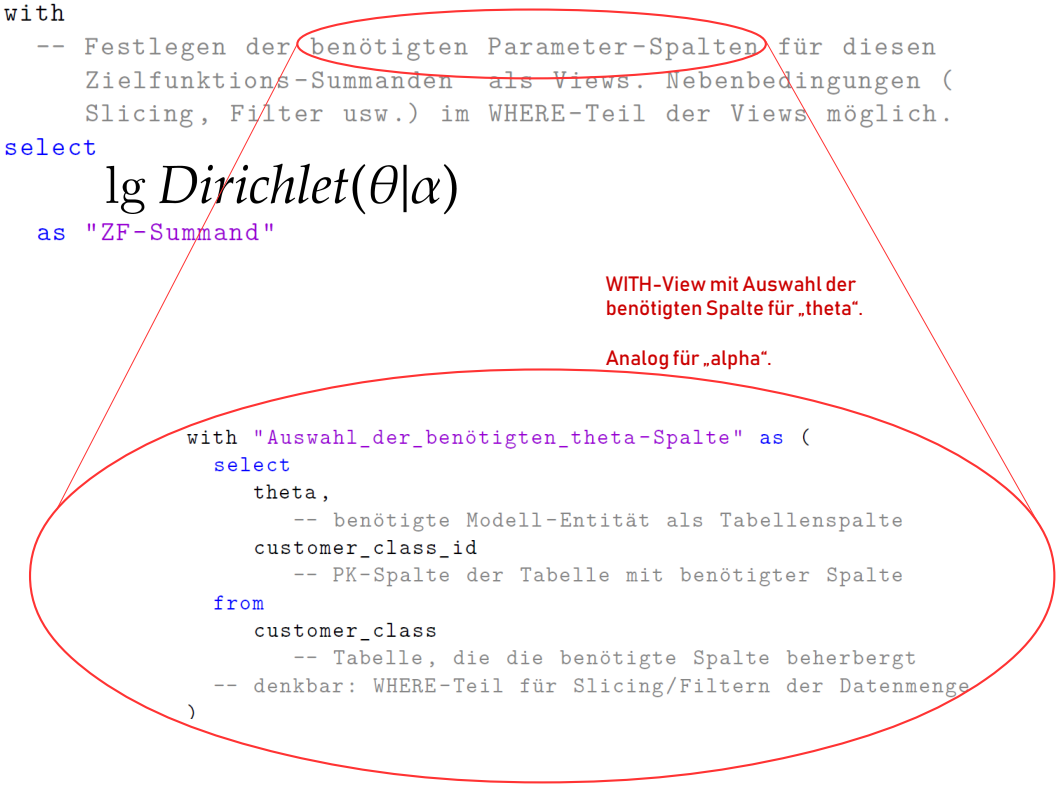


Abbildung 3.14.: Der View eines Beispiel-ZF-Summanden lg *Dirichlet*($\theta|\alpha$) beinhaltet in seinem WITH-Teil u.a. einen View für eine darin benötigte Spalte *theta*.

Wozu die WITH-Views mit der Auswahl der benötigten Spalten? Der Leser könnte sich fragen: warum muss man im Voraus extra Views für benötigte Spalten definieren; man könnte doch auch ohne Views direkt an entsprechenden Stellen „in situ“ die originalen Tabellenspalten aufrufen? Die Antwort darauf hat zwei Aspekte.

Der erste Aspekt ist die Abbildung der relationalen Abstraktion auf Arraystrukturen, die im ML-Backend (und potenziell auch im DMS-Backend) verwendet werden. Wenn der SQL-Programmierer bei jedem ZF-Summanden explizit niederschreibt, welche bei der Summandenformulierung benötigte Spalte aus welcher Tabelle kommt, kann der Compiler problemlos die Array-Einträge für die jeweiligen Spalten nachschlagen. Das ist wichtig bei der automatischen Übersetzung vom SQL-Code in eine PPL. Später werde ich nochmal darauf zurückkommen und zeigen, dass man die Festlegung der benötigten Spalten als Views nicht weiter vereinfachen oder ersetzen kann (zumindest nicht im SQL2Stan-Ansatz). Es hängt, um ein wenig vorauszugreifen, auch damit zusammen, dass man keine expliziten (Inner-)Joins von Tabellen haben möchte, wo die eindeutige Zuordnung von Spalten zu Tabellen verloren gehen kann.

Der zweite und letzte Aspekt der Antwort auf die oben geschilderte Frage des Lesers ist die Datenvorverarbeitung. Im WHERE-Teil von den im Voraus definierten Spaltenauswahl-Views könnte man flexibel und übersichtlich unterschiedliche Filter-bzw. Slicing-Angaben machen. Zum Beispiel, man möchte nicht alle Onlineshopkunden- beziehungsweise Shopbesucherdaten zum Training eines Klassifikators benutzen, sondern nur die höchstens eine Woche alten Daten. Mit einem Klassifikator ist ein geeignetes Bayes'schen Modell gemeint, welches nach der automatischen Inferenz zur Klassifikation dienen kann. Zu einem solchen Anwendungsfall der Bayes'schen Inferenz bietet diese Masterarbeit ein ausführlich beschriebenes Fallbeispiel.

Die Notwendigkeit, beim Formulieren eines Zielfunktions-Summanden die benötigten Spalten via Views extra auszuwählen, erlaubt es dem SQL-Programmierer außerdem, den SQL-Code eines ZF-Summanden übersichtlich zu gestalten. Zum Einen werden die eventuelle Filter- und Datenvorverarbeitungsdetails in die WITH-Views überführt, damit sie den SELECT-Teil der ZF-Summanden-Beschreibung nicht überladen. Zum Anderen stellt der Programmierer die benötigten Spalten im WITH-Teil zusammen und überdenkt gleich im Voraus, wie die betroffenen Modellentitäten hierarchisch zueinander passen und wie die Unterabfragen für diese Entitäten zu schachteln sein werden. Den WITH-Views können beliebige ausdrucksstarke Aliases verliehen werden, was die Übersichtlichkeit und Verständlichkeit vom SQL-Code zusätzlich verbessert.

Beispiel: unaggregierter ZF-Summand in SQL Nach dem Kennenlernen von diesem grundlegenden SQL-Programmiergerüst eines Zielfunktion-Summanden lohnt sich ein Blick auf syntaktisch echtes Beispiel. Aus der mathematischen Beschreibung des Zielfunktions-Summanden lg $Dirichlet(\theta|\alpha)$ entsteht SQL-Code, in dem im VIEW-Teil zuerst die benötigten Tabellenspalten (und dazu die dazugehörigen Zugriffsspalten) ausgewählt werden, die im SELECT-Teil als Parameter eines Verteilungsfunktion-Aufrufs genutzt werden.

In der Abbildung 3.14 wurde bereits skizziert, wie man im SQL2Stan-Dialekt den ZF-Summanden lg $Dirichlet(\theta|\alpha)$ beschreiben kann. Im weiter aufgeführten Listing 3.12 folgt die Formulierung von diesem Zielfunktionssummanden in SQL, wo die vorher abgebildete Skizze eine konkrete Gestalt in der SQL2Stan-Syntax annimmt.

Listing 3.12: SQL-Code eines unaggregierten ZF-Summanden lg $Dirichlet(\theta|\alpha)$

```
with
-- Auswahl der benötigten Parameter-Spalten für diesen
   Zielfunktions-Summanden
"posterior__theta" as (
  select
     theta,
     customer_class_id
  from
     customer_class
),
"prior__alpha" as (
  select
     alpha,
     customer_class_id
  from
     customer_class
)
select
  -- Aufruf der Verteilungsfunktion
     log Dirichlet(theta|alpha)
  dirichlet_lpdf(
     ARRAY(
         select
            theta
         from
            posterior__theta
         order by
            customer_class_id
     ),
     ARRAY(
         select
            alpha
         from
            prior__alpha
         order by
            customer_class_id
     )
  ) as "Log. Wahrscheinlichkeit für theta gegeben alpha.
     Diese SELECT-Anweisung dient nur der
        Modellspezifikation.
     Dieser SQL-Code wird nicht ausgeführt, nur compiliert.
     'lpdf' bedeutet 'log probability density function'"
```

Zielfunktions-Summand aggregiert (mit Summe) in SQL

Die einfachen, unaggregierten ZF-Summanden (lg *Verteilungsname* (*Parameter*)) sowie ihre SQL-Beschreibung dürften als abgehandelt gelten. Die anderen, summenaggregierten ZF-Summanden ($\sum_{\text{Laufindex}}$ lg *Verteilungsname* (*Parameter*)) bestehen logischerweise aus einer Abfrage einer Menge von Werten (hier: Ausgaben der Verteilungsfunktions-Aufrufe) sowie 2) aus einer Summenaggregationsfunktion über diese Werte. In SQL besteht ein solches Konstrukt also aus zwei Schichten – der äußeren Summenabfrage und der inneren Abfrage (mit einer oder mehreren Laufvariablen sowie einem Verteilungsfunktionsaufruf). Die innere Abfrage ruft die Verteilungsfunktion mehrmals auf [5]. Die äußere Summenabfrage aggregiert die Ergebnisse der mehrmaligen Verteilungsfunktionsaufrufe zu einem einzigen Wert. Das Listing 3.13 stellt ein SQL-Gerüst für summenaggregierte ZF-Summanden dar.

Listing 3.13: Silhouette vom SQL-Code eines summenaggregierten ZF-Summanden.

```
with "VIEW, über den summiert wird" as (
 with
   -- Auswahl der benötigten Parameter-Spalten für diesen
      Zielfunktions-Summanden. Nebenbedingungen (Slicing,
      Filter usw.) im WHERE-Teil der Views möglich.
   -- Auswahl der benötigten Summen-Laufindex-Spalte(n);
      ihre View-Namen müssen den Alias 'LoopIndex'
      beinhalten!
 select
   Spalte(n) als Summen-Laufindizes,
     -- So oft wird untere Verteilungsfunktion aufgerufen.
   Name der log. Verteilungsfunktion (
     -- Argumente der Verteilung: Parameterspalten als
        Arrays   (+ ORDER-BY-Angaben zur Bewahrung der
        Arraystruktur)
     -- Argumente der Verteilung dürfen über WHERE-
        Bedingungen auf die Werte von Summen-Laufindizes
        zugreifen
   ) as "log. Wahrscheinlichkeit"
)
select
 sum("log. Wahrscheinlichkeit")
from
 "VIEW, über den summiert wird";
```

Die Aggregation der Verteilungsaufruf-Ergebnisse (lg *Verteilungsname* (*Parameter*) im inneren Select-Statement) erfolgt über eine Summe im äußeren Select-Statement

[5] Die Verteilungsfunktion wird in der inneren Abfrage eines aggregierten ZF-Summanden so oft aufgerufen, wie die Laufvariable(n) neben dem Verteilungsfunktionsaufruf im SELECT-Teil dieser inneren Abfrage es vorgeben

($\sum_{\text{Laufindex}}$). Wenn man dieses SQL-Konstrukt interpretiert, ist es eine Art for-Schleife(n) über Laufindizes, wo eine (Verteilungs-)Funktion iterativ aufgerufen wird und ihre Ausgabe in jeder Iteration zu einer Variable addiert wird. Natürlich müssen die Funktionsaufrufe im Rumpf der Schleife auf den aktuellen Wert der Laufindizes zugreifen dürfen. Wie das im SQL2Stan-Dialekt gewährleistet wird, erkläre ich direkt in dem auf diesen Satz folgenden Paragraphen.

Unterzugriffe im Verteilungsaufruf eines Summanden (Abb. 3.15) Bisher hat man anhand der Beispiele gesehen, wie einfache Verteilungsfunktionen in SQL eingebettet werden können. lg $\mathit{Dirichlet}(\theta|\alpha)$ nahm z.B. ganze Spalten *theta* und *alpha* als Argumente entgegen. Doch wenn es in Richtung SQL-definierter Schleifen mit Laufindizes gehen soll, muss es eine Möglichkeit geben, wie man nur bestimmte Spaltenwerte anhand eines Laufindex (oder anhand der bereits ausgewählten Werte) auswählt. Das wird durch Unterzugriffe bewerkstelligt. Hier sieht man beispielsweise einen Summanden mit einer Menge von Unterzugriffen: $\sum_{n \in N} \lg \mathit{Categorical}(w_n|\phi_{z_{m_n}})$. Wie drückt man das im SQL2Stan-Dialekt aus?

Die SQL-Ausdrucksweise für solche Unterzugriffe in einem Argument eines Zielfunktions-Summanden ist in der Abb. 3.15 geschildert und lautet wie folgt:

- Ist der aktuelle Unterzugriff über den Laufindex (oder einen konkreten Wert) zu erfolgen, so nehme man dafür eines der folgenden sprachlichen Konstrukte:
  - WHERE id = **LoopIndex**.id
  - WHERE id = konkreter Wert

  Die Unterzugriffe mit konkreten Werten wurden im SQL2Stan-Prototypen nicht implementiert.

- Sind einer oder mehrere weitere Unterzugriffe in den aktuellen Unterzugriff verschachtelt, so so ist das in SQL wie folgt zu beschreiben:
  - WHERE id **IN** (SELECT-Abfrage eines weiteren Unterzugriffs)

  Es ist ein impliziter Join, direkt an der Stelle, wo er gebraucht wird. Warum man in SQL2Stan keine expliziten Joins verwendet (d.h. die Nutzung vom klassischen SQL-Keyword JOIN wird gemieden), wird später am Ende des Kapitels 3.3.5 („Übersetzung der Zielfunktion nach Stan“) erklärt.

Das Listing 3.14 liefert ein Beispiel zur Schilderung von Unterzugriffen in einem Verteilungsfunktions-Argument im Rumpf eines ZF-Summanden.

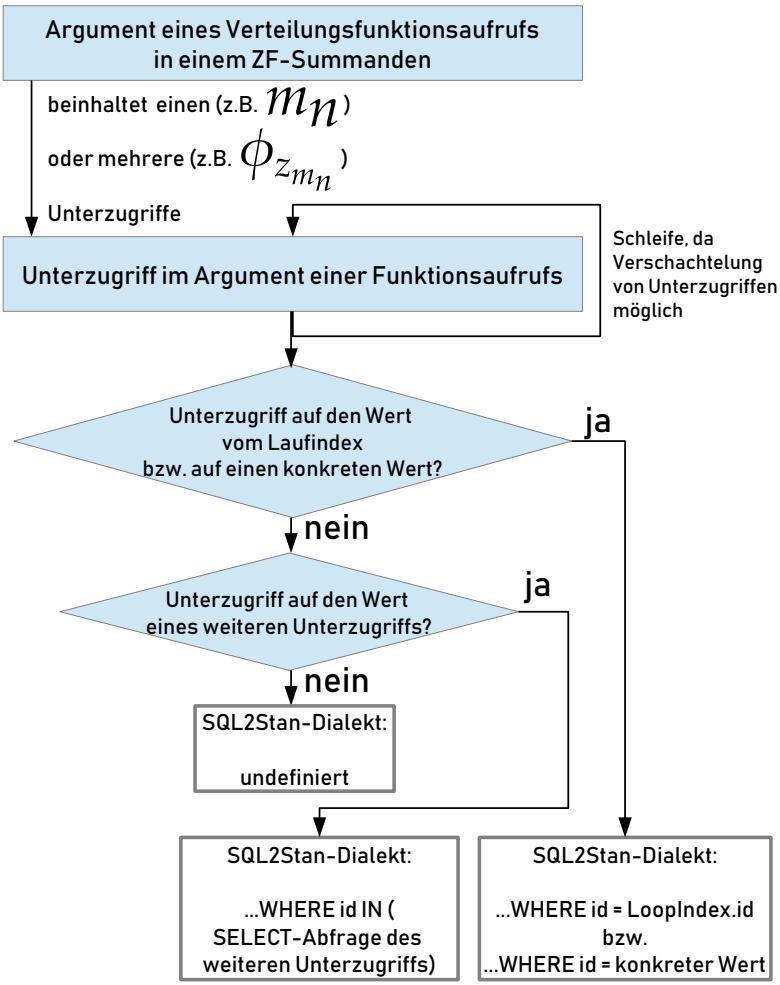


Abbildung 3.15.: Vorgaben des SQL2Stan-Dialekts im Bezug auf Unterzugriffe in einem Argument eines Zielfunktions-Summanden.

Listing 3.14: Unterzugriffe im Verteilungsfunktions-Argument $\phi_{z_{m_n}}$, beschrieben in SQL als Unterabfragen. Der ZF-Summand, der die Aufrufe der ungenannten Verteilungsfunktion birgt, sei summenaggregiert. Der Laufindex der Summe sei *product_purchase_id*

```
1 ARRAY(
2 select
3    phi
4 from
5    prior__phi_z
6 where
7    customer_class_id in (
```

```
8         select
9            customer_class_id
10        from
11           prior__z_doc
12        where
13           customer_id in (
14              select
15                 customer_id
16              from
17                 prior__doc_n
18              where
19                 product_purchase_id = LoopIndex.
                      product_purchase_id
20              order by
21                 product_purchase_id
22           )
23        order by
24           customer_id
25        )
26 order by
27    customer_class_id, product_id
28 )
```

Die Zeilen 1 und 28 im Listing 3.14 offenbaren, dass die Parameter einer Verteilungsfunktion durch den Ausdruck „ARRAY(...)“ explizit als Arrays deklariert werden; das hat mit UDF zu tun. Benutzerdefinierte Funktionen (User-Defined-Functions, kurz UDF) sind in SQL2Stan einfache Dummy-UDF ohne Funktion, die als Bezeichner für Verteilungsfunktionen fungieren. Da der SQL-Code nur als Grundlage für weitere Codeübersetzung dient, reichen auch Dummy-Funktionen. Die Verteilungs-UDF in SQL2Stan sind aktuell an die UDF-Einschänkungen von PostgreSQL (DBMS-Backend im SQL2Stan-Prototyp) gebunden. PostgreSQL-UDF dürfen keine Unterabfragen als Argumente entgegennehmen, dafür aber sehr wohl Arrays. Daher werden die Unterabfragen der UDF-Argumenten zwischen die Klammern vom Ausdruck „ARRAY(...)“ gepackt. Auf diese Weise kommt der SQL2Stan-Dialekt dem PostgreSQL-Parser entgegen. Diese Arrays sind fiktiv, da der SQL-Code nicht ausgeführt, sondern nur geparsed werden soll. Man soll diese ARRAY-Ausdrücke im SQL-Code der Funktionsargumente nicht mit den echten Arrays im Stan-Code verwechseln, die durch SQL-nach-Stan–Übersetzung des relationalen Schemas entstehen. Man sieht im Listing 3.14 auch ORDER-BY-Anweisungen (Zeilen 20-21, 23-24 und 26-27). Diese Anweisungen beinhalten tabellenspezifische Schlüsselspalten, geordnet nach ebenfalls der tabellenspezifischen „Sorting-Order“-Reihenfolge. Die ORDER-BY-Anweisungen sind zur expliziten Beibehaltung der Struktur von den Stan-Arrays gedacht. Im SQL2Stan-Dialekt soll es zu jedem Zeitpunkt klar sein, wie der SQL-Code auf Stan-Arrays abzubilden ist.

3.3.5. Übersetzung der Zielfunktion nach Stan

Kontextuelle Zuordnung von diesem Kapitel Der SQL2Stan-Workflow begann mit einem integrierten relationalen Schema, welches fast vollständig aus der grafischen Darstellung eines Bayes'schen Modells übersetzt wurde. Dieses relationale Schema beinhaltet alle Entitäten, die in das Bayes'sche Modell involviert sind. Das heißt, man kann die Namen der Entitäten aus diesem relationalen Schema zur expliziten Modellspezifikation nutzen. Es wurde im Vorfeld erklärt, wie die Entitäten des relationalen Schemas auf Arrays abgebildet werden können (jede Tabellenspalte entspricht einem Array). Es wurde damit begründet, dass man dadurch zum Einen Array-DBMS als DMS-Backend verwenden kann (z.B. SciDB), zum anderen kann man auf diese Weise ML- und Inferenzsoftware wie Stan an die relational beschriebenen Entitäten anbinden – die infrage kommenden ML-Backends arbeiten nämlich mit Arrays (oder Tensoren; der Unterschied ist nicht groß).

Ein SQL-Programmierer arbeitet bei SQL2Stan mit Relationen in einem SQL-Interface. Über diesen Datenbankkontext werden ihm Bayes'sche Modellierung und Bayes'sche Inferenz zugänglich gemacht. Er kann im SQL2Stan-Dialekt ein konkretes Bayes'sches Modell zwecks automatischer Inferenz spezifizieren. Dafür muss er durch sein Modellverständnis von der grafischen Darstellung des Bayes'schen Modells zu einer mathematischen Darstellung diesen Modells (Zielfunktion bzw. log-likelihood-Funktion) kommen. Ist das geschafft, so kann der SQL-Programmierer diese mathematische Darstellung – die einfach eine Summe ist – im SQL2Stan-Dialekt beschreiben. Bei der Spezifikation eines statistischen Modells in Form eines SQL-Summenstatements (Zielfunktion) sollen die Modellentitäten-Namen aus dem integrierten relationalen Schema genutzt werden.

Die Art, wie die Summanden der Zielfunktion in SQL formuliert werden, ist prinzipiell wie folgt zu beschreiben. Im WITH-Teil des SQL-Statements eines Summanden werden die im Summanden benötigten Tabellenspalten (inklusive der Schlüssel der Tabellen, aus denen die benötigten Spalten stammen) hinterlegt, damit man bei der Stan-Übersetzung auf die zu diesen Spalten korrespondierenden Arrays schließen kann. Anschließend formuliert man in SQL eine SELECT-Abfrage, in der eine bestimmte Verteilungsfunktion aufgerufen wird. Die Argumente von diesem Verteilungsfunktionsaufruf werden aus den im WITH-Teil stehenden Spalten zusammengesetzt. Jeder ZF-Summand beinhaltet also einen parametrisierten Aufruf eines Verteilungsfunktion, dessen Parameter bestimmte Modellentitäten sind. Bis zu diesem Teil wurde das SQL2Stan-Konzept bisher erklärt.

In diesem Kapitel wird erklärt, wie die Zielfunktion (ein ZF-Summand nach dem anderen) nach Stan übersetzt werden kann. Stan ist die Sprache des ML-Backends, welches sich um die Bayes'sche Inferenz kümmern soll. Die Aufrufe der Verteilungsfunktionen – ein solcher Aufruf pro ZF-Summand – werden in die probabilistische Programmiersprache Stan übersetzt. Das Ergebnis der Übersetzung wird im **model**-Block des generierten Stan-Programms platziert. Ein dadurch generiertes Stan-Programm implementiert die Bayes'sche Inferenz an dem statistischen Modell, welches in SQL definiert wurde.

Da die Übersetzung der SQL-formulierten Zielfunktion nach Stan darauf hinausläuft, die ZF-Summanden (und somit die darin enthaltenen Aufrufe der probabilistischen Verteilungsfunktionen) einen nach dem anderen automatisch auszuwerten, ist eine Unterscheidung wichtig. Es kommt bei der Übersetzung nähmlich darauf an, ob Verteilungsfunktionsaufruf direkt im Anschluss aggregiert wird oder nicht.

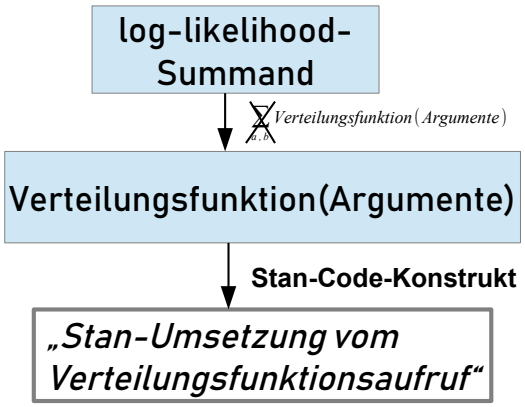


Abbildung 3.16.: Übersetzung eines ZF-Summanden, dessen Verteilungsfunktionsaufrufe nicht durch eine Summe aggregiert werden.

Verteilungsfunktionsaufrufe in Zielfunktionssummanden: summenaggregiert oder nicht? Zuallererst wird bei der Übersetzung geschaut, ob um den Summanden herum eine summenaggregierende SELECT-Abfrage steht. Ist das der Fall (wie in Abb. 3.17), dann kann man direkt davon ausgehen, dass die Verteilungsfunktion im Rumpf des Summanden mehrmals aufgerufen wird, und deren Ausgaben mehrmals zum Zielfunktionswert (**target**-Variable im Stan-Code) beitragen. In diesem Falle entsteht aus dem SQL-Code des Summanden ein for-Schleifen-Konstrukt im generierten/übersetzten Stan-Code. Anderenfalls, falls die Ausgabe des Verteilungsfunktionsaufrufs nicht summierend aggregiert wird (wie in Abb. 3.16), kann man sich direkt um die Übersetzung diesen Funktionsaufrufs kümmern. Abbildungen 3.16 und 3.17 bieten eine Übersicht zu dieser Unterscheidung.

„Stan-Umsetzung vom Verteilungsfunktionsaufruf“ – so lautet die Beschriftung der Platzhalter in den Abbildungen 3.16 und 3.17. Wie genau der Aufruf der Verteilungsfunktion zusammen mit seinen Argumenten nach Stan übersetzt wird, darf der Leser gleich erfahren. Dabei ist bloß wichtig zu unterscheiden, ob die Argumente des zu übersetzenden SQL-Funktionsaufrufs eine zu einem *diskreten Modellparameter* korrespondierende Spalte beinhalten – oder ob die in den Argumenten genutzten Spalten frei von diskreten Modellparametern sind.

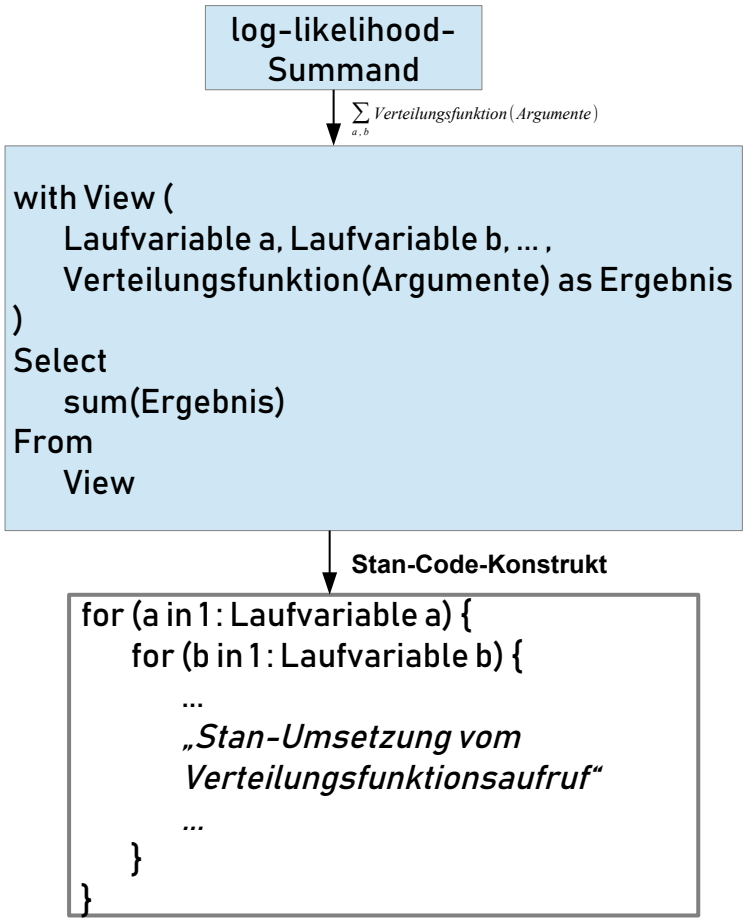


Abbildung 3.17.: Übersetzung eines ZF-Summanden, der eine Summenaggregation enthält.

Übersetzung von Summanden OHNE diskrete Modellparameter (Abb. 3.18)

Die nicht-beobachteten Referenzen zwischen den Modellentitäten werden in der Regel durch diskrete (ganzzahlige, integer-getypte) Modellparameter modelliert. Im integrierten Schema haben die Tabellenspalten für diskrete Parameterwerte den Typ „stan_parameter_int“. Die Werte für solche ganzzahligen Modellvariablen werden nicht geschätzt (Stan kann es nicht), daher gehören sie weder zur Einnoch zur Ausgabe der Inferenz. Diskrete Modellparameter dienen vielmehr dazu, im probabilistischen SQL2Stan-Programm die Referenzen zwischen den Tabellen (Fremdschlüssel) als unbekannt zu markieren. Die Übersetzung von Summanden ohne diskrete Parameter in Stan-Code verläuft unkompliziert (siehe Abb. 3.18).

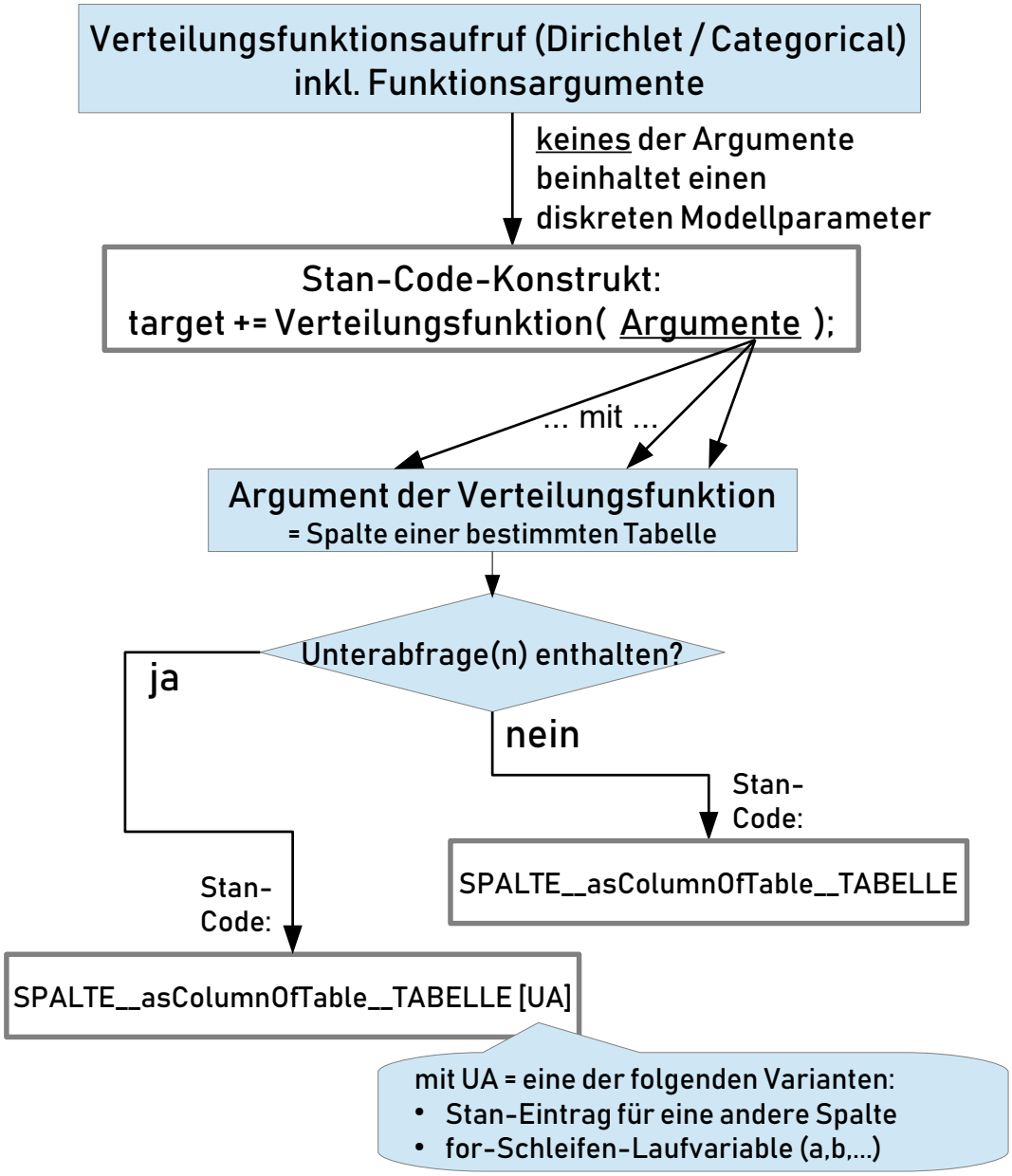


Abbildung 3.18.: Übersetzung eines Summanden, der in seinen Argumenten keine Spalten enthält, die zu einem diskreten Modellparameter korrespondieren.

Die Übersetzung vom SQL-Code eines ZF-Summanden ohne diskrete Modellparameter nach Stan ist in der Abb. 3.18 geschildert. Zur Verdeutlichung: der SQL-Code aus dem Listing 3.15 kann in den Sstan-Code aus dem Listing 3.16 übersetzt werden.

Listing 3.15: Generischer SQL-Code eines ZF-Summanden, der keine diskreten Modellparameter enthält. Die SQL2Stan-Syntax wird nur nachgeahmt.

```
with "benötigte_Spalte" as (
   select
      spalte_Y, -- kein diskreter Modellparameter
      primaerschluessel_der_tabelle_X
   from
      tabelle_X
)
select
   statistische_verteilung(
      ARRAY(
         select
            spalte_Y
         from
            benötigte_Spalte
      )
   )
```

Listing 3.16: Generischer Stan-Code eines ZF-Summanden, der keine diskreten Modellparameter enthält und der aus dem SQL-Code des Listings 3.15 übersetzt worden sei. Die Stan-Syntax wird nur nachgeahmt.

```
target += statistische_verteilung(spalte_Y__asColumnOfTable__tabelle_X);
```

Übersetzung von Summanden MIT einem diskreten Modellparameter (Abb. 3.19 und 3.20)

Erscheint in einem der Verteilungsfunktionsargumente ein diskreter Modellparameter, so nimmt die Komplexität der Inferenz-Implementierungsdetails zu:

- Ein *gamma*-Hilfsarray muss zu Beginn des **model**-Blocks im Stan-Programm definiert werden (siehe Abb. 3.19).
- Eine neue for-Schleife entsteht unmittelbar vor dem Verteilungsfunktions-Aufruf, dessen Argumente einen diskreten Modellparameter beinhalten (Abb. 3.19).
- Jeder Verteilungsfunktions-Aufruf, der einen diskreten Modellparameter einbezieht, schreibt seine Ergebnisse in eine Zelle vom *gamma*-Hilfsarray. Ist die Zelle noch leer, so wird sie mit dem ersten dort eingetragenen Ergebnis initialisiert. Jedes weitere Ergebnis von Verteilungsfunktions-Aufrufen, das in eine nichtleere Zelle vom *gamma*-Array reingeschrieben werden soll, aktualisiert additiv das in der Arrayzelle stehende Ergebnis, ohne es zu überschreiben. Übersicht in Abb. 3.19.

- Eine Schleife über *gamma* am Ende des **model**-Blocks mit Einsatz des Log-Sum-Exp-Tricks. Diese Schleife erhöht den Wert der **target**-Variable um die logarithmierten Wahrscheinlichkeiten von denjenigen Verteilungsfunktions-Aufrufen, die als eines ihrer Argumente einen diskreten Modellparameter genutzt hatten (siehe Abb. 3.20).

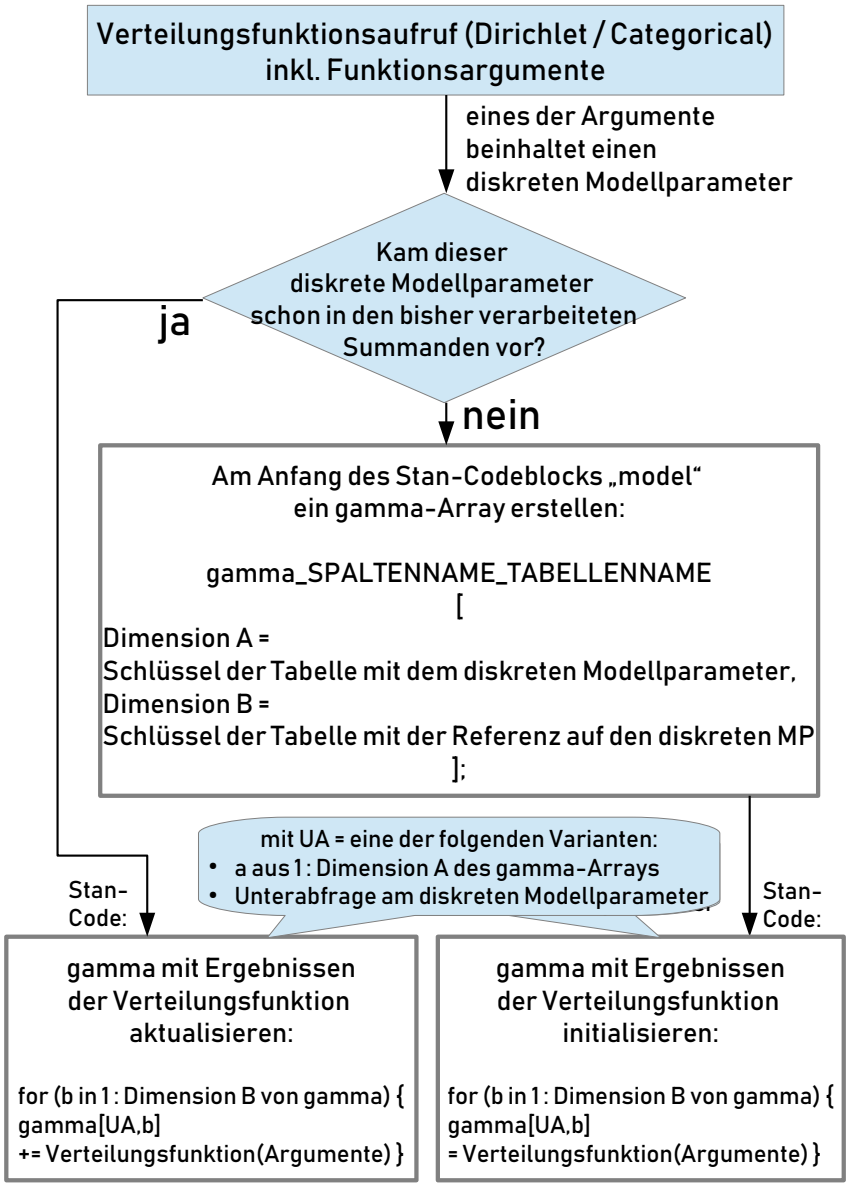


Abbildung 3.19.: Übersetzung vom Verteilungsfunktionsaufruf eines ZF-Summanden: eines der Funktionsargumente enthält eine zu einem diskreten Modellparameter korrespondierende Spalte.

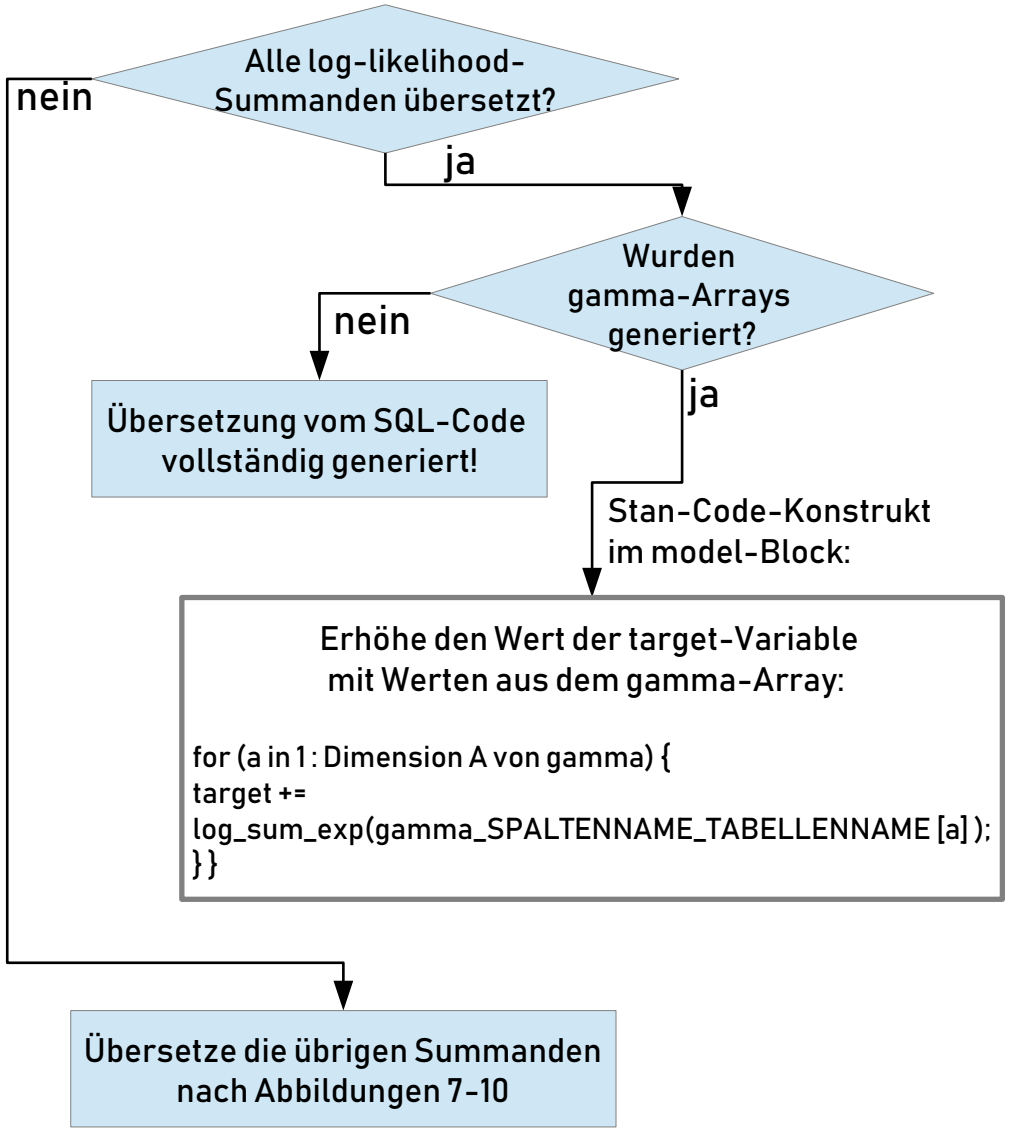


Abbildung 3.20.: Addieren von den Werten aus dem gamma-Array zur log-likelihood-Funktion am Ende des model-Blocks im Stan-Code. Kontext: Übersetzung vom Verteilungsfunktionsaufruf eines ZF-Summanden, der sich auf einen diskreten Modellparameter beruft.

Diskrete latenter Modellparameter in SQL und Stan Ein diskreter Modellparameter ist generell eine Ganzzahl-Spalte SPALTENNAME_DISKRET in der Tabelle TABELLENNAME. Logisch, dass TABELLENNAME.SPALTENNAME_DISKRET eine Fremdschlüssel-Spalte ist, weil ihre Werte IDs einer anderen Tabelle REFTAB referenzieren – doch diese Referenzen sind nicht beobachtbar; man hat für diese Fremdschlüsselspalte keine konkreten Werte. SPALTENNAME_DISKRET kann z.B. eine Spalte mit Kundenklassen in der Tabelle KUNDEN sein, und enthält unbeobachtbare Verweise auf die IDs der Tabelle KUNDENKLASSEN verweist.

Die Dimensionalität vom *gamma*-Hilfsarray für TABELLENNAME.SPALTENNAME setzt sich zusammen aus:

1. der ID der Tabelle TABELLENNAME (Primary-Key-Tabellenspalte TABELLENNAME.TABELLENNAME_ID)

2. der Anzahl der IDs in der Schlüsselspalte REFTAB_ID der Tabelle REFTAB, worauf TABELLENNAME.SPALTENNAME_DISKRET verweist

Solch ein *gamma*-Hilfsarray (siehe Listing 3.17) wird einmal pro diskreten Modellparameter (ergo einmal pro latenten Tabellenspalten-Querverweis) angelegt. In der PPL Stan kommt man um *gamma*-Hilfsarrays und entsprechende darauf aufbauende Code-Konstrukte nicht herum, wenn man mit diskreten Modellparametern arbeitet. Das liegt ganz einfach daran, dass Stan direkt keine diskreten Modellparameter schätzen kann [Sta19d]. Man kann mit Stan nur kontinuierliche (nichtganzzahlige) Modellparameter schätzen lassen, und die diskreten Modellparameter bei der Modellspezifikation über solche Einfälle wie *gamma*-Arrays marginalisieren.

Listing 3.17: Generisches Stan-Code-Gerüst zur Definition eines gamma-Hilfsarrays im model-Block des generierten Stan-Programms. Ein solcher gamma-Array muss bei Stan im Zusammenhang mit diskreten Modellparametern eingesetzt werden, da Stan sie nicht schätzen/samplen kann.

```
real
gamma__for__REFTAB_ID__asColumnOfTable__REFTAB
[
 Total__ofTable__TABELLENNAME__asNumberOfDistinct__TABELLENNAME_ID,
 Total__ofTable__REFTAB__asNumberOfDistinct__REFTAB_ID,
];
```

Um etwas vorzugreifen: der erzwungene Einsatz von komplexen Hilfskonstrukten (wie z.B. ebendiese *gamma*-Hilfsarrays) in der probabilististischen Programmierung ist eine der Gründe, warum man die probabilististische Programmierung deklarativer und einfacher gestalten soll. SQL2Stan nimmt dem SQL-Programmierer die Umsetzung von den Stan-PPL-spezifischen *gamma*-Hilfskonstrukten automatisch ab; somit abstrahiert und automatisiert SQL2Stan einen Teil der Inferenzimplementierung ab, der für einen Datenenthusiasten schwer durchzublicken wäre. Insofern ist das SQL2Stan-Interface etwas bedienungsfreundlicher als Stan.

1. Übersetzungsbeispiel mit einem diskreten Modellparameter: Initialisierung der gamma-Werte Die Stan-Code-Generierung unter Einbeziehung diskreter Modellparameter in die ZF lässt sich gut an einem Beispiel erklären. Angenommen, der Summand $\sum_{m \in M} \lg \mathit{Categorical}(z_m|\theta)$ (SQL-Code im Listing 3.18) sei **der erste Summand, der sich auf einen diskreten Parameter** z **bezieht**. D.h., im übersetzten Stan-Code muss ein *gamma*-Hilfsarray am Anfang des **model**-Blocks erzeugt werden.

Listing 3.18: SQL-Code des ZF-Summanden $\sum_{m \in M} \lg \mathit{Categorical}(z_m|\theta)$. Bei der Auswertung der entsprechenden ZF sei dies der erste Summand, der sich auf den latenten diskreten Modellparameter z (leere Spalte Customer_class_id in der Tabelle Customer) beruft.

```
with "log cat(z_m|theta)" as (
  with "posterior__z_m" as (
     select
            customer_class_id, -- latent, ganzzahlig/diskret
            customer_id
     from
            customer
  ),
  "prior__theta" as (
     select
            theta,
            customer_class_id
     from
            customer_class
  )
  select
     LoopIndex.customer_id,
     categorical_lpmf(
            ARRAY(
                   select
                          customer_class_id
                   from
                          posterior__z_m
                   where
                          customer_id = LoopIndex.customer_id
                   order by
                          customer_id
            ),
            ARRAY(
                   select
                          theta
                   from
```

```
33                          prior__theta
34                      order by
35                          customer_class_id
36                )
37        ) as categorical_lpmf
38    from
39        posterior__z_m as LoopIndex
40  )
41  select
42        sum(categorical_lpmf) as "log p(z|theta)"
43  from
44        "log cat(z_m|theta)"
```

Im ersten Argument des Verteilungsfunktions-Aufrufs (Zeilen 19-28 im Listing 3.18) wird die Spalte Customer_class_id der Tabelle Customer addressiert. Diese Spalte habe den Typ *stan_parameter_int*, d.h. es sei ein latenter diskreter Modellparameter. Für diesen Modellparameter wird bei der Übersetzung nach Stan ein *gamma*-Array erzeugt. Im generierten Stan-Code enthält der Name von diesem *gamma*-Array keinen Namen der Tabelle mit dem diskreten Modellparameter. Vielmehr wird im Namen des *gamma*-Arrays beschrieben, welche Modellentität durch diesen diskreten Modellparameter adressiert wird (d.h. welche Spalte welcher Tabelle dadurch latent referenziert wird), siehe Listing 3.19 (beispielbezogen).

Listing 3.19: Struktur des gamma-Array-Eintrags im Stan-Code.

```
gamma__for__customer_class_id__asColumnOfTable__customer_class[DIMS]
```

Die Dimensionalität vom *gamma*-Array (Platzhalter DIMS im Listing 3.19) wird dabei vorgegeben durch 1) Customer.Customer_id als Schlüssel der Tabelle, in der der diskrete Modellparameter ruht sowie 2) Customer_class.Customer_class_id als Schlüssel der Tabelle, den dieser diskrete Parameter referenziert. Der Code aus dem Listing 3.18 wird in den Stan-Code im Listing 3.20 übersetzt.

Listing 3.20: Stan-Code, der durch automatische Übersetzung des Codes aus dem Listing 3.18 entsteht. Unter anderem sieht man deutlich die Initialisierung vom gamma-Hilfsarray (Zeilen 14-15) sowie die Aktualisierung des Gesamtwertes der Zielfunktion (target) in Zeilen 21–23.

```
1  model {
2
3  real gamma__for_
4        _customer_class_id__asColumnOfTable__customer_class[
5              Total__ofTable__customer_
6                    _asNumberOfDistinct__customer_id,
7              Total__ofTable__customer_class_
8                    _asNumberOfDistinct__customer_class_id];
9
10 for (a in 1:Total__ofTable__customer_
11       _asNumberOfDistinct__customer_id) {
```

```
12 for (b in 1:Total__ofTable__customer_class
13       __asNumberOfDistinct__customer_class_id) {
14 gamma__for__customer_class_id__asColumnOfTable__customer_class[a,b] =
15 categorical_lpmf(b|theta__asColumnOfTable__customer_class);
16 }
17 }
18
19 for (a in 1:Total__ofTable__customer_
20       _asNumberOfDistinct__customer_id) {
21 target += log_sum_exp(
22       gamma__for__customer_class_id__asColumnOfTable__customer_class
23             [a]);
24 }
25 }
```

Zunächst ein Paar Erklärungsworte zu den ineinander verschachtelten Schleifen über Laufvariablen a und b im Stan-Code. Für jeden Customer_id-Wert (Laufindex a der äußeren for-Schleife) wird die Verteilungsfunktion „categorical_lpmf“ aufgerufen. Dieser Verteilungsfunktionsaufruf ist sowohl im SQL- als auch im Stan-Code vorgesehen. Es handelt sich um die Umsetzung des Zielfunktions-Summanden $\sum_{m \in M} \lg \mathit{Categorical}(z_m|\theta)$, und „categorical_lpmf“ benennt eine logarithmierte Wahrscheinlichkeitsfunktion einer kategorischen/multinomialen Verteilung (LPMF = Log Probability Mass Function). Diese Verteilungsfunktion nimmt einen diskreten Parameter als eines der Argumente entgegen. Wie bereits erwähnt: diskrete Modellparameter kann Stan nicht schätzen (bzw. nicht samplen), daher muss man zum einem *gamma*-Hilfsarray greifen. So wird Aufruf der kategorischen Verteilungsfunktion in eine zusätzliche for-Schleife über b eingewickelt. Diese zusätzliche Schleife läuft über die zweite Dimension vom *gamma*-Array (entspricht der Tabellenspalte Customer_class.Customer_class_id als einem Schlüssel der Tabelle, den der diskrete Parameter referenziert). Da $\sum_{m \in M} \lg \mathit{Categorical}(z_m|\theta)$ laut Annahme der erste Summand gewesen sein soll, der sich auf einen diskreten Parameter bezieht, wird der *gamma*-Array erstens am Anfang des Stan-**model**-Blocks definiert (Zeilen 3–8), und zweitens mit den Ergebnissen der Verteilungsfunktions-Aufrufe initialisiert (einfache, überschreibende Zuweisung via =-Operator innerhalb des Schleifenrumpfes in Zeile 14 des Listings 3.20).

Am Ende wird dem **model**-Block des Stan-Programms eine zusätzliche for-Schleife hinzugefügt, die diesmal über die erste Dimension vom *gamma*-Array läuft. Diese erste Dimension vom *gamma*-Array ist Customer.Customer_id als Schlüssel der Tabelle, in der der diskrete Modellparameter als eine *stan_parameter_int*-getypte Spalte drin steht. Die zusätzliche for-Schleife aggregiert die Inhalte des *gamma*-Arrays und sorgt für das Einfließen der aggregierten Wahrscheinlichkeitswerte in die Stan-interne **target**-Variable (siehe Zeilen 19–24 des Listings 3.20). Die **target**-Variable steht für den Wert der Zielfunktion.

2. Übersetzungsbeispiel: Aktualisierung der gamma-Werte, Behandlung der Unterzugriffe am diskreten Modellparameter Im vorigen Übersetzungsbeispiel hieß es, der beschriebene Summand sei der erste, der sich auf einen diskreten Parameter bezieht. Das heißt, er sei auch der erste Summand, durch den ein *gamma*-Array im Stan-Code erzeugt und initialisiert wurde. Ungeklärt bleibt also nur noch die Herangehensweise für den Fall, dass der *gamma*-Array bereits initialisiert ist. Jedes weitere Ergebnis von Verteilungsfunktions-Aufrufen, das in ein nichtleeres Feld vom *gamma*-Array reingeschrieben werden soll, das in der Arrayzelle von *gamma* stehende Ergebnis aktualisieren, ohne es zu überschreiben. Diese Lücke möchte ich anhand eines neuen Übersetzungsbeispiels schließen. Um die Gelegenheit auszunutzen werde ich am neuen Beispiel außerdem zeigen, wie der SQL2Stan-Übersetzer vorgeht, wenn der der SQL-Programmierer in der Abfrage eines diskreten Modellparameters (irgendwo in einem Verteilungsfunktions-Argument) zusätzliche Unterabfragen/Unterzugriffe definiert.

Als Beispiel wird diesmal der SQL-Code für einen anderen ZF-Summanden dienen, mathematisch definiert als $\sum_{n \in N} \lg \mathit{Categorical}(w_n|\phi_{z_{m_n}})$ (SQL-Code im Listing 3.21). Interessant an dem neuen Beispiel-ZF-Summanden sei eine didaktisch relevante Annahme, er wäre **nicht der erste übersetzte Zielfunktions-Summand, der sich auf einen diskreten Modellparameter** Customer_class_id aus der Tabelle Customer **bezieht**. Das bedeutet: im generierten Stan-Code wird man **keine Zuweisung** (=), sondern eine **Aktualisierung** (+=) der *gamma*-Array-Werte vorfinden. Ein weiteres neues Detail sei die Tatsache, dass das zweite Argument von der aufgerufenen Verteilungsfunktion *categorical_lmpf* eine ganze Reihe an Unterzugriffen hat. Mitten in diesen Unterzugriffen befindet sich der diskrete Parameter – und man hat ja gesehen, dass der generierte Stan-Code diskrete Parameter anders handhabt als beobachtete bzw. zu schätzende Modellvariablen.

Weiter im Listing 3.21 findet man den SQL-Code des in diesem zweiten Übersetzungsbeispiel adressierten ZF-Summanden.

Listing 3.21: SQL-Code des ZF-Summanden $\sum_{n \in N} \lg \mathit{Categorical}(w_n|\phi_{z_{m_n}})$. Bei der Auswertung der entsprechenden ZF sei dies nicht der erste Summand, der sich auf den latenten diskreten Modellparameter z (leere Spalte Customer_class_id in der Tabelle Customer) beruft.

```
1  with "log cat(w_n|phi_z_m_n)" as (
2    with "posterior__w_n" as (
3      select
4        product_id,
5        product_purchase_id
6      from
7        product_purchase
8    ),
9    -- prior-Zusammenstellung:
10   -- product_purchase_id = n --> customer_id = m
11               --> customer_class_id = z --> phi
```

```
12  "prior__m_n" as (
13    select
14      product_purchase_id,
15      customer_id
16    from
17      product_purchase
18  ),
19  "prior__z_m" as (
20    select
21      customer_id,
22      customer_class_id -- latent, ganzzahlig/diskret
23    from
24      customer
25  ),
26  "prior__phi_z" as (
27    select
28      customer_class_id,
29      phi
30    from
31      customer_class_product
32  )
33  select
34    LoopIndex.product_purchase_id,
35    categorical_lpmf(
36      ARRAY(
37        select
38          product_id
39        from
40          posterior__w_n
41        where
42          product_purchase_id = LoopIndex.product_purchase_id
43        order by
44          product_purchase_id
45      ),
46      ARRAY(
47        select
48          phi
49        from
50          prior__phi_z
51        where
52          customer_class_id in (
53            select
54              customer_class_id
55            from
```

```
56                  prior__z_m
57               where
58               -- Unterzugriff unter einem diskreten Modellparameter
59                  customer_id in (
60                     select
61                        customer_id
62                     from
63                        prior__m_n
64                     where
65                        product_purchase_id = LoopIndex.
                              product_purchase_id
66                     order by
67                        product_purchase_id
68                  )
69               order by
70                  customer_id
71            )
72         order by
73            customer_class_id,
74            product_id
75      )
76   ) as categorical_lpmf
77  from
78     posterior__w_n as LoopIndex
79 )
80 select
81   sum(categorical_lpmf) as "log p(w|phi)"
82 from
83   "log cat(w_n|phi_z_m_n)"
```

Der im Listing 3.21 gezeigte SQL-Code eines ZF-Summanden wird in ein Stan-Konstrukt im **model**-Block des generierten Stan-Programms übersetzt. Das Ergebnis der Stan-Übersetzung wird ein Stück weiter unten im Listing 3.22 dargestellt; um den Leser mit den bereits bekannten Prinzipien nicht zu langweilen, wurden aus dem Stan-Code zu diesem ZF-Summanden folgende, bereits kennengelernte Details rausgenommen: 1) Definition vom *gamma*-Array am Anfang des model-Blocks sowie 2) die *gamma*-bezogene *target*-aktualisierende (d.h. den ZF-Wert aktualisierende) Schleife am Ende des model-Blocks. In dem aus dem SQL-Code des obigen Listings 3.21 generierten Stan-Code (Listing 3.22) beachte man folgende Details:

- Aktualisierende (ergo eine nichtinitialisierende) Zuweisung += (Zeile 8 im Listing 3.22)

- Die dem den diskreten Modellparameter untergeordneten Unterabfragen im SQL-Code (Zeilen 57–68 im Listing 3.21) wurden nach Stan übersetzt, und wurden im Stan-Code zu Unterabfragen in der ersten Dimension von gamma (Listing 3.22, Zeile 8, Codegerüst **gamma__array[unterabfrage, b]**)

Listing 3.22: Der um einige dem Leser bereits bekannte Konzepte gekürzter Stan-Code des ZF-Summanden $\sum_{n \in N} \lg Categorical(w_n | \phi_{z_{m_n}})$, übersetzt aus dem SQL-Code im Listing 3.21. Bei der Auswertung der entsprechenden ZF sei dies nicht der erste Summand gewesen, der sich auf den latenten diskreten Modellparameter z (leere Spalte Customer__class__id in der Tabelle Customer) beruft.

```
1  model {
2
3  for (a in 1:Total__ofTable__product_purchase
4                    __asNumberOfDistinct__product_purchase_id) {
5  for (b in 1:Total__ofTable__customer_class
6                    __asNumberOfDistinct__customer_class_id) {
7
8  gamma__for__customer_class_id__asColumnOfTable__customer_class[
      customer_id__asColumnOfTable__product_purchase[a] ,b ] +=
9   categorical_lpmf( product_id__asColumnOfTable__product_purchase[a]
10   | phi__asColumnOfTable__customer_class_product[b] );
11
12 }
13 }
```

Die Anwesenheit der for-Schleife über b (Zeile 5 im Listing 3.22) sollte den Leser inzwischen nicht mehr wundern; das wurde ausreichend am ersten Beispiel zu Beginn des Paragraphen geschildert. Doch der Rumpf dieser Schleife sieht nun anders aus. Da die Werte von *gamma* diesmal nicht initialisiert, sondern aktualisiert werden müssen, die Zuweisung (=) im Rumpf der inneren for-Schleife durch eine Aktualisierung (+=) ersetzt.

Die Unterzugriffe auf den diskreten Modellparameter (z_{m_n} mit z alias *customer__class__id* als diskreter Modellparameter in der Tabelle *customer*) nehmen Einfluss auf die Übersetzung vom SQL-Code nach Stan. Man erkennt es an den veränderten *gamma*-Arrayzugriffen im Schleifenrumpf (Zeile 8 im Listing 3.22, Unterabfragen in der ersten Dimension von gamma). Alle Unterzugriffe am diskreten Modellparameter ersetzen im Stan-Code-Schleifenrumpf den Zugriff auf die erste Dimension von *gamma* [6]. Darüber hinaus erkennt man am übersetzten zweiten Argument der Verteilungsfunktion (*phi___asColumnOfTable___customer__class__product[b]* im Schleifenrumpf), dass der Zugriff auf den diskreten Modellparameter – mitsamt aller Unterzugriffe darunter – einfach durch die Laufvariable b der inneren b-Schleife ersetzt wurde. Mit anderen Worten: das Konstrukt z_{m_n} wurde im

[6] Im Kontext des letzten Beispiels: die erste *gamma*-Dimension ist definiert über Customer.Customer__id als Schlüssel der Tabelle, in der der diskrete Modellparameter vorkommt.

Stan-Code einfach durch die Laufvariable b ersetzt. Diese innere b-Schleife kam während der Übersetzung automatisch hinzu, weil der Summand einen diskreten Modellparameter benutzt.

An solchen Implementierungsfeinheiten wie der erläuerten Behandlung von diskreten Modellparametern erkennt man recht deutlich, warum man vor den Augen des Datenenthusiasten möglichst viel abstrahieren möchte, und warum man die Zielfunktion – ohnehin als Abstraktion der Inferenzalgorithmen – in SQL so einfach wie möglich gestalten möchte.

3.3.6. Modellspezifikation in SQL: warum keine expliziten Joins?

Beim Zusammenstellen der Zielfunktion hat man sich bestimmt gefragt, warum man die zum Definieren eines ZF-Summanden benötigten Spalten aus dem integrierten Schema herauspicken und als Views im WITH-Teil der ZF-Summandenabfrage zusammenstellen muss. Zwar wurde dies damit begründet, dass es dem Zweck der eindeutigen Zuordnung von den Stan-Arrays zu den beim SQL-Programmieren genutzten Relationsspalten dient, und dass man in diesen Views via WHERE-Bedingungen Slicing oder andere Filter auf die Modellentitäten anwenden kann.

Beispiel mit expliziten Inner-Joins

Wenn man die technische Seite dieser Argumente nicht versteht, könnte man sich fragen: „Warum nicht vielleicht mehrere Tabellen über explizite JOINs verknüpfen und statt mehreren WITH-Views (die auf einzelne Spalten und Zugriffe auf diese Spalten hinauslaufen) nur wenige Views mit vielen Spalten abfragen? Und wozu die ganzen ORDER-BY-Zusätze überall?“. Diese Fragen werden an einem speziell dafür konstruierten Beispiel beantwortet.

SQL ließe die Formulierung des ZF-Summanden $\sum_{n \in N} \lg \mathit{Categorical}(w_n | \phi_{z_{m_n}})$ – den man bereits im vorhergehenden Listing 3.21 kennenlernen durfte – auch in einem deutlich kürzeren SQL-Code zu, wie man ihn beispielsweise im Listing 3.23 sieht. Explizite Joins sind ja im SQL-Standard vorgesehen, daher hätten sie auch im SQL2Stan-Dialekt theoretisch syntaktisch unbedenklich und daher erlaubt sein können; das ist jedoch nicht der Fall, und der Leser wird in diesem Kapitel erfahren warum.

Listing 3.23: SQL-Code für den ZF-Summanden $\sum_{n \in N} \lg \mathit{Categorical}(w_n | \phi_{z_{m_n}})$ als didaktisches Beispiel für den syntaktisch möglichen, aber kontraproduktiven Einsatz von expliziten (Inner-)Joins im WITH-Teil des ZF-Summanden. Dieser Code ist semantisch gleich mit dem Code aus dem Listing 3.21, jedoch fällt dieser Code hier kürzer aus – durch den Einsatz von expliziten Joins. Die Schachtelung $\phi_{z_{m_n}}$ entfällt.

```
1 with "log cat(w_n|phi_z_m_n)" as (
2   "needed_tables_joined" as (
3      select -- alle benötigten Spalten auf einmal
4         phi,
```

```
5           customer_class_id,
6           customer_id,
7           product_purchase_id
8       from
9           customer_class_product
10          join -- expliziter Inner-Join
11          customer using (customer_class_id)
12          join -- expliziter Inner-Join
13          product_purchase using (customer_id)
14    )
15    select
16       LoopIndex.product_purchase_id,
17       categorical_lpmf(
18          ARRAY(
19             select
20                product_id
21             from
22                needed_tables_joined
23             where
24                product_purchase_id = LoopIndex.
                     product_purchase_id
25          ),
26          ARRAY(
27             select
28                phi
29             from
30                needed_tables_joined
31             where
32                product_purchase_id = LoopIndex.
                     product_purchase_id
33          )
34       ) as categorical_lpmf
35    from
36       needed_tables_joined as LoopIndex
37 )
38 select
39    sum(categorical_lpmf) as "log p(w|phi)"
40 from
41    "log cat(w_n|phi_z_m_n)"
```

Auf diese Join-orientierte Weise würden mehrere Tabellen zu einem einzigen gemeinsamen View zusammenfließen, der alle nötigen Spalten enthält.

Alle Unterzugriffe im zweiten Argument der Verteilungsfunktion *categorical_lp-*

mf, die früher mit der Zusammenstellung von $\phi_{z_{m_n}}$ zusammenhingen, entfallen in den Zeilen 26–33 (Listing 3.23) dabei vollständig! Allgemein ist es sehr schlecht, wenn wichtige Informationen über die Modellstruktur wegfallen – insbesondere bei einem Projekt, welches sich gegenüber Datenenthusiasten positioniert, die gern eigene Modellstrukturen definieren. Durch den Wegfall der modellstrukturspezifischen Details über die Parameterschachtelung (Unterzugriffe) ist es nicht mehr möglich, automatisch eine Inferenzimplementierung zu generieren. Die (Unter-)Zugriffe rund um den diskreten Modellparameter Customer.Customer_class_id (z) sind nämlich unbedingt notwendig, um die Manipulationen am *gamma*-Hilfsarray im generierten Stan-Code zu gestalten. Nach ihrem Wegfall kann man den Übersetzungsansatz mit den *gamma*-Arrays nicht mehr nutzen (und vermutlich auch keinen anderen Übersetzungsansatz mehr).

Didaktisch relevante Änderung am Listing 3.23

Die Modellstruktur ist also wichtig. Angenommen, man behält die dafür wichtigen Unterzugriffe bei, sodass keine Information über die Modellstruktur verloren geht, und nutzt dennoch einen expliziten (Inner-)Join im WITH-View der ZF-Summandenabfrage. Um die Modellstruktur über Unterabfragen zurückzubringen wird beispielsweise das zweite Argument des oberen Beispiels (Zeilen 26–33 im Listing 3.23) durch folgenden Code (Listing 3.24) ersetzt:

Listing 3.24: Das zweite Argument der Verteilungsfunktion *categorical_lpmf* aus dem Listing 3.23 (dortige Zeilen 26–33) wird detaillierter beschrieben: eer Programmierer teilt die Schachtelung der Zugriffe in diesem Argument ($\phi_{z_{m_n}}$) diesmal explizit mit. Ein expliziter Join sei im WITH-Teil des ZF-Summanden dennoch verwendet.

```
1  ARRAY(
2  select
3     phi
4  from
5     needed_tables_joined -- Join aus dem Listing 3.23
6  where
7     customer_class_id in (
8        select
9           customer_class_id
10       from
11          needed_tables_joined
12       where
13          customer_id in (
14             select
15                customer_id
16             from
17                needed_tables_joined
```

```
18                where
19                   product_purchase_id = LoopIndex.
                        product_purchase_id
20             )
21     )
22 )
```

Die wichtige Modellstruktur ist durch Schachtelung der Elemente gegeben. Alle benötigten Modellentitäten verstecken sich im expliziten Join.

Der Join wird aus Kostengründen nicht materialisiert Man möchte diesen Join aber nicht ausrechnen, materialisieren und auf irgendeine Weise in eine Array-/Matrizen- oder Vektor-Darstellung bringen. Für kleine bis mittlere Datensätze zieht es signifikante Rechenzeit- und Speicherkosten mit sich. Im Falle von Big Data wird die Materialisierung eines Joins zusätzlich zu den bereits bestehenden Tabellendaten vielleicht auch gar nicht gehen, und man wird schnell gezwungen, sich stattdessen einen anderen Ansatz zu überlegen. Zusammengefasst: die Vorberechnung des expliziten Joins und dessen Überführung in Arrays ist aufgrund von Kosten-Overheads zu vermeiden; die Tabellen müssen physikalisch so bleiben wie sie sind. Also wenn ein expliziter Inner-Join, dann höchstens auf einer abstrakten Ebene von SQL.

Eine rhetorische Frage: wie würde der Übersetzer von SQL nach Stan dann vorgehen? Er verfügt über die ZF-Summanden und ihre Struktur, und er weiss, welche Spalten in den expliziten Joins drin sind. Man bedenke, dass die Materialisierung vom Join umgangen werden soll.

Um es kurz zu halten: der Compiler wird versuchen, die Join-Zugriffe so zu übersetzen, dass diese als Zugriffe auf bereits vorhandene Tabellen umgesetzt werden. So kann das Ausrechnen des Joins vermieden werden. Das grundlegende Problem dabei ist jedoch, dass man beim Zugriff auf eine Join-Spalte nicht immer genau sagen kann, aus welcher am Join beteiligten Tabelle diese Spalte eigentlich herkommt. Dieses Problem macht die Verwendung expliziter Joins ungeeignet zur automatischen Abbildung der SQL-beschriebenen Zielfunktion auf Array-Operationen, weil SQL2Stan jeden Array einer konkreten Spalte aus einer konkreten Tabelle zuordnet. In folgenden Paragraphen wird diese Problematik ein wenig ausführlicher behandelt.

Rückblick: Struktur der Argumente von Verteilungsfunktionen Da die zur Formulierung eines ZF-Summanden benötigten Spalten durch einen Join in einem View zusammengebracht werden, so müssen die Argumente einer Verteilungsfunktion (wie z.B. ein solches Argument im Listing 3.24) auf die Join-Spalten zugreifen. Demzufolge wird der Übersetzer beim Auswerten der Verteilungsfunktionsargumente die darin enthaltenen Spalten dem Join zuordnen. Dieser wird nicht materialisiert, daher müssen die Spalten stattdessen den tabellen zugeordnet werden, die am Join beteiligt sind. Das behalten wir als Kontext bei, und rufen in Erinnerung, wie die

Argumente von Verteilungsfunktionen strukturiert sind – das ist wichtig beim Parsen und Auswerten.

Ein typisches Verteilungsfunktionsargument besteht aus einer SQL-Abfrage, in die möglicherweise andere gleichstrukturierte Unterabfragen eingebettet sind. Jede solche SQL-Abfrage im Verteilungsfunktionsargument besteht aus drei Teilen, die in der gleichen hierarchischen Ebene liegen: SELECT, FROM und WHERE. Beispiele für solche Teile findet man in den Zeilen 2, 4 und 6 (alternativ auch Zeilen 8–10–12 sowie Zeilen 14–16–18) im Listing 3.24.

Im SELECT-Teil steht eine Spalte, die man im Kontext des Verteilungsfunktion-Arguments abrufen möchte. Die Spalte im SELECT-Teil kommt aus einer bestehenden Tabelle (physikalisch gesehen also nicht aus dem nicht-zu-materialisierenden Ergebnis vom expliziten Inner-Join) und korrespondiert intern zu einem Array, weil SQL2Stan jede Tabellenspalte intern auf ein Array abbildet.

Im FROM-Teil steht der Name einer Tabelle, aus der die SELECT-ierte Spalte kommt. Wenn im FROM-Teil nur ein Join von mehreren Tabellen drinsteht, kann der Compiler die SELECT-ierte Spalte zu ihrer Herkunft – der ursprünglichen, ungejointen Tabelle – nicht immer genau zuordnen (das ist ein Problempunkt). Und schließlich im WHERE-Teil der SQL-Abfrage steht eine Unterabfragenbedingung: entweder eine neue Unterabfrage (eine SQL-Abfrage wie in Zeilen 12–14 des Listings 3.24) oder eine Bindung an einen Spaltenwert (wie in Zeilen 18–19 des Listing 3.24).

Angefangen vom Zugriff, der am tiefsten verschachtelt wurde (Listing 3.24, Zeilen 14–19), läuft der Übersetzer von innen nach außen die Unterabfragen entlang und versucht die Spalten aus den ausgewerteten Abfragen den tatsächlich existierenden, am expliziten Join beteiligten Tabellen zuzuordnen. Der Compiler will nämlich die Spalteninstanzen, die an den Unterabfragen beteiligt sind, auf Relationen (und somit auf Arrays) abbilden können. So schaut sich der Compiler einerseits die Spalte im WHERE-Teil einer (Unter-)Abfrage, andererseits die Spalte im SELECT-Teil an. Der Compiler muss im relationalen Schema eine Tabelle finden, die sowohl die Spalte im WHERE-Teil als auch die Spalte im SELECT-Teil einer (Unter-)Abfrage zusammenbringt. Das mag manchmal gehen. In der innersten Abfrage des zuletzt gezeigten SQL-Codes (Zeilen 14–19 im Listing 3.24) würde ein Übersetzer die ungejointe Tabelle Product_purchase finden, weil diese Tabelle sowohl die Spalte Customer_id (SELECT-Teil der Abfrage) als auch die Spalte Product_purchase_id (WHERE-Teil der Abfrage) beinhaltet. Das ist allerdings ein glücklicher Zufall, denn das Zuordnen von den Spalten eines expliziten Inner-Joins zu den an ihm beteiligten Tabellen kann nicht immer klappen. Der SQL-Programmierer kann in den WHERE-Bedingungen nämlich beliebig gestaltete Unterabfragen formulieren. In den SELECT- und WHERE-Teilen von Abfragen können beliebige Kombinationen aus Spalten eines Inner-Joins stehen. Somit kann nicht garantiert werden, dass der Compiler den teueren Join umgehen kann, indem er eine zu einem (Unter-)Zugriff passende, am expliziten Inner-Join beteiligte „echte“ Tabelle findet. „Zum Unterzugriff passend“ bedeutet: eine existierende Tabelle beinhaltet zwei Spalten A und B, sodass man mit durch Werte b $\in$ B Werte von a $\in$ A nachschlagen kann, ergo – den Unterzugriff A_B bewerkstelligen. Mit einer „echten“ Tabellen ist

einfach eine bereits existierende Tabelle gemeint, d.h. kein Resultat eines Joins, der materialisiert werden müsste.

Um die Join-Materialisierung garantiert umgehen zu können, müsste man dem SQL-Programmierer glaubhaft mitteilen, dass man Joins zwar nutzen kann, aber nur so, dass der Join nicht wirklich ausgerechnet werden müsste. Es wäre eine etwas bizarre Anforderung, die im Grunde darauf hinausläuft, dass man in einem Zugriffsschritt (d.h. bei einer Unterabfrage) nur zwischen den Spalten einer existierenden Tabelle zugreifend „springen“ darf. Diese Einschränkung auf die Rahmen nur einer existierenden Tabelle erinnert schon sehr deutlich die Umsetzung, die man bereits in all den obigen Kapiteln vorgestellt bekommen hat und die auch viel übersichtlicher ist. Der SQL2Stan-typischer Ansatz mit den Views, wo man konkret angibt, welche Spalte man braucht und aus welcher Tabelle sie kommt (plus der Schlüssel dieser Tabelle) verzichtet nicht auf Join; er macht sie nur implizit durch IN-Operatoren in den WHERE-Bedingungen. Dadurch verlagert SQL2Stan die impliziten Joins syntaktisch direkt an die Stellen, wo diese Joins benötigt werden.

Order-By-Angaben: Konsequenz der Abbildung von Tabellen-/View-Strukturen auf Arrays Der Übersetzer im SQL2Stan-Prototyp achtet zwar nicht auf die Order-By-Zusatzangaben (wie z.B. Zeilen 43–44 oder 72–74 im Listing 3.21), doch diese Angaben dienen der syntaktischen Folgerichtigkeit.

Die Argumente der Verteilungsfunktionen (in den ZF-Summanden) bedienen sich der Tabellenspalten als Datenquellen – im Sinne von geordneten, array-nahen Datenkonstrukten. Manchmal werden dabei jedoch implizite Joins nach Art „WHERE id IN Unterzugriff“ verwendet (wie in Zeilen 6–7 oder 12–13 des Listings 3.24). Die klassischen Arten der Join-Berechnung garantieren jedoch nicht, dass die vor dem Join gegebenen Sortierungen im fertigen Join beibehalten werden – es sei denn, die Join-Ergebnisse würden mit ORDER-BY-Befehlen geordnet. In SQL2Stan muss die Datenbank sowieso keine Joins berechnen, und die in SQL formulierte Zielfunktion dient nur zur Generierung der Inferenzimplementierung. Mit Order-By-Zusatzangaben in den Argumenten von Verteilungsfunktionen geht man syntaktisch einfach konsequent vor. Dadurch verfolgt man die Umsetzung der „Spalten sind Arrays“-Idee auf eine syntaktisch unwidersprüchliche Weise.

3.4. Fazit zum SQL2Stan-Konzept

Das Projekt SQL2Stan bettet die sprachliche Struktur der probabilistischen Sprache Stan in die Datenbanksprache SQL ein, um Datenmanagement und statistische Inferenz syntaktisch in einer gemeinsamen sprachlichen Abstraktion zusammenzuführen. Mit SQL2Stan muss ein SQL-Programmierer keine zusätzliche(n) probabilistische(n) Programmiersprache(n) erlernen und muss seine probabilistischen Programme nicht eigenhändig mit einem DMS verbinden. SQL2Stan schlägt einen Ansatz vor, um den Workflow zur probabilistischen Programmierung und Bayes'schen Inferenz vollständig in den Datenbankkontext einzupflegen.

Der SQL2Stan-Ansatz eignet sich für strukturierbare Daten. Sind die Daten bereits strukturiert, so formuliert man ihre Struktur auf eine relationale Weise und bindet daran ein statistisches Modell an. Sind die gegebenen Daten hingegen noch unstrukturiert, könnte man davon ausgehen, dass die Struktur einfach später nachkommt – dann kann man sich überlegen, bei ihrer nachträglichen Strukturierung zum SQL2Stan-Ansatz zu greifen.

1. Anforderung: Übersetzbarkeit Bayes'scher Modelle in DB-Schemata berücksichtigt Vom Entwurf eines Bayes'schen Modells bis zur Erstellung eines integrierten relationalen Schemas (wo alle zur Inferenz benötigten Modellentitäten drin stehen) soll ein SQL2Stan-Nutzer von [RH16] Gebrauch machen. Die Berücksichtigung der Übersetzbarkeit von Bayes'schen Modelle in relationale Schemata war im Motivationskapitel 1 eine von drei wichtigsten Anforderungen, die erfüllt werden müssen, damit ein neuer Ansatz sich gegen bisher vorgestellte Inferenzsysteme mit SQL-Interface positionieren kann. Diese Anforderung wird von SQL2Stan erfüllt. ML-Modelle werden dabei explizit nicht als „Blackbox" gehandhabt; SQL2Stan ermöglicht den schnellen Bau von Modellprototypen und begünstigt Experimente mit Bayes'scher Inferenz an DMS-gelagerten Daten.

2. Anforderung: unkomplizierte Spezifikation eigener (auch hierarchischer) Modelle Mit den Namen der Modellentitäten aus einem alle Modellentitäten umfassenden relationalen Schema kann der SQL2Stan-Nutzer in reinem SQL ein probabilistisches Programm schreiben. Mit anderen Worten, SQL2Stan-Nutzer können mit wenig Aufwand eigene statistische Bayes'Sche Modelle in einem einfachen probabilistischen SQL-Dialekt formulieren. Hierarchische Modelle (wie LDA oder Naïve Bayes) werden explizit unterstützt; sie lassen sich in SQL sogar leichter formulieren, als in der PPL Stan, weil der Nutzer komplexe Details (z.B. Behandlung diskreter Modellparameter über gamma-Hilfsarrays) nicht beachten muss.

3. Anforderung: ML- und DMS-Komponenten austauschbar Die Inferenzimplementierung wird aus dem nutzergeschriebenen SQL-Code automatisch generiert. Die Inferenzausgabe wird automatisch zusätzlöich zu den Inferenz-Eingabedaten über das nutzerdefinierte relationale Schema abrufbar gemacht. Das Datenmanagementsystem kümmert sich automatisch um die physikalische Datenverwaltung aller Modelldaten. SQL2Stan kümmert sich als Middleware zwischen dem DMS und der Inferenzsoftware um den Datentransport zwischen diesen Softwarekomponenten. Im SQL-Interface von SQL2Stan werden die Inferenzdetails genauso wegabstrahiert wie die genauen Details zum Datenmanagement. Diese high-level-Abstraktion macht sowohl das DMS- als auch ML-Backend austauschbar.

Das als Schnittstelle genutzte relationale Schema ist dank zusätzlicher SQL2Stan-typischer Constraints übersetzbar in eine Array- bzw. Vektor-Darstellung. Das erlaubt den Einsatz von vielen Softwarewerkzeugen, die solche Mittel wie Arrays, Vektoren, Tensoren und Relationen verwenden: u.a. Datenmanagementsysteme (v.a.

Array-DBMS wie SciDB) und ML-Frameworks (TensorFlow). Wie man sieht, obwohl das SQL2Stan-Interface zum Nutzer hin relational ist, muss es für Backends nicht unbedingt relational sein.

ML-Algorithmen austauschbar, auch bei gleichem ML-Backend Die ML-Schicht wird vor dem Nutzer ausgeblendet. SQL2Stan behält es im Verborgenen, welche Verfahren (Variational Inference, Hamiltonian Monte Carlo oder andere), welcher Sampler (No-U-Turn-Sampler oder vielleicht ein modifizierter LightLDA-Sampler) genutzt werden. Der SQL2Stan-Prototyp unterstützt sowohl VI als auch HMC als Inferenzalgorithmen. Dadurch werden bei SQL2Stan auch die Inferenzalgorithmen und ihre Implementierungsdetails austauschbar, ohne dass das SQL-Interface zum Nutzer hin geändert werden muss. Ferner besteht begründete [Yut+17; BKM17] Hoffnung, dass die Auswahl der geeigneten Inferenzimplementierung (u.a. auch welches Verfahren und welcher Sampler genommen wird) automatisiert werden kann – idealerweise anhand der automatisch analysierten Beschaffung der gegebenen Daten.

Deklarativ und parallelisierbar Die Deklarativität von SQL lässt viel Freiraum für Parallelisierung von Algorithmen [Jan+19]. Es gibt zwei Parallelisierungsansätze: *data parallel* (Daten werden unter den Prozessorkernen aufgeteilt, jeder Kern berechnet dieselben Parameter) sowie *model parallel* (jeder der Kerne bekommt gleiche Daten, auf denen die Kerne allerdings unterschiedliche Parameter berechnen). Viele ML-Systeme unterstützen entweder den einen oder den anderen Ansatz, doch manche Nutzer wollen beide Ansätze in einem System haben (siehe Projekt Angel [Jia+17]). Das deklarative SQL2Stan-Interface bietet keine Einschränkungen, um beliebige Parallelisierungsmethoden (z.B. Parameterserver zur Knotenkommunikation bei verteilten MCMC-Algorithmen E.2.11) für Prozesse in den Inferenzworkflow einzupflegen.

Fazit zum SQL2Stan-Prototypen Der umgesetzte Prototyp von SQL2Stan ist keine Big-Data-Lösung für BAyes'sche Inferenz: der Prototyp ist nicht für Rechnercluster ausgelegt, und muss Daten zwischen dem DMS-Backend (PostgreSQL) und der ML-Software (Stan) hin- und herkopieren. Doch der funktionierende Prototyp zeigt: es ist möglich, eine deklarativ bedienbare sprachliche Brücke zwischen dem Datenmanagement und den Werkzeugen für automatische Inferenz zu schlagen. Das gelingt, obwohl ML und DMS unterschiedliche Sprachen sprechen, unterschiedliche Modellarten nutzen, sich auf unterschiedliche Aspekte konzentrieren und unterschiedliche Details wegabstrahieren. Es konnte prototypisch belegt werden, dass für die Verknüpfung von Datenmanagement und Bayes'scher Inferenz SQL-Mittel ausreichen. Auch als Nutzer mit begrenztem Wissen über SQL, Datenbanken und statistische Verteilungen kann man mit Hilfe von [RH16] und SQL2Stan die Bayes'sche Inferenz an eigenen Modelle durchführen.

4. Fallbeispiel: Onlineshop-DB

Als Fallbeispiel wird uns ein Sportartikel-Onlineshop dienen, der über ein bereits vorhandenes Datenbank-Managementsystem verfügt. In der Datenbank werden angebotene Artikel, Kunden und durch Kunden getätigte Bestellungen gespeichert. Der Shopbetreiber möchte von Bayes'scher Inferenz profitieren. Der Anreiz, die ML-Funktionalität in ein bestehendes DBMS einzubinden, besteht konkret im Wunsch nach einem Kundenklassifikator. Eine Kundenklasse wäre eine Art Kategorie, die bestimmte Kunden mit ähnlichen Produktinteressen in sich vereint.

Ein Online-Versandhandel soll seine Kunden klassifizieren können, sodass er – unter anderem – seine Kundschaft besser erreichen kann. Der Shop soll wissen, welche Produktklasse eine Person am besten anspricht. Bei Bestandskunden, die ein Newsletter-Abo haben, sollen per E-Mail bestenfalls nur solche Angebote zugestellt werden, die sie zu neuen Käufen anregen sollen. Zum Beispiel, Bestandskunden würden über den Newsletter einmal im Monat die Top-10-Auswahl der Produkte bekommen, zusammengestellt anhand der durch Modellinferenz errechneten Werte für populäre Klassen und Produkte.

Man kann nicht nur Bestandskunden klassifizieren. Interessenten, die nur auf der Shopseite nach Produkten stöbern, ohne etwas zu kaufen, sollen zum Kauf animiert werden, indem ihnen relevante Werbung gezeigt wird. Relevant bedeutet, dass man die Interessen von einer Person klassifizieren können soll. Dazu wird ein Seitenbesucher einer Klasse zugewiesen. Die Klassifikation eines Seitenbesuchers geschieht anhand der Artikel, die vom genannten Subjekt entweder angeschaut wurden.

Man kann einen solchen Klassifikator sehr flexibel trainieren: wichtig sind nur Personen (egal, ob Bestandskunden oder Seitenbesucher) sowie Produkte, die mit diesen Personen in Verbindung stehen (durch Käufe oder einfach durch das Anschauen der Produktseite). Man kann den Klassifikator mit Produktseitenbesuchen trainieren und ihn auf Bestandskunden anwenden (und umgekehrt), man kann den auf Seitenbesucherdaten trainierten Klassifikator bei neuen Seitenbesuchern anwenden, und mit dem auf Kundenkäufen trainierten Klassifikator Bestandskunden klassifizieren. Demzufolge kann man in den im diesem Kapitel erklärten Konzepten die Begriffe Kunden-ID und Besucher-Session-ID gleichsetzen, wenn es einem besser gefällt. Es ist nur wichtig zu beachten, dass die Mischvarianten (Klassifikator trainiert auf Besucherdaten und anzuwenden auf Bestandkunden oder umgekehrt) eine zusätzliche Datenbank erfordern würden. In diesem wird es um eine Datenbank mnit Kundenkäufen gehen, die man zum Klassifikatortraining durch Bayes'sche Inferenz verwenden wird. Durch das leichte Umdenken von Bestandskunden auf Shopseitenbesucher kriegt der Leser zugleich ein zusätzliche Anwendungsvariante mit.

Ein durch Bayes'sche Inferenz bereitgestellter Naïve-Bayes-Kundenklassifikator nimmt Bezüge auf die Größenverhältnisse der jeweiligen Kundenklassen. Außerdem gibt er eine Art Produktranking für die Kundenklassen an (Wahrscheinlichkeiten des Auftretens eines bestimmten Produktes im Zusammenhang mit einer bestimmten Kundenklasse). Der Shopbetreiber kann diese beim Klassifikatorbau anfallenden Informationen sinnvoll nutzen, z.B. die Größenverhältnisse zwischen den Kundenklassen auswerten sowie die Bezüge der Kundenklassen zu den angebotenen Produkten analysieren, um seine Marketing- und Sortimenterweiterungs-Strategien zu verbessern.

Im diesem Fallbeispiel wird es um einen Onlineshop handeln, der sich auf Sportartikel spezialisiert. Von den Sportartikeln lässt sich oft ganz gut darauf schließen, für welche Sportart sie bestimmt sind (z.B. für Laufsport, Kraftsport, Joga, Radfahren, Tanzen oder Hometraining). Über Sportarten kann man Kunden klassifizieren: ein Shopbesucher habe sich z.B. Laufschuhe, Laufsocken und atmungsaktive T-Shirts angeschaut, daher ist es sehr wahrscheinlich, dass er sich für die Klasse „interessiert sich für Laufsport" qualifiziert. Dieser Person könnte man einige meistverkaufte Artikel aus dem Laufsport-Bereich am Rand der Webseite als eine Art personalisierte „Bestseller-Auswahl" anpreisen und mit etwas Glück einen neuen Kauf in die Wege leiten. Der praxisorientierte Gedanke dazu: der Aufruf des Klassifikators soll als SQL-Anfrage an die Shopdatenbank erfolgen – eine Datenbank, die man bereits hat. Der Klassifikator soll mit aktuellen Datenbankeinträgen täglich neu trainiert werden, damit die Kundenklassen up-to-date bleiben.

Entity-Relationship-Schema der Onlineshop-Datenbank Das Datenbankschema für solch einen Sportartikel-Onlineshop kann vereinfacht wie folgt aussehen:

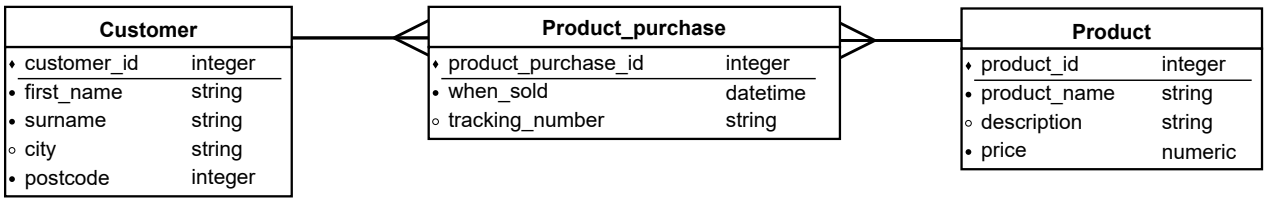


Abbildung 4.1.: Entity-Relationship-Schema der Onlineshop-Datenbank: beinhaltet ausschließlich die Entitäten zu Produkten, Kunden und Bestellungen im Onlineshop.

Die Vereinfachung betrifft z.B. die Abwesenheit von Bestellungen, insofern man eine Bestellung als Kauf mehrerer Artikel ansieht. Im hiesigen Schema gibt es nur Produktkäufe, und zwar in Form von Zuweisungen Artikel–Kunde. Es spielt keine Rolle, wann der Kunde etwas gekauft hat und ob er mehrere Artikel gekauft hat: seine gesamte Kaufhistorie wird sozusagen auf eine Liste von gekauften Produkten komprimiert.

Als Modell für die Klassifikation wurde Naïve Bayes aufgrund dessen konzeptioneller Einfachheit gewählt. Das Modell soll auf Kundenbestellungsdaten trainiert

werden. Das Modelltraining (ergo das Ermitteln von Werten für latente Variablen im Naïve-Bayes-Modell durch Bayes'sche Inferenz) offenbart konkrete Werte für versteckte, nicht-beobachtbare Modellparameter. Sobald alle Modelldaten, sowohl bekannte als auch geschätzte, in der Datenbank gelandet sind, kann die Klassifikation neuer Interessenten/Kunden per SQL-Anfrage erfolgen. Zum Trainieren des Modells braucht man im Groben die beobachtbaren Daten sowie ein ausformuliertes statistisches Modell (Naïve Bayes, gebunden an Entitäten der Onlineshop-Datenbank).

Die gegebenen Daten liegen in der Datenbank in entsprechenden Tabellenspalten und folgen einem bestimmten relationalen Schema. Das Naïve-Bayes-Modell kann (nach dem Verfahren aus [RH16]) in ein Datenbankschema überführt werden. Der Shopbetreiber lässt die relationalen Strukturen des Modells und der bekannten Datenbankdaten zusammenschmelzen, und erhält ein integriertes relationales Datenbankschema. Dadurch teilt er SQL2Stan explizit mit, welche Modelldaten bekannt bzw. unbekannt sind und welche Modelldaten geschätzt werden sollen. Wir beschäftigen uns zunächst mit dem Weg zum integrierten relationalen Datenbankschema, welches sowohl Naïve-Bayes- als auch Onlineshop-Entitäten beinhalten soll.

4.1. Klassifikator: (un-)supervised naïve Bayes

Ein passendes Bayes'sches Modell zur Kundenklassifikation ist Naïve Bayes. Seine Einfachheit und Robustheit verhelfen diesem Modell zu einer im Allgemeinen recht populären Klassifikator-Wahl [Wu+08, S. 24–27]. Es lohnt sich, einen näheren Blick auf dieses generative probabilistische Modell zu werfen.

4.1.1. Vom Bayes'schen Netz zum integrierten DB-Schema

Der Graph, den man in der Abbildung 4.2 sieht, ist ein Bayes'sches Netz. Solche Netze sind eine Möglichkeit zur grafischen Notation von Bayes'schen Modellen. Sie beschreiben generative Modelle – d.h. die statistische Art, wie die gegebenen Daten hätten „erwürfelt“ sein können – auf eine visuelle Art, und beinhalten Zufallsvariablen und Zufallsverteilungen. Ein Knoten im Graphen ist eine Zufallsvariable. Graue Knoten sind beobachtete Variablen, weisse Knoten stehen für latente/versteckte Variablen, und zuletzt die schwarzen kleinen Knoten sind Modell-Hyperparameter. Die gerichteten Kanten zwischen den Knoten entsprechen bedingten Zufallsverteilungen.

Aus solch einem Bayes'schen Netz, welches ein Datenenthusiast entweder aus einer Veröffentlichung bezieht oder selbst entwirft, soll ein integriertes Datenschema entstehen. Das integrierte Schema ist ein relationales Schema, welches die beobachteten Daten und mit den im Bayes'schen Netz modellierten unbeobachteten Daten in sich vereint. Ich verweise an der Stelle an das Kapitel 3.2, wo das integrierte relationale Schema genauer erklärt wurde.

Um ein integriertes Schema zu erstellen, braucht man vor allem eins: eine Schnittstelle zwischen dem maschinellen Lernen (das in unserem Falle vorerst nur durch

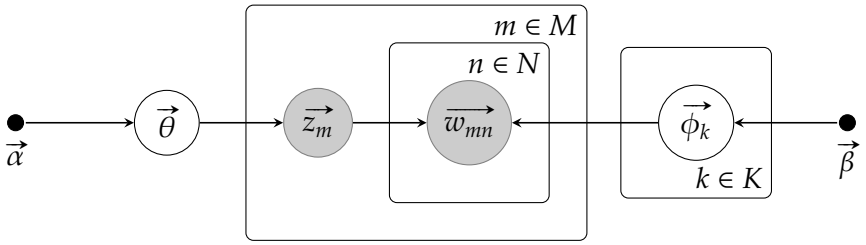


Abbildung 4.2.: Bayes'sches Netz als probabilistisches graphisches Modell zu supervised naïve Bayes (Variable z beobachtet)

ein Bayes'sches Netz vertreten ist) und den Datenbankdaten. Auf der einen Seite stehen Variablen wie m, auf der anderen – Spaltennamen wie „customer_id". Das Stichwort für eine derartige Schnittstelle lautet Entitäten-Matching.

Matching der Entitäten im Modell und in der Shop-DB

Es wäre wichtig, in Erinnerung zu rufen, welche Onlineshop-Entitäten in einem Klassifikatormodell wiederzufinden sind. Wir haben: Kunden, Kundenklassen, Produkte (Shopartikel) und schließlich Produktkäufe, die durch Kunden getätigt wurden. Hilfreich für das Modellverständnis: ein Produktkauf n ist eine 1-zu-1-Zuordnung Kunde–Artikel. Hat der Kunde mehrere Artikel bestellt, so kommen einfach zusätzliche Produktkäufe hinzu (z.B. Kauf n+1 als 1-zu-1-Zuordnung Kunde–neuer-Artikel). Mit diesem Kontext können die Modellvariablen im Bayes'schen Netz erklärt werden.

Die mit $n \in N$ beschriftete Ebene im Bayes'schen Netz (N-Plate in der Abb. 4.2) enthält die Modellvariablen zu den Artikelkäufen. Der Vektor $\overrightarrow{w_{mn}}$ ist ein Kauf: der Index m steht für die Kunden-ID, der Index n ist die Kauf-ID (eine fortlaufende, global eindeutige Identifikationsnummer für sämtliche Artikelkäufe im Shop), und der eigentliche Vektor $\overrightarrow{w_{mn}}$ gibt die ID des gekauften Produkts an. Die mit $m \in M$ beschriftete Ebene (M-Plate in der Abb. 4.2) enthält die Modellvariablen zu den Kunden. Der Vektor $\vec{z_m}$ gibt an, welcher Klasse der Kunde mit der Kunden-ID m angehört. Der Vektor $\vec{\theta}$ bringt die Klassenprävalenz zum Ausdruck. Klassenprävalenz ist einfach ein Ausdruck für die Häufigkeit von den Kundenklassen, denn manche sind häufiger anzutreffen, und manche seltener. Die Klassenprävalenz wird durch automatische Inferenz geschätzt. Was außerdem noch geschätzt werden will, ist der auf der K-Plate liegende Vektor $\vec{\phi_k}$ (Abb. 4.2): die Produktverteilung für Klassen $k \in K$ (welche Produkte sind in einer bestimmten Klassse häufiger anzutreffen, und welche weniger häufig). Das war es auch schon fast: die Vektoren $\vec{\alpha}$ und $\vec{\beta}$ wollen zuletzt als Modell-Hyperparameter erwähnt werden – eine Möglichkeit, das Modell probabilistisch anzupassen.

Modellentitäten werden mit Entitäten aus dem Schema der Onlineshop-Datenbank (Abb. 4.1) gematched. Unterhalb werden **Matching**-Details präsentiert, verpackt in eine Zusammenfassung der Modellvariablen und der damit verbundenen Größen:

- $K \equiv$ Anzahl von Kundenklassen (manuell festgelegt durch den Nutzer bzw. rechnerisch ermittelt, z.B. durch $\sqrt{V}$ als Wurzel aus der Anzahl von angebotenen Produkten).
  - $k \in K$ wird mit „customer_class_id“ in der Tabelle „Customer_class“ gematched.
- $V \equiv$ Sortimentgröße: Anzahl von Artikeln/Produkten
  - $v \in V$ wird mit „product_id“ in der Tabelle „Product“ gematched.
- $M \equiv$ Anzahl von Kunden
  - $m \in M$ wird mit „customer_id“ in der Tabelle „Customer“ gematched. Lässt sich u.U. durch eine Browser-Session-ID ersetzten (bei Shopseitenbesuchern, die keine Bestandskunden sind).
- $N \equiv$ Anzahl von insgesamt von Kunden bestellten Artikeln
  - $n \in N$ wird mit „product_purchase_id“ in der Tabelle „Product_purchase“ gematched.
- $z_m \equiv$ Zuweisung einer Kundenklasse zu einem Kunden; bei supervised naïve Bayes bekannt, bei **un**supervised naïve Bayes hingegen unbeobachtet.
  - Der Vektor $\overrightarrow{z_m}$ ist ein 1-aus-K-Vektor mit einer 1 am Index, der der Klassen-ID des Kunden m entspricht. K ist die Anzahl der Kundenklassen, k ist eine Kundenklassen-ID (z.B. k=1 sei Laufsport-interessiert, k=2 sei Kraftsport-begeistert usw.).
  - $z = \{z_1, ..., z_M\},\ z_m \in \{0,1\}^K,\ \forall m \in M : \sum_K z_{mk} = 1$
  - In der modellspezifischen Zuweisung z_m wird $z \in K$ mit „customer_class_id“ und $m \in M$ mit „customer_id“ gematched, in beiden Fällen in der Tabelle „Customer“.
- $w_{mn} \equiv$ Zuweisung: Produkt-ID $\rightarrow$ Artikelkauf durch einen Kunden.
  - Im Detail: $\overrightarrow{w_{mn}}$ ist ein 1-aus-V-Vektor aus Nullen mit einer einzigen 1 an der Stelle, deren Index der Produkt-ID vom gekauften Artikel entspricht. V ist die Anzahl der Shopartikel, v ist eine Produkt-ID.
  - $w = \{\vec{w}_1, ..., \vec{w}_N\},\ w_n \in \{0,1\}^V,\ \forall n \in N : \sum_v w_{nv} = 1$
  - In der modellspezifischen Zuweisung $w_{mn}V$ wird $n \in N$ mit „product_purchase_id“ (Schlüssel der Tabelle „Product_purchase“) gematched, und $w \in V$ mit „product_id“ bzw. $m \in M$ mit „customer_id“ als Fremdschlüssel.

- $\alpha \equiv$ Hyperparameter, Prior-Vektor für Klassenverteilung
  - $\alpha \in \mathbb{R}^K,\ \alpha_k > 0$
  - α kann mit keiner vorhandenen Entität im Datenbankschema des Onlineshops gematched werden, daher wird im Datenbankschema (zum statistischen Modell, und später auch im integrierten Modell) für diese modellspezifischen Variablenwerte eine Spalte „alpha“ in der Tabelle „Customer_class“.
- $\beta \equiv$ Hyperparameter, Prior-Vektor für Produktverteilung
  - $\beta \in \mathbb{R}^V,\ \beta_k > 0$
  - Für β wird im Datenbankschema für das statistische Modell (und später auch im integrierten Modell) eine Spalte „beta“ in der Tabelle „Product“ angelegt (analog zu α mit DB-Spalte *alpha*).
- Parameter θ = Prävalenz von Kundenklassen (welche Klassen stehen mehr im Vordergrund und welche weniger).
  - Bsp.: man habe zwei Kundenklassen, und θ sei ein Vektor (0.2,0.8), welcher die Themenprävalenz ausdrückt
  - $\theta \in (0,1)^K,\ \sum_K \theta_k = 1$
  - Für θ wird im Datenbankschema für das statistische Modell (und später auch im integrierten Modell) eine Spalte „theta“ in der Tabelle „Customer_class“ angelegt (analog zu α mit DB-Spalte *alpha*).
- Parameter ϕ_k = Produktbeliebtheit bei einer Kundenklasse, Verteilung von Artikeln innerhalb einer Klasse.
  - Hier wird angegeben, wie wahrscheinlich es ist für eine bestimmte Kundenklasse, dass man aus einem Bag-of-Items zu dieser Klasse eine bestimmte Ware zieht. Im klassentypischen Bag-of-Items ist es wahrscheinlicher, eins der in dieser Kundenklasse beliebten Produkte zu ziehen.
  - $\phi \in (0,1)^{K \times V},\ \forall k \in K : \sum_V \phi_{kv} = 1$
  - Für ϕ_k wird im Datenbankschema für das statistische Modell (und später auch im integrierten Modell) eine Spalte „phi“ in der Tabelle „Customer_class_Product“ angelegt. In diesem Kontext wird $k \in K$ mit „customer_class_id“ als Fremdschlüssel gematched.

Klassifikator als Datenbankschema

Das obige Matching, in dessen Rahmen die Modell-Entitäten mit denen aus der Onlineshop-DB gematched wurden, kann sehr hilfreich sein, wenn man das Bayes'sche Netz (als grafische Darstellung des Klassifikatormodells) in ein relationales Datenbankschema übersetzt. Zum Übersetzungsverfahren empfiehlt sich die Lektüre von

[RH16]. Das Datenbankschema, welches man aus dem gematchten und in den relationalen Kontext übersetzten Bayes'schen Netz für Naïve Bayes erhält, ist in der Abbildung 4.3 zu finden. Die genaue Übersetzung vom Bayes'schen Netz über Atomic Plate Model ins relationale Schema wird übersprungen, weil nur das Ergebnis dieser Übersetzung – das relationale Schema – von Relevanz ist.

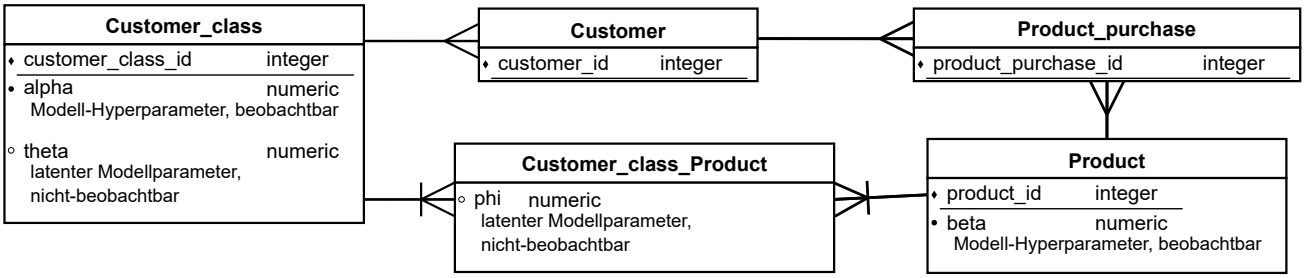


Abbildung 4.3.: Relationales Datenbankschema für den Naïve-Bayes-Klassifikator als Entity-Relationship-Diagramm

Integriertes Modell für den Onlineshop-Kundenklassifikator Beide Schemata – das originale Datenschema (Abb. 4.1) und das aus dem Bayes'schen Netz übersetzte Datenbankschema für das statistische Modell (Abb. 4.3) lässt man miteinander verschmelzen. Dadurch erhält man das integrierte Schema – die Schnittstelle sowohl für Ein- als auch für Ausgabe der Bayes'schen Inferenz. Sowohl die Shop-Datenbank- als auch die modellspezifischen Entitäten finden an der Stelle in einer gemeinsamen, relational gestalteten Datenmanagement-Abstraktion zusammen. Das integrierte relationale Schema für Naïve Bayes im Onlineshop-DB-Kontext ist in 4.4 abgebildet.

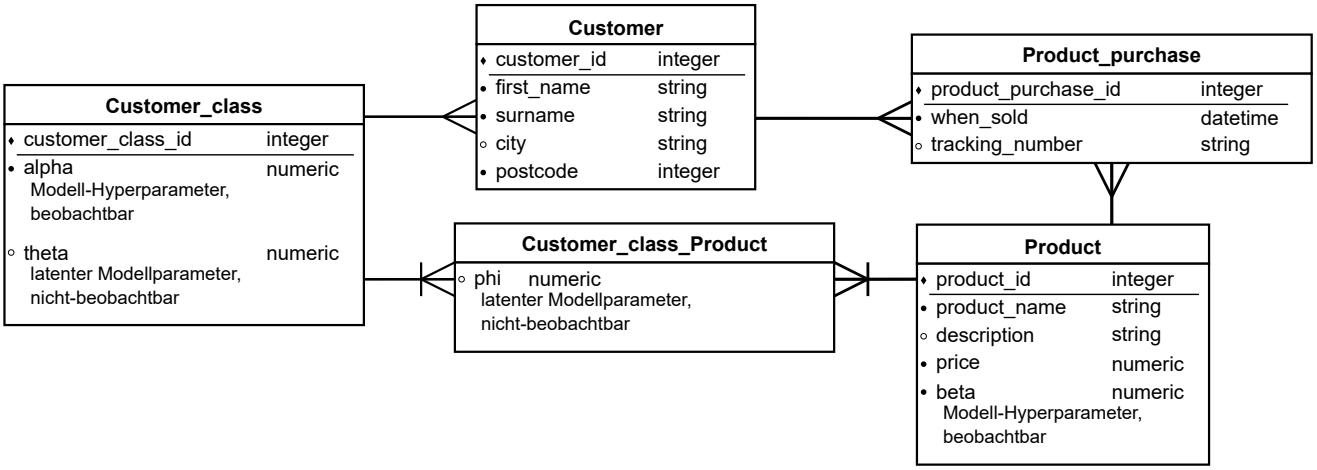


Abbildung 4.4.: Integriertes Schema als Entity-Relationship-Diagramm: hier wurden die Entitäten aus der Onlineshop-Datenbank mit denen des Naïve-Bayes-Klassifikators zusammengeführt.

Klassifikatorvarianten: supervised und unsupervised

Der bisher behandelte Klassifikator war eine bestimmte Variante von Naïve Bayes, nämlich supervised Naïve Bayes. Sie zeichnet sich dadurch aus, dass man im Voraus weiss, zu welchen Kundenklassen die Bestandskunden gehören. Man beobachtet sozusagen die Klassen von Bestandskunden, dementsprechend ist die Modellvariable $\overrightarrow{z_m}$ beobachtet (siehe Bayes'sches Netz in der Abb. 4.2.

Das probabilistischen graphischen Modell zu einer anderen Naïve-Bayes-Variante – unsupervised Naïve Bayes (siehe Bayes'sches Netz in der Abb. 4.5) – sieht strukturell genauso aus. Der einzige Unterschied zu supervised Naïve Bayes (Bayes'sches Netz in der Abb. 4.2) besteht darin, dass der BN-Knoten $\overrightarrow{z_m}$ nicht ausgegraut ist. Diese minimale Abwandlung ist so zu interpretieren, dass man beim Modelltraining nicht weiss, welcher Kundenklasse ein Kunde angehört. Damit der Unterschied zwischen supervised und unsupervised etwas klarer wird: vor dem Modelltraining kann etwas manuelle Vorarbeit geleistet werden, und zwar im Sinne von Kundenlabeling durch den Shopbetreiber. Der Shopbetreiber kann sich nämlich im Vorfeld die Bestellungen von einer Anzahl an Bestandskunden anschauen und diesen Kunden manuell Klassenlabels vergeben (die supervised-Variante). Diese Labeling-Vorarbeit kann mühsam oder teuer sein (z.B. manuell 100 Kunden Klassen zuweisen möchte nicht jeder), sie kann aber andererseits auch weggelassen werden – dann spricht man von der unsupervised-Variante. Je nach Anwendung (z.B. Analyse der Verkäufe anhand von populärsten Kundenklassen) könnte der Shopbetreiber nach dem Lernen eines unsupervised-Klassifikators auf eigene Faust die Klassen erkunden, um herauszufinden, wie die ermittelten Kundenklassen interpretiert werden sollen. Dazu würde er sich nach dem Modelltraining anschauen, welche Produkte die Klassenzuweisung am stärksten prägen, und daraus seine Schlüsse ziehen. Als Beispiel: Laufschuhe würden die Artikelverteilung der Laufsport-Klasse vermutlich so ziemlich dominieren, daher wüsste man die Laufsport-Klasse anhand dieses Wissens zu erkennen.

In SQL2Stan kann man beide Varianten von Naïve Bayes verwenden. Der Unterschied im SQL-Code ist dabei minimal: der Typ einer Spalte wird in der Kundentabellen-Definition geändert (aus einer beobachteten Spalte wird ein latenter diskreter Modellparameter) und dieser Spaltentyp durch zwei kurze SQL-Constraints bekräftigt.

Um beide Varianten zusammenzufassen: bei supervised naïve Bayes sind die Klassenzuweisungen von Kunden bekannt, bei unsupervised nicht. Der Nutzer muss sich für eine Naïve-Bayes-Variante entscheiden. Die Beschreibung der Modellstruktur (log-likelihood) bleibt praktischerweise unberührt davon, welche Daten gegeben sind und welche nicht.

Gedanken zum angewandten Modell-Serving

Interessant ist, dass der SQL-Programmierer bereits anhand der gematchten Modellvariablen sich Gedanken über den Modelleinsatz machen kann. Durch Bayes'sche Inferenz werden die Werte für Klassenprävalenz θ sowie die Produktverteilung in-

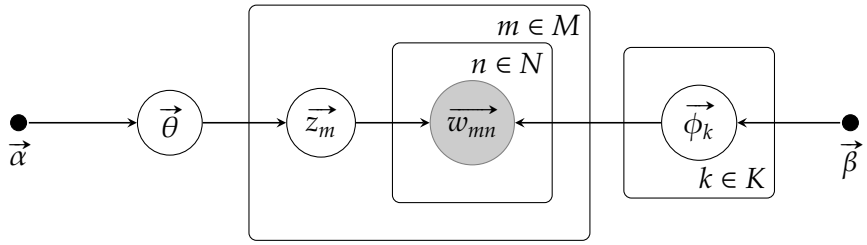


Abbildung 4.5.: Bayes'sches Netz zu unsupervised naïve Bayes (Variable z unbeobachtet)

nerhalb der Klassen ϕ geschätzt. Man kann man sie zum Verfassen monatlicher Newsletter anwenden: die Top-10-Auswahl der Produkte für den Monat beinhalte z.B. einfach jeweils zwei häufigste Produkte (d.h. zwei größte ϕ-Werte für Produkte aus der jeweiligen Klasse) aus den fünf am meisten prävalierenden Kundenklassen (fünf Klassen mit den größten θ-Werten).

Personalisierte Werbung für Seitenbesucher kann man wie folgt umsetzen:

1. Man wartet ab, bis der Shopseitenbesucher sich vier Produkte angeschaut hat (die Zahl vier wurde von mir willkürlich gewählt)

2. Anhand der angeschauten Produkte ($V_{\text{angeschaut}} \in V$ mit V als Menge aller Onlineshopartikel) wird der Besucher mit Naïve Bayes klassifiziert:

   - Nehme für den zu klassifizierenden Kunden diejenige Klasse $k \in K$, für die seine Produktauswahl am plausibelsten erscheint:
     $f(v \in V_{\text{angeschaut}}) = k \in K$ mit $max(\theta_k * \prod_{v \in V_{\text{angeschaut}}} \phi_{kv})$

3. Zeige dem Besucher die 10 häufigsten Produkte (geordnet nach ϕ-Werten) aus seiner Klasse.

4. Neuklassifikation bei neuen Erkenntnissen: die Schritte 2 und 4 sind jedes Mal neu durchzulaufen, sobald der gleiche Besucher (zu erkennen an gleicher Browser-Session-ID) sich zwei weitere neue Produkte anschaut.

4.1.2. Datenbankschema mit inferenzspezifischen Details

In SQL ließen sich die Entitäten aus dem integrierten Datenschema am besten in der Notation von Create-Table-Anweisungen (in SQL) ausdrücken. Manche Details aus dem Kapitel 3 (Konzept von SQL2Stan) werden wiederholt, damit der Lesefluss durch das Zurückblättern nicht gestört werden muss.

Die Ein- und Ausgabe der Bayes'schen Inferenz werden im integrierten Schema über Spaltentypen übermittelt.

Spaltentypen für Inferenzeingabe (beobachtete Daten) Die Spalten mit Werten für beobachtete Modellvariablen sind vom Spaltentyp „stan_data_real“ (*Stan* wie der Name der Inferenzsoftware, *data* steht für beobachtete Daten, *real* impliziert den numerischen Typ für reelle Zahlen).

Auch die „integer“-Spalten gehören zu beobachteten Daten. Für alle Dimensionen im Modell muss man konkrete Größen haben: im Falle von z.B. Kundenklassen muss man wissen, wie viele es davon gibt. Die „integer“-Spalten müssen bei SQL2Stan daher mit Identifikatorwerten gefüllt sein, denn man nutzt diese Werte zur Referenzierung.

Spaltentypen für Inferenzausgabe (latente Modellparameter) Die Spalten, in denen nach der automatischen Inferenz die geschätzten Werte für Modellparameter landen sollen, haben den Spaltentyp „stan_parameter_simplex“: falls sie besonderen Einschänkungen unterliegen (typischerweise wegen der sogenannten Simplex-Bedingung: die Werte müssen sich auf eine bestimmte Art zu einer 1 aufsummieren lassen), lautet der Typname vielmehr „stan_parameter_simplex_constrained“ und impliziert somit das Vorhandensein spezieller Constraints, welche diese besonderen Einschränkungen zum Ausdruck bringen.

Spaltentypen für Modellentitäten, die weder beobachtet noch schätzbar sind Der Spaltentyp „stan_parameter_int“ ist den Spalten mit diskreten Modellparametern vorbehalten. Statt diskret kann man einfach sagen: ganzzahlig, inhärent vom Typ integer. Solche ganzzahligen Modellvariablen-Werte werden nicht geschätzt; sie dienen vielmehr dazu, die Referenzen zwischen den Tabellen (Fremdschlüssel) als unbekannt zu markieren. Ein gutes Beispiel: die Klassen der Onlineshop-Kunden. Wenn man den Kunden vor dem Modelltraining keine Klassen manuell zuweist, dann ist die Zuweisung Kundenklasse–Kunde (die Spalte Customer_class_id in der Tabelle Customer) ein diskreter Modellparameter.

Anmerkungen zum folgenden integrierten Schema Weiter im Abschnitt folgen die Tabellendefinitionen für das supervised-naïve-Bayes-Modell, angepasst an die Entitätennamen einer für das Fallbeispiel ausgedachten Onlineshop-Datenbank; die Spalten für Namen der Kunden, Produkte oder Klassen wurden zwecks Übersichtlichkeit weggelassen.

Tabelle *customer_class* (Listing 4.1)

Diese Datenbanktabelle soll die Information über Kundenklassen beinhalten. Jede Klasse besitzt ihre eigene ID. Die Spalte *alpha* beinhaltet gegebene Hyperparameter-Werte für die Prior-Verteilung der Klassen. Die Spalte *theta* drückt die Klassenprävalenz aus.

Folgende Simplex-Beschränkung für die Spalte *theta* findet Anwendung in den SQL-Constraints: $\theta \in (0,1)^K,\ \sum_K \theta_k = 1$

Listing 4.1: Tabelle *customer_class* aus dem integrierten relationalen Schema für den Naïve-Bayes-Klassifikator für eine Onlineshop-Datenbank.

```
CREATE TABLE customer_class (
        customer_class_id   integer not null, -- k
        alpha   stan_data_real,
        theta   stan_parameter_simplex_constrained,
        PRIMARY KEY (customer_class_id),

        CONSTRAINT
           stan_SimplexConstraint__data_sums_to_one_in_col
           CHECK (argument_column('theta')),
        CONSTRAINT
           stan_SimplexConstraint__when_summed_over_col
           CHECK (argument_column('customer_class_id')),

        CONSTRAINT SortingOrder_Position_1
           CHECK (argument_column('customer_class_id'))
);
```

Tabelle *product* (Listing 4.2)

In dieser Tabelle werden die Shopartikel verwaltet. Jedes Produkt besitzt seine eigene ID. Die Spalte *beta* beinhaltet gegebene Hyperparameter-Werte für die Prior-Verteilung der Produkte.

Listing 4.2: Tabelle *product* aus dem integrierten relationalen Schema für den Naïve-Bayes-Klassifikator für eine Onlineshop-Datenbank.

```
CREATE TABLE product (
        product_id   integer not null, -- v
        beta   stan_data_real,
        PRIMARY KEY (product_id),

        CONSTRAINT SortingOrder_Position_1
           CHECK (argument_column('product_id'))
);
```

Tabelle *customer* im Falle von supervised naïve Bayes (Listing 4.3)

Die Tabelle mit Kundeninformation. Jeder Kunde wird mit einer ID in der Spalte eindeutig identifiziert. In diesem Falle wird als Klassifikator supervised naïve Bayes gewählt: also ist es davon auszugehen, dass man durch manuelle Labeling-Vorarbeit einem Teil der Kunden Klassen zugewiesen hat; siehe Fremdschlüssel-Spalte *customer_class_id*.

Listing 4.3: Tabelle *customer* aus dem integrierten relationalen Schema für den supervised-Naïve-Bayes-Klassifikator für eine Onlineshop-Datenbank.

```
CREATE TABLE customer (
        customer_id integer not null, -- m
        customer_class_id  integer  REFERENCES
           customer_class,
         -- z[m] beobachtet, d.h. supervised naive Bayes
        PRIMARY KEY (customer_id),

        CONSTRAINT SortingOrder_Position_1
           CHECK (argument_column('customer_id'))
);
```

Tabelle *customer* im Falle von unsupervised naïve Bayes (Listing 4.4)

Die Tabelle mit Kundeninformation. Jeder Kunde wird mit einer ID in der Spalte eindeutig identifiziert. Die Art des Klassifikators ist diesmal jedoch unsupervised naïve Bayes. Ohne manuelle Labeling-Vorarbeit im Bezug auf die Zuweisung von Klassen zu Kunden sind die Klassen der Kunden unbekannt/unbeobachtet. Dieser Umstand führt dazu, dass die Fremdschlüssel-Referenz auf die Kundenklassen-Tabelle als unbekannt markiert werden muss. Daher die Änderung des Typs der Spalte *customer_class_id* : der neue Spaltentyp weist nun darauf hin, dass diese Spalte ein diskreter Parameter ist. Man achte auf den geringen Unterschied zwischen un- und supervised-Klassifikatorspezifikation: würde man den Klassifikator nicht-deklarativ spezifizieren, würde diese Feinheit in deutlichen Implementierungsunterschieden münden (wie z.B. in Einführung von gamma-Hilfsarrays).

Listing 4.4: Tabelle *customer* aus dem integrierten relationalen Schema für den unsupervised-Naïve-Bayes-Klassifikator für eine Onlineshop-Datenbank.

```
CREATE TABLE customer (
        customer_id integer not null, -- m
        customer_class_id       stan_parameter_int,
         -- z[m] unbeobachtet, d.h. unsupervised naive Bayes
        PRIMARY KEY (customer_id),

```

```
7           CONSTRAINT SortingOrder_Position_1 CHECK (
               argument_column('customer_id')),
8           CONSTRAINT
               stan_Constraint__int_parameter_independent CHECK
                (true),
9           CONSTRAINT
               stan_Constraint__int_parameter_refers_to_table
10             CHECK (argument_table('customer_class'))
11  );
```

Tabelle *product_purchase* (Listing 4.5)

Die Tabelle, in der die Bestellungen gespeichert werden. Für das Modell ist es nur wichtig zu wissen, welcher Kunde (*customer_id*) welche Produkte (*product_id*) gekauft hat. Jeder Kauf (als Zuordnung Kunde–Artikel) besitzt seine eigene ID.

Listing 4.5: Tabelle *product_purchase* aus dem integrierten relationalen Schema für den Naïve-Bayes-Klassifikator für eine Onlineshop-Datenbank.

```
1   CREATE TABLE product_purchase (
2           product_purchase_id  integer not null, -- n
3           customer_id integer REFERENCES customer, -- m_n
4           product_id  integer REFERENCES product, -- w[n]
5           PRIMARY KEY (product_purchase_id),
6
7           CONSTRAINT SortingOrder_Position_1
8              CHECK (argument_column('product_purchase_id'))
9   );
```

Tabelle *customer_class_product* (Listing 4.6)

Diese Tabelle befasst sich allein damit, die Popularität/Kaufhäufigkeit der Produkte innerhalb einer Kundenklasse zu beschreiben – oder, mit anderen Worten – mit Produktverteilungen für Kundenlassen. Die Spalte *phi* drückt die Prävalenz der Artikel innerhalb einer Kundenklasse aus.

Folgende Simplex-Beschränkung für die Spalte *phi* findet Anwendung in den SQL-Constraints: $\phi \in (0,1)^{K \times V},\ \forall k \in K : \sum_V \phi_{kv} = 1$

Listing 4.6: Tabelle *customer_class_product* aus dem integrierten relationalen Schema für den Naïve-Bayes-Klassifikator für eine Onlineshop-Datenbank.

```
1   CREATE TABLE customer_class_product (
2           customer_class_id   integer
3                   REFERENCES customer_class, -- k
4           product_id   integer     REFERENCES product,  -- v
```

```
5           phi   stan_parameter_simplex_constrained,
6           PRIMARY KEY (customer_class_id, product_id),
7
8           CONSTRAINT
               stan_SimplexConstraint__data_sums_to_one_in_col
               CHECK (argument_column('phi')),
9           CONSTRAINT
               stan_SimplexConstraint__when_summed_over_col
               CHECK (argument_column('product_id')),
10          CONSTRAINT
               stan_SimplexConstraint__when_grouped_by_col
               CHECK (argument_column('customer_class_id')),
11
12          CONSTRAINT SortingOrder_Position_1
13             CHECK (argument_column('customer_class_id')),
14          CONSTRAINT SortingOrder_Position_2
15             CHECK (argument_column('product_id'))
16 );
```

4.1.3. Zielfunktion: mathematische Beschreibung

In großen relevanten Teilen von Machine Learning werden Zielfunktionen als Abstraktion für ML-Algorithmen genommen. Die Optimierung einer Zielfunktion ist das Ziel der automatischen Inferenz, die die Schätzwerte für Modellparameter liefert. Die Zielfunktion ist in unserem Kontext eine Wahrscheinlichkeitsfunktion, und die Optimierung der Zielfunktion steht für die Suche nach den Maxima der Wahrscheinlichkeitsfunktion.

In unserem Fallbeispiel wird ein Bayes'sches Modell eingesetzt; auch Bayes'sche Modelle verwenden Zielfunktionen zur Algorithmenabstraktion. Bei einem Bayes'schen Modell dient die (log-)likelihood-Funktion als Zielfunktion. Die algorithmische Implementierung der Bayes'schen Inferenz beschäftigt sich mit der automatischen Optimierung von dieser Zielfunktion. Je wahrscheinlicher es ist, dass eine Wertebelegung einer durch die Zielfunktion vorgegebenen Verteilung entspringt, desto näher die der Funktionswert am Optimum (der Zielfunktion). Die Suche nach einer Modellvariablenbelegung am Optimum der modellspezifischen Zielfunktion erfolgt in unserem Falle durch die Software Stan [Sta19a] mit Hilfe eines stichprobenbasierten Markov-Chain-Monte-Carlo-Verfahrens oder alternativ über ein Variational-Inference-Verfahren.

Für Naïve Bayes entspricht die likelihood-Funktion der Formel 4.1:

$$
\begin{aligned}
p(w, z, \phi, \theta|\alpha, \beta) = p(\theta|\alpha) & \\
& \cdot \prod_{m \in M} p(z_m|\theta) \\
& \cdot \prod_{k \in K} p(\phi_k|\beta) \\
& \cdot \prod_{n \in N} p(w_n|\phi, z)
\end{aligned}
\tag{4.1}
$$

An den zuvor gesehenen Abbildungen von Bayes'schen Netzen für Naïve Bayes 4.2 (supervised-Variante) und 4.5 (unsupervised-Variante) erkennt man, dass jeder Produktfaktor der Zielfunktion einem durch eine eingekende Kante bedingten Knoten im Bayes'schen Netz entspricht.

Die Wahrscheinlichkeiten, die in einer solchen Zielfunktion multiplizieren würde, sind in der Regel sehr klein. Würde man die Optimierung der Zielfunktion direkt am Produkt vornehmen wollen, wäre man sofort mit numerischen Problemen konfrontiert (v.a. mit Überläufen). Hilfreich ist hierbei ein einfacher Trick: man logarithmiert die likelihood-Funktion. Die Optima der log-likelihood-Funktion sind die gleichen wie bei der nicht-logarithmierten likelihood-Funktion. Numerisch lässt sich die logarithmierte Zielfunktion aber viel besser handhaben, denn aus den Multiplikationen werden Summen. Die logarithmierte likelihood-Funktion für Naïve Bayes entspricht die log-likelihood-Funktion in der Formel 4.2:

$$
\begin{aligned}
\lg\, p(w, z, \phi, \theta|\alpha, \beta) = \lg\, p(\theta|\alpha) + & \\
& + \sum_{m \in M} \lg\, p(z_m|\theta) \\
& + \sum_{k \in K} \lg\, p(\phi_k|\beta) \\
& + \sum_{n \in N} \lg\, p(w_n|\phi, z)
\end{aligned}
\tag{4.2}
$$

Die in der Formel 4.2 dargestellte log-likelihood-Funktion eignet sich fast zu probabilistischen Modellierung; es fehlen nur noch konkrete Verteilungsnamen anstelle von p-Platzhaltern. Die Wahrscheinlichkeitsfunktionen p, die man in jedem der Summanden der Zielfunktion sieht, sind in Wahrheit Verteilungen – so wie die gerichteten Kanten in den Bayes'schen Netzen. Im folgenden Schritt werden gleich zwei wesentliche Änderungen geschehen. Zum Einen wird aufgedeckt, welche Verteilungen sich hinter den p-Buchstaben in den Summanden verstecken. Zum Anderen wird die Verteilung aus dem letzten Summanden $p(w_n|\phi, z)$ explizit ausgedrückt. Das Ergebnis dieser detaileinbringender Änderungen – die vervollständigte

log-likelihood- bzw. Zielfunktion für Naïve Bayes – ist in der Formel 4.3 zu sehen.

$$\begin{aligned} \lg p(w, z, \phi, \theta | \alpha, \beta) = \lg \mathit{Dirichlet}(\theta|\alpha) + \\ + \sum_{m \in M} \lg \mathit{Categorical}(z_m|\theta) \\ + \sum_{k \in K} \lg \mathit{Dirichlet}(\phi_k|\beta) \\ + \sum_{n \in N} \lg \mathit{Categorical}(w_n|\phi_{z_{m_n}}) \end{aligned} \tag{4.3}$$

Kategorische Verteilung kurz erklärt Die kategorische Verteilung *Categorical* ist leicht zu verstehen. Ich werde es anhand der Wahrscheinlichkeit $\mathit{Categorical}(a|b)$ erklären. Man habe eine Prior-Verteilung b: zum Beispiel, eine Kundenklassen-Verteilung, die festlegt, dass die Häufigkeit für Klasse 1 rund 0.8 beträgt, und die Häufigkeit des Auftretens für Klasse 2 erfahrungsgemäß 0.2 beträgt. Der Posterior a sei eine Klasse: entweder Klasse 1 oder 2. Die Wahrscheinlichkeit $\mathit{Categorical}(a|b)$ ist somit die Wahrscheinlichkeit, dass man die Kundenklasse a aus der Verteilung b zieht.

Dem aufmerksamen Leser bleibe ich noch einer Erklärung schuldig: wie wurde aus $\lg p(w_n|\phi, z)$ (Formel 4.2) plötzlich $\mathit{Categorical}(w_n|\phi_{z_{m_n}})$ (Formel 4.3)? p ist hierbei kontextuell (aus dem Modellverständnis heraus) eine kategorische Verteilung, daher der Verteilungsfunktionsbezeichner *Categorical*. Deutlich spannender zeigt sich der Prior-Teil $\phi_{z_{m_n}}$. Diese Schachtelung kann man nicht ohne weiteres aus dem Bayes'schen Netz ablesen: im BN sieht man ja schließlich nur zwei in den w-Knoten eingehende Kanten. Der Prior-Teil $\phi_{z_{m_n}}$ bildet sich vielmehr aus dem Modellverständnis heraus. In Bayes'schen Netz lässt sich Folgendes ablesen: der Produktkauf n (mit w_{mn} als Produkt-ID des gekauften Artikels) wird bedingt durch die Produktbeliebtheit einer Kundenklasse ϕ_k sowie die Klassenzugehörigkeit z eines Kunden m, der diesen Produktkauf getätigt hat. Nun aber zum Modellverständnis: wie kann man einen Produktkauf, eine Produktbeliebtheits-Verteilung für eine Kundenlasse sowie eine Kunden-Klassen-Zuweisung unter ein gemeinsames Dach bringen? Die Antwort lautet: zuerst den Kunden finden, der den n-ten Kauf getätigt hat (d.h. m_n); dann seine Kundenklasse finden (z_{m_n}) und anschließend die Produktbeliebtheit für die gefundene Kundenklasse nachschlagen ($\phi_{z_{m_n}}$). Die Wahrscheinlichkeit, dass ein Produkt w_n beim Kauf n bestellt wurde, lässt sich als Wahrscheinlichkeit $\mathit{Categorical}(w_n|\phi_{z_{m_n}})$ modellieren.

Dirichlet-Verteilung kurz erklärt Die *Dirichlet*-Verteilung steht für eine Verteilung von Verteilungen. Mit bildreicheren Worten: man habe – ähnlich wie beim Zettelziehen aus einem Hut – einen Hut, gefüllt jedoch nicht mit Zetteln, sondern mit Verteilungen. Man kann aus dem Hut zufällig eine Verteilung ziehen und sie wieder zurücklegen. Der Hut ist, mit anderen Worten, eine Verteilung von Verteilungen: davon hängt nämlich ab, wie wahrscheinlich es ist, die eine oder andere Verteilung

zu ziehen. Die Wahrscheinlichkeit $Dirichlet(c|d)$ ist also die Wahrscheinlichkeit, dass man die Verteilung c aus der Verteilung d zieht.

4.1.4. Zielfunktion: SQL-Beschreibung

Die log-likelihood-Funktion für Naïve Bayes 4.3 beschreibt die Struktur von diesem Bayes'schen Modell. Es besteht aus Additionen, Summen, Verteilungsfunktionsaufrufen und Unterabfragen. Somit eignet sie sich sehr gut, um im SQL2Stan-Dialekt beschrieben zu werden.

Im diesem Kapitel werden die einzelnen Summanden der Naïve-Bayes-Zielfunktion als SQL-Code präsentiert, Summand nach Summand (d.h. ein SQL-View pro Summand).

Ich weise an der Stelle darauf hin, dass komplette SQL-Beschreibungen von allen in dieser Arbeit erläuterten Bayes'schen Modellen im „example_models“ Ordner des SQL2Stan-Prototypen liegen.

Erster Summand lg $Dirichlet(\theta|\alpha)$ (Listing 4.7)

Listing 4.7: Erster Summand der log-likelihood-Funktion für Naïve Bayes 4.3

```
with "posterior__theta" as (
  select
     theta,
     customer_class_id
  from
     customer_class
),
"prior__alpha" as (
  select
     alpha,
     customer_class_id
  from
     customer_class
)
select
  dirichlet_lpdf(
     ARRAY(
         select
            theta
         from
            posterior__theta
         order by
            customer_class_id
     ),
     ARRAY(
```

```
26            select
27               alpha
28            from
29               prior__alpha
30            order by
31               customer_class_id
32         )
33      )
```

Zweiter Summand $\sum_{m \in M} \lg Categorical(z_m|\theta)$ (Listing 4.8)

Listing 4.8: Zweiter Summand der log-likelihood-Funktion für Naïve Bayes 4.3

```
1  with "log cat(z_m|theta)" as (
2    with "posterior__z_m" as (
3       select
4              customer_class_id,
5              customer_id
6       from
7              customer
8    ),
9    "prior__theta" as (
10      select
11             theta,
12             customer_class_id
13      from
14             customer_class
15   )
16   select
17      LoopIndex.customer_id,
18      categorical_lpmf(
19             ARRAY(
20                    select
21                           customer_class_id
22                    from
23                           posterior__z_m
24                    where
25                           customer_id = LoopIndex.customer_id
26                    order by
27                           customer_id
28             ),
29             ARRAY(
30                    select
31                           theta
```

```
32                    from
33                         prior__theta
34                    order by
35                         customer_class_id
36              )
37       ) as categorical_lpmf
38    from
39       posterior__z_m as LoopIndex
40 )
41 select
42       sum(categorical_lpmf) as "log p(z|theta)"
43 from
44       "log cat(z_m|theta)"
```

Dritter Summand $\sum_{k \in K} \lg Dirichlet(\phi_k|\beta)$ (Listing 4.9)

Listing 4.9: Dritter Summand der log-likelihood-Funktion für Naïve Bayes 4.3

```
1  with "log dir(phi|beta)" as (
2    with "posterior__phi" as (
3       select
4              phi,
5              customer_class_id,
6              product_id
7       from
8              customer_class_product
9    ),
10   "prior__beta" as (
11      select
12             beta,
13             product_id
14      from
15             product
16   ),
17   "loop_over_customer_class_ids" as (
18      select
19             customer_class_id
20      from
21             customer_class
22   )
23   select
24      LoopIndex.customer_class_id,
25      dirichlet_lpdf(
26             ARRAY(
```

```
27                      select
28                           phi
29                      from
30                           posterior__phi
31                      where
32                           customer_class_id = LoopIndex.
                                customer_class_id
33                      order by
34                           customer_class_id,
35                           product_id
36              ),
37
38
39              ARRAY(
40                      select
41                           beta
42                      from
43                           prior__beta
44                      order by
45                           product_id
46              )
47    ) as dirichlet_lpdf
48    from
49       loop_over_customer_class_ids as LoopIndex
50 )
51 select
52       sum(dirichlet_lpdf) as "log p(phi|beta)"
53 from
54       "log dir(phi|beta)"
```

Vierter Summand $\sum_{n \in N} \lg Categorical(w_n|\phi_{z_{m_n}})$ (Listing 4.10)

Listing 4.10: Vierter Summand der log-likelihood-Funktion für Naïve Bayes 4.3

```
1  with "log cat(w_n|phi_z_m_n)" as (
2    with "posterior__w_n" as (
3       select
4          product_id,
5          product_purchase_id
6       from
7          product_purchase
8    ),
9    -- prior-Zusammenstellung:
10   -- product_purchase_id = n --> customer_id = m
```

```
11                --> customer_class_id = z --> phi
12    "prior__m_n" as (
13       select
14          product_purchase_id,
15          customer_id
16       from
17          product_purchase
18    ),
19    "prior__z_m" as (
20       select
21          customer_id,
22          customer_class_id
23       from
24          customer
25    ),
26    "prior__phi_z" as (
27       select
28          customer_class_id,
29          phi
30       from
31          customer_class_product
32    )
33    select
34       LoopIndex.product_purchase_id,
35       categorical_lpmf(
36          ARRAY(
37             select
38                product_id
39             from
40                posterior__w_n
41             where
42                product_purchase_id = LoopIndex.
                      product_purchase_id
43             order by
44                product_purchase_id
45          ),
46          ARRAY(
47             select
48                phi
49             from
50                prior__phi_z
51             where
52                customer_class_id in (
53                   select
```

```
54                      customer_class_id
55                   from
56                      prior__z_m
57                   where
58                      customer_id in (
59                         select
60                            customer_id
61                         from
62                            prior__m_n
63                         where
64                            product_purchase_id = LoopIndex.
                                 product_purchase_id
65                         order by
66                            product_purchase_id
67                      )
68                   order by
69                      customer_id
70                )
71             order by
72                customer_class_id,
73                product_id
74          )
75       ) as categorical_lpmf
76    from
77       posterior__w_n as LoopIndex
78 )
79 select
80    sum(categorical_lpmf) as "log p(w|phi)"
81 from
82    "log cat(w_n|phi_z_m_n)"
```

Addition von Summanden

Bevor wir zu der Zusammenführung der oben gezeigten ZF-Summanden durch Addition kommen, noch eine kleine Anmerkung zu dem SQL-Code der Summanden. Die Funktionen für Dirichlet- und kategorische Verteilung nehmen Arrays als Funktionsparameter entgegen, weil zumindest bei PostgreSQL (dem DBMS meiner Wahl im Rahmen dieser Arbeit) es nicht möglich sein würde, bloße SQL-Unterabfragen als Parameter einer User-Defined-Function einzusetzen. Die Verteilungsfunktionen werden in SQL2Stan einfach als Dummy-Funktionen gehandhabt – sie sind zwar definiert, implementieren aber nichts sinnvolles. Das liegt daran, dass SQL2Stan den benutzergeschriebenen SQL-Code nur compiliert und auswertet; der SQL-Code wird nicht ausgeführt.

Die log-likelihood- bzw. Zielfunktion für Naïve Bayes 4.3 setzt sich aus als Summe aus den oben beschriebenen Summandenviews zusammen. Die entsprechende SQL-

Umsetzung davon steht im Listing 4.11.

Das probabilistische SQL2Stan-Programm für das Onlineshop-Kundenklassifikator-Fallbeispiel setzt sich also aus dem SQL-Code für ZF-Summanden aus den Listings 4.7, 4.8, 4.9, 4.10 sowie ihrer zusammenführung im Listing 4.11.

Listing 4.11: log-likelihood-Funktion für Naïve Bayes als eine Summe, genau wie ihre mathematische Beschreibung in der Formel 4.3

```
SELECT
  "log p(theta|alpha)" +
  "log p(mu|beta)" +
  "log p(z|theta)" +
  "log p(d|z)"
  as "log p(d,z,mu,theta|alpha,beta)"
FROM
  "log p(theta|alpha)",
  "log p(mu|beta)",
  "log p(z|theta)",
  "log p(d|z)";
```

Das Bayes'sche Modell wäre somit vollständig im Datenbankkontext beschrieben und direkt mit den in der Datenbank gelagerten Daten in Verbindung gebracht. Der automatischen Inferenz (ergo dem Lernen des Kundenklassifikators) steht nun nichts mehr im Wege; vom Nutzer wird keine Programmiertätigkeit mehr erwartet.

4.2. Generierung einer Implementierung für automatische Inferenz

Aus den Tabellenspalten der Datenbank werden Datenstrukturen für die automatische Inferenz vorbereitet, und aus der SQL-Beschreibung der Zielfunktion wird eine Inferenzimplementierung generiert. SQL wird hierbei in die probabilistische Programmiersprache von Stan [Sta19a] übersetzt (daher auch der Projektname SQL2Stan).

Stan als PPL ist eine high-level-Programmiersprache: man sieht im Stan-Code nicht, wie die konkreten Inferenzalgorithmen implementiert sind. Vielmehr übersetzt die Software Stan ein probabilistisches Programm (den Stan-Code) in einen C++-Code, wo die Inferenzprozesse schlussendlich konkret implementiert werden. Auf der C++-Ebene nutzt Stan C++-Bibliotheken wie Boost, Eigen und Sundials. Das bei SQL2Stan vorgesehene mehrfache Übersetzung vom nutzergeschriebenen SQL-Code (SQL nach Stan nach C++) ist darauf bedacht, dass der Code in jedem Übersetzungsschritt in einer weniger abstrakte und implementierungsnähere Programmiersprache erfolgt. Es hat den Vorteil, dass an jedem Übersetzungsschritt automatische Codeoptimierungen möglich sind. Die Komplexität der Erzeugung vom effizienten Code wird somit vom SQL-Programmierer auf die Compiler (SQL-nach-Stan- sowie Stan-nach-C++-Compiler) übertragen.

Der vom SQL2Stan-Compiler generierte Stan-Code orientiert sich stark an den Aufbau des Naïve-Bayes-Implementierung [Car13d; Car13c] (siehe Stan-Code im den Anhangskapiteln D.2 und D.1), die von Bob Carpenter geschrieben wurde, einem der Core-Entwickler aus dem Stan-Entwicklerteam. Es ist wichtig hervorzuheben, dass keines der Bayes'schen Modelle in SQL2Stan fest hineincodiert ist. Beim Entwurf der Übersetzungskonzepte von SQL nach Stan mussten gängige Programmierungskonzepte aus den vorhandenen, professionell programmierten Stan-Beispielen extrahiert, verallgemeinert und als Übersetzungsmuster bereitgestellt werden. Es ging also um die Analyse, wie ein geschickter Stan-Programmierer den Code schreiben würde, um diese Erkenntnisse auf andere Beispiele zu übertragen (z.B. Latent Dirichlet Allocation von David M. Blei u.a. [BNJ03]).

In folgenden Unterkapiteln wird der aus der deklarativen SQL-Modellspezifikation generierte Stan-Code vorgestellt, der die automatische Inferenz für den naïve-Bayes-Klassifikator am Fallbeispiel imperativ implementiert. Der Leser kann Vergleiche zwischen dem SQL2Stan-generierten Code und handgeschriebener Implementierung von supervised naïve Bayes (Anhangskapitel D.1) sowie unsupervised naïve Bayes (Anhangskapitel D.2) ziehen, umd letztendlich zu erkennen, dass die Ähnlichkeiten im Code groß sind – die Codestruktur ist gleich, und die Unterschiede liegen an syntaktischen Synonymen (z.B. Funktionsaufrufe in unterschiedlichen Notationen).

4.2.1. Supervised Naïve Bayes

Der SQL2Stan-generierte Stan-Code für die supervised-Variante des Naïve-Bayes-Klassifikators ist im Listing 4.12 zu sehen. Zum Vergleich stünde der Stan-Code von Bob Carpenter im Anhangskapitel D.1. In der Definition der Tabelle „customer" wurde die Fremdschlüssel-Spalte *customer_class_id* nicht als unbekannt markiert. Der SQL2Stan-Compiler geht daher davon aus, dass diese Spalte beobachtete Daten enthält; somit enthält das probabilistische Modell keine latenten diskreten Modellparameter.

Listing 4.12: Stan-Implementierung für supervised Naïve Bayes, SQL2Stan-generiert

```
data {
int Total__ofTable__customer_class_
      _asNumberOfDistinct__customer_class_id;
int Total__ofTable__product_
      _asNumberOfDistinct__product_id;
int Total__ofTable__customer_
      _asNumberOfDistinct__customer_id;
int Total__ofTable__product_purchase_
      _asNumberOfDistinct__product_purchase_id;
int Total__ofTable__customer_class_product_
      _asNumberOfDistinct__customer_class_id;
int Total__ofTable__customer_class_product_
      _asNumberOfDistinct__product_id;
int customer_class_id__asColumnOfTable__customer_class[
```

```
      Total__ofTable__customer_class_
            _asNumberOfDistinct__customer_class_id];
int product_id__asColumnOfTable__product[
      Total__ofTable__product_
            _asNumberOfDistinct__product_id];
int customer_id__asColumnOfTable__customer[
      Total__ofTable__customer_
            _asNumberOfDistinct__customer_id];
int customer_class_id__asColumnOfTable__customer[
      Total__ofTable__customer_
            _asNumberOfDistinct__customer_id];
int product_purchase_id__asColumnOfTable__product_purchase[
      Total__ofTable__product_purchase_
            _asNumberOfDistinct__product_purchase_id];
int customer_id__asColumnOfTable__product_purchase[
      Total__ofTable__product_purchase_
            _asNumberOfDistinct__product_purchase_id];
int product_id__asColumnOfTable__product_purchase[
      Total__ofTable__product_purchase_
            _asNumberOfDistinct__product_purchase_id];
int customer_class_id__asColumnOfTable__customer_class_product[
      Total__ofTable__customer_class_product_
            _asNumberOfDistinct__customer_class_id,
      Total__ofTable__customer_class_product_
            _asNumberOfDistinct__product_id];
int product_id__asColumnOfTable__customer_class_product[
      Total__ofTable__customer_class_product_
            _asNumberOfDistinct__customer_class_id,
      Total__ofTable__customer_class_product_
            _asNumberOfDistinct__product_id];
vector[
      Total__ofTable__customer_class_
            _asNumberOfDistinct__customer_class_id
      ] alpha__asColumnOfTable__customer_class;
vector[
      Total__ofTable__product_
            _asNumberOfDistinct__product_id
      ] beta__asColumnOfTable__product;
}

parameters {
simplex[
      Total__ofTable__customer_class_
            _asNumberOfDistinct__customer_class_id
      ] theta__asColumnOfTable__customer_class;

simplex[
      Total__ofTable__customer_class_product_
            _asNumberOfDistinct__product_id
```

```
      ] phi__asColumnOfTable__customer_class_product[
            Total__ofTable__customer_class_product_
                  _asNumberOfDistinct__customer_class_id];
}

model {
target += dirichlet_lpdf(
      theta__asColumnOfTable__customer_class
    | alpha__asColumnOfTable__customer_class);

for (a in 1:Total__ofTable__customer_class_
      _asNumberOfDistinct__customer_class_id)
{
target += dirichlet_lpdf(
      phi__asColumnOfTable__customer_class_product[a]
    | beta__asColumnOfTable__product);
}


for (a in 1:Total__ofTable__customer_
      _asNumberOfDistinct__customer_id) {
target += categorical_lpmf(
      customer_class_id__asColumnOfTable__customer[a]
    | theta__asColumnOfTable__customer_class);
}

for (a in 1:Total__ofTable__product_purchase_
      _asNumberOfDistinct__product_purchase_id) {
target += categorical_lpmf(
      product_id__asColumnOfTable__product_purchase[a]
    | phi__asColumnOfTable__customer_class_product
          [customer_class_id__asColumnOfTable__customer
                [customer_id__asColumnOfTable__product_purchase
                      [a]]]);
}
}
```

4.2.2. Unsupervised Naïve Bayes

Die unsupervised-Variante des Fallbeispiel-Klassifikators wird als Stan-Code im Listing 4.13 implementiert. Zum Vergleich stünde der Stan-Code von Bob Carpenter im Anhangskapitel D.2. Hier wurde die Fremdschlüssel-Spalte *customer_class_id* in der Definition der Tabelle „customer“ als unbekannt/unbeobachtet markiert. SQL2Stan geht deshalb davon aus, dass diese Spalte eine latente Referenz enthält, ergo einen latenten diskreten Modellparameter, der von Stan nicht geschätzt werden kann. Dadurch verschwindet eine Zeile aus dem data-Codeblock (da die Kundenklassen nun unbeobachtet sind, und im data-Block stehen nur Einträge für gegebene Modelldaten), und es kommen zusätzliche Zeilen im model-Block hinzu:

ein gamma-Array sowie Schleifenoperationen am gamma-Array. Die Erklärung der hier in Kraft getretenen implementierungstechnischen Feinheiten (ein gamma-Array sowie Operationen an ihm) lässt sich im Kapitel 3.3.5 nachlesen.

Listing 4.13: Stan-Code für unsupervised Naïve Bayes, SQL2Stan-generiert

```
data {

int Total__ofTable__customer_class_
      _asNumberOfDistinct__customer_class_id;
int Total__ofTable__product_
      _asNumberOfDistinct__product_id;
int Total__ofTable__customer_
      _asNumberOfDistinct__customer_id;
int Total__ofTable__product_purchase_
      _asNumberOfDistinct__product_purchase_id;
int Total__ofTable__customer_class_product_
      _asNumberOfDistinct__customer_class_id;
int Total__ofTable__customer_class_product_
      _asNumberOfDistinct__product_id;
int customer_class_id__asColumnOfTable__customer_class[
      Total__ofTable__customer_class_
            _asNumberOfDistinct__customer_class_id];
int product_id__asColumnOfTable__product[
      Total__ofTable__product_
            _asNumberOfDistinct__product_id];
int customer_id__asColumnOfTable__customer[
      Total__ofTable__customer_
            _asNumberOfDistinct__customer_id];
int product_purchase_id__asColumnOfTable__product_purchase[
      Total__ofTable__product_purchase_
            _asNumberOfDistinct__product_purchase_id];
int customer_id__asColumnOfTable__product_purchase[
      Total__ofTable__product_purchase_
            _asNumberOfDistinct__product_purchase_id];
int product_id__asColumnOfTable__product_purchase[
      Total__ofTable__product_purchase_
            _asNumberOfDistinct__product_purchase_id];
int customer_class_id__asColumnOfTable__customer_class_product[
      Total__ofTable__customer_class_product_
            _asNumberOfDistinct__customer_class_id,
      Total__ofTable__customer_class_product_
            _asNumberOfDistinct__product_id];
int product_id__asColumnOfTable__customer_class_product[
      Total__ofTable__customer_class_product_
            _asNumberOfDistinct__customer_class_id,
      Total__ofTable__customer_class_product_
            _asNumberOfDistinct__product_id];
vector[
```

```
      Total__ofTable__customer_class_
            _asNumberOfDistinct__customer_class_id
      ] alpha__asColumnOfTable__customer_class;
vector[
      Total__ofTable__product_
            _asNumberOfDistinct__product_id
      ] beta__asColumnOfTable__product;
}

parameters {
simplex[
      Total__ofTable__customer_class_
            _asNumberOfDistinct__customer_class_id
       ] theta__asColumnOfTable__customer_class;

simplex[
      Total__ofTable__customer_class_product_
            _asNumberOfDistinct__product_id
       ] phi__asColumnOfTable__customer_class_product[
             Total__ofTable__customer_class_product_
                   _asNumberOfDistinct__customer_class_id];
}

model {
real gamma__for__customer_class_id__asColumnOfTable__customer_class[
      Total__ofTable__customer_
            _asNumberOfDistinct__customer_id,
      Total__ofTable__customer_class_
            _asNumberOfDistinct__customer_class_id];

target += dirichlet_lpdf(
      theta__asColumnOfTable__customer_class
    | alpha__asColumnOfTable__customer_class);

for (a in 1:Total__ofTable__customer_class_
      _asNumberOfDistinct__customer_class_id)
{
target += dirichlet_lpdf(
      phi__asColumnOfTable__customer_class_product[a]
    | beta__asColumnOfTable__product);
}

for (a in 1:Total__ofTable__customer_
      _asNumberOfDistinct__customer_id) {
for (b in 1:Total__ofTable__customer_class_
      _asNumberOfDistinct__customer_class_id) {
gamma__for__customer_class_id__asColumnOfTable__customer_class[a,b] =
categorical_lpmf(
      b
```

```
    | theta__asColumnOfTable__customer_class);
}
}

for (a in 1:Total__ofTable__product_purchase_
      _asNumberOfDistinct__product_purchase_id) {
for (b in 1:Total__ofTable__customer_class_
      _asNumberOfDistinct__customer_class_id) {
gamma__for__customer_class_id__asColumnOfTable__customer_class[
      customer_id__asColumnOfTable__product_purchase[a],b] +=
      categorical_lpmf(
            product_id__asColumnOfTable__product_purchase[a]
          | phi__asColumnOfTable__customer_class_product[b]);
}
}

for (a in 1:Total__ofTable__customer_
      _asNumberOfDistinct__customer_id) {
target += log_sum_exp(
      gamma__for__customer_class_id_
            _asColumnOfTable__customer_class[a]);
}
}
```

4.3. Bewertung der generierten Inferenzimplementierung

Die Bewertung des generierten Stan-Codes erfordert den **Vergleichsgegenstand** (den SQL2Stan-generierten Stan-Code aus den Kapiteln 4.2.1 und 4.2.2) sowie das **Optimum** (den von Bob Carpenter verfassten Stan-Code für gleiche Modelle, siehe Anhangskapitel D.1 und D.2).

Der Leser hat bereits im Kapitel 4.2 erfahren, dass der generierte Code sich stark an sein handgeschriebenes Vorbild orientiert, und daher dem Leistungoptimum für den gewählten Ansatz sehr nahe ist. Somit wird es bei den Vergleichen keine Überraschungen geben.

Zu den Rahmenbedingungen: als Datensatz dienen die mit Hilfe eines Skripts von Bob Carpenter [Car13e] generierte „Onlineshop“-Daten. Die Inferenzsoftware Stan wird im Rahmen der Experimente in der Windows-Version cmdstan-2.17.1 genutzt. Gerechnet wird an einem Windows-10-Laptop mit einem Intel-Core-i7-6700HQ-Prozessor (4 Kerne, 8 Threads) und 12 GB RAM. Die Modelldaten wurden mit folgenden Größen generiert:

- Anzahl der Klassen $K = 6$ (willkürlich festgelegt; man hätte als K z.B. auch $\sqrt{V} = \lceil\sqrt{350}\rceil = 19$ nehmen können.)
- Anzahl der Shopartikel $V = 350$
- Anzahl der Kunden $M = 100$

- Hyperparameter-Vektor $\vec{\alpha}$ (K-groß) = (1,1,1,1,1,1)
- Hyperparameter-Vektor $\vec{\beta}$ (V-groß) = (0.1,0.1,...,0.1)

Wichtig zu wissen: die Schätzungsergebnisse sowie die Laufzeiten lassen sich nicht perfekt vergleichen. Das liegt an dem gewählten Ansatz, nämlich der automatischen Inferenz durch Markov-Chain-Monte-Carlo-gestütztes Samplingverfahren (konkret: No-U-Turn-Sampler [HG14], das Standard-Inferenzverfahren von Stan) [1]. Solche Schätzungverfahren, die im Rahmen der Bayes'schen Inferenz verwendet werden, sind approximativ und nicht-deterministisch. Dabei wird, grob gesagt, eine Verteilung angegeben und es werden viele Stichproben aus dieser Verteilung simuliert (Monte-Carlo-Simulation). Dabei werden die zuvor gezogenen Werte beim Ziehen neuer Stichproben einbezogen, wodurch eine Markov-Kette entsteht. Durch die Natur der näherungsweisen Modellparameterinferenz kann man also bei gleicher Implementierung und gleichen Daten mehrmals die Parameterinferenz starten, um mehrere voneinander etwas abweichende Schätzungsergebnisse zu erhalten bei mehrmals geringfügig unterschiedlichen Algorithmuslaufzeiten.

Die Ergebnisse für naïve-Bayes-Kundenklassifikation fallen im Bezug auf die Schätzungsergebnisse und die Laufzeiten dennoch recht vergleichbar und verwertbar aus (siehe Tabelle 4.1). Zum Belegen, dass beide Stan-Implementierungen eine vergleichbare Ausgabe liefern, wurde übersichtshalber die durch Modellinferenz geschätzte Klassenprävalenz $\vec{\theta}$ ausgewählt, weil sie aus nur sechs Zeilen besteht. Die Schätzungsergebnisse für das Modellparameter $\vec{\phi}$ bestehen aus 6×350 Zeilen; sie liegen der Arbeit zwar bei (im Ordner „Gegenueberstellung von Inferenzergebnissen“) und weisen tatsächlich sehr ähnliche Werte auf, werden aber aus offensichtlichen Platzgründen nicht hier im Text aufgeführt. Rundungsbedingte Artefakte in den Schätzungswerten sind nicht ausgeschlossen.

Die Zeitmessergebnisse beziehen sich nur auf die automatische Modellparameterinferenz. Der SQL2Stan-Prototyp hat aktuell einen wesentlichen Nachteil: da die geschätzten Parameterwerte aus einer textuellen Stan-Ausgabedatei in die Datenbank mit Hilfe von UPDATE-Befehlen eingelesen werden, kommen noch zusätzlich etwa 0,3 Sekunden Bearbeitungszeit pro Parameterwert (d.h. pro zu aktualisierende Tabellenzeile) zur Laufzeit hinzu.

[1] Man hätte auch ADVI (Automatic Differentiation Variational Inference) [Kuc+17] als Inferenzalgorithmus nehmen können. Er läuft deutlich schneller als im Vergleich zum No-U-Turn-Sampler, doch in Stan ist ADVI aktuell nur prototypisch umgesetzt.

Tabelle 4.1.: Gegenüberstellung von ausgewählten Schätzungsergebnissen sowie Laufzeiten: handgeschriebener versus SQL2Stan-generierter Stan-Code. Klassen-ID-Permutationen (bei jedem unsupervised-Inferenzdurchlauf eine andere ID für dieselbe Klasse) berücksichtigt.

	supervised NB		unsupervised NB	
	Carpenter	**SQL2Stan**	**Carpenter**	**SQL2Stan**
t(Warm-up), s	491	488	2520	2435
t(Sampling), s	390	386	1882	1915
t(Gesamt), s	881	875	4402	4350
theta[1]	0.41	0.41	0.4	0.4
theta[2]	0.31	0.31	0.31	0.31
theta[3]	0.095	0.094	0.0097	0.01
theta[4]	0.095	0.093	0.095	0.18
theta[5]	0.047	0.048	0.087	0.087
theta[6]	0.047	0.048	0.098	0.011

4.4. Serving: Anwendung des trainierten Klassifikators

Im Falle von Naïve Bayes (ganz gleich, ob supervised oder nicht) ist es recht einfach, eine Anwendung zu bauen, welche die Shop-Bestandskunden oder Seitenbesucher anhand der mit ihnen in Verbindung stehenden Produkten klassifiziert. Dies lässt sich mit SQL bewerkstelligen (siehe Listing 4.14). Es gibt jedoch eine wichtige Voraussetzung: der Klassifikator muss trainiert sein (d.h. die Inferenz muss durchgelaufen sein, und die inferierten Werte müssen über das relationale Schema abrufbar sein). Ein derartiges Modellserving über SQL ist in vielerlei Datenmanagementsystemen möglich (klassische RDBMS, Apache Spark, Apache Hive, SciDB und andere Systeme mit SQL-basierten deklarativen Abfragesprachen).

Listing 4.14: Klassifikation leicht gemacht: diese Abfrage hat einen Platzhalte $products_of_interest$ für eine kommagetrennte Liste von Produkten, die ein zu klassifizierender Kunde oder Seitenbesucher gekauft oder sich angeschaut hat. Die erste Zeile der Ausgabe dieser SQL-Abfrage ist die Klassen-ID, die dem Kunden bzw. Seitenbesucher zugewiesen werden soll.

```
1 select
2    customer_class_id,
3    sum(log(phi))+log(theta) as log_probability
4 from
5    customer_class_product
6       join
7    customer_class
8       using (customer_class_id)
```

```
9  where
10    product_id in ($products_of_interest$)
11 group by
12    customer_class_id,
13    theta
14 order by
15    log_probability
16 desc;
```

Zuerst ein Paar Worte zu dem, was die SQL-Abfrage im Listing 4.14 beinhaltet. In ihrem WHERE-Teil findet eine Nutzereingabe statt: der Platzhalter $products_of_interest$ steht für eine Liste von Produkt-IDs, die ein unklassifizierter Nutzer entweder gekauft oder sich angeschaut hat. Solch eine Liste kann z.B. so ausschauen: (1,12,42,204,18,7). Für jede von diesen Produkt-IDs und für jede Personenklassen-ID werden die Werte von *phi* nachgeschlagen – die Häufigkeiten des Auftretens eines bestimmten Artikels innerhalb einer bestimmten Kundenklasse. Diese *phi*-Werte werden logarithmiert und aufsummiert; so erhält man klassenweise die logarithmierte Wahrscheinlichkeit dafür, dass diese Produkte für die jeweilige Kundenklasse sprechen. Dazu wird die logarithmierte Auftrittshäufigkeit von korrenspondierenden Klassen addiert (log(*theta*)).

Das Resultat der SQL-Abfrage besteht aus zwei Spalten: einerseits die Kundenklassen-ID, andererseits die logarithmierte Wahrscheinlichkeit dafür, dass diese Klassen-ID der zu klassifizierenden Person zugewiesen werden soll. In der Ausgabe einer solchen Abfrage (wie im Listing 4.14) gibt es so viele Zeilen wie Kundenklassen. Die Entscheidung des Klassifikators – die Klasse, die einer Person zugewiesen werden soll – stünde in der ersten Zeile der Ausgabe, da die Zeilen nach absteigender log-probability sortiert sind.

Für Naïve-Bayes-Beispiele zu SQL2Stan stehen bash-Skripte bereit (Dateiname: classify_a_customer.sh), damit die Klassifikation eigenhängig ausprobiert werden kann. Erwähnenswert in diesem Zusammenhang ist auch, dass für alle Schritte des Inferenzworkflows bash-Skripte geschrieben wurden, damit der SQL2Stan-Prototyp unaufwändig getestet werden kann. Man soll einfach den dem Prototypen beiliegenden Anleitungsschritten folgen.

5. Schlussfolgerungen zum SQL2Stan-Ansatz

Das Alleinstellungsmerkmal von SQL2Stan-Ansatz

SQL2Stan ist ein neuer Ansatz zur deklarativen probabilistischen Programmierung in SQL für Datenmanagementsysteme. Dieser Ansatz hebt sich unter sämtlichen Projekten durch eine einmalige **Kombination** von vorteilhaften Merkmalen hervor, welche bei anderen Ansätzen teilweise fehlen:

- SQL2Stan ist einfach zu bedienen.
  - SQL2Stan orientiert sich an SQL-Programmierer. Die Interfacesprache ist reines SQL. Mit SQL2Stan muss ein SQL-Programmierer keine zusätzliche PPL erlernen, um Bayes'sche Inferenz auf datenbankgespeicherten Relationen anzuwenden. Er muss nur SQL beherrschen und einige wenige Zufallsverteilungen kennen.
  - SQL2Stan ist zugänglicher als die ohnehin einfache PPL Stan, weil komplexe Implementierungsdetails (z.B. Spezialbehandlung diskreter Modellparameter) automatisiert und vor dem Nutzer versteckt werden.
- Das Konzept von SQL2Stan erlaubt die Spezifikation beliebiger probabilistischer (auch hierarchischer) Bayes'scher Modelle. Somit können sehr unterschiedliche domänenspezifische Modelle unaufwändig nachimplementiert oder entworfen werden.
- Diese Masterarbeit zu SQL2Stan, gepaart mit dem Ansatz zur Übersetzung von Bayes'schen Modellen in Entity-Relationship-Schemata [RH16], deckt nahtlos den gesamten Workflow zur Bayes'schen Inferenz auf Datenbankdaten ab. Dazu gehören Entwurf eines probabilistischen Modells (Bayes'sches Netz), Datenmanagementabstraktion für Modellentitäten (relationale Schemata), probabilistische Programmierung (modellspezifische Zielfunktion deklarativ im SQL2Stan-Dialekt beschreiben), die Bayes'sche Inferenz selbst (automatisch generierter Stan-Code) sowie die Abfrage von Inferenzergebnissen (relationale Schemata). Die Übergänge zwischen all díesen Workflow-Schritten sind entweder automatisiert oder (falls manuell durchzuführen) ausreichend dokumentiert. Dieser Maß an theoretischer und praktischer Abdeckung des gesamten Inferenzworkflows mit DMS-Mitteln wurde von keiner der bisherigen Systemen oder Veröffentlichungen erreicht.

- SQL2Stan ist eine abstrakte SQL-Schicht, welche die Bayes'sche Inferenz mit dem Datenmanagement verknüpft. Das Inferenzbackend, die Inferenzalgorithmen und das Datenmanagementbackend sind unter dieser Schicht austauschbar. Die Backend-Komponenten brauchen nicht zwingend SQL- oder relationale Schnittstellen, sie müssen nur mit Arrays bzw. Vektoren arbeiten können.

- Die Deklarativität sowie die high-level-Abstraktion im SQL-Dialekt von SQL2Stan bieten viel Freiheit im Bezug auf automatische Codeoptimierung und Parallelisierung.

Arbeitsbeiträge rund um den SQL2Stan-Prototypen

Im Rahmen der Arbeit am SQL2Stan-Prototypen wurde eine domänenspezifische Sprache auf SQL-Basis entworfen, in der man Bayes'sche Modelle über logarithmierte Wahrscheinlichkeitfunktionen beschreiben kann. Zusätzlich wurden Prinzipien vorgestellt und implementiert, anhand derer der entworfene SQL-Dialekt in eine probabilistische Programmiersprache Stan automatisch übersetzt werden kann, um die Bayes'sche Inferenz mit Stan als Backend zu implementieren.

Der im Rahmen dieser Masterarbeit entstandene SQL2Stan-Prototyp verhält sich so, als wäre die Funktionalität zur deklarativ spezifizierbaren Bayes'schen Inferenz integriert mit dem Datenbanksystem. In Wahrheit muss der Prototyp die Modelldaten zwischen der DMS-Schicht (PostgreSQL-Datenbank) und ML-Schicht (Stan) bewegen, was ihn wesentlich verlangsamt und im aktuellen Zustand für Big-Data-Auswertung ungeeignet macht. Der Prototyp eignet sich dennoch zur Demonstration von dem SQL2Stan-Ansatz an drei kleineren Beispielen, um eine bessere (u.a. mit enger integrierten Softwareschichten) Implementierung von diesem Ansatz in Aussicht zu stellen.

SQL2Stan ist ein ausbaufähiger Vorschlag, der automatischen Inferenz und dem Datenmanagement mit SQL eine gemeinsame Sprache zu geben, die auch von Nichtspezialisten für ihre eigenen domänenspezifischen Inferenzaufgaben sinnvoll genutzt werden kann. Solange die Daten strukturierbar sind und in den Kontext Bayes'scher Modelle eingepflegt werden können, kann der deklarative SQL2Stan-Ansatz eingesetzt werden.

Ein Beispiel für gesammelte Erfahrungen

Der Übersetzer von SQL2Stan hätte aus technischer Sicht sehr viel besser sein können. Der Grund dafür war die Software jfq [Blu19] als eine zur Auswertung von JSON-formatierten Parserbäumen durchaus ungünstige Wahl. Dieses Projekt implementiert die browserorientierte JSON-Abfragesprache JSONata [Col19] für den Einsatz in der Kommandozeile. JSONata orientiert sich an XPath (eine sehr flexible und mächtige Abfragesprache für XML), lässt allerdings eine der Hauptstärken von XPath – die ausgiebigen Navigationsoperatoren (z.B. Schwesterknoten, Elternknoten usw.) – außen vor. Das macht die Auswertung von JSON-Parserbäumen in JSONATA sehr viel umständlicher. Anstelle von Jsonata wünschte ich mir einfach

eine XPath-Version für JSON, die aus einer Kommandozeile heraus bedienbar ist, aber sowas existiert scheinbar leider nicht – bzw. wenn, dann nur für den Browser.

Ein weiterer großer Schmerzpunkt: jfq als Implementierung von JSONata für die Kommandozeile unterstützt aus unbekannten Gründen keine Kommentare im Code, die vom JSONata-Sprachstandard vorgesehen sind (notiert mit /* und */). Ohne Kommentare ist es schwer, den JSONata-Code verständlich zu gestalten; mit Kommentaren kann jfq die JSONata-Befehlsdateien nicht ausführen. Daher ein kleines Fazit: man soll keine Parserbaum-Compiler mit JSONata bauen. Falls es keine andere Option gibt (wie in meinem Falle [1]), dann möglichst nicht mit jfq.

Potenzielle Weiterentwicklungen

SQL2Stan ließe sich sprachlich erweitern; für den Dialekt selbst ist es überhaupt kein Problem. Besonders wichtig wären hierbei zusätzliche Stan-spezifische Verteilungsfunktionen, die sich zu den zwei aktuell unterstützten (Dirichlet und Categorical) dazugesellen würde. Der SQL2Stan-Compiler ließe sich in dieser Hinsicht erweitern. Aus den erläuterten Gründen würde ich aber empfehlen, einen neuen Compiler zu bauen, der nicht auf JSONata und jfq basiert – das würde die Compilercodewartung sehr viel leichter machen. Den sprachlichen Aspekt betrifft außerdem die mögliche Berücksichtigung meiner eigenen Verbesserungsvorschläge, die ich beim Erläutern des SQL2Stan-Konzeptes in die Arbeit hineingebracht habe.

SQL2Stan ließe sich nicht nur sprachlich, sondern auch technisch verbessern. Eine tolle Neuerung wäre die DMS-Backend-Umstellung von PostgreSQL auf beispielsweise SciDB als ein Array-DBMS, denn SQL2Stan arbeitet im Verborgenen sowieso mit Arrays. Denkbar wäre vermutlich auch die Abbildung vom SQL2Stan-Dialekt auf PyMC4 (mit TensorFlow Probability als ML-Backend), was allerdings einen neuen Compiler benötigen würde. Im letzteren Falle hieße der Compiler vermutlich SQL2PyMC. Stan hat nämlich bestimmte Einschränkungen (Sampling von diskreten Modellparametern nicht möglich, LDA-ähnliche Modelle werden nicht mit voller Robustheit Bayes'scher Inferenz unterstützt), und durch einen ML-Backend-Austausch könnte man versuchen, diesen Problemen vorzubeugen. Und nicht zuletzt stünde die engere Integration vom ML- mit dem DMS-Backend. Durch die erwähnte Umstellung auf ein Array-DBMS zur physikalischen Datenverwaltung würde man einen großen Schritt in diese Richtung tun.

[1] Als SQL-Parser nahm ich das Projekt libpg_query [Fit19]. Es ist ein PostgreSQL-Parser für die Kommandozeile, und er gibt die aus dem SQL-Code generierten Parserbäume im JSON-Format zurück. So war ich gezwungen, eine JSON-Abfragesprache zu finden, die sich nicht auf die Browserfunktionalität einschränkt. JSONata verspricht XPath-ähnliche Sprachkonstrukte und vermag dadurch attraktiv zu wirken.

Abkürzungsverzeichnis

ADVI Automatic Differentiation Variational Inference

BN Bayes'sches Netz, Bayesian Network

DB Datenbank

DBMS Datenbankmanagementsystem, eine Art von DMS

DE Datenenthusiast nach [Han12]

DMS Datenmanagementsystem

ER Entity-Relationship(-Diagramm)

HMC Hamiltonian Monte Carlo

LDA Latent Dirichlet Allocation

MCMC Markov Chain Monte Carlo

ML Maschinelles Lernen, Machine Learning

PK Primary Key, (zusammengesetzter) Tabellenschlüssel

PPL Probabilistische Programmiersprache, Probabilistic Programming Language

UDF User Defined Function, benutzerdefinierte Funktion

VM Virtuelle Maschine

ZF Zielfunktion, modellspezifische (log-)likelihood-Funktion

Literatur

[Abe17] S. Abeywardana. *Stan vs PyMc3 vs Edward*. 2017. URL: https://towardsdatascience.com/stan-vs-pymc3-vs-edward-1d45c5d6da77.

[Abu+18] F. Abuzaid u. a. „DIFF: A Relational Interface for Large-scale Data Explanation“. In: *Proc. VLDB Endow.* 12.4 (Dez. 2018), S. 419–432. ISSN: 2150-8097. DOI: 10.14778/3297753.3297761. URL: https://doi.org/10.14778/3297753.3297761.

[AJ15] J. Adler und R. Johnson. *Strata+Hadoop Conference Video: The Sushi Principle – Raw Data Is Better*. 2015. URL: https://www.oreilly.com/library/view/strata-hadoop/9781491924143/video210840.html.

[Apa19] Apache Software Foundation. *Homepage of Apache Singa: an upcoming open source machine learning library, with a flexible architecture for scalable distributed ML model training, flexible in terms of supported hardware, developed mainly for healthcare applications*. 2019. URL: https://singa.incubator.apache.org/en/index.html#.

[Bai+17] P. Bailis u. a. „Infrastructure for Usable Machine Learning: The Stanford DAWN Project“. In: *CoRR* abs/1705.07538 (2017). arXiv: 1705.07538. URL: http://arxiv.org/abs/1705.07538.

[Bai+18] P. Bailis u. a. *The Stanford DAWN project: Slides*. 2018. URL: https://dawn.cs.stanford.edu/assets/dawn-overview.pdf.

[Bay+17] D. Baylor u. a. „TFX: A TensorFlow-Based Production-Scale Machine Learning Platform“. In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '17. Halifax, NS, Canada: ACM, 2017, S. 1387–1395. ISBN: 978-1-4503-4887-4. DOI: 10.1145/3097983.3098021. URL: http://doi.acm.org/10.1145/3097983.3098021.

[BBM18] L. Berti-Équille, A. Bonifati und T. Milo. „Machine Learning to Data Management: A Round Trip“. In: *ICDE*. IEEE Computer Society, 2018, S. 1735–1738. DOI: 10.1109/ICDE.2018.00226. URL: https://hal.archives-ouvertes.fr/hal-01795315/file/PID5217775.pdf.

[Bet17a] M. Betancourt. „A conceptual introduction to Hamiltonian Monte Carlo“. In: *arXiv preprint arXiv:1701.02434* (2017).

[Bet17b] M. Betancourt. *Diagnosing Biased Inference with Divergences*. 2017. URL: https://betanalpha.github.io/assets/case_studies/divergences_and_bias.html.

[Bis06] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 0387310738.

[BKM17] D. M. Blei, A. Kucukelbir und J. D. McAuliffe. „Variational inference: A review for statisticians“. In: *Journal of the American Statistical Association* 112.518 (2017), S. 859–877. URL: https://arxiv.org/abs/1601.00670.

[Blu19] G. Blue. *jfq: JSONata on the command line*. 2019. URL: https://github.com/blgm/jfq.

[BNJ03] D. M. Blei, A. Y. Ng und M. I. Jordan. „Latent Dirichlet Allocation“. In: *J. Mach. Learn. Res.* 3 (März 2003), S. 993–1022. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=944919.944937.

[Boe+16] M. Boehm u. a. „SystemML: Declarative Machine Learning on Spark“. In: *Proc. VLDB Endow.* 9.13 (Sep. 2016), S. 1425–1436. ISSN: 2150-8097. DOI: 10.14778/3007263.3007279. URL: https://doi.org/10.14778/3007263.3007279.

[Bro+03] P. A. Bromiley u. a. „Bayesian and non-Bayesian probabilistic models for medical image analysis“. In: *Image and Vision Computing* 21.10 (2003), S. 851–864.

[Bui+13] L. Buitinck u. a. „API design for machine learning software: experiences from the scikit-learn project“. In: *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 2013, S. 108–122.

[Bür16] P. Bürkner. *A talk about brms: an R Package for Bayesian MultilevelModels using Stan*. 2016. URL: https://www.uni-muenster.de/imperia/md/content/psyifp/ae_holling/brms_talk_26.02.16.pdf.

[Bür19] P. Bürkner. *GitHub repository of brms: R package for Bayesian generalized multivariate non-linear multilevel models using Stan*. 2019. URL: https://github.com/paul-buerkner/brms.

[Cai+13] Z. Cai u. a. „Simulation of Database-valued Markov Chains Using SimSQL“. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. New York, New York, USA: ACM, 2013, S. 637–648. ISBN: 978-1-4503-2037-5. DOI: 10.1145/2463676.2465283. URL: http://doi.acm.org/10.1145/2463676.2465283.

[Cai+19] Z. Cai u. a. *Homepage of SimSQL: a system for stochastic analytics implemented at Rice University*. 2019. URL: http://cmj4.web.rice.edu/SimSQL/SimSQL.html.

[Car+15] P. Carbone u. a. „Apache Flink™: Stream and Batch Processing in a Single Engine“. In: *IEEE Data Eng. Bull.* 38.4 (2015), S. 28–38. URL: http://asterios.katsifodimos.com/assets/publications/flink-deb.pdf.

[Car+17] B. Carpenter u. a. „Stan: A probabilistic programming language“. In: *Journal of statistical software* 76.1 (2017), S. 1–32. ISSN: 1548-7660. DOI: 10.18637/jss.v076.i01. URL: https://www.jstatsoft.org/v076/i01.

[Car13a] B. Carpenter. *Stan example model: LDA*. 2013. URL: https://github.com/stan-dev/example-models/blob/master/misc/cluster/lda/lda.stan.

[Car13b] B. Carpenter. *Stan example model: LDA model data*. 2013. URL: https://github.com/stan-dev/example-models/blob/master/misc/cluster/lda/lda.data.R.

[Car13c] B. Carpenter. *Stan example model: supervised naive Bayes*. 2013. URL: https://github.com/stan-dev/example-models/blob/master/misc/cluster/naive-bayes/naive-bayes-sup.stan.

[Car13d] B. Carpenter. *Stan example model: unsupervised naive Bayes*. 2013. URL: https://github.com/stan-dev/example-models/blob/master/misc/cluster/naive-bayes/naive-bayes-unsup.stan.

[Car13e] B. Carpenter. *Supervised naive Bayes data simulator/generator*. 2013. URL: https://github.com/stan-dev/example-models/blob/master/misc/cluster/naive-bayes/sim-naive-bayes.R.

[Car17] B. Carpenter. *Hello, world! Stan, PyMC3, and Edward: Kommentar Nr. 500516 „That makes sense. We aren't at all targeting those machine learning models...“* 2017. URL: https://statmodeling.stat.columbia.edu/2017/05/31/compare-stan-pymc3-edward-hello-world/.

[Car18] F. Carrone. *Inteview with Thomas Wiecki about PyMC and probabilistic programming*. 2018. URL: https://notamonadtutorial.com/inteview-with-thomas-wiecki-about-probabilistic-programming-and-pymc-66a12b6f3f2e.

[Cas19] Cascading Maintainers. *Homepage of Cascading Ecosystem: a collection of applications, languages, and APIs for developing data-intensive applications*. 2019. URL: https://www.cascading.org/.

[Cho+15] F. Chollet u. a. *Homepage of Keras: The Python Deep Learning library*. 2015. URL: https://keras.io.

[Chu19] N. Y. Chuan. *GitHub repository of Rafiki: a distributed system that supports training and deployment of machine learning models using AutoML, built with ease-of-use in mind*. 2019. URL: https://github.com/nginyc/rafiki.

[Cla05] J. S. Clark. „Why environmental scientists are becoming Bayesians“. In: *Ecology Letters* 8.1 (2005), S. 2–14. DOI: 10.1111/j.1461-0248.2004.00702.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1461-0248.2004.00702.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1461-0248.2004.00702.x.

[Coh+09] J. Cohen u. a. „MAD Skills: New Analysis Practices for Big Data“. In: *Proc. VLDB Endow.* 2.2 (Aug. 2009), S. 1481–1492. ISSN: 2150-8097. DOI: 10.14778/1687553.1687576. URL: https://doi.org/10.14778/1687553.1687576.

[Col19] A. Coleman. *JSONata: JSON query and transformation language*. 2019. URL: https://github.com/jsonata-js/jsonata.

[Com19] Computation and Cognition Lab at Stanford University. *GitHub repository of WebPPL: Probabilistic programming for the web*. 2019. URL: https://github.com/probmods/webppl.

[Cra+17] D. Crankshaw u. a. „Clipper: A Low-Latency Online Prediction Serving System“. In: *NSDI*. USENIX Association, 2017, S. 613–627. URL: https://www.usenix.org/system/files/conference/nsdi17/nsdi17-crankshaw.pdf.

[Dav18] C. Davidson-Pilon. *Bayesian Methods for Hackers*. 2018. URL: https://github.com/CamDavidsonPilon/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers.

[DBS13] X. L. Dong, L. Berti-Équille und D. Srivastava. „Data Fusion: Resolving Conflicts from Multiple Sources“. In: *WAIM*. Bd. 7923. Lecture Notes in Computer Science. Springer, 2013, S. 64–76. URL: https://arxiv.org/abs/1503.00310.

[De+16] S. De u. a. „BayesWipe: A Scalable Probabilistic Framework for Improving Data Quality“. In: *J. Data and Information Quality* 8.1 (Okt. 2016), 5:1–5:30. ISSN: 1936-1955. DOI: 10.1145/2992787. URL: http://doi.acm.org/10.1145/2992787.

[Des+04] A. Deshpande u. a. „Model-driven Data Acquisition in Sensor Networks“. In: *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*. VLDB '04. Toronto, Canada: VLDB Endowment, 2004, S. 588–599. ISBN: 0-12-088469-0. URL: http://dl.acm.org/citation.cfm?id=1316689.1316741.

[DF15] M. Dishman und M. Fowler. „Video: Agile Architecture“. In: *at O'Reilly Software Architecture Conference*. 2015. URL: https://www.youtube.com/watch?v=VjKYO6DP3fo.

[DG08] J. Dean und S. Ghemawat. „MapReduce: Simplified Data Processing on Large Clusters“. In: *Commun. ACM* 51.1 (Jan. 2008), S. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: http://doi.acm.org/10.1145/1327452.1327492.

[DG13] M. S. Divya und S. K. Goyal. „ElasticSearch: An advanced and quick search technique to handle voluminous data“. In: *Compusoft* 2.6 (2013), S. 171. URL: https://www.ijact.in/index.php/ijact/article/download/380/327.

[DKM10] D. Deutch, C. Koch und T. Milo. „On Probabilistic Fixpoint and Markov Chain Query Languages“. In: *Proceedings of the Twenty-ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS '10. Indianapolis, Indiana, USA: ACM, 2010, S. 215–226. ISBN: 978-1-4503-0033-9. DOI: 10.1145/1807085.1807114. URL: http://doi.acm.org/10.1145/1807085.1807114.

[DM06] A. Deshpande und S. Madden. „MauveDB: Supporting Model-based User Views in Database Systems“. In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. SIGMOD '06. Chicago, IL, USA: ACM, 2006, S. 73–84. ISBN: 1-59593-434-0. DOI: 10.1145/1142473.1142483. URL: http://doi.acm.org/10.1145/1142473.1142483.

[Don+14] X. L. Dong u. a. „From Data Fusion to Knowledge Fusion“. In: *Proc. VLDB Endow.* 7.10 (Juni 2014), S. 881–892. ISSN: 2150-8097. DOI: 10.14778/2732951.2732962. URL: http://dx.doi.org/10.14778/2732951.2732962.

[DS317] DS3Lab. *GitHub repository of MLog: the experiments' code of MLog demo paper*. 2017. URL: https://github.com/DS3Lab/MLog.

[DS319] DS3Lab. *GitHub repository of ease.ml: a Scalable Auto-ML System*. 2019. URL: https://github.com/DS3Lab/easeml.

[Eis16] R. Eisele. *The log-sum-exp trick in Machine Learning*. 2016. URL: https://www.xarg.org/2016/06/the-log-sum-exp-trick-in-machine-learning/.

[ela18] elastic.co. *Why Elasticsearch SQL?* 2018. URL: https://www.elastic.co/guide/en/elasticsearch/reference/6.3/sql-overview.html#sql-why.

[Fär+12] F. Färber u. a. „The SAP HANA Database–An Architecture Overview.“ In: *IEEE Data Eng. Bull.* 35.1 (2012), S. 28–33.

[Fen+12] X. Feng u. a. „Towards a Unified Architecture for in-RDBMS Analytics“. In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. SIGMOD '12. Scottsdale, Arizona, USA: ACM, 2012, S. 325–336. ISBN: 978-1-4503-1247-9. DOI: 10.1145/

2213836.2213874. URL: http://doi.acm.org/10.1145/2213836.2213874.

[Fit19] L. Fittl. *libpg_query: a C library for accessing the PostgreSQL parser outside of the server environment*. 2019. URL: https://github.com/lfittl/libpg_query.

[Goo+12] N. D. Goodman u. a. „Church: a language for generative models“. In: *CoRR* abs/1206.3255 (2012). URL: https://arxiv.org/abs/1206.3255.

[Gor+14] A. D. Gordon u. a. „Tabular: A Schema-driven Probabilistic Programming Language“. In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. San Diego, California, USA: ACM, 2014, S. 321–334. ISBN: 978-1-4503-2544-8. DOI: 10.1145/2535838.2535850. URL: http://doi.acm.org/10.1145/2535838.2535850.

[Gro+15] M. Grover u. a. *Hadoop Application Architectures: Designing Real-World Big Data Applications*. "O'Reilly Media, Inc.", 2015. ISBN: 9781491900086.

[Ham12] L. C. Hamilton. *Statistics with STATA - Version 12*. 8th. Boston, MA, USA: Duxbury Press, 2012. ISBN: 0840064632, 9780840064639.

[Han12] P. Hanrahan. „Analytic Database Technologies for a New Kind of User: The Data Enthusiast“. In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. SIGMOD '12. Scottsdale, Arizona, USA: ACM, 2012, S. 577–578. ISBN: 978-1-4503-1247-9. DOI: 10.1145/2213836.2213902. URL: http://doi.acm.org/10.1145/2213836.2213902.

[HDR10] D. Husmeier, R. Dybowski und S. Roberts. *Probabilistic Modeling in Bioinformatics and Medical Informatics*. 1st. Springer Publishing Company, Incorporated, 2010. ISBN: 1849969124, 9781849969123.

[Hel+12] J. M. Hellerstein u. a. „The MADlib Analytics Library: Or MAD Skills, the SQL“. In: *Proc. VLDB Endow.* 5.12 (Aug. 2012), S. 1700–1711. ISSN: 2150-8097. DOI: 10.14778/2367502.2367510. URL: http://dx.doi.org/10.14778/2367502.2367510.

[HG14] M. D. Homan und A. Gelman. „The No-U-turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo“. In: *J. Mach. Learn. Res.* 15.1 (Jan. 2014), S. 1593–1623. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=2627435.2638586.

[Hin19] A. Hinneburg. *GitHub repository of TopicExplorer: a web-based topic model browser*. 2019. URL: https://github.com/hinneburg/TopicExplorer.

[HMG] R. Herbrich, T. Minka und T. Graepel. „TrueSkill^{TM}: A Bayesian Skill Rating System“. In: *Advances in Neural Information Processing Systems (NIPS) 20*. MIT Press, S. 569–576. URL: https://www.microsoft.com/en-us/research/publication/trueskilltm-a-bayesian-skill-rating-system/.

[Hof+13] M. D. Hoffman u. a. „Stochastic variational inference“. In: *The Journal of Machine Learning Research* 14.1 (2013), S. 1303–1347.

[How06] B. Howe. „Gridfields: model-driven data transformation in the physical sciences“. In: (2006). URL: https://pdxscholar.library.pdx.edu/open_access_etds/2676/.

[HSI17] J. Hilbe, R. de Souza und E. Ishida. *Bayesian Models for Astrophysical Data: Using R, JAGS, Python, and Stan*. Cambridge University Press, Mai 2017. ISBN: 9781108210744. DOI: 10.1017/CBO9781316459515. URL: https://books.google.de/books?id=7D2wDgAAQBAJ.

[Jam+08] R. Jampani u. a. „MCDB: A Monte Carlo Approach to Managing Uncertain Data“. In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. SIGMOD '08. Vancouver, Canada: ACM, 2008, S. 687–700. ISBN: 978-1-60558-102-6. DOI: 10.1145/1376616.1376686. URL: http://doi.acm.org/10.1145/1376616.1376686.

[Jan+19] D. Jankov u. a. „Declarative Recursive Computation on an RDBMS: Why You Should Use a Database For Distributed Machine Learning“. In: *Proceedings of the VLDB Endowment* 12.7 (2019), S. 822–835. URL: http://www.vldb.org/pvldb/vol12/p822-jankov.pdf.

[JER17a] R. Jagerman, C. Eickhoff und M. de Rijke. „Computing Web-scale Topic Models Using an Asynchronous Parameter Server“. In: *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '17. Shinjuku, Tokyo, Japan: ACM, 2017, S. 1337–1340. ISBN: 978-1-4503-5022-8. DOI: 10.1145/3077136.3084135. URL: http://doi.acm.org/10.1145/3077136.3084135.

[JER17b] R. Jagerman, C. Eickhoff und M. de Rijke. „Computing Web-scale Topic Models using an Asynchronous Parameter Server“. In: *SIGIR*. ACM, 2017, S. 1337–1340.

[Jia+14] Y. Jia u. a. „Caffe: Convolutional Architecture for Fast Feature Embedding“. In: *Proceedings of the 22Nd ACM International Conference on Multimedia*. MM '14. Orlando, Florida, USA: ACM, 2014, S. 675–678. ISBN: 978-1-4503-3063-3. DOI: 10.1145/2647868.2654889. URL: http://doi.acm.org/10.1145/2647868.2654889.

[Jia+16] D. Jiang u. a. „Cohort Query Processing“. In: *Proc. VLDB Endow.* 10.1 (Sep. 2016), S. 1–12. ISSN: 2150-8097. DOI: 10.14778/3015270.3015271. URL: https://doi.org/10.14778/3015270.3015271.

[Jia+17] J. Jiang u. a. „Angel: a new large-scale machine learning system“. In: *National Science Review* 5.2 (2017), S. 216–236.

[Kag12] Kaggle Team. *Observing Dark Worlds Competition Winner*. 2012. URL: http://blog.kaggle.com/2012/12/19/1st-place-observing-dark-worlds/.

[Kar+18a] K. Kara u. a. „ColumnML: Column-store Machine Learning with On-the-fly Data Transformation“. In: *Proc. VLDB Endow.* 12.4 (Dez. 2018), S. 348–361. ISSN: 2150-8097. DOI: 10.14778/3297753.3297756. URL: https://doi.org/10.14778/3297753.3297756.

[Kar+18b] B. Karlaš u. a. „Ease.Ml in Action: Towards Multi-tenant Declarative Learning Services“. In: *Proc. VLDB Endow.* 11.12 (Aug. 2018), S. 2054–2057. ISSN: 2150-8097. DOI: 10.14778/3229863.3236258. URL: https://doi.org/10.14778/3229863.3236258.

[Kat18] J.-P. Katoen. „Probabilistic Programming: Quantitative Modeling for the Masses?“ In: *Keynote of the MMB 2018 Conference, Erlangen* (2018). URL: https://www.mmb2018.de/files/mmb2018_keynote_reduced.pdf.

[KBY17] A. Kumar, M. Boehm und J. Yang. „Data Management in Machine Learning: Challenges, Techniques, and Systems“. In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: ACM, 2017, S. 1717–1722. ISBN: 978-1-4503-4197-4. DOI: 10.1145/3035918.3054775. URL: http://doi.acm.org/10.1145/3035918.3054775.

[KE06] A. Kemper und A. Eickler. *Datenbanksysteme - Eine Einführung, 6. Auflage*. Oldenbourg, 2006. ISBN: 3-486-57690-9.

[Kle17] M. Kleppmann. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. "O'Reilly Media, Inc.", 2017. ISBN: 9781449373320.

[KNP15] A. Kumar, J. Naughton und J. M. Patel. „Learning Generalized Linear Models Over Normalized Data“. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: ACM, 2015, S. 1969–1984. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2723713. URL: http://doi.acm.org/10.1145/2723372.2723713.

[Kra+13] T. Kraska u. a. „MLbase: A Distributed Machine-learning System“. In: *Conference on Innovative Data Systems Research (CIDR) '13*. Bd. 1. 2013, S. 2–1. URL: http://cidrdb.org/cidr2013/Papers/CIDR13_Paper118.pdf.

[Kuc+17] A. Kucukelbir u. a. „Automatic Differentiation Variational Inference“. In: *J. Mach. Learn. Res.* 18.1 (Jan. 2017), S. 430–474. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=3122009.3122023.

[Li+17a] T. Li u. a. „Ease.ml: Towards Multi-tenant Resource Sharing for Machine Learning Workloads“. In: *CoRR* abs/1708.07308 (2017). arXiv: 1708.07308. URL: http://arxiv.org/abs/1708.07308.

[Li+17b] X. Li u. a. „MLog: Towards Declarative In-database Machine Learning“. In: *Proc. VLDB Endow.* 10.12 (Aug. 2017), S. 1933–1936. ISSN: 2150-8097. DOI: 10.14778/3137765.3137812. URL: https://doi.org/10.14778/3137765.3137812.

[Lin19] LinkedIn. *GitHub repository of TonY: a framework to natively run deep learning frameworks on Apache Hadoop*. 2019. URL: https://github.com/linkedin/TonY.

[LM12] P. Leyshock und D. Maier. „Agrios: A hybrid approach to scalable data analysis systems“. In: 2012. URL: https://www.semanticscholar.org/paper/Agrios%3A-A-Hybrid-Approach-to-Scalable-Data-Analysis-Maier/33e642854e1cb2f45646558c8be138c3969041cf.

[MAD19] MADLib. *Homepage of Apache MADlib: Big Data Machine Learning in SQL*. Apache. 2019. URL: https://madlib.apache.org/.

[Man+15] V. K. Mansinghka u. a. „BayesDB: A probabilistic programming system for querying the probable implications of data“. In: *CoRR* abs/1512.05006 (2015). URL: https://arxiv.org/abs/1512.05006.

[Mar+15] Martín Abadi u. a. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. URL: https://www.tensorflow.org/.

[McD18] D. McDiarmid. *Einführung in Elasticsearch SQL mit praktischen Beispielen – Teil 1*. 2018. URL: https://www.elastic.co/de/blog/an-introduction-to-elasticsearch-sql-with-practical-examples-part-1.

[Men+16] X. Meng u. a. „MLlib: Machine Learning in Apache Spark“. In: *J. Mach. Learn. Res.* 17.1 (Jan. 2016), S. 1235–1241. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=2946645.2946679.

[Mil+05] B. Milch u. a. „BLOG: Probabilistic Models with Unknown Objects“. In: *Probabilistic, Logical and Relational Learning*. Bd. 05051. Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005. URL: https://www.microsoft.com/en-us/research/publication/blog-probabilistic-models-unknown-objects/.

[MIT19] MIT Probabilistic Computing Project. *GitHub repository of BayesDB on SQLite: a Bayesian database table for querying the probable implications of data as easily as SQL databases query the data itself*. 2019. URL: https://github.com/probcomp/bayeslite.

[Möb18a] C. Möbus. *Structure and Interpretation of WebPPL*. 2018. URL: https://uol.de/en/lcs/probabilistic-programming/webppl-a-probabilistic-functional-programming-language/structure-and-interpretation-of-webppl/.

[Möb18b] C. Möbus. *WebPPL - A Probabilistic Functional Programming Language*. 2018. URL: https://uol.de/en/lcs/probabilistic-programming/webppl-a-probabilistic-functional-programming-language/.

[NET19a] .NET Foundation. *GitHub repository of Infer.NET: a framework for running Bayesian inference in graphical models*. 2019. URL: https://github.com/dotnet/infer.

[NET19b] .NET Foundation and Contributors. *The Infer.NET Modelling API*. 2019. URL: https://dotnet.github.io/infer/userguide/The%20Infer.NET%20modelling%20API.html.

[Nii86] H. P. Nii. „Blackboard Systems, Part One: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures“. In: *AI Magazine* 7.2 (1986), S. 38–53. URL: http://i.stanford.edu/pub/cstr/reports/cs/tr/86/1123/CS-TR-86-1123.pdf.

[NJ01] A. Y. Ng und M. I. Jordan. „On Discriminative vs. Generative Classifiers: a comparison of logistic regression and naive Bayes“. In: *NIPS*. MIT Press, 2001, S. 841–848. URL: https://ai.stanford.edu/~ang/papers/nips01-discriminativegenerative.pdf.

[Ooi+14] B. C. Ooi u. a. „Contextual Crowd Intelligence“. In: *SIGKDD Explor. Newsl.* 16.1 (Sep. 2014), S. 39–46. ISSN: 1931-0145. DOI: 10.1145/2674026.2674032. URL: http://doi.acm.org/10.1145/2674026.2674032.

[Ora19] Oracle. *Oracle SQL Developer Data Modeler*. 2019. URL: https://www.oracle.com/de/database/technologies/appdev/datamodeler.html.

[Par+17] Y. Park u. a. „Database Learning: Toward a Database That Becomes Smarter Every Time“. In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: ACM, 2017, S. 587–602. ISBN: 978-1-4503-4197-4. DOI: 10.1145/3035918.3064013. URL: http://doi.acm.org/10.1145/3035918.3064013.

[PBP17] R. Pradhan, S. Bykau und S. Prabhakar. „Staging User Feedback Toward Rapid Conflict Resolution in Data Fusion“. In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: ACM, 2017, S. 603–618. ISBN: 978-1-4503-4197-4. DOI: 10.1145/3035918.3035941. URL: http://doi.acm.org/10.1145/3035918.3035941.

[Per19] Perry de Valpine's research group at UC Berkeley. *How we make MCMC comparisons: efficiency*. 2019. URL: https://nature.berkeley.edu/~pdevalpine/MCMC_comparisons/nimble_MCMC_comparisons.html.

[Pfe07] A. Pfeffer. „The Design and Implementation of IBAL: A General-Purpose Probabilistic Language". In: *Introduction to statistical relational learning* (2007), S. 399. URL: https://dash.harvard.edu/handle/1/25105000.

[Pfe09] A. Pfeffer. „Figaro: An object-oriented probabilistic programming language". In: *Charles River Analytics Technical Report* 137 (2009), S. 96. URL: https://www.cs.tufts.edu/~nr/cs257/archive/avi-pfeffer/figaro.pdf.

[PyM18] PyMC Developers. *Theano, TensorFlow and the Future of PyMC*. 2018. URL: https://medium.com/@pymc_devs/theano-tensorflow-and-the-future-of-pymc-6c9987bb19d5.

[PyM19a] PyMC Development Team. *GitHub repository for PyMC3: Bayesian Modeling and Probabilistic Programming in Python and Probabilistic Machine Learning with Theano*. 2019. URL: https://github.com/pymc-devs/pymc3.

[PyM19b] PyMC Development Team. *GitHub repository for PyMC4 (Pre-release): Probabilistic Programming in Python, now with flowing tensors*. 2019. URL: https://github.com/pymc-devs/pymc4.

[PyM19c] PyMC development team. *GitHub repository of PyMC: Bayesian Stochastic Modelling in Python*. 2019. URL: https://github.com/pymc-devs/pymc.

[Rat+17] A. J. Ratner u. a. „Snorkel: Fast Training Set Generation for Information Extraction". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: ACM, 2017, S. 1683–1686. ISBN: 978-1-4503-4197-4. DOI: 10.1145/3035918.3056442. URL: http://doi.acm.org/10.1145/3035918.3056442.

[Rei18] W. Reisz. *Interview: Mike Lee Williams on Probabilistic Programming, Bayesian Inference, and Languages Like PyMC3*. 2018. URL: https://www.infoq.com/podcasts/mike-lee-williams-probabilistic-programming.

[Rek+17] T. Rekatsinas u. a. „HoloClean: Holistic Data Repairs with Probabilistic Inference". In: *Proc. VLDB Endow.* 10.11 (Aug. 2017), S. 1190–1201. ISSN: 2150-8097. DOI: 10.14778/3137628.3137631. URL: https://doi.org/10.14778/3137628.3137631.

[RH16] F. Rosner und A. Hinneburg. „Translating Bayesian Networks into Entity Relationship Models, Extended Version“. In: *CoRR* abs/1607.02399 (2016). arXiv: 1607.02399. URL: http://arxiv.org/abs/1607.02399.

[Rip+19] B. Ripley u. a. *MASS Package for R: Support Functions and Datasets for Venables and Ripley's MASS (Modern Applied Statistics with S)*. 2019. URL: https://cran.r-project.org/web/packages/MASS/MASS.pdf.

[Roy11] D. M. Roy. „Computability, inference and modeling in probabilistic programming“. Diss. Massachusetts Institute of Technology, 2011. URL: http://danroy.org/papers/Roy-PHD-2011.pdf.

[SFD93] M. Stonebraker, J. Frew und J. Dozier. „The SEQUOIA 2000 Project“. In: *SSD*. Bd. 692. Lecture Notes in Computer Science. Springer, 1993, S. 397–412. URL: https://www.researchgate.net/publication/221471564_The_SEQUOIA_2000_Project.

[SG13] S. Singh und T. Graepel. „Automated probabilistic modeling for relational data“. In: *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*. CIKM '13. San Francisco, California, USA: ACM, 2013, S. 1497–1500. ISBN: 978-1-4503-2263-8. DOI: 10.1145/2505515.2507828. URL: http://doi.acm.org/10.1145/2505515.2507828.

[Shi+16] X. Shi u. a. „UniAD: A Unified Ad Hoc Data Processing System“. In: *ACM Trans. Database Syst.* 42.1 (Nov. 2016), 6:1–6:42. ISSN: 0362-5915. DOI: 10.1145/3009957. URL: http://doi.acm.org/10.1145/3009957.

[SKW15] T. Salimans, D. P. Kingma und M. Welling. „Markov Chain Monte Carlo and Variational Inference: Bridging the Gap“. In: *ICML*. Bd. 37. JMLR Workshop and Conference Proceedings. JMLR.org, 2015, S. 1218–1226.

[SL91] J. Seib und G. Lausen. „Parallelizing Datalog Programs by Generalized Pivoting“. In: *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. PODS '91. Denver, Colorado, USA: ACM, 1991, S. 241–251. ISBN: 0-89791-430-9. DOI: 10.1145/113413.113435. URL: http://doi.acm.org/10.1145/113413.113435.

[SN10] A. Smola und S. Narayanamurthy. „An architecture for parallel topic models“. In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), S. 703–710. URL: http://vldb.org/pvldb/vldb2010/papers/R63.pdf.

[SS00] C. Shenoy und P. P. Shenoy. „Bayesian network models of portfolio risk and return“. In: *Computational finance conference 1999*. CF '99. New York, USA: MIT Press, 2000, S. 87–106. ISBN: 0262011786, 026251107X. URL: https://www.researchgate.net/publication/2855827_Bayesian_Network_Models_of_Portfolio_Risk_and_Return.

[SS01] S. Sarkar und R. S. Sriram. „Bayesian models for early warning of bank failures“. In: *Management Science* 47.11 (2001), S. 1457–1475.

[Sta18] Stan Development Team. *Brief Guide to Stan's Warnings*. 2018. URL: https://mc-stan.org/misc/warnings.html.

[Sta19a] Stan Development Team. *Homepage of Stan: a platform for statistical modeling and high-performance statistical computation*. 2019. URL: https://mc-stan.org/.

[Sta19b] Stan Development Team. *Stan development repository on GitHub*. 2019. URL: https://github.com/stan-dev/stan.

[Sta19c] Stan Development Team. *Stan User's Guide*. 2019. URL: https://github.com/stan-dev/stan/releases/download/v2.18.0/users-guide-2.18.0.pdf.

[Sta19d] Stan Development Team. *Stan User's Guide: Latent Discrete Parameters*. 2019. URL: https://mc-stan.org/docs/2_19/stan-users-guide/latent-discrete-chapter.html.

[Sta19e] StataCorp LLC. *Bayesian Analysis with Stata*. 2019. URL: https://www.stata.com/features/bayesian-analysis/.

[Sto+07] M. Stonebraker u. a. „One size fits all? Part 2: Benchmarking results“. In: *Proc. CIDR*. 2007. URL: https://www.researchgate.net/publication/220988181_One_Size_Fits_All_Part_2_Benchmarking_Studies.

[Sto+09] M. Stonebraker u. a. „Requirements for Science Data Bases and SciDB“. In: Conference on Innovative Data Systems Research (CIDR), 2009.

[Sto+13] M. Stonebraker u. a. „SciDB: A database management system for applications with complex analytics“. In: *Computing in Science & Engineering* 15.3 (2013), S. 54–62. URL: https://ieeexplore.ieee.org/document/6461866.

[Tam+05] P. Tamayo u. a. „Oracle Data Mining - Data Mining in the Database Environment“. In: *The Data Mining and Knowledge Discovery Handbook*. Springer, 2005, S. 1315–1329.

[Ten19a] Tensorflow. *GitHub repository of Tensorflow Probability: Probabilistic reasoning and statistical analysis in TensorFlow*. 2019. URL: https://github.com/tensorflow/probability.

[Ten19b] Tensorflow.org. *TensorFlow Probability: a library for probabilistic reasoning and statistical analysis*. 2019. URL: https://www.tensorflow.org/probability.

[The16] Theano Development Team. „Theano: A Python framework for fast computation of mathematical expressions“. In: *arXiv e-prints* abs/1605.02688 (Mai 2016). URL: http://arxiv.org/abs/1605.02688.

[The19a] The Apache Software Foundation. *GitHub repository of Apache SystemML: a machine learning platform optimal for big data*. 2019. URL: https://github.com/apache/systemml.

[The19b] The Apache Software Foundation. *Homepage of Apache HAWQ: Apache Hadoop Native SQL*. 2019. URL: https://hawq.apache.org/.

[The19c] The Apache Software Foundation. *Homepage of Apache HBase: a Hadoop database, a distributed, scalable, big data store*. 2019. URL: https://hbase.apache.org/.

[The19d] The Apache Software Foundation. *Homepage of Apache Hive: a data warehouse software for reading, writing, and managing large datasets residing in distributed storage using SQL*. 2019. URL: https://hive.apache.org/.

[The19e] The Apache Software Foundation. *Homepage of Apache Impala: a open source, native analytic database for Apache Hadoop*. 2019. URL: https://impala.apache.org/.

[The19f] The Apache Software Foundation. *Homepage of Apache Pig: a platform for analyzing large data sets that consists of a high-level language for expressing data analysis programs, coupled with infrastructure for evaluating these programs*. 2019. URL: https://pig.apache.org/.

[The19g] The Apache Software Foundation. *Homepage of MLlib: Apache Spark's scalable machine learning library*. 2019. URL: https://spark.apache.org/mllib/.

[The19h] The Apache Software Foundation. *Spark SQL: Apache Spark's module for working with structured data*. 2019. URL: https://spark.apache.org/sql/.

[Tra+16] D. Tran u. a. *Edward: A library for probabilistic modeling, inference, and criticism*. cite arxiv:1610.09787. 2016. URL: http://arxiv.org/abs/1610.09787.

[Tra+17] D. Tran u. a. „Deep Probabilistic Programming“. In: *CoRR* abs/1701.03757 (2017).

[Tra18] D. Tran. *Upgrading from Edward to Edward2*. 2018. URL: https://github.com/tensorflow/probability/blob/master/tensorflow_probability/python/edward2/Upgrading_From_Edward_To_Edward2.md.

[Tra19] D. Tran. *Edward2: a probabilistic programming language for TensorFlow Probability*. 2019. URL: https://github.com/tensorflow/probability/blob/master/tensorflow_probability/python/edward2/README.md.

[Ube19] Uber AI Labs. *GitHub repository of pyro: Deep universal probabilistic programming with Python and PyTorch*. 2019. URL: https://github.com/pyro-ppl/pyro.

[Vav+13] V. K. Vavilapalli u. a. „Apache Hadoop YARN: Yet Another Resource Negotiator“. In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC '13. Santa Clara, California: ACM, 2013, 5:1–5:16. ISBN: 978-1-4503-2428-1. DOI: 10.1145/2523616.2523633. URL: http://doi.acm.org/10.1145/2523616.2523633.

[Waa08] F. M. Waas. „Beyond Conventional Data Warehousing - Massively Parallel Data Processing with Greenplum Database - (Invited Talk)“. In: *BIRTE*. Bd. 27. Lecture Notes in Business Information Processing. Springer, 2008, S. 89–96. URL: https://link.springer.com/chapter/10.1007/978-3-642-03422-0_7.

[Wal04] B. Walsh. „Markov Chain Monte Carlo and Gibbs Sampling“. In: *Lecture Notes for EEB 581, version 26, April* (Jan. 2004). URL: http://nitro.biosci.arizona.edu/courses/EEB596/handouts/Gibbs.pdf.

[Wan+16a] W. Wang u. a. „Database Meets Deep Learning: Challenges and Opportunities“. In: *SIGMOD Rec.* 45.2 (Sep. 2016), S. 17–22. ISSN: 0163-5808. DOI: 10.1145/3003665.3003669. URL: http://doi.acm.org/10.1145/3003665.3003669.

[Wan+16b] W. Wang u. a. „Effective Deep Learning-based Multi-modal Retrieval“. In: *The VLDB Journal* 25.1 (Feb. 2016), S. 79–101. ISSN: 1066-8888. DOI: 10.1007/s00778-015-0391-4. URL: http://dx.doi.org/10.1007/s00778-015-0391-4.

[Wan+18] W. Wang u. a. „Rafiki: Machine Learning As an Analytics Service System“. In: *Proc. VLDB Endow.* 12.2 (Okt. 2018), S. 128–140. ISSN: 2150-8097. DOI: 10.14778/3282495.3282499. URL: https://doi.org/10.14778/3282495.3282499.

[Wu+08] X. Wu u. a. „Top 10 algorithms in data mining“. In: *Knowledge and information systems* 14.1 (2008), S. 1–37.

[WW11] S. S. Wang und M. P. Wand. „Using Infer.net for statistical analyses“. In: *The American Statistician* 65.2 (2011), S. 115–126.

[Yah19] Yahoo Inc. *GitHub repository of TensorFlowOnSpark: brings TensorFlow programs to Apache Spark clusters*. 2019. URL: https://github.com/yahoo/TensorFlowOnSpark.

[Yut+17] L. Yut u. a. „LDA*: A Robust and Large-scale Topic Modeling System“. In: *Proc. VLDB Endow.* 10.11 (Aug. 2017), S. 1406–1417. ISSN: 2150-8097. DOI: 10.14778/3137628.3137649. URL: https://doi.org/10.14778/3137628.3137649.

[YW16] T. Yarkoni und J. Westfall. „Bambi: A simple interface for fitting Bayesian mixed effects models“. In: (2016). URL: https://osf.io/rv7sn/download.

[Zah+16] M. Zaharia u. a. „Apache Spark: A Unified Engine for Big Data Processing“. In: *Commun. ACM* 59.11 (Okt. 2016), S. 56–65. ISSN: 0001-0782. DOI: 10.1145/2934664. URL: http://doi.acm.org/10.1145/2934664.

[Zay18] Y. Zaykov. *The Microsoft Infer.NET machine learning framework goes open source*. 2018. URL: https://www.microsoft.com/en-us/research/blog/the-microsoft-infer-net-machine-learning-framework-goes-open-source/.

[Zha+12] K. Zhai u. a. „Mr. LDA: A Flexible Large Scale Topic Modeling Package Using Variational Inference in MapReduce“. In: *Proceedings of the 21st International Conference on World Wide Web*. WWW '12. Lyon, France: ACM, 2012, S. 879–888. ISBN: 978-1-4503-1229-5. DOI: 10.1145/2187836.2187955. URL: http://doi.acm.org/10.1145/2187836.2187955.

[Zlo75] M. M. Zloof. „Query-by-example: The Invocation and Definition of Tables and Forms“. In: *Proceedings of the 1st International Conference on Very Large Data Bases*. VLDB '75. Framingham, Massachusetts: ACM, 1975, S. 1–24. ISBN: 978-1-4503-3920-9. DOI: 10.1145/1282480.1282482. URL: http://doi.acm.org/10.1145/1282480.1282482.

A. Replikation der SQL2Stan-Experimente

Der Leser soll sich frei fühlen, den SQL2Stan-Prototypen eigenhändig zu testen. Notwendig sind hierfür: PostgreSQL, libpg_query, jfq. Befolgen Sie dazu bitte die Installations- und Nutzungsanleitungen (INSTALLATION INSTRUCTIONS.txt sowie README.txt) im Ordner „SQL2STAN“. Die Installationsanweisungen sind für **Ubuntu 18.10 Cosmic Cuttlefish** ausgelegt; die Durchführbarkeit genauer Installationsanweisungen ist nicht garantiert für andere Linux-Systeme.

A.1. Virtuelle Maschine für SQL2Stan

Bequemlichkeitshalber können Sie eine virtuelle Ubuntu-18.10-Maschine für VirtualBox oder VMWare herunterladen unter `https://www.osboxes.org/ubuntu/#ubuntu-1810-vbox`. Die Downloadgröße beträgt etwa 1,7 Gb; für eine ausgepackte und lauffähige VM wird man jedoch eher 17 Gb freien Speicherplatzes benötigen. Binden Sie die heruntergeladene *.vdi-Datei als Festplatte einer neuer VM ein [1]. Bitte starten Sie die VM [2]. Kopieren Sie anschließend den SQL2Stan-Ordner in das Wurzelverzeichnis (Home) der virtuellen Maschine [3]. Dort können Sie die SQL2Stan-Installationsanweisungen befolgen und die in dieser Arbeit dargelegten Experimente eigenhändig replizieren.

Zur Bequemlichkeit können Sie in der VM die Bildschirmauflösung ändern [4], den Bildschirm-Timeout verlängern und das Tastatur-Layout auf Deutsch umstellen.

[1] Maschine → neu → Typ: Linux, Version: Ubuntu 64 bit → Weiter → Weiter → vorhandene Festplatte verwenden → heruntergeladene *.vdi-Datei auswählen → Erzeugen

[2] **VM-Zugangsdaten**: Username „osboxes.org“ bzw. „root“, Passwort „osboxes.org“.
Troubleshooting: bei Laptops ohne Numpad wird Numlock automatisch aktiviert sein, wodurch man den Passwortbuchstaben o nicht eingeben kann (o wird durch eine 6 ersetzt); schalten Sie in diesem Falle Numlock aus (Fn+F12), um das Passwort problemlos eintippen zu können.

[3] Eine der Möglichkeiten zum Transport von Daten zwischen dem Host und der VM: einen USB-Speicherstick anschließen, die zu kopierenden Daten auf ihn verschieben und den USB-Stick für die VM sichtbar machen (VirtualBox: in der laufenden VM in der oberen Leiste auf Geräte gehen → USB → den Namen des angeschlossenen USB-Sticks anklicken; er wird kurz darauf als Speichermedium in der VM erscheinen.)
Troubleshooting zum USB-Stick-Einsatz: siehe Ticket `https://www.virtualbox.org/ticket/15050`

[4] Ubuntu-VM → Settings → Devices → Displays

A.2. Beispielmodelle in SQL2Stan replizieren

Zur Replikation aller drei in der Arbeit behandelten Beispielmodelle (supervised und unsupervised Naïve Bayes sowie LDA) steht im Ordner „SQL2Stan/example_models“ alles bereit: SQL-Code, Modelldaten und sogar Bash-Skripte zum automatisierten Einlesen von fertigen Modelldaten in die Datenbank. Es ist möglich, eigene Datensätze für diese Beispielmodelle zu generieren, die entsprechenden Generatoren (R-Skripte mit dem Namen source_for_simulated_data.R) befinden sind in den Ordnern der Beispielmodelle. Die Bedienung vom SQL2Stan-Prototypen ist ausführlich erklärt in der Datei README.txt im SQL2Stan-Ordner.

A.3. Umstellung der Inferenzalgorithmen

Stan unterstützt zwei Arten von Inferenzalgorithmen. Stan hat NUTS (No-U-Turn-Sampler) [HG14] – einen genaueren, aber langsameren Markov-Chain-Monte-Carlo-basierten Inferenzalgorithmus (Stans Standard-Inferenzalgorithmus). Außerdem verfügt Stan über ADVI (Automatic Differentiation Variational Inference) [Kuc+17], der deutlich schneller läuft, aber etwas weniger präzise Ergebnisse liefert und noch ein Prototyp ist. SQL2Stan greift auf Stan als ML-Backend zurück; somit hat auch SQL2Stan Zugriff auf beide Inferenzalgorithmen. Ich empfehle ADVI, insofern man einfach daran interessiert ist, das die Inferenz schnell durchläuft.

Umstellung des SQL2Stan-Prototypen auf ADVI Finden (z.B. über Tastenkombination Steuerung+F in Ihrem Texteditor) und ersetzen Sie in der SQL2Stan-Datei „continue_workflow.sh“ das Wort „sample“ durch das Wort „variational“; speichern Sie diese Änderung anschließen ab. Diese Umstellung des Inferenzalgorithmus ist selbstredend umkehrbar auf dem analogen Wege.

B. Extra-Beispiel: Topic Modelling mit LDA

An einem anderen Beispiel, welches zugunsten des kleineren Textumfangs nicht so ausführlich wie das Onlineshop-Klassifikator-Fallbeispiel behandelt wird, möchte ich zeigen, dass die am Naïve-Bayes-Modell erarbeiteten Übersetzungskonzepte auch an einem anderen Bayes'schen Modell in einem anderen Kontext funktionieren. Der Code der Inferenzimplementierung, den SQL2Stan im folgenden Latent-Dirichlet-Allocation-Beispiel generieren wird, weicht strukturell diesmal merklich vom handgeschriebenen Code von Bob Carpenter ab – und bleibt dabei dennoch inhaltlich korrekt und vergleichbar effizient.

B.1. LDA für Information Retrieval

Als Nebenschauplatz für datenbank-gestützte Anwendungen kann man Information Retrieval nennen. An dieser Stelle sollte das Projekt Topic Explorer [Hin19] erwähnt werden. Es ist ein Themenmodell-Browser, der sogar technisch unversierten Nutzern helfen kann, textuelle Daten anhand ihrer Themen zu analysieren. In Hintergrund der Topic-Explorer-Anwendung läuft eine relationale Datenbank (MariaDB); in dieser Datenbank liegen typischerweise vorverarbeitete Textkorpora. Für die Anwendungsdaten soll im Hintergrund automatische Inferenz zwecks Parameterschätzung durchlaufen, als generatives probabilistisches Modell wird dazu Latent Dirichlet Allocation (kurz: LDA) [BNJ03] genutzt.

Die Einbindung von automatischer Inferenz mit Hilfe vom SQL2Stan-Ansatz würde im Rahmen des TopicExplorer-Projekts gut funktionieren. Ein Datenmanagementsystem wird bereits verwendet, und es ist sogar schon relational; die Modellparameterwerte können durch automatische Inferenz anhand der DB-gelagerten Daten approximiert werden. Die Zielfunktion, vorgegeben durch das LDA-Modell, kann in SQL formuliert werden, damit eine sinnvolle Inferenzimplementierung generiert werden kann. Die Nutzung der deklarativen probabilistischen Programmierung am integrierten Datenschema – das, worauf sich SQL2Stan spezialisiert – wäre also bei diesem Projekt definitiv denkbar.

B.2. Modellbeschreibung zu LDA

Ohne Umschweife: in dr Abb. B.1 ist ein Bayes'sches Netz für LDA zu sehen, und im Anschluss die Erklärung der darin genutzten Modellvariablen.

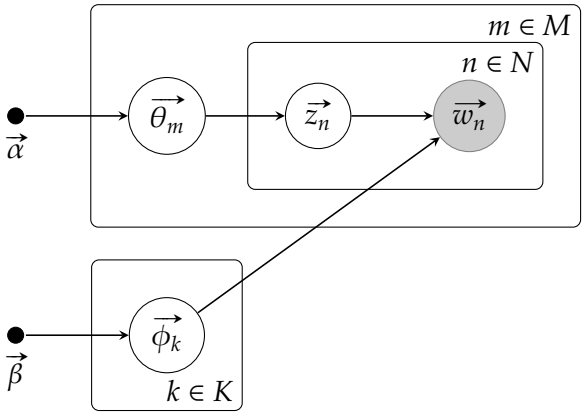


Abbildung B.1.: Bayes'sches Netz als grafische Darstellung des LDA-Modells (Latent Dirichlet Allocation)

Die Modellvariablen aus dem Bayes'schen Netz B.1 lassen sich auf folgende Weise mit den Entitäten einer datenbankgespeicherten Textdokumentensammlung matchen:

- $K \equiv$ Anzahl von Themen/Topics
  - $k \in K$ wird mit „topic_id" in der Tabelle „Topic" gematched.
- $V \equiv$ Größe des Vokabulars
  - $v \in V$ wird mit „word_id" in der Tabelle „Word" gematched.
- $M \equiv$ Größe vom Textkorpus: Anzahl der Dokumente
  - $m \in M$ wird mit „document_id" in der Tabelle „Document" gematched.
- $N \equiv$ Anzahl der Wortinstanzen/Tokens im Textkorpus. Es ist bekannt, aus welchem Dokument ein Token stammt, d.h. m_n lässt sich nachschlagen.
  - $n \in N$ wird mit „token_id" in der Tabelle „Token" gematched.
- $z_n \equiv$ Zuweisung eines Themen-IDs zu einem Token
  - Der Vektor $\vec{z_n}$ ist ein 1-aus-K-Vektor mit einer 1 am Index k, der mit dem Index eines der Topics $k \in K$ korrenspondiert.
  - $z = \{z_1, ..., z_N\},\ z_n \in \{0,1\}^K,\ \forall n \in N : \sum_K z_{nk} = 1$
  - In der modellspezifischen Zuweisung $z_n V$ wird $z \in K$ mit „topic_id" und $n \in N$ mit „token_id" gematched, in beiden Fällen als Fremdschlüssel in der Tabelle „Document_Token_Topic".

- $w_n \equiv$ Zuweisung: Wort-ID $\rightarrow$ Wortinstanz-ID.
  - Im Detail: $\overrightarrow{w_n}$ drückt aus, welches Wort an der Stelle n im Textkorpus steht; die Dokumentzugehörigkeit m_n ist stets implizit gegeben.
  - $w = \{\vec{w_1}, ..., \vec{w_N}\},\ w_n \in \{0,1\}^V,\ \forall n \in N : \sum_v w_{nv} = 1$
  - In der modellspezifischen Zuweisung $w_n V$ wird $w \in V$ mit „word_id“, und $n \in N$ mit „token_id“ gematched, in beiden Fällen als Fremdschlüssel in der Tabelle „Document_Token_Word“.

- $\alpha \equiv$ Hyperparameter, Prior-Vektor für Themenverteilung
  - $\alpha \in \mathbb{R}^K,\ \alpha_k > 0$
  - α kann mit keiner vorhandenen Entität im originalen relationalen Datenbankschema für Textkorpora gematched werden, daher wird im Datenbankschema (zum statistischen Modell, und später auch im integrierten Modell) für diese modellspezifischen Variablenwerte eine Spalte „alpha“ in der Tabelle „Topic“ angelegt.

- $\beta \equiv$ Hyperparameter, Prior-Vektor für Wortverteilung
  - $\beta \in \mathbb{R}^V,\ \beta_k > 0$
  - Für β wird im Datenbankschema für das statistische Modell (und später auch im integrierten Modell) eine Spalte „beta“ in der Tabelle „Topic_Word“ angelegt (analog zu α).

- Parameter θ_m = Themenverteilung für das Dokument m (welche Themen stehen im Dokument mehr im Vordergrund – bzw. sind wahrscheinlicher – und welche weniger).
  - $\forall m \in M : \theta_m \in (0,1)^K,\ \sum_K \theta_{m_k} = 1$
  - Für θ_m wird im Datenbankschema für das statistische Modell (und später auch im integrierten Modell) eine Spalte „theta“ in der Tabelle „Document_Topic“ angelegt. Dabei wird $m \in M$ mit dem Fremdschlüssel „document_id“ in der Tabelle „Document_Topic“ gematched.

- Parameter ϕ_k = Verteilung von Worten innerhalb eines Themas.
  - Hierher gehören die Wahrscheinlichkeiten, mit denen man aus einem themenspezifischen Bag-of-Words ein bestimmtes Wort zieht.
  - $\phi \in (0,1)^{K \times V},\ \forall k \in K : \sum_V \phi_{kv} = 1$
  - Auch für ϕ_k, analog wie bei θ_m weiter oben, wird im Datenbankschema für das statistische Modell (und später auch im integrierten Modell) eine Spalte „phi“ in der Tabelle „Topic_Word“ angelegt. Dabei wird $k \in K$ mit dem Frendschlüssel „topic_id“ in der Tabelle „Topic_Word“ gematched.

B.3. Relationales Datenbankschema für LDA-Anwendungen

Übersprungen werden: die Übersetzung des Bayes'schen Netzes in ein Atomic Plate Model und anschließend in ein Entity-Relationship-Diagramm sowie die Integration des letzteren mit einem Entity-Relationship-Diagramm für vorhandene Daten. Das Ergebnis dieser Schritte sei ein ER-Diagramm, welches alle nötigen LDA-Modellentitäten beinhaltet, und dieses ist als ein minimalistisches ER-Diagramm in der Abb. B.2 zu sehen. Dieses ER-Diagramm lässt sich manuell oder automatisch in ein (integriertes) relationales Modell überführen, das sich in der Create-Table-Notation (SQL) überführen lässt. Die platzintensive SQL-Beschreibung des relationalen Schemas findet der Leser im SQL2Stan-Prototypen im Ordner „SQL2Stan/-example_models/LDA/createTable".

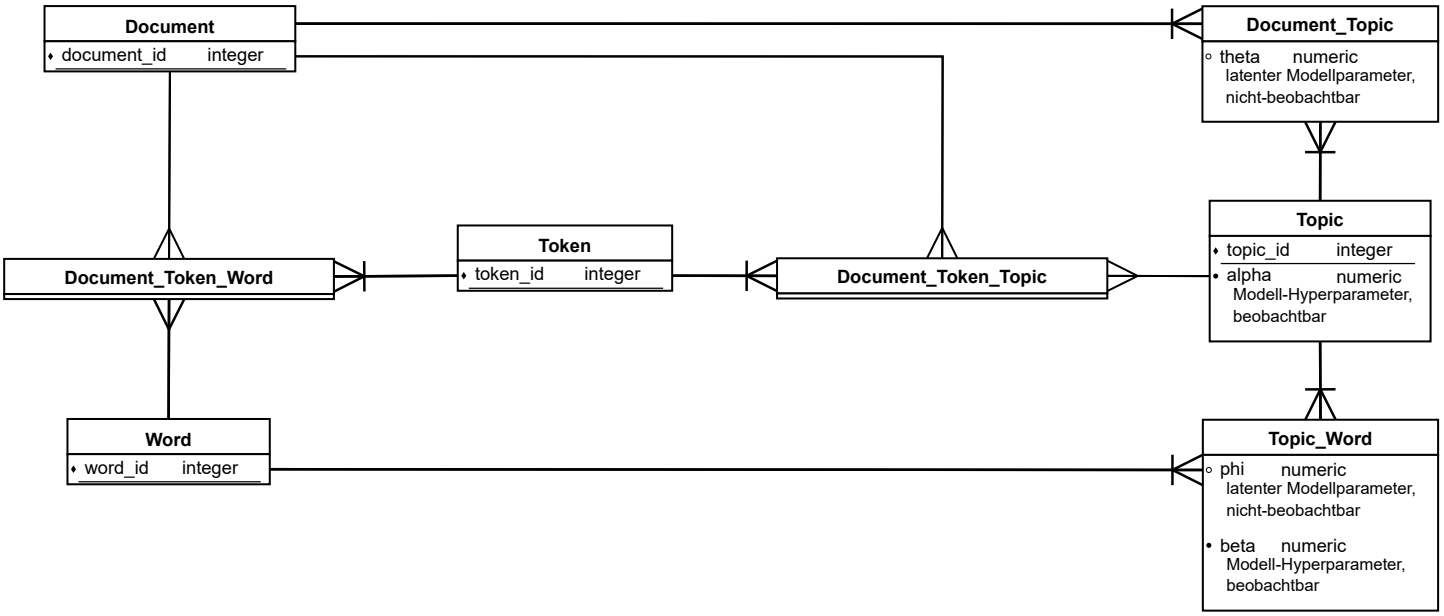


Abbildung B.2.: Entity-Relationship-Diagramm für das integrierte Schema (Entitäten der gegebenen Daten mit LDA-Modellentitäten gematched)

B.4. Von der SQL-Zielfunktion zum probabilistischen Programm

Wir überspringen die nicht-logarithmierte Beschreibung der Wahrscheinlichkeitsfunktion dieses generativen Modells, und schauen uns direkt die log-likelihood-Funktion an. Als Zielfunktion für LDA stellt sich diese Zielfunktion im mathematischen Sinne auf die in der Formel **??** präsentierte Weise zusammen.

$$\begin{aligned}\lg p(w,z,\phi,\theta|\alpha,\beta) = &\sum_{m\in M} \lg \mathit{Dirichlet}(\theta_m|\alpha)\\ &+ \sum_{k\in K} \lg \mathit{Dirichlet}(\phi_k|\beta)\\ &+ \sum_{n\in N} \lg \mathit{Categorical}(z_n|\theta_{m_n})\\ &+ \sum_{n\in N} \lg \mathit{Categorical}(w_n|\phi_{z_n})\end{aligned}$$

Diese Zielfunktion wird vom Nutzer in SQL ausgedrückt; der Leser findet die SQL-Beschreibung der LDA-Zielfunktion im SQL2Stan-Prototypen im Ordner „SQL2Stan/-example_models/LDA/logLikelihood“.

Mit der im SQL2Stan-Dialekt niedergeschriebenen Zielfunktion gäbe es somit eine Abstraktion des Inferenzalgorithmus für LDA. SQL2Stan übersetzt diese in SQL beschriebene Aufgabenspezifikation nach Stan. Der Stan-Code als Ergebnis dieser Übersetzung ist im Listing B.1 dargelegt.

Listing B.1: Stan-Implementierung für LDA, SQL2Stan-generiert

```
data {
int Total__ofTable__document_
      _asNumberOfDistinct__document_id;
int Total__ofTable__word_
      _asNumberOfDistinct__word_id;
int Total__ofTable__token_
      _asNumberOfDistinct__token_id;
int Total__ofTable__topic_
      _asNumberOfDistinct__topic_id;
int Total__ofTable__document_token_word_
      _asNumberOfDistinct__token_id;
int Total__ofTable__document_token_topic_
      _asNumberOfDistinct__token_id;
int Total__ofTable__document_topic_
      _asNumberOfDistinct__document_id;
int Total__ofTable__document_topic_
      _asNumberOfDistinct__topic_id;
int Total__ofTable__topic_word_
      _asNumberOfDistinct__topic_id;
int Total__ofTable__topic_word_
      _asNumberOfDistinct__word_id;
int document_id__asColumnOfTable__document[
      Total__ofTable__document_
            _asNumberOfDistinct__document_id];
int word_id__asColumnOfTable__word[
      Total__ofTable__word_
            _asNumberOfDistinct__word_id];
int token_id__asColumnOfTable__token[
      Total__ofTable__token_
```

```
            _asNumberOfDistinct__token_id];
int topic_id__asColumnOfTable__topic[
      Total__ofTable__topic_
            _asNumberOfDistinct__topic_id];
int word_id__asColumnOfTable__document_token_word[
      Total__ofTable__document_token_word_
            _asNumberOfDistinct__token_id];
int token_id__asColumnOfTable__document_token_word[
      Total__ofTable__document_token_word_
            _asNumberOfDistinct__token_id];
int document_id__asColumnOfTable__document_token_word[
      Total__ofTable__document_token_word_
            _asNumberOfDistinct__token_id];
int token_id__asColumnOfTable__document_token_topic[
      Total__ofTable__document_token_topic_
            _asNumberOfDistinct__token_id];
int document_id__asColumnOfTable__document_token_topic[
      Total__ofTable__document_token_topic_
            _asNumberOfDistinct__token_id];
int topic_id__asColumnOfTable__document_topic[
      Total__ofTable__document_topic_
            _asNumberOfDistinct__document_id,
      Total__ofTable__document_topic_
            _asNumberOfDistinct__topic_id];
int document_id__asColumnOfTable__document_topic[
      Total__ofTable__document_topic_
            _asNumberOfDistinct__document_id,
      Total__ofTable__document_topic_
            _asNumberOfDistinct__topic_id];
int topic_id__asColumnOfTable__topic_word[
      Total__ofTable__topic_word_
            _asNumberOfDistinct__topic_id,
      Total__ofTable__topic_word_
            _asNumberOfDistinct__word_id];
int word_id__asColumnOfTable__topic_word[
      Total__ofTable__topic_word_
            _asNumberOfDistinct__topic_id,
      Total__ofTable__topic_word_
            _asNumberOfDistinct__word_id];
vector[
      Total__ofTable__topic_
            _asNumberOfDistinct__topic_id
      ] alpha__asColumnOfTable__topic;
vector[
      Total__ofTable__topic_word_
            _asNumberOfDistinct__word_id
      ] beta__asColumnOfTable__topic_word[
            Total__ofTable__topic_word_
                  _asNumberOfDistinct__topic_id];
```

```
}

parameters {
simplex[
      Total__ofTable__document_topic_
            _asNumberOfDistinct__topic_id
       ] theta__asColumnOfTable__document_topic[
             Total__ofTable__document_topic_
                   _asNumberOfDistinct__document_id];
simplex[
      Total__ofTable__topic_word_
            _asNumberOfDistinct__word_id
       ] phi__asColumnOfTable__topic_word[
                  Total__ofTable__topic_word_
                        _asNumberOfDistinct__topic_id];
}

model {
real gamma__for__topic_id__asColumnOfTable__topic[
      Total__ofTable__document_token_topic_
            _asNumberOfDistinct__token_id,
      Total__ofTable__topic_
            _asNumberOfDistinct__topic_id];

for (a in 1:Total__ofTable__document_
      _asNumberOfDistinct__document_id) {
target += dirichlet_lpdf(
      theta__asColumnOfTable__document_topic[a]
    | alpha__asColumnOfTable__topic);
}

for (a in 1:Total__ofTable__topic_
      _asNumberOfDistinct__topic_id) {
target += dirichlet_lpdf(
      phi__asColumnOfTable__topic_word[a]
    | beta__asColumnOfTable__topic_word[a]);
}

for (a in 1:Total__ofTable__token_
      _asNumberOfDistinct__token_id) {
for (b in 1:Total__ofTable__topic_
      _asNumberOfDistinct__topic_id) {
gamma__for__topic_id__asColumnOfTable__topic[a,b] =
categorical_lpmf(
      b
    | theta__asColumnOfTable__document_topic[
            document_id__asColumnOfTable__document_token_word
                              [a]]);
}
```

```
}

for (a in 1:Total__ofTable__token_
      _asNumberOfDistinct__token_id) {
for (b in 1:Total__ofTable__topic_
      _asNumberOfDistinct__topic_id) {
gamma__for__topic_id__asColumnOfTable__topic[a,b] +=
categorical_lpmf(
      word_id__asColumnOfTable__document_token_word[a]
    | phi__asColumnOfTable__topic_word[b]);
}
}

for (a in 1:Total__ofTable__document_token_topic_
      _asNumberOfDistinct__token_id) {
target += log_sum_exp(
      gamma__for__topic_id__asColumnOfTable__topic[a]);
}
}
```

B.4.1. Codevergleich: SQL2Stan-generiert versus handgeschrieben

Den SQL2Stan-generierten Stan-Code aus dem Listing B.1 kann man mit dem Stan-Code für LDA D.3 vergleichen, den Bob Carpenter implementiert hat. Lässt man beim Code-Vergleich die Namensunterschiede der Arrays/Vektoren und Funktionen außer Acht, fallen einem folgende strukturelle Unterschiede auf:

- Die Art, wie der diskrete latente Parameter z_n in die Zielfunktion eingebunden wird, ist anders:
  - Bei B. Carpenter wird die Modellvariable z nirgendwo genannt, ihre Anwesenheit ist implizit. Der gamma-Array ist um eine Dimension kleiner und wird in jedem n-Durchlauf neu erstellt. Noch in der gleichen Schleife über N, am Ende jeder Iteration werden die Werte des gamma-Arrays in die LogSumExp-Funktion gepackt, und das Ergebnis wird zum Zielfunktionswert addiert.
  - SQL2Stan erstellt den gamma-Array für z direkt am Anfang des model-Blocks und benennt ihn direkt mit Angabe der dazugehörigen Tabelle und Spalte Außerdem ist dieser gamma-Array um eine Dimension größer als sein handgeschriebener Pendant. Das Einfließen vom Ergebnis der LogSumExp-Funktion auf den gamma-Arraywerten in die Zielfunktion wird in einer extra Schleife am Ende des model-Blocks eingebettet.
  - Die Vorgehensweise von B. Carpenter ist vermutlich geschickter, weil er das Aufrufen vom gamma-Array auf der rechten Seite der Zuweisung vermeidet. Dadurch werden die sogenannten *deep copies* vermieden, die die Effizienz negativ beeinflussen können.

Auch im SQL2Stan-generierten Code findet man keine gamma-Array-Aufrufe rechts von der Zuweisung; der Compiler, der darauf achten soll und sonst eine deep-copy-Warnmeldung ausgibt, schweigt. Mir ist jedoch nicht bekannt, inwiefern in Stan die Zuweisung $a = a + b$ der Zuweisung $a += b$ gleicht. Im schlimmsten und nicht unwahrscheinlichen Falle, falls sie vom Compiler als gleich betrachtet werden, werden ineffizienterweise *deep copies* zur Vermeidung von Aliasing angelegt [1].

- Nutzung der log-categorical-Verteilungsfunktion:
  - B. Carpenter verzichtet auf die Nutzung der Funktionen *categorical* bzw. *categorical_lpmf* und nimmt stattdessen eine andere Schreibweise.
  - Im von SQL2Stan generierten Code wird die Logarithmierung der Wahrscheinlichkeiten für kategorische Verteilungen durch die Funktion *categorical_lpmf* bewerkstelligt.
- Die letzten zwei Summanden der Zielfunktion, mathematisch formuliert als $\sum_{n \in N} \lg \mathit{Categorical}(z_n|\theta_{m_n}) + \sum_{n \in N} \lg \mathit{Categorical}(w_n|\phi_{z_n})$:
  - Bob Carpenter fasst die Summe der beiden Summanden in einer einzigen Schleife zusammen
  - Der Beitrag der letzten beiden Summanden zum Zielfunktionswert wird bei SQL2Stan nicht zusammengefasst: sie bleiben als zwei einzelne Summanden erhalten, so wie in der mathematischen Beschreibung der Zielfunktion.

B.4.2. Laufzeitvergleich

Dennoch definieren sowohl B. Carpenters Code für LDA D.3 als auch die SQL2Stan-generierte LDA-Implementierung (Listing B.1) funktionell dasselbe, und sind bei einem kleinen Beispiel vergleichbar in ihrer Laufzeit. Dies bestätigen die Ergebnisse der LDA-Parameterwerte-Schätzung auf Beispieldaten [Car13b] von B. Carpenter, die von ihm mit folgenden Größen generiert wurden:

- Anzahl der Themen/Topics $K = 2$
- Größe des Vokabulars $V = 5$
- Anzahl der Dokumente (Größe vom Textkorpus) $M = 25$
- Hyperparameter-Vektor $\vec{\alpha}$ (K-groß) = (0.5, 0.5)
- Hyperparameter-Vektor $\vec{\beta}$ (V-groß) = (0.2, 0.2, 0.2, 0.2, 0.2)

[1] In einem anderen Übersetzer nach Stan (brms: aus R in Stan) wird man unter Umständen mit dem gleichen Problem konfrontiert, siehe `https://discourse.mc-stan.org/t/deep-copy-issue/6036/3`. Dort weist der generierte Stan-Code scheinbar eine Variable auf, die sowohl auf der linken als auch auf der rechten Seite der Zuweisung erscheint.

In der Tabelle B.1 werden die Laufzeiten sowie zehn Schätzwerte für die latenten Modellvariable $\vec{\phi}$ [2] präsentiert. Weitere 50 geschätzten Werte für den Modellparameter $\vec{\theta}$ liegen der Arbeit bei (im Ordner „Gegenueberstellung von Inferenzergebnissen“).

Tabelle B.1.: Gegenüberstellung von ausgewählten Schätzungsergebnissen sowie Laufzeiten: handgeschriebener versus SQL2Stan-generierter Stan-Code. Klassen-ID-Permutationen (bei jedem Inferenzdurchlauf eine andere ID für dieselbe Klasse) berücksichtigt.

	Latent Dirichlet Allocation	
	Carpenter	**SQL2Stan**
t(Warm-up), s	4.519	6.361
t(Sampling), s	4.498	3.935
t(Gesamt), s	9.017	10.296
phi[1,1]	0.0067	0.0068
phi[1,2]	0.0027	0.0026
phi[1,3]	0.28	0.28
phi[1,4]	0.41	0.41
phi[1,5]	0.3	0.31
phi[2,1]	0.39	0.4
phi[2,2]	0.34	0.33
phi[2,3]	0.26	0.26
phi[2,4]	0.0033	0.0036
phi[2,5]	0.0092	0.0085

Das genutzte Inferenzverfahren ist No-U-Turn-Sampler, standardmäßig für Stan v2.17. NUTS ist eine MCMC-Sampling-gestützte Inferenzmethode, weshalb jeder Inferenzvorgang immer leicht unterschiedliche Schätzungswerte liefert.

Fazit Die Schätzungen beider LDA-Implementierungen sind beim gleichen Datensatz sehr ähnlich. Die Laufzeiten beider LDA-Implementierungen sind bei dem gegebenen und recht kleinen Datensatz vergleichbar. Stan-Entwickler (u.a. B. Carpenter) halten LDA und LDA-ähnliche Modelle als ungeeignet für Bayes'sche Inferenz, und setzen daher keinen Schwerpunkt darauf [Car17]. LDa-ähnliche probabilistische Modelle lassen sich in Stan zwar beschreiben und inferieren, jedoch ohne Robustheitsgarantie [Car17]. Das erklärt, warum Bob Carpenter einen sehr kleinen Beispiel-Datensatz für LDA anbietet: es soll einfach zeigen, dass LDA in Stan modellierbar und inferierbar ist; es soll nicht zeigen, dass es robust ist (was es nicht ist). Insofern genügt die Vergleichbarkeit beider LDA-Implementierungen an einem kleinen Datensatz, denn an einem großen Datensatz würde man Stan als

[2] $\phi_{k,v}$ mit k als Thema und v als Wort aus dem Vokabular

Inferenz-Backend sowieso nicht nehmen wollen. Ich gehe aufgrund der *deep-copies*-bezogenen Bedenken dennoch davon aus, dass die Carperter-LDA-Implementierung bei größeren Datensätzen etwas schneller sein könnte (Größenordnung unbekannt und aufgrund der LDA-in-Stan-Problematik irrelevant). Es bleibt offen, wie das SQL2Stan-typische Auftreten der gamma-Array-Aufrufe auf der rechten Seite von Zuweisungen zu unterbinden ist, sodass die Codestruktur dennoch generisch bleibt.

C. Inferenzdiagnose

Abgesehen vom Modellserving nach der durchgeführten Bayes'schen Inferenz kann ein fortgeschrittener Datenenthusiast (oder ein Statistiker) mehr über den Ablauf vom Inferenzprozess wissen wollen, um möglicherweise Änderungen am Modell, an den Einstellungen des Inferenzalgorithmus oder an den Modellparametern vorzunehmen. Bis auf die inferierten Modellvariablenwerte blendet SQL2Stan standardmäßig alle Inferenzausgabendetails aus. Es wäre dennoch möglich, SQL2Stan so zu erweitern, dass man die in Stan eingebaute Inferenzprozess-Analyse (Inferenzdiagnose) optional abfragen kann.

Die Inferenzausgabe basiert auf den Stichproben, die im Laufe des Markov-Chain-Monte-Carlo-Sampling (die Standard-Inferenzmethode von Stan) gezogen wurden. Man kann sie mit Stan-interner Funktionalität [1] auf potenzielle Probleme untersuchen [Sta19c]:

- Divergente Übergänge (orig.: divergent transitions): wenn das simulierte Hamilton-System und das echte auseinanderstreben. Zum Kontext [Sta18]: Stan – unser Backend für die automatische Inferenz – nutzt HMC (Hamiltonian-Monte-Carlo bzw. Hybrid-Monte-Carlo-Algorithmus), um die Zielverteilung zu untersuchen. Die Zielverteilung ist gegeben durch die Zielfunktion (Posterior, zu finden im model-Block des Stan-Codes) und die gegebenen Daten. HMC untersucht die Zielverteilung, indem es die Entwicklung eines Hamilton-Systems simuliert – eines dynamischen Systems, welches durch die Hamiltonsche Mechanik reguliert wird. Die Übergänge sind Schritte, mit denen das simulierte Hamilton-System evolutioniert. Wenn es viele divergente Übergänge gibt, kann man versuchen, diese Übergänge/Schritte kleiner zu machen (z.B. delta auf 0.99 statt 0.8 setzen), doch dann wird die Laufzeit deutlich größer. Mehr Input zur dieser Problematik der divergenten Übergänge ist in [Bet17b] zu finden.

- Zu kleine E-BFMI-Werte (Begriffserklärung: [Bet17a, S. 44]): während der Adaptationsphase der Markov-Ketten ist etwas schiefgelaufen, und die Markov-Letten können die Zielverteilung (den Posterior) nicht effizient abtasten/absondieren [Sta18]. Reparametrisierung des Modells oder die Erhöhung der Anzahl von Sampling-Iterationen könnten hilfreich sein.

[1] Stan-Skripte für Analyse der MCMC-Ausgabe: stansummary, diagnose.

- Zu kleine effektive Stichprobengröße (orig. low effective sample sizes, low ESS) [Per19]: effektive Stichprobengröße ist die errechnete Anzahl von unabhängigen Stichproben, die die gleiche Genauigkeit besäßen wie die korrelierten MCMC-Stichproben. ESS hängt mit der Effizienz der MCMC-Algorithmen zusammen.
- Zu hohe $\hat{R}$-Werte: bezieht sich auf die Ergebnisse der split-$\hat{R}$ [2] multi-chain [3] Konvergenzdiagnostik nach Gelman/Rubin. Zur Orientierung: $\hat{R}$ soll möglichst nah an einer 1 sein.

Zusätzliche Informationen über die geschätzten Werte (wie Standardfehler, Standardvarianz, Quantile usw.) sind ebenfalls abrufbar. Aktuell werden solche Details vor dem Nutzer einfach ausgeblendet. Mit geringem Aufwand können die Informationen zur statistischen Diagnostik als zusätzliche textuelle SQL2Stan-Ausgabe hinzugefügt werden.

[2] Markov-Kette wird zweigeteilt, $\hat{R}$ wird für beide Hälften berechnet.
[3] D.h. für alle simulierten Markov-Ketten.

D. Anhang: Stan-Code von Bob Carpenter

D.1. Supervised Naïve Bayes [Car13c]

```
data {
  // training data
  int<lower=1> K;               // num topics
  int<lower=1> V;               // num words
  int<lower=0> M;               // num docs
  int<lower=0> N;               // total word instances
  int<lower=1,upper=K> z[M];    // topic for doc m
  int<lower=1,upper=V> w[N];    // word n
  int<lower=1,upper=M> doc[N];  // doc ID for word n
  // hyperparameters
  vector<lower=0>[K] alpha;     // topic prior
  vector<lower=0>[V] beta;      // word prior
}
parameters {
  simplex[K] theta;   // topic prevalence
  simplex[V] phi[K];  // word dist for topic k
}
model {
  // priors
  theta ~ dirichlet(alpha);
  for (k in 1:K)
    phi[k] ~ dirichlet(beta);
  // likelihood, including latent category
  for (m in 1:M)
    z[m] ~ categorical(theta);
  for (n in 1:N)
    w[n] ~ categorical(phi[z[doc[n]]]);
}
```

D.2. Unsupervised Naïve Bayes [Car13d]

```
data {
  int<lower=2> K;               // num topics
  int<lower=2> V;               // num words
  int<lower=1> M;               // num docs
  int<lower=1> N;               // total word instances
  int<lower=1,upper=V> w[N];    // word n
  int<lower=1,upper=M> doc[N];  // doc ID for word n
  vector<lower=0>[K] alpha;     // topic prior
  vector<lower=0>[V] beta;      // word prior
}
parameters {
  simplex[K] theta;   // topic prevalence
  simplex[V] phi[K];  // word dist for topic k
}
model {
  real gamma[M,K];

  theta ~ dirichlet(alpha);
  for (k in 1:K)
    phi[k] ~ dirichlet(beta);

   for (m in 1:M)
     for (k in 1:K)
       gamma[m,k] <- categorical_log(k,theta);
   for (n in 1:N)
     for (k in 1:K)
       gamma[doc[n],k] <- gamma[doc[n],k]
                          + categorical_log(w[n],phi[k]);
   for (m in 1:M)
     increment_log_prob(log_sum_exp(gamma[m]));

   // to normalize s.t. gamma[m,k] = log Pr[Z2[m] = k|data]
   // gamma[m] <- gamma[m] - log_sum_exp(gamma[m]);
}
```

D.3. Latent Dirichlet Allocation (LDA) [Car13a]

```
data {
  int<lower=2> K;               // num topics
  int<lower=2> V;               // num words
  int<lower=1> M;               // num docs
  int<lower=1> N;               // total word instances
  int<lower=1,upper=V> w[N];    // word n
  int<lower=1,upper=M> doc[N];  // doc ID for word n
  vector<lower=0>[K] alpha;     // topic prior
  vector<lower=0>[V] beta;      // word prior
}
parameters {
  simplex[K] theta[M];   // topic dist for doc m
  simplex[V] phi[K];     // word dist for topic k
}
model {
  for (m in 1:M)
    theta[m] ~ dirichlet(alpha);  // prior
  for (k in 1:K)
    phi[k] ~ dirichlet(beta);     // prior
  for (n in 1:N) {
    real gamma[K];
    for (k in 1:K)
      gamma[k] <- log(theta[doc[n],k]) + log(phi[k,w[n]]);
    increment_log_prob(log_sum_exp(gamma));  // likelihood
  }
}
```

E. Anhang: ausführlicher Related-Work-Bericht

Die Thematik der Zusammenführung vom Datenmanagement und Machine Learning bietet einige Einblicke in wissenschaftliche Projekte, die dafür unterschiedliche Ansätze vorschlagen. Deklarativität, die damit verbundene automatische (Code-)Optimierung sowie SQL werden in den relevanten Veröffentlichungen oft erwähnt.

E.1. SQL2Stan im Kontext anderer Projekte

Verfahren zur statistischen Inferenz für Machine-Learning-Modelle stellen einen relevanten Teil von ML dar. Projekte, die Datenmanagement mit ML zusammenführen, bieten einerseits eher imperative und andererseits eher deklarative Nutzerschnittstellen an [1]. Bei der Wahl eines Programmierparadigmas richtet man sich nach der Zielgruppe. Projekte mit imperativen Programmiersprachen (z.B. TensorFlow Probability [Ten19a; Ten19b]) zeichnen sich dadurch aus, dass sie weniger high-level-Abstraktionen bieten, dafür aber eine breitere Funktionalität haben. Projekte mit deklarativen Nutzerschnittstellen verzichten bewusst auf imperative Sprachstrukturen und lassen sich in der Regel leichter bedienen. Der Nutzer eines deklarativen Interfaces muss sich weniger mit den Implementierungsdetails auseinandersetzen. Die Tendenz zur Nutzerfreundlichkeit, vor allem durch Deklarativität im nutzergeschriebenen Code, lässt sich an vielen Beispielen erkennen.

In drei Sätzen: Kontextpositionierung von SQL2Stan SQL2Stan ist ein System, das 1) mit Hilfe von SQL sowohl das Datenmanagement als auch Bayes'sche Inferenzalgorithmik abstrahiert, 2) das Spezifizieren von eigenen (auch hierarchischen) generativen Modellen durch weniger programmierfachkundige Nutzer in SQL unterstützt und 3) ein einfaches relationales Datenbankschema als Schnittstelle für Inferenz-Ein- und Ausgabe propagiert, die mit normalen SQL-Queries abgefragt werden kann. Das relationale Schema (siehe Kap. 3.2) kann manuell aus einer Darstellung des Bayes'schen Modells abgeleitet werden. Die Modellspezifikation über eine Zielfunktion und das Definieren eines relationalen Schemas sind bewusst getrennt, um Relationen als Schnittstelle zur Interaktion mit Modelldaten möglichst übersichtlich zu gestalten.

[1] Die Formulierung „eher“ deutet daraufhin, dass manche Interface-Sprachen durchaus Anleihen aus einem gegensätzlichen Paradigma machen können. Als Beispiel dient SimSQL, wo der SQL-Dialekt zur probabilistischen Programmierung imperative (zustandsorientierte) Merkmale besitzt.

Tabelle E.1.: Übersicht von Machine-Learning-Systemen, geteilt nach Anbindung der Datenmanagementfunktionalität sowie nach Möglichkeit des eigenständigen Modellebauens durch den Programmierer. Aufgeführte Projekte mit einem Stern-Präfix (*) vor ihrem Namen legen viel Wert auf Deklarativität.

Datenmanagement-funktionalität \ Schnittstelle für ML-Modelle:	Bibliothek/ Blackbox	Modellspezifikation
DM-Integration nicht vorgesehen	R (MASS [Rip+19]) Python (Scikit-learn [Bui+13]) Stata [Sta19e; Ham12] Keras [Cho+15]	R (brms [Bür19; Bür16]) Python (PyMC3 [PyM19a], pyro [Ube19], Bambi [YW16]) Stata [Sta19e; Ham12] Keras [Cho+15] Infer.net [NET19a; WW11] WebPPL [Com19; Möb18b] Stan [Sta19b; Car+17] Church [Goo+12] IBAL [Pfe07] Figaro [Pfe09] * Tabular [Gor+14]
DM integriert / integrierbar	* MADlib [MAD19] * BayesDB [MIT19] * ease.ml [DS319; Li+17a] Rafiki [Chu19; Wan+18] Spark MLlib [The19g; Men+16]	* SimSQL [Cai+19; Cai+13] * **SQL2Stan** (diese Arbeit) * MLog [DS317; Li+17b] TensorFlow Probability [Ten19a; Ten19b] Python (PyMC4 [PyM19b; PyM18], to be released) Agrios [LM12] SystemML [The19a; Boe+16]

Systeme, die zum näheren Vergleich ausgewählt wurden Es gibt viele Systeme, die Machine Learning bzw. Datenanalyse in den Datenbankkontext einbinden (siehe Zeile „DM integriert“) in der Tabelle **??**). Zum Vergleichen der Konzepte mit SQL2Stan sollen allerdings nur diejenigen Tools genommen werden, die 1) über die Funktionalität zur Bayes'schen Inferenz verfügen, und 2) diese auf relationalen Daten mit Hilfe einer deklarativen Datenbanksprache umsetzen. Zu den mit SQL2Stan zu vergleichenden Projekten gehören daher MADlib [MAD19], BayesDB [MIT19], SimSQL [Cai+19; Cai+13] – allesamt Inferenzsysteme, die über ein SQL-Interface verfügen. Hinzu kommt das Inferenztool Tabular [Gor+14]. Tabular wird nicht über SQL bedient, sondern über das Annotieren relationaler Schemata für Spreadsheets, was sich konzeptionell auch im Allgemeinen auf relationale Schemata (eine Form der Datenbankabstraktion) übertragen lässt. Tabular-Lösungen sich schlecht skalierbar (eignen sich nicht für Big Data) und das System hat kein angebundenes DMS, doch die Tabular-Sprache erlaubt theoretisch eine skalierbare Cloud-Infrastruktur als DMS-Backend (nach dem Vorbild von `keikao.io`). Zuletzt kann man begrenzt das System MLog [DS317; Li+17b] mit SQL2Stan vergleichen. MLog ist eine ML-Bibliothek für ein Array-DBMS namens SciDB [Sto+13]. Diese Bibliothek ist im SQL2Stan-Kontext relevant und interessant, weil es in seinem deklarativen Interface auf relationalen Datenbankschemata basiert, sprachliche Anleihen aus SQL macht und diese mit einer anderen Datenbanksprache Datalog verbindet. Dennoch ist MLog ein auf Tensoren fixiertes ML-Werkzeug, einsetzbar hauptsächlich für komplexe ML-Modellierung (u.a. neuronale Netzwerke, Deep Learning [Li+17b, S. 1936]), wodurch es sich funktionell deutlich von SQL2Stan abspaltet. Nach meinem Verständnis der dazu gelesenen Veröffentlichungen sollte MLog auch für Bayes'sche Inferenz sprachlich und technisch offen sein, was sich allerdings anhand der sehr knapp gehaltenen MLog-Dokumentation weder beweisen noch widerlegen lässt.

Spezifikation eigener Modelle im Datenbankkontext Nur wenige ML-Projekte erlauben ihren Nutzern leichte Spezifikation eigener Bayes'schen ML-Modelle zwecks automatischer Inferenz definieren. Zu diesen wenigen Systemen gehören SQL-Systeme SimSQL und SQL2Stan, ein relationales Spreadsheet-System Tabular und potenziell auch ein Datalog-SQL-Mischling MLog. Die über SQL bedienbare Projekte MADlib und BayesDB verfolgen hingegen das Konzept „maschinelles Lernen als Blackbox“. Mit anderen Worten: mit ML als Blackbox wird keine einfache Schnittstelle zum Bauen eigener ML-Modelle bereitgestellt. MADlib sieht für den durchschnittlichen Nutzer keine Möglichkeit vor, eigene Bayes'sche Modelle zu definieren. Einem MADlib-Nutzer mit ausreichend Programmiererfahrung und Motivation steht jedoch frei, neue Modelle eigenhändig über zusätzliche MADlib-Module in C++ zu implementieren. Für alle anderen Nutzer bietet MADlib vielmehr eine Funktionsbibliothek an mit einer Reihe an hilfreichen Datenanalyse-Werkzeugen und vorimplementierten Bayes'schen Modellen. In einem anderen System (BayesDB) werden Modelle automatisch durch das System generiert. Dem BayesDB-Nutzer wird nur eine deklarative Schnisstelle angeboten, mit der er die au-

tomatische Modellgenerierung beeinflussen kann. Mit anderen Worten: ein BayesDB-Nutzer kann kein konkretes Bayes'sches Modell (z.B. LDA oder naïve Bayes) nachimplementieren.

Im Kontrast zu den Einschränkungen von MADlib und BayesDB und MADlib erscheint das unkomplizierte Definieren eigener (Bayes'scher) ML-Modelle sehr attraktiv. Das leichte Definieren eigener statistischer Modelle begünstigt und beschleunigt das Entwerfen neuer Modelle und das Experimentieren mit bestehenden Modellen. So können modellspezifische Prototypen schnell zum Laufen gebracht werden. Außerdem kann man somit viele bereits veröffentlichte Bayes'sche Modelle aus unterschiedlichsten Domänen leicht nachimplementieren. Bayes'sche Modelle fanden Einzug in unterscheidlichste Bereiche wie Umweltforschung [Cla05], Astrophysik [HSI17] [Kag12], Medizin und Bioinformatik [HDR10][Bro+03], Investment und Banken [SS01] [SS00], „Hacking" [Dav18]. Für eine einzelne Domäne gibt es wahrscheinlich nicht sehr viele Bayes'sche Modelle. Doch die Gesamtmenge aller domänenspezifischen Bayes'schen Modelle kann beachtlich sein, und es ist zu erwarten, dass im Laufe der Forschungsarbeiten weitere Modelle hinzukommen. Daher ist es sinnvoll, den Domänenspezialisten die statistische Inferenz mit flexibel und leicht definierbaren Bayes'schen Modellen zu ermöglichen. Umso sinnvoller ist es, wenn diese Funktionalität darüber hinaus an ein skalierbares Datenmanagement-System angebunden ist und in einer ohnehin bekannten Datenmanagement-Sprache wie SQL bedient wird.

Die Spezifikation eines eigenen Bayes'schen Modells ist im Prinzip die Beschreibung der strukturellen Zusammensetzung von einem statistischen Modell. SimSQL und Tabular lassen ihre Nutzer ihre Modelle direkt innerhalb der Datenbankschemata spezifizieren. Mit anderen Worten: probabilistisch programmiert wird direkt in der Tabellenbeschreibung. Das machen MLog und SQL2Stan anders. Bei MLog werden auf einer relationale Weise (ähnlich zu Create-Table-Anweisungen) Tensoren angelegt, mit denen dann in weiteren Programmteilen über tensorale Datalog-ähnliche Regeln programmiert wird. Das relationale Schema und die Beschreibung der Modellstruktur (ergo das probabilistische Programm) werden in SQL2Stan ebenso getrennt behandelt. Die SQL2Stan-Modellspezifikation befindet sich nicht in relationalen Schema, sondern sie besteht aus einem zusätzlichen SQL-Statement abseits der Create-Table-Statements. In SQL2Stan beinhalten die Tabellendefinitionen nur Spaltentypen (Angaben, ob Modelldaten einer Spalte beobachtet oder unbeobachtet) sowie modellspezifische Constraints zu den Spalten (konkret: aus welchen Spalten besteht der Tabellenschlüssel, Reihenfolge dieser Schlüsselspalten, Summe-der-Werte-ergibt-1-Bedingungen für Spalten).

Relationalschematische API zu beliebigen Anwendungen SimSQL, Tabular, SQL2Stan und MLog sehen sich gewiss als Datenanalyse-Werkzeuge, die auf der relationalen Sichtweise basieren. Die relationale Sicht beruht sich für die ersten drei Systeme auf klassischen Datenbanktabellen; bei MLog ist eine Relation bzw. eine Tabelle einfach ein Tensor. In allen Fällen impliziert die relationale Herange-

hensweise eines: man arbeitet mit strukturierten oder strukturierbaren Daten. Alle vier Systeme verfügen auf ihre eigene Art über ein integriertes relationales Schema, wo alle Entitäten des ML-Modells sich in relationalen Schemaentitäten wiederfinden lassen. Der Begriff „integriert“ bedeutet in diesem Zusammenhang, dass das relationale Schema für gegebene Daten mit dem relationalen Schema des Modells verschmolzen wird. Bei einem solchen integrierten Datenbank-Schema wird die Ein- und Ausgabe der automatischen Inferenz in einer gemeinsamen Darstellung für das Datenmanagement-System untergebracht. Das integrierte Schema dient als API für beliebige Anwendungen, die die Inferenzergebnisse verwerten.

SimSQL handhabt die Bereitstellung einer solchen API auf eine spezielle Weise. Das Datenbankschema für die gegebenen Daten ist in SimSQL fest. Sie wird zwecks Inferenz um zusätzliche stochastische Tabellen erweitert (ergo: um eine Menge evolutionierender Tabellen für jedes zufallsbedingte Modellparameter). Die inferenzalgorithmisch bedingte Evolution der stochastischen Tabellen läuft in Schleifen ab, daher werden diese Tabellen mit Indizes versehen (*Tabelle[0]* zum Initialisieren der Werte, *Tabelle[i]* zum wiederholten Aktualisieren der Tabellenwerte; der Programmierer braucht den i-Wert, um die Tabellenwerte abzufragen). Im Rumpf der Create-Table-Anweisungen (Tabellendefinitionen) gibt der Nutzer den probabilistischen SQL-Programmcode an. SQL2Stan, Tabular und MLog kommen hingegen ohne Tabellen- bzw. Tensorversionierung aus. Tabular und SQL2Stan unterhalten in ihrem Code keine indexierten Zustände von Tabellen. Eine bestimmte Tabelle im integrierten Schema gibt es bei Tabular und SQL2Stan immer nur einmal. Ein versteckter, zu schätzender Modellparameter ist außerdem keine extra Tabelle, sondern eine Spalte in einer bereits existierenden Tabelle. Ähnlich wie bei SQL2Stan und Tabular verhält es sich auch bei MLog: ein Tensor entspricht einer Relation bzw. einer Tabelle. MLog unterstützt sogar eine Operation zum Überführen relationaler Tabellen in Tensoren [Li+17b, S. 1935].

Das Spezifizieren eine Modells erfolgt in SimSQL und Tabular direkt in den Definitionen relationaler Tabellen. Bei SQL2Stan und MLog hingegen werden ML-Modell*zusätzlich* zu den Relationen formuliert, und nicht innerhalb von Relationen/Tabellen. In SimSQL legt der Nutzer mit Hilfe von stochastischen Tabellen fest, wie ein von ihm gewähltes generatives Modell strukturiert ist. Die Modellstruktur über evolutionierende stochastische Tabellen zu spezifizieren ist sicherlich eine ausgesprochen Bayes'sche (siehe Kapitel 3.1.1) Art der probabilistischen Programmierung in SQL. Man stellt sich vor, wie ein Sampling-Verfahren die in das Modell involvierten Werte zu aktualisieren hätte, und schreibt dies im Rumpf der stochastischen Tabelle auf [Cai+13, S. 40]. Diese Herangehensweise ist leider sehr auf eine Klasse von Inferenzalgorithmen (MCMC via Gibbs-Sampling [Wal04]) fixiert, und zeigt sich somit inkompatibel mit anderen Inferenzalgorithmen (z.B. VI [Hof+13] oder Kombination aus VI und MCMC [SKW15]). SQL2Stan beruft sich hingegen auf eine der Stärken von Stan, und kann bei unveränderter Modellspezifikation problemlos zwischen MCMC und VI umschalten (MCMC+VI wird somit zumindest rein sprachlich unterstützt). Tabular legt nicht dar, welche Inferenzalgorithmen im Hintergrund verwendet werden. Im Bezug auf MLog ist folgendes

anzumerken: viele Fakten stehen für MLog nicht fest, und es ist ohnehin nur meine Spekulation anhand der bisher gesehenen veröffentlichten Informationen, dass MLog als Sprache Bayes'sche Inferenz unterstützen könnte. Diese Spekulation wird begründet zum Einen durch die MLog-Beispiele [Li+17b, S. 1934], anhand derer man problemlos weitere sprachliche Mittel für Bayes'sche Inferenz in MLog vorstellen kann. Zum Anderen zählt als Grund zu einer solchen Spekulation die Tatsache, dass MLog TensorFlow als ML-Backend verwenden kann. Das TF-Modul TensorFlow Probability eignet sich für Bayes'sche Inferenz und unterstützt MCMC und VI.

Wie der Leser bereits merken sollte, die Arten der Modellspezifikation in SimSQL (versionierte stochastische Tabellen mit Modellstruktur-Angaben im Rumpf) und SQL2Stan (Tabellen und explizite Modellformulierung getrennt) sind unterschiedlich, und dafür gibt es einen Grund. SQL2Stan baut das Hauptkonzept vom Stand der wissenschaftlichen Arbeit [RH16] aus (Bayes'sche Modelle sind übersetzbar in ER-Diagramme und somit in relationale Modelle), und implementiert die Bayes'sche Modellierung über dieses Konzept hinaus in reinem SQL unter Einbeziehung von Stan-verwandten sprachlichen Mitteln. SimSQL hingegen basiert auf einer anderen Vision aus dem SimSQL-Vorgänger MCDB, und hatte keinen Fokus auf die hilfreiche Übersetzung Bayes'scher Modelle in den Datenbankkontext. SimSQL führte die Schleifenerzeugung und Versionierung stochastischer Tabellen als eine Erweiterung der SQL-Sprache ein, weil man im Vorgänger MCDB keine hierarchischen Modelle (wie beispielsweise LDA) benutzen konnte. Einen kleinen Einschub in diesem Vergleich verdient MLog. MLog trennt die relationale Sicht von der ML-Problemformulierung ähnlich wie SQL2Stan. Der Grund hierfür: relationale DMS- und statistische ML-Modellierung werden in MLog durch unterschiedliche Sprachkonzepte gehandhabt. Die tensorale Abstraktion für das SciDB-Datenmanagement ist in MLog relational und SQL-ähnlich; tensorale MLog-Regeln zur ML-Abstraktion sind hingegen nicht SQL-, sondern Datalog-basiert.

Zurück zum relationalen Interface von SimSQL: zum Abrufen von Inferenzergebnissen in SimSQL werden stochastische Tabellen über spezielle, rein SimSQL-typische SQL-Queries (Compute-Statements) abgefragt, die vom Nutzer Angaben über bestimmte MCMC-spezifische Details erfordern.

Expressivität von Datenbank-Dialekten zur probabilistischen Programmierung Eine offene Frage ist der Expressivitätsvergleich zwischen MLog, SimSQL, Tabular und SQL2Stan.

Zur Expressivität von MLog lässt sich nur auf die Aussage aus [Li+17b] berufen, dass es das Ziel von MLog sei, die von anderen deklarativen ML-Systemen nicht unterstützten Modelle zu unterstützen. MLog verfügt in seiner Interfacesprache über einen Verbund aus tensoraler Algebra und SQL; dieses ML-System ein modernes Array-DBMS-Backend und funktionell umfangreiche low-level-ML-Backends (u.a. Theano, Tensorflow). MLog hat also theoretisch sehr viel Potenzial für vielseitige Expressivität zur Formulierung von ML-Aufgaben, doch der Schwerpunkt von MLog

scheint doch eher auf neuronalen Netzen und Deep Learning zu liegen. Man kann also aktuell schwer sagen, was sich in MLog ausdrücken lässt und was nicht, und ob man MLog mit Systemen zur Bayes'schen Inferenz vergleichen sollte. Da jedoch MLog starke Ähnlichkeiten zu Datalog aufweist, kann man an der Stelle [DKM10] erwähnen. In diesem Paper wurden Ansätze vorgestellt, wie man MCMC-Verfahren (eine Klasse von Algorithmen, die zwecks Bayes'scher Inferenz eingesetzt werden kann) über eine deklarative, Datalog-ähnliche Sprache umsetzen kann.

Der SQL2Stan-Dialekt wird aktuell auf einen kleinen Teil der PPL Stan abgebildet. Stan ist flexibel und ausdrucksstark. Das lässt hoffen, dass SQL2Stan durch Spracherweiterung (vorrangig um neue Verteilungsfunktionen) ausdrucksstärker gemacht werden kann. Hätte SQL2Stan alle von Stan unterstützten Verteilungsfunktionen im Vokabular, könnte es ohne jede Zweifel mehr Bayes'sche Modelle beschreiben als Tabular. Laut dem neuesten Stand soll SimSQL abgesehen von Bayes'schen Modellen außerdem Deep Learning unterstützen, was für eine bessere Expressivität im Vergleich zu SQL2Stan spricht.

Tabular wird über elegante, gut lesbare Spreadsheet-Code-Einzeiler bedient. Wenn es aber um die Ausdrucksstärke geht, ist Tabular nicht immer flexibel genug – und zumindest in einer Hinsicht weniger flexibel als SQL2Stan. Ich sehe beispielsweise keine Möglichkeit zur Formulierung eines hierarchischen Modells wie Naïve Bayes oder LDA in Tabular, was in SQL2Stan kein Problem ist. Im Verteilungsfunktionsumfang von Tabular fehlt dazu einfach die Dirichlet-Verteilung. Ich fand außerdem keine Möglichkeit, Unterabfragen bei den Funktionsparametern (z.B. nicht die ganze Spalte als einer der Parameter, sondern nur bestimmte Werte aus der Spalte) zu tätigen oder solche Unterabfragen gar zu schachteln, wie man es in SQL2Stan tun kann.

Das Projekt MCDB als Vorgänger von SimSQL hatte inhärente Probleme mit der Expressivität, da dort keine hierarchischen Modelle wie LDA ausgedrückt werden konnten. In SimSQL als Nachfolgerprojekt wurde dieses Problem durch die Einführung von schleifenerzeugten stochastischen Tabellen beseitigt. Der SQL2Stan-Dialekt wurde hingegen von Anfang an mit Rücksicht auf hierarchische Modelle entworfen. Die Grundlage für das SQL2Stan-Konzept waren die PPL Stan und zustandsfreie/nicht-versionierte Tabellen aus dem integrierten relationalen Datenbankschema.

Es wäre interessant zu wissen, wo die Grenzen der Expressivität in SQL2Stan liegen. Nach dem aktuellen Stand sind diese Grenzen überschaubar abgesteckt. Aktuell verfügt SQL2Stan über zwei unterstützte Verteilungsfunktionen (kategorische und Dirichlet-). Der SQL2Stan-Compiler ließe sich aber mit wenig Aufwand um weitere solche Funktionen aus dem Stan-Sprachumfang erweitern. Ferner besitzt SQL2Stan die Einschänkung „ein Modellparameter pro Constraint-Typ pro Tabelle"; ein konzeptueller Verbesserungsvorschlag zur Behebung dieser Einschränkung wurde aber im Kapitel 3.2.2 vorgestellt. Eine mögliche Faustregel zur Beschreibung potenzieller Expressivität von SQL2Stan soll vorerst lauten: gibt es ein generatives Modell als ein Bayes'sches Netz, so erlaubt SQL2Stan, die explizit gemachten Details für dieses Modell in SQL zu formulieren und automatisch eine datenma-

nagementgebundene Inferenzimplementierung dazu zu generieren.

E.2. Relevante Systeme und Veröffentlichungen

Beim Durchlesen von einem Interview [Rei18] mit Mike Lee Williams, einem amerikanischen Astrophysiker, der als Forschungsingenieur neue Ideen im maschinellen Lernen und künstlicher Intelligenz zum Leben erweckt, fielen mir einiger seiner Aussagen auf. Für Mike Williams sei die probabilistische Programmierung in seinem Kern deklarativ: man beschreibe die Welt, drückt auf „Start“ – und die probabilistische Programmiersprache kümmert sich um die daraus entstehenden Implikationen (Beispiel: automatische Inferenz). Wenn man ein praktisches, reales Problem habe, gebe es laut Williams zwei PPL, die einem helfen können: Stan und PyMC3. Stan wurde als Backend für Bayes'sche Inferenz im SQL2Stan-Projekt gewählt; es ist sowohl eine Sprache als auch eine Software. Stan wird das erste Unterkapitel der Übersicht über relevante Systeme gewidmet.

Zur Erinnerung des Lesers daran, was probabilistische Programmierung überhaupt ist: es ist das Definieren von einem Bayes'schen statistischen Modell in einem Computerprogramm. Der Nutzer baut ein Modell, gibt Modelldaten an – und analysiert in aller Bälde die automatisch geschätzten Inferenzergebnisse. Das Paradigma der probabilistischen Programmierung kommt immer dann ins Spiel, wenn 1) ein Rechenproblem mit Unsicherheiten arbeitet, 2) oder bei einem Rechenproblem bestimmte Strukturen instrumentalisiert werden sollen (z.B. Hierarchien) oder wenn 3) man bestimmtes Vorwissen besitzt, welches man in das Modell einbinden möchte [Car18].

E.2.1. Stan und SQL2Stan

Stan [Sta19a; Car+17] ist eine probabilistische Programmiersprache zum Definieren von statistischen Modellen in der gleichnamigen ML-Software Stan. Sie ist imperativ: in einem Stan-Programm legt der Nutzer eine modellspezifische logarithmierte Wahrscheinlichkeitsdichtefunktion (Zielfunktion) fest, welche in die Modellstruktur sowohl 1) die zu schätzenden Modellparameter als auch 2) die gegebenen Daten und Konstanten einbindet. SQL2Stan nimmt Stan als Machine-Learning-Backend zur probabilistischen Programmierung in SQL, weil es unschwer schien, SQL auf einen Teil von Stan abzubilden. Die Zielfunktion in Stan ist eine log-likelihood-Funktion, die eine Summe logarithmierter Wahrscheinlichkeiten darstellt. Eine Summe in SQL darzustellen schien einleuchtend unaufwändig zu sein. Außerdem wirkte die Stan-Sprache bei einfacheren Beispielen wie naïve Bayes oder LDA subjektiv gut lesbar. Die Formulierungen zur statistischen Modellierung in dieser Sprache schienen knapp und die Sprachstruktur angenehm zurückhaltend, was die Imperativität der Sprache sowie die in den Code eingepflegten Importe von externen Bibliotheken angeht. Nicht zuletzt spielte die Verfügbarkeit von Stan eine Rolle: Stan-Interfaces sind in Sprachen R, Python, shell, MATLAB, Julia und Stata verfügbar; Stan läuft auf den meisten relevanten Plattformen (Linux, Mac, Windows).

Als Software zur bayes'schen Statistik bietet Stan die Funktionalität zur Bayes'schen Inferenz mit Hilfe von Markov-Chain-Monte-Carlo- sowie Variational-Inference-Methoden. Stan kann Gradienten eines Modells berechnen, und implementiert hierfür automatisches Differenzieren im Rückwärtsmodus [2]. An der Stelle möchte ich für weitere Details auf die Veröffentlichung [Car+17] hinweisen. Die im Moment aktuellste praxisnahe Benutzeranleitung für Stan befindet sich unter [Sta19c].

E.2.2. TensorFlow Probability und Edward

Bereits im einführenden Kapitel wurde gesagt: Systeme, die die Integration der ML- und Datenmanagementaspekte nicht *ineinander*, sondern *miteinander* verfolgen, scheinen das meiste Potenzial innezuhaben. Durch die *miteinander*-Integration von Komponenten (u.a. DMS und ML) können ihre Stärken und technischen Fortschritte ausgenutzt werden [LM12]. Es sollte daher niemanden wundern, warum Softwarestacks als modulare Alleskönner das Machine Learning mit anderen wichtigen Aspekten (wie Datenmanagement, -Reinigung und Modellserving) verknüpfen. Beispiele für solche Stacks sind TensorFlow Extended [Bay+17] oder DAWN (noch in Entwicklung, siehe Kapitel E.2.24). In beiden Fällen handelt es sich um Ende-zu-Ende-ML-Entwicklungsplattformen, die Machine Learning nur als einen Teil eines größeren Workflows sehen: es geht also nicht nur um das Schreiben vom ML-Code, sondern vor allem darum, wie man ML-gestützte Prozesse in Betrieb nimmt. Die TensorFlow-Bibliothek wird als Basis für mehrere Softwareprojekte zur Datenanalytik verwenden. Meist kann man es sich so vorstellen: TensorFlow bietet eine Infrastruktur für Operationen auf Tensoren (beliebig dimensionierten Arrays) als eine Art Fundament für darauf aufbauende Projekte mit high-level-Abstraktionen. Der Aspekt des Datenmanagements ist bei TensorFlow durch die Anbindung von Apache Spark (siehe TensorFlowOnSpark [Yah19]) oder Hadoop (siehe TonY [Lin19]) umsetzbar.

TensorFlow Probability (kurz: TFP; GitHub-Repository: `https://github.com/tensorflow/probability`) ist eine im Rahmen dieser Arbeit relevante Python-Bibliothek für TensorFlow. Ihr Ziel: das Kombinieren von probabilistischen Modellen und Deep Learning leichter zu machen. Ein starker Fokus fällt dabei auf generative Deep-Learning-Modellen (*deep generative models*) [Abe17]. Die Zielgruppe besteht daher aus Datenforschern (engl.: data scientists), Statistikern, ML-Spezialisten und Programmierern, die ihr Domänenwissen kodieren wollen, um eigene Daten besser zu verstehen und anhand der Daten Vorhersagen zu treffen [Ten19b].

Das TF-Probability-System ist schichtweise wie folgt aufgebaut:

0. TensorFlow (low-level-Schicht für numerische Operationen)
1. Statistische „Baublöcke“ (Wahrscheinlichkeitsverteilungen, Bijectors zur Variablentransformation)

[2] Eine Erklärung für Interessierte: `https://rufflewind.com/2016-12-30/reverse-mode-automatic-differentiation`

2. Modellspezifikation (probabilistische Programmierung, neuronale Netze und ihre Schichten, trainierbare Verteilungen)

3. Probabilistische Inferenz (Markov-Chain-Monte-Carlo, Variational Inference)

In der zweiten Schicht – dort, wo Modelle gebaut werden – wird eine spezielle probabilistische Sprache Edward2 utilisiert. Edward2 ist eine umgearbeitete Version der Sprache Edward, die nur die essentiellen Sprachteile beinhaltet – das beschreibt das TFP-Team mit dem Wort „destilliert“ (für Details siehe [Tra18]). Der sprachliche Vorgänger von Edward2 – Edward [Tra+16] – ist eine Turing-vollständige PPL zur automatischen Inferenz und zum Zusammenstellen von Darstellungen probabilistischer Modelle [Tra+17], die versucht, die probabilistische Programmierung ein Stück weit mit Deep Learning zu verknüpfen. Der Kernpunkt von Edward bzw. Edward2: probabilistische Programmierung mit gleicher Flexibilität und Recheneffizienz wie beim klassischen Deep Learning [Tra+17, S. 1]. Wenn man mit Edward2 im TFP-Kontext arbeitet, schreibt man als Nutzer Python-Code. Ein Paar erklärende Code-Beispiele zu Edward2 findet man unter [Tra19].

Der Ansatz dieser Arbeit bezieht sich auf die Übersetzung von SQL nach Stan und nicht nach Edward. Das liegt daran, dass Edward als Zielsprachen zur Codeübersetzung komplexer erscheinen als Stan. Außerdem sei Edward zumindest nach dem Stand von September 2017 in Sachen Dokumentation und Beispiele hinter Stan und PyMC3 gewesen [Abe17], doch Edward2 schaut sehr vielversprechend aus. Es wäre interessant zu wissen, ob man SQL2Stan-Spezifikationen automatisch nach Edward2 übersetzen kann.

Bei SQL2Stan liegt einem Workflow oft ein Bayes'sches Netz zugrunde – ein grafisches Modell, darstellbar als ein gerichteter azyklischer Graph. Ein neuronales Netz, so wie man es in die TFP-Modellspezifikation involvieren kann, hat eine ähnliche Struktur (gerichteter Graph), wenn man von den Unterschieden zwischen Bayes'schen und neuronalen Netzen absieht. Inwiefern die Konzepte von SQL2Stan auf neuronale Netze übertragbar sind, ist eine offene Frage.

Zum Schluss sollte erwähnt werden, dass TensorFlow wenig Deklarativität bietet und für bestimmte ML-Modelle im geringeren Maße skalierbar und langsamer arbeitet als eine deklarative relationale DMS-Plattform SimSQL [Jan+19].

E.2.3. PyMC

Der aktuelle Release von PyMC – PyMC3 (`https://docs.pymc.io/`)– ist eine Bibliothek für Python zur probabilistischen Programmierung. Dieses Framework erlaubt den Nutzern, mit Hilfe von unterschiedlichen numerischen und statistischen Methoden automatische Modellanpassung (bzw. automatische Inferenz) für Bayes'sche Modelle durchzuführen. Modellvergleiche und -diagnostik sowie flexible Unterstützung von Gaußschen Prozessen gibt es dabei inklusive [Car18].

Man kann zwischen PyMC3 und Stan gewisse Parallelen ziehen: beide arbeiten mit generativen Bayes'schen Modellen, die in einer für Statistiker zugänglichen Sprache spezifiziert werden können. Auch der algorithmische Werkzeugkasten ist

ähnlich: beide arbeiten z.B. mit No-U-Turn-Samplern aus der MCMC-Familie von Algorithmen bzw. mit VI (Variational Inference). Sowohl Stan als auch PyMC3 versuchen, die Bayes'sche Modellierung einfach und zugänglich zu gestalten. Beide Systeme sind nicht an ein Datenmanagement-System geknüpft, aus dem sie die Modelldaten schöpfen und in das sie die Rechenergebnisse reinschreiben können. Man kann PyMC3 am besten mit dem R-Interface für Stan vergleichen: beide Systeme sind von der Funktionalität sehr ähnlich. Stan ist eine domainspezifische Sprache, und in PyMC schreibt man in Python.

Stan ist hervorragend dokumentiert und ist als Software für Nutzer gedacht, die aus dem statistiklastigen Umfeld kommen. PyMC3 orientiert sich seinerseits mehr an Python-Nutzer (viele Data-Mining-Spezialisten sind Python-Nutzer) [Abe17]. Technisch gesehen ist Stan eine domainspezifische Sprache mit einer Implementierung in C++, und hat im Gegensatz zu PyMC leider kein Backend wie Theano (wird nach PyMC3 eingestellt) oder TensorFlow (wird in PyMC4 eingeführt), obwohl das Potenzial für zusätzliche Beschleunigung durch GPU-seitige Berechnungen in sich verbirgt [Abe17].

Im Vergleich zu anderen probabilistischen Programmiersprachen wie Figaro, Church oder Anglican, die Turing-vollständig sind und sehr viele Einsatzmöglichkeiten an den Tag legen, konzentriert sich PyMC3 vielmehr auf Benutzerfreundlichkeit und Recheneffizienz bei der Lösung von den meisten reellen Problemen [Abe17].

Als Backend für PyMC3-Berechnungen dient die Python-Bibliothek Theano. PyMC4 wird nach Release auf TensorFlow Probability zurückgreifen, was PyMC4-Projekte skalierbar machen und das Bauen von generativen Deep-Learning-Modellen (*deep generative models*) erleichtern wird. PyMC4 befindet sich noch in Entwicklung [PyM19b].

E.2.4. Infer.net

Ein anderes System zur probabilistischen Programmierung nennt sich Infer.net [NET19a; NET19b; WW11]. Es ist ein Machine-Learning-Framework zur Bayes'schen Inferenz auf grafischen Modellen, welches in jeder .NET-Sprache (d.h. auch in C#) benutzt werden kann.

In Infer.net kann der Nutzer ein eigenes statistisches Modell spezifizieren und bestimmte Inferenz-Abfragen an dieses Modell formulieren. Für diese Zwecke verfügt Infer.net über eine Modellbau-API [NET19b]. Die Nutzerangaben werden zusammen compiliert und in eine Inferenzimplementierung (Zielsprache C#) übersetzt, die unter Einbeziehung von beobachteten Daten die Inferenzausgabe ausrechnet. Beim Spezifizieren von Modellen arbeitet man hier mit Zielfunktionen (so wie bei Stan). Die Art, wie der Nutzer das Modell spezifiziert, entspricht strukturell einem Faktorgraphen der modellspezifischen Wahrscheinlichkeitsverteilungsfunktion. Die grundlegende Funktionalität zur automatischen Inferenz ist vergleichbar mit Stan und PyMC3. Stan und PyMC sind im Vergleich zu Infer.net in den letzten Jahren jedoch viel innovativer gewesen und wurden deutlich aktiver entwickelt: die Version 2.6 von Infer.net kam im November 2014 raus, die nachfolgende Version 2.7 wurde

etwa dreieinhalb Jahre später veröffentlicht.

In der Infer.net-Version 0.3 vom Oktober 2018 wurde Infer.net als *open-source*-Projekt unter der MIT-Lizenz veröffentlicht (siehe offizielle Mitteilung [Zay18]). Zweifellos trägt dieser Schritt dazu bei, dass die Entwicklung von Infer.net durch die Community vorangetrieben wird. Es bleibt abzuwarten, wie Infer.net sich weiterentwickelt. Zu erwarten ist, dass Infer.net künftig an Relevanz verlieren wird, während Stan und vor allem PyMC4 künftig immer mehr Relevanz in der probabilistischen Programmierung gewinnen werden.

E.2.5. WebPPL

Die Einbindung von dem recht neuen Paradigma der probabilistischer Programmierung in andere Programmierbereiche scheint viele Interessen anzusprechen. Das Open-Source-Projekt WebPPL (`http://webppl.org/`) pflegt eine probabilistische Programmiersprache in die Browser-Konsole sowie in node.js ein [Com19; Möb18b; Möb18a]. Browserbasierte Anwendungen mit automatischer Inferenz für die in JavaScript spezifizierten Modelle – ein frischer Wind im Bereich der akademischen PPL, der aus dem „Computation & Recognition"-Labor an der Stanford-Universität weht. Ähnlich wie in SQL2Stan handelt es sich an dieser Stelle um Einbettung von PPL-Sprachdetails in eine ursprünglich nicht als PPL entworfene Sprache. Durch diese sprachliche Einbettung soll die automatische Inferenz und probabilistische Programmierung für eine bestimmte Benutzergruppe (JavaScript-Programmierer im Falle von WebPPL) zugänglicher gemacht werden. WebPPL sieht jedoch keine Anbindung an Datenmanagementsysteme vor.

E.2.6. BayesDB

Ein interessantes Projekt, das sich mit der Verknüpfung von Datenmanagement und statistischer Inferenz befasst, ist BayesDB [Man+15]. BayesDB ist ein System zur probabilistischen Porgrammierung sowie zum Abfragen von wahrscheinlichkeitsbasierten Implikationen aus den Daten, die in einer Datenbank liegen. Das Projekt setzt sich zum Ziel, die statistische Inferenz auch für nicht-Statistiker verfügbar zu machen. Die Benutzerzielgruppe soll allerdings mit relationalen Datenbanken vertraut sein, woran sich Ähnlichkeiten zu SQL2Stan erkennen lassen. Die Ergebnisse der Datenauswertung können bei BayesDB lassen sich wie gängige DB-Daten über SQL-Queries abgefragt werden [Man+15, S. 1–2]. Mit BQL – einer SQL-ähnlichen Sprache – bietet BayesDB ein einfaches Interface für Datenanalyse. Statistische Modelle werden durch den BayesDB-Nutzer nicht spezifiziert. Vielmehr werden für einen Datenanalyse-Prozess mehrere probabilistische Modelle automatisch durch das System generiert. Intern handelt es sich dabei um eine gewichtete Sammlung von automatisch generierten probabilistischen Datenmodellen. Der Nutzer kann die automatische Generierung von solchen Data-Mining-Modellen deklarativ (und somit indirekt) beeinflussen bzw. eine externe Datenmodell-Sammlung importieren [Man+15, S. 24]. Die meisten algorithmischen Entscheidungen (wie z.B. die

Modellauswahl via Bayes'sche Modellselektion oder Implementierung der Inferenz) werden vor dem Nutzer verborgen. Data-Mining-Modelle einerseits und Datenressourcen andererseits sollen in BayesDB unabhängig voneinander austauschbar sein.

Deklaratives Nutzerinterface: erweitertes SQL Als Nutzerinterface wird ein etwas erweitertes SQL benutzt, getauft mit dem Namen BQL (Bayesian Query Language). Die wichtigsten hinzugefügten Befehle sind SIMULATE, ESTIMATE und INFER. SIMULATE steht fürs Erwürfeln neuer Daten: gemeint sind damit sowohl fehlende Eigenschaften in bereits existierenden Datenpunkten (leere Spalten in bereits existierenden Zeilen) als auch neue Datenpunkte, die hypothetisch aus der gleichen Grundgesamtheit stammen wie die vorhandenen Daten. ESTIMATE schätzt Werte für Wahrscheinlichkeitsdichtefunktionen zwecks Vorhersagen und leitet verschiedene informationstheoretische Werte ab (z.B. Wahrscheinlichkeiten, dass bestimmte Variablen voneinander abhängen). INFER generalisiert multimodale Wahrscheinlichkeitsverteilungen auf einzelne Werte; dies ist nützlich für Vorhersage der Eigenschaften bestimmter Datenpunkte. Mit BQL kann eine Reihe von Datenanalyse-Aufgaben anvisiert werden: Säuberung und Erkundung von Daten, konfirmatorische Analyse, Vorhersagen, Suche nach statistisch ähnlichen DB-Einträgen. Die BayesDB-Abfragesprache BQL (Bayesian Query Language, ähnlich zu SQL) sieht Spalten als Inferenzobjekte. Es wird davon ausgegangen, dass den Spaltendaten eine bestimmte Wahrscheinlichkeitsverteilung zugrunde liegt, die von den spaltenspezifischen Beobachtungen eingeschränkt ist. Eine Spalte entspricht einer Zufallsvariable, und eine Zeile entspricht einer Beobachtung.

Deklarative, automatisierte Modellierung über Meta-Modelle Das statistische Backend von BayesDB arbeitet mit generativen Modellen und bietet intern ein entsprechendes mathematisches Interface mit statistischen Operationen, um das Generieren von konkreter Implementierung für Inferenzaufgaben zu ermöglichen. Abgesehen von der deklarativen Abfragesprache BQL bietet BayesDB eine weitere Sprache an – MML (Meta-Modelling Language). In MML wird auf eine ebenfalls deklarative Weise, was BayesDB beim automatischen Generieren von Data-Mining-Modellen berücksichtigen soll. Der Nutzer programmiert mit Hilfe von MML kein konkretes probabilistisches Modell für seinen Datensatz, er gibt nur ein Meta-Modell zum automatischen Generieren eines solchen Modells an. In einem Meta-Modell stehen Details drin, die automatisch generierte Modelle innehaben müssen, z.B. stochastische Abhängigkeit der einen Spalte von der anderen oder Angabe eines bestimmten externen diskriminativen Modells für einzelne Spalten. Einzelne Spalten dürfen also im Rahmen der Meta-Modell-Formulierung einem konkreten, bereits trainierten diskriminativen Modell folgen, welches dann als Python-Code (d.h. nicht in SQL) von außerhalb der BayesDB-Infrastruktur eingelesen werden muss.

Um zu erfahren, was der Begriff „diskriminativ" in diesem Zusammenhang bedeutet, lohnt sich ein kleiner theoretischer Einschub (basierend auf [NJ01]). Bisher ging es in dieser Arbeit nur um generative Modelle, und man lernte zwei generative

Klassifikatoren kennen (LDA und naïve Bayes). Generative probabilistische Modelle modellieren mit statistischen Mitteln die Herkunft der Daten, die man beobachtet. Sie heißen „generativ“, weil man damit neue Datenpunkte generieren kann. Generative Klassifikatoren modellieren die Verbundverteilung $p(x, y)$ mit den gegebenene Daten x und der Klasse y und vermitteln Vorhersagen über die Wahrscheinlichkeit $p(y|x)$ nach dem Satz von Bayes. Diskriminative Modelle sind leicht anders. Die Bezeichnung „diskriminativ“ („der Unterscheidung helfend“, wichtig für Klassifikatoraufgaben) bedeutet im Gegenteil zu „generativ“, dass man die Datenherkunft explizit nicht modelliert und sich beim Modellieren nur auf die gegebenen Daten stützt. Ein diskriminativer Klassifikator modelliert direkt $p(y|x)$, d.h. er bildet die Eingabe x direkt auf die Klassen y ab. Diskriminative Modelle machen also weniger Annahmen über die Daten (z.B. keine Annahmen zu der der Grundgesamtheit zugrundeliegenden Verteilung, aus der die gegebenen Daten stammen könnten), dafür hängen sie umso mehr von der Qualität der Daten ab.

Damit zurück zu BayesDB: das System erlaubt dem Nutzer ein Meta-Modell in MML zu beschreiben, um die automatische Modellgenerierung durch das System wegweisend einzuschränken. BayesDB generiert aus dem Meta-Modell nicht nur ein statistisches Modell, sondern direkt eine Mehrzahl von generativen Modellen, die die multivariate Verteilung der Grundgesamtheit repräsentieren. Dabei werden die beobachteten Daten als ein Teil der Grundgesamtheit angesehen. Der aus dem Meta-Modell generierte Satz an generativen Mischungsmodellen kann vielseitig für Zwecke der Datenanalyse eingesetzt werden. Verschiedene Aufgaben werden mit Hilfe von BMA-Techniken (Bayesian Model Averaging) umgesetzt [Man+15, S. 1–3].

Typischer BayesDB-Workflow Konzeptuell geht ein BAyesDB-Nutzer im Regelfall wie folgt vor [Man+15, S. 4–5]. Er lädt strukturierte Daten rein (z.B. als CSV). Eine Zeile in der CSV-Datei soll einer Beobachtung entsprechen (z.B. einem Patienten), und jede Spalte steht für jeweils eine Variable (z.B. Alter des Patienten). Die Datentypen für die Spalten kann der Nutzer automatisch erkennen lassen (die Altersangaben werden z.B. als integer-Zahlen erkannt). Dann erweitert der Nutzer das spezifizierte relationale Schema um ein Meta-Modell, indem er z.B. die (Un-)Abhängigkeiten zwischen den Spaltendaten und ggf. diskriminative spaltenspezifische Modellierungdetails angibt. BayesDB stellt ein Standard-Meta-Modell zur Verfügung, welches der Nutzer durch seine Angaben anpasst. Das angepasste Meta-Modell wird von System dazu genutzt, eine Mehrzahl von Data-Mining-Modellen zu generieren. Anschließend muss die Initialisierung von einer bestimmten Anzahl von generativen Modellen (z.B. 100 Stück) in Auftrag gegeben werden. Die Initialisierung von dieser Mehrzahl an Modellen geschieht anhand des durch den Nutzer angepassten Meta-Modells. Sobald die Modelle an die gegebenen Daten angepasst sind, kann der Nutzer inferenz- bzw. schätzungsbezogene BQL-Abfragen starten: z.B. zum Schätzen der Variablenkorrelationen, zum Vorhersagen bestimmter Merkmale von Datenpunkten oder zum Generieren neuer Daten.

Abwesenheit expliziter probabilistischer Programmierung BayesDB ist extra dafür konzipiert, die Modellspezifikation wegzuabstrahieren. Der Nutzer spezifiziert in keiner der BayesDB-Sprachen explizit konkrete probabilistische Modelle. Stattdessen wird eine Sammlung von Modellen implizit (anhand des in MML formulierten Meta-Modells) generiert und im Hintergrund zwecks Datenanalyse genutzt [Man+15, S. 40]. In der Sprache MML – geschaffen, um diesen impliziten Prozess der Generierung von einem Satz an Modellen anzupassen – stellt der Nutzer bloß Schranken für den Algorithmus, der die probabilistischen Modelle für die Sammlung generiert. Der Nutzer wählt dadurch keine konkreten Modelle, denn diese Einschänkungen für die Modellgenerierung, die er formuliert, beschreiben nicht konkret die Struktur des Modells, das am Ende genutzt wird. Diese Art der impliziten probabilistischen Programmierung, in der der Nutzer nicht mit konkreten probabilistischen Modellen in Berührung kommt, steht zweifellos im Kontrast zu herkömmlichen PPL wie Stan [Car+17], Church [Goo+12], BLOG [Mil+05] und Figaro [Pfe09]. Die soeben genannten probabilistische Programmiersprachen befolgen das Konzept „ein Programm – ein spezifisches probabilistisches Modell“: man hält für die statistische Inferenz an einem konkreten Modell fest. In manchen Bereichen ist es erwünscht, eigene Daten mit konkreten Bayes'schen Modellen (aus wissenschaftlichen Publikationen oder eigenständig entworfen) zusammenzufügen und darauf statistische Inferenz durchzuführen. Die Vielfalt an bereits veröffentlichten Bayes'schen Modellen ist domänenübergreifend groß und sollte nicht ignoriert werden.

Die Deklarativität und Abstraktivität von BQL und MML spricht für sich. Aus meiner Sicht steht wenig im Weg, sie noch besser zu machen durch beispielsweise eine Erweiterung des Systems um eine Art „Umschaltung“ zwischen automatischer Modellgenerierung à la BayesDB und expliziter Modellspezifikation wie bei SQL2Stan. Weiss der Nutzer, welches Modell er einsetzen möchte, dann soll er sich das Modell auch unaufwändig zunutze machen können – beispielsweise auf die Art von SQL2Stan. Wenn der Nutzer hingegen kein passendes Modell vor Augen hat, welches er einsetzen könnte, kann ein Datenanalyse-System auf die Art von BayesDB automatisch ein Modell generieren. Man bedenke außerdem, dass BayesDB zum Parametrisieren automatischer Modellgenerierung zum Teil auf Python zurückgreifen muss, was im Kontrast zu dem deklarativen SQL-verwandten BayesDB-Interface sprachlich inkonsistent wirkt. SQL2Stan zeigt, dass man Inferenzaufgaben durchgehend in SQL definieren kann.

Der BayesDB-Prototyp [MIT19] nutzt eine relationale SQlite-Datenbank als DMS (SQL2Stan-Prototyp nutzt zu diesem Zweck ebenfalls ein RDBMS) und materialisiert die Daten zwecks Inferenz in einer Datei [3] – ebenso wie der SQL2Stan-Prototyp. Das bezeugt, dass SQL2Stan nicht der einzige Prototyp eines Inferenz-Systems ist, welches zwar darauf bedacht ist als Vermittlung zwischen Big-Data-DMS- und ML-Software zu agieren, technisch jedoch (noch) nicht soweit ist und daher nur als funktionierender Entwurf eines theoretisch skalierbaren Konzepts fun-

[3] Siehe den Programmcode unter `https://github.com/probcomp/bayeslite/blob/master/src/backends/loom_backend.py` am Kommentar „Ingest data into loom“, Stand 28.01.19

giert.

Abgesehen von der automatischen Inferenz von Modellen addressiert BayesDB zielgerichtet die Hauptprobleme der angewandten statistischen Inferenz (wie Erkundung, Säuberung von Daten sowie konfirmative Analyse).

E.2.7. brms

Das Projekt brms [Bür19; Bür16] übersetzt R-Code nach Stan. Somit ist es neben SQL2Stan ein weiteres Projekt, welches die PPL Stan als Ziel der automatischen Codegenerierung benutzt, um die probabilistische Programmierung für eine bestimmte Nutzergruppe einfacher zu machen. Paul Bürkner von der WWU Münster bettet mit Hilfe von brms die probabilistische Sprache Stan in die imperative Programmiersprache R ein, obwohl Stan von sich aus ein R-Interface hat. Dadurch soll die automatische Differenzierung, die vom Stan-Backend bewerkstelligt wird, dem breiteren Publikum (den R-Programmierern) zugänglicher werden, damit sie nicht zusätzlich Stan erlernen müssen. In brms wird, genau wie bei SQL2Stan wird, intern versucht, einen lesbaren und effizienten Stan-Code aus dem nutzerspezifizierten Code zu generieren [Bür16].

E.2.8. Agrios

Das Projekt Agrios [LM12] beschäftigt sich damit, wie man die Datenanalyse mit R skalierbar machen kann. R ist eine imperative, datenanalyse-freundliche Programmiersprache, die von sich aus keine DMS-Funktionalität und auch keine Anbindung an Datenmanagementsysteme bietet. Agrios ist ein Beitrag, in dem die Sprache R mit einem skalierbaren Datenmanagement auf der sprachlichen Ebene verbunden wird. Wichtig: der im Jahr 2012 prototypisch umgesetzte Agrios-Ansatz zur Ausführung vom R-Code auf SciDB-gelagerten Daten ist mit hoher Wahrscheinlichkeit ab 2013 obsolet geworden durch das funktionell gleiche SciDB-interne Programmiermodul SciDB-R [LM12, S. 59–60]. Agrios ist im Kontext von SQL2Stan dennoch interessant, weil es auch bei Agrios um deklarative Verknüpfung von Datenanalyse mit Datenmanagement geht.

Als DBMS wurde SciDB [Sto+13] gewählt: ein Datenmanagement-System für wissenschaftliche Big-Data-Use-Cases, wo relationale DBMS nicht effizient arbeiten. Statt Tabellen nutzt SciDB Arrays, weil sie in vielen Fällen natürlicher für die Modellierung wissenschaftlicher Anwendungsdaten erscheinen sollen. Solche DMS-seitigen Argumente wie die Bevorzugung von Arrays für Datenanalyse sowie die Unzufriedenheit mit relationalen DBMS für wissenschaftliche Zwecke tangieren den SQL2Stan-Ansatz. Bei SQL2Stan werden zwar relationale Tabellen als Datenschema-Abstraktion genutzt, diese werden jedoch in Arrays und Vektoren übersetzt (ein Array/Vektor pro Spalte). Theoretisch spricht nichts gegen den Einsatz von SciDB als Datenmanagement-Backend für SQL2Stan. Eine für SQL2Stan typische relationale Abstraktion der Datenmanagement-Details, die in Wahrheit auf Arrays abbildet, kann in Einklang mit einem Array-DBMS wie SciDB gebracht werden.

Die in SciDB gespricherten Arrays können in Agrios über AQL (Array Query Language) angesprochen werden – eine deklarative Anfragesprache, die SQL weitgehend ähnelt. SciDB verfügt außerdem noch über eine weitere, funktionale Sprache namens AFL. AFL ist ähnlich zu relationaler Algebra [LM12, S. 7]. Das Agrios-Projekt stellt Ideen vor, wie die Sprache R in die SciDB-Sprache AFL übersetzt werden kann. So können diejenigen Datenanalytiker, die bisher mit an kleineren Datensätzen R gearbeitet haben, ihre Programme übersetzen, sodass diese auch mit größeren Datensätze im Array-RDBMS SciDB laufen können. Für die R-Programmierer impliziert das einige Vorteile: unter anderen müssen die R-Programmierer dadurch nicht mehrere Versionen gleicher Skripte unterhalten (z.B. einmal für kleiner Datensätze, einmal für große Datensammlungen).

Bemerkenswert ist das folgende Detail: laut SciDB-Team stellt SQL als Interface-Sprache viele Wissenschaftler unzufrieden [LM12, S. 1], hauptsächlich weil SQL die typische RDBMS-Anfragesprache ist und Arrays ein – im Vergleich zu relationalen Modellen – natürlicheres Mittel zur Modellierung der Daten in ihren Forschungsbereichen darstellt. Dennoch bietet SciDB eine deklarative Anfragesprache AQL an, die SQL stark ähnelt. Das heißt, die Unzufriedenheit mit RDBMS liegt gar nicht an SQL, sondern an Schwierigkeiten, mit denen der Einsatz kalssischer RDBMS für bestimmte Anwendungszwecke verbunden ist. AQL belegt, dass die SQL-typische Deklarativität innerhalb der DMS nach wie vor gefragt ist; die deklarativen Mittel und die dahinter verborgenen technischen Lösungen müssen nur mehr den Erwartungen der Domänenspezialisten entsprechen.

E.2.9. SciDB

SciDB [Sto+13] ist ein Array-DBMS – ein Datenmanagement-System für wissenschaftliche Big-Data-Use-Cases, wo relationale DBMS nicht ohne hochprofessionell durchgeführte Optimierungen effizient arbeiten können. Als ein Array-DBMS nutzt dieses System Arrays anstatt Tabellen, weil sie in vielen Fällen natürlicher für die Modellierung wissenschaftlicher Anwendungsdaten erscheinen sollen. SciDB eignet sich als DMS für array-gestützte bzw. tensorale Anwendungen und wird daher beispielsweise in den Projekten Agrios [LM12] und MLog [DS317; Li+17b] eingesetzt.

SciDB achtet auf die Anforderungen [Sto+09], die der wissenschaftliche Alltag an die DB-Managementsysteme stellt. „In situ“-Daten im HDF5-Format (Daten, die noch nicht in ein DBMS eingelesen wurden) sollen unterstützt werden [Sto+09, S. 6], weil viele Wissenschaftler sich darüber beklagten, das Einlesen der Daten in ein DBMS halte sie vom Lösen der eigentlichen Forschungsaufgaben ab.

In SciDB als einem Array-DBMS können Arrays über eine SQL-ähnliche deklarative Sprache AQL (Array Query Language) abgefragt werden. AQL-Queries werden intern in eine andere, prozedurale SciDB-Sprache AFL übersetzt (Array Functional Language), welche die AQL-Abfragen implementiert und die als zweitrangige Interfacesprache fungiert. Außerdem bietet SciDB von Haus aus eine eingebaute R-Programmierumgebung zur Datenvisualisierung und -Analyse an.

Array-DBMS gegenüber RDBMS Laut dem SciDB-Team stellt SQL als Interface-Sprache viele Wissenschaftler unzufrieden [Sto+09, S. 1], hauptsächlich weil SQL die typische RDBMS-Abfragesprache ist und Arrays ein im Vergleich zu relationalen Modellen natürlicheres Mittel zur Modellierung der Daten in ihren Forschungsbereichen darstellt. Das Simulieren von multidimensionalen Arrays über den Tabellen sei ineffizient und unflexibel [Sto+07, S. 6]. Dies wurde im Rahmen des Projekt Sequoia [SFD93] an der Berkeley-Universität festgestellt. Dieses Projekt setzte sich als Ziel, das relationale Postgres-DBMS für wissenschaftliche Nutzer zugänglich zu machen. Das Projekt Sequoia hatte damit keinen Erfolg. Der Fehler lag in der Wahl eines für die Projektaufgabenstellung schlecht geegneten DBMS. Postgres unterstützte keine großen multidimensionalen Arrays, und genau diese wurden von den Spezialisten stets benötigt. Die Zielgruppe des Sequoia-Projekts waren Wissenschaftler aus der *Earth Science research group* an der kalifornische Santa-Barbara-Universität sowie die *climate modelling group* an der UCLA. Sie arbeiteten hauptsächlich mit großen multidimensionalen Array-Daten. Der für dieses schwerweigende Problem erdachte Workaround war die Simulierung von Arrays über den Tabellen (obwohl man eigentlich mit Tabellen arbeitete), und das war keine gute Lösung.

In SQL2Stan arbeitet man ebenfalls mit Arrays und Tabellen, und umgeht den Fehler, der den Forschern im Sequoia-Projekt unterlaufen war. Dafür wird die Simulierung von Darstellungen im Vergleich zu Sequoia einfach umdreht: der SQL2Stan-Prototyp arbeitet im Hintergrund mit multidimensionalen Arrays und gibt nach außen hin (im Benutzerinterface) nur vor, dass es sich um Tabellenspalten handelt. SQL2Stan sieht das relationale Schema in der Eigenschaft einer Datenmanagement-Abstraktion gegenüber dem Nutzer. Das, was auf der DMS-Ebene unter dieser relationalen Abstraktion abläuft, muss nicht einmal zwingend array-basiert sein. Ob unter dieser DMS-Abstraktion tatsächlich Tabellen und Spalten (in RDBMS), Arrays (in Array-DBMS), DataFrames (in Dataflow-Engines wie Apache Spark) oder vielleicht sogar noch unstrukturierte Daten lauern, sollte für den Nutzer keine Rolle spielen. Man scheiterte in Sequoia mit dem Simulieren von Arrays auf/über Tabellen; über einen anderen Weg (wie bei SQL2Stan) kann man aber hoffen, durch das Simulieren von Tabellen das konkrete Datenmanagement strukturierbarer Daten flexibel wegzuabstrahieren und dadurch den Problemen mit ungeeigneten DBMS aus dem Weg zu gehen. Außerdem: allein Tabellen oder auschließlich Arrays können manche Nutzer nicht zufriedenstellen, die für ihre Zwecke eher eine Mischung aus speziellen DBMS brauchen [Sto+09, S. 2] [How06]. Eine SQL2Stan-artige Spezifikation, die die genaue Datendarstellung im DMS verbirgt und alle Implementierungsdetails dem Compiler überlässt, würde sicherlich auch über eine Mischung von DMS hinweg abstrahieren können.

Es wäre sehr spannend zu wissen, inwiefern SciDB als Dateninfrastruktur für den SQL2Stan-Ansatz dienen könnte. SQL2Stan verwendet das relationale Schema und SQL vor allem als Basis zur Codegenerierung, stellt allerdings Arrays für die Inferenzsoftware bereit. Wenn es nach außen deklaratives SQL und relationale Schemata sind, innen aber imperative Befehle auf Arrays werkeln, lässt sich SciDB als ein Array-DBMS zur SQL2Stan-Datenverwaltung gut vorstellen.

E.2.10. MLog

MLog [DS317; Li+17b] ist ein Ansatz zur Integration von Machine Learning ins Datenmanagement mit Hilfe von einer deklarativen high-level-Sprache. Im Groben geht es hier darum, dass die relationale Algebra und SQL um lineare Algebra und Tensoren erweitert wird, ohne dass die Deklarativität verloren geht.

Moderne Datennamanagement-Systeme wie MADlib [MAD19; Hel+12], SAP PAL [Fär+12], Apache Spark [Zah+16] mit MLlib [The19g; Men+16] und SciDB [Sto+13] verfügen über eingebaute Funktionalität zur Datenanalyse, sind eng an die relationale Datenabstraktion gebunden – und seien dennoch wenig flexibel in der Auswahl von Typen der einsetzbaren ML-Modelle, weil sie maschinelles Lernen nur als Blackbox-Funktionen unterstützen [Li+17b, S. 1933]. Frameworks wie TensorFlow [Mar+15], Theano [The16] und Caffe [Jia+14] seien hingegen viel flexibler, weil sie auf das Konzept „ML als Blackbox“ verzichten und die mathematische Struktur hinter den ML-Modellen gegenüber dem Nutzer offenbaren. Die ML-Algorithmik kann sogar deklarativ gestaltet werden, wie z.B. in Keras: dort definiert der Nutzer nur Arrays und die logischen Abhängigkeiten zwischen ihnen, anstatt modellspezifische ML-Algorithmen zu schreiben. Diese Deklarativität des ML-Teils deckt allerdings die Datenmanagement-Abstraktion nicht ab, und dies resultiert in 1) der Notwendigkeit, sich als Nutzer um die Datenverwaltung zu kümmern sowie 2) in der Inkompatibilität mit relationalen Ökosystemen, in denen die meisten Unternehmen ihre Daten lagern [Li+17b, S. 1933].

Das Projekt MLog versucht, das deklarative, Keras-ähnliche Machine Learning vom Gespann des deklarativen, SciDB- oder PostgreSQL-ähnlichen Datenmanagementsystems ziehen zu lassen. Die Art der relationalen SciDB-Abstraktion von Daten wird in MLog weitgehend wiederverwendet, um vor allem auf die Erkenntnisse aus der Datenmanagement-Forschung zurückzugreifen und zugleich die Benutzer relationaler Datenbanken anzusprechen. Die Abfragesprache über das SciDB-Datenmodell wird in MLog erweitert, damit die Nutzer eigene ML-Modelle spezifizieren können. MLog möchte *die* Machine-Learning-Bibliothek werden, wenn SciDB als DMS verwendet wird.

Die Modellspezifikation soll viele Anleihen aus der relationalen Datenbankmodellierung inkorporieren (als Beispiel: Create-Tensor-Anweisungen aus MLog, die einen sehr an Create-Table-Statements aus SQL erinnern). Die Wahl der MLog-Syntax fiel auf eine deklarative Abfragesprache, die als eine Mischung aus SQL-ähnlichen Anweisungen (Tendenz: erweitertes SQL) und Datalog-ähnlichen Regeln angesehen werden kann. Das Hauptkonzept in der MLog-Sprache sind die Tensor-Views (engl. tensoral views): alle Daten seien Tensoren, alle Operationen darauf seien dementsprechend in der Tensoralgebra ausdrückbar. MLog ist kompatibel mit relationalen Datenmodellen und existierenden Abfragesprachen für Datenmanagement-Modelle, weil jeder Tensor logisch wie eine Relation definiert ist. Ein Tensor T der Dimension D entspreche einer Relation R(T) mit D+1 vielen Attributen (das zusätzliche Attribut ist ein Zahlenwert, auf den der Tensor die entgegegenommenen Vektoren abbildet).

Die Ähnlichkeit von MLog mit Datalog besteht darin, dass ein Teil des MLog-Programms aus Regeln besteht, die im Datalog-Stil aufgebaut sind: *Regelkopf :- Regeloperator(Regelrumpf)*. Die tensoralen Regeln werden vom Abfragenoptimierer in ein echtes Datalog-Programm übersetzt. Somit werden die Datenabhängigkeiten zwischen den Tensoren erfasst und ein statisch optimierter physischer Plan für das Backend (z.B. TensorFlow oder Theano) erstellt. Ein Aspekt der Leistungsoptimierung bei MLog ist das Finden der zusammenhängenden Komponente im Datalog-Programm (mit Pivotverfahren, siehe [SL91]), die parallel ohne Kommunikation ausgeführt werden können – so werden die zu parallelisierenden Codeteile automatisch gefunden. MLog bedient sich also der Optimierungsmöglichkeiten und -Verfahren, die von der Deklarativität von Datalog dargeboten werden.

SQL kommt ins Spiel, wenn der Programmierer die benötigten Tensoren für das MLog-Programm definiert. Dies geschieht mit Hilfe der Create-Tensor-Statements, die syntaktisch und semantisch weitgehend konsistent mit dem SciDB-Interface seien. MLog kann außerdem aus einer relationalen Darstellung einer Tabelle einen Tensor generieren und diesen aus einer Datei mit Daten füllen.

Ein MLog-Programm besteht aus drei Teilen: den bereits genannten 1) Definitionen von Tensoren und 2) Tensor-Regeln sowie 3) den eigentlichen Rechenaufgaben. Der dritte und letzte Teil des Programms – die Rechenaufgabe – ist einfach eine deklarative Abfrage (formuliert als eine Tensor-Regel), in der ein Problem zur automatischen Optimierung vorliegt (z.B. finde Tensorinstanzen, sodass die Verlustfunktion auf ihnen maximal ist). Daran lässt sich erkennen, dass ein MLog-Programm nicht nur über die Datalog-typische Vorwärtsverkettung („wenn Faktum, dann Folgerung“) abgefragt werden kann, sondern auch über Rückwärtsverkettung („wenn Bedingung, dann Faktum“) wie z.B. über die argumentum-maximi-Funktion.

MLog und SQL2Stan haben einige Gemeinsamkeiten: beide Ansätze 1) erweitern die relationale Abstraktion um ML-spezifische Details, 2) lassen den Nutzer eigene ML-Modelle definieren, 3) verstecken die Implementierungsdetails hinter einer SQL-verwandten deklarativen Schicht und 4) (nach meinem Verständnis) ermöglichen den Autausch sowohl vom ML-Backend als auch vom Datenmanagement-System, ohne dass die deklarativ beschrieben Anwendungsschicht umgeschrieben werden muss.

E.2.11. Konzept: Parameterserver für bessere ML-Skalierbarkeit

Während die Skalierbarkeit von den Datenmanagement-Infrastrukturen inzwischen gut erforscht ist, wird die Skalierbarkeit von statistischen Datenanalyse-Algorithmen wird immer relevanter. Viele solche Algorithmen (z.B. MCMC-basierte Inferenzalgorithmen) bauen auf Sampling-Methoden auf, und profitieren von größer werdenden Datenmengen. Die klassischen Inferenzalgorithmen stehen dem Beanspruchen dieser „Mengenprofite“ jedoch im Wege: sie sind nicht so gut skalierbar [JER17a]. SQL2Stan unterstützt Sampling-basierte Inferenz-Backends, und sogar darüber hinaus – dieser nach außen hin abstrakte SQL-Ansatz würde im Inferenz-Backend auch Parameterserver unterstützen. Parameterserver ist ein Konzept zur effizienten Par-

allelisierung samplinggestützter Algorithmen. In diesem Unterkapitel möchte ich dem Leser dieses Konzept näher bringen und damit zeigen, dass es sich lohnt, die genauen Vorgänge statistischer Inferenz vor dem Nutzer wegzuabstrahieren (wie es in SQL2Stan gehandhabt wird).

Es ist einleuchtend, dass man für skalierbare algorithmische Problemlösungen zu verteilten Systemen (Rechnerclustern) greifen muss. Eines der Probleme, welche aus der Umsetzung der verteilten statistischen Inferenz erwachsen, ist die Synchronisierung zwischen den Rechnereinheiten. Wie soll der Zustand eines Samplers, der zum Beispiel für die statistische Inferenz genommen wird, zwischen allen Clusterknoten synchronisiert werden? Die Frage der Synchronisierung kann gelöst werden durch eine spezielle Architektur, welche sich *Parameter Server* nennt. Die damit verbundenen architekturtechnischen Annahmen auf der Ebene des Multikern-Prozessors und, vor allem, auf der Clusterebene werden im Paper [SN10] vorgestellt. Ein Rechnercluster besteht aus mehreren Rechnern; auf der Ebene eines Rechners darf jeder Prozessorkern des Rechners unsynchronisiert Samplingprozesse durchführen. Erst nach dem unter den Prozessorkernen parallelen Sampling werden Aktualisierungen des Problemlösung-Zustands für den einzelnen Rechner durchgeführt. Auf der Clusterebene (d.h. einzelne parallel arbeitende Akteure sind keine Prozessorkerne, sondern Rechner) wird auf eine spezielle Blackboard-Architektur [Nii86, S. 38–53] zurückgegriffen, um die Kommunikation zwischen den Clusterknoten möglichst parallelisierbar zu bewerkstelligen. Das „Schwarze Brett“ (Blackboard) ist ein Software-Architekturmuster zur Organisation von Problemlösungsprozessen. Metaphorisch kann man es sich tatsächlich wie eine Tafel vorstellen, an der mehrere Spezialisten versuchen, ein Problem zu lösen, welches keiner von ihnen allein lösen könnte. Die Spezialisten kommunizieren ausschließlich über die hierarchisch geordneten Einträge am Schwarzen Brett. Dadurch, dass die Beteiligten nicht direkt miteinander kommunizieren, ist der Prozess der Problemlösung sehr gut prarallelisierbar. Ein sehr wichtiges ergänzendes Detail: werden die Informationen an der Tafel ergänzt oder geändert, kann die Schwarze Tafel selbst bestimmte, am Lösungsprozess beteiligte Spezialisten zum Agieren aufrufen.

Das Schwarze Brett im Kontext der Parallelisierung samplinggestützter Prozesse ist ein verteilter Parameter-Server, der die Zustandsvariablen (Parameter) und ihre Werte (Parameterwerte) bereitstellt. Die am Lösungsprozess beteiligten Rechnerinstanzen (Sampler) teilen sich das Schwarze Brett, d.h. sie teilen sich diesen gemeinsamen Satz an Zustandsvariablen, und sie können *unsynchronisiert* die Werte der Zustandsvariablen aktualisieren. Selbstverständlich involviert die Aktualisierung zwar Operationen „lock“ und „release“, allerdings müssen sich die Sampler vor der Aktualisierung einer Variable auf der Blackboard nicht miteinander absprechen.

Im Bereich des Topic Modelling auf verteilten Systemen wurden unterschiedliche Ansätze vorgestellt, doch ihre komplexen, hinsichtlich der Software und Hardware maßgeschneiderten Implementierungen lassen sich in der Praxis nur schwer in die gegebenen Workflows/Pipelines integrieren. Im Kontrast zu solchen *custom*-Lösungen kann auch das weit verbreitete Framework Apache Spark genutzt werden (siehe Projekt APS-LDA mit Glint-Parameterserver [JER17b]), um das Konzept

des Parameter-Servers und des verteilten Samplings zu implementieren. Dadurch kann das an den Kontext des Topic Modelling geknüpfte Modelltraining leichter in die bereits gegebenen Datenverarbeitungs-Pipelines integriert werden. Ein anderer Vorteil, den Apache Spark als ein in-memory-Framework bietet: die Daten werden von Anfang bis zum Ende der Pipeline im Hauptspeicher gehalten, die Zwischenergebnisse müssen nicht auf Festplatten geschrieben werden. Grob gesagt, das Ziel vom Glint-Parameterserver ist, seine zwei grundlegenden Interface-Operationen – 1) Abfrage und 2) Aktualisierung der Parameterwerte, die in einer verteilten Matrix gespeichert seien – effizient für das Spark-Framework zu implementieren, und dabei von den infrastrukturspezifischen Optimierungsmethoden zu profitieren.

Der SQL2Stan-Ansatz unterstützt die Nutzung eines Parameterservers, weil die konkrete Implementierung der bei der Inferenz nutzbaren Samplingprozesse in diesem Ansatz sowieso vollständig verborgen und austauschbar bleibt. Unter der sprachlichen Abstraktionsschicht von SQL2Stan können also auch parallel und verteilt laufende Sampler zwecks Bayes'scher Inferenz laufen. Sogar die Art der Sampler selbst kann theoretisch variieren und vielleicht sogar automatisch (je nach Beschaffenheit von Daten) gewählt werden, wie es im Projekt LDA* [Yut+17] der Fall ist.

E.2.12. Buch: Designing data-intensive applications

Das Buch „Designing data-intensive applications“, geschrieben von M. Kleppmann [Kle17], legt Ideen hinter Big-Data-Infrastrukturen dar. Darin gab es einige Abschnitte, die auch für die SQL2Stan-Thematik relevant waren.

Data Query Languages Im Abschnitt [Kle17, S. 43], wo Abfragesprachen diskutiert werden, nimmt die Deklarativität einen wichtigen Platz ein. Nach dem prinzip der Deklarativität spezifiziert der Programmierer nur das Muster der Ergebnisse, aber keinen Algorithmus, der diese Ergebnisse errechnet. Mit deklarativen Sprachen lassen sich also Implementierungsdetails verstecken. Das System kann automatisch Änderungen und Optimierungen vornehmen, ohne dass eine deklarativ spezifizierte Abfrage geändert werden muss. Deklarative Sprachen sich gut für als Basis für weitere Codegenerierung, weil sie: 1) keine imperativen aufeinanderfolgenden Befehle beinhalten, und die Reihenfolge der Anweisungen somit irrelevant machen [4], 2) viel freundlicher gegenüber Parallelisierung sind, verglichen mit imperativen Sprachen, 3) funktionell eingeschränkter erscheinen im Vergleich zu imperativen Abfragesprachen, was sie für potenzielle Nutzer durch geringere Sprachkomplexität zugänglicher macht und für den Compiler mehr Spielraum für automatische Optimierungen freilässt.

[4] Eine Ausnahme dazu fällt mir an der Stelle dennoch ein: es ist nicht egal, in welcher Reihenfolge man Tabelle mit Create-Table-Statements erzeugt, falls eine Tabellenspalte einen Fremdschlüssel der anderen Tabelle beinhaltet. Es liegt natürlich an der Konsistenzsicherungsmechanismen der Datenbanken und hat insofern keinen Bezug zu imperativen Befehlen. Die Reihenfolge der Anweisungen spielt hier trotzdem eine Rolle.

Batch Processing Systems In einem weiteren Abschnitt [Kle17, S. 390] geht es um Stapelverarbeitungssysteme (engl. batch processing systems). Solch ein System zeichnet sich dadurch aus, dass es in periodischen Abständen einen Stapel Daten entgegennimmt (einen sogenannten Batch Job). Die Verarbeitung von solch einer Stapelverarbeitungsauftrag dauert oft lange (Minuten bis Tage). Es handelt sich also nicht um ein Onlinesystem: der Auftraggeber ist nicht immer ein Nutzer, der davor sitzt und auf den Abschluss der Verarbeitung wartet. Die Leistung solcher Systeme wird gern gemessen am Durchsatz – der Zeit, die das System zum Verarbeiten von einer bestimmten Menge Daten braucht. Die Ausgabe von einem Batch-Job ist oft eine Art von Datenbank [Kle17, S. 412–414]: für die Ausgabe können Abfragen formuliert werden, um an dort enthaltene Informationen ranzukommen. Der Autor beruft sich auf die optimale Praxis, dass für die Ausgabe der Stapelverarbeitung innerhalb des Batch-Jobs (nicht-extern) eine neue Datenbank erstellt werden kann. Nach dem Ende des Batch-Workflows kann diese Datenbank, wo die Ergebnisse des Batch-Jobs drin stehen, nicht mehr verändert werden. Man könnte auch sagen, die Datenbank mit der Ausgabe eines Batch Jobs sei *immutable*. Dies dient u.a. der Minimierung von Irreversibilität. So werden die Batch-Job-Eingaben nicht verändert, was z.B. sehr von Vorteil für agile Softwareentwicklung ist [DF15]. Wenn ein Batch-Job fehlerhaft ist, kann der Programmierer den Fehler beheben und den Job nochmals starten, ohne dass an Daten ein Schaden entsteht. Diese Denkweise [5] kann man sich ganz gut im Falle von SQL2Stan vorstellen, das im Prinzip auch eine Art Batchverarbeitungssystem ist. Denkt man nur an das Onlineshop-Beispiel mit der Kundenklassifikation zurück: man kann den Kundenklassifikator täglich neu trainieren, weil in einem Onlineshop ständig neue Käufe getätigt werden bzw. ständig neue Artikel angesehen werden; man kann diesen Klassifikator ja nicht nur mit Käufen von Kunden trainieren, sondern alternativ auch mit Produktseitenbesuchen und Session-IDs der Besucher. Das wiederholte Modelltraining und die damit einhergehende Inferenz der Werte für latente Modellvariablen wäre somit ein Batch-Job, und das Ergebnis davon wäre eine gefüllte, abfragbare Datenbank, deren ursprüngliche Werte (beobachtbare Daten) nicht überschrieben wurden.

Die Infrastrukturen für Batch-Processing-Systeme charakterisiert der Autor mit Hilfe von drei relevanten Arten [Kle17, S. 415]: Massive Parallel Processing Database (kurz: MPP-DB), Implementierungen von MapReduce [DG08] und Dataflow-Engines. Ich werde versuchen, sie im gegenseitigen Vergleich zu beleuchten und nebenbei das SQL2Stan-Konzept in das Gesamtbild einzuflechten. Um etwas vorzugreifen: SQL2Stan als eine recht hoch angesetzte Softwareschicht (die ja zwischen ML und DMS vermitteln soll) passt im Zusammenhang mit allen drei genannten DMS-Profilen.

[5] Denkweise, dass die Ausgabe des Batch-Jobs oft eine Art Datenbank sei, dass die Batch-Jobs in regelmäßigen Abständen durchlaufen können, und dass sie gefahrlos neugestartet werden können.

MPP-Datenbanken versus MapReduce MPP-DB spezialisieren sich auf die Ausführung analytischer SQL-Queries auf Daten, die über mehrere Rechner verteilt sind. Datenbanken setzen eine bestimmte Strukturierung von Daten voraus, und die Struktur von Daten folgt einem Schema/Modell. Typisch ist die analytische Vorarbeit vor dem Reinladen der Daten. Das heißt, die Daten werden noch vor dem Einlesen in eine Datenbank DBMS-spezifisch modelliert. Bei dem dazugehörigen Entwurf eines Datenbankschemas muss der Nutzer mögliche Abfragemuster auf den Daten berücksichtigen. Durch diese Arbeit vor dem Einlesen der Daten fällt es alelrdings leichter, mit den Daten nach deren Einlesen zu arbeiten. Anders als bei Datenbanken schaut es aus bei der Kombination aus MapReduce und einem verteilten Dateiensystem. Eine Datei im DFS (distributed file system) ist einfach nur eine Folge von Bytes ohne schematische Struktur, und diese Bytefolge kann alles sein: Text, Bilder, Sensordaten usw. Man packt die Daten einfach in ein DFS und überlegt erst später, wie man sie verarbeitet. Dadurch werden die Daten viel schneller verfügbar als z.B. bei MPP-DB, auch wenn sie roh und dadurch anfangs schwerer zu handhaben sind. Das entspricht dem sogenannten „Sushi-Prinzip“: roh sei gut [AJ15], was nach meinem Verständnis dem Datennutzungsprinzip *magnetic* [6] nicht unverwandt bleibt. Das Hadoop-Framework (als Implementierung von MapReduce) bietet die Möglichkeit, die Daten zuerst einfach roh zu sammeln und erst dann zu planen, wie man mit diesem Daten verarbeitungstechnisch vorgeht. Für die analytische Verarbeitung braucht man dennoch eine Struktur, die man über die vorhandenen Daten legen kann. Bei MPP-Datenbanken kümmert man sich um die Strukturierung der Daten im Vorfeld (vor dem Sammeln der Daten), bei MapReduce kann dies auch nach dem Einlesen von Daten geschehen. Beide Herangehensweisen haben eine gemeinsame Eigenschaft: wenn die Daten in dem einen Falle vor dem Einlesen mit einer Struktur versehen werden, oder erst nach dem Einlesen in ein DMS – in beiden Fälle sind die Daten strukturierbar. Im Bezug auf strukturierbare Daten passt der SQL2Stan-Ansatz ganz gut in den Kontext. Bei SQL2Stan kann man theoretisch ein integriertes Schema vor dem Datensammeln formulieren (wie im Prototypen), oder aber auch erst nach dem Einlesen der Daten (wird vom Prototypen nicht unterstützt). Das ist erst dadurch ermöglich, da SQL2Stan den DMS-Aspekt abstrakt behandelt, und daher sowohl MPP-DB als auch MapReduce als DMS unterstützt. Obwohl SQL2Stan mit seiner relationalen Sicht auf Daten sehr einer MPP-DB ähnelt, bedient es sich außerdem algorithmischer Verfahren (MCMC bzw. VI), die nachgewiesenermaßen parallelisiert auf MapReduce laufen können [**lovell2012parallel**; Zha+12].

Ein typisches MPP-DB sei ein monolithisches System aus eng miteinander verbundenen Komponenten (physische Datenverwaltung, Anfrageplanung, -Optimierung und -Ausführung) [Kle17, S. 416]. Diese Komponenten können bei Bedarf extra an spezifische Bedürfnisse einer konkreten DB angepasst werden (typisches Anpas-

[6] *Magnetic, Agile, Deep* als Prinzipien, die das MADlib-Entwicklungsteam als charakteristisch für den modernen Umgang mit zu analysierenden Daten bezeichnete [Hel+12]. Magnetisch bedeutet in dem Sinne: „möglichst viele rohe Daten sammeln, ausgesiebt wird später“.

sungsziel: bestimmte Arten von Anfragen sollen schnell laufen). Hadoop/MapReduce sei im Gegensatz zu monolithisch aufgebauten MPP-DB offener [Kle17, S. 416–417] [Vav+13].

Grenzen von SQL SQL – eine typische Anfragesprache in MPP-Datenbanken – verfügt über elegante Semantik und versteckt geschickt die Implementierung von dem, was in der Datenbank mit den Daten geschieht. Ein Nicht-Programmierer schreibt nur eine Anfrage und keinen Programm-Code. Die Deklarativität der Sprache macht die Datenbanken entschieden zugänglicher. Wenn man sich jedoch mit Machine Learning beschäftigt, stößt man mit einfachem SQL schnell an seine Grenzen. Es gibt zwar Projekte wie BayesDB, SimSQL und SQL2Stan, die sich auf strukturierte Daten und automatische Inferenz spezialisieren und SQL-Interfaces anbieten. Allerdings kann man keineswegs behaupten, SQL sei für alle Batch-Workflows des maschinellen Lernens geeignet. Es liegt daran, dass viele Verarbeitungsprozesse in Machine Learning sehr anwendungsspezifisch sind, und man kommt nicht umher sie zu implementieren. Allein das Formulieren deklarativer Abfragen reicht nicht mehr aus, wenn man anwendungssepzifische Details implementieren muss. Dann man wird gezwungen, zu einem allgemeineren Verarbeitungsmodell zu wechseln und imperativen (also nicht-deklarativen) Code zu schreiben. MapReduce bietet die Möglichkeit an, auf den verteilt gelagerten Daten eigenen imperativen Code auszuführen, und eignet sich daher gut für einige Arten von Machine-Learning-gestützte Batch-Verarbeitung. Zu dieser flexiblen Programmierbarkeit muss allerdings eins angemerkt werden: ohne Abstraktionen, die auf MapReduce aufbauen, ist es für den Nutzer schwierig, eigene, insbesondere komplexe Jobs zu schreiben; man wird z.B. die Join-Algorithmen selbst implementieren müssen [Kle17, S. 419] [Gro+15]. Mit der Programmierung von MapReduce auf höheren Abstraktionsbenen (wie bei Apache Hive [Zah+16; The19d]) geht die Job-Erstellung viel leichter. Interessant ist, dass man mit einer MapReduce-Infrastruktur dennoch nicht auf MPP-DB-typische Abfragesprache SQL verzichten muss. Eine SQL-Query-Engine lässt sich über ein verteiltes Dateisystem (wie z.B. HDFS mit MapReduce) aufsetzen, so wie beispielsweise Apache Hive es tut. Auf diese Weise erhält man sogar zwei Verarbeitungsmodelle für Batch-Jobs – SQL und MapReduce. Doch auf diese zwei Verarbeitungsmodelle reichen nicht immer aus [Kle17, S. 416], daher wurden noch weitere Ansätze auf der Basis von Hadoop vorgeschlagen [Vav+13]. Auch soll es nicht heißen, dass HDFS zur Verwendung von MapReduce prädestiniert: es gibt Datenbanken, die sich der HDFS-Infrastruktur zur Datenverwaltung bedienen, aber kein MapReduce nutzen. Als Beispiele dienen HBase [The19c] (eine Online-Transaction-Processing-Datenbank), Impala [The19e] (analytische DB im MPP-Stil), Apache HAWQ [The19b] (relationale MPP-DB, genauer genommen eine SQL-Abfrage-Engine für Hadoop).

Fehler in Batch-Jobs MPP-Datenbanken versuchen, so viele Daten wie möglich im Hauptspeicher zu halten, um die Anzahl der Festplatten-I/O-Operationen zu

minimieren. Bei technischen Fehlern (z.B. Absturz von einem Cluster-Knoten) versuchen die MPP-DB die Abfrage nochmals durchzuführen. Die Abfrage wird komplett abgebrochen und neugestartet, hauptsächlich weil die Kosten für eine Abfrage (wenige Sekunden bis wenige Minuten Verarbeitungszeit) es einfach meist erlauben. Und dennoch: einen ganzen Job aufgrun5d des Fehlschagens einer Aufgabe neuzustarten ist nicht empfehlenswert – umso weniger, wenn die Daten und die Batch-Jobs größer werden. MapReduce kann hingegen die fehlgeschlagene Aufgabe einfach neustarten. MapReduce-Systeme sind erpicht darauf, die Daten während der Routinen auf die Festplatte zu schreiben [Kle17, S. 417]: zum Teil für mehr Toleranz gegenüber technischen Fehlern und Ausfällen, zum Teil aufgrund der Annahme, dass die Daten sowieso nicht in den Hauptspeicher reinpassen würden. Für MapReduce-Systeme ist ein Batch-Job eine Pipeline aus Aufgaben, und eine Jobaufgabe wartet auf das materialisierte (in eine Systemdatei geschriebene) Ergebnis der anderen Jobaufgabe vor ihr. Diese Pipeline-Bauweise bringt Nachteile mit sich (siehe [Kle17, S. 420]). Um diese Nachteile aus dem Weg zu räumen, entstanden als MapReduce-Nachfolger sogenannte Dataflow-Engines wie Spark oder Flink: sie modellieren den Fluss der Daten über verschiedene Verarbeitungsschritte hinweg. Der gesamte Batch-Job bzw. Workflow ist für die Dataflow-Engines ein einziger Job, und somit keine Kette von einzelnen untergeordneten Jobs. Auch Dataflow-Engines könnten im Hintergrund als DMS für SQL2Stan arbeiten: es gibt zum Einen Ansätze, ML mit Dataflow-Engines zu verheiraten (z.B. Apache Spark und MLlib [Men+16]), und zum Anderen gibt es SQL-Engines für Dataflow-Engines (z.B. Spark SQL [The19h]). Die Zusammenfassung zum sprachlichen Aspekt der DMS – ob SQL oder eine andere high-level-Sprache – verdient einen eigenen Paragraphen.

High-level APIs Das Programmieren von Jobs für verteilte Systeme (z.B. für MapReduce) ohne Abstraktion auf höheren Ebenen kann sehr anspruchsvoll sein [Kle17, S. 426–427]. Man möchte als Nutzer weniger Code schreiben und den geschriebenen Code auch plattformunabhängiger machen. So wurde versucht, die konkrete Implementierung weiter zu abstrahieren, und es entstanden sogenannte high-level APIs. Für MapReduce entstanden high-level-Sprachen und APIs (Beispiele: Hive [The19d], Pig [The19f], Cascading [Cas19]). Auch Dataflow-Engines wie Spark und Flink bieten von sich aus APIs, die in höheren Abstraktionsebenen angesiedelt sind. Der Aufbau von den Dataflow-APIs erinnert an SQL-Blöcke der Queries für relationale Datenbanken: man bedient sich Joins anhand der Werte in bestimmten Feldern, man gruppiert Tupel anhand von bestimmten Schlüsseln, man filtert die Tupel hinsichtlich einer bestimmten Bedingung, aggregiert Tupel usw. Interessanterweise ähneln die Spark-Interfaces wie DataFrames und DataSets weitgehend der Vorstellung von Tabellen, wie man sie von herkömmlichen Datenbanken kennt – und ausgerechnet diese tabellenähnliche Interfaces genießen die Unterstützung automatischer Optimierungsmethoden in Apache Spark.

Die high-level-angesetzte Code-Struktur dient nicht nur dem Zweck, weniger

Code zu schreiben. Sie hilft auch, den Code interaktiv und „nach und nach“ zu schreiben. Ein Entwickler lässt in regelmäßigen Abständen den Code durchlaufen und schaut nach, ob das Verhalten des Geschriebenen in die gewünschte Richtung geht. Die Frameworks, die mit Batch-Jobs auf Daten arbeiten, setzen in letzter Zeit auf Deklarativität in ihren Nutzerinterface-Sprachen [Kle17, S. 427]. Es kann z.B. von Vorteil sein, bei Joins deklarativ vorzugehen. Man setze demnach einen Join als einen relationalen Operator ein („hier wird ein Join gebraucht“) anstatt den Code für den Join-Algorithmus hinzuschreiben. Systeme wie Hive, Spark und Flink unterstützen diese Deklarativität und verfügen über kostenbasierte Anfrageoptimierer, die entscheiden können, welcher Join-Algorithmus in einem konkreten Falle genommen werden soll, und können die Joins sogar umordnen, um die Menge der Zwischenergebnisse zu reduzieren. MapReduce und seine Dataflow-Nachfolger weichen in ihrer nicht-vollkommenen Deklarativität wesentlich von dem vollständig deklarativen Abfragemodell von SQL ab. MapReduce nimmt als Kern die Idee, dass für jede Gruppe von Einträgen eine zustandsfreie nutzerdefinierte Funktion (UDF) aufgerufen wird, – ein Mapper oder ein Reducer. Diese UDF darf beliebigen Code ausführen, um die Ausgabe zu generieren. Diese Freiheit, beliebigen Code in UDFs zu packen, erlaubt es, Werkzeuge aus einem stattlichen Ökosystem aus Bibliotheken auszuwählen, um innerhalb der UDFs Aufgaben wie Parsing, Bildanalyse, numerische/statistische Analyse usw. zu bewältigen. Das freie Einbinden von nutzerdefinierter Zusatzfunktionslität hat den MPP-DB eine lange Zeit gefehlt. Dieser Nachteil wurde z.T. bewältigt, man sah sich dann aber mit weiteren Schwierigkeiten konfrontiert. Auch wenn das Schreiben von UDFs in den MPP-DB zum heutigen Zeitpunkt unterstützt wird, gebe es manchmal Probleme damit. Das Benutzen von UDFs sei manchmal unständlich, und die Integration mit verbreiteten Systemen für Paket- und Dependency-Management (z.B. Maven, npm, Rubygems) lässt einiges zu wünschen übrig [Kle17, S. 428].

Als Fazit zu den high-level APIs kann man folgendes sagen. Viele Systeme, insbesondere Dataflow-Engines, führen deklarative Aspekte in ihre high-level APIs sowie automatische Abfrageoptimierer ein. Sie ähneln somit mehr den MPP-Datenbanken und können vergleichbare Leistungen abliefern, bleiben aber flexibler: sie sehen es nicht so streng mit den Datenformaten, und bieten mehr Freiheit mit beliebig konstruierbaren UDFs abseits der Deklarativität. Für SQL2Stan, was ein deklaratives SQL-Interface bietet, macht es wenig Unterschied, welche Art von Infrastruktur zugrunde liegt. Ob MPP-Datenbanken, MapReduce-Implementierungen oder Dataflow-Engines – alle diese Infrastrukturen bietet auf die eine oder andere Weise SQL-basierte deklarative Abfragesprachen an. Wenn man in SQL die automatische Inferenz (als einer der ML-Bereiche) zugänglicher machen kann, kommt man den Nutzern ein wenig entgegen, egal, welche DMS-Infrastruktur genutzt wird. Der Ansatz sowie der Prototyp von SQL2Stan zeigen: wenn man von Bayes'schen Netzen ausgeht, reichen auch die in einer high-level-API anzusiedelnden SQL-Statements aus, um einen auf einem eigenen Modell basierten Inferenzworkflow zu definieren. Es muss natürlich nicht immer etwas Benutzerdefiniertes sein. Wiederverwendbare, fertige Implementierungen von ML-Algorithmen gibt es innerhalb einer relationalen

MPP-DB (siehe MADlib [Hel+12]) oder als Module für verschiedene Infrastrukturen wie MapReduce-Implementierungen, Apache Spark [Zah+16] oder Apache Flink [Car+15] [Kle17, S. 428].

E.2.13. MADlib

MADlib [MAD19; Hel+12] ist ein Projekt, welches bestimmte vorimplementierte ML- und Datenanalyse-Funktionen in SQL einbindet. Man kann die MADlib-Funktionalität auch eigenhändig erweitern, doch es ist mit signifikantem Implementierungsaufwand (nicht in SQL) verbunden.

M.A.D.: drei Prinzipien für moderne Datenanalyse Die Praxis der Datenanalyse evolutioniert, und die für die Industrie typischen Herangehensweisen altern schnell im Hinblick auf neue Anforderungen [Coh+09]. Zum Beispiel, Unternehmensdatenlager (engl.: enterprise data warehouses, kurz: EDW) dienen als eine zentrale, skalierbare Datenquelle. Sie kann mit Hilfe von Werkzeugen für Geschäftsanalytik (engl.: business intelligence, kurz: BI) von den geschäftlichen Entscheidungsträgern abgefragt werden, und die Abfrageergebnisse werden als Berichte präsentiert, ausgerechnet mit Hilfe von datenintensiven und dennoch hauptsächlich einfachen aggregierenden Algorithmen. Die heutigen Gegebenheiten tragen dazu bei, dass dieses Bild sich ändert. Der Speicherplatz ist sehr günstig. Auch die kleinen Arbeitsgruppen in einem Unternehmen können mit ihrem Budget für eigene Zwecke eine riesige Datenbank auf die Beine stellen. Dabei nimmt auch stets die Anzahl an Quellen zu, aus denen Daten geschöpft werden können. Log-Files der Software oder E-Mails allein können in massiven zu speichernden Datenmengen resultieren; die Daten selbst sind vielfältig und ihre Mengen sind zum Teil enorm. Und schlussendlich wurde die Kultur vieler Unternehmen von der Erkenntnis durchdrungen, dass bessere und „smartere“ Datenanalyse Geld sparen oder sogar Geld einbringen kann. All das führe laut [Coh+09, S. 2] dazu, dass man sich neuen Aspekten der Datenanalyse zuwenden muss, die von der EDW+BI-Sichtweise abweichen. Diese Aspekte haben die Abkürzung **MAD**:

- *M wie Magnetisch*: ein modernes Datenlager (Data Warehouse) komme nicht umher, alle verfügbaren Datenquellen „anzuzapfen“ und die Daten wie ein Magnet anzuziehen – ungeachtet auf die Qualität von Daten.
- *A wie Agil*: die Praxis von Data Warehousing baue darauf auf, dass man langfristig denkt; so werden die Datenlager mit langfristiger Nutzung im Sinn entworfen und umgesetzt. Im Gegensatz dazu soll ein modernes Datenlager es den Analytikern ermöglichen, die Daten so schnell wie möglich aufzunehmen, zu verarbeiten, zu produzieren oder anzupassen. Die physischen und logischen Aspekte der Datenbank sollen agil bleiben, und somit offen für stetige Evolution sein.
- *D wie Deep*: die für BI-Werkzeuge klassischen *Roll-Ups* (Blick auf die Daten aus einer höhergestuften Perspektive, z.B. Monats- statt Wochenauskunft)

> und *Drill-Downs* (Blick auf eine tiefere Detailstufe, „Zoom-In“) reichen nicht aus. Moderne Datenanalysen bedienen sich viel komplexerer Methoden, die weit darüber hinaus gehen. Ein modernes Datenlager soll daher sowohl eine Datenmanagement-Lösung sein als auch eine Infrastruktur, auf der auch komplexe statistische Analysealgorithmen laufen können.

Standardmäßige Data-Mining-Methoden in kommerziell-gerichteten Datenbanksystemen seien zwar hilfreich, aber nicht sehr vielfältig. Aus Hunderten von statistischen Bibliotheken werden nur einige wenige davon von kommerziellen Datenbanken korrespondiert, und diese wenigen Bibliotheken werden typischerweise mit einem statistischen Softwarepaket mitgeliefert (wie R, SAS oder MATLAB) [Coh+09, S. 3]. Hinsichtlich Data-Mining-Aspekte beschränkt sich die Funktionalität von diesen SAoftwarepaketen auf festcodierte „Blackbox“-Implementierungen – eine nicht immer optimale Lösung, insbesondere wenn man bedenkt, dass die vollwertigen statistischen Pakete wie R oder MATLAB den Nutzern durchaus erlauben, die Routinen aus den Funktionsbibliotheken eigenhändig zu modifizieren und zu erweitern. Hoffentlich schimmert in diesem Kontext auf ein wenig durch, dass SQL2Stan mit der nutzerseitigen Modellspezifikation auch ein Stück weit in dieser Richtung mitgeht: der SQL2Stan-Nutzer muss nicht aus vorimplementierten Modellen für automatische Inferenz auswählen, sondern kann mit wenig Aufwand ein eigenes generatives Modell spezifizieren. Zwar händigt das SQL2Stan-System dem Nutzer nicht alle Zügel aus – dieser SQL-Dialekt ist nun mal nicht so flexibel wie die probabilistische Sprache Stan selbst – aber mit bestimmten Einschränkungen ist immer zu rechnen, wenn die Arbeit des Nutzers etwas vereinfacht werden soll.

Die Datenanalyse unter der Berücksichtigung von MAD-Aspekten setze voraus, dass die Analytik sich nicht auf „Blackbox“-Implementierungen beschränken darf – die Anzahl von Fällen, wo die vorimplementierten Data-Mining-Routinen nützlich sein können, fiele bescheiden aus. Die Verfechter des MAD-Prinzips der Datenanlyse plädieren für mehr Flexibilität – u.a. für Modifizierbarkeit und Erweiterbarkeit der Data-Mining-Werkzeuge für Data-Warehouse-Anwendungen.

MADlib als Implementierung der MAD-Prinzipien Aus den Bemühungen, den MAD-Anforderungen an das moderne industrielle Datenmanagement gerecht zu werden, entstand das Projekt Apache MADlib [MAD19; Hel+12]. MADlib ist eine kostenlose, quelloffene Bibliothek für statistische Datenanalyse innerhalb von Datenbanken, sodass die Notwendigkeit im Import und Export von Daten zwischen dem Datenmanagement und der Analytiksoftware entfällt. Diese Bibliothek bietet eine Reihe an fertigen Implementierungen für Machine-Learning-, Data-Mining- sowie statistische Verfahren (siehe `https://madlib.apache.org/docs/v1.10/index.html`). Als fachkundiger Nutzer mit Programmiererfahrung kann man eigene Funktionalität in das MADlib-System hineinbringen, dazu hat es eine C++-Schnittstelle zur Implementierung von den nutzerdefinierten Funktionen (UDF). Es wird dennoch einen gewissen Mehraufwand bedeuten, etwas von Grund auf zu implementieren, wenn einem die in der Funktionsbibliothek enthaltenen Methoden nicht

gefallen oder nicht ausreichen. Für Clustering sowie Topic Modelling gibt es nur k-Means- bzw. LDA-Implementierungen, und wenn man an der Stelle etwas anderes haben möchte, muss man es selbst implementieren bzw. die entsprechende Community darum bitten. Das erschwerte Bauen von eigenen statistischen Modellen ist ein damit einhergehender Kritikpunkt: ich habe in MADlib keine Schnittstelle gefunden, mit der man mit wenig Aufwand eigene statistische Modelle wie in SQL2Stan oder SimSQL definieren kann. Das kann sehr wohl von Nachteil sein, wenn man als ein im Programmieren unerfahrener Nutzer auf die Entwicklung der Funktionsbibliothek warten muss. Konkretes Beispiel: der Naïve-Bayes-Klassifikator, den ein SQL2Stan-Nutzer mit wenig Aufwand selbst in SQL nachimplementieren kann, lässt sich bei MADlib auf sich warten. Als Teil der MADlib-Funktionalität ist Naïve Bayes aktuell in einem frühen Entwicklungsstadium (siehe `https://madlib.apache.org/docs/latest/group__grp__early__stage.html`).

In der Eigenschaft der Datenmanagement-Infrastruktur dienen bei MADlib PostgreSQL-basierte Lösungen [7] – entweder PostgreSQL auf einem einzelnen Rechner, oder aber Greenplum Database [Waa08] als Cluster aus mehreren Rechnern mit PostgreSQL-verwandten Clusterknoten-Instanzen. MADlib kann also eine bei Bedarf skalierbare MPP-Datenbank mit fortgeschrittener „in-database“-Analytik angesehen werden.

E.2.14. MauveDB

Eine interessante Idee, statistische Modell in das Datenmanagement hineinzubringen, liefert das schon etwas ältere Projekt MauveDB [DM06]. Sie besteht darin, zwischen den beobachteten datenbankgelagerten Daten und der Anwendung eine Softwareschicht einzupflegen, die die Qualität der Daten verbessert. Die Schnittstelle dieser Schicht seien Views, die ihre Daten nicht von der Disk, sondern von einem probabilistischen Modell schöpfen. Derartige Views werden in SQL definiert. MauveDB und SQL2Stan sind ebenbürtig in der Hinsicht, dass sie Datenmanagement und komplexe statistische Datenverarbeitung in einer gemeinsamen Abstraktionsebene zusammenführen und dem Nutzer ein einfaches relationales SQL-Interface anbieten. Der Unterschied beider Systeme besteht darin, dass SQL2Stan aus gegebenen Daten und einem gegebenen statistischen Modell Werte für unbekannte Modellgrößen ableitet, während MauveDB versucht, die Qualität der gegebenen Daten zu verbessern, ohne dabei konkrete statistische Modelle zu benötigen.

Als gutes Beispiel für datenbankgelagerte und oft verbesserungsbedürftige Anwendungsdaten dienen Sensordaten. Sensordaten sind oft unvollständig, unpräzise oder einfach falsch – und somit eingeeignet, um direkt in Anwendungen verwendet zu werden. Sie müssen also verarbeitet werden. Die gängige Methode zum Umgang mit derartigen Datenqualitätsproblemen ist die Verwendung von generativen Datenmodellen, mit denen man an eine robustere Interpretation von Daten kommt. Zu dem Zeitpunkt der Veröffentlichung [DM06, S. 1] unterstützten DBMS keine statistischen modellbasierten Workflows auf Datenbankdaten; besonders unhandlich sei

[7] Unabhängig davon fiel die Wahl des DBMS für den SQL2Stan-Prototypen ebenfalls auf PostgreSQL.

es gewesen, die trainierten Modelle regelmäßig zu aktualisieren, sobald neue Daten reinkommen. Viele Wissenschaftler und Ingenieure, die in ihrer Arbeit von generativen Datenmodellen profitieren, nutzen oft gar keine Datenbanken für Archiv- oder Datenabfrage-Zwecke – und falls doch, dann bestenfalls als Dauerspeicher für unverarbeitete Daten. MauveDB versucht, DBMS für diese Leute attraktiver zu machen, und baut auf den Konzepten von traditionellen Datenbanken auf, wonach die physische Herkunft von Daten wegabstrahiert wird.

In einer klassischen Datenbank dient ein View dem Zweck logischer Datenunabhängigkeit. Das MauveDB-Projekt arbeitet mit sogenannten „model-based views“: solch ein View blendet aus, dass die Daten eigentlich aus aus einem generativen Algorithmus stammen, dem ein probabilistisches Modell zugrunde liegt. Die modellbasierten Views dienen als relationale Schnittstelle für Daten mit automatisch verbesserter Qualität. D.h. die Ausgabe von solch einem modellbasierten View kann von den ursprünglichen Datenbankdaten abweichen. Diese Views werden über SQL-Queries abgefragt, doch die Ergebnisse der Abfragen sind nicht direkt Daten aus der Datenbank, sondern ihre statistisch verarbeitete Version. Als eine Schicht zwischen den Datenbankdaten und der Anwendung eignen sich modellbasierte MauveDB-Views als verbesserte Datenquelle für Anwendungen eignen [DM06, S. 3].

Der MauveDB-Ansatz ist nützlich für bestimmte ML-Anwendungen, doch er bietet keine Schnittstelle zum einfachen Einpflegen von eigenen probabilistischen Modellen und setzt sich – im Gegensatz zu BayesDB – nicht zum Ziel, die Erkundung und Säuberung von Daten sowie konfirmative Analyse zu addressieren [Man+15, S. 40–41].

Eine ähnliche Motivation wie MauveDB (Verbesserung von Sensordaten durch probabilistisch gestützte Datenverarbeitung) verfolgt das Projekt BBQ [Des+04]. Die Zielstellung von BBQ: ein Sensorennetz soll man wie eine Datenbank abfragen können, wobei statische Fehlerkorrekturen an den Sensordaten automatisch im Hintergrund laufen. Dazu wurde im Projekt BBQ die Datenbanksprache SQL um bestimmte probabilistische Sprachdetails (z.B. Konfidenzintervalle) erweitert. Doch auch hier, so wie bei MauveDB, darf der Nutzer keine eigenen Modelle bauen. Das liegt vermutlich in beiden Fällen (sowohl bei MauveDB als auch bei BBQ) daran, dass die ML-gestützte Glättung einfacher Sensordaten keine große Modellvielfalt braucht. Man hatte vermutlich nicht wirklich vor, die Nutzer über die wenigen vorimplementierten Modelle hinausgehen zu lassen. Hätten diese Tools ihre Funktionalität um benutzerdefinierbare generative Modelle erweitern wollen, müssten sie auf ähnliche Modellbau-Ansätze wie bei SQL2Stan oder SimSQL zurückgreifen.

E.2.15. Tabular

Das Microsoft-Projekt Tabular ist ähnlich zu SQL2Stan aufgrund des Programmiersprachen-Designs, das sich rund um das relationale Datenschema dreht. Doch im Falle von SQL2Stan geht es um Modellspezifikation in SQL außerhalb des relationalen Schemas, während Tabular-Nutzer direkt im relationalen Spreadsheet-Schema statistische Modelle definieren. Genauer genommen: der Tabular-Programmierer erweitert

das relationale Schema um Modellangaben (was ist bekannt, was ist unbekannt, was muss ermittelt werden) und definiert dadurch eine Inferenzaufgabe.

Die anvisierte Zielgruppe von Tabular und SQL2Stan ist gleich. Es handelt sich um die sogenannten „data enthusiasts“ [Han12] – Spezialisten aus beliebigen Domänen, die sich wünschen etwas zu modellieren und sich mit gängigen relationalen Mitteln der Datenanalyse (SQL bzw. Spreadsheets) auskennen. Sie glauben, dass Daten Antworten zu ihren Fragen liefern können, und dass diese Antworten mit Hilfe von Modellen ans Licht kommen können. Diese Leute sind nicht zwangsläufig Mathematiker oder Programmierer, wissen aber etwas über Verteilungen und relationale Schemata. Trotz ihrer eingeschränkten Erfahrungs- und Wissensbasis soll es für diese Zielgruppe dennoch ermöglicht werden, statistische Modelle zu erstellen und damit neue Erkenntnisse aus Daten zu inferieren.

Tabular hat ein Spreadsheet-Interface: darin werden relationale Tabellen beschrieben, und der Nutzer formuliert statistische Modelle durch die Erweiterung der relationalen Spreadsheet-Schemata. Spreadsheet-Systeme genießen, ähnlich wie relationale DBMS-Systeme, domänenübergreifend eine große Verbreitung. Es dürfte nicht unüblich sein, dass Domänenspezialisten bereits aus eigener Erfahrung Spreadsheets oder SQL kennen, bevor sie sich durch Systeme wie Tabular oder SQL2Stan an probabilistische Programmierung über die für sie bekannten Bedienungskonzepte herantrauen. Durch die Erweiterung der Definition von Spalten des relationalen Schemas um zusätzliche rechenaufgabenbedingte Angaben besitzt Tabular eine leichte Ähnlichkeit zu QEB (Query by Example [Zlo75]); man hat in beiden Fällen keine Textrepräsentation einer Abfrage wie in SQL, sondern einfach ein grafisch dargestellte Tabellenstruktur, die man stellenweise verfeinert.

Spreadsheets und DMS Zugegeben, die meisten der Spreadsheet-Systeme (wie Microsoft Excel) sind keine Datenmanagement-Lösungen und somit ungeeignet für Big-Data-Analyse. Allerdings ist es nicht unmöglich, Spreadsheets mit Cloud-Technologien skalierbar zu machen (z.B. `keikai.io` scheint genau das zu bewerkstelligen).

Interface: Annotation von Spreadsheet-Tabellen Die Idee von Tabular besteht darin, dass man probabilistische Programmierung in einer high-level-Sprache durch die Annotation vom relationalen Schema bewerkstelligtn. D.h. für jede Tabellenspalte wird angegeben, 1) welchen Typ sie hat, 2) ob die Daten beobachtbar oder latent sind und ob sie geschätzt werden müssen sowie 3) wie die Daten in der jeweiligen Relationsspalte verteilt sind (angegeben mit Hilfe statistischer Modellausdrücke). Die Interfacesprache ist kein SQL, aber sie erinnert sehr an SQL-Tabellendefinitionen (Create-Table-Statements). Diese Ähnlichkeit mit SQL-Tabellendefinitionen lässt sich in der Abbildung E.1 erfassen, wo ein Tabular-Programm für das probabilistische TrueSkill-Modell [HMG] abgebildet ist. Das relationale Schema in diesem Tabular-Programm ergibt sich aus zwei Tabellen („Players“ und „Matches“). Die erste Spalte der Tabellendefinition (direkt unter dem Tabellennamen) beinhaltet

Namen der Spalten der jeweiligen Tabelle. Die zweite Spalte der Tabellendefinition beinhaltet die Datentypen für Tabellenspalten. Die dritte Spalte der Tabellendefinition gibt an, ob die Daten der jeweiligen Relationsspalte als Inferenz-Eingabe (input), -Ausgabe (output) dienen sollen oder als nicht zu schätzende Modellentitäten in den Inferenzprozess einbezogen werden müssen (latent). Zuletzt drückt die vierte Spalte der Tabellendefinition aus, wie die latenten Modellvariablen verteilt sind bzw. wie die Inferenzausgabe zu ermitteln ist.

Players			
Name	**string**	**input**	
Skill	**real**	**latent**	Gaussian(25.0,0.01)
Matches			
Player1	**link**(Players)	**input**	
Player2	**link**(Players)	**input**	
Perf1	**real**	**latent**	Gaussian(Player1.Skill,1.0)
Perf2	**real**	**latent**	Gaussian(Player2.Skill,1.0)
Win1	**bool**	**output**	Perf1 > Perf2

Abbildung E.1.: Beispiel des sog. *schema-driven probabilistic programming* in Tabular, aus [Gor+14, S. 1]

Ein Tabular-Nutzer startet also mit einem DB-Schema und erweitert es mit probabilistischen Notationen (siehe E.1). Die Spalten von Relationen entsprechen Variablen eines durch den Nutzer zu formulierenden Bayes'schen Modells.

Das bereits beleuchtete Projekt BayesDB sieht sich strukturell ähnlich zu Tabular, allerdings heben die Projektverantwortlichen besonders hervor, dass Tabular im Gegensatz zu BayesDB die Konzepte und das Vokabular des probabilistischen Modellierens vor dem Nutzer nicht verstecke [Man+15, S. 40]. Tabular spezialisiert sich auf nutzerspezifizierte statistische Modelle, was SQL2Stan auch tut und was BayesDB nicht anbietet.

SQL2Stan vs Tabular: Unterschiede bei der Abbildung von Daten auf Dimensionen Die Erkenntnis von Tabular-Entwicklern, dass grafische Modelle sich auch in einer Datenbanktabellen-Sprache beschreiben lassen, findet man in SQL2Stan wieder. Es gibt jedoch einen Unterschied darin, wie diese Projekte dies tun, und dieser liegt im Begriff der Dimensionen. Ein Leser ohne besondere Lust auf detaillierte technische Vergleiche kann diesen Paragraphen überspringen und getrost mit dem Fazit fortfahren, dass Tabular (im Vergleich zu SQL2Stan) aufgrund bestimmter Vereinfachungen weniger expressiv ist und dadurch manche relevante Bayes'sche Modelle nicht unterstützt.

In SQL2Stan wird jede Nicht-Primärschlüssel-Spalte als Modellvariable angesehen, und die Primärschlüsselspalten definieren die Dimensionalität. Ein zusammengesetzter Schlüssel aus zwei Tabellenspalten impliziert zwei Dimensionen. Ich erlaube mir an der Stelle einfach auf das Kapitel 3.2.2 (Paragraph „Mit Sorting-Order-Constraints zur Array-Ansicht “) dieser SQL2Stan-Arbeit hinzuweisen, in

dem die Konstruktion von ein- und mehrdimensionalen Arrays aus Tabellenspalten erklärt wird. SQL2Stan impliziert Dimensionen aus dem zusammengesetzten Tabellenschlüssel, um die Übersetzung von SQL nach Stan zu ermöglichen, weil Stan-PPL mit Arrays und mehrdimensionalen Arrayzugriffen arbeitet. Tabular sieht es leicht anders: eine Tabelle impliziert nach dessen Konzept immer eine einzige Dimension. Dementsprechend hat eine Tabelle mit einem zusammengesetzten Schlüssel aus zwei Spalten dennoch eine Dimension, definiert durch die Zeilen-ID. Diese Annahme ist zwar recht elegant und erspart zusätzliche Angaben (wie z.B. die Zugriffsreihenfolge über Dimensionen, die sonst entstehen würden). Dadurch verliert Tabular jedoch die Möglichkeit, die Spaltendaten in mehrdimensionalen Arrays unterzubringen, Gruppen von Werten auszuwählen (z.B. ein Zugriff auf ein zweidimensionales Konstrukt über die erste Dimension liefert eine Ausgabe von der Größe der zweiten Dimension) und manche probabilistische Constraints zu definieren (z.B. Summation von gruppierten Werten zu einer 1).

Man kann in Tabular beispielsweise kein LDA-Modell [BNJ03] formulieren, weil Tabular keine Dirichlet-Verteilung zum Bauen von Modellen anbietet. Das Fehlen der Dirichlet-Verteilung (und bestimmt einiger anderen Verteilungsfunktionen) im Modell-Baukasten [Gor+14, S. 325] von Tabular ist vermutlich der Einschränkung auf eine einzige Dimension von Daten geschuldet. Multidimensionale Arrays waren allerdings einer der Gründe, warum SciDB [Sto+13] – ein Array-DBMS für wissenschaftliche Big-Data-Zwecke – als Alternative zu RDBMS [Sto+09] überhaupt erst entstanden ist. Arrays seien ein natürlicheres Mittel zur Modellierung der Daten in vielen Forschungsbereichen [Sto+09, S. 1]. Daher sollte man vielleicht die in Tabular inkorporierte Vereinfachung von mehrdimensionalen Daten auf eine eindimensionale Liste nochmals überdenken.

SQL2Stan vs Tabular: Unterschiede bei Unterabfragen Auffällig ist, dass die Tabular-Modellausdrücke, die die Verteilung von den Daten in einer Spalte definieren, immer nur ganze Spalten als Parameter entgegennehmen. Unterabfragen bei den Parametern, z.B. nicht die ganze Spalte als einer der Parameter, sondern nur bestimmte Werte aus der Spalte sind nicht möglich – ein großer Nachteil. Ich sehe keine Möglichkeit, in Tabular so etwas auszudrücken (angelehnt an den Supervised-Naïve-Bayes-Klassifikator für einen Onlineshop aus dem Kapitel 4): für jeden Produktkauf aus der Tabelle Purchases schlage die Kunden-ID nach, für den jeweiligen Kunden schlage die Kundenklassen-ID nach, und für die jeweilige Kundenklasse schlage die Artikelverteilung nach. Wenn man in Tabular solche sprachlichen Unterzugriffs-Konstrukte erlauben würde, würde man viel mehr Modelle beschreiben können, und zugleich die Eleganz der Tabular-Sprache (ergo die knappe Annotation des relationalen Schemas) durch zunehmende Modellkomplexität zerstören. Um diese Problematik auf die Spitze zu treiben: mit Unterabfragen in den Parametern von Tabular-Modellausdrücken würde der nutzergeschriebene Code vermutlich ähnlich umfassend aussehen wie bei SQL2Stan. Mit Stan lassen sich mehr Modelle schreiben als mit Tabular, da SQL2Stan die Weichen zur Einbettung von Stan in eine

Datenbanksprache stellt.

SQL2Stan vs Tabular: Fazit Sowohl Tabular als SQL2Stan übernehmen für den Nutzer den Transport von Eingabe- und Ausgabedaten zwischen der PPL und DBMS. Tabular schlägt eine SimSQL-ähnliche (und zum Teil auch SQL2Stan-ähnliche) Art vom Programmiersprachen-Design vor, die sich „schema-driven probabilistic programming“ nennt. Man fängt dabei mit einem relationalen Schema an und erweitert es um inferenzspezifische Angaben. SQL2Stan tut das ebenfalls, sogar für die gleiche Zielgruppe, allerdings mit einer anderen, komplexeren Herangehensweise, die sprachlich potenziell mehr bieten kann. Was SimSQL ist welche Rolle dieser neue Name im Kontext probabilistischer Programmierung für Datenbanken spielt, wird der Leser direkt im folgenden Unterkapitel erfahren.

E.2.16. SimSQL

SimSQL [Cai+13] ist ein Prototyp einer verteilten relationalen Big-Data-Plattform für Bayes'sche Inferenz mit Markov-Chain-Monte-Carlo-Algorithmen. Es verknüpft die Datenmanagementaspekte mit ML-Aspekten (vor allem mit automatischer Inferenz) ähnlich wie SQL2Stan, und bietet genau wie SQL2Stan eine deklarative SQL-Schnittstelle zum Bauen Bayes'scher Modelle an. Auch Deep Learning soll mit SimSQL möglich sein, und im Vergleich zu TensorFlow zeigt SimSQL für bestimtme Einsatzzwecke eine bessere Leistung [Jan+19].

Ein SimSQL-Nutzer kann in SQL mit wenig Programmieraufwand Markov-Ketten spezifizieren, simulieren und abfragen. Die Markov-Ketten werden von einer verteilten Dateninfrastruktur mit beobachteten Modelldaten versorgt. Mit anderen Worten kann der Nutzer Inferenzaufgaben eigenhändig definieren, und zwar auf eine deklarative, relationale Weise [Cai+13, S. 1]. Dazu formuliert er (ähnlich wie bei SQL2Stan) für seine Daten einen modellspezifischen generativen Algorithmus; das Bauen von eigenen Bayes'schen Modellen ist somit problemlos möglich.

SimSQL ist der Nachfolger vom Projekt MCDB [Jam+08], eines prototypischen SQL-basierten DB-Systems, welches zwecks stochastischer Analyse auf DB-Daten entworfen wurde.

SQL2Stan ist SimSQL konzeptionell ähnlich, und SimSQL geht in der Implementierung von seinem Vorhaben deutlich weiter (es konnte z.B. gezeigt werden, dass SimSQL-Prototyp mit Hilfe eines Hadoop-Clusters skalierbar ist). SimSQL und SQL2Stan nehmen unterschiedliche Wege, um die Rechenprobleme der statistischen Inferenz in SQL zu definieren.

Probabilistische Programmierung innerhalb des Datenschemas Das zentrale Mittel der probabilistischen Programmierung in SimSQL ist eine relationale Tabelle [Cai+13, S. 1]. Dabei sind zwei Arten von Tabellen zu unterscheiden: einerseits die DB-Tabellen mit beobachteten Daten sowie stochastische Tabellen andererseits. Letztere werden zum Beschreiben von generativen Prozessen genutzt, d.h. zur Spezifikation statistischer Bayes'scher Modelle.

Eine stochastische Tabelle kann man sich als Ergebnis eines Zufallsprozesses vorstellen: die Tabelleninhalte sind nicht die in der Datenbank gelagerten Daten, sondern Daten, die durch statistische Verteilungen simuliert werden. Ein Teil der Einträge in stochastischen Tabellen sind Zufallsvariablen (inkl. dazugehörigen Wahrscheinlichkeitsverteilungen), der andere Teil der Einträge darin sind deterministische, gegebene/beobachtete Datenwerte. Stochastische Tabellen können interessanterweise nicht nur von originalen Datenbank-Tabellen abhängen, sondern auch von anderen stochastischen Tabellen. Somit kann man stochastische Tabellen rekursiv definieren; manchmal wird aus dem „kann“ ein „muss“, wenn es sich um hierarchische Modelle wie LDA handelt. MCDB [Jam+08], der Vorgänger von SimSQL, ließ den Nutzer noch keine hierarchischen Modelle definieren, und die SimSQL-typische rekursive Erstellung von mehreren stochastischen Tabellen (und ihre Verwaltung durch Indizes im Tabellennamen) war die neue Lösung für dieses Problem. Im Vergleich dazu wurde der SQL2Stan-Dialekt direkt mit starker Referenz auf eine ausgewachsene PPL Stan und hierarchische statistische Modelle entworfen; somit sollte es bei SQL2Stan weniger MCDB-ähnliche schwerwiegende inhärente Expressivitätprobleme geben.

Stochastische Tabellen in SimSQL werden mit Hilfe von Create-Table-Statements beschrieben; sprachlich gesehen also in SQL. Wichtig zu wissen, dass es bei SimSQL kein reines SQL ist (bei SQL2Stan hingegen schon), weil die Create-Table-Statements u.a. For-Each-Schleifen verwenden, was im SQL-Sprachstandard nicht vorgesehen ist. Der SimSQL-Dialekt benötigt also definitiv einen eigenen Parser. Der SQL2Stan-Dialekt braucht, im Gegensatz zu SimSQL, keinen eigenen Parser (PostgreSQL-Parser reicht aus), und ist dadurch offener gegenüber der Entwicklung neuer Compiler durch Drittentwickler. Demzufolge könnte beispielsweise jemand mit Hilfe eines SQL-2-Tensorflow-Probability-Compilers die SQL2Stan-Spezifikationen nicht nach Stan, sondern nach Edward2 für TensorFlow übersetzen.

Der SQL-Dialekt von SimSQL ist notwendigerweise erweitert worden um „variable generation“-Funktionen (kurz: VG-Funktionen), um dem Programmierer den Bau eigener Bayes'schen Modelle in SQL zu ermöglichen. Solche VG-Funktionen (wie z.B. Dirichlet, Normal oder Multinomial) findet man in den meisten PPL, auch in Stan sowie SQL2Stan. VG-Funktionen bilden aus meiner Sicht einen wichtigen Teil vom syntaktischen Baukasten zum Zusammenstellen von generativen Algorithmen durch den Nutzer. Durch VG-Funktionen wird die Unbestimmtheit von Daten verankert, allerdings nicht im relationalen Datenschema: in SimSQL wird sie erst in stochastischen Tabellen mit VG-Funktionsaufrufen sichtbar [Cai+13, S. 2]. Die Tabellendefinitionen in SQL2Stan können gewiss auch als eine Art von stochastischen Tabellen (allerdings ohne Anteile probabilistischer Programmierung) gesehen werden.

Durch stochastische Tabellen simulierte Daten können wie alle anderen Daten über SQL abgefragt werden. Das gibt dem Nutzer die Möglichkeit, anstelle von unbestimmten Daten einfach statistische Verteilungen zu nehmen. Unbestimmt sind beispielsweise Daten, die nie eingetragen wurden und ergänzt werden wollen bzw. Daten, die man vorhersagen will. Ein kritischer Punkt ist, dass der SimSQL-Nutzer

auch bei Abfragen von unbestimmten Daten mit SQL kein Standard-SQL nutzt, sondern SimSQL-spezifische Compute-Statements. Diese Compute-Abfragen sind deutlich komplexer gestaltet als klassische Select-Queries, u.a. weil sie algorithmische Details zum Inferenzprozess erfordern (z.B. Anzahl der MCMC-gestützten Schätzungsiterationen). Das trägt keineswegs zur Einfachheit vom Interface bei, kann einen Datenenthusiasten etwas überfordern und schränkt dazu noch die algorithmische Flexibilität von SimSQL ein (man muss zwecks Inferenz ja nicht unbedingt einen MCMC-Algorithmus nehmen). SQL2Stan bietet hingegen ein Interface, welches man als Nutzer von einer relationalen Datenbank erwarten würde: alle Tabellen sind vor und nach der statistischen Inferenz mit den für SQL ganz klassischen Select-Queries abfragbar; im Hintergrund kann bei SQL2Stan sogar im Prototypen MCMC oder aber VI als Inferenzalgorithmus laufen.

Komplexität relationaler Schemata: SQL2Stan versus SimSQL Die Spezifikation eines statistischen Modells erfolgt in SimSQL durch das Definieren von stochastischen Tabellen, ergo zusätzlichen modellspezifischen Tabellen. Für eine bedingte Modellvariable ist eine versionierte stochastische Tabelle vonnöten. Die Versionierung erfolgt mit Indizes, und die Evolution der stochastischen Tabellen von der einen Version zur nächsten läuft schleifenweise ab. Das SimSQL-System versucht, nicht nur die Ein- und Ausgabe der Inferenz in das relationale Schema einzubetten (wie SQL2Stan es tut). SimSQL versucht darüber hinaus die Modellspezifikation (Teile des probabilistischen Programms sind direkt in der Tabellendefinitionen drin) sowie auch die Zwischenergebnisse der Inferenz (über Inhalte der probabilistischen Tabellen mit einem bestimmten Index) in das relationale Schema einzubetten. In SQL2Stan erfolgt die Modellspezifikation hingegen abseits der Tabellendefinitionen, in einem extra SQL-Statement mit der modellspezifischen log-likelihood-Abfrage. Die Tabellen in SQL2Stan werden im Gegensatz zu SimSQL nicht rekursiv ausgebaut, weil dafür keine Notwendigkeit besteht (nicht einmal bei hierarchischen Modellen wie LDA): jede stochastische Tabelle wird nur einmal definiert und mit inferenzspezifischen Informationen versehen (ähnlich wie bei Tabular). In SQL2Stan wird vielmehr darauf gebaut, dass man ein Bayes'sches Modell in ein relationalen Schema übersetzen kann, welches sich mit dem relationalen Datenbankschema der gegebenen Daten verschmelzen lässt [RH16]. Doch auch wenn man die Daten und das Modell in einer gemeinsamen Datenmanagementabstraktion (im integrierten Schema) unterbringt, braucht der Inferenzprozess eine Implementierung. Sie kann in SQL2Stan aus der modellspezifischen log-likelihood-Abfrage abgeleitet werden.

Die Beschreibung der Modellstruktur liegt bei SQL2Stan also abseits der Datenmanagementabstraktion, anders als bei SimSQL. Der Aufbau vom statistischen Modell wird in SQL2Stan demzufolge nicht in den Create-Table-Anweisungen beschrieben, sondern in einer zusätzlichen SQL-Abfrage.

SimSQL beruft sich, im Gegensatz zu SQL2Stan, auf kein integriertes Schema, welches mit Hilfe bestimmter Verfahren (wie [RH16]) aus einer grafischen Beschreibung des Bayes'schen Modells übersetzt werden kann. Bei SimSQL gibt es ein-

fach ein Datenschema für die gegebenen Daten. Die modellspezifischen latenten, zu schätzenden Modellvariablen kommen hinzu durch die zusätzliche Spezifikation stochastischer Tabellen. Pro statistisch bedingte Modellvariable muss eine oder mehrere stochastische Tabelle(n) definiert werden. Durch die iterative (in Schleifen formulierte) Definition von ebendiesen stochastischen Tabellen wächst das Datenbankschema. Man bringt nämlich die Zustände von den stochastischen Tabellen in das Datenbankschema hinein; eine neue Version der Tabelle impliziert einen neuen Tabellennamen, und somit eine zusätzliche Tabelle im relationalen Schema, darum auch das Wachstum des relationalen Schemas. Derartige Komplikationen sind bei SQL2Stan nicht vonnöten, und das relationale Schema wirkt in SQL2Stan deutlich übersichtlicher und einfacher.

Der SimSQL-Ansatz setzt also voraus, dass für einen latenten Modellparameter eine Reihe stochastischer Tabellen erzeugt wird, die durch Indizes auseinanderzuhalten sind. Nach der statistischen Inferenz liegen die Inferenzergebnisse in den stochastischen Tabellen vor. Eine stochastische SimSQL-Tabelle mit einem bestimmten Index ist die API für den Zugriff auf die Schätzungsergebnisse einer Modellvariable. Zusatzinformationen müssen bei Bedarf aus ursprünglichen Tabellen mit gegebenen Daten herange-JOIN-ed (oder im Vorfeld als Ausgabespalten der jeweiligen stochastischen Tabelle spezifiziert) werden; diese Umstände sind in SQL2Stan aufgrund des integrierten Schemas viel seltener (wenn überhaupt) vonnöten. Das integrierte DB-Schema in SQL2Stan dient als API zwischen der Inferenzausgabe und der an sie gebundenen Anwendung. Der Anwendungsprogrammierer, der mit dieser API arbeitet, muss dementsprechend nur wissen, wie die Ein- und Ausgabe vom Inferenzprozess aussieht, ohne sich mit statistischer Modellierung und Verteilungen auszukennen. Bei SimSQL muss der Anwendungsprogrammierer mehr angeben und verstehen: die Anzahl der Schätzungsiterationen (d.h. die Version/Indexzahl der benötigten stochastischen Tabelle) sowie die Anzahl der unabhängigen Sampling-Pfade, über die gemittelt wird, sollen angegeben werden.

Die Unterschiede in den SQL-Interfaces von SQL2Stan bzw. SimSQL liegen auch daran, dass SQL2Stan sich sprachlich an die Stan-PPL orientiert, und die deklarativen SQL-formulierten Tabellenbeschreibungen zusammen mit dem zusätzlichen generativen Modell-Verteilungsstatement nach Stan übersetzt werden. Es wäre interessant zu wissen, ob man SQL2Stan-formulierte deklarative Inferenzaufgaben nach SimSQL übersetzen kann. SimSQL ist schon viel fortgeschrittener als der SQL2Stan-Prototyp. Vielleicht könnte man SimSQL (und somit indirekt Hadoop) zur Datenverwaltung nehmen und den Nutzern mit SQL2Stan einfach eine andere Art vom deklarativen probabilistischen SQL-Interface anbieten, die leichter zu überschauen und zu bedienen ist. Die Übersetzung von SQL2Stan nach SimSQL scheint zumindest auf den ersten Blick nicht unmöglich zu sein.

Fazit: SimSQL im Vergleich zu SQL2Stan Wenn man sprachliche Anforderungen der Systeme SQL2Stan und SimSQL vergleicht, fällt auf, dass SQL2Stan für einen SQL-Programmierer einfacher ist.

SQL2Stan wird über reines SQL ohne inferezalgorithmische Details bedient, während SimSQL erweitertes SQL nutzt und für gleiche Zielstellungen die Angabe zusätzlicher komplexer Details erfordert. SQL2Stan ist darauf bedacht, dass das relationale Schema als DMS-Schnittstelle der Inferenz möglichst leicht zu erzeugen (aus einer grafischen Darstellung eines Bayes'schen Modells ableitbar), möglichst einfach gestaltet (probabilistische Programmierung in reinem SQL abseits vom relationalen Schema) und ohne Zusatzwissen mit Standard-SQL-Queries abfragbar ist. SimSQL schlägt keine Leitfäden vor zur einfachen Erzeugung des relationalen Schemas als DMS-Inferenzschnittstelle. SimSQL bringt die probabilistische Programmierung im relationalen Schema unter und macht es auf eine schwieriger zu durchschauende Weise (Versionierung von Tabellen, Abfragen der Tabellen über detailüberladene Queries). Bei SQL2Stan kann man bei Bedarf zwischen zwei Nutzerrollen wählen: der Die-Inferenzergebnisse-Abfragende und der Die-Modellspezifikation-Durchführende, und für jede dieser Nutzerrollen ist das SQL-Interface einfach. Bei SimSQL sind diese Rollen schwer zu trennen, was zu Komplikationen führen kann.

E.2.17. Expressivität deklarativer Sprachen für ML-Algorithmen

Machine Learning und Deklarativität werden öfter im Zusammenhang angesprochen, und die Bezüge zum Datenmanagement sind fast immer erkennbar.

Markov-Chain-Monte-Carlo und Deklarativität Einigen Forschern ist aufgefallen, dass man probabilistische Algorithmen wie Markov-Chain-Monte-Carlo (kurz MCMC) auch in deklarativen Sprachen beschreiben kann. MCMC wird in vielen Bereichen angewendet, wie beispielsweise Bioinformatik oder statistische Physik, und dessen Anwendungen spielen eine zunehmend größere Rolle in theoretischer Informatik [DKM10, S. 1]. Auch Stan als Software für probabilistische Programmierung benutzt u.a. MCMC als algorithmisches Backend für automatische Inferenz.

In [DKM10] werden Ideen präsentiert, wie man MCMC in einer deklarative, Datalog-ähnlichen Sprache ausrücken kann. Mit Hilfe von Deklarativität verfoglt man mehrere Ziele: Unabhängigkeit von der Einsatzdomäne, Wiederverwendbarkeit von Algorithmen, Erhöhung von Produktivität und im Allgemeinen das Ermöglichen der Programmierung von MCMC-Anwendungen auf einer höheren Abstraktionsebene. Man verspricht sich, durch Nutzung von höheren Abstraktionsebenen beim Programmieren mit Markov-Ketten neue Erkenntnisse zu gewinnen über 1) die Ähnlichkeiten zwischen den Anwendungen, die MCMC verwenden, 2) die Möglichkeiten der Zerlegung von diesen Anwendungen in ihre Komponenten und 3) das Kombinieren von den algorithmischen Komponenten zu neuen, komplexen Anwendungen. Dazu werden deklarative Sprachen (z.B. Datalog) aus der datenbankbezogenen Literatur für Komplexitäts- und Experessivitätsanalysen herangezogen [DKM10, S. 1]. Ich möchte darauf verweisen, dass im MLog-Projekt (siehe Abschnitt E.2.10) ausgerechnet Datalog-Methoden zur Optimierung sowie erweitertes SQL verwendet werden, um Machine Learning deklarativ mit dem Datenmanagement zu verknüpfen.

Als Fazit sollte man unterstreichen, dass deklarative Abfragesprachen nicht nur nachgewiesenermaßen [DKM10] zur MCMC-Aufgabenspezifikation eignen; darüber hinaus wird daran geforscht, wie die Auswertung solcher deklarativen ML-Abfragen automatisch optimiert werden kann. Das befürwortet die Wahl von SQL als Interfacesprache im Prototyp eines Inferenzsystems, welcher im Rahmen dieser Arbeit implementiert wurde.

Verbesserung der Expressivität durch prozedurale Sprachanteile Ein Fazit für diesen Paragraphen kommt schon im Voraus: durch Erweiterung einer deklarativen Sprache um prozedurale Sprachkonstrukte lässt sich die Expressivität dieser Sprache verbessern, ohne dass die automatische Optimierung darunter leidet [Shi+16]. Für SQL2Stan bedeutet das eine Art Plan B, falls man nicht alles in reinem SQL beschreiben kann und stattdessen auf eine Mischung aus SQL und einer prozeduralen Programmiersprache setzen muss. D.h. sollte es sich herausstellen, dass der SQL2Stan-Ansatz eine nicht-deklarative Spracherweiterung benötigt, wäre solch eine Spracherweiterung nicht zwingend mit Einbüßen der Optimierbarkeit verbunden.

Manche Bereiche, wie beispielsweise Ad-Hoc-Datenverarbeitung [8], legen einen großen Wert auf die Optimierbarkeit der deklarativen Abfragen. Außerdem sind die deklarativen Abfragen oft leicht zu formulieren. Zur Beschreibung komplexer Probleme können die Nutzer manchmal wollen, hauptsächlich prozedural zu programmieren (mit Konstrukten *for, do* und *while*) und die deklarativen Abfragen (z.B. in SQL) in den prozedural geschriebenen Code zu packen. Sprachlich geht das gut, allerdings können manche Systeme einen auf diese Weise geschriebenen Code nicht gut optimieren, weil unterschiedliche Engines mit der Ausführung vom Code aus unterschiedlichen Programmierparadigmen aufgetragen werden. Der prozedural geschriebene Code wird an die eine Ausführungsengine geleitet, die die eingebetteten deklarativen Code-Teile als Blackboxes handhabt. Eine andere Engine optimiert und führt den deklarativen Code aus. Auch wenn beide Arten von Engines ihre Arbeit gut erledigen, geht bei dieser Blackbox-Sichtweise viel Optimierungspotenzial verloren. Das Projekt UniAD [Shi+16] visiert diese Problematik an und schlägt vor, wie man die Ausführung vom prozeduralen Code mit eingebetteten deklarativen Abfragen unifiziert optimieren kann.

SQL-Deklarativität für Explanation Engines Explanation Engines sind Datenanalyse-Werkzeuge, die den Nutzer bei einer Reihe von ML-Aufgaben unterstützen, darunter 1) Feature Selection (mit anderen Worten, Auswahl von (Wirkungs-)Variablen für das Modelltraining) und 2) Extraktion von Gemeinsamkeiten von Datenpunkten [Abu+18, S. 1]. Sie sind anwendbar für beispielsweise Analyse vom Nutzerverhalten oder zur Ursachenermittlung. Explanation Engines seien nach aktuellem Stand eigenständige Datenverarbeitungssysteme, und damit einhergehend nicht nutzbar

[8] Der Nutzer formuliert schnellerhand einige Abfragen bezogen auf einen Datensatz, beobachtet die Ausgabe, ändert die Abfrage nach und nach – und lenkt die Ausgabe somit ad-hoc in die von ihm gewollte Richtung.

in Verbindung mit traditionellen, SQL-basierten Workflows. Das schränke ihre Anwendbarkeit und Erweiterbarkeit ein. Das Projekt DIFF [Abu+18] stellt daher einen relationalen aggregierenden Operator *diff* vor, mit dem die ML-gestützte erklärende Funktionalität der Explanation Engines mit der deklarativen relationalen Abfrageverarbeitung in SQL zusammengebracht werden kann. Wie man sieht, wird neben der Bayes'schen Inferenz mindestens ein anderer ML-Bereich in den SQL-sprechenden Datenbankkontext übertragen.

E.2.18. SystemML

Es muss natürlich nicht immer eine Datenbanksprache sein, um ML-Aufgaben deklarativ zu definieren. Eine Lücke, die von SystemML [The19a; Boe+16] geschlossen wird, ist das deklarative Machine Learning auf verteilten Systemen. Dekalrativität ist im ML-Bereich also ausdrücklich erwünscht.

Während es bei SQL2Stan darum geht, die probabilistische Programmierung für automatische Bayes'sche Inferenz deklarativ mit klassischen DBMS-Sprachmitteln zu gestalten, spezialisiert sich SystemML auf deklaratives Machine Learning im viel breiteren Sinne, und beschränkt die Vision auf ML nicht auf Bayes'sche Inferenz.

SystemML visiert Machine Learning auf verteilten Systemen (vor allem Apache Spark [Zah+16]) an. Der Nutzer spezifiziert ML-Algorithmen in einer R- bzw. Python-ähnlichen Syntax, mit dementsprechend typischen imperativen Sprachkostrukten. Die nutzerbeschriebenen Aufgaben werden in das Ökosystem von Apache Spark integriert; die Verwaltung der verteilten Dateninfrastruktur wird von Spark übernommen.

Die Zielgruppe für SystemML ist klar eine andere als bei SQL2Stan: man orientiert sich an erfahrene Programmierer, die neue ML-Algorithmen für spezielle Big-Data-Zwecke entwerfen wollen. Die Nutzung von diesem System setzt fundierte Kenntnisse in ML-Anwendungsdesign und -Programmierung voraus. Allein durch die Erfahrungs- und Wissensvoraussetzungen wären die typischen Nutzer keine Datenenthusiasten (wie bei SQL2Stan), die mit wenig Aufwand ein spezielles ML-Modell entwerfen und etwas aus ihren Daten inferieren möchten. Somit lohnt es sich nicht, die zu unterschiedlichen deklarativen Sprachkonzepte von SQL2Stan und SystemML zu vergleichen.

E.2.19. TensorFlow Extended

Der Aspekt der Integration von ML mit DMS (und anderen Komponenten) nicht *ineinander*, sondern *miteinander* (wie von SQL2Stan anvisiert), wird zum Teil vom Framework TensorFlow Extended angesprochen. TensorFlow Extended (kurz: TFX) [Bay+17] ist eine Ende-zu-Ende-Plattform zum Erstellen und Unterhalten von ML-Pipelines.

Warum mehr Komponenten außer ML und DMS? Oft geht es auch um viel mehr als nur Datenmanagement in ML-gestützten Anwendungen. Die Komponenten, wel-

che das maschinelle Lernen umgeben, gehören dazu: beispielsweise Datenanalyse, -Transformation und -Validierung sowie insbesondere das „Serving“ (der Einsatz von gelernten Modellen). In vielen realen Einsatzfeldern von Machine Learning (z.B. bei Empfehlungsdienten) ändern sich die Datensätze ständig, und es müssen ständig neue trainierte Modelle produziert werden. D.h. Modelltraining und -Nutzung sollen ununterbrochen laufen, und die trainierten ML-Modelle müssen robust und valide sein. Es läuft darauf hinaus, dass nicht nur das eigentliche Schreiben des ML-Codes, sondern auch alles rund um das maschinelle Lernen orchestriert werden muss, damit die ständig produzierten trainierten ML-Modelle auch sinnvoll eingesetzt werden können. Die den ML-Code umgebenden Komponenten sind z.B. „Learner“ zum Produzieren von gelernten Modellen anhand von Daten, Module zur Analyse und Validierung sowohl von Daten als auch Modellen, und die eigentliche Infrastruktur, welche den Zugriff auf die ML-Modell-gestützen Dienste anbietet.

Das Organisieren von den im Workflow zusammenwirkenden Komponenten kann allerdings zeitaufwändig sein. Die ML-Aufgaben variieren, und der für das Komponentenzusammenspiel verantwortliche Code-„Kleber“ kann oft nicht wiederverwendet werden, weil die Komponenten je nach Aufgabe unterschiedlichen Bedürfnissen (z.B. Datenrepräsentation, Speicherinfrastruktur) gerecht sein müssen. Mit größerem Anteil an zweckgebundenem, ja maßgeschneidertem Code verlieren ML-Dienste an Flexibilität. Dabei wird mancher Code wird daher mehrfach geschrieben. Der Aufwand beim Erstellen und Warten eines ML-Dienstes veranlasste Google dazu, eine Plattform zu bauen, die generisch genug ist, um auf ihr die Entwicklung von allen – auch untypischen – ML-gestützten Aufgaben zu ermöglichen. Die Plattformkomponenten sollen verwendbar und zusammenstellbar sein, egal welchen ML-Dienst man auf dieser Plattform baut. Diese Plattform nennt sich TensorFlow Extended (kurz TFX) [Bay+17] soll eine Ende-zu-Ende-ML-Plattform sein, mit standardisierten Komponenten, einfacher Plattformkonfiguration, schnelleren Produktionszyklen von ML-gestützten Diensten und Robustheit gegenüber vielen möglichen Fehlern (beispielsweise inkonsistente Daten, inoptimale Nutzerkonfiguration oder Fehler in den unterliegenden Softwareschichten).

High-level APIs Auch bei TFX lässt sich die Tendenz zu high-level-APIs wiederfinden. Viele konkrete Implementierungsdetails versucht man vor dem Nutzer zu verstecken und dafür deklarative Ansätze einzusetzen [Bay+17, S. 1391–1392] – vom Gedankenhergang nicht unähnlich zu dem, was SQL2Stan seinerseits versucht zu bewerkstelligen. TFX ist eine Allzweck-ML-Plattform: das, was TensorFlow und seine Komponenten können, wird für mehr Wiederverwendbarkeit, Nutzerfreundlichkeit und Robustheit unter Produktionsbedingungen ausgelegt. Ein großer Wert wird dabei auf das Serving – das Nutzen/Einsetzen vom gelernten Modell – gelegt, welches vom Modul mit dem Namen TensorFlow Serving übernommen wird.

Hier, genauso wie bei SystemML, umfasst ein System sehr viel mehr als deklarative probabilistische Programmierung zwecks automatischer Inferenz, weshalb man SQL2Stan mit diesen Frameworks nicht wirklich vergleichend auf eine Stufe setzen

kann. Auch die Zielgruppen von SQL2Stan und TFX sind zweifellos unterschiedlich (keine SQL-Programmierer, sondern ML-Spezialisten). Doch der Kern mancher Argumente – mehr Abstraktion, wenig Aufwand für den Nutzer – bleibt dem Wesen vom SQL2Stan-Ansatz verwandt.

E.2.20. Clipper

Serving wird als wichtiger Teil der ML-Workflows betrachtet, und zwar nicht nur im oben erwähnten TensorFlow-Extended-Projekt. Clipper [Cra+17] ist ein System für Vorhersagesysteme mit niedrigen Antwortzeiten, und es konzentriert sich auf Serving. Dabei soll die Qualität vom Serving eines Vorhersagesystems verbessert werden. Qualität ist in diesem Zusammenhang gleichzusetzen mit geringen Wartezeiten, hohem Durchsatz und guter Vorhersagegenauigkeit. Serving wird im SQL2Stan-Kontext wiederholt angesprochen, weil SQL2Stan versucht, die gleichen Qualitätsmerkmale beim Serving zu gewährleisten. Das Serving im Falle von SQL2Stan ist das Abfragen der Ergebnisse von Bayes'scher Inferenz; die Schnittstelle dafür ist eine relationale DMS-Schnittstelle. DMS verfügen über automatische Abfrageoptimierung, um geringe Wartezeiten zwischen einer Abfrage und der Ausgabe vom Abfrageergebnis sicherzustellen. Hoher Durchsatz (statistische Verarbeitung von möglichst vielen Daten in einem möglichst knappen Zeitraum) und gute Vorhersagegenauigkeit (direkt verbunden mit der Zuverlässigkeit der Inferenzergebnisse) sind Qualitätsmerkmale, bei denen sich SQL2Stan auf die Eigenschaften des ML-Backends und dessen Inferenzalgorithmen verlässt. SQL2Stan bemüht sich also um qualitativ gutes Serving in Form der Wiederverwendung von Inferenzergebnissen; Clipper beschränkt sich nicht auf Bayes'sche Inferenz. Sowohl SQL2Stan als auch Clipper machen sich Abstraktionsmethoden zunutze.

Die Clipper-Architektur besteht im Groben aus zwei Schichten: 1) der Modellabstraktions- sowie 2) der darüberliegenden Modellauswahl-Schicht. In der ersten Schicht wird eine API angeboten, die die Heterogenität der ML-Werkzeuge und -Modelle abstrakt verbirgt [Cra+17, S. 613]. Diese API erlaubt ein für die Anwendung transparentes Ändern bzw. Austauschen von Modellen. Die Modellauswahl-Schicht liegt auf der ersten Schicht und lässt verschiedene Modelle in einem Qualitätsvergleich gegeneinander antreten. Die Vorhersagen, geliefert von unterschiedlichen Modellen, werden in der Modellauswahl-Schicht dynamisch ausgewählt und kombiniert.

Grundlegend in dieser Architektur ist das Isolieren von der Anwendung von den Unterschiedlichkeiten der ML-Frameworks durch Abstraktion. Bei einer bereits laufenden Anwendung kann das darunterliegende Modell oder ein ML-Framework ausgetauscht werden. Modelle werden in der Clipper-API als eine Art „Blackbox" verwendet. Manche Ideen aus dem Clipper-Projekt findet man auch in SQL2Stan wieder: die Komplexität und Heterogenität der ML-Plattformen werden für die das gelernte Modell nutzende Schicht wegabstrahiert, und das Serving soll für den Nutzer zugänglicher gemacht werden (bei SQL2Stan: Modellnutzung via SQL-Abfrage).

E.2.21. Verknüpfung von Deep Learning mit Datenmanagement

Für Deep Learning als einen Bereich von ML wurde untersucht [Wan+16a], in welcher technologischen Beziehung Deep Learning und Datenmanagement zueinander stehen. Es stellte sich heraus, dass diese zwei unterschiedliche Bereiche sich technologisch gar nicht mal so unähnlich sind: z.B. verteiltes Rechnen und Speichermanagement nehmen sowohl in Datenbanken als auch bei Deep Learning eine zentrale Rolle ein [Wan+16a, S. 17]. Wie man sieht, Bayes'sche Inferenz ist keineswegs die einzige ML-Sparte, deren Verbindung mit DMS-Konzepten wissenschaftlich untersucht wird.

Datenbanken binden in ihre klassischen Rechenprobleme (Indizes, Transaktions- und Speichermanagement) weniger Unsicherheiten bei Daten ein. Deep Learning ist hingegen gut, um die mit Unsicherheit verbundenen Werte vorherzusagen. Das kann zugute kommen, falls man sich mit der Lösung von Datenbankproblemen beschäftigt. Abgesehen von typischen DB-Problemen gibt es „knowledge fusion" [Don+14] oder „crowdsourcing" [Ooi+14]. Das sind zwar Datenbankprobleme, doch sie sind probabilistisch. Für solche und weitere DB-Probleme – wie z.B. Arbeit rund um die Abfragepläne – lassen sich Deep-Learning-Techniken einsetzen (siehe [Wan+16a, S. 20–21] für mehr).

Datenbanken bieten ihrerseits Technologien, die im Bereich Deep Learning willkommen sind [Wan+16a, S. 18–20], wie beispielsweise Operation Scheduling sowie Speichermanagement kommen dem stand-alone-Modelltraining in Deep Learning zugute. Beim verteilten Training spielen solche Datenbankstärken eine Rolle wie beispielsweise Kommunikation und Synchronisierung von Clusterknoten, Parallelisierung, Konsistenzsicherung sowie Fehlertoleranz. Beide Seiten – sowohl ML als auch DBMS – können etwas also voneinander lernen.

E.2.22. Rafiki

Das Projekt Rafiki [Wan+18] beschäftigt sich ebenfalls mit der Zusammenführung von Machine Learning – hauptsächlich Deep Learning – und Datenmanagement. Grob ausgedrückt geht es bei Rafiki u.a. darum, dass man ML-Funktionalität als einen Cloud-Service an das Datenmanagement anbindet. So kann z.B. ein SQL-Programmierer direkt im SQL-Code einen ML-gestützten Klassifikator als eine SQL-Funktion nutzen. Der Klassifikator selbst muss allerdings im Vorfeld von einem Deep-Learning-Experten über eine Rafiki-Schnittstelle programmiert, trainiert und als einen Cloud-Service verfügbar gemacht werden. Die SQL-Schnittstelle von Rafiki hat Ähnlichkeiten mit bereits geschilderten System MADlib (siehe Anhangskapitel E.2.13): in beiden Systemen sind ML-Modelle über SQL-Funktionen zugänglich, und in beiden Fällen kann der SQL-Programmierer keine ML-Modelle eigenhändig in SQL spezifizieren – er kann nur die ML-Modelle verwenden, die von anderen Spezialisten zur Nutzung in SQL implementiert wurden. Der Leser wird bereits jetzt gemerkt haben, dass die Rollen von dem „Verwerter der ML-Workflowergebnisse" (greift über SQL auf das trainierte ML-Modell zu) und des „ML-Programmierers"

(definiert das ML-Modell) leicht trennbar sind – diese Rollenverteilung findet man in SQL2Stan wieder.

Motivation von Rafiki Das Anwenden vom maschinellen Lernen auf großen Datensätzen (Big Data), so Rafiki, wurde zu einer impliziten Voraussetzung oder zumindest einer Erwartung für die meisten analytischen Aufgaben. Es sei im Allgemeinen dennoch nicht leicht, ML-Anwendungen zu entwickeln, denn ML-Modelle seien weder leicht zu bauen noch zu trainieren [Wan+18, S. 12]. Es gibt also „Schmerzpunkte“ beim Implementieren und Anpassen von datensatzspezifischen Modellen. Dazu können Datenbanknutzer nicht problemlos ML-gestützte Anwendungen mit den für sie vertrauten DB-Technologien entwickeln. Es ist fast unmöglich, Datenbanken vollumfänglich mit ML-Funktionalität auszustatten – man kann nicht alle ML-Modelle in einer DBMS-Engine unterbringen. Außerdem setzt die nutzerseitige Optimierung vom Modelltraining und dem Serving voraus, dass man entsprechendes Expertenwissen im ML-bereich besitzt. Und trotz alledem existieren datenbankgebundene Big-Data-Anwendungen, die in den Genuss vom maschinellen Lernen kommen wollen.

Traditionelle Datenbanksysteme können mit großen Datenmengen umgehen und werden traditionell zum Speichern und Analysieren von strukturierten Daten (oder aber räumliche-zeitlichen Angaben, Graphen usw.) verwendet. Andere Arten von Daten – wie Bilder, Text oder domänenspezifische Daten (medizinische Einträge, Sensordaten usw.) – bilden einen bedeutenden Anteil an Big Data [Wan+16b]. Es lohnt sich, auch diese Daten zu analysieren, um die Analyseergebnisse in datenbankbasierten Anwendungen zu verwenden. Ein Beispiel dafür wäre Stimmungserkennung („sentiment detection“) in den DB-gelagerten Produktrezensionen, die einem helfen kann, die Verkaufszahlen zu interpretieren [Wan+16a, S. 1]. Eine graphische Darstellung einer dafür geeigneten Datenanalyse-Pipeline ist in der Abb. E.2 zu finden.

Abbildung E.2.: Datenanalyse-Pipeline nach [Wan+18, S. 1]

Für die ersten drei Etappen wurden klassischerweise Datenbanken genutzt. Machine Learning kann seine Stärke insbesondere im vierten Schritt ausspielen. Die Pipeline impliziert das Anbinden von ML an das Datenmanagement auf die eine oder andere Weise. Eine der möglichen Option hierzu: das Bereitstellen von ML-Funktionalität über eine Cloud nach dem Prinzip „Software as a service“. Mit diesem Ansatz beschäftigt sich Rafiki – ein System zum Bereitstellen von Training- und Inferenzdiensten für ML-Modelle über eine Cloud. Durch das Auslagern der ML-spezifischen Komplexität in eine Cloud werden vor dem (Datenbank-)Nutzer einige Details wegabstrahiert, wie beispielsweise: Hardware-Ressourcenverwaltung,

Zusammenstellen von Deep-Learning-Modellen, Hyperparameter-Tuning, Optimierung der Vorhersagegenauigkeit usw. Stattdessen muss der Nutzer nur seine Datensätze hochladen, das System zum Lernen konfigurieren und das gelernte Modell für das Serving freigeben [Wan+18, S. 2].

Rollenverteilung: Nutzer und ML-Experte Das Ausblenden von den Details der Deep-Learning-Algorithmik macht das System zugänglicher für Datenbanknutzer: der DB-Nutzer schreibt SQL-Abfragen und kann darin bestimmte nutzerdefinierte Funktionen aufrufen (UDF). Die UDF kommunizieren über eine Web-API mit Rafiki und stellen die vom Rafiki-Clouddienst errechneten Inferenzergebnisse durch. Das Vorbereiten vom Rafiki-Dienst erfordert jedoch Tätigkeit eines Deep-Learning-Experten, der mit Hilfe von Apache SINGA [Apa19] Python-Skripte schreiben muss, die den Ablauf vom Training und Serving von einem Deep-Learning-Modell festlegen [Wan+18, S. 12].

Bezug zu SQL2Stan Wie passt Rafiki zum SQL2Stan-Kontext? Es geht in diesem Projekt im Gegensatz zu SQL2Stan nicht um automatische Bayes'sche Inferenz und das einfachere Bauen von bayesschen Modellen im Datenbank-Umfeld. Es geht bei Rafiki um Deep Learning als ein bequem aus SQL heraus abfragbarer Cloud-Service, der von Deep-Learning-Spezialisten für die Nutzung durch Nicht-Spezialisten vorbereitet werden kann. Sowohl SQL2Stan als auch Rafiki erwachsen aus dem Wunsch, einen Teil von Machine Learning für die große Datenbank-Community zugänglicher zu machen. Sie beide verwenden Abstraktionen für konkrete algorithmische ML-Vorgänge und betten die ML-Funktionalität in SQL ein.

E.2.23. ease.ml

Die foranschreitende Entwicklung von Machine Learning – aus der Sicht von ease.ml [DS319; Li+17a] insbesondere im Bereich neuronaler Netze – bringen neue Herausforderungen mit sich. Für einige Anwendungsszenarien liefern speziell dazu entworfene und gut optimierte ML-Modelle gute Lösungen, die mit den menschengemachten Lösungen vergleichbar sind. Diese Tendenz beobachtet man vor allem an Beispielen der Bildklassifikation und Stimmenerkennung. Dennoch bleibt es in vielen Situationen nicht immer klar, was geeignete Modelle sind und wie man die Modellparameter anpassen soll, wenn man eine ML-Aufgabe im Sinn hat [Kar+18b].

In letzter Zeit gab es einige Arbeiten zur Thematik, wie man maschinelles Lernen zugänglicher für Endnutzer ohne ML-Expertise machen soll (u.a. auch SQL2Stan). Machine Learning soll deklarativer werden – so lautet hier das Motto. Deklarativ bedeutet im Kontext der ML-Workflows mit neuronalen Netzen, dass den Nutzer nicht mehr für Modellauswahl und -Tuning zuständig ist. Ease.ml wurde entworfen mit Rücksicht auf effizienten Mehrnutzerbetrieb und Fähigkeit zur Erweiterung des Systems um neue Modelle und Funktionen. Es stelle ein Ende-zu-Ende-System dar, das 1) das Problem der automatische Modellselektion aus einer kostenbewussten

Mehrnutzer-Perspektive adressiert und 2) das deklarative Nutzerinterface auf einer höheren Abstraktionsebene anbietet. Der Systemnutzer soll demnach nur das Ein- und Ausgabeschema definieren, und das System erledige den Rest (Schema-Matching, Modellauswahl usw.). Den Workflow kann man sich wie folgt vorstellen [Kar+18b, S. 2055]:

- Nutzer lädt seine Daten hoch.
- ease.ml inferiert aus den Daten ein Schema und zeigt es dem Nutzer, der es ggf. manuell anpasst.
- ease.ml sucht in den bereits bekannten Modellen nach Kandidaten, mit denen das Schema gematched werden kann.
- ease.ml trainiert Modelle und informiert den Nutzer interaktiv darüber, welches Modell momentan am besten zur Anwendung passt.

  Sobald das erste Modell trainiert ist, wird es zum Serving freigegeben. Falls ease.ml während des Modellservings feststellt, dass ein anderes Modell doch besser ist, schaltet es automatisch zum besseren Modell um, ohne die Serving-API zu ändern.
- Die Menge der dem System bekannten Modelle wird von den Entwicklern erweitert, sobald es neue relevante Veröffentlichungen gibt (z.B. eine neue Architektur für neuronale Netze zum Klassifizieren von Bildern).

SQL2Stan beschäftigt sich zwar nicht mit neuronalen Netzen, und dennoch findet man sehr verwandte Ansätze in SQL2Stan und ease.ml – mehr Deklarativität, mehr Implementierungsdetails wegabstrahieren, mehr Zugänglichkeit gegenüber Nutzern ohne ML-Expertise.

E.2.24. DAWN: Data Analytics for What's Next

Das Projekt DAWN (`https://dawn.cs.stanford.edu/`) an der Stanford-Universität beschäftigt sich mit der Entwicklung von Werkzeugen für Ende-zu-Ende–ML-Anwendungsentwicklung und Demokratisierung von der Entwicklung der Anwendungen mit künstlicher Intelligenz. DAWN ist ein Akronym für das namensgebende Motto *Data Analytics for What's Next*. DAWN wird als Teil des Related Work deshalb aufgeführt, weil es die Zugänglichmachung von ML für breitere Massen thematisiert (so wie SQL2Stan) und als Pendant zur anderen Ende-zu-Ende-ML-Plattform TensorFlow angesehen werden kann.

In einem Satz: DAWN Das Projekt kümmert sich um die Bereitstellung von einem Softwarestack zur ML-Anwendungsentwicklung, damit die Kosten der Inbetriebnahme von Machine Learning für Domänenspezialisten gesenkt werden können.

Motivation von DAWN Der Einsatz von Machine Learning hat ein enormes Potenzial, dennoch ist es oft noch zu teuer und zeitaufwendig durch Mangel an Ende-zu-Ende-Werkzeugen, um die Anwendungsentwicklung im gesamten Workflow zu unterstützen. Neue ML-Anwendungen sind kostspielig und benötigen große Teams von Domänenexperten, Datenwissenschaftlern und DevOps (siehe z.B. Apple Siri, Amazon Alexa). Auch innerhalb von Organisationen, denen ML bereits zu Diensten steht, ist ML eine seltene Gunst, die nur für einen kleinen Teil von Teams und Anwendungen einsatztauglich ist [Bai+17].

Ferner benötigen viele ML-Modelle nahezu Unmengen an Trainingsdaten, welche zu bekommen mitunter schwer und/oder teuer ist (je nach Anwendungsbereich). Z.B. das Erkennen von Hunden auf Bildern kann sehr gut klappen, da man Millionen von entsprechend gelabelten Bildern im Internet finden kann. Eine ganz andere Geschichte ist, Krebs auf Bildern zu erkennen – das geht leider nicht so leicht, es sei denn, man investiert Jahre von Experten-Arbeitszeit, um ausreichend viele gelabelte Trainingsdaten zu erzeugen.

Ein fertiges ML-Produkt braucht auch Wartung: es muss bereitgestellt, betrieben und überwacht werden, besonders wenn kritische Geschäftsprozesse davon abhängen.

Das Projekt dreht sich um eine zentrale Frage [Bai+17, S. 1]: wie soll jeder Domänenexperte eigenhändig qualitative Datenanwendungen entwickeln können, ohne ein Team von PhD in Machine Learning, Big Data und verteilten Systemen heranzuziehen und ohne sich mit moderner Hardware auszukennen?

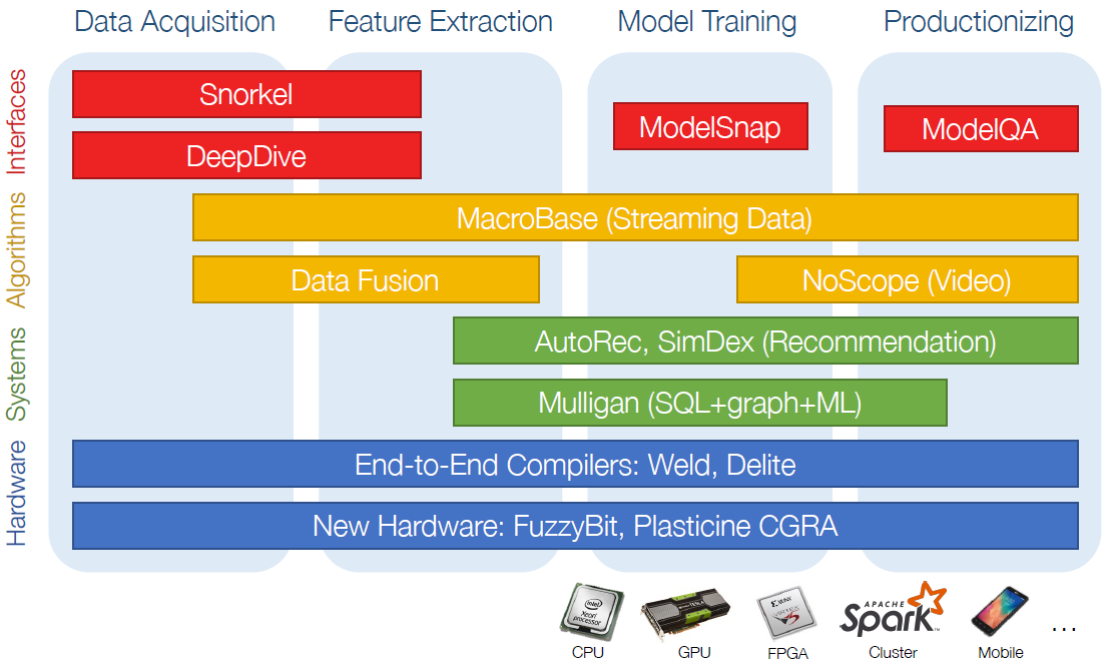


Abbildung E.3.: DAWN-Softwarestack

Eine wichtige Beobachtung, die von den Projektzuständigen erwähnt wird: der Großteil vom Aufwand in industriellen ML-Anwendungen steckt *nicht* im Ausdenken neuer Algorithmen oder Modelle, sondern in anderen Bereichen wie Datenvorverarbeitung, Auslese von Eigenschaften und Indienststellung/Wartung des fertigen Produkts. Diese Bereiche brauchen bessere Werkzeuge und Infrastrukturen.

Zur Datenvorverarbeitung gehören das Sammeln, Produzieren und Säubern von genügend qualitativ möglichst hochwertigen Trainingsdaten, die in ML-Algorithmen eingespeist werden. Ohne diese Daten kann ML nicht funktionieren.

Die Auswahl sowie Extraktion von Eigenschaften bedeutet zu entscheiden, welche Aspekte von Daten sind am wichtigsten sind. Was würde z.B. ein Domänenexperte implizit oder explizit über einen gegebenen Datenpunkt sagen?

Der Bereich „Wartung“ umfasst Indienststellung, Überwachung, Debugging von einem fertigen Produkt sowie die Robustheit des Produktes ggü. Änderungen in den Daten. [Bai+17, S. 2]

Dementsprechend sind die Grundsätze von DAWN wie folgt ausgelegt [Bai+17, S. 2]:

- Ende-zu-Ende-ML-Workflow: nicht nur Modelltraining, sondern der gesamte Workflow (von Datenvorverarbeitung bis Wartung, siehe Abb. E.3)
- Stärkung der Rolle von Domänenexperten: sie sollen ihre Domänenkenntnisse maschinell auswertbar niederschreiben können. Die Systeme sollen den Nutzern ohne ML-Expertise entsprechende Werkzeuge bereitstellen.
- Ende-zu-Ende-Optimierung: der Workflow soll schnell sein, sowohl im Modelltraining als auch im Betrieb. Die Hardware soll besser ausgenutzt werden. Statistische Algorithmen sollen an bestimmten Stellen toleranter ggü. Rechengenauigkeit werden, was einen Geschwindigkeitsschub bringt.

Die Schwerpunkte der Forschung im Rahmen des DAWN-Projekts [Bai+17, S. 3–5] werden in folgenden Paragraphen dargelegt.

Neue Interfaces für ML ..., angefangen bei Modellspezifikation bis hin zur Modellüberwachung, gerichtet an Experten ohne tiefergehende ML-Kenntnisse. Modellspezifikation soll vereinfacht werden durch empirisches ML (Bereiche: Datenvorverarbeitung, Eigenschaftenverarbeitung). Die damit einhergehende Frage lautet: ist es möglich, ML-Systeme zu entwickeln, indem man einfach beobachtet, wie die Domänen-Experten beim Labeling der Daten vorgehen? Nach meinem Verständnis ist an dieser Stelle das Unterprojekt *Snorkel* [Rat+17] relevant. Die Domänen-Experten nutzen oft bestimmte heuristische Regeln, um ein Label für einen Datenpunkt zu definieren. Zum Beispiel, wenn in einer Dokumentensammlung oft die Aussage „Ofen vorheizen“ vorkommt, wird der Domänenexperte sicherlich behaupten, darin gehe es wohl ums Kochen. Bei *Snorkel* muss der Nutzer Labeling-Funktionen schreiben (ergo Codestücke, die den Daten heuristisch Label zuweisen). Diese nutzergeschriebenen Funktionen werden vom System als eine implizite Beschreibung

eines generativen Modells für echte Label gehandhabt. Mit den Labeling-Funktionen versucht das System, Modelle aus diesen Regeln zu liefern (Regelsatz auf Datensatz anwenden → unüberwachtes ML, u.a. Entrauschen von Regeln, Lernen von deren Genauigkeit → Trainieren von überwachten ML-Modellen mit entstandenen probabilistischen Labels). Es gibt hierbei keinen traditionellen Trainingsdatensatz und der Nutzer betreibt selbst auch kein *feature engineering* [Rat+17]. **Menschenverständliche Erklärung von Ergebnissen** (Bereiche: Eigenschaftenverarbeitung, Wartung) werden hinsichtlich neuer Iterfaces auch relevant. Gegeben eine ML-Implementierung, wie sind die Ergebnisse des ML-Modells zu interpretieren? Große, komplexe Modelle können sehr genaue Ergebnisse liefern, aber deren Interpretation ist schwierig; aktuell scheint die korrelationsbasierte Analyse von Attributen zu funktionieren. Ein anderer Aspekt betrifft ebenso die neuen ML-Interfaces – **Debugging und Beobachtbarkeit** (Bereiche: Eigenschaftenverarbeitung, Wartung). Die beobachtbaren Phänomene entwickeln sich weiter, und die Modelle sollen es auch tun (z.B. die Panne mit Google Flu Trends, welches 2008 hervorragend funktionierte, und 2013 mit der Vorhersage komplett fehlschlug). Sobald die ML-Modelle in Betrieb genommen werden, müssen sie beobachtet und aktualisiert werden. Dazu braucht man kostengünstige, nützliche Tools, um die Qualität der ML-Modelle zu beobachten.

Ende-zu-Ende-ML-Systeme ..., die den gesamten ML-Workflow beinhalten und ihre internen Bestandteile vor dem Nutzer verbergen (wie z.B. bei einer Suchmaschine oder einer SQL-Datenbank [Bai+18]). Dazu gehören:

- **Klassifikation innerhalb massiver Datenströme** (Bereiche: Datenvorverarbeitung, Eigenschaftenverarbeitung, Wartung), oder genauer gesagt, Klassifikation und Aggregation von Daten (wie Sensordaten, Video-Verarbeitung) mit bekannten Ansätzen wie Caching, inkrementelle Memoization oder branch-and-bound-Schnitte im Lösungsraum – alles in einem unifiziertem Framework, zusammen mit einem Software-Werkzeugkasten für Klassifikation (daher auch Ende-zu-Ende).

- **Personalisierte Vorschläge** (Bereiche: Eigenschaftenverarbeitung, Wartung). Trotz der Einfachheit von Ein- und Ausgaben, und hinzu auch reichlicher Vertretung von Algorithmen für personalisierte Vorschläge in der Literatur muss in der Praxis jede *recommendation engine* praktisch von Grund auf gebaut werden, indem man low-level-Algorithmen und Werkzeuge miteinander verkettet.

  Das DAWN-Projekt will eine Ende-zu-Ende-Plattform für personalisierte Vorschläge anbieten, die ein einfaches Interface für Eingaben anbietet, sowie automatische Modellanpassung, -Überwachung und -Neutraining.

- **Kombinieren von der Vorhersage und dem Treffen einer Entscheidung** (Bereiche: Datenvorverarbeitung, Eigenschaftenverarbeitung, Wartung).

Heutzutage wird die Kombination von Vorhersage („was wird passieren“) und der damit ausgelösten Aktion („was ist zu machen, wenn etwas passiert“) von verschiedenen Systemen erledigt, z.B. von einer automatisierten Vorhersage-Engine und einem menschlichen „Entscheider“. Ausnahmen dazu gibt es wenig (z.B. ein selbstfahrendes Auto).

DAWN soll die Integration der Kombination „Vorhersage–Aktion“ ermöglichen, angefangen bei bloßen Benachrichtigungen bis hin zu physischen Manipulation der Umgebung (beispielsweise „sende einen Roomba-Roboter, der überprüfen soll, ob ein Student im Raum sitzt“).

- **Vereinheitlichung von SQL, Graphen und linearer Algebra** (Bereich: Wartung). ML-Pipelines bestehen aus verschiedenen Operationen wie SQL, Berechnung auf Graphen, ML-Training und -Evaluierung. Die meisten Ausführungs-Engines bedienen sich der Optimierungsmethoden nur für eins von diesen Berechnungsmustern.

  Das Ziel von DAWN sei daher u.a. eine Ausführungs-Engine, die für alle Berechnungsmuster optimiert ist. Viele von diesen Berechnungsmustern lassen sich in eine Join-Instanz umwandeln (ein relationaler Join). Die Engine optimiert die Ausführung nach dem Prinzip Single-Instruction–Multiple-Data. Ein optimierter Join-Operator, optimiert auf diese Weise, läuft schnell sowohl für SQL als auch für Graphen. Indem man zu den Operationen mit SQL- und Graphen noch klassische ML-Berechnungsmuster hinzufügt (wie Operationen aus linearer Algebra oder Operationen auf dünnbesetzten Matrizen), sollte die auch Optimierung der ML-Workloads durch die Ausführungs-Engine hinzukommen.

Neue Nährböden für das Wachstum von ML Schnelles und kosteneffektives Training sowie sowie eine ebenso optimale Werkzeug-Indienststellung im Bereich ML erfordern neue rechnerische Grundlagen (von der Unterstützung von Programmiersprachen bis zu den *distributed runtimes* und beschleunigter Hardware). Zu diesen Grundlagen gehören:

- **Compiler für die Ende-zu-Ende-Optimierung** (Bereiche: Eigenschaftenverarbeitung, Wartung). Moderne ML-Anwendungen beinhalten eine immer buntere Mischung aus Bibliotheken und Produkten wie *Apache Spark, TensorFlow, scikit-learn* und *Pandas*. Auch wenn jedes von diesen Produkten für sich allein optimiert ist – die Pipelines aus der Praxis beinhalten mehrere Bibliotheken. So kann die Produktion in größeren Maßstäben ein ganzes Team von Entwicklern erfordern, das die ganze Anwendung in low-level-Code auf eine optimale Weise neuschreiben muss.

  Eine neue Laufzeitumgebung *Weld* soll dieser Problematik vorbeugen: sie soll daten-intensiven Code zwischen unterschiedlichen Bibliotheken und Funktionen optimieren, und so automatisch schnelle Implementierungen für ML-

Aufgaben (Training oder Anwendung) generieren. *Weld* kann bereits die Datenanalyse mit *Spark, Pandas* und *TensorFlow* durch die Optimierung von den Operatoren darin um das Zehnfache beschleunigen; die Beschleunigung von den Anwendungen, die diese Produkte kombinieren, kann das 30-fache erreichen. *Weld* soll auch mit heterogenee Hardware kompatibel sein (FPGAs, mobile Prozessoren, GPUs).

- **Reduzierte Genauigkeit und nichtexakte Verarbeitung** (Bereich: Wartung). ML-Operatoren sind stochastisch und probabilistisch. Das lässt sich vorteilhaft auszunutzen, indem man Asynchronität und Stochastik in die Berechnungsalgorithmen hereinbringt (siehe z.B. das Projekt *HogWild!*), um die Konvergenzzeit zu verbessern.

  Es sollte außerdem möglich sein, Berechnungen mit low-precision-Arithmetik (Stichwort *HogWild!-style algorithms*) auszuführen, ohne die Genauigkeit zu kompromittieren.

- Einsatz **rekonfigurierbarer Hardware** (Bereiche: Eigenschaftenverarbeitung, Wartung). Berechnung wird immer mehr ein Flaschenhals für datenintensive ML-Analysen, sowohl beim Training als auch bei der Inferenz. Die Tendenz: FPGAs werden immer wichtiger als eine hochgradig programmierbare Alternative zu den CPUs (auch im Bezug auf das Verhältnis Leistung-pro-Watt).

- **Verteilte Laufzeitumgebungen** (Bereich: Wartung). Das Kombinieren von ML mit verteilten Systemen ist schwer, und kann viele Fragen aufwerfen: verhält sich ein Modell inoptimal, weil es auf zu viele Server verteilt sind, oder weil es inoptimal spezifiziert ist? Was ist der optimale Anteil an Asynchronität? Wie soll ein optimales Framework für das verteilte Modelltraining aussehen? Welche Vorteile kann ein verteiltes System bei der Inferenz bringen, wenn man das Modell einsetzen soll?

  Das wichtigste Interessenfeld hier ist die Ausnutzung aller verfügbaren Ressourcen, die einem Cluster zur Verfügung stehen.

Wichtige Punkte, an denen der Nutzen von DAWN gemessen werden soll, sind zum Einen die Zeit und die Kosten für die Spezifikation eine ML-Anwendung (inkl. Fertigstellung von Datenquellen sowie Auswahl von relevanten Eigenschaften), ebenso wie der Zeit- und Kostenaufwand für den Einsatz der Anwendung unter Produktionsbedingungen (unter Berücksichtigung der Hardware und menschliche Aufsicht). Auch der Nutzgrad für den Endnutzer soll mit in die Bewertung der Güte von diesem Softwarestack einfließen. Die Integration von den Komponenten aus verschiedenen Schichten vom DAWN-Stack soll essentiell sein, und u.a. durch Hackathons u.ä. unterstützt werden.

Fazit im Zusammenhang mit SQL2Stan SQL2Stan ist selbstverständlich kein Softwarestack zum Bauen von KI-Anwendungen und verfügt nicht über derart gro-

ße Ambitionen, doch dieser Ansatz hebt im Kontext Bayes'scher Inferenz hervor, dass neue Nutzerschnittstellen für ML-Anwendungen auch für Nichtspezialisten zugänglicher gemacht werden sollen, und dass das Zusammenspiel von den an der Anwendungsentwicklung beteiligten Komponenten (wie, im Falle von SQL2Stan, das Datenmanagement und automatische Inferenz) ausgebaut werden soll.

E.2.25. Publikation: Cohort Query Processing

Im Umfeld von der sogenannten Informatik des menschlichen Verhaltens (engl.: *behavioral informatics*), ist die statistische Kohortenanalyse (*cohort analysis*) einer der dort genutzten datenanalytischen Ansätze. Statistische Untersuchung des menschlichen Verhaltens ist eine der Domänen, wo SQL als eine deklarative Sprache den Umgang mit den Analysealgorithmen erleichtern kann.

Bei der Kohortenanalyse ist eine Kohorte ist eine Gruppe von Personen. Da diese Art von Analyse aus der Informatik des menschlichen Verhaltens ruht, muss darin auch das Verhalten verankert werden. Der Anteil der Verhaltens in der Kohortenanalyse lässt sich darauf zurückzuführen, dass es zwei Arten von Aspekten geben kann, die das menschliche Verhalten beeinflussen können [Jia+16, S. 1]: zum Einen, das Älterwerden, und zum Anderen soziale Veränderungen (Personen verhalten sich anders, wenn die Gesellschaft anders ist). Mit der sozialwissenschaftlich veranlagten Kohortenanalyse kann man u.a. diese Effekte auf das Verhalten mit Hilfe von statistischen Methoden auseinanderhalten und getrennt untersuchen.

Auch in diesem Falle läuft es darauf hinaus, dass man strukturierte Daten relational abbildet und in SQL deklarativ domänenspezifische statistische Rechenaufgaben für die Daten definiert. Im Paper [Jia+16] werden Ansätze vorgeschlagen, wie man Kohortenanalyse im relationalen DBMS-Kontext betreiben kann. Man bedient sich des relationalen Datenmodells (wie in gängigen DBMS). Sprachlich gesehen wird für diesen Zweck die Datenbanksprache SQL um drei neue Operatoren erweitert, damit der Nutzer kohortenanalytische Aufgaben deklarativ beschreiben kann. Die Deklarativität vom SQL-Nutzerinterface wird in dieser Studie nicht nur zur Benutzerfreundlichkeit eingesetzt wird, sondern auch zwecks automatischer Optimierung – ergo zum Teil aus den gleichen Gründe wie bei SQL2Stan. Das Entwicklerteam baute für sein Projekt eine spezielle spaltenorientierte Engine namens COHANA, die die SQL-Abfragen für Kohortenanalyse optimiert. Das Erweitern vom DBMS um diese Engine und der Einsatz von SQL mit domänenspezifischen Zusatzoperatoren brachte eine spürbare Beschleunigung in der Ausführung von kohortenanalytischen Abfragen mit sich, verglichen mit dem Einsatz von nichterweiterten DBMS und SQL.

Man sieht also, dass auf SQL als Träger für analytische Funktionalitäten oft eingesetzt wird, nicht nur in SQL2Stan.

E.2.26. Elasticsearch

Sogar NoSQL-Datenbanken (Not-only-SQL, nichtrelationale Datenbanken) mögen aus Benutzerfreundlichkeit versuchen, etwas relational und SQL-unterstützend zu werden. Als ein Beispiel hierfür dient ElasticSearch [DG13], der Kern des Elastic-Softwarestacks und ein sehr verbreitetes System für Suchmaschinen-Server. Elastic-Search bietet ab der Version 6.3 vom Juni 2018 auch SQL-Unterstützung als ein experimentelles, eingeschränktes Feature an. Das kann man als Argument dafür verwerten, dass man SQL und relationale Sichtweise auf strukturierbare Daten – beide grundlegend in SQL2Stan – nicht verwerfen soll, auch wenn man mit unstrukturierten Daten arbeitet.

NoSQL-Datenbanken dienen zur Speicherung von nichtnormalisierten Daten (wie z.B. MongoDB als ein dokument-orientiertes NoSQL-DBMS). Klassische SQL-Datenbanken konzentrieren sich hingegen nahezu vollständig auf Relationen zwischen den Datenentitäten und, damit einhergehend, auf Normalisierungen und Constraints, um diese Relationen aufrechtzuerhalten. Es sind unterschiedliche Sichtweisen auf die zu speichernden Daten. Warum macht ein Elasticsearch als ein NoSQL-System diesen Schritt in die entgegengesetzte Richtung? Kurz: es ist ein Kompromiss, um das System für mehr Anwender zugänglicher zu machen.

Als SQL-Anwender unter den Elasticsearch-Nutzern sollen sich laut [McD18] zwei Zielgruppen herauskristallisieren. Die erste Zielgruppe seien Neulinge im Elastic-Softwarestack, die von der domainspezifischen Sprache von Elasticsearch überfordert fühlen bzw. die vollständige Syntax nicht erlernen möchten: „wenn Anwender beispielsweise eine vorhandene SQL-basierte Anwendung aus Performance- und Skalierbarkeitsgründen konvertieren, benötigen sie möglicherweise einfach nur die entsprechende Abfrage, ohne dazu die komplette Syntax erlernen zu müssen“ [McD18]. Im Bezug auf den Erfolgsgrad beim Erlernen von ElasticSearch soll dies von Vorteil sein. Das Lernen der Elasticsearch-DSL soll durch den Bezug zum vorhandenen SQL-Wissen erleichtert werden. Die zweite Zielgruppe seien Personen, die die domainspezifische Sprache von Elasticsearch gar nicht erlernen möchten oder müssen. Das seien beispielsweise Datenwissenschaftler (ganz nach dem Motto „warum soll ich eine DSL lernen, wenn ich bloß die Daten extrahieren und diese extern weiterverarbeiten will?“) oder technisch im geringeren Maße sachkundige Business-Intelligence-Nutzer, die mit SQL ihr täglich Brot verdienen.

SQL ist in Elasticsearch 6.3 nativ integriert (ohne externe Hardware, Prozesse, Bibliotheken oder Laufzeitumgebungen), effizient in der Ausführung und im Sinne der Implementierung keine Abstraktion von der Elasticsearch-DSL, sondern ihre deklarative Erweiterung [ela18].

E.2.27. Übersicht: Machine Learning für Datenmanagement-Aufgaben

Bisher ging es viel darum, wie man ML skalierbar macht – das Thema war, um es allgemein auszudrücken, Datenmanagement für das maschinelle Lernen. Doch auch in umgekehrter Richtung – Machine Learning für Datenmanagement – wurde in den letzten Jahren viel geforscht. ML kann viele Datenmanagement-Aufgaben unterstützen: Bayes'sche Analyse für Datensäuberung [De+16] und -Reparatur [Rek+17], Informationsintegration [DBS13] [PBP17] oder für lernende Datenbanken [Par+17]. Eine große Übersicht dazu lässt sich in [BBM18] finden.

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Abschlussarbeit bzw. den entsprechend gekennzeichneten Anteil der Abschlussarbeit selbstständig verfasst, erstmalig eingereicht und keine anderen als die angegebenen Quellen und Hilfsmittel einschließlich der angegebenen oder beschriebenen Software benutzt habe. Die den benutzten Werken bzw. Quellen wörtlich oder sinngemäß entnommenen Stellen habe ich als solche kenntlich gemacht.

Ort, Datum

Unterschrift Dmitry Trofimov