Heinrich Heine
Universität
Düsseldorf

INSTITUT FÜR INFORMATIK
Softwaretechnik und Programmiersprachen
Universitätsstr. 1 D–40225 Düsseldorf

Eine Machine-Learning-Bibliothek für Prolog

Bachelorarbeit

im Studiengang Informatik
zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

vorgelegt von
Dean Samuel Schmitz

Beginn der Arbeit: 24. August 2022
Abgabe der Arbeit: 24. November 2022

Erstgutachter: Univ.-Prof. Dr. M. Leuschel
Zweitgutachter: Dr. C. Bolz-Tereick

Selbstständigkeitserklärung

Hiermit versichere ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, den 24. November 2022

Dean Samuel Schmitz

Zusammenfassung

Prolog ist eine auf Logik basierende deklarative Programmiersprache, die im Bereich AI gut für Expertensysteme [Mer12] und Natural-Language-Proccessing [LF11] geeignet ist. Der AI Bereich hat sich mit der Zeit stark in Richtung Daten getriebenes Lernen und damit in Richtung Machine-Learning entwickelt. Deshalb geht es in dieser Arbeit darum, eine Machine-Learning-Bibliothek für Prolog bereitzustellen.

Zu diesem Thema gibt es nur wenige Arbeiten und diejenigen, die sich damit befassen, behandeln nur einen sehr kleinen Teil aus Machine-Learning. Nur eine erst kürzlich erschienene Arbeit [CCC22] befasst sich mit dem Konzept, eine Machine-Learning-Bibliothek direkt in Prolog umzusetzen.

Daher gibt es momentan noch keine umfangreiche Machine-Learning-Bibliothek für Prolog, wie es sie mit MLpack in C++ oder scikit-learn in Python gibt.

Da die MLpack Bibliothek schon eine große Sammlung an effizienten Machine-Learning Methoden bietet, war nun der Gedanke, anstatt eine komplett neue ML Bibliothek zu entwickeln, einfach MLpack für Prolog nutzbar zu machen.

Dafür wurde die C/C++ Schnittstelle von SICStus Prolog genutzt, um die jeweiligen MLpack Methoden in eine oder mehrere Funktionen aufzuteilen, und diese mit der Schnittstelle an nutzbare Prolog Prädikate zu binden.

Es wurden 30 der MLpack Methoden mit der Prolog-MLpack-Bibliothek für Prolog verfügbar gemacht. Wie z. B. Random Forest, linear Regression, KMeans und Kernel PCA.

Dadurch ist die Bibliothek vergleichbar mit den anderen Sprach-Anbindungen von MLpack. Sie ist gut für simple Machine-Learning Probleme und bietet einen einfachen Zugang in den Machine-Learning Bereich, jedoch kann sie nicht alle Vorteile der C++ Implementierung von MLpack an Prolog übertragen.

Mit dieser Bibliothek erweitert sich der Nutzungsbereich von Prolog, was Prolog möglicherweise auch in weiteren Bereichen als eine breiter nutzbare Sprache populärer machen könnte.

Dieser Prozess, externer Projekte mit der C/C++ Schnittstelle an Prolog anzubinden, lässt sich sehr wahrscheinlich auch in anderen Projekten anwenden. Jedoch sollte man dabei beachten, dass die Schnittstelle in seine Möglichkeiten recht limitiert ist und eher für simple Projekte geeignet ist.

Danksagung

Ich möchte mich bei David Geleßus für seine Unterstützung während des gesamten Prozesses der Bachelorarbeit bedanken.

Und bei meinen Eltern, die mir beim Fehlersuchen geholfen haben.

Inhaltsverzeichnis

1 Einleitung und Motivation

Das Thema der Arbeit ist die Bereitstellung der **Machine-Learning-Bibliothek MLpack** mit **SICStus Prolog**.

Prolog wurde in den 1970ern entwickelt, mit dem Gedanken, Wissen mit Logik darstellen zu können und mit Deduktion Konsequenzen aus diesem Wissen schließen zu können [Kow88].

Damit ist Prolog eine **deklarative Programmiersprache**, die auf logischen Aussagen, in Form von Fakten und Regeln, basiert. Diese werden dann dafür genutzt, um zu beschreiben was das Problem ist, das man Lösen will. Anstatt wie es bei üblichen imperativen Sprachen der Fall ist, zu beschreiben wie das Problem gelöst werden soll.

Diese Fakten und Regeln werden in der Wissensdatenbank abgespeichert. Um dann eine Lösung zu erhalten, stellt man an Prolog eine Anfrage, mit vorgegebenen oder variablen Werten als Argumenten. Prolog nutzt dann Deduktion, um diese Anfrage gegen die Wissensdatenbank zu lösen. Man erhält dann zurück, ob die vorgegebenen Werte zu einer Lösung führen. Wenn Variablen mit angegeben wurden, kann man alle möglichen Werte der Variable durchgehen, die zu einer Lösung führen.

Prolog eigenet sich dadurch am Besten in Bereichen, die durch seinen deklarativen Ansatz einen besonderen Vorteil erlangen. Daher hat Prolog eher einen spezifischen Nutzungsrahmen. Zum Beispiel in **Parsern**, zum **Optimieren** oder im Bereich **KI** für **Expertensysteme** oder **Natural-Language-Processing**

Jedoch hat sich das Feld der künstlichen Intelligenz über die Jahre mehr in die Richtung des datenbasierten Lernens entwickelt und damit ist Machine Learning ein starker Fokus in der KI.

Aber Machine Learning ist keine Stärke von Prolog und profitiert nicht von seinem deklarativen Grundsatz. Besonders der Umgang mit Matrizen und Vektoren an Daten und das umwandeln und anwenden mathematischer Funktionen auf diese Daten sind fundamentale Themen in Machine Learning, aber nicht gut unterstützt in Prolog.

Das würde Prolog eigentlich zu keinem guten Kandidaten machen, da es jedoch in anderen KI Bereichen wie oben genannt deutliche Vorteile mitbringt, könnte es sich trotzdem lohnen, Prolog um Machine-Learning Methoden zu erweitern.

Wenn man im Bereich Machine Learning nach einfach zugänglichen Bibliotheken für Prolog sucht, findet man nur eine sehr kleine und nicht besonders umfangreiche Auswahl, was den Einstieg in das Thema mit Prolog schwierig und aufwändig macht.

Nun gibt es die **ML-Bibliothek MLpack**, die eine umfangreiche Auswahl an ML-Methoden und Datensätzen bietet.

Sie ist in C++ geschrieben und bietet Anbindung für die Sprachen **Python**, **Julia** und

Go. Plus eine **CLI** Version, wenn man **MLpack** ohne viel drumherum ausprobieren will.

MLpack ist **open source** und wird seit 2007 stetig weiterentwickelt. Der neuste Release war 3.4.2 in 2020. Daher sind die Methoden auf einem relative aktuellen und gut erhaltenen Stand [mlp].

Die Hauptziele, die **MLpack** dabei versucht einzuhalten, sind:

- **Effizienz**. Besonders, wenn es um den Umgang mit großen Datensätzen geht.
- **Intuitives Nutzen**. Weshalb viel Wert auf eine umfangreiche und klare Dokumentierung der C++ Implementierung und der anderen Sprach-Anbindungen gelegt wurde.
- **Flexibilität**. Um einfach Ergebnisse aus dem Konzeptionieren in andere nutzbare Produkt Projekte zu Implementieren.

Daher die Idee, die **C/C++ Schnittstelle** von **SICStus Prolog** zu nutzen, um die **MLpack** Bibliothek an **Prolog** zu binden.

2 Verwandte Arbeiten

Im Moment ist die beliebteste Sprache für Machine Learning **Python**. **Sklearn**, **NumPy** und **ggplot** stellen den Umgang mit Daten, viele Mathematik und ML Methoden und das Plotten der Daten und Ergebnisse zur Verfügung. Das zusammen mit **Jupyter-Notebook** ermöglichen einen extrem einfachen Einstieg in Machine Learning.

Das argumentieren die beiden Arbeiten [RPN20] und [MG17] ebenfalls. Die Arbeit [RPN20] gibt zusätzlich dazu den einen Artikel, mit einer Umfrage über die genutzte Software in den Bereichen „Analytics, Data Science und Machine Learning“ von KDnuggets an [Pia], in der Python das Ranking mit über 60% anführt.

Wohingegen Prolog, besonders im Bereich der ML-Bibliotheken, nichts vergleichbares hat.

Es gibt vereinzelte **Github Projekte** [Riga], [Rigb], [Ang], [Nea], die eine kleine Menge an Machine-Learning Methoden direkt in Prolog implementieren.

Aber sie sind nicht für den generellen Gebrauch oder zum Lernen nutzbar, oder sind sehr alt wie z. B.[Hop].

Eine neue Arbeit [CCC22] ist während des Schreibens dieser Arbeit erschienen, die sich auch damit befasst eine Machine-Learning-Bibliothek für Prolog zu erstellen. Mit dem Unterschiedlichen Ansatz die Bibliothek komplett in Prolog selbst zu entwickeln. In ihr werden Konzepte für den **API-Ansatz** der Bibliothek und ein Prototyp für **neuronale Netze** vorgestellt.

Projekte oder Arbeiten, die versuchen eine umfangreiche **Machine-Learning-Bibliothek** in Prolog zu schreiben oder eine existierende Machine-Learning-Bibliothek, einer anderen Sprache wie mlpack, in Prolog zu implementieren, gibt es noch nicht.

Nur noch weniger verwand mit diesem Thema, gibt es Arbeiten wie [OFDR17], die Prolog um algebraische Funktionalität erweitert, sodass es besser möglich ist, Machine Learning Probleme in Prolog selber definieren zu können.

Oder die Arbeit [LLG12], die Prolog mithilfe von Machine-Learning Methoden verbessern wollen.

3 Implementierung

In diesem Kapitel werden **MLpack** und **SICStus** vorgestellt und im Detail erklärt, wie die **MLpack Methoden** mit **SICStus** als **Prolog Bibliothek** implementiert werden.

3.1 Warum mlpack

Wie schon in der Einleitung erwähnt, ist **MLpack** [CEL+18] eine Sammlung an Machine Learning Methoden, mit den Zielen effizient, intuitiv und flexibel zu sein.

Und das sind auch die Stärken, die MLpack trotz der starken Beliebtheit von Python immer noch relevant machen [mlp]. Den nach den eigenen Angaben ist MLpack deutlich schneller mit seinen Methoden, im Vergleich zu den anderen Machine Learning Bibliotheken [CE]. Diese Effizienz erreicht es durch die **C++ Armadillo Bibliothek** [SC16] [SC18], die effiziente Lineare-Algebra Funktionen und Datentypen mitbringt und gut mit großen Datensätzen skalieren.

Dadurch, dass man für das Nutzen von MLpack nur ein C++ Compiler braucht, ist es gut für dieses Projekt geeignet. Es kann damit einfach mit der C/C++ Schnittstelle von SICStus Prolog in nutzbare Prolog Prädikate umgewandelt werden.

Zudem ist die ausführliche Dokumentation mit detaillierten Informationen über die verschiedenen Methoden und deren Inputs und Outputs sehr hilfreich.

Daher wäre es ideal, all diese Vorteile, die MLpack mitbringt, auf die Prolog Bibliothek in diesem Projekt zu übertragen.

3.2 Vorstellung der SICStus C/C++ Schnittstelle

SICStus [CM10] ist ein aktuelles Prolog-Entwicklungs-System, dass einige vorgefertigte Module beinhaltet, wie z.B. die C/C++ Schnittstelle und dabei auf **Schnelligkeit** und

Effizienz setzt.

Das Ziel ist, die Methoden von MLpack als Prolog Prädikate nutzbar zu machen. Da SICStus eine C/C++ Schnittstelle anbietet, die simpel und leicht einsetzbar ist und diese auch brauchbar dokumentiert hat, bietet sich SICStus für dieses Projekt gut an. Zudem hat es für **Eclipse** eine **IDE-Integrierung** namens **SPIDER**, die beim Implementieren und Schnellen austesten hilfreich sein wird.

Es gibt auch andere Projekte, die eine Schnittstelle für C/C++ bieten, und sich als mögliche Alternativen anbieten würden:

1. **Dynamic calling C from Prolog** [Wiea], beinhaltet eine recht ausführliche Dokumentation.
2. **SWI-Prolog C++ interface** [Wieb], hat nur Beispiele als Dokumentation angegeben und erweitert die C Schnittstelle von SWI-Prolog für C++.
3. **GNU PROLOG C Interface** [Dia], ist vom groben Umfang und Umsetzung der Schnittstelle sehr vergleichbar mit SICStus.

(1) und (2) wirken, von dem groben Überblick den ich mir von ihnen gemacht habe, im Vergleich mit SICStus, etwas komplizierter und weniger intuitiv in der Umsetzung der Schnittstelle.

Erst im Detail wird sich zeigen, ob eine der Alternativen nützliche Vorteile bringt oder nicht.

SICStus ist, wegen der oberhalb genannten Punkten, meiner Meinung nach momentan die beste Wahl.

3.3 Einfaches Beispiel der C/C++ Schnittstelle

Hier ein Beispiel zur Demonstration der Schnittstelle im kleinen Rahmen, um später im detaillierteren Beispiel einen besseren Überblick zu behalten.

Man benötigt mindestens 2 Dateien:

- Eine Prolog Datei: **demo.pl**
- Eine C/C++ Datei: **demo.cpp**

Die Dateien müssen den gleichen Namen haben, da sonst der Compiler einen Fehler wirft.

Quellcode 1: Wichtigster Teil der c++ Datei

```
1: #include <sicstus/sicstus.h>
2: /* ex_glue.h is generated by splfr from the foreign/[2,3] facts.
3:    Always include the glue header in your foreign resource code.
4: */
5: #include "demo_glue.h"
```

Diese zwei Zeilen sind auf der C++ Seite für die Verbindung am wichtigsten.

Die erste Zeile holt alle in SICStus enthaltenen C Funktionen. Und die zweite Zeile Included die **glue Datei**, die beim gemeinsamen Kompilieren der C++ und PL Datei erstellt wird. Diese muss denselben Namen **demo** haben.

Quellcode 2: Prolog Implementation von **funk/3**

```
1: :- load_foreign_resource(demo).
2: foreign_resource(demo, [funk]).
```

Die erste Zeile definiert, wie die fremde Ressource, die geladen werden soll, heißt. In diesem Beispiel daher **demo**. Die zweite Zeile definiert, welche Funktionen von der Ressource geladen werden sollen. In diesem Fall wollen wir die Funktion **funk()** von **demo.cpp** verbinden.

Die Funktion **funk()** ist so definiert:

Quellcode 3: Definition Funktion funk

```
1: SP_integer funk(SP_integer zahl, SP_integer *ergebnis1)
2: {
3:         *ergebnis1 = zahl + 1;
4:         return (zahl * 2);
5: }
```

Nachdem man nun in der **Prolog-Datei** definiert hat, wie die fremde Datei heißt und welche die fremden Funktionen sind, die man laden möchte, muss man nun definieren, wie die Verbindung der Funktionen und deren Parameter genau ablaufen soll:

Quellcode 4: Prolog Implementation von **funk/3**

```
1: foreign(funk, funk(+integer, -integer, [-integer]))
```

Jede Funktion, die man laden will, muss mit dem **foreign/2,3** Prädikat definiert werden. Die Funktion **funk()** soll im ersten Argument eine Zahl bekommen, im zweiten Argument ein erstes Ergebnis in der Variable **ergebnis1** speichern und zurückgeben. Als Rückgabewert der Funktion soll ein zweites Ergebnis als Integer zurückgegeben werden.

Das definiert man nun mit **foreign** in Prolog so:

1. Arg: funk (Name der fremden Funktion. Muss mit Kleinbuchstaben anfangen)
2. Arg: c (Ist bei foreign/2 standardmäßig c und kann daher weggelassen werden)
3. Arg: funk(+integer, -integer, [-integer]) (Hier wird dann das Prädikat und dessen Parameter definiert, welches man dann in Prolog nutzen kann.)

Man kann dem Prädikat einen anderen Namen geben, als den der C++ Funktion.

Hierbei bedeuten

+integer -> Input Argument mit dem Datentyp Integer.

-integer -> Variablen Argument das auf der C++ Seite einen Integer Wert annimmt.

[-integer] -> Output Argument mit dem Datentyp Integer.

Bei den Argumenten ist die Reihenfolge wichtig, denn das erste Argument des Prädikates ist, dann auch das erste Argument der C++ Funktion. Die Ausnahme ist das **[-argument]**, welches an jeder Stelle stehen kann, da man in C++ nur ein direktes **return** Argument definieren kann.

Um das **funk/3** Prädikat dann zu benutzen, muss man die **demo.cpp** und **demo.pl** Dateien einmal kompilieren.

Dazu nutzt man das **splfr** Tool von SICStus mit dem Kommand:

```
1: $ absolute\path\to\splfr demo.pl demo.cpp
```

(Falls das nicht im Ordner der Demo Dateien ausgeführt wird, müssen die Pfade zu den Demo Dateien noch mit angehangen werden.)

Während des Kompilierens wird die **demoglue.h** Datei erstellt und mit der **demo.so** Datei ersetzt, die nun als Verbindung zwischen den Beiden fungiert.

Jetzt kann man das **funk/3** Prädikat innerhalb von SICStus nutzen.

Der Aufruf von **funk/3** würden dann so aussehen:

Quellcode 5: Prolog Aufruf von **funk/3**

```
1: %% Die demo.pl im SICStus toplevel laden
2: ?-[demo].
3: ...
4: %% Nun hat man zugriff auf alle in demo.pl definierten Praedikate
5: %% In diesem Fall auf funk/3
6: ?-funk(7,Ergebnis1,Ergebnis2).
7: Ergebnis1 = 8,
8: Ergebnis2 = 14 ?
9: yes
```

10: ?-

In der Dokumentation zur C/C++ Schnittstelle von SICStus gibt es weitere Beispiele.[1] [2]

3.4 Detaillierte Implementierung der ML-Methoden

Nun werde ich die Implementation der beiden Methoden **Mean_Shift** und **Linear_SVM** im Detail vorstellen.

Mean_Shift und **Linear_SVM** habe ich als Beispiele herausgesucht, da sie jeweils die beiden verschiedenen Strukturen für das Implementieren der Methoden demonstrieren und dabei relativ überschaubar bleiben.

Mean_Shift zeigt, wie die Methode kompakt in ein einzelnes Prädikat verpackt implementiert wird.

Linear_SVM zeigt, wie die Methode in viele einzelne Prädikate aufgeteilt wird.

3.4.1 Mean_Shift Implementierung

Jede Methode hat in ihrem Ordner 4 Dateien.

Im Fall von **Mean_Shift** sind das:

- mean_shift.pl
- mean_shift.cpp
- mean_shift_test.pl
- Makefile

Wie die Test und die Makefile Dateien genutzt werden, wird im Kapitel 4 Ergebnisse erklärt.

In diesem Kapitel werden die Dateien **mean_shift.cpp** in Quellcodes 6 bis 10 und **mean_shift.pl** in Quellcodes 11 und 12 vorgestellt.

C++ Datei Fangen wir zu erst mit **mean_shift.cpp** an.

[1] //sicstus.sics.se/sicstus/docs/latest4/html/sicstus.html/Foreign-Code-Examples.html

[2] https://sicstus.sics.se/sicstus/docs/latest4/html/sicstus.html/Mixing-C-and-Prolog-Examples.html

Quellcode 6: mean_shift.cpp Includes

```
1: #include <sicstus/sicstus.h>
2: #include "mean_shift_glue.h"
3:
4: #include <mlpack/methods/mean_shift/mean_shift.hpp>
5: #include <mlpack/core.hpp>
6:
7: // including helper functions for converting between arma structures
8: // and arrays
9: #include "../../helper_files/helper.hpp"
```

Die Zeilen 1 und 2 aus Quellcode 6 sollten durch Quellcode 1 bekannt sein, nur mit dem neuen Namen **mean_shift** statt **demo** für die glue Datei.

Neu hinzugekommen sind die Zeilen 4 und 5, die zum einen die **mean_shift** Implementation von MLpack laden und zum anderen die grundsätzlichen Funktionen von MLpack laden. Und mit Zeile 7 werden meine eigenen helper Funktionen geladen.

In Quellcode 7 ist der Kopf der **meanShift()** Funktion.

Quellcode 7: mean_shift.cpp Kopf der Funktion meanShift

```
1: void meanShift(
2:     double radius,
3:     SP_integer maxIterations,
4:     float *dataMatArr, SP_integer dataMatSize,
5:         SP_integer dataMatRowNum,
6:     float **assignmentsArr, SP_integer *assignmentsArrSize,
7:     float **centroidsMatArr, SP_integer *centroidsMatColNum,
8:         SP_integer *centroidsMatRowNum,
9:     SP_integer forceConvergence,
10:    SP_integer useSeeds)
11: {
12:    ...
13: }
```

Beim Namen der Funktion ist es wichtig, dass der Anfangsbuchstabe kleingeschrieben ist, da er sonst in Prolog als Variabel gewertet wird. Die Zeilen 2 bis 9 sind die Input und Output Parameter. Besonders ist hier der Datentyp **SP_integer** , der von SICStus kommt, um Integer zwischen Prolog und C konvertieren zu können. **Float *dataMatArr** ist dabei Input Parameter der zuerst als ein langes float Array bei C++ ankommt und später zu einer Matrix umgewandelt wird. Das Ouput Gegenstück dazu ist **float **centroidsMatArr**, wo man von Prolog einen Pointer für ein **float Array** bekommt, dem dann in C++ ein echtes **float Array** zugewiesen wird und danach als Rückgabewert zurück an Prolog übergeben wird.

Dann schauen wir uns nun den Körper von **meanShift()** an. Der Großteil der Funktionen in der Bibliothek folgen demselben Schema und können in grob 3 Teile aufgeteilt werden.

Quellcode 8: mean_shift.cpp 1.Teil: Konvertieren der Matrizen

```
1:  void meanShift(...)
2:  {
3:      // convert the Prolog array to arma::mat
4:      mat data = convertArrayToMat(dataMatArr, dataMatSize,
5:          dataMatRowNum);
6:      // create the ReturnVector
7:      arma::Row<size_t> assignmentsReturnVector;
8:      // create the ReturnMat
9:      mat centroidsReturnMat;
10:
11:         ...
12: }
```

Hier im Quellcode 8 sieht man den ersten Teil. Indem die eingegebene **Data-Matrix** zu einer **arma::mat** Matrix konvertiert wird und die beiden Rückgabewerte **assignmentsReturnVector** und **centroidsReturnMat** als arma Datentypen initialisiert werden, damit sie dann im 2. Teil die erstellten Daten abspeichern können.

Quellcode 9: mean_shift.cpp 2.Teil Methoden Funktionsaufruf

```
1:  void meanShift(...)
2:  {
3:          ...
4:
5:      try
6:      {
7:          MeanShift<>(radius, maxIterations)
8:          .Cluster(data, assignmentsReturnVector, centroidsReturnMat,
9:              (forceConvergence == 1), (useSeeds == 1));
10:     }
11:     catch(const std::exception& e)
12:     {
13:         raisePrologSystemExeption(e.what());
14:         return;
15:     }
16:
17:         ...
18: }
```

In Quellcode 9, dem 2. Teil, wird das MeanShift Objekt mit den Parametern **radius** und **maxIterations** initialisiert. Von dem dann die Funktion **Cluster()** aufgerufen wird, was die übergebenen Datenpunkte in Cluster aufteilt, und jedem Cluster eine Koordinate im Zentrum zuweist. Die Outputs der Funktion werden dann in den Rückgabewerten aus dem 1. Teil gespeichert.

Damit, falls dabei etwas schiefläuft, die Fehlermeldungen in SICStus besser angezeigt werden, ist der Aufruf mit einem **try-catch** Block umgeben. Die C++ Terminal Ausgaben

werden in SICStus größtenteils mitten im Text von anderen Ausgaben angezeigt und nicht an der Stelle, wo sie tatsächlich aufgerufen wurden. Zudem gibt es einem die Möglichkeit, die Fehlermeldungen selber für Prolog anzupassen, falls die originalen Fehlermeldungen nicht klar genug sind.

Quellcode 10: mean_shift.cpp 3.Teil Rückgabe Werte zurückgeben

```
1:  void meanShift(...)
2:  {
3:          ...
4:
5:      // return the Matrix
6:      returnMatrixInformation(centroidsReturnMat,
7:          centroidsMatArr, centroidsMatColNum, centroidsMatRowNum);
8:      // return the Vector
9:      returnVectorInformation(assignmentsReturnVector,
10:         assignmentsArr, assignmentsArrSize);
11: }
```

Und zuletzt der Quellcode 10 mit dem 3. Teil, wo die nun befüllten Rückgabewerte mithilfe meiner Helfer-Funktionen an die Rückgabe-Parameter verteilt und in die **float *arr** Form konvertiert werden.

Prolog Datei Betrachten wir nun die Prolog Implementierung.

Quellcode 11: mean_shift.pl Kopfzeilen

```
1:  :- module(mean_shift, [meanShift/9]).
2:
3:  %% requirements of library(struct)
4:  :- load_files(library(str_decl),
5:                  [when(compile_time), if(changed)]).
6:
7:  %% needed for using the array type
8:  :- use_module(library(structs)).
9:  :- use_module('../../helper_files/helper.pl').
10:
11: %% type definitions for the float array
12: :- foreign_type
13:         float32          = float_32,
14:         float_array      = array(float32).
15:
16: ...
17:
18: %% Defines the functions that get connected from meanShift.cpp
19: foreign_resource(mean_shift, [meanShift]).
20:
21: :- load_foreign_resource(mean_shift).
```

Die letzten beiden Zeilen dürften aus Quellcode 2 bekannt sein.

Neu dazu gekommen ist in Zeile 1, dass für die Datei ein Modul definiert wird, um kontrollieren zu können, welche Prädikate der Datei exportiert werden und welche nur intern genutzt werden können.

In diesem Fall wird das Prädikat **meanShift/9** exportiert, was nicht direkt das mit foreign definierte C++ Prädikat ist.

Um Prolog Listen zwischen C++ und Prolog konvertieren zu können, wird das Modul **struct** von SICStus benötigt. Das wird mit Zeile 8 geladen. Dazu benötigt **struct** auch die Zeilen 4 und 5, um richtig funktionieren zu können.

In den Zeilen 12 bis 14 werden die beiden Datentypen **float32** und **float_array** für diese Datei definiert. In der Schnittstelle wird **float32** in C++ zu **double** konvertiert und **float_array** zu einem **float *arr**.

Zusätzlich wird mein eigenes helper Modul in Zeile 10 geladen, das vor allem Prädikate enthält, die normale Prolog Listen zu **float_arrays** und umgekehrt konvertieren kann.

Quellcode 12: mean_shift.pl meanShift Prädikat Definition

```
1:  ...
2:
3:  meanShift(Radius, MaxIterations, DataList, DataRows, AssignmentsList,
4:      CentroidsList, ZCols, ForceConvergence, UseSeeds) :-
5:          MaxIterations >= 0,
6:          convert_list_to_float_array(DataList, DataRows,
7:              array(Xsize, Xrownum, X)),
8:          meanShiftI(Radius, MaxIterations, X, Xsize, Xrownum, Y, Ysize,
9:              Z, ZCols, ZRows, ForceConvergence, UseSeeds),
10:         convert_float_array_to_list(Y, Ysize, AssignmentsList),
11:         convert_float_array_to_2d_list(Z, ZCols, ZRows, CentroidsList).
12:
13: foreign(meanShift, c, meanShiftI(
14:     +float32, +integer,
15:     +pointer(float_array), +integer, +integer,
16:     -pointer(float_array), -integer,
17:     -pointer(float_array), -integer, -integer,
18:     +integer, +integer)).
19:
20: ...
```

Im Hauptteil der Datei Quellcode 12 hat man einmal das **meanShift/9** Prädikat, dass dann mit dem meanShift-Modul exportiert wird. Und die externe Prädikat-Definition, welche die **meanShift()** C++ Funktion an das Prädikat **meanShiftI/12** bindet.

Der grobe Aufbau der foreign Anbindung ist genauso wie beim Quellcode 4. Nur mit den extra Parametern **+float32** und **+pointer(float_array)**.

Die andere Prädikat-Definition **meanShift/9** ist als eine Zwischenschicht gedacht, zwischen dem Nutzer und der **foreign** Definition, die sich um das Konvertieren der Prolog Listen zu **float_arrays** und umgekehrt kümmern soll. Dass sieht man zum Beispiel in Zeile 6, wo mithilfe des **convert_list_to_float_array/3** aus dem helper-Modul, die Data-Liste zu dem **float_array** X konvertiert wird, mit den extra Informationen **Xsize** und **Xrownum**. Diese sind deshalb wichtig, da in C++ sobald die Daten konvertiert wurden, es keine Möglichkeit mehr gibt, die Länge des übergebenen Arrays herauszufinden. Und **Xrownum** ist dafür wichtig, um zu wissen, welche Form die **Daten-Matrix** hat.

Bei Vektoren braucht man nur das **float_array** und seine Länge, wie man in Zeile 10 sehen kann. Hier wird das von der Funktion zurück gegebene **float_array** Y in eine Prolog-Liste konvertiert, ebenfalls mit einer der Prädikate aus dem helper-Modul.

Außerdem soll **meanShift/9**, die Parameter auch auf zusätzliche Anforderungen überprüfen, wie man in Zeile 5 sehen kann, wo **MaxIterations** nur positive Zahlen annehmen soll.

Damit soll der Nutzer des Prädikats einfach normale Prolog-Listen als Parameter eingeben können, ohne sich Gedanken über das Konvertieren machen zu müssen.

Dem entsprechend sieht der Ablauf eines Aufrufs für **Mean_Shift** folgendermaßen aus:

Nutzer ⇒ meanShift/9 ⇒ meanShiftI/12 ⇒ meanShift(..) ⇒ MLpack::meanshift Funktionen ⇒ (Rückgabe Werte) meanShift(..) ⇒ meanShiftI/12 ⇒ meanShift/9 ⇒ Nutzer

3.4.2 Linear_SVM Implementierung

Mit der **LinearSVM** Implementierung als Beispiel möchte ich eine andere Art, die MLpack Methoden an Prolog zu binden, demonstrieren. Zwar ist die Interaktion mit der C++ Schnittstelle fast identisch, jedoch wird die Methode etwas anders für Prolog nutzbar gemacht.

Quellcode 13: linear_SVM.cpp Includes

```
1: #include <sicstus/sicstus.h>
2: #include "linear_SVM_glue.h"
3: #include <mlpack/methods/linear_svm/linear_svm.hpp>
4: #include <mlpack/core.hpp>
5:
6: // including helper functions for converting between arma structures
7: // and arrays
8: #include "../../helper_files/helper.hpp"
```

Wie man im Vergleich mit Quellcode 6 sehen kann, gibt es keinen großen Unterschied bei den **Includes**.

Aufteilung in mehrere Funktionen Bei der Implementation der Funktionen sieht man, dass nicht nur eine Funktion definiert ist, die sich um alle Parameter und die MLpack **LinearSVM** Aufrufe kümmert, sondern mehrere Funktionen, die jeweils eine Aufgabe der **LinearSVM** Methode übernehmen.

Quellcode 14: linear_SVM.cpp Erste drei Funktionen

```
1:  void initModelWithTrain(
2:      float *dataMatArr, SP_integer dataMatSize,
3:          SP_integer dataMatRowNum,
4:      float *labelsArr, SP_integer labelsArrSize,
5:      SP_integer numClasses, double lambda, double delta,
6:      SP_integer fitIntercept,
7:      char const *optimizer)
8:  {
9:      // convert the Prolog array to arma::mat
10:     mat data = convertArrayToMat(dataMatArr, dataMatSize,
11:         dataMatRowNum);
12:     // convert the Prolog array to arma::rowvec
13:     arma::Row<size_t> labels = convertArrayToVec(labelsArr,
14:         labelsArrSize);
15:
16:     linearSVM = LinearSVM<>(numClasses, lambda, delta,
17:         (fitIntercept == 1));
18:
19:     if (strcmp(optimizer, "lbfgs") == 0)
20:     {
21:         linearSVM.Train<ens::L_BFGS>(data, labels, numClasses);
22:     }
23:     else if (strcmp(optimizer, "psgd") == 0)
24:     {
25:         linearSVM.Train<ens::ParallelSGD<>>(
26:             data, labels, numClasses,
27:             ens::ParallelSGD(100, 4));
28:     }
29:     else
30:     {
31:         raisePrologDomainExeption(
32:             optimizer, 8, "The given Optimizer is unkown!",
33:             "initModelWithTrain");
34:         return;
35:     }
36: }
37:
38: void initModelNoTrain(
39:     SP_integer numClasses, double lambda, double delta,
40:     SP_integer fitIntercept)
41: {
42:     linearSVM = LinearSVM<>(numClasses, lambda, delta, true);
43: }
```

```
44:
45: void classify(
46:     float *dataMatArr, SP_integer dataMatSize, SP_integer
47:         dataMatRowNum,
48:     float **labelsArr, SP_integer *labelsArrSize,
49:     float **scoresMatArr, SP_integer *scoresMatColNum,
50:         SP_integer *scoresMatRowNum)
51: {
52:     // convert the Prolog array to arma::mat
53:     mat data = convertArrayToMat(dataMatArr, dataMatSize,
54:         dataMatRowNum);
55:
56:     // get the ReturnVector
57:     arma::Row<size_t> labelsReturnVector;
58:     // get the ReturnMat
59:     mat scoresReturnMat;
60:
61:     try
62:     {
63:         linearSVM.Classify(
64:             data, labelsReturnVector, scoresReturnMat);
65:     }
66:     catch(const std::exception& e)
67:     {
68:         raisePrologSystemExeption(e.what());
69:         return;
70:     }
71:
72:
73:     // return the Vector
74:     returnVectorInformation(labelsReturnVector, labelsArr,
75:         labelsArrSize);
76:     // return the Matrix
77:     returnMatrixInformation(scoresReturnMat, scoresMatArr,
78:         scoresMatColNum, scoresMatRowNum);
79: }
80: ...
81: // more Funktions after that : classifyPoint, computeAccuracy, train
```

Der Aufbau der einzelnen Funktionen ist dabei aber fast identisch zu Quellcodes 8 bis 10.

Sie folgen ebenfalls der Struktur: Eingabe Werte konvertieren, Model-Funktion Aufruf, Rückgabe Werte konvertieren.

Die MLpack Methoden sind als Klassen mit einigen Funktionen definiert, die sich wie im Fall vom LinearSVM nicht immer in nur eine Funktion zusammenfassen lassen, ohne Teil der originalen Funktionalität zu verlieren. Zudem würde die Menge an möglichen Parametern, die Funktion in der Benutzung überladen.

Globales LinearSVM Objekt Auch neu dazu gekommen ist die **LinearSVM** Klasse als globales Objekt. (Zeile 6 Quellcode 15)

Welches dafür da ist, dass alle Funktionen Zugriff auf das initialisierte **LinearSVM-Objekt** haben und dessen Funktionen aufrufen können.

Man kann in Zeile 42 Quellcode 14 sehen, wie das **LinearSVM-Objekt** initialisiert und global gespeichert wird und dann in Zeile 63 die **Classify()** Funktion auf dem globalen Objekt aufgerufen wird.

Quellcode 15: linear_SVM.cpp Globales linearSVM Objekt

```
1:  ...
2:  using namespace std;
3:  using namespace mlpack::svm;
4:
5:  // Global Variable of the LinearSVM object so it can be
6:  // accessed from all functions
7:  LinearSVM<> linearSVM;
8:
9:
10: void initModelWithTrain(
11:     float *dataMatArr, SP_integer dataMatSize, SP_integer dataMatRowNum,
12:     float *labelsArr, SP_integer labelsArrSize,
13: ...
```

Die andere Möglichkeit, das **LinearSVM-Objekt** an die Funktionen zu verteilen, wäre es immer als Parameter bei jeder Funktion mitzugeben.

String und Bool Parameter Wie **Boolean** und **String** Parameter im Projekt genutzt werden, kann man in der Implementation der Funktion **initModelWithTrain()** sehen Quellcode 16.

Quellcode 16: linear_SVM.cpp String und Bool Parameter

```
1:  void initModelWithTrain(
2:      float *dataMatArr, SP_integer dataMatSize, SP_integer dataMatRowNum,
3:      float *labelsArr, SP_integer labelsArrSize,
4:      SP_integer numClasses, double lambda, double delta,
5:      SP_integer fitIntercept,
6:      char const *optimizer)
7:  {
8:      // convert the Prolog array to arma::mat
9:      mat data = convertArrayToMat(dataMatArr, dataMatSize,
10:         dataMatRowNum);
11:     // convert the Prolog array to arma::rowvec
12:     arma::Row<size_t> labels = convertArrayToVec(labelsArr,
13:         labelsArrSize);
14:
```

```
15:     linearSVM = LinearSVM<>(
16:         numClasses, lambda, delta, (fitIntercept == 1));
17:
18:     if (strcmp(optimizer, "lbfgs") == 0)
19:     {
20:         linearSVM
21:         .Train<ens::L_BFGS>(data, labels, numClasses);
22:     }
23:     else if (strcmp(optimizer, "psgd") == 0)
24:     {
25:         linearSVM
26:         .Train<ens::ParallelSGD<>>(data, labels, numClasses,
27:         ens::ParallelSGD(100, 4));
28:     }
29:     else
30:     {
31:         raisePrologDomainExeption(
32:             optimizer, 8, "The given Optimizer is unkown!",
33:             "initModelWithTrain");
34:         return;
35:     }
36: }
```

Da es in Prolog kein direktes Äquivalent für **Booleans** gibt, wurden **Boolean** Parameter mit **Integer** Werten ersetzt, wie man in Zeile 4 und 12 Quellcode 16 sehen kann. Dafür wird der übergebene Wert abgefragt, ob er gleich eins ist und gibt **True** aus und in allen anderen Fällen **False**.

Und **Strings** werden hauptsächlich dafür benutzt, um ein bestimmtes Verhalten oder wie in Zeilen 14 und 22 der zu verwendende Optimierer auszuwählen. Der Input für **Strings** in Prolog sind **Atoms**, was aber auch bedeutet, dass **Strings** mit Großbuchstaben am Anfang und bestimmte Symbole wie **-** wegfallen.

Darauf, wie die Methoden **kompiliert**, **verwendet** und **getestet** werden können, gehe ich später im Kapitel 4 Ergebnisse ein.

3.5 Welche Design Entscheidungen wurden getroffen und warum

Eigene C++ Dateien, welche die MLpack Funktionen aufrufen Mit eine der ersten Entscheidung war es, anstatt die MLpack C++ Dateien der Methoden direkt mit SICStus zu verbinden, eigene C++ Dateien für jede Methode hinzuzufügen. Diese dienen als Brücke zwischen MLpack und SICStus, indem sie die Methoden-Objekte laden und die Funktionen, die für Prolog verfügbar gemacht werden sollen, selbst implementiert.

Dazu übernehmen die eigenen C++ Dateien das Konvertieren der Datentypen, die von und an die Prolog-Schnittstelle geschickt werden und rufen dann nur noch die Funktionen der

MLpack Methoden Objekte auf und geben die erlangten Daten an Prolog zurück.

Der Hauptgrund dafür ist, dass die Methoden in MLpack als Klasse definiert sind, die nicht kompatibel mit der C/C++ Schnittstelle sind, da sie nur allein stehende Funktionen nutzen kann.

Nutzen des „struct“ Moduls Für das Übertragen der **Matrizen** und der **Vektoren** gab es zwei Möglichkeiten.

Entweder die Prolog-Listen als **Term** an C++ weitergeben und dort den Term auswerten und in die gewünschten **arma** Datentypen konvertieren.

Oder die Prolog-Listen mithilfe des SICStus moduls **struct** in den **array** Datentyp konvertieren, den man mit **pointer(array(float_32))** als Argument in der C++ Schnittstelle nutzen kann und automatisch in ein C++ **float array** konvertiert wird.

Ich habe mich dann für die zweite Variante entschieden, da sie in SICStus deutlich besser dokumentiert ist. Die Prädikate für das Auslesen und Bearbeiten von Prolog Termen in C++ haben fast gar keine Dokumentation, so dass oft nicht klar ist, wie sie richtig genutzt werden sollen oder wie die Inputs und Outputs aussehen.

Globales Methoden Objekt oder Methoden Objekt als Parameter Bei dieser Entscheidung ging es darum, wie auf das initialisierte Objekt der jeweiligen Methoden-Klassen zugegriffen werden soll.

Zum Einen gibt es die Möglichkeit, ein **globales Objekt** der Methode zu definieren, auf das die Funktionen dann zugreifen und dessen Funktionen aufrufen.

Zum Anderen kann man das Objekt der Methode auch als **Parameter** immer an die Funktionen weiter geben. Dafür würde man in einer Initialisierungs-Funktion die Adresse des erstellten Objekts an Prolog zurückgeben und dort dann bei jedem anderen Funktionsaufruf die Adresse mitgeben.

Ich hab mich dabei für das globale Objekt entschieden, da es die einfachere umsetzbare Option war. Und ich der Meinung bin, dass es Nutzerfreundlicher wäre, wenn man die Methode als Adresse nicht immer bei jedem Funktionsaufruf mit führen müsste. Bei beiden Ansätzen sorgt das **Template-System** für einige Probleme 3.6, jedoch sind sie bei dem globalen Objekt Ansatz etwas einfacher zu umgehen, indem man alle Funktionen in eine vereind.

Dazu kommt auch, dass ich den zweiten Ansatz beim testen nicht erfolgreich umzusetzen konnte. Es könnte an meinen nicht besonders ausführlichen Grundkenntnissen von **Pointer** und **Adressen** in C++ liegen, aber es wäre auch möglich, dass der Ansatz, so wie ich ihn mir vorgestellt habe, nicht mit der Schnittstelle umsetzbar ist.

Code Wiederholungen Reduzieren mit Helfer-Dateien Um die Menge an Code zu verringern und eine bessere Übersicht innerhalb der Methoden Dateien zu behalten, habe ich extra **Helfer-Dateien** hinzugefügt. Diese beinhalten Funktionen und Prädikate, die besonders das Konvertieren von Datentypen übernehmen, wie zum Beispiel von **arma::mat** zu **float* array**. Denn die Konvertierungs-Vorgänge werden oft im gesamten Projekt benutzt und unterscheiden sich inhaltlich meistens nur anhand der Variablennamen.

Ähnliche Ordner Struktur wie MLpack Bei der Ordnerstruktur des Projekts hab ich versucht, mich an die Struktur von MLpacks C++ Implementation zu orientieren, damit man einfacher zwischen beiden hin und her wechseln kann und die Methoden an der gleichen Stelle findbar sind.

Extra „Schicht“ an Prolog Prädikaten Zu jeder mit **foreign/3** definierten Funktion habe ich ein weiteres Prädikat mit gleichem Namen hinzugefügt ,die zum einen die simplen Parameter, wie **Integer** und **Floats**, auf besondere Eigenschaften überprüft, z.B. ob sie positiv sind oder im Intervall $[0, 1]$ liegen. Zum anderen übernimmt es das Konvertieren der Prolog-Listen zu **float_array structs**. Dadurch können diese extra Prädikate für die Methoden Module nach außen exportiert werden, anstatt der **foreign** definierten C++ Prädikate. Wodurch man bei den exportierten Prädikaten nur noch normale Prolog-Listen übergeben muss und intern das Konvertieren übernommen wird.

Ohne diesen Schritt müsste jemand, der die Methoden einfach nur ausprobieren will, das Konvertieren der Prolog Listen selber übernehmen, was den Einstieg mit den Methoden unnötig schwerer machen würde.

3.6 Implementierungsprobleme oder entstandene Limitierungen

Keine Matrizen als Parameter Das **struct-Modul** ermöglicht es zwar, zweidimensionale Arrays zu definieren und innerhalb von Prolog zu nutzen, aber wenn man versucht, diese als Parameter in der Schnittstelle zu benutzen, bekommt man einen Compiler Fehler darüber, dass es kein erlaubter Datentyp sei. Daher kann man nur eindimensionale Arrays für die Schnittstelle nutzen. Deshalb sind Matrizen im Projekt als lange Listen, mit der extra Information über eine der Matrix Dimension, definiert.

Dieses Problem entsteht durch die Entscheidung, die ich im Punkt „Nutzen des ‚struct‘ Moduls“ getroffen habe. Hier könnte es den Konvertierungsprozess etwas vereinfachen, die Matrizen doch als Prolog **Term** an C++ zu übergeben.

Template Klassen Einige der MLpack Methoden nutzen Templates bei der Klassen-Definition, um z. B. verschiedene **Tree** Klassen akzeptieren zu können, ohne für jede eine eigene Version der Klasse zu schreiben.

Das ist für die C++ Implementation sehr nützlich, für dieses Projekt bedeutet das jedoch einige Probleme.

Für die meisten Methoden habe ich ein **globales Objekt** der jeweiligen Methoden Klasse definiert, das dann in den angebundenen Funktionen benutzt wird. Aber bei den Methoden mit den **Template-Klassen** ist das mit nur einem globalen Objekt nicht möglich, da in C++ zwei Objekte, welche unterschiedliche **Template-Klassen** haben, wie zwei komplett unterschiedliche Objekte angesehen werden. Und dadurch nicht unter einer Variabel gespeichert werden können.

Bei zwei oder drei möglichen Inputs für das Template ist das zwar unangenehm, aber immer noch ertragbar. Wenn es aber dann mehrere einstellbare **Template-Klassen** in einer Methode gibt, z.B. einen **TreeType** mit 6 möglichen Input und einen **Optimizer** mit 4 möglichen Inputs, müsste man 24 **globale Objekte** anlegen für alle möglichen Kombinationen und dann bei jeder Funktion abfragen, welches der 24 Objekte gerade genutzt werden soll.

Und je mehr Templates es gibt, umso schlimmer wird das Problem.

Glücklicherweise kam dieses Problem eher selten auf, da einige der Methoden, die Templates nutzen, eine **gewrappte Model Version** der Methode haben, die es ermöglichen, die **Template-Klassen** beim Initialisieren mit **Enums** auszuwählen. Die **gewrappte** Model Version zählt dann als einzelne Klasse, die man wieder als einzelnes globales Objekt nutzen kann.

Leider ist das nicht der Fall für alle Methoden mit Templates, weshalb ich bei diesen Methoden dann alles in eine Funktion vereinen musste, um so kein globales Objekt zu brauchen.

Beispiele dafür wären DBScan, KMeans oder Kernal_PCA.

Starke Abhängigkeit von MLpack Version Das Projekt nutzt beim Kompilieren die MLpack Version, die auf dem System installiert ist, daher ist es von dieser Version abhängig. Ich hab das Projekt mit der **3.4.2 Version** auf **Ubuntu 22.04** entwickelt und getestet. Bei anderen Versionen kann sich innerhalb von MLpack das Verhalten verändert haben, was dann zu Fehlern beim Kompilieren des Projekts oder zu fehlgeschlagenden Tests führen könnte. Besonders bei älteren Versionen von Ubuntu bin ich auf das Problem gestoßen. Dass z.B. für **Ubuntu 18.04** in der dafür verfügbaren **MLpack System-Installation** einige Methoden nicht enthalten sind und dementsprechend auch im Projekt nicht nutzbar sind.

Kein Methoden Parameter Überladen Die C++ Schnittstelle unterstützt kein Methoden-Parameter überladen. Beim Versuch Funktionen mit demselben Namen und unterschiedlichen Inputs als foreign Ressource zu definieren kommt es zum **„clashing foreign**

declarations“ Fehler. Daher wurden Funktionen, wie z.B. **Search** mit zwei Überladungen, zu den Funktionen **SearchWithQuery** und **SearchNoQuery** umbenannt. Das hat den positiven Effekt, dass bei den resultierenden Prädikaten anhand des Namens das Verhalten einfacher abgelesen werden kann.

C++ Implementation Vorteile gehen teilweise Verloren Die Geschwindigkeit- und Effizienz-Vorteile, die C++ Version bringt, werden dabei nicht beeinflusst, den die Funktionen der Methoden werden vom Projekt einfach nur aufgerufen und die Ergebnisse an Prolog weitergegeben.

Die Vorteile des **Template-Systems** gehen zum Teil verloren, denn man kann nur noch die vorgefertigten Klassentypen in den Methoden auswählen. Mit dem Template System kann man z.B. selbst geschriebene und konfigurierte TreeTypes benutzen oder den eigenen Optimizer selber vorher initialisieren, was mit der Schnittstelle wegfällt.

Dazu kommt, dass sich die Anbindung im Gegensatz zur C++ Version, nun mehr für simple Machine Learning Projekte geeignet ist und eher keine gute Wahl für komplexe Projekte ist. Genauso wie es der Fall für die anderen Sprachen Anbindungen von MLpack ist.

Zuletzt verliert es einiges von seiner Flexibilität, da es zum einen abhängig von SICStus Prolog ist, was keine frei zugängliche Software wie C++ ist. Zudem hat es dasselbe Problem wie viele andere Machine-Learning-Bibliotheken, dass die Ergebnisse, die man in der Entwicklung und beim Konzeptionieren erzielt, nur aufwändig in eine dauerhaft nutzbare Form umgewandelt werden können.

4 Ergebnisse

Hier dokumentiere ich, wie die Bibliothek kompiliert, getestet und benutzt werden kann, welche MLpack Methoden es in die Implementation geschafft haben und wo sie dokumentiert sind.

4.1 Wie wird die Bibliothek kompiliert

Um die C++ und Prolog der Methoden miteinander zu verbinden, müssen sie einmal mit dem SICStus Tool **splfr** kompiliert werden. Dafür ist in jedem Methoden-Ordner eine **Makefile** angelegt, die den nötigen Befehl für die jeweilige Methode aufruft. Und zusätzlich ist im Root Ordner des Projekts ein **Makefile** angelegt, welches alle anderen **Makefiles** aufrufen kann.

Die Eingabemöglichkeiten der **Makefiles** sind dabei:

- **make** (Für das Kompilieren)
- **make clean** (Entfernt, die beim Kompilieren entstandene Datei)

Der Aufruf des **splfr Tools** für das Kompilieren sieht dann z.B. bei **mean_shift** so aus:

Quellcode 17: SPLFR mean_shift Aufruf

```
1: $ /usr/local/sicstus4.7.1/bin/splfr -larmadillo -fopenmp -lmlpack
2: -lstdc++ -cxx --struct mean_shift.pl mean_shift.cpp
3: ../../helper_files/helper.cpp
```

Wichtig ist hierbei der absolute Pfad des **splfr Tools**. Wenn es woanders installiert wurde, muss der Pfad für die Makefiles geändert werden. Der einfachste Weg dafür ist es das Root Makefile zu nutzen und dort einfach die **SPLFR_PATH** Variabel anzupassen.

Die angegebenen Compiler Flaggen haben folgende Bedeutungen:

- **-larmadillo** (Lädt die installierte Armadillo C++ Bibliothek)
- **-fopenmp** (Lädt OpenMP für parallele Prozesse)
- **-mlpack** (Lädt die installierte MLpack Version)
- **-lstdc++** (Lädt die STD C++ Bibliothek)
- **-cxx** (Stehlt sicher, dass der Compiler den Code als C++ anerkennt)
- **–struct** (Gibt an, dass das SICStus struct-Modul genutzt wird)

Alle C++ Dateien, die von **mean_shift.cpp** included werden und nicht mit den Compiler Flaggen geladen werden, müssen mit angegeben werden.

Wie in diesem Fall **helper.cpp**.

Wenn viele der Methoden bearbeitet wurden, kann das Kompilieren mit dem Root Makefile einige Minuten dauern.

4.2 Wie kann die Bibliothek genutzt werden

Zuerst muss das Git-Projekt[3] auf den eigenen Rechner geklont werden.

[3] https://gitlab.cs.uni-duesseldorf.de/stups/abschlussarbeiten/prolog-mlpack-libary

Da es die C/C++ Schnittstelle von SICStus nutzt, und ich bisher keine Anbindung der Bibliothek an Swi-Prolog hinzugefügt habe, kann die Bibliothek nur mit SICStus kompiliert und benutzt werden.

4.2.1 Nutzen der Bibliothek im eigenen Projekt

Alle Methoden sind kompiliert in der Bibliothek enthalten. Das heißt solange **splfr** und **MLpack** an den selben Positionen installiert sind kann man die Bibliothek direkt in SICStus nutzen, ohne sie neu zu kompilieren.

Zum Laden der Methoden Prädikate wird dieser Befehl genutzt (**mean_shift** als Beispiel):

Quellcode 18: Laden des **meanShift** Modules

```
1: :- use_module('Pfad/zu/.../methods/mean_shift/mean_shift.pl').
```

Der Pfad kommt natürlich darauf an, wo die Bibliothek abgespeichert wird.

4.2.2 Bibliothek bearbeiten oder anpassen

Falls man die Bibliothek anpassen will, muss man die Methoden, die man bearbeitet hat, neu kompilieren.

(Wie das geht, steht im Unterkapitel 4.1)

Beim Kompilieren wird die glue-Datei **MethodenName.so** erstellt, die die Prolog und C++ Dateien verbindet.

Danach können die Methoden wieder, wie im Unterkapitel 4.2.1, genutzt werden.

Zudem ist es wichtig, dass auf dem System die aktuellste Version von MLpack installiert ist. In meinem Fall **MLpack 3.4.2** auf **Ubuntu 22.04**, da bei älteren Versionen die System-Installation von MLpack nicht alle Methoden beinhaltet und so beim compilieren die fehlenden MLpack Methoden Dateien nicht gefunden werden. Sodass es zu Compiler Fehlern kommt.

4.3 Wie testet man die Bibliothek

Zum Testen gibt es wie beim Kompilieren in jedem Methodenordner eine **Test-Prolog-Datei**. Und im **Root Ordner** eine **test_all.pl** Datei, die alle anderen Tests ausführen kann.

Beim Ausführen der Tests ist es wichtig, dass die SICStus Instanz als grundlegenden Arbeits-Ordner den **Root Ordner** der Bibliothek eingestellt hat. Sonst führen die Pfade

zu dem **Iris-Datensatz** an die falsche Stelle, was die Tests unbrauchbar macht.

Mit **test_all.pl** wird dieses Prädikat benutzt:

Quellcode 19: test_all.pl alle Tests ausfuehren

```
1: %% loading the test_all.pl file
2: -? [test_all].
3:
4: %% run all tests included in test_all.pl
5: -? run_tests.
```

Alle Tests zusammen können 1-2 Minuten zum Durchlaufen brauchen.

Wenn man nur einen Test ausführen will, sieht der Befehl dann so aus (**mean_shift** als Beispiel):

Quellcode 20: mean_shift_test.pl ausfuehren

```
1: -? use_module('src/methods/mean_shift/mean_shift_test.pl').
2: %% loading the test Module for mean_shift
3:
4: -? run_mean_shift_tests.
```

4.4 Liste der nutzbaren Methoden

In die **Prolog-MLpack-Bibliothek** haben es 30 Methoden reingeschafft.

Wie z.B. **DBScan**, **kernel_PCA**, **KMeans**, **linear_SVM**, **mean_Shift**, **Perceptron**, **Random_Forest** usw.

Liste der implementierten Methoden [4]

Jedoch sind das nicht alle Methoden aus MLpack.

Decision_Stump (wurde bei der neueren MLpack Version rausgenommen), **GMM**, **KRANN**, **CF**, **DET**, **HMM**, **Preproccessing**, **MVU** und **SVD Methoden Sammlung** habe ich nicht mehr rechtzeitig fertig bekommen.

Besonders GMM, HMM und Preproccessing haben eine deutlich komplexere Implementation und würden noch mehr Zeit benötigen.

Zusätzlich enthält MLpack noch die Methoden **ann**, **matrix_completion**, **nystroem_method**, **reinforcement_learning** und **sparse_autoencoder**, die von den anderen Sprach-Anbindungen nicht als eigenständige Methode implementiert wurden. Mir war dann

[4] https://gitlab.cs.uni-duesseldorf.de/stups/abschlussarbeiten/prolog-mlpack-libary/-/wikis/Home#prolog-methods

nicht ganz klar war, ob es nicht nur Methoden sind, die von anderen Methoden genutzt werden, weshalb ich sie beim Implementieren in der Prioritätenliste eher niedrig eingeschätzt habe. Letztendlich bin ich nicht zu ihnen gekommen.

Die Dokumentation der Methoden richtet sich an die **C++ MLpack** Dokumentation.[5]

Und besonders stark an die Dokumention der **Python Anbindung** [6] von MLpack angelehnt.

Die **volle Dokumentation** [7] der Methoden findet man in der Gitlab Wiki-Spalte der Prolog-MLpack-Bibliothek.

5 Auswertung und Fazit

Jetzt, da man einen Überblick darüber hat, wie die Bibliothek implementiert wurde und was sie im vollen Umfang zu bieten hat, stellt sich natürlich die Frage, ob sich der Aufwand gelohnt hat.

5.1 Erwartungen während der Arbeit

Zu Beginn sah das Projekt recht einfach umsetzbar aus. Da man die in C++ geschriebene Bibliothek MLpack und die dazu passende Schnittstelle von SICStus hatte, wirkte der Implementierungsprozess simple und direkt. Beim Implementieren der ersten Methode sind dann aber im Detail immer wieder Probleme aufgetaucht, die den Prozess komplizierter machten als es zuerst wirkte.

Nachdem die erste Methode implementiert war, wurde es einfacher, den Rest der Methoden, mit nur kleineren Problemen bei einzelnen Methoden, auch zu implementieren.

Die überaschend große Menge an Funktionen pro Methode hat im **Dokumentation und Testen** Teil der Arbeit deutlich mehr Zeit in Anspruch genommen als geplant. Hier waren die Methoden, die in nur einer Funktion zusammen gefasst wurden, am einfachsten umzusetzen.

5.2 Auswertung der Design Entscheidungen und Limitierungen

Wie es bei einem Projekt üblich ist, trifft man Entscheidungen, von denen man erst später weiß, ob sie vorteilhaft waren oder ob sie Probleme mit sich bringen.

[5] https://www.mlpack.org/doc/mlpack-3.4.2/doxygen/
[6] https://www.mlpack.org/doc/mlpack-3.4.2/python_documentation.html
[7] https://gitlab.cs.uni-duesseldorf.de/stups/abschlussarbeiten/prolog-mlpack-libary/-/wikis/Home

Größtenteils waren die Entscheidungen hilfreich und oder nötig für das Projekt, wie z.B. „Eigene C++ Dateien, welche die MLpack Funktionen aufrufen“3.5, wofür es keine wirkliche Alternative gab. Oder die Entscheidungen „Code Wiederholungen Reduzieren mit Helfer-Dateien“3.5, „Ähnliche Ordner Struktur wie MLpack“3.5 und „Extra ‚Schicht‘ an Prolog Prädikaten“3.5, die hauptsächlich das Implementieren erleichtert haben.

Und was Probleme angeht, gab es einige wie „Keine Matrizen als Parameter“ 3.6, „Template Klassen“ 3.6, „Kein Methoden Parameter Überladen“ 3.6, die nicht aus den eigenen Entscheidungen, sondern aus den grundsätzlichen Limitierungen der Schnittstelle entstanden sind.

Besonders die **Template-Klassen** haben für viel Kopfzerbrechen gesorgt.

Bei „Nutzen des ‚struct‘ Moduls“3.5 wäre es im Nachhinein wahrscheinlich besser gewesen, die Matrizen und Vektoren als Terme an C++ zu übergeben, um so ein paar der Konvertierungs-Schritte weglassen zu können.

Beim Problem „Starke Abhängigkeit von MLpack Version“3.6 hab ich die falsche Entscheidung getroffen, mich beim Testen der Eingaben und Ausgaben zu sehr auf die C++ Implementierung von MLpack zu verlassen. Dadurch ist die Prolog-MLpack-Bibliothek sehr abhängig von meiner genutzten MLpack Version. Das ist zwar für das Nutzen der Bibliothek nicht allzu einschränkend, da ein gewisser Grad an Abhängigkeit unvermeidbar ist. Aber für das Testen der Bibliothek ist das ein starkes Problem, da schon ein kleiner Unterschied in der MLpack Version zu falschen Tests führt.

Und zuletzt das Problem „C++ Implementation Vorteile gehen teilweise Verloren“3.6, welches aus einer Mischung meiner Entscheidungen und grundsätzlicher Limitierungen der Schnittstelle entstanden ist. Denn die Vorteile, die Template-Klassen bieten, wird man mit der Schnittstelle nicht an die Bibliothek übertragen können. Die Flexibilität wird dadurch gehindert, dass man von Prolog aus keinen Zugriff auf die von mir hinzugefügten globalen Methoden Objekt hat, was das Exportieren der Ergebnisse in andere Projekte erschwert. Dadurch, dass ich ein paar der Methoden etwas kompakter implementiert habe, fehlt der Bibliothek die Individualisierung-Möglichkeiten der jeweiligen Methoden, die von der C++ Implementierung geboten werden.

5.3 Zukünftige Verbesserungen/Erweiterungen

An dem Projekt gibt es noch einige Stellen, die man Verbessern könnte wie:

- Die fehlenden Methoden noch hinzufügen. (GMM,KRANN,CF,DET,HMM, ...)
- Die Abhängigkeit der Bibliothek an eine bestimmte MLpack Version verringern, besonders für das Testen der Prädikate.

- Anstatt von globalen Methoden Objekten, doch Methoden Objekte als Parameter übergeben, um so besseren Zugriff auf sie zu haben.
- Das Kompilieren optimieren, denn alle Methoden einmal neu zu kompilieren kann 5-10 Minuten dauern. Hier fehlt mir die Expertise, wie man das verbessern könnte.
- Matrix und Vektoren Funktionalitäten für die Daten hinzufügen, z.B. das Auslesen von CSV oder das Vorbereiten der Daten auf die Methoden. Denn das sind Prädikate, die diese Bibliothek noch gar nicht enthält.
- Vergleiche mit der C++ und Python Variante auf Geschwindigkeit.

5.4 Fazit

Zusammengefasst ist die Prolog-MLpack-Bibliothek gut für simple Machine Learning Probleme nutzbar, aber sobald man komplexe Probleme angehen will und die Ergebnisse, die man erreicht in ein nutzbares Produkt portieren will, ist die C++ Variante die bessere Wahl.

Damit reiht sich die Bibliothek mit zu den anderen Sprach-Anbindung von Prolog, die ebenfalls eher für simple Probleme gedacht sind und auch nicht alle Vorteile von MLpack ausschöpfen können.

Für das Projekt war MLpack eine gute Wahl, da die gute Dokumentation und die größtenteils simple Struktur der Bibliothek den Prozess deutlich vereinfacht hat.

Die Schnittstelle von SICStus erfüllt ihren Zweck, aber man merkt, an vielen Stellen, dass sie eigentlich für deutlich einfachere C/C++ Programme gedacht ist und dass sich nicht alle Features von C++ zu Prolog übersetzen lassen. Daher wäre hier das größte Potenzial für Verbesserungen.

Damit bietet die Prolog-MLpack-Bibliothek 30 Machine-Learning Methoden Ansätze, die innerhalb von SICStus Prolog benutzt werden können.

Quellcodeverzeichnis

Literatur

[Ang] Angelopoulos: *mlu : Machine learning utilities*. http://stoics.org.uk/~nicos/sware/mlu/

[CCC22] Ciatto, Giovanni ; Castigliò, Matteo ; Calegari, Roberta: Logic Programming library for Machine Learning: API design and prototype. (2022)

[CE] Curtin, Soni ; Edel: *An Automatic Benchmarking System*. https://www.mlpack.org/static/pub/2014automatic.pdf

[CEL+18] Curtin, Ryan R. ; Edel, Marcus ; Lozhnikov, Mikhail ; Mentekidis, Yannis ; Ghaisas, Sumedh ; Zhang, Shangtong: mlpack 3: a fast, flexible machine learning library. In: *Journal of Open Source Software* 3 (2018), 726. http://dx.doi.org/10.21105/joss.00726. – DOI 10.21105/joss.00726

[CM10] Carlsson, Mats ; Mildner, Per: *SICStus Prolog – the first 25 years*. http://dx.doi.org/10.48550/ARXIV.1011.5640. Version: 2010

[Dia] Diaz: *Gnu Prolog, Interfacing Prolog and C*. http://www.gprolog.org/manual/html_node/gprolog065.html

[Hop] Hoppe: *Machine Learning Algorithms Implemented in Prolog*. https://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/learning/systems/learn_pl/0.html

[Kow88] Kowalski, Robert A.: The early years of logic programming. In: *Communications of the ACM* 31 (1988), Nr. 1, S. 38–43

[LF11] Lally, Adam ; Fodor, Paul: Natural language processing with prolog in the ibm watson system. In: *The Association for Logic Programming (ALP) Newsletter* 9 (2011)

[LLG12] Lu, Benjie ; Liu, Zhiqing ; Gao, Hui: An adaptive prolog programming language with machine learning. In: *2012 IEEE 2nd International Conference on Cloud Computing and Intelligence Systems* Bd. 1 IEEE, 2012, S. 21–24

[Mer12] Merritt, Dennis: *Building expert systems in Prolog*. Springer Science & Business Media, 2012

[MG17] Müller, Andreas C. ; Guido, Sarah: *Einführung in Machine learning mit Python: Praxiswissen data science*. O'Reilly, 2017

[mlp] *A Vision for an Efficient Prototype-to-Deployment Machine Learning Library*. https://www.mlpack.org/papers/vision.pdf

[Nea] Neaves: *Basic machine learning algorithms in prolog*. https://github.com/samwalrus/machine_learning

[OFDR17] Orsini, Francesco ; Frasconi, Paolo ; De Raedt, Luc: kProbLog: an algebraic Prolog for machine learning. In: *Machine Learning* 106 (2017), Nr. 12, S. 1933–1969

[Pia] Piatetsky: *Python leads the 11 top Data Science, Machine Learning platforms: Trends and Analysis*. https://www.kdnuggets.com/2019/05/poll-top-data-science-machine-learning-platforms.html

[Riga] Riguzzi: *cplint*. https://github.com/friguzzi/cplint

[Rigb] Riguzzi: *cplint-datasets*. https://github.com/friguzzi/cplint_datasets

[RPN20] Raschka, Sebastian ; Patterson, Joshua ; Nolet, Corey: Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence. In: *Information* 11 (2020), Nr. 4, S. 193

[SC16] Sanderson, Conrad ; Curtin, Ryan: Armadillo: a template-based C++ library for linear algebra. In: *Journal of Open Source Software* 1 (2016), Nr. 2, 26. http://dx.doi.org/10.21105/joss.00026. – DOI 10.21105/joss.00026

[SC18] Sanderson, Conrad ; Curtin, Ryan: A User-Friendly Hybrid Sparse Matrix Class in C++. In: Davenport, James H. (Hrsg.) ; Kauers, Manuel (Hrsg.) ; Labahn, George (Hrsg.) ; Urban, Josef (Hrsg.): *Mathematical Software – ICMS 2018*. Cham : Springer International Publishing, 2018. – ISBN 978–3–319–96418–8, S. 422–430

[Wiea] Wielemaker: *Dynamic calling C from Prolog*. https://github.com/JanWielemaker/ffi

[Wieb] Wielemaker: *SWI-Prolog C++ interface*. https://github.com/SWI-Prolog/packages-cpp