Ausarbeitung:
Ein optimierter Klassifizierungsmechanismus zum *machine learning*
Gruppe: ‚sparc'
Nicolas Meisberger
Robin Franzke

Zu Beginn der Projektausschreibung hatten wir die freie Wahl zwischen 13 sich jeweils differenzierenden Projekten bzw. Programmierarbeiten im Team. Wir entschieden uns direkt für das oben angegebene Projekt, da wir uns stark für künstliche Intelligenz und das ‚künstliche Lernen' interessieren. Da einer von uns bereits eine Seminararbeit darüber geschrieben hatte, war es nicht allzu schwer, das Grundverständnis des Ganzen zu verstehen.
Unser Ziel ist es also einen Algorithmus zu implementieren, welcher aus Datensätzen lernen soll um später klassifizieren zu können. Wir bekamen eine in der Projektbeschreibung angegebene Orientierung bzgl. des Algorithmus'. Den dort angegebene HULLER-Algorithmus sollten wir mit Hilfe eines ‚Papers' (http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.105.8314) verstehen und implementieren.
Zunächst werden wir in dieser Ausarbeitung anfangs kurz auf die grobe Struktur des HULLER-Algorithmus, welche ebenso im angegeben ‚Paper' beschrieben wird, darstellen und verständlich wiedergeben. Danach gehen wir explizit auf unsere Implementierung und Umsetzung des Ganzen ein.

HULLER-Algorithmus:
Wir bekommen anfangs eine sogenannte „Trainingsmenge" als Eingabe die Punkte enthält, welche entweder als ‚negativ' oder ‚positiv' deklariert sind und eine gewissen Dimension mit sich führen. Wir können uns das Ganze so vorstellen, dass in einem Raum (bezogen auf die jeweilige Dimension) einzelne Punkte durch ihre angegebene Klassifizierung und Koordinaten dargestellt werden. Daraufhin können wir eine sogenannte ‚Convex-Hülle' ermitteln für die jeweilige Klassifizierungs-Klasse, d.h. wir verbinden die Punkte so, dass jede Klassifizierungsart einen ‚Rahmen' bildet. Des Weiteren betrachten wir die Punkte der ‚Convex-Hülle' und deren geringsten Abstand zur nächsten ‚Convex-Hülle' (etwa durch Median-Berechnung der Punkte der Klassifizierungs-Klasse). Wir deklarieren hier die Punkte für die ‚positive'-Klassifizierung als X_P und die Punkte der ‚negativen'-Klassifizierung mit X_N. Wir können eine Hyperebene zwischen die einzelnen ‚Convex-Hüllen' legen, wenn wir die im Paper angegebene Formel nutzen.
Eine relevante Rolle spielt noch der α-Wert der jeweiligen Klassifizierungs-Klassen, welcher für die jeweilige Klasse aufsummiert 1 ergeben muss und die Gewichtung der einzelnen Punkte ist.

Aus Platzgründen bzgl. der Beschreibung des HULLER-Algorithmus referenziere ich an dieser Stelle auf das ‚Paper'.

Unsere Implementierung:

Zunächst überlegten wir, wie wir am Besten das Programm aufteilen. Wir entschlossen uns dafür, dass wir es aus 2 Hauptfunktionen aufbauen; einmal wäre das das „Lernprogramm“ und der „Klassifizierer“.
Das Lernprogramm soll also die Trainingsmenge als Eingabe bekommen, diese einlesen und dann daraus das Hauptmodell erstellen, mit dem der Klassifizierer weiterarbeiten kann. Dieses ist dafür da, um später richtig klassifizieren zu können. Die erste Hauptfunktionen des Klassifizierers ist also die Fähigkeit, das erstelle Modell einlesen zu können. Nach dem Einlesen erhalten wir die zu klassifizierende Datenmenge als Eingabe, für die natürlich logischerweise der Klassifizierer zuständig ist. Des Weiteren findet dann die erwartete Klassifizierung und deren Ausgabe statt.

- point.c ; hullerutil.c

Wir fingen zunächst an, diverse mächtige ‚structs’ zu bilden, mit denen wir die relevanten Punkte aus den Mengen repräsentieren können. Das ist besonders wichtig, da diese Punkte einige Eigenschaften mit sich liefern. In einer struct können wir dadurch alles auf einmal speichern. Beispielsweise wäre das anfangs die struct ‚samples’, welche 2 Arrays mit sich führt. Dieses Array beinhaltet Zeiger, welche auf Punkte referenzieren, die – entweder positiv oder negativ – klassifiziert sind. Des Weiteren wird hier die jeweilige später vorgestellte Alphaliste mitgeführt sowie die Anzahl der positiven und negativen Punkte. (Jeder einzelne Punkt erhält eine alpha-Gewichtung)
Eine weitere struct ‚huller’ führten wir ein um die relevanten (im Paper beschrieben) Punkte repräsentieren zu können, diese wären X_P und X_N, sowie deren Skalarprodukte, wofür wir uns eine extra Funktion geschrieben haben (double dotP). Wir nutzen im gesamten Programm statt ‚float’ den Datentypen ‚double’, um eine bessere Genauigkeit zu erhalten.
Da wir beim Einlesen der Punkte auch einen Punkt im „dim“-dimensionalen Raum erstellen müssen, schrieben wir die Funktion „point *createPoint“, welche diversen Speicher für das Array und für das struct durch dynamischen ‚calloc’ reserviert. „dim“ wäre hier die die Anzahl der n-Attribute der Objekte. Die dazu entgegenwirkende Funktion (also die zum Speicher freigeben) nannten wir ‚destroyPoint’, die mit der in C eingebauten Funktion ‚free’ agiert. Für spätere Testzwecke implementierten wir eine ‚randomPoint’-Funktion, die uns einen zufälligen dim-dimensionalen Punkt erstellt und ausgibt. Die Ausgabefunktion lautet ‚printPoint’ und iteriert über die Koordinateneinträge, gibt diese und die dazugehörige Dimension aus per printf.

Eine sehr relevante Rolle spielt unsere ‚readSamples’-Funktion, welche die gegebenen Datensätze einlesen kann. Ein Datensatz kann mehrere zehntausende Einträge besitzen, deswegen ist hierbei die Performance sehr wichtig. Wir gehen in einem Datensatz von folgender Attribut-Formatierung aus:
(+/-) 1:%lf 2:%lf ... dim:%lf)
Unsere Einlesefunktion hat folgende Parameter: einen Pointer auf die Datei (Trainingsmenge), die Dimensionenanzahl und einen Zeiger auf das struct ‚samples’. Aus Sicherheitsgründen reservieren wir für den Einlesebuffer per ‚malloc(500000)’ genügend Speicher – wir wissen nämlich nicht, wie lang eine Zeile werden kann. Wir können die Funktion ‚fscanf’ hier nicht nutzen, da die Zeilen im Datensatz verschieden viele Einträge haben können. Stattdessen benutzen wir ein zeilenweises Einlesen der Einträge, wofür wir eine while-Schleife und diverse Hilfsvariablen nutzen. Hierbei werden die schon vorhandenen Klassifizierungen und die weiteren

Einträge im struct ‚samples' gespeichert. Gut zu wissen wäre an dieser Stelle, dass man nach dem Einlesen eines Leerzeichens wissen sollte, dass eine Komponente zu Ende ist und ggf. die Neue beginnt. Dementsprechend wird die Komponente (reicht von „fundort_letztes_leerzeichen" bis „aktueller fundort") ausgewertet. Die erste Komponente ist zwingend immer die Klassifizierung (+-1).
Eine der relevanten Hauptfunktionen wäre „initHuller", welche den Huller initialisiert, d.h. diese Funktion berechnet den Durchschnitt von„AVGCOUNT"- (zuvor definiert) vielen positiven und von „AVGCOUNT"-vielen negativen Punkten, um X_P und X_N auszurechnen. Hierbei nutzen wir viele selbstgeschriebene Hilfsfunktionen wie ‚pointDiv', ‚pointCopy', ... , was an eine Assembler-ähnliche Programmierung erinnert. Am Ende der Funktion werden noch die wichtigen Skalarprodukte berechnet, (sehr wichtig für die spätere Ebenenaufstellung).
Es existiert noch eine weitere Hauptfunktion, die ‚updateHuller'-Funktion, welche als Parameter 3 Zeiger auf structs bekommt um die relevanten Daten entsprechend speichern zu können.

- huller.c

Im Paper werden diverse Gleichungen beschrieben um entsprechende Werte zu updaten, was logischerweise auch nötig ist im Laufe des Einlesens und der Verarbeitung der Daten. „updateHuller" springt durch jeweilige if-Abfragen in die gewünschten Gleichungen. Es wird im gesamten Programm strikt nach positiven und negativen Punkten separiert.
Um entsprechende Punkte auszugeben entwickelten wir eine ‚printSamples' und ‚printPoints' Funktion. Wir geben ‚Huller' beispielsweise in der Learn-Funktion in stdout aus. Alle anderen Einträge/Elemente werde in stderr ausgegeben. So kann man bequem mit der im Terminal einzugebenben ‚Funktion' > arbeiten, die die Ausgaben in einer separaten Datei speichert.
Kurz zum allgemeinen Verständnis: Grundsätzlich sind alle alpha-Werte = 0, außer die, die bei ‚initHuller' für das Average gewählt werden, diese sind 1/AVGCOUNT, da sie aufaddiert 1 ergeben müssen.
Die erste Hauptfunktion vom Klassifizierer ist das Einlesen des erstellen Modells. Dies haben wir in der Funktion ‚HullerFromFile' verwirklicht. Unser Einlesen funktioniert im Grunde wie ein DFA. Sobald ein Schlüsselwort kommt, wechselt unsere Funktion in den nächsten Zustand. Bis das jedoch passiert, werden die Werte eingelesen (X_P ,X_N, X_PX_N, X_P X_P ,X_N X_N) und dementsprechend gespeichert. Erreichen wir das in unserer ausgelagerten Datei (durch ‚>'-Befehl) Wort „ende", wissen wir, dass das Modell fertig eingelesen wurde. Diese Datei heißt bei uns „modell.txt" – kann jedoch beliebig genannt werden.

- alphalist.c

Da wir im wesentlichen auch auf die Optimierung des Algorithmus' sowie auf die Perfomance achten sollen, implementierten wir sogenannten „Alpha-Listen". Da unser Algorithmus oft und viel mit den Alpha-Werten arbeitet, sollte so eine Datenstruktur recht schnell sein im Bezug auf das Auslesen und das Sortieren. Die für die Ebene zu berechnenden Punkte sind für uns die Relevanten, die einen Alpha-Wert ungleich null haben. Diese Alpha-Werte verändern sich dauernd (z.B. bei dem ‚updateHuller'-Aufruf). Zunächst jedoch zu den Alphalisten. Anfangs erstellen wir wieder eine mächtige struct namens alphalist, welche „nur" int-Werte bzgl. dem Vorkommen der Indizes von Punkten gleich null und Indizes von Punkten ungleich null sowie deren jeweilige Anzahl speichert. Die eigentliche Alphaliste bauten wir wieder aufgrund der strikten ‚positiv' und ‚negativ'-Klassifizierung in 2 separate Listen auf, welche einmal Indizes von den positiven- und einmal die Indizes von den negativen Punkten speichern, (zu finden in der Datei ‚alphalist.c'). Diese agiert

wieder mit ‚malloc' und ‚calloc', um entsprechenden Speicher (z.B. bezogen auf die Größe der Alphaliste) reservieren und nutzen zu können. Da wir im gesamten Projekt auf diverse Funktionen referenzieren, arbeiten wir sehr viel mit Pointern, gekennzeichnet mit ‚*Funktinosname'. Die wieder entgegenwirkende, speicherlösende Funktion heißt ‚destroyAlphalist', welche nach der Rückgabe der Alphaliste der vorherig ausgeführten Funktion (createAlphaList) den zuvor reservierten Speicher wieder frei gibt. Um nun im Laufe der „Alpha-Wert-Generierung" neue Alpha-Werte in die Liste eintragen zu können, schrieben wir die Funktion ‚addAlpha'. Um nun ein Alpha-Wert aus der Liste in Zeit o(1) löschen zu können, benötigen wir die Funktion ‚removeAlpha', welche zunächst als Parameter einen Pointer auf die Alpha-Liste, einen Integer-Wert sowie einen double-Alpha-Wert bekommt. Durch eine simple if-Abfrage können wir ermöglichen, dass die Zeit o(1) wirklich sichergestellt wird und die Alpha-Liste sich bei jedem Durchlauf um die Länge von 1 verringert. Durch realloc erreichen wir, dass der Speicherblock einen neuen Speicherblock erhält.
Diverse andere Funktionen finden sich im Programmierblock, welche Alpha-Werte zwischen den Listen (Alpha-Liste 1 = alle Alphas mit Wert = 0, Alpha-Liste 2 = alle Alphas mit Wert != 0) aufgrund von Wertänderung verschieben und diese ausgeben können.

- huller.c

Wir erhielten von einer hochrangigen Person einen Tipp, dass wir dadurch ein besonderes Abbruchkriterium bauen können, welches uns hilft, nicht sinnlos über Funktionen zu iterieren, wenn sich sowieso nichts mehr bei den Alpha-Werten ändert, deswegen fügten wir dieses Kriterium in ‚mainHuller' ein. Wir überprüfen beim Einlesen mit der Hilfsvariable ‚nochange_alpha', wie lange sich die Alpha-Werte nicht geändert haben. Falls sich jedoch ein Alpha-Wert ändert, setzen wir den Zähler entsprechend zurück und fangen neu an zu zählen. Woher wissen wir jedoch, ob sich ein Alpha-Wert ändert oder nicht? Genau deswegen definierten wir EPSILON mit 0.0000000001 und iterieren mit 2 for-Schleifen über positive und negative Punkte. Wir schauen im Grunde einfach, ob der jeweilig betrachtete Alpha-Wert => EPSILON bzgl. seiner Veränderung hat.
Falls sich keine Alpha-Werte mehr ändern, brechen wir die Lernfunktion ab. (Wieso sollten wir auch ohne Lernerfolg über den ganzen Datensatz iterieren, wenn wir keine Informationen daraus gewinnen können?!)

Eine weitere Aufgabe – neben dem Implentieren des HULLER-Algorithmus – sollten wir die Performance/Laufzeit und die Trefferstatistik mit dem in einer C-Bibliothek eingebauten Lernalgorithmus ‚libsvm' vergleichen. Wir stießen auf folgende Ergebnisse:

Trefferquote mit ALPHACONV = 16 und EPSILON = 0,0000000001

Datenset I
LIBSVM: Cross Validation Accuracy = 82.5545%

Huller:
Trefferquote: 74.143070 %
Trefferquote: 31.594635 %
Trefferquote: 67.809240 %
Trefferquote: 71.833085 %
Trefferquote: 48.211624 %
Trefferquote: 24.813711 %
Trefferquote: 68.330849 %
Trefferquote: 74.515648 %
Trefferquote: 73.323398 %
=> 59,37%

Datenset II

LIBSVM:Cross Validation Accuracy = 81.6336%

Huller:
Trefferquote: 74.489796 %
Trefferquote: 73.979592 %
Trefferquote: 73.915816 %
Kritischer FEHLER! Es gibt aufgrund von rechnerischen Ungenauigkeiten keine Punkte mehr mit alpha!=0
Trefferquote: 73.724490 %
Trefferquote: 44.961735 %
Trefferquote: 62.181122 %
Trefferquote: 74.489796 %
Trefferquote: 74.489796 %
=> 69%

Datenset III

LIBSVM:Cross Validation Accuracy = 83.2967%

Huller:
Trefferquote: 71.633086 %
Trefferquote: 67.815483 %
Kritischer FEHLER! Es gibt aufgrund von rechnerischen Ungenauigkeiten keine Punkte mehr mit alpha!=0
Trefferquote: 74.549311 %
Trefferquote: 74.708378 %
Trefferquote: 74.708378 %
Trefferquote: 74.708378 %
Kritischer FEHLER! Es gibt aufgrund von rechnerischen Ungenauigkeiten keine Punkte mehr mit alpha!=0
Trefferquote: 68.981972 %
=>72%

Datenset IV

LIBSVM:Cross Validation Accuracy = 83.5599%

Huller:
Trefferquote: 67.597293 %
Trefferquote: 25.380711 %
Trefferquote: 74.619289 %
Trefferquote: 25.380711 %
Trefferquote: 25.380711 %
Trefferquote: 74.619289 %
Trefferquote: 25.380711 %
Trefferquote: 74.619289 %
Trefferquote: 74.619289 %
=>52%

Laufzeit:
Huller (Nehme Laufzeit des Lernens und des Klassifizierens)
(Epsilon- und ALPHACONV-Wert wie oben)
Datei: a1a.t2 (~31k Einträge)
1.229s + 0.397s = 1.626s

LIBSVM (Gemessen mit svm-train ohne Parameter)
56.712s

Fazit der Benchmark-Analyse:
Wir erkennen anhand der angegebenen Benchmark-Daten, dass unser implementierter HULLER-Algorithmus wesentlich schneller bezogen auf die Laufzeit gegenüber der von LIBSVM ist. Eine negative Erkenntnis springt uns jedoch direkt ins Auge wenn wir unsere Trefferquoten betrachten, denn die vom HULLER-Algorithmus ist wesentlich ungenauer als die von LIBSVM. Nun kann der Anwender also entscheiden, ob ein schnelles (jedoch ungenaues) Ergebnis her muss – dann wendet er unseren HULLER-Algorithmus an – oder ob ein langsamen (und genaueres) Ergebnis her soll – dann wendet er LIBSVM an.

Um nun noch analysieren zu können, wie viele Instruktionen per Zyklus (bzw. wie viele Instruktionen der CPU in einem Takt) schafft, rechnen wir die sogenannte IPC-Zahl mit ‚perf' aus, der genaue Ermittlungs-Command lautet:

```
perf stat -B -r 5 ./huller learn a1a.svm 124 >m
```

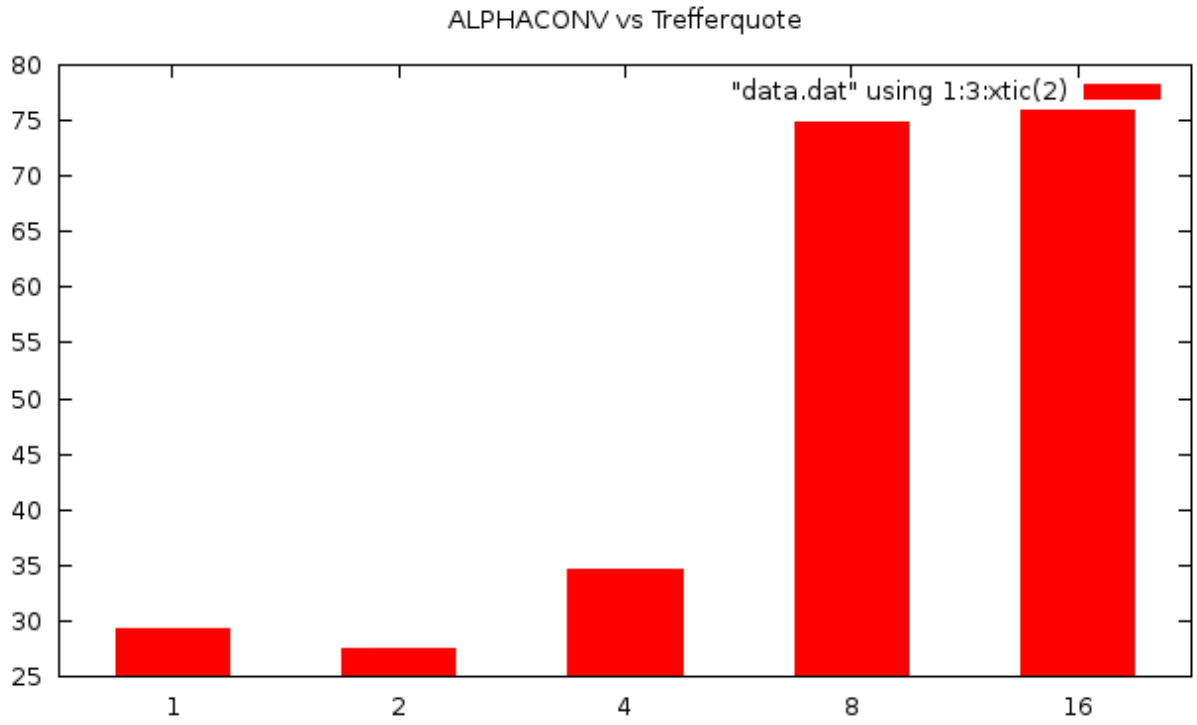
Dieser Befehl spuckt uns ein Ergebnis aus, wir geben hier jedoch nur das ‚gekürzte' relevante an. Unsere IPC-Zahl beträgt 1,52, d.h. wir haben 1,52 Instruktionen pro Zyklus :

```
377.133.447 Instruktionen          #    1,52  insns per cycle
```

ALPHACONV - Trefferquote

1	-	29,38%
2	-	27,5%
4	-	34,6%
8	-	74,8%
16	-	75,9%

„ALPHACONV“ gibt uns einfach nur an, wie lange die Alphas um EPSILON gleich sein müssen, damit wir terminieren.
Als Graph sieht das Ganze wie folgt aus:



Zusatz:

Wir programmierten ausschließlich von Nicols Meisberger’s Laptop, da es auf dem MacBook von Robin Franzke schwierige Installations- und Befehlsprobleme gab bzgl. Linux, Valgrind und Bibliotheken-Einbindungen.