Implementierung und Evaluation von Mixing-Modell-Alternativen für XCS

Markus Weis

1 EINLEITUNG

In modernen Michigan-Style Learning Classifier Systemen (LCS) wird üblicherweise aus einzelnen Classifier Vorhersagen eine Gesamtvorhersage berechnet. In XCS beispielsweise wird die Prediction über eine Fitness gewichtete Summe der einzelnen Predictions berechnet. Diese Fitness ist ein heuristisch Parameter, der inkrementell berechnet wird. Wilson schreibt, dass es wohl verschieden Möglichkeiten gibt die Gesamtvorhersage zu berechnen, behält aber dieses Modell ohne weitere Ausführung in seinen späteren Arbeiten bei [4]. Drugowitsch setzt in seinem Buch "Design and Analysis of Learning Classifier Systems - A Probabilistic Approach"[1] dort an. Er beschreibt verschiedene Mixing Modelle und wertet diese empirisch aus. Eines dieser Modelle wird in der vorliegenden Arbeit beschrieben und eine Implementierung versucht: Inverse Variance Mixing. Aus den heuristisch Modellen schnitt dieses stets am besten ab, während die Laufzeit der analytischen Lösung nicht angemessen ist. Das Modell wird im Open-Source Projekt scikit-XCS [3], einer scikit-learn Erweiterung für XCS, implementiert und anschließend ausgewertet, ob es sich um eine sinnvolle Mixing-Alternative handelt.

2 INVERSE VARIANCE MIXING

Die Definition des Inverse Variance Mixing ist [1] entnommen.

$$\tau_k^{-1} = (c_k - D^X) \sum_n^N m_k(x_n)(\hat{w}_k^T x_n - y_n)^2$$

Dabei ist τ_k^{-1} die Varianz und folglich τ_k die inverse Varianz. c_k ist der Matchcount des Classifiers k, D^X ist die Dimension des Input Raums. $m_k(x)$ ist eine Matching-Funktion, für die gilt: $m_k(x) = 1$, falls die Bedingung des Classifiers k auf x passt und $m_k(x) = 0$ sonst. Der hintere Term stellt das quadratische Fehlermaß der Regression dar. Da das vorliegende XCS System nur für Klassifikation geschrieben wurde, muss hier angepasst werden.

Die erste Implementierung lautet wie folgt:

$$\tau_k^{-1} = (c_k - D^X) \sum_n^N m_k(x_n) f_k(x_n)$$

Wobei $f_k(x)$ den Klassifikationsfehler darstellen soll: $f_k(x) = 0$, falls Classifier k die richtige Klasse für Input x hat und $f_k(x) = 1$, sonst.

Später wurde statt der Klasse der Fehler der Prediction als Fehlermaß verwendet:

$$\tau_k^{-1} = (c_k - D^X) \sum_n^N m_k(x_n)|p_k - P_k(x)|$$

p_k ist die Prediction des Classifiers k, $P_k(x)$ ist der Reward, der ausgeschüttet wird, wenn Classifier k auf Input x seine Klasse vorhersagt. Anstelle des Betrags wurde auch versucht den Fehler zu quadrieren, dies hatte aber keine relevanten Auswirkungen.

Schließlich wird die inverse Varianz nicht direkt benutzt, sondern der sogenannte Gating Parameter g_k über das aktuelle Matchset berechnet:

$$g_k(x) = \frac{m_k(x)\tau_k}{m_i(x)\sum_i^K \tau_i}$$

Dadurch wird sichergestellt, dass die Summe aller Gating Parameter für jeden Input 1 ergibt.

3 PROBLEME

Im Buch wurde Inverse Variance Mixing nie im XCS-Context benutzt. Stattdessen wurde sie im allgemeineren Context von LCS definiert. LCS Classifier im Buch wurden durch Batch Learning trainiert und nicht iterativ, wie es bei XCS der Fall ist. Ein erster Ansatz der Implementierung hier, war einfach die Fitness der einzelnen Classifier durch die inverse Variance zu ersetzten. Daraus ergaben sich aber 2 Probleme:

1. Da man für alle Classifier, die für die aktuelle Prediction verantwortlich sind, die Fitness updatet, müsste man für alle diese die inverse Varianz neu berechnen. Dazu wird aber für jeden Classifier das komplette Trainingsset durchlaufen. Dies führt zu einer nicht akzeptablen Laufzeit. Hier müsste wie im restlichen XCS ein iterativer Ansatz verfolgt werden. Dies wurde hier nicht gemacht, da man sich dadurch noch weiter von der ursprünglichen Idee entfernt und weiter Hyperparameter eingefügt werden müssten. Stattdessen wurde die Laufzeit erstmal in Kauf genommen und geschaut, ob auf diese Weise gute Ergebnisse auftreten.

2. Das Erzeugen neuer Classifier benutzt die Fitness als Auswahlkriterium. Allerdings ist die Inverse Variance zu Beginn des Training Vorgangs noch nicht als sinnvoll zu betrachten. Beispielsweise wird ein Classifier, der auf einen Input perfekt passt stets bevorzugt, egal ob es allgemeinere Classifier gibt, die ähnlich gut und eventuell genereller sind.

Wir sind schließlich zu dem Schluss gekommen, die Fitness des ursprünglichen XCS Systems in Ruhe zu lassen und die Inverse Variance nur für das Mixing zu benutzten. Das Problem mit der Laufzeit bleibt aber bestehen, da die Prediction bei nicht Explorationsiterationen für die Auswahl der Action berechnet werden muss. Schließlich wurden 2 Alternativen implementiert. Die eine macht die Prediction während des Trainings bereits mit der inversen Varianz und zeigt deshalb eine sehr schlecht Laufzeit auf, sie wird mit *continous-update* bezeichnet. Bei der anderen, gennant *only-mixing*, wird die neue Mixing Art nur bei Predictions benutzt, die nicht während des Trainings auftreten, also nur im fertigen Model. Das hat den Vorteil, dass die inverse Varianz für alle Classifier am Ende des Trainingsprozesses berechnet werden kann und für die einzelnen Prediction nur der Gating Parameter aktualisiert werden muss.

Wie bereits beschrieben handelt es sich bei der eigentlichen Definition um Regression. Bei Regression ist es unwahrscheinlich bzw. unmöglich, je nach Modell, auf einen perfekten Classifier zu treffen. Das Problem bei einem perfekten Classifier ist, dass der hintere Teil der Formel, also das Fehlermaß, Null ergibt. Das wird zum Problem, wenn man zur Berechnung der inversen Varianz den Umkehrbruch bildet. Im Buch ist das nicht weiter definiert. In dieser Implementierung habe ich in diesem Fall Unendlich (numpy.inf) eingesetzt. Dadurch bekommt man schließlich im Mixing Probleme. Zwar bekommt man bei einem Mixing-Parameter mit $\tau_k = \infty$ bei allen anderen Classifiern, die nicht Unendlich als inverse Varianz haben, 0 raus, allerdings ist $\frac{\infty}{\infty}$ bzw. $\frac{numpy.inf}{numpy.inf}$ nicht definiert, was für alle Classifier mit $\tau_k^{-1} = \infty$ geschieht:

$$g_k(x) = \frac{m_k(x)\tau_k}{m_i(x)\sum_i^K \tau_i} = \frac{\infty}{\infty}$$

Stattdessen wird g_k im Falle eines Classifiers k im Matchset mit $\gamma_k = \infty$ wie folgt berechnet: $g_k = \frac{1}{c_{inf}}$ für alle Classifier k mit $\gamma_k = \infty$ und $g_k = 0$ für alle anderen Classifier. c_{inf} ist die Anzahl der Classifier mit $\gamma_k = \infty$.

Ein weiteres Problem ergab sich im Fall, dass der Matchcount c_k für einige Classifier geringer als die Dimension des Inputs war, sodass sich für die Varianz und die negative Varianz ein negativer Wert ergibt. Im Buch ist nicht beschrieben, wie in diesem Fall mit den Classifiern umgegangen werden soll. In meiner Implementierung werden diese Classifier bei der Berechnung der Prediction einfach ignoriert.

4 ERGEBNISSE

Die Implementierung wurde auf Multiplexern verschiedener Größe evaluiert: 6 Bit, 11 Bit und 20 Bit. Die Trainingsdauer und die Hyperparameter orientieren sich an [2]. Dort ist zu entnehmen, wie viele Iterationen XCS braucht, um die verschieden großen Multiplexer perfekt zu lernen. Letztendlich wäre jedes der hier zu evaluierenden Implementierungen bei den Problemen zu einer perfekten Accuracy gekommen, wenn man nur lange genug trainiert. Es wäre wohl sinnvoll die Implementierungen auch auf Problemen zu testen, bei denen XCS keine perfekte Lösung finden kann. Dies wird hier allerdings nicht gemacht. Es wird für 4000, 10000, 40000 Iterationen für den 6-Bit, 11-Bit und 20-Bit Multiplexer trainiert. Die durchschnittliche Accuracy der Cross-Validation der verschiedenen Implementierungen ist der Tabelle 1 zu entnehmen. Dabei wird jeweils eine 10x5 Cross-Validation durchgeführt. Es sei hier nochmal erwähnt, dass das kontinuierliche Update der inversen Varianz zu sehr langsamer Laufzeit führt und deshalb die Ergebnisse nicht auf größeren Datensätzen ausgewertet werden können. Es wird neben den beiden bereits erwähnten Implementierungen der inversen Varianz noch eine Gleichgewichtung der Classifier vorgenommen, um den Effekt des Mixings vergleichen zu können (*envely-distributed*).

	6 Bit	11 Bit	20 Bit
normal	0,98 (+/- 0,03)	0,98 (+/- 0,02)	0,89 (+/- 0,05)
evenly-distributed	0,88 (+/- 0,06)	0,92 (+/- 0,04)	0,84 (+/- 0,05)
only-mixing	0,87 (+/- 0,06)	0,90 (+/- 0,04)	0,84 (+/- 0,05)
continuous-update	0,93 (+/- 0,04)	0,93 (+/- 0,03)	0,85 (+/- 0,05)

Tabelle 1: Durchschnittliche Accuracy der verschiedenen Implementierung mit Standardabweichung in Klammern.

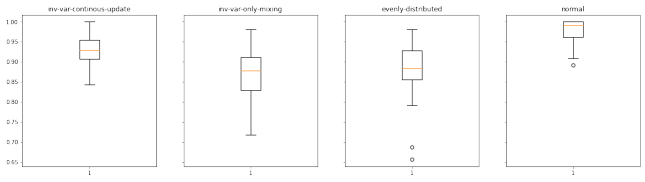

Abbildung 1: Accuracy Werte des 6 Bit Multiplexers

Es ist hier bereits zu erkennen, dass die Ergebnisse des inverse Variance Mixing schlechter sind als die des normalen XCS-Mixing. Genauere Auswertungen der verschiedenen Problem-Größen sind hier aufgeführt.

4.1 6 Bit Multiplexer

Die Boxplots zu den Accuracy Werten aus Tabelle 1 sind in Abbildung 1 zu sehen. Das normale XCS schneidet am besten ab. Evenly-distributed ist ähnlich gut wie only mixing. Continuous-update ist besser als diese beiden Implementierung, allerdings deutlich schlechter als das normale XCS.

Schließlich wurde noch ein TTest durchgeführt, um zu überprüfen, ob die Unterschiede der Algorithmen statische

	inv-var-continous-update	inv-var-only-mixing	evenly-distributed	normal
inv-var-continous-update	0.000000	0.004355	0.034659	0.000425
inv-var-only-mixing	0.000000	0.000000	0.559876	0.047220
evenly-distributed	0.000000	0.000000	0.000000	0.001125
normal	0.000000	0.000000	0.000000	0.000000

Abbildung 2: 5x2 Cross-validation paired ttest auf dem 6 Bit Multiplexer

Signifikanz aufweisen. Hierfür wurde die Funktion *paired_ttest_5x2cv*() der MLxtend Library von Sebastian Raschka verwendet. Es wurde ein Signifikanzwert von $\alpha = 0,05$ verwendet. Das bedeutet die Unterschiede zwischen den Implementierungen sind signifikant, falls das Ergebnis der des *paired_ttest_5x2cv*() einen Wert unter 0,05 liefert. Diese Einträge sind gelb hinterlegt. Die Ergebnisse sind Abbildung 2 zu entnehmen.

Der TTest erkennt Unterschiede zwischen allen Verteilungen, außer zwischen evenly-distributed und inv-var-only-mixing. Dies spricht gegen die Verwendung der inv-var-only-mixing Implementierung, da der zusätzliche Aufwand keinen Nutzen zeigt.

4.2 11 Bit Multiplexer

Die Boxplots sind in Abbildung 3 zu sehen. Wieder schneidet das normale XCS am besten ab. Die Accuracy von *normal* ist mit 0,98 genauso gut wie beim 6 Bit Multiplexer. Allerdings sind die anderen Implementierung mindestes genauso gut, wie ihre entsprechende Ausführung auf dem 6 Bit Multiplexer. Das liegt bei den Inverse-Variance-Mixing-Alternativen wahrscheinlich an der größeren Menge an Trainingsbeispielen, die unabhängig von der Anzahl der Trainingsiterationen durchlaufen werden.

Der TTest erkennt hier Unterschiede zwischen allen Verteilungen (siehe Abb. 4)

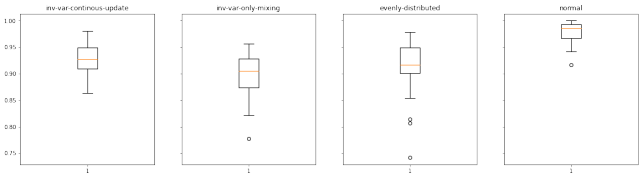

Abbildung 3: Accuracy Werte des 11 Bit Multiplexers

4.3 20 Bit Multiplexer

In Abbildung 5 ist zu sehen, dass die Accuracy aller Implementierungen nicht das Niveau der kleineren Multiplexern erreicht. Bei längerem Training hätte sich die Performance wohl noch verbessert, allerdings war beim 20 Bit Multiplexer schon die Grenze erreicht, bei der sich das kontinuierliche Update noch in vernünftiger Laufzeit ausführen ließ.

	inv-var-continous-update	inv-var-only-mixing	evenly-distributed	normal
inv-var-continous-update	0.000000	0.002321	0.014073	0.000045
inv-var-only-mixing	0.000000	0.000000	0.011643	0.000222
evenly-distributed	0.000000	0.000000	0.000000	0.000037
normal	0.000000	0.000000	0.000000	0.000000

Abbildung 4: 5x2 Cross-validation paired ttest auf dem 11 Bit Multiplexer

Das liegt vor allem an der großen Anzahl der Trainingsbeispiele (2000), die bei jeder Iteration für jeden Classifier komplett durchlaufen werden müssen. Wieder schneidet *normal* am besten ab: 0.89. Die anderen Implementierungen liegen mit 0,84 - 0,85 knapp dahinter. Der Vorsprung von *normal* ist nicht so groß wie bei den 6 und 11 Bit Multiplexern. *evenly-distributed* ist wieder fast so gut, wie *continuous-update* und *only-mixing*. Der TTest bestätigt diese Eindrücke teilweise: Zwischen *only-mixing* und *continuous-update* besteht kein signifikanter Unterschied. Er besagt allerdings auch, dass *normal* und *continuous-update* keine unterschiedlichen Verteilungen besitzen. Da die Laufzeit von *continuous-update* aber wie bereits beschrieben derart schlecht ist, ist *normal* trotzdem deutlich zu bevorzugen.

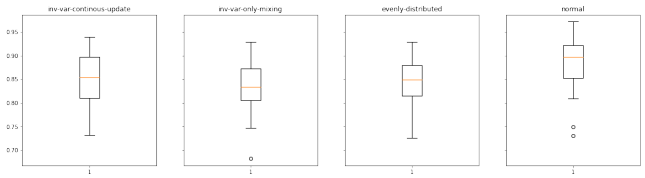

Abbildung 5: Accuracy Werte des 20 Bit Multiplexers

	inv-var-continous-update	inv-var-only-mixing	evenly-distributed	normal
inv-var-continous-update	0.000000	0.254030	0.000533	0.139650
inv-var-only-mixing	0.000000	0.000000	0.000194	0.007966
evenly-distributed	0.000000	0.000000	0.000000	0.001319
normal	0.000000	0.000000	0.000000	0.000000

Abbildung 6: 5x2 Cross-validation paired ttest auf dem 20 Bit Multiplexer

5 FAZIT

Die hier beschrieben Ergebnisse zeigen, dass das Inverse Variance Mixing, wie es hier implementiert ist, sicher nicht

als sinnvolle Alternative zum gewöhnlichen XCS-Mixing benutzt werden kann. In dieser Implementierung wurde allerdings notwendigerweise stark von der eigentlichen vorgeschlagenen Heuristik abgewichen. Welche der Abweichungen zu den schlechten Resultaten führt ist schwer zu sagen. Es handelte sich hierbei eher um einen praktischen Versuch und die formale Korrektheit wurde dabei vernachlässigt. Dies war aufgrund der Komplexität des Themas und der beschränkten Zeit notwendig. Es kann also sein, dass es durchaus sinnvolle Implementierung von Inverse Variance Mixing in XCS gibt, v.a. da hier nur Klassifikation betrachtet wurde. Auffällig ist, dass das Mixing mit gleicher Verteilung der Classifier relativ gut abschneidet. Dies deutet darauf hin, dass das XCS Mixing nicht optimal ist. Weiter Mixing-Modelle sollten daher ausprobiert werden.

LITERATUR

[1] Jan Drugowitsch. 2008. *Design and Analysis of Learning Classifier Systems - A Probabilistic Approach*. Vol. 139. https://doi.org/10.1007/978-3-540-79866-8

[2] Muhammad Iqbal, Will Browne, and Mengjie Zhang. 2013. Learning complex, overlapping and niche imbalance Boolean problems using XCS-based classifier systems. *Evolutionary Intelligence* 6 (11 2013). https://doi.org/10.1007/s12065-013-0091-1

[3] UrblsLab. 2020. scikit-XCS. *Github* (2020). https://github.com/UrbsLab/scikit-XCS

[4] Stewart W. Wilson. 1995. Classifier Fitness Based on Accuracy. *Evol. Comput.* 3, 2 (June 1995), 149–175. https://doi.org/10.1162/evco.1995.3.2.149