Computational Intelligence

M. Sc. Andreas Buschermöhle, M. Sc. Jan H. Schoenke

Übungsblatt 3 - Evolutionäre Algorithmen III: Genetische Programmierung

Abgabe bis Mittwoch, 28.05.2014, 12:00 Uhr

In diesem Übungsblatt werden Algorithmen in Matlab geschrieben und zur Auswertung ein- und zweidimensionale Plots angefertigt. Erstellen Sie im Dateiordner Ihrer Gruppe in Stud.IP einen Ordner mit dem Namen `Blatt 3` und laden Sie zu zweit sowohl die m-Files, als auch die fertigen Matlab-Figures mit sinnvoller Achsenbeschriftung rechtzeitig in diesen Ordner hoch. Geben Sie in Vips je nach Aufgabenstellung die textuelle Interpretationen Ihrer Ergebnisse sowie ggf. die Namen der Dateien an, die sich auf die jeweilige Aufgabe beziehen.

Es soll die Genetische Programmierung in Matlab als Optimierungsverfahren in einem Framework umgesetzt werden und dieses Framework für zwei verschiedene Beispiele verwendet werden. Für die Umsetzung des Frameworks sind in Stud.IP verschiedene Dateien vorgegeben, die den Umgang mit Bäumen als Datenstruktur in Matlab vereinfachen.

- `tree = treeRandGenDeep(maxDepth, descPropab, nrOp, nrTerm)`

  Diese Funktion erzeugt einen binären Baum mit einer maximalen Tiefe von `maxDepth`. Die Variable `descPropab` gibt die Wahrscheinlichkeit für den weiteren Abstieg bei der Erzeugung des Baumes an. Wird `descPropab` $= 1$ gewählt, so wird immer ein vollständiger Baum erzeugt, für `descPropab` < 1 können vorzeitig Blätter entstehen. Die Initialisierung der Knoten bzw. Blätter erfolgt mit Zufallszahlen aus der Menge $\{1, ..., \texttt{nrOp}\}$ bzw. $\{1, ..., \texttt{nrTerm}\}$, wobei `nrOp` bzw. `nrTerm` die Anzahl der Operatoren bzw. Terminalsymbole angibt.

- `[tree left] = treeRemove(tree, node)`

  Diese Funktion entfernt einen Teilbaum aus einem Baum. Hierzu wird der zu bearbeitende Baum in `tree` und der Knoten, der entfernt werden soll, in `node` übergeben. Der bearbeitete Baum wird im Rückgabewert `tree` übergeben. Das Flag `left` zeigt an ob der entfernte Knoten `node` das linke (`left = true`) oder rechte (`left = false`) Kind des Elternknoten war.

- `tree = treeAppend(tree, parentNode, subtree, left)`

  Diese Funktion hängt einen Teilbaum an einen bestehenden Knoten an. Hierzu wird der zu bearbeitende Baum in `tree` und der Knoten, an den der Teilbaum angehängt werden soll, in `parentNode` sowie der einzufügende Teilbaum in `subtree` übergeben. Über das Flag `left` kann bestimmt werden ob der einzufügende Teilbaum `subtree` auf der linken (`left = true`) oder rechten (`left = false`) Seite des Elternknoten `parentNode` angehängt wird.

- `subtree = treeGetSub(tree, node)`

  Diese Funktion liefert einen Teilbaum aus einem Baum. Hierzu wird der zu Grund liegende Baum in `tree` und der Knoten, dessen Teilbaum gesucht ist, in `node` übergeben.

- `treeShow(tree, nodeName, leafName, dx, dy)`

  Mit dieser Funktion lässt sich ein Baum anzeigen. Der Baum wird im Parameter `tree` übergeben und die Beschreibung der Operatoren und Terminal-Symbole in `nodeName` und `leafName`. Über die beiden optionalen Parameter `dx` und `dy` wird der Abstand des beschreibenden Textes zum jeweiligen Knoten angegeben.

Aufgabe 3.1: Framework für Gen. Programmierung (10+5+5+10 = 30 P)

Für die Umsetzung des Frameworks greifen Sie zum Einen auf die in Stud.IP vorgegebenen Funktionen zurück, zum Anderen entwickeln Sie selbst Funktionen für die Mutation, Rekombination und Selektion und verbinden all diese zu einem Optimierungsverfahren.

a) Schreiben Sie eine Funktion, die den Mutationsoperator für die *Teilbaum-Mutation* umsetzt. Verwenden Sie dazu folgenden Funktionskopf:

```
tree = treeMutate(tree,mutateProb,maxMutateDepth,descprobab, nrOp, nrTerm)
```

b) Schreiben Sie eine Funktion, die den Rekombinationsoperator durch *Austausch von Teilbäumen* umsetzt. Verwenden Sie dazu folgenden Funktionskopf:

```
[child1 child2] = treeCrossover(parent1,parent2)
```

c) Schreiben Sie eine Funktion, die aus einer gegebenen Population (Wald) eine neue Population erstellt. Dabei wird jedes Individuum der neuen Population entweder durch Mutation oder durch Rekombination erstellt. Die Selektion der Eltern erfolgt dabei immer Fitness-proportional. Für jedes Individuum der neuen Population wird zufällig entschieden ob es durch Mutation oder durch Rekombination entsteht. Da bei der Rekombination immer zwei Nachkommen entstehen und diese beide aufgenommen werden, können durch Rekombination nur dann Nachkommen erzeugt werden, wenn in der neuen Population noch zwei oder mehr Individuen fehlen. Verwenden Sie folgenden Funktionskopf:

```
newForest = treeNextGeneration(forest,fitness,mutateCrossoverProb,
                               mutateProb,maxMutateDepth,descprobab,
                               nrOp, nrTerm)
```

d) Verbinden Sie nun die einzelnen Funktionen zu einem Optimierungsverfahren, welches auf Genetischer Programmierung beruht. Gehen Sie dabei davon aus, dass Sie die Fitnessfunktion als `function_handle` übergeben bekommen und dass die Fitnessfunktion direkt mit der aktuellen Population (Wald) aufgerufen werden kann und die Fitness jedes einzelnen Individuums (Baum) zurückgibt. Der Wald besteht im einfachsten Fall aus einem Cell-Array von Bäumen. Der Zugriff auf Elemente in einem Cell-Array erfolgt über geschweifte Klammern `{}`. Jeder Baum soll zur Initialisierung mit einer festen Maximaltiefe erstellt werden und hat dabei eine feste Wahrscheinlichkeit für den Abstieg. Nutzen Sie eine *Hall of Fame*! Verwenden Sie folgenden Funktionskopf:

```
function [bestInd ...] = gpOpt(nrTrees, nrGen, fitnessFkt, nrOp, nrTerm,
                               mutateCrossoverProb, maxStartDepth,
                               mutateProb, maxMutateDepth, descprobab)
```

- `nrTrees`: Anzahl der Bäume im Wald.
- `nrGen`: Anzahl der Generationen, die evolviert werden.
- `fitnessFkt`: Die zu maximierende Fitnessfunktion als `function_handle`.
- `nrOp`: Anzahl der Operatoren.
- `nrTerm`: Anzahl der Terminalsymbole.
- `mutateCrossoverProb`: Wahrscheinlichkeit für die Erzeugung von Nachkommen durch Mutation.
- `maxStartDepth`: Maximale Tiefe des initialen Waldes.
- `mutateProb`: Mutationswahrscheinlichkeit jedes Knoten bzw. Blattes bei der Mutation.
- `maxMutateDepth`: Maximale Tiefe des zufälligen Teilbaumes bei der Mutation.
- `descprobab`: Abstiegswahrscheinlichkeit des zufälligen Teilbaumes bei der Mutation.

Aufgabe 3.2: Symbolische Regression (5 + 5 + 10 + 10 + 10 = 40 P)

In dieser Aufgabe wird das Framework für Genetische Programmierung eingesetzt, um auf einem Datensatz symbolische Regression durchzuführen. Der Datensatz besteht dabei aus Paaren von Eingabe- und Ausgabewerten $(\vec{x}_i, y_i)$. Ziel der symbolischen Regression ist es, die Zusammenhänge zwischen den Ein- und Ausgabevariablen $(\vec{x}, y)$ in dem Datensatz durch eine (möglichst einfache) Funktion $f(\vec{x})$ zu erklären, sodass gilt: $f(\vec{x}_i) = y_i \forall i$. Ist dies für alle im Datensatz enthaltenen Paare der Fall, kann (unter Umständen) davon ausgegangen werden, dass die Funktion f allgemein den Zusammenhang zwischen den Eingabe- und Ausgabevariablen beschreibt: $f(\vec{x}) = y$.
Die Funktion f wird dabei durch einen Baum repräsentiert. Im Rahmen dieser Aufgabe beschränken wir uns auf binäre Bäume. Die Knoten eines solchen Baumes stehen dabei für Operatoren wie zum Beispiel: $\{+, -, \cdot, /\}$, die Blätter enthalten als Terminalsymbole die Eingabevariablen $\{x_1, ..., x_n\}$ und einige Konstanten wie zum Beispiel $\{1, 0, \pi, -1\}$. Durch die Kombination der verschiedenen einfachen Operatoren in der Baumstruktur lassen sich auch komplexere Funktionen darstellen.
Im Verlauf der symbolischen Regression wird ein Baum evolviert, der eine Funktion repräsentiert, welche die Daten möglichst gut beschreibt. Die Güte der Beschreibung der Daten wird am Fehler zwischen den im Datensatz vorliegenden Ausgabedaten y_i und den von der Funktion bestimmten Schätzungen $f(\vec{x}_i)$ bemessen. Im Rahmen dieser Aufgabe soll dazu der *Root Mean Squared Error*(RMSE) verwendet werden:

$$RMSE(\vec{x}, y, f) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(y_i - f(\vec{x}_i))^2}. \tag{1}$$

a) Schreiben Sie eine Fitnessfunktion, die den RMSE für jeden Baum eines Waldes bezüglich eines Datensatzes bestimmt und auf eine Fitness abbildet. Verwenden Sie für die Abbildung des RMSE auf die Fitness folgende Funktion:

$$fit = 1/(1 + RMSE). \tag{2}$$

Mittels der Funktion `tree2fun(tree, ops, terms)` (siehe Stud.IP) können Sie eine durch einen Baum `tree` repräsentierte Funktion in ein `function_handle` umwandeln. Das erzeugte `function_handle` nimmt nur einen Aufrufparameter `x` entgegen. In den Beschreibungen der Operatoren `ops` wird ein Leerzeichen für jeden der beiden zu verarbeitenden Operanden erwartet. Die Beschreibung der Terminalsymbole kann beliebige Konstanten enthalten. Bei Funktionen in mehreren Variablen müssen die einzelnen Eingabevariablen $\{x_1, ..., x_n\}$ in den Terminalsymbolen durch `x(1,:)`,...,`x(n,:)` dargestellt werden. Verwenden Sie folgenden Funktionskopf:

```
fit = evalFitSymReg(forest,dataX,dataY,ops,terms)
```

- `forest`: Der zu bewertende Wald.
- `dataX`: Eingabewerte: $(n \times m)$-Matrix.
- `dataY`: Ausgabewerte: $(1 \times m)$-Matrix.
- `ops`: Beschreibung der Menge der Operatoren.
- `terms`: Beschreibung der Menge der Terminalsymbole.

b) Verwenden Sie das Framework (aus Aufgabe 3.1) zusammen mit der in Teil a) geschriebenen Fitnessfunktion, um die Funktion

$$f(x) = x^3 + x^2 + x + 1 \tag{3}$$

mit symbolischer Regression aus einem Datensatz zu extrahieren. Verwenden Sie als Datensatz die folgenden Eingabewerte (`dataX`) und Ausgabewerte (`dataY`):

```
nrData = 1001;
dataX = linspace(-10,10,nrData);
dataY = dataX.^3 + dataX.^2 + dataX + 1;
```

Nutzen Sie als Operatoren `ops` und Terminalsymbole `terms` die folgenden Beschreibungen:

- `ops = {'( + )','( - )',' .* ',' ./ '};`
- `terms = {'1','0','-1','x(1,:)'};`

Verwenden Sie für die weiteren Parameter des GP die folgende Belegung:

- `nrTrees = 1000`
- `nrGen = 20`
- `mutateCrossoverProb = 0.5`
- `maxStartDepth = 4`
- `mutateProb = 0.1`
- `maxMutateDepth = 3`
- `descprobab = 0.2`

Mit folgendem Matlab-Befehl können Sie in einem lokalen Kontext, der die Variablen `dataX`, `dataY`, `ops` und `terms` enthält, die Fitnessfunktion aus Aufgabenteil a) in das korrekte Format für das Framework aus Aufgabe 3.1 umwandeln:

```
fit = @(forest) evalFitSymReg(forest,dataX,dataY,ops,terms);
```

Schreiben Sie nun ein kurzes Skript, welches die notwendigen Parametereinstellungen vornimmt und führen Sie die symbolische Regression durch. Speichern Sie zur Analyse der Regression in jeder Generation die Fitness des besten und schlechtesten Individuums und den Durchschnitt der Population, speichern Sie zusätzlich für jede Generation das beste bisher gefundene Individuum und dessen Fitness (*Hall of Fame*). Da sich die Länge des Genoms der Bäume im Verlauf der Regression verändern kann, speichern Sie zusätzlich die mittlere Größe (Anzahl Konten und Blätter; NICHT die Tiefe(!)) der Bäume jeder Generation. Fügen Sie diese Werte der Liste von Rückgabewerten der Funktion `gpOpt` hinzu. Stellen Sie in einem 2×2-Subplot die Fitness im Verlauf der Generationen, die mittlere Größe im Verlauf der Generationen, das beste gefundene Individuum und das Ergebnis der Regression im Vergleich zu den Ein- und Ausgabedaten dar.

c) Diskutieren Sie die Wahl der Abbildung in Gleichung (2) vor dem Hintergrund des Effekts des *Surival of the fattest*. Um dem Effekt des *Surival Of The Fattest* entgegenzuwirken soll die Größe des Baumes antiproportional auf die Fitness wirken. Erweitern Sie die Fitnessfunktion aus Aufgabenteil a) entsprechend und wiederholen Sie die Untersuchungen aus Aufgabenteil b). Plotten Sie Ihre Ergebnisse wie in Aufgabenteil b), lassen sich Veränderungen erkennen? Interpretieren Sie Ihre Ergebnisse!

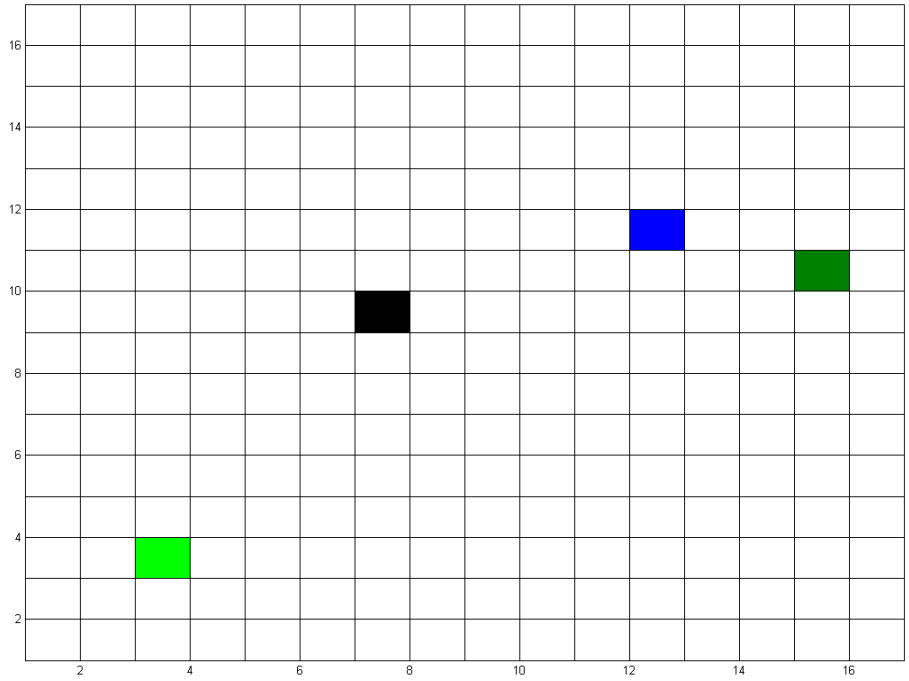
d) Bisher waren die Ein- und Ausgabedaten für die symbolische Regression perfekt, d.h. ohne Störeinflüsse wie Rauschen oder Ausreißer. Im Folgenden wird untersucht wie robust die symbolische Regression aus Aufgabenteil b) und c) gegenüber verrauschten Ausgabedaten ist. Dazu werden die Daten `dataX` und `dataY` zunächst wie in Aufgabenteil b) erzeugt. Anschließend werden die Ausgabedaten `dataY` mit einem normalverteilten Rauschen gestört. Das Rauschniveau soll 10% des Wertebereichs von `dataY` betragen (`randn` skalieren). Wiederholen Sie die Untersuchungen der Aufgabenteile b) und c). Da der Einfluss des Rauschens dem Zufall unterliegt, wählen Sie einige repräsentative Ergebnisse aus und diskutieren Sie diese. Unterscheiden sich die Ergebnisse für die Fitnessfunktionen der Aufgabenteile b) und c)?

e) Ein wichtiger Effekt bei der Regression auf verrauschten Daten ist das sogenannte *Overfitting*. Beim *Overfitting* wird die Ausdrucksstärke des Regressionsverfahrens nicht mehr allein dafür verwendet den grundsätzlichen Zusammenhang zwischen den Daten zu beschreiben. Vielmehr wird das Ergebnis der Regression auch an den Einfluss des Rauschens angepasst. Dadurch wird der Fehler auf den Trainingsdaten kleiner, da der Effekt des Rauschens kompensiert wird. Insgesamt ist das Ergebnis jedoch schlechter, da Effekte von der Regression modelliert werden, die in den zugrundeliegenden Daten nicht vorhanden sind, sondern nur durch das Rauschen induziert werden.

Um diesen Effekt beobachten zu können, wird in diesem Aufgabenteil die Menge der Daten im Datensatz reduziert. Für die Eingabedaten `dataX` werden zufällig gleichverteilt elf Zahlen aus dem Intervall $[-10, 10]$ gezogen. Für die Ausgabedaten `dataY` werden zunächst die korrekten und unverrauschten Werte entsprechend der Daten in `dataX` bestimmt und diese anschließend mit dem gleichen Rauschen wie in Aufgabenteil d) gestört.

Wiederholen Sie die Untersuchungen der Aufgabenteile b) und c). Da der Einfluss des Rauschens und die Wahl der Datenpunkte in `dataX` dem Zufall unterliegt, wählen Sie einige repräsentative Ergebnisse aus und diskutieren Sie diese. Welche Eigenschaften des hier verwendeten Genetischen Programms wirken dem Effekt des *Overfitting* entgegen?

Aufgabe 3.3: Postkutsche (10 + 10 + 10 = 30 P)



In dieser Aufgabe soll durch genetische Programmierung das Verhalten einer Postkutsche optimiert werden. Im Wilden Westen muss eine Postkutsche von ihrer Heimatstadt in die Nachbarstadt fahren, um dort Post abzuholen. Diese muss sie dann zurück in ihre Heimatstadt bringen. Unterwegs lauert jedoch ein Bandit, der die Kutsche überfällt, wenn er ihr begegnet, und die Post stiehlt. Daher sollte die Kutsche auf dem Rückweg dem Banditen ausweichen. Jedoch sollte die Kutsche auch keinen zu langen Weg wählen, da sie nur begrenzte Nahrungsmittelvorräte mitführt und der Kutscher irgendwann verhungert.
In der Simulation ist die Landschaft durch ein 16×16 Feld beschrieben (siehe Grafik). Auf diesem befindet sich an einem festgelegten Ort die Heimatstadt (Hellgrün) und die Nachbarstadt (Dunkelgrün). Die Postkutsche (Blau/Gelb mit Post) kann sich in der Landschaft in jedem Zug um ein Feld vertikal oder horizontal bewegen. Der Bandit (Schwarz) bewegt sich zwischen zwei festen Punkten auf und ab, entscheidet jedoch in jedem Schritt zufällig, ob er stehenbleibt oder weitergeht. Nach 100 Schritten sind die Nahrungsmittelvorräte aufgebraucht und die Kutsche kann nicht weiterfahren.
Um eine Entscheidung zu treffen, wie sich die Kutsche in einer Situation verhalten soll, lassen sich Entscheidungsbäume (Decision Trees) verwenden. Hier stehen die Entscheidungsmöglichkeiten *Ist der Räuber nördlich / östlich / südlich / westlich von mir?*, *Ist das Ziel nördlich / östlich / südlich / westlich von mir?* und *Habe ich Post bei mir?* zur Verfügung. Diese las-

sen sich durch ganze Zahlen repräsentieren. Am Ende wird dann die Entscheidung getroffen, ob man sich nach Norden, Osten, Süden oder Westen bewegt, was wiederum durch ganze Zahlen repräsentiert werden kann. Ein solcher Entscheidungsbaum lässt sich nun von Hand erstellen oder aber durch ein Optimierungsverfahren auf ein Problem anpassen. Hier soll Genetische Programmierung verwendet werden, um einen guten Entscheidungsbaum zu finden. Für die Simulation der Postkutsche in Matlab steht folgende Funktion in Stud.IP bereits zur Verfügung:

```
fitness = playRound(genome, show).
```

Diese Funktion spielt eine Runde des Westernszenarios mit dem Kutschverhalten basierend auf dem übergebenen Genom. Über den Parameter `show` lässt sich einstellen, ob die Runde visualisiert werden soll. Als Rückgabewert wird die Fitness des Genoms geliefert.

a) Betten Sie die Funktion `playRound` in eine Fitnessfunktion für das Framework aus Aufgabe 3.1 ein und optimieren Sie das Verhalten der Kutsche. Zur Beschreibung der Menge der Operatoren und Terminalsymbole könne Sie folgende Darstellung verwenden:

   - `ops = { 'hasPost'; 'RN'; 'RE'; 'RS'; 'RW'; 'TN'; 'TE'; 'TS'; 'TW'}`
   - `terms = {'N'; 'S'; 'E'; 'W'}`

   Speichern Sie einen resultierenden Entscheidungsbaum. Welche Parameter haben Sie verwendet, um sicher ein Verhalten der Kutsche zu bekommen, dass die Post abholt?

b) Erklären Sie, warum die Fitnessfunktion so gewählt wurde, wie in der Funktion `playRound` zu sehen? Haben Sie Verbesserungsvorschläge, um die Kutsche noch zielgerichteter zu evolvieren? Wie können Sie das Problem des *Surival of the fattest* lösen?

   Untersuchen Sie den Einfluss Ihrer Verbesserungen an der Fitnessfunktion und stellen Sie Ihre Ergebnisse dar!

c) Betrachten Sie verschiedene gut bewertete Endergebnisse Ihrer Evolution. Bildet sich ein bestimmtes Muster in diesen Bäumen heraus und lässt sich der Aufbau der Bäume interpretieren?