Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnikai és Informatikai Rendszerek Tanszék

Gépi szabálytanulás alkalmazása biomarkerek felismerésére biológiai hálózatokban

Készítette
Nyéki Hunor

Konzulens
Dr. Szalay Kristóf Zsolt

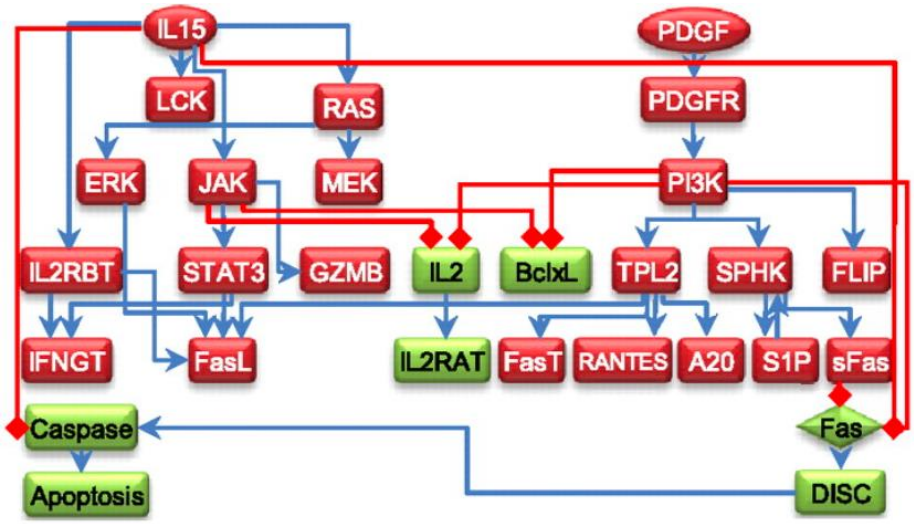
2018

TARTALOMJEGYZÉK

1. Bevezetés

A XXI. század orvosbiológiai kutatásainak kétségkívül egyik legizgalmasabb területe az élő sejtek pontosabb működésének megismerése érdekében létrehozott **biológiai hálózatok** kutatása. Ez az újfajta megközelítés lehetővé teszi, hogy a sejtekben lejátszódó folyamatokat ne külön, a nagy egészből kiragadva értelmezzük, hanem azokat rendszerszinten is értelmezni tudjuk.

Biológiai hálózatokra vegyük példának az **emberi sejtekben** jelen lévő fehérjék és köztük lévő interakciók kapcsolatát **leíró gráfot**, ahol a csúcspontok az egyes fehérjéknek felelnek meg, az élek pedig a fehérjék közt létrejövő kölcsönhatásoknak.



1. ábra: Egy kisebb biológiai hálózatot leíró gráf [1]

Elméletben, míg egy ilyen gráf konstrukciója lehetővé tenné a sejtes folyamatok pontos szimulációját, a gyakorlatban jórészt kivitelezhetetlen, mivel nem ismerjük az összes sejten belül lejátszódó pontos mechanizmust. Azonban míg egy átfogó, 100%-ig pontos modell felépítésére jelenleg nincs lehetőség, bizonyos részfolyamatokat, viselkedéseket továbbra is fontos kutatni és modellezni.

Egy gyakorlatias egészségügyi alkalmazásként felmerül, hogy kisebb biológiai hálózatokat építsünk azzal a céllal, hogy megértsük **különböző rákos megbetegedéseknek** a kialakulását, esetleg azoknak a reszponzivitását prediktálni tudjuk célzott terápiás kezelésekre. Ezeket a gyógyszereket a modellünkben tudjuk úgy értelmezni, hogy azok egy vagy több csúcspontot aktiválnak vagy nyomnak el.

A gyógyszerek fejlesztése során két eset van, ahol a fentieknek hozzáadott értéke lehet. Az egyik eset az, amikor az érdekel minket, hogy egy adott gyógyszerre milyen módon tud esetleg később **mutálódással egy rákos sejt rezisztenssé** válni. A másik eset, amikor meglévő gyógyszerek esetén kíváncsiak vagyunk arra, hogy azok hatásosságát tudjuk-e még fokozni. Ebben az esetben a gyógyszer nem öli meg elég nagy mértékben a megbetegedett sejteket, de lehet, ha a meglévők mellett még egyéb fehérjékre is célzott gyógyszerrel hatnánk, megnövelné a hatásfokot, a sejt **szenzitívebb** lenne a kezelésre. A szenzitív és rezisztens fehérje kombinációkra összefoglaló néven hivatkozhatunk **biomarkerekként**.

1.1. Biomarker keresés számítástechnikai akadályai

A bevezetésben ismertetett alkalmazáslehetőség kecsegtető, de ha elkezdünk utána számolni a számítási költségeinek annak, hogy mennyi időbe telik egy hálózatban különböző hosszúságú biomarkereket keresni, borúsabb képet kapunk.

A szimulációk elvégzéséhez a **Turbine Kft. hálózatszimulációs programját** fogom használni a továbbiakban. Egy szimuláció ebben a rendszerben, egy adott kisebb méretű hálózat (50-100 csúcs) kiértékelése során jobb esetben körülbelül **0.5 másodpercet** vesz igénybe egy manapság átlagosnak mondható teljesítményű PC-n, számoljunk a továbbiakban ezzel.

Egy adott hálózatban a szimulálandó kombinációk száma több dologtól függ. Az első kérdés, ami felvetődik, hogy egyáltalán **milyen mélységig** van értelme szimulációkat végezni, azaz milyen hosszú kombinációk érdekelhetnek minket. Másrészt a lehetőségek számát nagyban befolyásolja a **hálózat mérete is**. Rákos sejtekkel és biológiai hálózatok építésével foglalkozó cikkek alapján változatos méretű hálózatokat találunk ([1], [2], [3], [4], [5][6]) , az egyszerűség kedvéért a későbbi számításoknál **válasszuk optimistán 100-at tipikus értéknek**.

Megjegyzés: Az átlagos hálózatméretnek választott érték különösen alacsony annak fényében, hogy jelenleg ismereteink alapján 20-25 ezer a becsült száma a fehérje kódoló géneknek, ezekből elméletben jóval nagyobb méretű hálókat lehet építeni. [7]

Az **egy hosszúságú kombinációk számát** az alábbi képlet adja, ahol ’k’ a kombinációk számát, ’n’ a hálózatban lévő csúcspontok száma:

$$k_1 = n * 2$$

A kettes szorzó oka, hogy **egy fehérje egy biomarkerben szerepelhetnek aktív** (bekapcsolt, gain of function) **és gátolt** (kikapcsolt, loss of function) állapotban.

Tetszőleges hosszúságú biomarkerek számát az alábbi képlet segít meghatározni, ahol L a kombinációk hosszát jelenti:

$$k_L = \binom{n}{L} * 2^L$$

Szemléltetés céljából töltsünk ki egy táblázatot, hogy lássuk, a kombinációk számának növelésével milyen mértékben változik a szimulációval töltött futási idő egy átlagos hálózat esetén:

Kombináció hossz	Kombinációk száma	Futási idő
1	200	100 másodperc
2	19800	165 perc
3	161700	179.7 óra
4	62739600	363.04 nap
5	2409200640	38 év

A fentiek alapján láthatjuk, hogy még egy viszonylag **kisebb hálózat esetén is hamar tarthatatlanul elszáll a futási idő**, ha egy adott gyógyszerhez komplex biomarkereket akarunk találni egy hálózatban.

1.2. Gépi tanulás a szimulációk számának csökkentésére

Nagy méretű hálózatoknál megtalálni komplex biomarkereket nyers erővel számításigényes feladat. Különböző kompromisszumok meghozatalával azonban ez az erőforrás igény jelentősen mérsékelhető.

A nyers erővel történő megoldás olyan, mintha tűt próbálnánk keresni a szénakazalban. Biológiai hálózatokban azonban **megfigyelhetők különböző szabályszerűségek**, ami lehetőséget ad egyszerűsítésekre. Szenzitivitásra vagy rezisztenciára utaló biomarkerek esetén egy hálózatból kinyerhető **élettani paraméterek közül** azok lesznek érdekesek, amik segítenek eldönteni, hogy egy adott sejt a rárakott gyógyszerrel és egyéb mutációkkal, amiket ép vizsgálunk **élőnek minősül**, **vagy sem**.

Mivel elsősorban **komplex biomarkereket** keresünk (egy hosszúságú kombinációk keresése még jól kezelhető lenne nyers erővel is), tételezzük fel, hogy a vizsgált hálózatunkban létezik ilyen. Ez esetben tudhatjuk előre, hogy lesznek olyan csúcspontok a hálózatban, amiknek a ki/be kapcsolgatása **önmagában nem fogja** átbillenteni egyik állapotból a másikba a sejtet, **de az egyszerre történő megváltoztatásuk igen**. Ha ez nem így volna, és minden csúcspont döntő hatással lenne a sejt apoptotikusságára, akkor csak egy hosszúságú biomarkereket találhatnánk a hálózatban.

Ha feltesszük tehát, hogy léteznek az általunk vizsgált hálózatban összetett biomarker kombinációk, akkor ezek esetén valami hasonló szabályszerűséget kell látnunk:

Mutációk (mutált csúcs: gain(1)/loss(0) of function)	**Ítélet**
A:0	ALIVE
B:1	ALIVE
A:0+B:1	DEAD

Ha ennél tovább megyünk, létrejöhet az az eset is, hogy az állapottér részleges ismeretéből magas valószínűséggel meg tudjunk jósolni hiányzó szabályszerűségeket:

Mutációk	**Ítélet**
A:0+B:1+C:1	DEAD
A:0+C:1	DEAD
B:1+C:1	ALIVE
C:1	ALIVE

A fenti esetben, ha **szenzitív** biomarkert keresnénk, akkor a legvalószínűbb tippünk **intuitívan az A:0** lenne, hisz az szerepel minden kombinációban, ami megöli a sejtet, és nem szerepel egy olyanban sem, ami mellett a sejt még életben marad. Az adatok önmagukban nem garantálják, hogy a feltételezésünk helyes, de megmutatnak mégis egy esetleges jó jelöltet, amit érdemes lehet megvizsgálnunk.

Adja magát az ötlet ezek alapján, hogy ne szimuláljuk le az összes létező kombinációt, hanem szimuláljuk le inkább azoknak csak egy részhalmazát, és próbáljunk a fenti módszerrel gépi tanulással új szabályokat kinyerni. A generált, ígéretes bizonyosságú szabályokat utólag már gyorsabban le tudjuk ellenőrizni, hogy kiszűrjük a fals találatokat, mintha le kellett volna szimulálnunk az egész állapotteret.

1.3. Célkitűzés

A dolgozatom célja a fentebb vázolt biomarker keresés problémának az előző pontban felvetett módon történő megoldásának elemzése, és a gyakorlatban történő megvalósítása.

A dolgozat során kiválasztok a módszer tesztelésének céljából egy biológiai hálózatot, amin komplex rezisztenciát jelző biomarkereket próbálok kinyerni az összes létező kombináció leszimulálása nélkül. Elemzés tárgyát fogja képezni a megfelelő tanuló eljárás kiválasztása is.

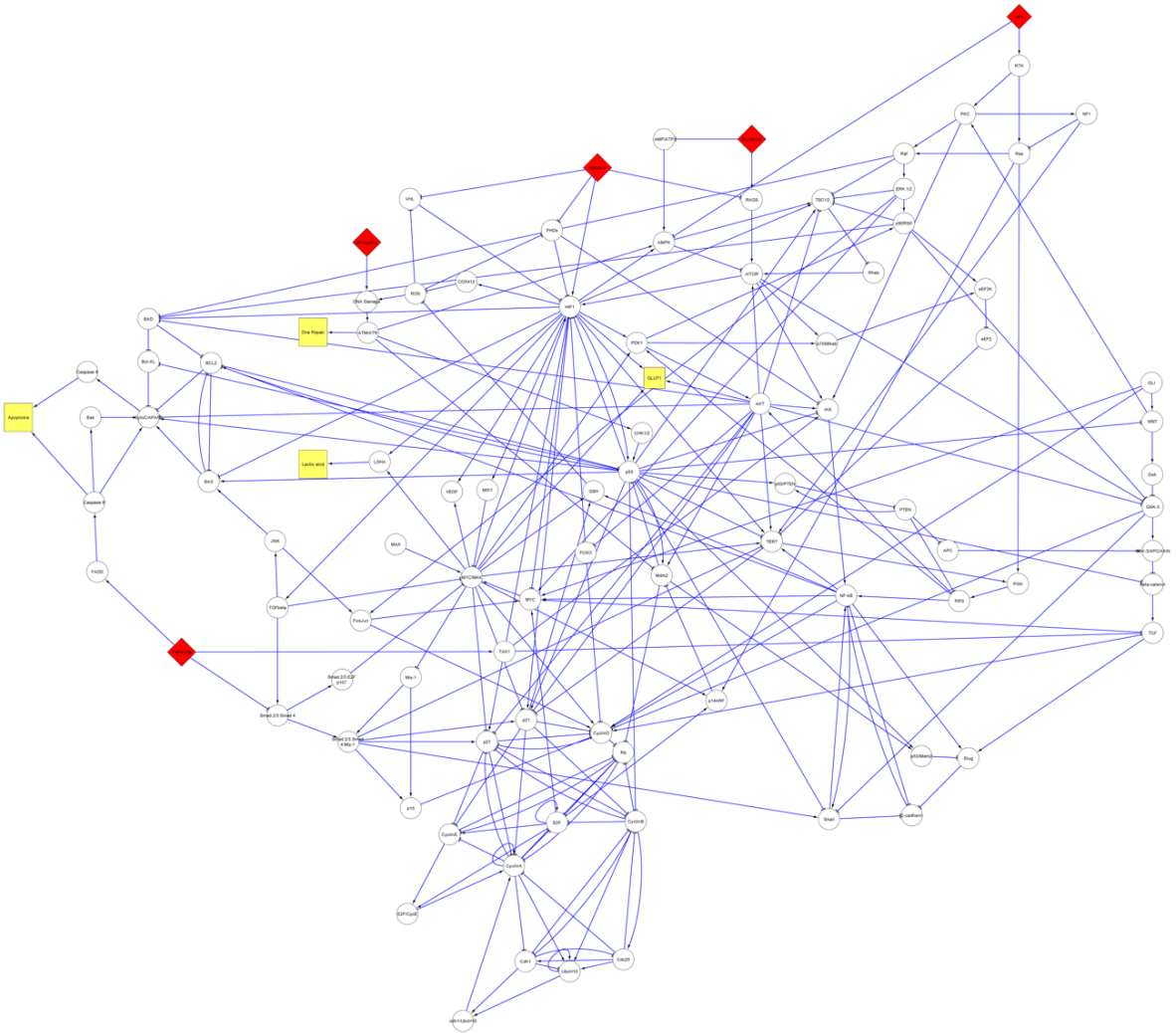
A gyakorlati implementációt a Turbine hálózatszimuláló keretrendszeren belül fogom megvalósítani. A szoftver rendelkezik egy beágyazott Python interpreter-el, amin keresztül a tanuló adatokat generáló script-et valósítom meg. Szintén Python-ban fogom implementálni a választott gépi tanuló eljárást, mivel a nyelv jól használható gyors prototípusok elkészítésére, ami egy ilyen kísérleti jellegű feladatnál előnyös.

A célt elértnek és a módszert sikeresnek tekintem, ha a jó bizonyossággal prediktált szabályok között működő kombinációkat nyer ki a tanuló algoritmus, amik a betanító adatokban explicit nem szerepeltek.

2. Vizsgált biológiai hálózat

Biológiai hálózatok kiválasztása során több különböző opció is felmerült ([1], [2], [3], [4], [5][6]). Abból a célból, hogy a tanító eljárás működésére jobban fókuszálni tudjak, hálózat dinamika szempontjából könnyebben átlátható a **boolean alapú** hálózatok mellett döntöttem. Boolean dinamika alatt azt értjük, hogy a hálózaton belül egy csúcspont csak be vagy kikapcsolt állapotban lehet, az állapot eldöntését pedig egy diszkrét időpillanatban egyéb csúcsoktól függő logikai szabályok adják meg.

A boolean alapú biológiai hálózatok közül különösen ígéretesnek tűnt a *Herman F. Fumiã, Marcelo L. Martins* által publikált *Boolean Network Model for Cancer Pathways: Predicting Carcinogenesis and Targeted Therapy Outcomes* című tanulmányban leközölt hálózat [2]. A cikkben leközölt mesterséges sejtről könnyen megállapítható, hogy apoptotikus-e (azaz épp programozott sejthalál megy végbe) vagy sem, mivel a hálózatban van egy kijelölt csúcspont (apoptosis csúcs), ami pont ezt hivatott jelölni.



2. ábra: Huzalozási diagram a Fumiã, Martins cikkben publikált hálózathoz [2]

A háló mérete a vizsgálatok szempontjából ideálisnak tekinthető, a teljes hálózat közel 100 csúcspontból áll, ami egy olyan méret, ahol már érezhető a nyers erővel történő komplex rezisztens biomarker keresésének a lassúsága.

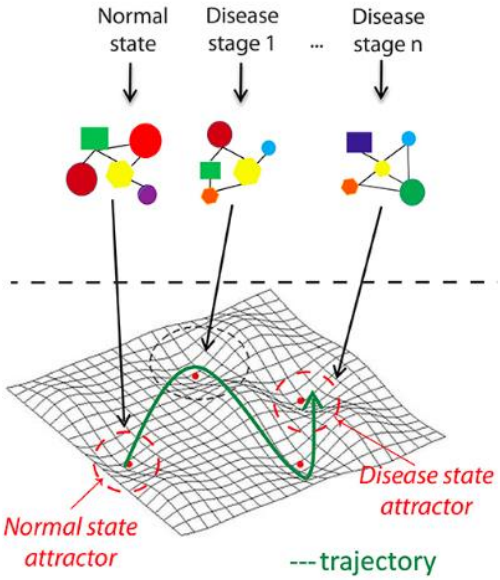
A hálózat rendelkezik különböző **input csúcspontokkal** (az ábrán pirossal jelölt csúcsok), amelyeknek a beállításával a hálózat egy adott életjelenséget mutat. **Kiinduló állapotnak** egy olyan kombinációt érdemes választani, ami esetén a hálózat **életben van**. Én a későbbiekben ennek az eléréséhez alábbi csúcspont értékeket rögzítettem:

Csúcspont	**Állapot**
GF	1
Nutrients	1
Mutagenic	0
TNF_A	0
HYPOXIA	0

Ezután, hogy rezisztenciát vizsgálhassunk a hálózaton, keresnünk kell egy alkalmas **célzott gyógyszert**, ami a sejthálót **apoptotikus állapotba** viszi át. A hálóban a BCL-2 csúcspont kikapcsolásával ez a hatás elérhető, ezért erre a célra a BCL-2 inhibitálására (gátlására) képes gyógyszerhatást modelleztem, azaz a **BCL-2 csúcspontot 0-ra** mutáltam (rögzítettem) minden későbbi szimulációban.

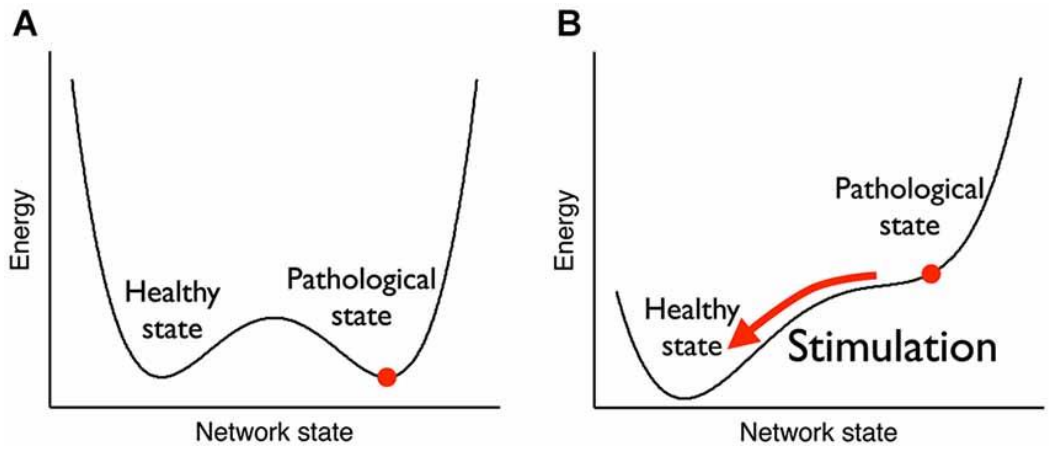
Megjegyzés: ha egy valódi gyógyszerrel szeretnénk elérni ezt a hatást, az Obatoclax nevű, BCL-2 fehérjecsaládot gátló gyógyszer jöhet szóba

A hálózat állapotának kiértékelésekor az általam használt szimulációs program a hálózat **attraktorait** (stabil vagy ciklikus állapotok, amikbe a hálózat a szimuláció során beleragad) adja vissza a hálózat kezdeti bemeneti értékeinek és mutációinak függvényében. Minden attraktor esetén a program megadja azoknak a **méretét** is, azaz azt, hogy a futtatott szimulációk közül hány esetben került a hálózat egyik vagy másik attraktor állapotba, ami korrelál az attraktor mélységével.



3. ábra: Szemléltető ábra, egy hálózat attraktor állapotainak ábrázolása. [9]

A kiértékelés során a hálózatot **élőnek fogom tekinteni**, ha az attraktor állapotba került szimulációk többségében a kimeneti csúcsok közül (az ábrán sárgával jelölt) az **apoptosis csúcs kikapcsolt** állapotban van.



4. ábra: Szemléltető ábra, így tolódhatnak el egy hálózat attraktorai, ha a csúcsok paraméterei megváltoznak. [8]

3. Tanuló eljárás kiválasztása

A problémához megfelelő tanuló algoritmushoz az alábbi elvárásokat fogalmazhatjuk meg:

- Legyen képes szabályszerűségeket felismerni (**modellt alkotni**) a bemeneti adatok alapján, és becsülni azok jóságát
- Az alkotott modell legyen emberi szemmel is **értelmezhető**
- **Kevés előfeltételezéssel** éljen a vizsgált problémával kapcsolatban
- Előnyös, ha **minél kevesebb adatból** képes a tanulásra (ne legyen szükség túl sok szimulációra)

A követelmények közül a legtöbb eljárást kizáró megoldás főként a második. Ha tradicionális **neurális hálózattal** próbálnánk tanítani, a betanulás után nincs lehetőségünk arra, hogy betekintést nyerjünk az alkotott modellbe, **csak élsúlyokat láthatunk**. Nem felel meg az értelmezhetőség bonyolultsága miatt továbbá a legtöbb széles körben használt [11] gépi tanuló eljárás sem (SVM, Bayes háló,stb.).

Előnytelennek bizonyulnak a felügyelt tanulást végző eljárások is, ahol nyers adatokat külön szedjük tanuló és validációs halmazra, hiszen a felbontásban értékes szabályszerűségek veszhetnek el (képzeljük el az 1.2-es fejezetben vett második példánál a tanítást az adathalmaznak csak az első felével) és az elvégzett költséges szimulációk egy része így kárba megy.

A problémákra a legkecsegtetőbb megoldást kizárásos alapon a ’90-es években szárnyra kapó **szabály alapú tanulást** végző algoritmusok (LCS – Learning Classifier Systems) jelentik. Ezek a rendszerek eleget tesznek a fentebbi követelményeknek, limitált előfeltételezésekkel is jól értelmezhető szabályhalmazokat állítanak elő. Tréningelhetők megerősítéses tanulással, így a szimulált adatok egészét fel tudja használni a betanulás folyamatában. Ezek közül az eljárások közül az **XCS** nevű eljárást fogom implementálni, mivel ez már képes minden létrejött szabályhoz egy megbízhatóság alapú fitness értéket is rendelni [10].

4. XCS

Az **XCS** algoritmus a **Learning Classifier System** szabály alapú tanulást végző eljárásainak családjába tartozik. Működés szempontjából egy **tanuló és egy genetikus komponensből áll**.

A tanulás folyamata **iteratív**, a genetikus komponens minden lépésben karbantart egy **szabályokból álló populációt**, ami a rendszer éppen aktuális elképzeléseinek, felismert szabályszerűségeinek a halmaza. Adott körben a meglévő szabályok között szelekció mehet végbe, továbbá meglévő szabályok „kereszteződhetnek” és „mutálódhatnak”.

A populációban lévő szabályok jutalmazását, megerősítését a **tanuló komponens** végzi, véletlenszerűen kiválasztott tanuló adatpontok szerint (tehát gyakorlatilag a tanuló adatok 100%-át felhasználhatja a tanuláshoz). A rendszer bemeneti adatként feltételeket és a feltételek alapján helyesnek minősülő ítéleteket vár.

A tanuló adatpontok összességére a továbbiakban a rendszer **környezeteként** fogunk hivatkozni, az adott iterációban vizsgált pont feltételeit pedig észlelt **szituációnak** nevezzük. Egy, a populációban nyilvántartott szabályra szokás még **osztályozóként** (vagy angolul **classifier** néven) is hivatkozni. Az XCS sajátossága, hogy az osztályozók mellé nyilvántart egy **megbízhatóság alapú fitness pontot** is. [13]

Példa:

Az alábbi példában egy hipotetikus tanuló folyamat egy adott időpillanatát ragadjuk ki. A zárójelben lévő számok a populációs résznél a szabályok jóságára tárolt becslését szimbolizálják a rendszernek.

Tanuló adatok (környezet):

Hideg van és december -> Tél van
Hideg van és június -> Nyár van

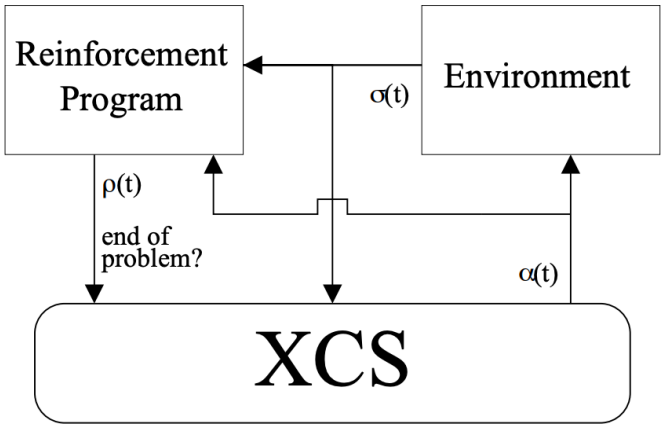
Szituáció adott lépésben:

Hideg van és december

Populáció adott lépésben:

december -> Tél van (1.0)
Hideg van és december -> Tél van (1.0)
Hideg van -> Nyár van (0.5)

Az iteratív tanulás folyamatáról az alábbi ábra ad áttekintést egy adott időpillanatban (t).

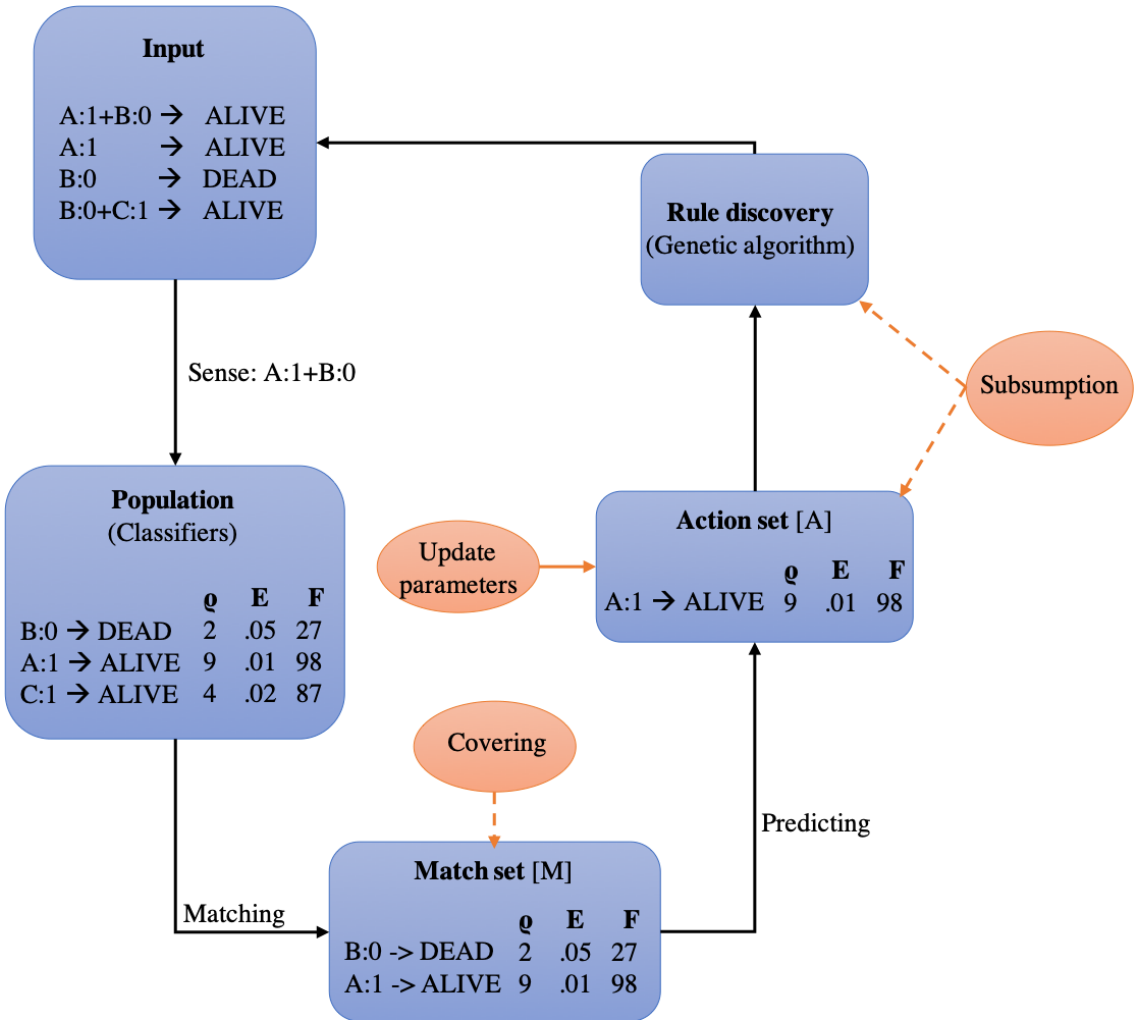


5. ábra: XCS tanulás folyamata [12]

Mint az az ábrán látható, egy iteráció abból áll, hogy véletlenszerűen kiválasztunk, észlelünk egy szituációt (**σ(t)**) a környezetből, amire a jelenlegi tudása alapján az algoritmus (**XCS**) javasolni fog egy ítéletet (**α(t)**). A végrehajtott akció jóságát (**ρ(t)**) ezután a megerősítéses tanuló komponens fogja kiértékelni, ami alapján a tudásbázis frissítésre kerül. Az iterálást folytathatjuk egy megadott számú lépésig, vagy befejezhetjük akkor is, ha már kellően magas megerősítést kap a rendszer adott lépésben.

4.1. Az algoritmus sematikus áttekintése

Ha mélyebb képet akarunk kapni az algoritmus működéséről, ami elkerülhetetlen ahhoz, ha később megfelelően szeretnénk használni, érdemes áttekinteni a rendszert egy picit mélyebben. Egy konkrét folyamatábrát az XCS működéséről felvázolni nehéz, mivel ugyanezen név alatt több kisebb variánsa is létezik az algoritmusnak, azonban az alábbi ábra egy jó kiindulópont lehet a megértéshez:



6. ábra: XCS algoritmus sematikus működése

Az ábrán a könnyebb követhetőség kedvéért szerepelnek különböző minta bemeneti adatok és egy képzelt állapota a rendszernek. A populáció elemei mellett fel vannak tüntetve pontszámok, ezek jelentése a következő alfejezetben bővebben kifejtésre kerül, most elég a megértéshez annyi, hogy ezekből megállapítható egy szabály pontossága.

Az algoritmus haladása folyamán különböző halmazok alakulnak ki. A bemeneti adatokból (környezet) észlelt szituációhoz a populációból (jelölése: [P]) kiválasztódnak szabályok, amelyek fedik az észlelt eshetőséget, ezt nevezzük Match halmaznak (**Match set**, jelölése: [M]). A Match set elemei közül egy predikciós eljárás során kerülnek ki az

Action set (jelölése: [A]) elemei. Ebben a halmazban a Match set azon elemei vannak, amik a legkifizetődőbbnek ítélt akciót javasolják. Az ítélet várható kifizethetőségét a rendszer az [M]-ben lévő szabályok tárolt paraméterei alapján számítja ki.

Az ítélet kiválasztása után [A]-nak az elemeinek a paraméterei frissülnek. A jutalmazás módjának megválasztására többféle lehetőségünk van. Az egyik legegyszerűbb megoldás, hogy jutalmat kap a rendszer, ha a bemeneti adatokban a szituációhoz tartozó ítéletet javasolta, és nem kap jutalmat, ha attól eltért. Ez persze tetszőlegesen implementálható a probléma függvényében.

A paraméterek frissítésének végeztével megkezdődik a **szabálytanulás genetikus része**. Abból a célból, hogy a populációba új szabályok kerüljenek, bizonyos időközönként XCS esetén [A] elemeiből új szabályokat származtat, a valódi genetikához hasonlóan **öröklődéssel** (szülő szabályokból gyerek szabályok származtatása, és azoknak egy részének keresztezése) és **mutáció** (véletlenszerű hiba az öröklődés menetében, például véletlenszerűen törlünk egy feltétel részt) segítségével.

Mind [A]-ban, mind a genetikus algoritmus során alkalmazhatunk **subsumption eljárást**, azaz engedélyezhetjük, hogy az adott ponton a jól prediktáló **általánosabb szabályok magukba olvasztják** a specifikusabb szabályokat. Általánosabbnak akkor értünk egy szabályt, ha az kevesebb feltételből áll, mint a specifikusabb társa, ugyanakkor az általánosban található feltételek megtalálhatók a specifikusabban is, és ezen felül ugyanazt a cselekedetet is javasolják (például: A:1 → ALIVE általánosabb, mint A:1+B:0 → ALIVE, de nem általánosabb, mint A:1+B:0 → DEAD).

Az ábrával kapcsolatban nem esett még szó arról, hogy [P] hogyan fog feltöltődni kezdeti értékekkel. Erre a problémára tipikus megoldás a **covering eljárás**, ami az [M] halmaz létrehozásakor történik. Ha üres a kezdeti populációnk (esetleg a későbbiekben bár nem üres, de mégse tartalmaz elég szabályt, amikből [M] alakulhatna), az **algoritmus automatikusan generál szabályokat**, amivel feltöltheti [M]-et. Ennek a menete, hogy véletlenszerűen hoz létre új szabályokat, figyelve arra, hogy azok bekerülhessenek a Match halmazba. Erre a legegyszerűbb, ha a rendszer kiindulópontként veszi az aktuális lépésben észlelt szituációt, és véletlenszerűen elhagy belőle feltételeket. Az így kapott új szabály biztosan illeszkedni fog a kiinduló pontra. Az ítéletet a generált osztályozóhoz az algoritmus véletlenszerűen választja ki.

Példa:
Új szabály generálása covering során, [M] feltöltéséhez.

Észlelt szituáció:

A:0+B:1+C:0+D:1 →ALIVE

Covering során generált új szabály:

B:1+C:0 →DEAD

4.1.1. A populáció szabályainak pontozása

Az XCS használata során minden egyes iterációs lépésben a populációban lévő **szabályok pontozása folyamatosan frissül**. Egy szabály jóságának eldöntése érdekében több különböző mező is a rendelkezésünkre áll arról, hogy képet kaphassunk mekkora bizonyossága van a rendszernek az általa alkotott összefüggésekben.

Egy **classifier** első sorban így épül fel:

- Egy **feltétel lista**, ami a szabály állapotait tárolja
- **Javasolt ítélet**, amit a feltétel listához kapcsol
- **Becslés** (prediction estimate, **ρ**) az elvárt jutalomra, ha a szabály által javasolt ítéletet választja a rendszer

Ezeken kívül még számos egyéb paramétert tart karban a rendszer, mint:

- A szabály **fitnesz** értéke (**F**)
- A szabály **tapasztalata**, azaz hányszor lett egy action set eleme
- Az **átlagos mérete** az [A]-knak, amikben a classifier szerepelt
- A **predikciós hiba** nagysága (**E**)
- A szabályhoz tartozó **időbélyeg**, azaz hány lépéssel ezelőtt lett létrehozva
- Kisebb szabályok száma, amiket ez a classifier magában foglal

4.1.2. Az algoritmus paraméterei

Az XCS működésé nagy mértékben testre szabható az adott lépések során alkalmazható korlátok, valószínűségi mértékek és egyéb paraméterek testreszabásával [12]. Az alábbiakban a kiemelten fontosokat sorolom fel, amiket a későbbiekben is finomhangolni fogok:

- **N**: A populáció méretének a maximuma. Ha ennél több szabály kerülne a populációba, véletlenszerű törlés (roulette wheel algoritmusos szelekcióval) kezdődik, amíg a kívánt méretet el nem érjük.
- **B**: Tanulási ráta, a szabályok **ρ, E** és **F** paramétereinek a frissítése során van szerepe.
- **α**, **e_0**, és **v**: Classifier fitness számolása közben használt konstansok
- **θ_{GA}**: Megadja, hogy hány iterációnként hajtódik végre az algoritmus genetikus része.
- **χ**: Genetikus algoritmus futása közben a valószínűségét adja meg annak, hogy szabálykereszteződés történjen.
- **μ**: Megadja annak a valószínűségét, hogy egy utód szabályban egy feltételrészlet mutál.
- **θ_{del}**: Törlési korlát, aminél ha egy szabály tapasztalata nagyobb, akkor a fitness értéke is beleszámít a törlési valószínűségébe
- **δ**: [P] szabályainak átlag fitness értékének az a hányada, ami alatt egy szabály fitness értéke beleszámít a törlésének valószínűségébe.
- **$P_{\#}$**: Wildcard használatának (azaz feltételrészlet eltávolításának) valószínűsége az eredeti szabályból covering eljárás során.
- **ρI, eI, FI**: Kezdeti paraméterei (becslés, fitness, predikciós hiba) egy újonnan létrehozott szabálynak.

- $\mathbf{p_{explr}}$: Explorációs valószínűség. Ha ez az érték 0-nál nagyobb, és kockadobással ennél nagyobb számot dobunk egy iterációban, akkor a javasolt ítéletet nem kifizetődés alapján választjuk [M] elemei alapján, hanem véletlenszerűen.
- $\boldsymbol{\theta_{mna}}$: [M] minimális mérete, ha ennél kisebb, megindul a covering eljárás.
- **doGASubsumption**: Eldönti, hogy végrehajtódjon-e subsumption eljárás a genetikus algoritmus folyamán, vagy sem.
- **doActionSetSubsumption**: Eldönti, hogy végrehajtódjon-e subsumbtion eljárás [A] elemein, vagy sem.

4.2. Az eljárás előnyei és hátrányai

Az eljárás egyik legnagyobb előnye, hogy **könnyen interpretálhatók az eredmények**. A populációban lévő szabályok emberi szemmel is könnyen értelmezhetők. A szabály populációt továbbá a szabályok paraméterei alapján **szűrni is tudjuk**, hogy a legjobbnak vélt becslések közül válogathassunk.

Az algoritmus sztochasztikus működésének mivoltából jól alkalmazható **komplex vagy nagy méretű problémák esetén**. Az algoritmus ezen felül rendelkezik egy **implicit általánosító nyomással**, amivel a jól működő általános szabályok nagyobb eséllyel tudnak szelektálódni arra, hogy tovább örököltessék a tulajdonságaikat. Többek között ez a tulajdonsága is jó választássá teszi biomarker keresés megvalósítására.

Előnyösnek tekinthető az eljárás azon tulajdonsága, hogy nem tesz semmilyen előfeltételezést a probléma környezetről, amit fel kell derítenie, ebből kifolyólag flexibilisen lehet alkalmazni több különböző biológiai hálózaton is. **Kevesebb tanuló adattal is** jól tud működni, a bemeneti adatok egy részéből nem kell validációs halmazt generálnia, így az adatgeneráláshoz szükséges szimulációk számát is mérsékli.

Az eljárás egyik nagy hátránya a limitált elterjedtsége, ebből kifolyólag igen **korlátozott az implementáltsága** különböző könyvtárakban, szoftverekben [13]. A sztochasztikus működésből kifolyólag **kevés elméleti eredmény** áll rendelkezésre az XCS tanulásának konvergenciájáról, elméleteiről.

Mint azt láttuk, az algoritmus rengeteg szabályozható és optimalizálható paraméterrel rendelkezik. Ezeknek a többségének a megválasztása nehezen indokolható, a megoldandó **problémához igazítani őket nem mindig triviális**. Bár a legtöbb paraméterre vannak megfogalmazva általános ajánlások, az iterációs lépések száma és a populáció maximális nagyságának a meghatározása mindenképp olyan, amiket gyakorlati tapasztalatok alapján lehet csak beállítani.

5. Implementáció

5.1. Tanuló adatok generálása

A tanuló adatok generálásához saját scriptet írtam, ami a Turbine hálózatszimulációs szoftver a beágyazott Python interpreterén keresztül futtat. A scriptben importált **pubi_utils csomagban** valósítottam meg segédfüggvényeket, mint például a kódban látható **simulate függvényt**, ami egy mutációkat és értékeiket tartalmazó dictionary alapján elvégzi a szükséges szimulációkat, hogy utána a hálózat kiértékelhető legyen. A szimulációk végeztével megkapjuk a hálózatnak az adott körülményekhez tartozó attraktorait, azaz a különböző stabil állapotokat, amikbe kerül. Az **eval** könyvtárban található output függvény felel azért, hogy a különböző élettani paramétereit kiolvassa a hálózatnak a kapott attraktorok alapján.

Az alábbi script a megvalósított **adatgenerálást** tartalmazza. A kód parancssori argumentumként elkéri, hogy **milyen hosszúságú kombinációk** halmazára vagyunk kíváncsiak, továbbá ezeknek körülbelül **mekkora részhalmazát** szeretnénk megkapni (utóbbi egy valószínűségi mutató, a nagy számok törvényéből következően jól fogja közelíteni a kívánt méretet). Kimenetként a program egy CSV file-ba írja ki az eredményeket.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import turbine
import sys
import pubi_utils
import eval
import spec
import itertools
import yaml
import time
import random
import math

if __name__=="__main__":
    #Cleanup workspace and disc if necesarry
    pubi_utils.clean()

    #Argumentum validity check
    max_subset_length = int(sys.argv[1])
    cover_chance = float(sys.argv[2])
    if max_subset_length < 1:
        raise RuntimeError('Max subset length lesser than 1!')
    if float(cover_chance) < 0 or float(cover_chance) > 1:
        raise RuntimeError("Invalid cover chance! Should be between [0:1]!")


    #Start clock
```

```
    start_time = time.time()

    #Generate result file
    with open('biomarker_results.txt', 'w') as outfile:
        #Generate header
        with open("outputs.yaml", 'r') as stream:
            output_levels = yaml.load(stream) #load evaluation levels
        outfile.write('Mutations\tBase_Mutations\t')
        for level in output_levels:
            outfile.write(level + '\t')
        outfile.write('\n')

        #Get node list of the network
        node_dict = pubi_utils.parse_nodelist()
        node_list = [node['name'] for node in node_dict]

        #Remove base mutations from nodelist
        for node in spec.BASE_MUTATIONS:
            node_list.remove(node)

        #Calculate number of possible combinations
        f = math.factorial
        possible_combinations = (f(len(node_list)) // f(max_subset_length) //
                f(len(node_list)-max_subset_length)) * 2 ** max_subset_length
        print("Preparing to simulate " + str(possible_combinations) +
             " possible combinations with " + str(cover_chance) + " chance!")

        #Iterate through all possible combinations
        for subset in itertools.combinations(node_list, max_subset_length):
            #Generate all possible mutations for current combination
            for mut_comb in range(0, 2 ** max_subset_length):
                #Roll the dice (decide to cover or not)
                if cover_chance >= random.random():
                    #Current combination to dict
                    mut_dict = {}
                    for i in range(0, len(subset)):
                        format_string =
                                    '{0:0' + str(max_subset_length) + 'b}'
                        mut_dict[subset[i]] =
                                    format_string.format(mut_comb)[i]

                    #Write current combination and base mutations to file
                    combi_str = '+'.join([str(node) + ':' +
                                str(mut_dict[node]) for node in mut_dict])
                    base_mutations = '+'.join([str(node) + ':' +
                                    str(spec.BASE_MUTATIONS[node])
                                    for node in spec.BASE_MUTATIONS])
                    outfile.write(combi_str + '\t' + base_mutations + '\t')

                    #Simulate current combination
                    pubi_utils.simulate(mut_dict)
```

```
                output = eval.evaluate(spec.ATTRACTOR_NAME, "alive")

                #Write results to outfile
                for level in output_levels:
                    outfile.write(str(output[level]) + '\t')
                outfile.write('\n')

    #Print runtime
    print("It took %s seconds to generate results" % (time.time() -
                                                      start_time))
```

A betanítás során két különböző összetételű adathalmazt generáltam. Az **első adathalmazban** viszonylag változatos összetételű kombinációkat szimuláltam 1-5 maximális hosszokkal. A halmaz 2165 elemet tartalmaz. A használt fedettségi összetételek a következők:

Kombináció hossz	Lefedettség
1	0.2
2	0.05
3	0.0001
4	0.00001
5	0.0000005

A **második tanító halmazban** törekedtem arra, hogy túlsúlyban legyenek a hosszabb szabályok, és kevesebb rövid szabályszerűség legyen. A halmaz 5201 elemet tartalmaz.

Kombináció hossz	Lefedettség
1	0.05
2	0.07
3	0.0005
4	0.00005
5	0.0000008
6	0.00000002

A két különböző összetételű halmaz megválasztásának az oka, hogy érdekelt, hogy a rövid szabályokat magasabb arányban tartalmazó halmazon végzett betanulás specifikusabb és pontosabb szabályokat eredményez-e. A második halmaztól intuitívan hosszabb, kevésbé specifikus, de visszaellenőrzések során pontatlanabbnak bizonyuló szabályokat várok.

5.2. XCS implementálása

A dolgozat ezen a pontján felmerül kérdésként, hogy a tanuló algoritmust miért saját kezűleg implementálom ahelyett, hogy egy meglévő könyvtárat használnék.

Az algoritmus maga kevésbé ismert széles körben, mint más népszerűbb tanuló algoritmusok, ezért nem bővelkedik előre implementált könyvtárakban, azonban Python 3 alá létezik egy **XCS package**, ami megvalósítja az algoritmust [14].

A könyvtár maga kisebb kísérletek elvégzése után ígéretesnek tűnt, azonban szembe ötlött egy hatékonyságbeli hibája. A könyvtárban a tanítás során az algoritmus úgynevezett **bitstring** (1-esekből, 0-ásokból, és wildcard (#) karakterekből álló string) **típusú adatokkal** dolgozik. A biomarker probléma esetén egy mutációs kombinációt át tudunk írni ilyen reprezentációra, ha a funkciónyerő mutációkat 1-essel, a funkcióvesztő mutációkat 0-ással jelöljük. Egy ilyen szabály hossza megegyezik a hálózatban lévő csúcsok számával, minden egyes helyiérték egy-egy ponttal lesz megfeleltethető. A szabályban nem szereplő helyiértékeket wildcard-okkal jelölhetjük.

Példa

Hálózatban szereplő csúcspontok: [A...G], 7 csúcspont

Egy egyszerűbb szabály: A:0+B:1+G:1 -> ALIVE

A szabály reprezentációja bitstring-ként: 01####1 -> ALIVE

A **probléma hatékonyság szempontjából** ennél a reprezentációnál ott jelentkezik, amikor két szabályt megpróbálunk összehasonlítani egymással, mivel a **komparációk maximális száma** többé nem az összehasonlított szabályok hosszától, hanem a **hálózat méretétől fog függeni**. Nagy hálózatokban (például 1000 csúcs fölött) rövid szabályokat (például 1..5 hosszú) kifejezetten nem hatékony ezzel a módszerrel komparálni.

Egy jobb reprezentáció Pythonban, amit a **saját implementációmban** használtam, ha az egyes **szabályokat dictionary típusú objektumokban** tároljuk, amik gyakorlatban hash tábláknak felelnek meg. Ekkor két szabály összehasonlításának idejét **felülről korlátozza** a hosszabb szabályban lévő **feltételek száma**, hisz elemek keresése hash táblákban O(1) alatt megvalósítható. További előnye, hogy egyéb felhasználás esetén bináris helyett **több állapotú feltételeket is könnyebben** felvihetővé tesz a rendszerbe, nem feltétlenül kell dekomponálnunk egy feltételt több bináris változóra.

Az implementációm során bevezettem a **Classifier**, **Situation** és **XCS osztályt**. A Classifier és Situation osztályok recordtype típusúak, értelemszerűen a szabályok és tanuló adatok paramétereit foglalják magukba.

```
Classifier = recordtype("Classifier", ["condition", "action", "prediction",
"prediction_error", "fitness", "exp",    "ts",        "ass"  ,    "n"],
default = 0)
Situation = recordtype("Situation", ["condition", "action"])
```

Az XCS osztály implementálása folyamán nagyrészt a *M. V. ButzS. W. Wilson* által publikált *An algorithmic description of XCS* tanulmányt követtem [12]. A tanulmánytól abban tértem el, hogy a probléma bináris reprezentációja helyett a fent említett dictionary típusú reprezentációt használtam, és ehhez alakítottam a függvények működéseit.

Az alábbi kódrészlet egy jó példa arra, hogy mennyivel hatékonyabb ez a megközelítés komparálás esetén:

```
#Return true if classifier's codnition matches the situation's condition   (C
is subset of Situation)
#Match = all env vars in classifier are in a subset of the situation's env
vars + these env var's values are matching
    def does_match(self, classifier, situation):
        #Check all env vars in classifier if it matches the situation
        for env_var in classifier.condition:

            #if an env var of classifier is not in situation, FALSE
            if env_var not in situation.condition:
                return False

            #if the value of the env var in classifier doesn't match it's
             pair in the situation, FALSE
            if classifier.condition[env_var] != situation.condition[env_var]:
                return False

        #It's a match
        return True
```

5.2.1. A tanuló paraméterek meghatározása

A tanuló algoritmus felparaméterezéséhez viszonylag kevés kiindulási pontot találtam az irodalomban. Az alábbi paraméterek beállítása során az **alábbi javaslatokat** vettem figyelembe [12]:

- A populáció maximális mérete legyen elég nagy ahhoz, hogy covering eljárás csak az első iterációban történjen
- A tanulási sebesség (learning rate) legyen 0.1 és 0.2 között
- A fitness meghatározásánál alkalmazott alfa paramétert általában 0.1-re szokás állítani
- A fitness meghatározása során használt e0 paraméter adja meg a hibahatárt, ami mellett két classifier azonos pontosságúnak tekinthető. Érdemes ezt a maximális jutalomnak 1% körül meghatározni
- A classifier fitness-ének számolásakor használt hatvány paraméter (v) tipikusan 5
- A theta_GA threshold tipikusan 25-50 között van
- A keresztező valószínűséget 0.5-1.0, míg a mutációs valószínűséget 0.01-0.05 közé szokás beállítani
- A törlő (theta_del) treshold tipikusan 20
- Wildcard-valószínűség gyakran 0.33 (nagyobb értékek lassítják a pontosabb szabályok evolúcióját)
- A classifier paraméter inicializációs értékeinek 0 közeli számok a megfelelőek
- A felfedezési valószínűség (p_explr) esetében 0.5 lehet egy jó kompromisszum, de ez problémafüggő

A fenti ajánlásokat használtam kiindulópontként, az eltéréseket attól empirikus alapon változtattam meg, amikor úgy tűnt, hogy a tanulás folyamatát jobb irányba tolják el. A **subsumption eljárások kikapcsolása** mellett döntöttem, mivel különösképp általános, hosszabb szabályok megtalálása érdekes biomarkerek keresésénél. A beállítás bekapcsolása akkor tud hasznos lenni, ha a probléma mögött egy jól definiált célfüggvény áll, a mi esetünkben azonban a lehetséges problématérnek csak egy szeletéből kapott szimulációs adatokkal dolgozunk, így a classifier feltételek szándékos specializálása ezekre subsumption eljárásokkal felvetheti a **túltanulás veszélyét** is.

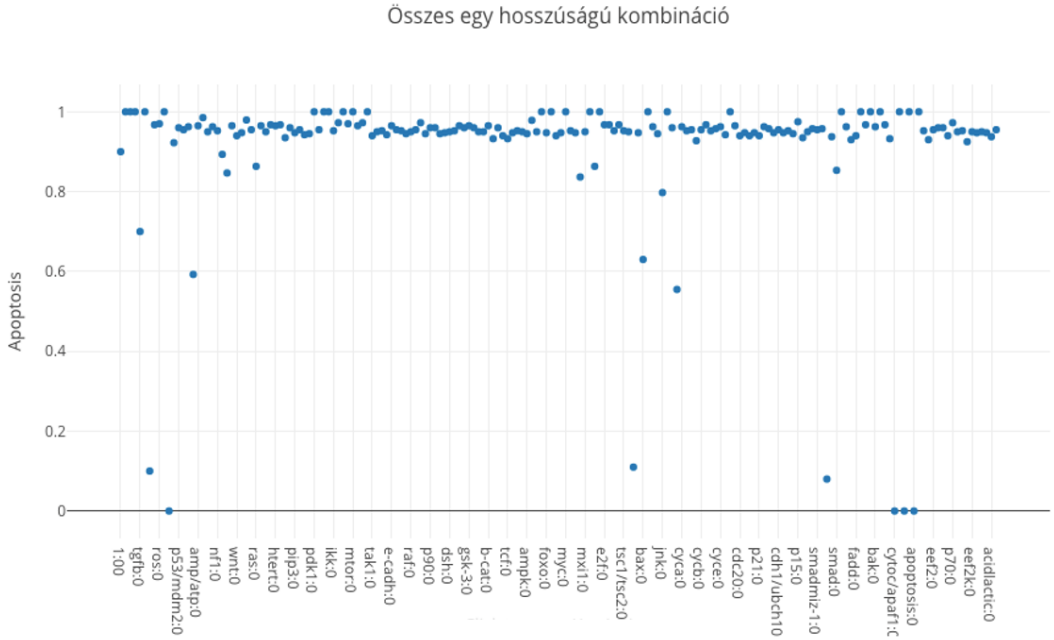
Az XCS osztályból való örököltetéshez kötelezően definiálandónak állítottam be a get_reward függvényt, hogy az problémaspecifikusan felülírható legyen. A biomarker keresés problémájához ezt úgy állítottam be, hogy ha a megfelelő ítéletet javasolja a rendszer, 1 jutalmat adjon visszatérési értékként, egyébként ne adjon semennyit.

```
from xcs import *

class BiomarkerXCS(XCS):
    def get_reward(self, situation, action):
        if (action == situation.action):
            return 1
        return 0
    def init_params(self):
        #parameters
        self.N = 22000 # maximum size of the population (sum of classifier
                         numerosities)
        self.theta_mna = 2   # minimum number of different actions needed in
                               a match set
        self.p_wildcard = 0.33 # wildcard probability (chance of using
                                 wildcard when covering)
        self.pI = 0.0001 # Initial prediction estimate of new classifier
        self.eI = 0.0001 # Initial prediction error of new classifier
        self.FI = 0.0001 # Initial fitness of new classifier
        self.p_explr = 0.5 #Probability of choosing an action randomly during
                             action selection
        self.theta_del = 20 # Deletion threshold
        self.e0 = 0.1 #used in calculating the fitness of classifier
        self.alfa = 0.1 #used in calculating the fitness of classifier
        self.v = 5 #used in calculating the fitness of classifier
        self.beta = 0.1 #learning rate (for classifier )
        self.theta_GA = 40 #GA will be applied in a set, when avarage time
                            since the last GA is greater than theta_GA
        self.mu = 0.04 #probability of mutating an allele in the offspring
        self.delta = 0.5 #mean fitness, below which a classifier may be
                           considered in its probability of deletion
        self.theta_sub = 20 #Subsumption threshold (experience needed, before
                             a classifier can subsume)
        self.kszi = 0.5 #probability of applying crossover in GA
        self.do_A_set_subsumption = False
        self.do_GA_subsumption = False
```

5.3. Az elvárható eredmények feltérképezése

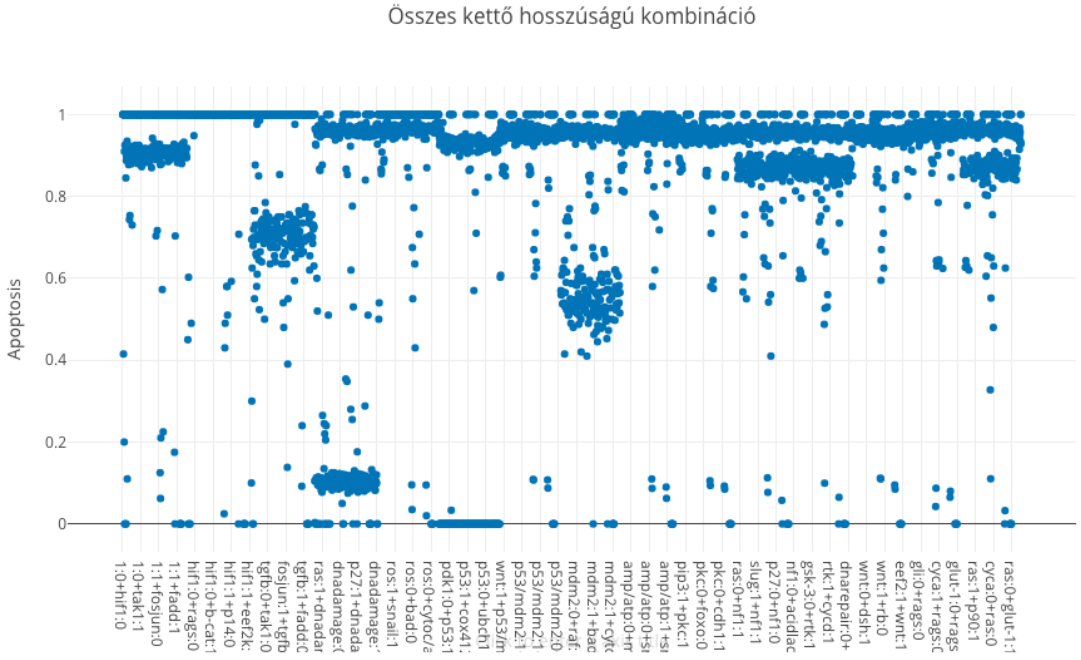
Mielőtt elkezdtem volna az adatokon tanítást és kiértékelést, megpróbáltam valamilyen képet szerezni arról, hogy nagyjából milyen összetételű szabályokra kellene számítanom a későbbiekben. Ahhoz, hogy erről képet kaphassak, leszimuláltam az összes 1 és 2 hosszúságú kombinációt, hogy azok összetételére betekintést nyerhessek.

Emlékeztetőül, a hálózatban azokat a kombinációkat fogjuk élőnek tekinteni, ahol az attraktorok kereséséhez futtatott szimulációk legalább 50%-ában az apoptosis csúcs ki volt kapcsolva. Az egyszerűség kedvéért a továbbiakban amikor az apoptosis csúcs értékére hivatkozunk, annak a szimulációk során vett kiátlagolt értékét fogjuk érteni, azaz 0.5-ös apoptosis alatt tekintjük élőnek a hálózatot.



7. ábra: Egy hosszúságú kombinációk hatása átlagoltan az apoptosis csúcsra

Mint az a fenti ábrán látható, **egyes kombinációk közül, viszonylag kevés** (szám szerint 7 darab, de azok közül az apoptosis:0 kombináció biológiailag nem igazán értelmezhető) van, ami 0.5-ös apoptosis érték alá viszi, azaz felére leszti a gyógyszerrel megölt sejtet.



8. ábra: Kettő hosszúságú kombinációk hatása átlagoltan az apoptosis csúcsra

Kettő hosszúságú kombinációk esetén már jobb a helyzet, 1239 szabály is van 16380-ból, ami megfelel a kritériumnak. Ez arányaiban **kiegyensúlyozottabb**, mint az előző esetben és szemmel láthatóan jobban szórnak is az eredmények.

A különbség a két kombinációs hossz között összetételben megmutatja, hogy jó eséllyel számíthatok arra, hogy komplex, specifikus biomarkereket találjak a tanítás közben, olyanokat, amik nem bonthatók tovább elemeikre. Előfeltételezésként tenném meg még ezen a ponton, hogy a második tanító szettnél a specifikusabb szabályoknál jó eséllyel számíthatok arra, hogy fals pozitívak lesznek nagy arányban, ha túláltalánosít a rendszer és sok ilyet generál.

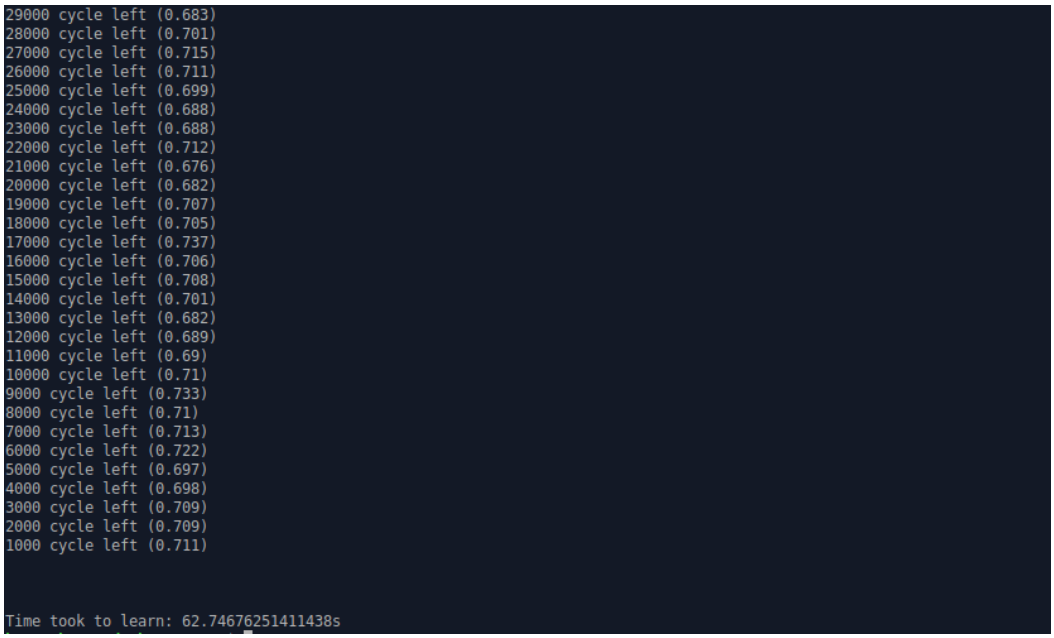
5.4. Tanítás és eredmények kiértékelése

A tanítás megkönnyítése és az eredmények kiértékelése érdekében egy rövid scriptet írtam, ami a tanulás végén a populációt egy excel fájlba menti ki:

```
#Create problem
bio_problem = BiomarkerXCS()

#Add possible actions
bio_problem.add_possible_action("ALIVE")
bio_problem.add_possible_action("DEAD")

#Add env variables
for n in nodes:
    bio_problem.add_env_variable(p)

#Add situations
for s in rules:
    bio_problem.add_situation_to_env(s.condition, s.action)

#Start learning
start_time = time.time()
bio_problem.run_experiment(50000)
stop_time = time.time()

#Print learning time
print("Time took to learn: " + str(stop_time - start_time) + "s")

#write to excel
workbook = xlsxwriter.Workbook('rules.xlsx')
worksheet = workbook.add_worksheet(input_file_path[:30])

#Create column headers
worksheet.write(0,0, "Condition")
worksheet.write(0,1, "Action")
worksheet.write(0,2, "Fitness")
worksheet.write(0,3, "Reward")
worksheet.write(0,4, "Error")
worksheet.write(0,5, "Numerosity")
worksheet.write(0,6, "Exp")

row = 1
col = 1
for cl in bio_problem.population:
    line = ""
    for c in cl.condition:
        line += c + ":" + cl.condition[c] + "+"
    line = line[:-1]
    worksheet.write(row, 0, line)
    worksheet.write(row, 1, cl.action)
    worksheet.write(row, 2, cl.fitness)
    worksheet.write(row, 3, cl.prediction)
```

```
    worksheet.write(row, 4, cl.prediction_error)
    worksheet.write(row, 5, cl.n)
    worksheet.write(row ,6, cl.exp)
    row += 1
workbook.close()
```

A tanítás során hátramaradt **iterációs számot** több különböző számot kipróbálva kísérleteztem ki. Próbáltam figyelemmel tartani, hogy 1000 ciklusonként mekkora az átlagosan kiosztott jutalom (az én implementációmban az maximum 1, minimum 0). Mindkét tanító halmaznál nagyjából azt tapasztaltam, hogy **50000 ciklus** elég ahhoz, hogy stagnálni kezdjen, ezért itt állítottam le a keresést. A magasabb ciklusszámot próbáltam elkerülni, mivel nem szerettem volna megkockáztatni, hogy túltanuljon a rendszer.



9. ábra: A tanulás folyamata futás közben

Mint a fenti ábrán látható, a betanulás nagyjából egy percbe tellett, ami futási idő szempontjából ígéretes. A hátralévő futási ciklusok száma mellett láthatók a legutolsó ezer tanuló ciklusban átlagosan kiosztott jutalom. A kapott eredményeket ezután megszűrtem, hogy csak a rendszer által legjobbnak tartott szabályokat kelljen visszaellenőriznem.

A kapott szabályok szűrését Excelben végeztem. **Szűrőfeltételeknek** az alábbiakat választottam:

- Egy szabály **fitness**-e legyen **nagyobb, mint 0.5**
- Csak **ALIVE**-ot javasló szabályokra vagyunk kíváncsiak, mivel rezisztenciát keresünk
- A kapott **átlagos jutalom** legyen **több, mint 0.5** (különben a 0 jutalmat kapó ALIVE szabályok kvázi 1 jutalmat kapó DEAD szabályoknak felelnek meg)
- **Predikciós hiba** legyen **kisebb, mint 0.1**

Ezután a maradék szabályokat leszimulálva az első esetben az alábbi eredményt kaptam:

Condition	Action	Fitness	Reward	Error	Numerosity	Exp	Szimulált_APOP
rb:1+cyca:0	ALIVE	0,510652	1	0	6	7	0,965
ampk:0+chk1/2:1	ALIVE	0,520581	1	0	6	8	0,945
myc:1+chk1/2:0	ALIVE	0,532183	1	0	7	10	0,09
ubch10:1+nf1:1	ALIVE	0,560749	1	0	7	9	0,87
mdm2:1+bcl-xl:1	ALIVE	0,574286	1	0	13	12	0,5
ldha:0+cytoc/apaf1:0	ALIVE	0,57884	1	0	6	8	0
ldha:1+dnadamage:0	ALIVE	0,581513	1	0	8	12	0,1025
dsh:1+chk1/2:0	ALIVE	0,589063	1	0	16	22	0,1377551
ubch10:1+dnadamage:0	ALIVE	0,600493	1	0	9	27	0,09
mdm2:1+dnadamage:0	ALIVE	0,627276	1	0	10	15	0
mxi1:0+vegf:0	ALIVE	0,648283	1	0	13	13	0,96
jnk:0+atm/atr:1+chk1/2:0+tgfb:0	ALIVE	0,651308	1	0	50	10	0,113402062
p15:1+dnadamage:0+phd:0+vegf:1	ALIVE	0,759521	1	0	23	15	0,1275
chk1/2:0+pten:0	ALIVE	0,84336	1	0	27	31	0,0625
ras:0+dnadamage:0	ALIVE	0,849889	1	0	38	18	0
p53:0	ALIVE	0,871026	0,992017	0,026389	178	789	0
caspase9:0	ALIVE	0,910367	0,964998	0,067227	173	743	0
nf-kb:0+chk1/2:0	ALIVE	0,937023	1	0	43	48	0,089893617
apoptosis:0	ALIVE	0,946499	1	0	176	541	0

10. ábra: A kapott szűrt szabályok az első tanuló halmaznál

Az első esetben 19-ből 15 szabályt jól prediktált a rendszer, ezzel közel **79%-os pontosságot** ért el. Az **átlagos szabályhossz** ebben az esetben **2,05** volt. A szabályok között találhatók komplexebb, jól működő szabályok, ez abból a szempontból, hogy pont ilyenek szabályokra várunk javaslatot, sikeresnek tekinthető. A szabályokat a tanuló adathalmazzal összenézve kijelenthető, hogy a rendszer **talált olyan szabályszerűségeket**, **ami a tanuló adatok között egy-az-egyben nem volt jelen**. Egy ilyen szabály például az nf-kb:0+chk1/2:0 kombináció, amik együttesen a bemenő adatok között csak ezekben a szabályokban szerepel:

Mutations	**Apoptosis**
nf-kb:0+dnadamage:1+chk1/2:0+tsc1/tsc2:0	0.09631578947368422
p53/pten:0+nf-kb:0+chk1/2:0+gli:1	0.0
wnt:0+nf-kb:0+miz-1:1+chk1/2:0+cdc20:1	0.0895
cdh1/ubch10:1+vhl:1+nf-kb:0+chk1/2:0+cycd:1	0.018000000000000002

Ezzel elmondhatjuk, hogy az eljárás **alkalmas új szabályok kinyerésére**. A példa jól mutatja továbbá, hogy véletlenszerűen hosszabb szabályok szimulálásával kinyerhetők általánosabb összefüggések is az eljárással.

A második tanuló adathalmazzal hasonló találati arányt kapunk:

Condition	Action	Fitness	Reward	Error	Numerosity	Exp	Szimulált_APOP
dnadamage:0+myc/max:1	ALIVE	0,3146851	1	0	2	7	0,1
e2f/cycline:1+chk1/2:0	ALIVE	0,33498789	1	0	4	5	0,089
rheb:0+p15:1	ALIVE	0,348302667	1	0	3	5	0,9525
dnarepair:1+pip3:1+chk1/2:0	ALIVE	0,348916771	1	0	4	4	0
mtor:1+caspase9:0	ALIVE	0,353493788	1	0	3	5	0
e2f/cycline:1+atm/atr:0	ALIVE	0,358815267	1	0	3	4	0,1075
p70:0+chk1/2:0	ALIVE	0,37414452	1	0	3	7	0,0994
cox412:1+cytoc/apaf1:0	ALIVE	0,386275973	1	0	1	5	0
dnadamage:0+pten:1	ALIVE	0,396913892	1	0	8	5	0,24
smad:1+cytoc/apaf1:0	ALIVE	0,39993071	1	0	3	7	0
glut-1:0+atm/atr:0	ALIVE	0,403509085	1	0	5	9	0,065
gsk-3:0+myc:0	ALIVE	0,431505454	1	0	4	5	1
smade2f:1+chk1/2:0	ALIVE	0,445913852	1	0	3	8	0,0788
bax:1+atm/atr:0	ALIVE	0,453905893	1	0	3	7	0,285
vhl:1+cytoc/apaf1:0+tgfb:0+e-cadh:0	ALIVE	0,468603159	1	0	36	6	0
nf-kb:1+e2f:1+apc:1+atm/atr:0	ALIVE	0,542995145	1	0	54	8	0,03
bax:0+glut-1:1	ALIVE	0,55499497	1	0	29	8	0,55
gsk-3:0+cytoc/apaf1:0	ALIVE	0,772314055	1	0	12	16	0
p53:0	ALIVE	0,940167679	0,990829821	0,033167366	142	551	0
apoptosis:0	ALIVE	0,983270334	1	0	193	707	0

11. ábra: A kapott szűrt szabályok a második tanuló halmaznál

Megjegyzés: A szűrési feltételeken ebben az esetben finomítottunk, mivel 0.5-ös fitness felett viszonylag kevés szabály volt.

A **találati arány** az előzőhöz hasonlóan itt is elég magas, **85%-os**. A különbség, amit előzetes becslések alapján vártunk, nem jelentkezett jelentős mértékben a szűrt szabályoknál, az **átlagos hosszuk 2.15**. Ez alig több, mint az első adathalmaznál. Ennek több oka lehet, előfordulhat, hogy nem volt elég iterációs lépés, ezért a rendszer még túl általános maradt, esetleg a felparaméterezés nem engedte az adott problémánál a specifikusabb találatok elérését. Könnyen megeshet azonban az is, hogy nincs nagy fajlagos arányban a hálózatban meglévő különböző hosszúságú kombinációk között olyan, amely rezisztenssé tehetné a sejtet, hiszen az egész állapottérről sajnos nincs képünk.

Mindkét tanuló halmaz felépítése jóval **kevesebb szimulációt** igényelt, mint az akár az összes 2 hosszú szabály leszimulálása lett volna. A **tanulás ideje elhanyagolható** volt a megoldandó probléma szempontjából mindkét esetben. Ennek ellenére **képes volt működő hosszú kombinációs szabályokat megtalálni**, amiket nyers erővel sokkal időigényesebb lett volna.

6. Összegzés

A dolgozat során bemutattam egy alternatív eljárást arra, hogy hogyan lehet nekikezdeni biomarkerek keresésének biológiai hálózatokban, ha el szeretnénk kerülni az összes létező variáció leszimulálását, esetleg a háló méretéből kifolyólag ez nem is lehetőség.

A módszer előnye, hogy az eredetileg szükséges szimulációs lépéseknek a töredékével is megkaphatunk működő, hosszú szabályokat, így kis szerencsével rengeteg számítási kapacitást megspórolhatunk. Ha jó szűrőfeltételeket alkalmazunk a kapott szabályokra, azok jó arányban működőképesek is. Továbbá az eredmények könnyen interpretálhatóak, ami kutatóknak, biológusoknak megkönnyíti a kapott eredmények feldolgozását.

A módszer hátránya, hogy nem tudjuk, hogy mennyi ígéretes szabály lenne még, ami gyógyszerek fejlesztéséhez még további hasznos iránymutatást adna, viszont ezt nyers erővel sem lenne módunk kideríteni. Ez gyógyszerfejlesztéskor azonban nem feltétlenül probléma, egy jól eltalált célpont kombináció is elég egy új célzott terápiás szer szabadalmaztatásához.

A tanulás sztochasztikus mivoltából és a rengeteg paraméteréből kifolyólag az XCS használata nem feltétlenül egyszerű, nehéz megmondani látatlanból, hogy mikor találtuk meg az optimális beállításokat.

A módszer továbbfejlesztésének vizsgálatához érdemes lenne több különböző hálózaton is kipróbálni a működését. Érdekes lehet vizsgálni, hogy kisebb hálózatok esetén, ahol fel tudnánk deríteni az összes állapotot nyers erővel, megmérjük hogy a tanuló adathalmaz méretének növelésével

Az adatok biológiai feldolgozását megkönnyítendően hasznos lehet még az utófeldolgozás folyamatába beilleszteni egy biomarker dekomponáló lépést, azaz a hosszabb szabályokat rész szabályokra bontani, és úgy is leszimulálni, hátha általánosíthatóbbak még tovább.

7. Köszönetnyilvánítás

A dolgozatom elkészítésében a segítséget és észrevételeket szeretném megköszönni a konzulensemnek, Dr. Szalay Kristóf Zsoltnak, és a biológiai kérdésekben való iránymutatást Varga Júliának.

8. Irodalomjegyzék

[1] Ranran Zhang, Mithun Vinod Shah, Jun Yang, Susan B. Nyland, Xin Liu, Jong K. Yun, Réka Albert, and Thomas P. Loughran Jr, *Network model of survival signaling in large granular lymphocyte leukemia,* PNAS October 21, 2008 105 (42) 16308-16313

[2] Herman F. Fumiã, Marcelo L. Martins, *Boolean Network Model for Cancer Pathways: Predicting Carcinogenesis and Targeted Therapy Outcomes,* July 26, 2013

[3] Gaowei Wang, Xiaomei Zhu, Jianren Gu, Ping Ao, *Quantitative implementation of the endogenous molecular–cellular network hypothesis in hepatocellular carcinoma* Interface Focus. 2014 Jun 6; 4(3): 20130064.

[4] Zhu X., Yuan R., Hood L., Ao P., *Endogenous molecular-cellular hierarchical modeling of prostate carcinogenesis uncovers robust structure,* Prog Biophys Mol Biol. 2015 Jan;117(1):30-42

[5] Ranran Zhang, Mithun Vinod Shah, Jun Yang, Susan B. Nyland, Xin Liu, Jong K. Yun, Réka Albert, and Thomas P. Loughran Jr., *Network model of survival signaling in large granular lymphocyte leukemia,* PNAS October 21, 2008 105 (42) 16308-16313;

[6] Ruoshi Yuan, Xiaomei Zhu, Jerald P. Radich, and Ping Aoa, *From molecular interaction to acute promyelocytic leukemia: Calculating leukemogenesis and remission from endogenous molecular-cellular network,* Sci Rep. 2016; 6: 24307.

[7] https://hu.wikipedia.org/wiki/Hum%C3%A1n_genom, lap utolsó módosítása: 2018. március 30., 08:11

[8] https://ncase.me/attractors/

[9] https://www.frontiersin.org/articles/10.3389/fgene.2017.00048/full

[10] https://www.mitpressjournals.org/doi/10.1162/evco.1995.3.2.149

[11] https://en.wikipedia.org/wiki/Machine_learning

[12] M. V. ButzS. W. Wilson, An algorithmic description of XCS, S. Soft Computing (2002) 6: 144.

[13] https://en.wikipedia.org/wiki/Learning_classifier_system, lap utolsó módosítása: 20 August 2018, at 19:50 (UTC)

[14] https://pythonhosted.org/xcs/