*tanulás (AI)*

*multinominal-logistic-regression*

*raw C*

**A kód működéséről:**

dot\_product():

Ez a függvény két vektor skaláris szorzatát számolja ki. A skaláris szorzat az összes megfelelő pozícióban lévő elem szorzatának összege.

sigmoid():

Ez a függvény végzi a sigmoid aktivációs függvény számítását, ami az inputot (x) átalakítja egy értékbe, ami 0 és 1 között van. Ez az érték általában interpretálható valószínűségként.

train():

Ez a függvény végzi a modell tanítását. A gradient descent algoritmust alkalmazza a modell súlyainak frissítésére **(nagy bementi halmazok esetén érdemes véletlenszerűen választani!),** hogy a modell minél jobban illeszkedjen az adatokhoz.

predict():

Ez a függvény használja a tanult súlyokat és a bemeneti adatokat a predikcióra. A sigmoid aktivációs függvényt használja annak megállapítására, hogy a kimeneti osztály 1 vagy 0 legyen.

A modell tanítása során a train() függvény minden iterációban végigmegy az összes adatponton, és a gradient descent algoritmust használva frissíti a súlyokat úgy, hogy a modell minél közelebb kerüljön a megfelelő osztályok meghatározásához. A predikcióhoz a predict() függvény a tanult súlyokat használja, hogy a bemeneti adatokat osztályba sorolja a sigmoid aktivációs függvény segítségével.

**Skaláris szorzat:**

A skaláris szorzat szerepe a multinomiális logisztikus regresszióban az, hogy kiszámolja a bemeneti tulajdonságvektor és a súlyvektor közötti lineáris kombinációt. Ez az érték azt jelzi, hogy mennyire illeszkedik a bemeneti adat a modell által tanult mintákhoz.

Az illeszkedés mértékét a lineáris kombinációhoz hozzáadott konstans, azaz az intercept határozza meg. A sigmoid aktivációs függvény ezután ezt a lineáris kombinációt átalakítja egy olyan értékké, ami valószínűségértékként értelmezhető, jelzi, hogy az adott bemeneti adat milyen valószínűséggel tartozik az egyes osztályokba.

Tehát a skaláris szorzatot arra használjuk, hogy a modell számára megadott bemeneti adatok alapján meghatározzuk, hogy milyen valószínűséggel tartozik az egyes osztályokba. A sigmoid aktivációs függvény ezután a valószínűséget egy 0 és 1 közötti értékben reprezentálja, ami segít a predikcióban és a modell tanulásában.

**Továbbfejlesztési lehetőség (multiLogRInit)**

**Validáció hozzáadása:** Egy validációs halmaz használata a tanítás során segítene megakadályozni a túlzott illeszkedést (overfitting) és segíthetne a modell teljesítményének értékelésében.

**Regularizáció:** Regularizációs módszerek, például L1 vagy L2 regularizáció hozzáadása a súlyvektorokhoz. Ez segíthet abban, hogy a modell általánosabb legyen és csökkentheti a túltanulást.

**Kötegelt tanítás (Batch Training):** Jelenleg a kód minden egyes mintára frissíti a súlyokat. A kötegelt tanítás során több mintát is figyelembe vehetünk a súlyok frissítésekor, ami hatékonyabb lehet, különösen nagy adathalmazok esetén.

**Konvergencia ellenőrzése:** A függvényhez hozzáadhatunk egy konvergencia ellenőrzést, amely figyeli a költségfüggvény vagy más metrikák változását az iterációk során, és meghatározza, mikor érte el a modell a konvergenciát.

**Kimeneti információk javítása:** A függvény kimenetét bővíthetjük olyan információkkal, mint például a tanítási hiba, a konvergencia időtartama stb.

**Inicializációs stratégiák:** Jobb inicializációs stratégiák alkalmazása a súlyvektorok kezdeti beállításához, például Xavier vagy He inicializáció.

**Hiba kezelés:** A kód ellenőrizhetné a bemeneti adatok érvényességét és kezelhetné a hibás eseteket.

**Validáció**

A validáció a tanítási folyamat során azért fontos, mert segít megakadályozni a túltanulást (overfitting), és általában javítja a modell teljesítményét. A validáció folyamata a következő lépésekből áll:

1. **A tanítás során a modell paramétereit frissítjük a tanító adatokon.**
2. **A validációs halmazon értékeljük a modell teljesítményét:** Ez azt jelenti, hogy a tanítási adatoktól független adatokon kiértékeljük a modell teljesítményét. A validációs halmaz gyakran a tanító adatok egy részhalmaza, amelyet nem használtunk fel a tanításhoz.
3. **A validációs veszteség (loss) vagy pontosság (accuracy) alapján döntünk:** Például ha a validációs veszteség túl magas vagy az pontosság túl alacsony, ez jelzi, hogy a modell nem teljesít jól a tanulási folyamat során, és esetleges túltanulás lehet.
4. **Döntés a tanítás folytatásáról vagy befejezéséről:** Ha a validációs veszteség elért egy előre meghatározott küszöbértéket, akkor leállíthatjuk a tanítást, mert ez azt jelzi, hogy a modell valószínűleg már jól illeszkedik a tanító adatokra és további tanítás nem szükséges.

A fenti példában a multiLogRTrain függvény a validációs halmazon értékeli a modell teljesítményét a tanítási iterációk során. Ezután az aktuális validációs veszteséget összehasonlítjuk egy előre meghatározott küszöbértékkel, és ha ez a veszteség elégségesen kicsi, akkor kilépünk a tanítási ciklusból.

Ez a folyamat segít abban, hogy megakadályozzuk a túltanulást és biztosítsuk, hogy a modell általánosítása jobb legyen, azaz jobban teljesítsen új, nem látott adatokon is.

**Négyzetes hiba validálásra:**

loop (< maxIteration) {

// Training

....

// Validáció

validLoss = 0.0

loop (< validDataCounter) {

p = sigmoid(weight,validDataInput[])

validLoss += ((dataOutput[] – p) \* (dataOutput[] -p))

}

validLoss /= validDataCounter

}

**Kódrészlet**

**bool multiLogRTrain(Setup \*s, Data \*d, Data \*validationData) {**

**for (int iteration = 0; iteration < s->maxIteration; ++iteration) {**

**for (int i = 0; i < d->sampleCounter; ++i) {**

**// Számítsuk ki a sigmoid függvény értékét a súlyvektor és bemeneti adatok dotproductjére**

**double p = calcSigmoid(calcDotProduct(d->weight, d->x[i], s->featureSize));**

**// Frissítsük a súlyvektort a gradient descent algoritmus segítségével**

**for (int j = 0; j < s->featureSize; ++j) {**

**\*(d->weight + j) += s->learningRate \* (d->y[i] - p) \* d->x[i][j];**

**}**

**}**

**// Validációs halmaz értékelése**

**double validation\_loss = 0.0;**

**for (int i = 0; i < validationData->sampleCounter; ++i) {**

**double p = calcSigmoid(calcDotProduct(d->weight, validationData->x[i], s->featureSize));**

**validation\_loss += (d->y[i] - p) \* (d->y[i] - p);**

**}**

**validation\_loss /= validationData->sampleCounter;**

**// Ha elég jó a validációs veszteség, akkor kilépünk a tanításból**

**if (validation\_loss < 0.01) {**

**printf("A tanítás korai befejezése a megfelelő validációs veszteség elérése miatt.\n");**

**return true;**

**}**

**}**

**return true;**

**}**

A validation\_loss változó összegzi a validációs adathalmaz minden egyes mintájára vonatkozó veszteségeket. Az (d->y[i] - p) \* (d->y[i] - p) kifejezés a négyzetes hiba (mean squared error) számítására szolgál.

A négyzetes hibát gyakran használják a regressziós modellek teljesítményének értékelésére. Ez a hiba az adott minta valós és becsült kimeneteinek különbségének négyzete. Az összes ilyen különbség négyzetének átlaga az adathalmazon a négyzetes hiba.

A négyzetes hiba előnyei közé tartozik, hogy a kisebb hibákat relatív nagyobb súllyal veszi figyelembe, ami azt jelenti, hogy a nagy hibákra történő túlérzékenyebb, és ezáltal jobban mutatja azokat a hibákat, amelyek nagy hatással lehetnek a modell teljesítményére. Ezenkívül a négyzetes hiba differenciálható, ami fontos a gradient descent algoritmusokkal való munka során.

Ezért az (d->y[i] - p) \* (d->y[i] - p) kifejezés használata segíti az összes veszteség összegzését és értékelését a validációs adathalmazon.

**Regularizáció**

A regularizáció jó, mert segít megelőzni a túltanulást (overfitting), amely gyakori probléma a gépi tanulásban és a statisztikában. A túltanulás akkor jelentkezik, amikor a modell túlságosan illeszkedik a tanító adatokhoz, és nem képes általánosítani a még nem látott adatokra.

A regularizáció célja a modell komplexitásának szabályozása és azáltal, hogy büntetést szab ki a súlyok méretére, segít megelőzni a túltanulást. Itt van néhány oka, hogy miért jó a regularizáció:

1. **Általánosítás javítása:** A regularizáció segít a modell általánosításában, vagyis abban, hogy jobban teljesítsen új, még nem látott adatokon is. Ez azáltal történik, hogy a regularizáció szabályozza a modell paramétereinek méretét, így a modell nem túl összetett és nem tanul meg olyan zajt vagy részleteket, amelyek csak a tanító adatokon jelentkeznek.
2. **Túltanulás csökkentése:** A regularizáció csökkenti a túltanulást, mivel a súlyokat kisebbeként tartja. Ezáltal a modell kevésbé fogja túlilleszteni a tanító adatokat, és jobban fog általánosítani.
3. **Stabilitás növelése:** A regularizáció hozzájárul a modell stabilitásának növeléséhez, mivel a súlyok szabályozásával kevésbé lesznek érzékenyek a kis változásokra a bemeneti adatokban vagy a tanítási folyamatban.
4. **Modellek jobb összehasonlíthatósága:** A regularizáció segíthet abban, hogy különböző modelleket könnyebben összehasonlítsunk, mivel segít csökkenteni a modell paramétereinek számát, így azok összehasonlíthatóbbak lesznek.

Ezek az okok mutatják meg, hogy miért jó a regularizáció, és miért használják széles körben a gépi tanulásban és a statisztikában. Segít a modell stabilabbá tételében és jobb teljesítmény elérésében mind a tanító, mind a tesztadatokon.

**Ebben a példában a regularizációs paramétert (lambda) hozzáadjuk a Setup struktúrához. A súlyvektor frissítésekor hozzáadjuk a regularizációs tagot a gradient descent algoritmushoz, ami segít abban, hogy a súlyvektorokat kisebbeként tartsuk, és ezzel csökkentsük a túltanulás (overfitting) lehetőségét. A regularizációs paraméter beállítása azt szabályozza, hogy mennyire szigorú legyen a regularizáció.**

**\*(d->weight + j) +=**

**s->learningRate \* ((d->y[i] - p) \* d->x[i][j] - s->lambda \* d->weight[j])**

**Kódrészlet**

**// Frissítsük a súlyvektort a gradient descent algoritmus segítségével**

**for (int j = 0; j < s->featureSize; ++j) {**

**// Regularizáció hozzáadása a súlyvektorhoz**

**\*(d->weight + j) += s->learningRate \* ((d->y[i] - p) \* d->x[i][j] - s->lambda \* d->weight[j]);**

**}**

**Lambda meghatározása**

A regularizációs paraméter, általában jelölve lambda-val (λ), a regularizáció erejét szabályozza. Tehát a kérdés, hogy milyen értéket válasszunk lambda-nak, az attól függ, hogy milyen komplexitást akarunk adni a modellnek és mennyire szeretnénk elkerülni a túltanulást.

Nincs egyetlen helyes válasz a lambda értékére, és gyakran az érték kiválasztása próbálgatás és hibázás eredménye, valamint a modell validációs teljesítményének figyelembevétele. Általában az alábbiakat kell figyelembe venni a lambda értékének kiválasztásakor:

1. **Kezdeti becslés**: Először próbáljon meg olyan lambda értéket választani, amely nagyjából azonos nagyságrendű, mint a súlyvektorok elemei. Például, ha a súlyok értékei 0 és 10 között vannak, akkor a lambda értéke is valahol ebben a tartományban lehet.
2. **Keresztezési validáció**: A keresztezési validáció segíthet abban, hogy különböző lambda értékeket teszteljen és értékeljen. Azaz különböző lambda értékekkel futtathatja a modellt, majd kiértékelheti a validációs halmazon a modell teljesítményét.
3. **Elkerülni a túlzott vagy túl alacsony regularizációt**: Ha a lambda értéke túl nagy, a modell lehet, hogy túl egyszerű lesz és alulillesztés (underfitting) jelentkezik. Ha a lambda értéke túl kicsi, akkor a modell túl bonyolult lehet és túltanulás (overfitting) következhet be. Tehát az optimális lambda értéket olyan kell lennie, hogy közvetlenül elkerülje mindkét problémát.
4. **Grid search vagy más hiperparaméter optimalizációs módszerek**: A grid search és más hiperparaméter optimalizációs módszerek segíthetnek abban, hogy automatikusan megtalálják az optimális lambda értéket a modell számára a validációs halmazon történő keresés útján.

Összességében a lambda értékének kiválasztása egy átgondolt folyamat, amely magában foglalja a modell és az adatok jellegét, valamint a validációs teljesítmény figyelembevételét. Az optimális érték megtalálása lehet némi kísérletezést igényel, de az általános gyakorlat az, hogy olyan lambda értéket válasszunk, amely segít megelőzni a túltanulást anélkül, hogy túlzottan leegyszerűsítené a modellt.

**Kötegelt tanítás (Batch Training)?**

A kötegelt tanítás (batch training) egy olyan módszer a gépi tanulásban, amelyben a modellt egy adott időszakban az összes rendelkezésre álló adatot használva frissítik, és nem csak egyetlen mintát vagy kis csoportot. Ez a módszer hatékony lehet, különösen nagy adathalmazok esetén, mivel lehetővé teszi a számítások párhuzamosítását és gyorsabb tanulást eredményezhet.

Fontos megjegyezni, hogy a kötegelt tanítás során az összes adatot egyszerre dolgozzák fel egy tanítási iteráció során. Ez eltér a Stochastic Gradient Descent (SGD) módszertől, ahol minden iterációban csak egy véletlenszerűen kiválasztott mintát használnak a súlyok frissítéséhez. Míg a SGD zajosabb és változékonyabb lehet, a kötegelt tanítás stabilabbá teheti a tanulási folyamatot.

**Kódrészlet**

**bool multiLogRTrain(Setup \*s, Data \*d) {**

**for (int iteration = 0 ; iteration < s->maxIteration ; ++iteration) {**

**// Inicializáljuk a gradiens vektort nullával minden iteráció előtt**

**double gradient[s->featureSize] = {0};**

**for (int i = 0; i < d->sampleCounter; ++i) {**

**double p = calcSigmoid(calcDotProduct(d->weight, d->x[i], s->featureSize));**

**for (int j = 0; j < s->featureSize; ++j) {**

**gradient[j] += (d->y[i] - p) \* d->x[i][j];**

**}**

**}**

**// Átlagoljuk a gradienseket a tanító adatok számával**

**for (int j = 0; j < s->featureSize; ++j) {**

**gradient[j] /= d->sampleCounter;**

**}**

**// Frissítjük a súlyvektorokat a gradiens alapján**

**for (int j = 0; j < s->featureSize; ++j) {**

**d->weight[j] += s->learningRate \* gradient[j];**

**}**

**}**

**return true;**

**}**

**Batch Training haszna**

A batch training, vagy kötegelt tanítás, számos előnnyel jár a gépi tanulásban:

1. **Stabilitás**: A kötegelt tanítás során az összes rendelkezésre álló adatot egyszerre használjuk fel a súlyok frissítéséhez, ami általában stabilabb tanulási folyamatot eredményez, mivel a tanítás során nem érzékeny a zajra vagy a kis változásokra a bemeneti adatokban.
2. **Hatékonyság**: A batch training hatékonyabb lehet a számítások szempontjából, különösen akkor, ha nagy adathalmazokkal dolgozunk. Az összes adat egyidejű feldolgozása lehetővé teszi a számítások párhuzamosítását és a hatékonyabb hardverkihasználást.
3. **Pontosság**: A batch training általában jobb eredményeket eredményezhet, mivel az összes rendelkezésre álló adatot figyelembe veszi a súlyok frissítése során, így a modell jobban képes a rendelkezésre álló adatokra illeszkedni.
4. **Regularizáció**: A batch training általában stabilabb és kevésbé érzékeny a túltanulásra (overfitting), mivel az összes adatot egyidejűleg használjuk fel a súlyok frissítéséhez, így a tanulási folyamat kevésbé zajos és kevésbé változékony.

Összességében a batch training nagyon hatékony lehet, különösen nagy adathalmazok esetén, mivel stabilabb tanulási folyamatot eredményez és jobb eredményeket biztosít, miközben hatékonyan használja ki a rendelkezésre álló számítási erőforrásokat.

**Konvergencia ellenőrzése**

A konvergencia ellenőrzése a gépi tanulásban az egyik fontos lépés annak biztosítására, hogy a tanítási folyamat lefutása során a modell valóban konvergáljon egy stabil állapotba, és ne vágja le a tanulási folyamatot túl korán, vagy ne folytassa a tanulást feleslegesen sokáig. Ennek az ellenőrzésére gyakran használnak különböző konvergencia kritériumokat, amelyek alapján eldönthető, hogy mikor érte el a modell a konvergenciát.

1. Számoljuk meg az aktuális iteráció során a gradiens változását.
2. Állítsuk le a tanítást, ha a gradiens változása elég kicsi, vagy ha elérjük a maximális iterációs számot.

**Kódrészlet**

**#define TOLERANCE 0.0001 // Konvergencia tolerancia**

**#define MAX\_ITERATIONS 1000 // Maximális iterációs szám**

**bool multiLogRTrain(Setup \*s, Data \*d) {**

**double prevGradientChange = INFINITY; // Az előző iterációban mért gradiens változás**

**for (int iteration = 0; iteration < s->maxIteration; ++iteration) {**

**// Inicializáljuk a gradiens vektort nullával minden iteráció előtt**

**double gradient[s->featureSize] = {0};**

**for (int i = 0; i < d->sampleCounter; ++i) {**

**double p = calcSigmoid(calcDotProduct(d->weight, d->x[i], s->featureSize));**

**for (int j = 0; j < s->featureSize; ++j) {**

**gradient[j] += (d->y[i] - p) \* d->x[i][j];**

**}**

**}**

**// Átlagoljuk a gradienseket a tanító adatok számával**

**for (int j = 0; j < s->featureSize; ++j) {**

**gradient[j] /= d->sampleCounter;**

**}**

**// Számítsuk meg a gradiens változását**

**double gradientChange = 0;**

**for (int j = 0; j < s->featureSize; ++j) {**

**gradientChange += fabs(gradient[j]);**

**}**

**// Ellenőrizzük a konvergenciát**

**if (gradientChange < TOLERANCE || gradientChange >= prevGradientChange) {**

**printf("Konvergencia elérése az iteráció: %d\n", iteration);**

**break;**

**}**

**// Frissítjük a súlyvektorokat a gradiens alapján**

**for (int j = 0; j < s->featureSize; ++j) {**

**d->weight[j] += s->learningRate \* gradient[j];**

**}**

**// Frissítjük a gradiens változását az aktuális iterációhoz**

**prevGradientChange = gradientChange;**

**}**

**return true;**

**}**

Ebben a módosított kódban az iteration változót a ciklus belsejében használjuk az aktuális iteráció számának nyomon követésére. A gradiens változását minden iterációban számoljuk, és összehasonlítjuk az előző iterációban mért gradiens változással. Ha a gradiens változása kisebb, mint a megadott tolerancia (TOLERANCE), vagy ha a gradiens változása nem csökken, akkor megállítjuk a tanítást, mivel a modell valószínűleg elérte a konvergenciát.

Ez a megközelítés segít abban, hogy megállítsuk a tanítást, amikor a modell konvergált egy stabil állapotba, és megakadályozzuk a felesleges iterációk futtatását a tanulási folyamat befejezése után.

**Más módszerek**

Igen, léteznek más módszerek is a konvergencia ellenőrzésére a gépi tanulásban. Néhány közülük:

1. **Költségfüggvény változása:** Figyelhetjük a tanulási folyamat során a költségfüggvény értékének változását. Amikor a költségfüggvény értéke már nem csökken jelentősen vagy stabilizálódik, akkor a modell valószínűleg konvergált.
2. **Validációs hiba ellenőrzése:** A validációs hibát figyelemmel kísérhetjük a tanulási folyamat során. Amikor a validációs hiba már nem csökken jelentősen vagy növekszik, akkor a modell konvergált lehet.
3. **Súlyok változásának figyelése:** Megfigyelhetjük a súlyok változását a tanulási folyamat során. Amikor a súlyok már nem változnak jelentősen vagy stabilizálódnak, akkor a modell konvergált lehet.
4. **Keresztvalidáció (cross-validation):** A keresztvalidáció segítségével több részre bontjuk a tanító adathalmazt, és külön-külön validációs adatokat használunk a tanulási folyamat során. Amikor a modell eredményei nem változnak jelentősen a keresztvalidáció során, akkor a konvergencia valószínű.
5. **Early stopping:** Az early stopping módszerben megállítjuk a tanítást, amikor a validációs hiba már nem csökken vagy növekszik egy bizonyos számú iteráció után.
6. **Gradient norma ellenőrzése:** A gradiens normáját figyelhetjük a tanulási folyamat során. Amikor a gradiens normája elég kicsi, vagy nem változik jelentősen, akkor a modell konvergálhat.

Ezek csak néhány példa a konvergencia ellenőrzésére használt módszerek közül. A választott módszer attól függ, hogy milyen jellegű a probléma és milyen információ áll rendelkezésre a tanulási folyamat során. Fontos azonban megérteni, hogy a konvergencia ellenőrzése kulcsfontosságú lépés a gépi tanulási modellek tanítása során annak érdekében, hogy megakadályozzuk a túltanulást és biztosítsuk a stabil tanulási folyamatot.

**Kimeneti információk javítása (növelése)**

A kimeneti információk javítása szintén fontos része az optimalizációnak a gépi tanulásban. Jó kimeneti információk segíthetnek abban, hogy követhető legyen a tanulási folyamat, megértsük a modell viselkedését, és könnyebben felismerjük a problémákat. Néhány módszer a kimeneti információk javítására:

1. **Részletesség növelése:** Tegyük részletesebbé a kimeneti információkat a tanulási folyamat során. Például, adjunk hozzá több részletet az iterációk számáról, **a költségfüggvény értékéről**, a validációs hibáról stb.
2. **Loggolás és vizualizáció:** Használjunk logolást és vizualizációt a tanulási folyamat során. Például, loggolhatjuk az iterációk számát, a költségfüggvény értékét, a validációs hibát stb., majd vizualizálhatjuk ezeket a változókat diagramokon vagy grafikonokon, hogy könnyebben követhető legyen a tanulási folyamat.
3. **Megfelelő formázás:** Gondoskodjunk arról, hogy a kimeneti információk jól formázottak és érthetőek legyenek. Például, használjunk érthető címkéket és formázást a kimeneti sorokon belül.
4. **Információk összehasonlítása:** Hasonlítsuk össze a különböző tanulási kísérletek eredményeit. Például, használjunk különböző hiperparamétereket vagy modelleket, majd hasonlítsuk össze azokat az eredményeket.
5. **Rendszerezés és szűrés:** Rendszerezzük és szűrjük az információkat, hogy csak a legfontosabb részleteket jelenítsük meg. Például, csak a fontos metrikákat és változókat jelenítsük meg a kimeneti információkban.
6. **Interaktív kimenetek:** Ha lehetséges, használjunk interaktív kimeneteket, amelyek lehetővé teszik a felhasználó számára a kimeneti információk vizsgálatát és manipulálását.

Ezek csak néhány példa a kimeneti információk javítására használt módszerek közül. Fontos megjegyezni, hogy a jó kimeneti információk segíthetnek abban, hogy hatékonyabb legyen a gépi tanulási folyamat, és könnyebben érthető legyen a modell viselkedése és teljesítménye.

**Kódrészlet**

**bool multiLogRTrain(Setup \*s, Data \*d) {**

**for (int iteration = 0; iteration < s->maxIteration; ++iteration) {**

**double cost = 0; // Költségfüggvény érték inicializálása**

**for (int i = 0; i < d->sampleCounter; ++i) {**

**double p = calcSigmoid(calcDotProduct(d->weight, d->x[i], s->featureSize));**

**for (int j = 0; j < s->featureSize; ++j) {**

**d->weight[j] += s->learningRate \* (d->y[i] - p) \* d->x[i][j];**

**}**

**// Költségfüggvény kiszámítása és hozzáadása a teljes költséghez**

**cost += (d->y[i] \* log(p)) + ((1 - d->y[i]) \* log(1 - p));**

**}**

**// Átlagolt költségfüggvény kiszámítása**

**cost /= d->sampleCounter;**

**// Loggolás az iteráció és a költségfüggvény értékével**

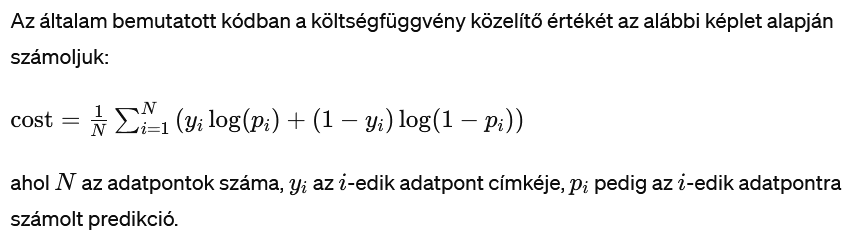
**printf("Iteráció: %d, Költségfüggvény értéke: %f\n", iteration, cost);**

**}**

**return true;**

**}**

Ebben a példában a multiLogRTrain függvény minden iterációja után logoljuk az iteráció számát és a költségfüggvény értékét. Ezáltal részletesebb információkat kapunk a tanulási folyamatról, amelyek segíthetnek abban, hogy könnyebben követhető legyen a folyamat és megértsük a modell viselkedését. A kódot tovább lehet fejleszteni például hozzáadva vizualizációkat vagy más információkat a logoláshoz a felhasználó számára.



Ez a költségfüggvény azonban csak egy közelítő érték, mivel az átlagos költségfüggvény kiszámítása az adatpontok számának átlagolásával történik. A pontos költségfüggvény értékének kiszámításához az összes adatpont címkéjének és a modell predikcióinak ismeretében kellene számítani. A költségfüggvény pontos értékének kiszámítása azonban nem szükséges a tanulási folyamat során, hanem a tanulási folyamat ellenőrzéséhez és az algoritmus megfelelő működésének biztosításához elegendő egy közelítő érték meghatározása.

**Inicializációs startégiák**

Az súlyok inicializációjának stratégiája kulcsfontosságú a gépi tanulási algoritmusok hatékonysága és konvergenciája szempontjából. Néhány gyakori súlyok inicializációs stratégia közé tartozik a következők:

1. **Zérus inicializáció**: Az súlyokat nullára inicializáljuk. Ez azonban általában nem a legjobb megoldás, mert az összes súly ugyanaz lesz, és a neurális hálózat nem fogja tudni megtanulni a bemeneti adatokból.
2. **Véletlen inicializáció**: Az súlyokat véletlenszerűen inicializáljuk egy adott eloszlásból, például egyenletes vagy normális eloszlásból. Ez általában jobb eredményeket eredményez, mert így a súlyok nem kezdődnek ugyanabban a pontban, és a tanulási folyamatnak több lehetősége van a konvergenciára.
3. **Xavier inicializáció**: Az súlyokat az előző réteg neuronjainak számával arányosan inicializáljuk. Ez a stratégia megpróbálja optimalizálni az aktivációk szórását és a gradiensek nagyságát, ami elősegítheti a gyorsabb konvergenciát.
4. **He inicializáció**: Hasonlóan működik, mint a Xavier inicializáció, de ez a módszer az előző réteg neuronjainak számával arányosan skálázza az súlyokat, és az aktivációk szórását egy kicsit eltérően kezeli.

A megfelelő inicializációs stratégia kiválasztása attól függ, hogy milyen típusú hálózatot használunk, és milyen a probléma jellege. Általában a véletlen inicializáció jó kiindulópont lehet, és később finomhangolhatjuk más módszerekkel, például a Xavier vagy a He inicializációval. A választott inicializációs stratégia befolyásolhatja a tanulási sebességet és a modell teljesítményét is.

**Kódrészlet**

**// Box-Muller transzformáció normális eloszlású véletlen számok generálásához**

**double generateRandomNormal() {**

**double u1 = rand() / (RAND\_MAX + 1.0);**

**double u2 = rand() / (RAND\_MAX + 1.0);**

**return sqrt(-2 \* log(u1)) \* cos(2 \* M\_PI \* u2);**

**}**

**// Súlyok véletlenszerű inicializációja normális eloszlásból**

**void initializeWeights(double \*weights, int size) {**

**srand(time(NULL)); // inicializáljuk a random generátort az aktuális idő függvényében**

**for (int i = 0; i < size; ++i) {**

**weights[i] = generateRandomNormal();**

**}**

**}**