Kapitola 21

Státnice I3: Metody strojového učení

21.1 Úvod

Formálně se strojové učení definuje následovně: Počítač se učí řešit úlohu třídy T se zkušeností E a úspěšností měřenou kritériem P, jestliže se jeho úspěšnost se vzrůstající zkušeností zvyšuje.

Př.: T = parsing, E = treebank, P počet správně určených hran.

Je třeba určit druh znalostí, které se bude počítač učit, a jejich reprezentaci a navrhnout konkrétní algoritmus učení. Typicky se jedná o aproximaci nějaké cílové funkce, která realizuje úkol, jenž chceme počítač naučit (reálná funkce – regrese, diskrétní – klasifikace).

Zkušenost může být přímá (zápis pravidel) a nepřímá ("dobré" postupy na základě trénovacích dat). Učení může být supervised (s učitelem, kdy E = trénovací data), semi-supervised (systém generuje postupy a něco je známkuje) nebo neřízené.

21.2 Učení založené na konceptu

Učení se konceptů lze definovat jako získání vymezení nějakého konceptu na základě vzorku pozitivních a negativních trénovacích příkladů (instancí) – trénovacích dat $D = \langle x_i, y_i \rangle_{i=1}^{n}$, kde $X = x_1, x_2, \ldots x_n$ jsou atributy a $y_1 \ldots y_n \in Y = \{0, 1\}$ je klasifikace instancí. Snažíme se najít cílovou funkci $c : X \to Y, c(x_i) = y_i$. Hledáme ji nad prostorem hypotéz (modelem) H; tj. hledáme $h \in H : h(x_i) = c(x_i)\ \forall i \in 1, \ldots n$.

Učení se konceptů můžeme chápat jako vyhledávání. V množině hypotéz se nachází prostor možností (version space) – podmnožina hypotéz konzistentních s trénovacími daty (množina cílových funkcí, které klasifikují trénovací data správně) $VS_{H,D}$ a v této podmnožině se někde nachází optimální hypotéza. Pomocí strojového učení se jí přibližujeme, od startovací hypotézy se postupně dostaneme do něčeho, co je konzistentní s trénovacími daty, ale ne nutně k optimu.

Hledáme takovou hypotézu, která by odpovídala cílové funkci na všech trénovacích příkladech: $h \in H : \forall x \in X : h(x) = c(x)$. Hypotéza, která je "vhodně dobrou" aproximací cílové funkce nad trénovacími daty, bude i "vhodně dobrou" aproximací nad ostatními daty.

Uspořádání hypotéz

Zvolme si reprezentaci hypotéz jako n-tici "povolených" hodnot proměnných z trénovacích příkladů, přičemž pro hypotézu může být hodnotou:

- Specifická hodnota dané proměnné (tj. povolená je jedna hodnota)
- "?" (tj. na hodnotě této proměnné nezáleží, povolené je všechno)
- "$\emptyset$" (tj. nevyhovuje žádná hodnota této proměnné)

Pokud instance x vyhovuje všemi svými hodnotami proměnných povoleným hodnotám pro hypotézu h, potom $h(x) = 1$, tj. hypotéza h klasifikuje x jako pozitivní příklad.

Potom nejobecnější hypotéza je taková, která cokoliv klasifikuje jako pozitivní případ, tj. $(?, ?, ?, \ldots, ?)$ a nejspecifičtější hypotéza taková, která vše klasifikuje jako negativní, tj. $(\emptyset, \emptyset, \ldots, \emptyset)$.

Pro prohledávání prostoru hypotéz máme vodítko: uspořádání podle specifičnosti.

- Hypotéza $h \in H$ je obecnější než hypotéza $j \in H$ ($h \geq_g j$), právě když $\forall x \in X : j(x) = 1 \Rightarrow h(x) = 1$ (můžeme říct, že j je specifičtější než h). Relace $\geq_g$ je částečné uspořádání nad prostorem H.
- Hypotéza $h \in H$ je striktně obecnější než $j \in H$ ($h >_g j$), pokud h je obecnější než j, ale j není obecnější než h.

Ještě definujme hranici obecnosti (general boundary) G vzhledem k H a D jako množinu nejobecnějších hypotéz z H konzistentních s D. Proti tomu Hranice specifičnosti (specific boundary) S vzhledem k H a D je množina nejspecifičtějších hypotéz z H konzistentních s D.

Algoritmus Find-S

Nalezne nejspecifičtější hypotézu, která ještě vyhovuje trénovacím datům.

1. Vezmi nejspecifičtější hypotézu $h \in H$
2. $\forall x$ pozitivní trénovací příklad a $\forall a_i$ atribut reprezentace h: pokud hodnota a_i u x neodpovídá h, nahraď hodnotu a_i v h nejbližší obecnou hodnotou, která odpovídá x
3. Po projití všech pozitivních příkladů vrať upravené h

Find-S vrátí nejspecifičtější hypotézu konzistentní s pozitivními trénovacími příklady. Za předpokladů, že prohledávaný soubor hypotéz obsahuje cílovou funkci c a trénovací data neobsahují chyby, bude nalezená hypotéza konzistentní i s negativními trénovacími příklady, tj. s celými trénovacími daty.

Je-li víc správných hypotéz, nalezneme takto jen jednu z nich, navíc neodhalíme nekonzistenci trénovacích dat. Nemusí existovat ani jen jedna nejspecifičtější hypotéza.

Problém nalezení všech konzistentních hypotéz řeší algoritmus Candidate-Elimination.

Algoritmus Candidate-Elimination

Algoritmus pracuje s prostorem možností, určeným hranicí obecnosti a hranicí specifičnosti. To je možné proto, že platí následující věta:

Věta o reprezentaci prostoru možností: $VS_{H,D} = \{h \in H : \exists s \in S, \exists g \in G : (g \geq_g h \geq_g s)\}$.

Postup algoritmu:

1. Do G, S přiřaď množinu nejobecnějších, resp. nejspecifičtějších hypotéz z H
2. Postupuj přes trénovací příklady $d \in D$:
3. Je-li příklad $d \in D$ pozitivní:
   (a) Odstraň z G hypotézy nekonzistentní s d.
   (b) Každou s d nekonzistentní hypotézu z S odstraň a přidej do S její minimální zobecnění konzistentní s d, pro které existuje nějaká obecnější hypotéza v G. Odstraň pak z S hypotézy, které jsou obecnější než jiné hypotézy v S.
4. Je-li příklad $d \in D$ negativní:
   (a) Odstraň z S hypotézy nekonzistentní s d.
   (b) Každou s d nekonzistentní hypotézu z G odstraň a přidej do G její minimální specializaci konzistentní s d, pro kterou existuje nějaká specifičtější hypotéza v S. Odstraň pak z G hypotézy, které jsou specifičtější než jiné hypotézy v G.
5. Na konci vrať G, S jako reprezentaci prostoru možností.

Induktivní předpoklad

Různé algoritmy různým způsobem zobecňují údaje získané z trénovacích dat, aby byly schopny klasifikovat i dříve neviděné příklady. Pro zachycení této strategie definujme:

- Je-li algoritmus L natrénovaný na datech D_c schopný klasifikovat instanci x_i, je tato klasifikace (ozn. $L(x_i, D_c)$) induktivně odvoditelná z oné instance a trénovacích dat. Píšeme $(D_c \wedge x_i) \triangleright L(x_i, D_c)$. To ale ještě neznamená, že je z instance a dat dokazatelná – algoritmus v sobě může obsahovat navíc induktivní předpoklad.
- Induktivní předpoklad (inductive bias) algoritmu L je minimální množina tvrzení B taková, že pro libovolnou cílovou funkci c definovanou nad instancemi X a odp. trénovací data D_c platí: $\forall x_i \in X : (B \wedge D_c \wedge x_i) \vdash L(x_i, D_c)$, tj. klasifikace x_i algoritmem L natrénovaným na D_c je dokazatelná z instance, trénovacích dat a induktivního předpokladu.

Algoritmus Candidate-Elimination předpokládá, že cílovou funkci lze popsat pomocí naší zvolené reprezentace, tj. nachází se v našem prostoru hypotéz: $B = \{c \in H\}$. Bude-li tento předpoklad splněn, pak najde optimální hypotézu. Je jasné, že tento předpoklad splněn nebude, když např. dvě konkrétní hodnoty nějakého atributu budou určovat pozitivní případy a zbytek negativní. Find-S oproti Candidate-Elimination ještě navíc předpokládá, že všechny příklady jsou negativní, pokud jejich protějšek nepřináší novou znalost.

Kdybychom zvolili reprezentaci hypotéz disjunkcemi všech možných hodnot atributů, tj. nechali induktivní předpoklad prázdný, Candidate-Elimination nebude schopný zobecňovat za hranici trénovacích příkladů (nalezená hypotéza bude čistě disjunkcí všech trénovacích příkladů).

Pro algoritmy učení je tedy třeba uvažovat, jak silné mají indukční předpoklady. Ty jsou někdy velice striktní, jako v případě Candidate-Elimination, někdy znamenají jen preferenci určitého typu hypotéz (např. co nejkratších, podle kritéria Occamovy břitvy).

21.3 Rozhodovací stromy

Rozhodovací stromy jsou induktivní algoritmy, vytvořené ke klasifikaci instancí, které popisují pomocí "disjunkce konjunkcí". Můžeme je použít, když instance jsou popsány atributy a hodnotami a cílová funkce je diskrétní. V trénovacích datech můžou být i chyby nebo nevyplněné atributy.

Cílová funkce je tu reprezentována právě rozhodovacím stromem (sekvencí rozhodnutí `if-then-else`), klasifikace se provádí průchodem stromu od kořene k listům, přičemž se v každém uzlu testuje hodnota jednoho atributu instance a na jejím základě se určí další směr (zvolí se hrana). Každý list určuje klasifikaci instance.

ID3 algoritmus

ID3 je algoritmus, který vytváří rozhodovací stromy na základě trénovacích dat. Postupuje top-down – konstruuje strom od kořene. V každém uzlu si klade otázku: Který atribut je nejlepší pro tento uzel? a řeší ji statistickým testem na základě maximální entropie. Počet větví vedoucích z uzlu je počet možných hodnot atributu, takže ho známe hned po vytvoření uzlu. Trénovací data jsou při vytváření dělena podle atributů v daném uzlu.

Pracuje rekurzivně:

1. Vyřeší singulární případy:
   (a) Všechna trénovací data jsou pozitivní případy / negativní případy: vytvoří jednouzlové stromy s (+) nebo (-).
   (b) Nemáme (už) žádné atributy: vytvoří jednouzlový strom s (nejčastější hodnota cíl. funkce v relevantní části trénovacích dat).
2. Vybere atribut A, který nejlépe klasifikuje trénovací data a:
   (a) Pro každou jeho možnou hodnotu udělá poduzel, rozdělí trénovací data podle těchto hodnot.
   (b) Máme-li pro danou hodnotu atributu trénovací data, volá se rekurzivně (na danou část trénovacích dat a atributy s vynechaným A).
   (c) Pokud nejsou trénovací data, vloží list s (nejčastější hodnota cíl. funkce v celých trénovacích datech).
3. Vrátí výsledný (pod)strom.

Prochází se celý hypothesis space, ale hladovým způsobem bez backtrackingu. Výstupem je 1 rozhodovací strom, tj. jedna hypotéza.

Výběr nejlepšího atributu probíhá na základě informačního zisku. Který atribut má nejvyšší informační zisk (míra očekávaného snížení entropie, vlastně vzájemná informace cílové klasifikace a atributu), ten je vybrán. Platí: $G(D, A) = H(D) - \sum_{v \in \text{values}(A)} (\frac{|D_v|}{|D|} \cdot H(D_v))$, kde:

- $G(D, A)$ je informační zisk pro data D a atribut A,
- $H(D)$ je entropie dat D vzhledem k cílové funkci,
- D_v jsou data, která mají hodnotu atributu A rovnou v.

ID3 prochází úplný prostor hypotéz tím, že začíná od nejjednodušší a postupně ji zesložiťuje, netestuje ale zdaleka všechny možnosti. Na rozdíl od předchozích algoritmů v každém kroku uvažuje celá trénovací data. Induktivní předpoklad ID3 je (přibližně) preference kratších stromů.

Vylepšení ID3 (zhruba odpovídá algoritmu C4.5)

Jedním z možných problémů je overfitting, tj. přílišné přizpůsobení se trénovacím datům. Řekneme, že hypotéza $h \in H$ overfituje trénovací data, pokud existuje hypotéza $h' \in H$ taková, že h má menší chybu než h' na trénovacích datech, ale na celé distribuci instancí je to naopak.

Vyvarovat se overfittingu lze prořezáváním (pruning) – nahrazením některých podstromů listem s nejčastější hodnotou cílové funkce na trénovacích datech. Lze provádět buď rovnou zastavením tvorby stromu dřív, než dosáhne ideální klasifikace trénovacích dat, nebo zpětně strom zjednodušit za použití heldout dat pro validaci při trénování (reduced error pruning).

Složitější je strategie rule-post-pruning spočívá v následujícím postupu:

1. Vytvoření rozhodovacího stromu.
2. Jeho převedení na pravidla (počet pravidel = počet listů), kdy 1 pravidlo je 1 cesta od listu ke kořeni.
3. Pro každé pravidlo zahodit ty podmínky, jejichž vyhozením se přesnost pravidla zlepší.
4. Pravidla setřídit podle přesnosti na trénovacích datech, reálná data klasifikovat použitím pravidel v tomto pořadí.

Proti prořezávání nad stromem nemusíme vyhazovat jen z konce stromu, máme jemnější rozlišení.

Je-li některý atribut A spojitý (ale cílové ohodnocení stále diskrétní), lze ho převést na diskrétní nalezením jednoho předělu, pro který dostaneme nejvyšší informační zisk.

Má-li atribut více hodnot (např. nějaké datum), informační zisk ho bude preferovat, což často vede k vytvoření stromu hloubky 1, který funguje jen na trénovací data. Musíme pak změnit test výběru atributů, místo zisku informace používáme např. ziskový poměr (gain ratio):

- Rozštěpení informace (split information): $SI(D,A) = -\sum_{v \in \text{values}(A)} \frac{|D_v|}{|D|} \log_2(\frac{|D_v|}{|D|})$
- Ziskový poměr: $GR(D,A) = \frac{G(D,A)}{SI(D,A)}$, kde $G(D,A)$ je informační zisk pro trénovací data D a atribut A

Jsou i jiné alternativní způsoby výběru nejlepšího atributu, např. distance-based measure, kdy se vybírá atribut, který dělí data co nejrovnoměrněji podle nějaké metriky (směrodatná odchylka, Gini koeficient).

Neznáme-li hodnotu některého z atributů, můžeme mu přiřadit nejčastější hodnotu, nebo rozhodování rozdělit podle všech možných hodnot a na konci udělat průměr výsledků vážený podle jejich pravděpodobnosti výskytu.

Pokud jsou některé atributy důležitější než jiné, zavádí se cena atributu (cost of the attribute) a jiné kritérium výběru atributu.

21.4 Neuronové sítě

Vznik neuronových sítí je motivován biologicky, funkcí mozku jako spojení mnoha neuronů. Umělé neuronové sítě ale jsou poněkud jednodušší – jedná se o propojenou množinu jednotek (neuronů), z nichž každá z několika reálných hodnot na vstupu vydá jednu reálnou hodnotu na výsupu. Živé neurony proti tomu vydávají mnohem složitější elektrické signály.

Neuronová síť bývá typicky zapojená jako orientovaný acyklický graf, ale nemusí to být nutné. Tady se ale omezíme jen na takové sítě, protože na ně známe jednoduchý algoritmus trénování – backpropagation.

Neuronové sítě jsou vhodné pro složitá a potenciálně zašuměná vstupní data, jako např. vstupy z kamer a mikrofonů (rozpoznávání řeči, rozpoznávání obličejů apod.). Jsou ale použitélná i pro data se symbolickou reprezentací, kde dosahují výsledků srovnatelných s rozhodovacími stromy. Cílová funkce může být reálná i diskrétní a může jít i o vektor hodnot. Trénování zpravidla trvá dlouho, ale klasifikace nových instancí je velmi rychlá. Natrénované hodnoty se často nedají dobře interpretovat, ale fungují.

Perceptron

Perceptron je vlastně jeden druh neuronu často používaného v neuronových sítích. Jde o jednotku, která ze vstupu $x_1, \ldots, x_n$ spočítá hodnotu výstupu $o(x_1, \ldots, x_n)$. Podle druhu výpočtu rozlišujeme :

- Lineární perceptron, který počítá lineární kombinaci vstupů na základě natrénovaných vah – $w_0 + w_1 x_1 + \ldots w_n x_n$. Můžeme přidat $x_0 = 1$ a zapsat jeho výpočet jako $o(\vec{x}) = \vec{w} \cdot \vec{x}$.
- (Základní) prahový perceptron, který počítá funkci $o(\vec{x}) = sgn(\vec{w} \cdot \vec{x})$ a vrací diskrétní hodnoty $-1, 1$.
- Sigmoidovou jednotku, který používá sigmoidovou (logistickou) funkci $\sigma(y) = \frac{1}{1+e^{-y}}$ a počítá $o(\vec{x}) = \sigma(\vec{w} \cdot \vec{x})$. I když funguje na stejném principu jako dva předchozí a vlastně dává podobné výsledky jako prahový perceptron (její obor hodnot je interval $[-1, 1]$), podle jména se mezi perceptrony nepočítá.

Jak je vidět, prostor hypotéz perceptronu je množina všech možných reálných vah ($H = \{\vec{w} | \vec{w} \in \mathbb{R}^{n+1}\}$).

Uvažujme nyní prahový perceptron. Ten se vlastně dá považovat za reprezentaci "rozhodující nadroviny" v n-dimenzionálním prostoru instancí (to jsou vlastně body v n-rozměrném prostoru). Vrací 1 pro body, které leží na jedné straně nadroviny a -1 pro ty na druhé straně. To ale znamená, že správně umí klasifikovat jen problémy, kde opravdu pozitivní a negativní příklady jdou oddělit nadrovinou. Takovým problémům (a množině příkladů) se říká lineárně separovatelné.

Př. Perceptron se umí naučit booleovské funkce AND a OR, ale neumí XOR, protože není lineárně separovatelná.

Následují dvě možnosti, jak perceptron natrénovat. Další alternativou by bylo i lineární programování, ale to funguje jen v lineárně separovatelném případě a navíc se nedá použít jako základ pro složitější sítě.

Perceptron Training Rule

Nejjednodušším způsobem, jak natrénovat váhy (prahového) perceptronu, je použít perceptron training rule – začneme s náhodnými vahami a iterativně zkoušíme klasifikovat trénovací příklady a jakmile dojde ke špatné klasifikaci, váhy změníme. Formálně v každém kroku provedeme:

$$w_i := w_i + \Delta w_i, \text{ kde } \Delta w_i = \eta(t - o)x_i$$

Hodnota o je výstup perceptronu a t správná klasifikace. Hodnotě η říkáme learning rate – určuje, jak ostře nové příklady ovlivňují původní hodnoty vah. Většinou je nastavená na nějakou malou hodnotu, můžeme ji i snižovat s počtem iterací.

V případě, že trénovací příklady jsou lineárně separovatelné, takovýto algoritmus po konečném počtu iterací dojde k vektoru vah, se kterým perceptron správně klasifikuje všechny trénovací příklady.

Gradient Descent (Delta Rule)

Jiný způsob je gradient descent (klesání podle gradientu). Uvažujme nyní lineární perceptron, abychom mohli snadno derivovat. Stanovíme si funkci, která hodnotí trénovací chybu hypotézy (tj. vektoru vah) $E(\vec{w})$ a iterativně budeme hledat, kterým směrem jde gradient a vydáme se přesně opačným (tj. proti směru největšího růstu funkce). Tak dojdeme pro "hezké funkce" do místa, kde je gradient nulový a tedy je tam (bohužel jen lokální) minimum chybové funkce. Pro lineárně separovatelné trénovací instance bychom se měli dostat k nulové chybě, ale i pro neseparovatelné dostaneme (snad) to nejlepší, co můžeme. Velmi vhodnou funkcí trénovací chyby je:

$E(\vec{w}) = \frac{1}{2}\sum_{d\in D}(t_d - o_d)^2$, kde D jsou trénovací data a t_d, o_d skutečná hodnota a výstup perceptronu jako v předchozí sekci.

Potom metoda gradient descent postupuje stejně iterativně jako v předchozím případě:

$w_i := w_i + \Delta w_i$, kde $\Delta w_i = -\eta \nabla E(\vec{w})$ (bereme záporný gradient, takže jdeme opačným směrem)

Pokud si upravovací pravidlo rozložíme na jednotlivé složky, můžeme vypočítat:

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} \text{ a } \frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i}\frac{1}{2}\sum_{d\in D}(t_d - o_d)^2 = \frac{1}{2}\sum_{d\in D} 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \vec{w}\cdot\vec{x_d}) = \sum_{d\in D}(t_d - o_d)(-x_{id})$$

x_{id} značí vstup x_i v trénovacím příkladu D. Dosazení do původní rovnice dává pravidlo:

$$\Delta w_i = \eta \sum_{d\in D}(t_d - o_d)x_{id}$$

Pro zrychlení výpočtu a snížení rizika "zamrznutí" v lokálním minimu se používá stochastická aproximace (inkrementální nebo stochastický gradient descent), kdy váhy upravuji ne podle gradientu na celých trénovacích datech, ale jen s přihlédnutím k jednomu trénovacímu příkladu :

$$\Delta w_i = \eta(t - o)x_i$$

Protože prahový perceptron čistě vrací znaménko výstupu lineárního perceptronu, je možné perceptron natrénovat touto metodou jako lineární a pak použít prahy (ale pozor, neminimalizujeme počet špatně klasifikovaných příkladů prahového perceptronu, ale sumu chyby lineárního perceptronu, což nemusí vést ke stejným výsledkům).

Neuronová síť

Perceptrony se dají spojovat do sítí a potom jsou schopny se naučit i lineárně neseparovatelné problémy (to ale neplatí o lineárních perceptronech – lineární kombinace lineárních kombinací je pořád lineární kombinace). Protože síťovat lineární perceptrony tedy nemá smysl a prahové perceptrony používají funkci, která se nedá dobře derivovat, použijeme sigmoidové jednotky. Derivace sigmoidové funkce je totiž $\sigma'(y) = \sigma(y)(1-\sigma(y))$. Někdy se používá i varianta sigmoidové funkce s $e^{-k\cdot y}$ nebo funkce tanh.

Takovéto jednotky můžeme tedy zasadit do sítě ve tvaru acyklického grafu. Trénování vah ale pak bude složitější, používá se na to algoritmus backpropagation.

Algoritmus backpropagation

Máme-li pevně dánu množinu jednotek a jejich spojení, která má libovolný počet vstupů a výstupů, můžeme natrénovat váhy každého vstupu pomocí algoritmu backpropagation. Používáme přitom stejných myšlenek jako u gradient descent, když se snažíme minimalizovat chybovou funkci:

$E(\vec{w}) = \frac{1}{2}\sum_{d\in D}\sum_{k\in \mathrm{Out}}(t_{kd} - o_{kd})^2$ – jde o stejnou funkci jako předtím, jen upravenou pro množinu výstupů Out.

Stejně jako předtím prohlédáváme prostor hypotéz definovaný všemi možnými vektory vah a máme garantovanou konvergenci k lokálnímu minimu chybové funkce. Algoritmus vypadá (ve stochastické aproximaci) následovně (x_{ji} je vstup z i-té jednotky do j-té a w_{ji} odpovídající váha):

1. Inicializuje všechny váhy malými náhodnými čísly
2. Dokud není splněna nějaká podmínka konvergence, opakuje (pro jednotlivé trénovací příklady $\langle\vec{x}, \vec{t}\rangle$, může je procházet i víckrát):
   (a) Propaguje vstup dopředu sítí – vloží $\vec{x}$ na vstup a spočítá výstup o_u z každé jednotky u v síti

(b) Propaguje chyby a úpravy vektorů vah zpátky:

i. Pro každou výstupní jednotku k spočte chybu $\delta_k = o_k(1-o_k)(t_k-o_k)$

ii. Pro každou vnitřní (skrytou) jednotku h spočte chybu $\delta_h = o_h(1-o_h)\sum_{j\in\text{Downstream}(h)} w_{jh}\delta_j$

iii. Upraví všechny váhy v síti $w_{ji} = w_{ji} + \Delta w_{ji}$, kde $\Delta w_{ji} = \eta\delta_j x_{ji}$

Downstream(h) značí všechny jednotky, jejichž vstupy jsou přímo napojené na výstup jednotky u. Pokud jednotky nejsou spojené cyklicky, lze najít takové pořadí jednotek, ve kterém lze spočítat hodnotu δ_h pro všechny.

Pro upravovací pravidla vycházíme z gradientové metody, která by pro složitější síť měla ve stochastické variantě být $\Delta w_{ji} = -\eta\frac{\partial E_d}{\partial w_{ji}}$ pro naši chybovou funkci definovanou výstupem pro instanci d.

- Protože w_{ji} ovlivňuje zbytek sítě jen prostřednictvím uzlu j, můžeme použít řetízkové pravidlo a $\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial \text{In}(j)}\frac{\partial \text{In}(j)}{\partial w_{ji}} = \frac{\partial E_d}{\partial \text{In}(j)}x_{ji}$, kde $\text{In}(j) = \sum_{i\in\text{Upstream}(j)} w_{ji}x_{ji}$ a Upstream(j) jsou všechny jednotky, jejichž výstupy jsou napojené na vstup j-té jednotky. Definujeme $\delta_j = -\frac{\partial E_d}{\partial \text{In}(j)}$.

- Pro výstupní jednotky z podobné úvahy vyplyne, že $\frac{\partial E_d}{\partial \text{In}(j)} = \frac{\partial E_d}{\partial o_j}\frac{\partial o_j}{\partial \text{In}(j)}$ a z toho – z definice chybové funkce (všechny části její sumy krom j-tého výstupu budou mít nulové derivace) a z definice sigmoidové funkce ($o_j = \sigma(\text{In}(j))$) – spočteme $\delta_j = (t_j - o_j)o_j(1-o_j)$.

- Podobně i pro skryté jednotky $\frac{\partial E_d}{\partial \text{In}(j)} = \sum_{k\in\text{Downstream}(j)}\frac{\partial E_d}{\partial \text{In}(k)}\frac{\partial \text{In}(k)}{\partial o_j}\frac{\partial o_j}{\partial \text{In}(j)}$, protože skrytá jednotka ovlivňuje chybu jen přes své přímé výstupy; dostáváme rekurentní vzorec. Protože jediný vstup do k, který se mění v závislosti na j, je $w_{kj}x_{kj}$, ale vlastně $x_{kj} = o_j$, je $\frac{\partial \text{In}(k)}{\partial o_j} = w_{kj}$. Poslední člen řetízku spočteme stejně jako pro výstupní jednotky a z toho $\delta_j = o_j(1-o_j)\sum_{k\in\text{Downstream}(j)}\delta_k w_{kj}$.

Vlastnosti neuronových sítí

Induktivní předpoklad algoritmu backpropagation se dá stanovit přesně jen těžko, intuitivně jde ale o cca lineární interpolaci mezi trénovacími příklady jako body n-rozměrného prostoru.

Garance pouze lokálního minima v praxi až tolik nevadí – díky tomu, že máme víc vah, je také méně pravděpodobné, že bude lokální minimum ve všech rozměrech vektoru. Inicializujeme-li malými hodnotami vah, bude výstup díky vlastnostem sigmoidové funkce přibližně lineární kombinace, až se zvýšením vah dostaneme výraznou nelinearitu.

Varianta algoritmu ještě přidává moment (α) a úprava vah částečně závisí na předchozí iteraci, čímž se snaží předejít zamrznutí v lokálním minimu (má pak určitou "setrvačnost"):

$$\Delta w_{ji}(n) = \eta\delta_j x_{ji} + \alpha\Delta w_{ji}(n-1)$$

Můžeme zkusit také natrénovat více sítí s lehce odlišnými počátečními vahami a vybrat si tu nejlepší.

Neuronová síť rozdělená do vrstev, v rámci nichž nejsou spoje, je schopná:

- S dvěma vrstvami přesně simulovat libovolnou boolovskou funkci (až na to, že počet potřebných jednotek může být až exponenciální v počtu vstupů)

- Spojité funkce lze aproximovat dvěma vrstvami do libovolné přesnosti, když první vrstva jsou sigmoidy a druhá lineární perceptrony.

- Libovolná funkce je aproximovatelná třema vrstvami – dvěma sigmoidovými a jednou lineární.

Konvergence a overfitting

Neuronová síť má velkou volnost v nastavování vah, proto může až příliš přesně aproximovat trénovací data, čímž se ztrácí použitelnost na data nová. Podmínku konvergence je třeba volit "rozumně":

- Samotné čekání, až změny vah budou dostatečně malé, většinou nestačí.

- Je vhodné např. penalizovat příliš velké váhy (po každé iteraci je trošku snižovat), čímž podporujeme "lineárnější" rozhodovací funkce.

- Velmi úspěšné je kromě trénování zároveň testovat síť na další datové množině a pamatovat si váhy, které na ní měly nejmenší chybu. Pokud se chyba na této množině výrazně zvýší oproti nejlepší hodnotě, trénování skončí a vezmeme váhy s nejmenší chybou na ladící množině.

- Předchozí metodu lze provádět i křížovou validací (rozdělím trénovací data na k množin a použiji v každém kroku $k-1$ z nich na trénování a zbylou na testování)

21.5 Učení založené na příkladech

Hlavní myšlenka učení založeného na příkladech (instance-based learning) zní: který trénovací příklad je nejbližší aktuální testovací instanci? V paměti vždy uchováváme všechny trénovací příklady, kdykoliv lze přidat další a pro každý testovací příklad v nich hledám ty nejpodobnější. Proto je nutné zavést míru závislosti příkladů. Lze např. použít euklidovskou míru $d(x,y) = \sqrt{\sum_{i=1}^{n}(a_i(x) - a_i(y))^2}$, ale není to nutné. Cílová funkce může být diskrétní (s def. oborem $V = \{v_1, \ldots, v_n\}$) nebo spojitá.

K-nearest neighbor

K-nearest neighbor nebo k-nn je algoritmus typu instance-based learning. Počítám s uloženými trénovacími daty E v paměti. Postup pro nějakou testovací instanci x_q:

- Je-li $x_q \in E$, potom mám přímo hodnocení.
- Jinak najdu k nejbližších $x_1, \ldots, x_k \in E$ a:
  - Pro diskrétní cílovou funkci $\hat{f}(x_q) = \text{argmax}_{v \in V} \sum_{i=1}^{k} \delta(v, f(x_i))$, kde $\delta = 1$ pro stejné hodnoty argumentů, a 0 pro různé (tj. přiřadím x_q takovou hodnotu, kterou má nejvíce z oněch k sousedů).
  - Pro spojitou cílovou funkci $\hat{f}(x_q) = \frac{\sum_{i=1}^{k} f(x_i)}{k}$, tj. udělám průměr hodnocení.

K-nn nikdy nepočítá explicitní obecnou hypotézu, pracuje jen v kontextu. Pro $k = 1$ de facto odpovídá Voronoi diagramu. Vylepšením může být vážení podle vzdáleností (distance-weighted k-nn), kde:

- V diskrétním případě $\hat{f}(x_q) = \text{argmax}_{v \in V} \sum_{i=1}^{k} w_i \cdot \delta(v, f(x_i))$, kde navíc $w_i = \frac{1}{d(x_q, x_i)^2}$.
- Ve spojitém případě $\hat{f}(x_q) = \frac{\sum_{i=1}^{k} w_i f(x_i)}{\sum_{i=1}^{k} w_i}$.

Díky vážení můžeme takhle jít dokonce přes celá trénovací data (globální (Shepardova) metoda).

Algoritmus je robustní vzhledem k šumu v trénovacích datech. Problémem je pomalý výpočet vzdáleností ke všem trénovacím datům, pro zrychlení se vytváří indexy, pomocí k-d stromů. Problém je taky, když mám hodně atributů a relevantních jen pár. Pak je metrika vzdáleností zavádějící – i když se shodují důležité atributy, mohou být od sebe instance v ostatních atributech "daleko". Řešením je váha atributů.

Lokálně vážená lineární regrese

Lokálně vážená lineární regrese (locally weighted linear regression) je konstrukce lokální aproximace cílové funkce na okolí kolem testovacího příkladu. Používá se často ve statistice. Používají se ale i jiné než lineární funkce. Definujeme:

- Regrese je aproximace reálné funkce.
- Reziduum je odchylka aproximace od reálné funkce, tj. $\hat{f}(x) - f(x)$.
- Jádro (kernel function) je funkce K vzdálenosti d za účelem získání váhy trénovacích příkladů $w_i = K(d(x_i, x_q))$. Často je to gaussovská funkce se střední hodnotou μ a rozptylem σ^2.

Aproximace hodnotící funkce se provádí ve tvaru: $\hat{f}(x) = w_0 + w_1 a_1(x) + \cdots + w_n a_n(x)$ (kde $a_i(x)$ je hodnota i-tého atributu příkladu). Úkolem je pak najít taková w_i, která minimalizují chybovou funkci. K tomu slouží metoda gradient descent. Chybová funkce se volí tak, aby byla jen lokální, máme tu tři možnosti (z nichž ta poslední je nejvýhodnější). Nechť $NN_k(x_q)$ je množina k nejbližších sousedů x_q:

1. $E_1(x_q) = \frac{1}{2} \sum_{x \in NN_k(x_q)} (\hat{f}(x) - f(x))^2$
2. $E_2(x_q) = \frac{1}{2} \sum_{x \in D} (\hat{f}(x) - f(x))^2 \cdot K(d(x_q, x))$ – nejpřesnější, ale výpočetní složitost závisí na velikosti trénovacích dat $|D|$.
3. $E_3(x_q) = \frac{1}{2} \sum_{x \in NN_k(x_q)} (\hat{f}(x) - f(x))^2 \cdot K(d(x_q, x))$ – toto je vhodná aproximace druhého přístupu, výpočetní složitost závisí jen na k.

21.6 Vyhodnocování hypotéz

Hypotéza, kterou nám vydá algoritmus učení, nemusí být vhodná. Je proto potřeba umět hypotézy ověřovat a případně porovnávat – ostatně i některé algoritmy učení (jako rule-post-pruning u rozhodovacích stromů) v sobě porovnávání hypotéz přímo obsahují.

Statistická formalizace pro diskrétní hypotézy

Mějme množinu všech možných datových instancí X, kterou budeme považovat za náhodnou veličinu s rozdělením $\mathcal{D}$. Nad instancemi X je definovaná cílová funkce f, kterou aproximujeme hypotézami.

Pro hypotézu h definujme:

- Chybu na datech (sample error) $\text{error}_S(h)$ jako část vzorku dat S, které h ohodnotí špatně, tj. $\text{error}_S(h) = r/n$ pro r špatně ohodnocených instací z n celkem.
- Skutečnou chybu (true error) $\text{error}_{\mathcal{D}}(h)$ jako pravděpodobnost p, že h ohodnotí špatně náhodně vybranou instanci z $\mathcal{D}$.

Chceme znát skutečnou chybu, ale můžeme ji jen odhadnout za pomoci chyby na datech. Skutečná chyba je vlastně náhodná veličina z alternativního rozdělení s parameterem p. Na základě náhodného výběru (nezávislých stejně rozdělených náhodných veličin) o velikosti n pak tento parametr odhadujeme. Náš odhad, funkce náhodného výběru, je další náhodná veličina. U takového odhadu je třeba dbát na:

- Vychýlení (bias) – je třeba, aby střední hodnota našeho odhadu se rovnala střední hodnotě původní náhodné veličiny (tj. abychom měli nulový bias).
- Rozptyl – čím menší rozptyl, tím lepší odhad, protože tak dostáváme menší očekávanou odchylku od skutečné hodnoty parametru.

Podmínka nezávislosti je důležitá, jinak nelze provést nevychýlený odhad (tj. netestovat na trénovacích datech, protože nejsou nezávislá!).

Vezmeme-li n nezávislých pokusů, které s pravděpodobností p dopadnou špatně, potom počet r pokusů, které selhaly, je binomicky rozdělená náhodná veličina. Odhad p jako r/n je nevychýlený, protože střední hodnota binomicky rozdělené veličiny je $E(r) = np$, tj. $E(r/n) = p$. Víme, že $\text{Var}(r) = np(1-p)$. Rozptyl odhadu $\text{error}_S(h) = r/n$ potom je:

$$\text{Var}(\text{error}_s(h)) = \frac{\text{Var}(r)}{n^2} = \frac{p(1-p)}{n} \approx \frac{\text{error}_S(h)(1-\text{error}_S(h))}{n}$$

(Oboustranný) interval spolehlivosti (pro danou pravděpodobnost) určuje interval, ve kterém se podle našeho odhadu bude s danou pravděpodobností N nacházet skutečná hodnota neznámého parametru – je to interval, ve kterém se na obě strany od našeho odhadu nachází dané procento pravděpodobnostní masy z rozdělení, které má náš odhad. Protože to je pro binomické rozdělení příliš namáhavé určit, pomůžeme si za přepokladu, že náš vzorek je dostatečně velký ($n > 30$), podle Centrální limitní věty aproximací normálním rozdělením a platí:

$$P(\text{error}_{\mathcal{D}}(h) \in \left(\text{error}_S(h) \pm z_N \sqrt{\tfrac{\text{error}_S(h)(1-\text{error}_S(h))}{n}}\right)) = N, \text{ kde } z_N \text{ jsou kvantily normálního rozdělení.}$$

Typicky se používá $N = 0.95$. Je možné používat i jednostranné intervaly spolehlivosti, např. pro ověření, jaká je pravděpodobnost, že chyba bude větší než určitá konstanta.

Rozdíl chyby dvou hypotéz

Pro porovnání dvou různých hypotéz (např. při prořezávání rozhodovacích stromů) se vlastně odhaduje rozdíl jejich chyb $d \equiv \text{error}_{\mathcal{D}}(h_1) - \text{error}_{\mathcal{D}}(h_2)$. To můžeme provést pomocí dvou nezávislých náhodných výběrů S_1, S_2 a $\hat{d} \equiv \text{error}_{S_1}(h_1) - \text{error}_{S_2}(h_2)$. $\hat{d}$ je nevychýlený odhad d.

Pokud uvažujeme dost velké náhodné výběry S_1, S_2, můžeme považovat odhady $\text{error}_{S_1}(h_1)$ a $\text{error}_{S_2}(h_2)$ za přibližně normálně rozdělené náhodné veličiny. Jejich rozdíl pak bude taky normálně rozdělený a roztpyl bude odpovídat součtu rozptylů, což nám dává i intervaly spolehlivosti. V typickém případě $S_1 = S_2$ a pak rozptyl typicky bude menší než tento odhad, ale pořád je možné s ním takto počítat.

Můžeme takto i spočítat, jaká je pravděpodobnost, že jedna z hypotéz dává větší chybu – je to $P(d > 0) = P(\hat{d} < E(\hat{d}) + \hat{d})$, což nám dává jednostranný interval a stačí určit jeho masu pravděpodobnosti.

Porovnávání algoritmů

Chceme-li porovnat nejen hypotézy, ale který ze dvou algoritmů učení L_A, L_B dosáhne v průměru lepší aproximace nějaké cílové funkce f, musíme odhadnout průměrný rozdíl výkonu d přes všechny n-prvkové trénovací množiny z distribuce $\mathcal{D}$. V praxi ale máme jen trénovací data S a testovací data T, takže náš odhad bude:

$$\hat{d} = \text{error}_T(L_A(S)) - \text{error}_T(L_B(S))$$

Pro zlepšení tohoto odhadu se často používá křížová validace – rozdělení dat D do k disjunktivních množin a provedení k testů, kdy se pokaždé použije jiná z množin jako testovací data všechny zbylé jako trénovací data. Odhad chyby $\bar{d}$ je průměr chyb $\hat{d}_i$ přes všech k množin. Rozptyl tohoto odhadu se stanoví jako výběrový rozptyl:

$$s^2_{\bar{d}} = \frac{1}{k(k-1)} \sum_{i=1}^{k} (\hat{d}_i - \bar{d})^2$$

Interval spolehlivosti tohoto odhadu se pak stanoví jako $P(d \in (\bar{d} \pm t_{N,k-1} s_{\bar{d}})) = N$, kde $t_{N,k-1}$ jsou kvantily t-rozdělení o $k-1$ stupních volnosti.

Kromě křížové validace je také možné vybírat testovací podmnožinu z dat, která máme k dispozici, úplně náhodně a opakovat testy libovolněkrát, ale testovací množiny už nebudou nezávislé, protože se dřív nebo později stane, že některé prvky vyberu víckrát.

21.7 Výpočetní aspekty strojového učení