MPUM - Projekt 3

Marcin Wykpis

1. Analiza danych

Powyższy wykres przedstawia rozkład kolejnych cech z pliku `phishing.data`.
Na pomarańczowo zaznaczone zostały wiersze, w których pożądaną predykcją byłoby -1, natomiast na niebiesko wiersze z wartością 1

Dane wydają się dość zrównoważone dla każdej z cech. Najmniej równomiernie rozłożone wydają się cecha 22 i 26. Sugeruje to, że bardzo proste drzewa decyzyjne (o stopniu 1 lub 2) mogą już dawać satysfakcjonujące wyniki. Przyjrzymy się temu w dalszej części raportu.

2. Przygotowanie danych

Cechy w pliku `phishing.data` należą do różnych dziedzin. Część z nich przyjmuje wartości $\{-1, 0, 1\}$, inne $\{-1, 1\}$ lub $\{0, 1\}$. Problem z dziedziną $\{-1, 0, 1\}$ jest taki, że możemy przez przypadek przemycić informację o pewnym nadanym porządku.
Na przykład w algorytmie `kNN` stosując odległość euklidesową, dochodzimy do wniosku, że -1 jest dalej 1 niż 0. Jest to raczej niepożądana własność, szczególnie że nie mamy informacji, co reprezentują kolejne cechy. Zakładanie takiego porządku jest więc niepożądane.

Żeby uniknąć wyżej opisanego problemu, zastosowałem tzw. `One Hot Encoder`. Rozwiązanie to sprawdza się dobrze gdy operujemy na dziedzinie o małej liczbie elementów.
Dla każdej cechy $x \in \{-1, 0, 1\}$ wymieniłem ją na 3 cechy: „Czy $x = -1$”, „Czy $x = 0$”, „Czy $x = 1$” przyjmujące wartości $\{0, 1\}$.
Cechy $x \in \{-1, 1\}$ zmapowałem również na $\{0, 1\}$. Wówczas dla każdej cechy $x \in \{0, 1\}$. Kolumnę wynikową pozostawiłem bez zmian.

3. Podział danych

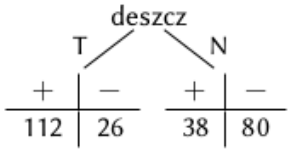
Dane podzieliłem w stosunku $2 : 1$ zbioru treningowego do zbioru testowego.
Zadbałem również o to, żeby zarówno dane z pozytywną jak i negatywną diagnozą były podzielone w tym stosunku. Podział danych jest losowany. Zmieniając parametr `seed` uzyskujemy różne podziały zbioru.

4. Drzewa Decyzyjne

Drzewa decyzyjne polegają na rekurencyjnym podziale po kolejnych cechach tak, aby otrzymać binarne drzewo „decyzji". Algorytm wykonujemy zachłannie, wyliczając dla każdej cechy, który z podziałów jest najlepszy. Poniższy slide z wykładu demonstruje mechanizm wyliczania, jak dobry jest dany podział.

Przykład: biegać czy zostać w domu?

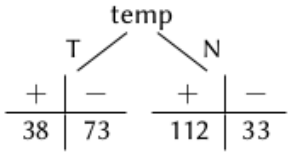
Tabelę doświadczeń możemy przedstawić w następujący sposób



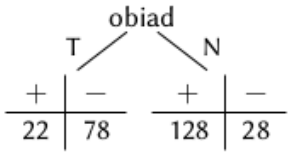
$$G_T = 1 - \left(\frac{112}{138}\right)^2 - \left(\frac{26}{138}\right)^2 = 0.306$$

$$G_N = 1 - \left(\frac{38}{118}\right)^2 - \left(\frac{80}{118}\right)^2 = 0.437$$

$$G = 0.306 \cdot \frac{138}{256} + 0.437 \cdot \frac{118}{256} = 0.366$$



$$G_T = 0.450$$
$$G_N = 0.352$$
$$G = 0.394$$



$$G_T = 0.343$$
$$G_N = 0.295$$
$$G = 0.314$$

Co może wydać się ciekawe, algorytm ten jest również odporny na nieprzeskalowane dane. Również problem opisany w sekcji „Przygotowanie danych" nie będzie tutaj miał miejsca. Rzeczywiście dla dziedziny $\{-1, 0, 1\}$ możemy przejrzeć wszystkie (3) możliwe podziały, znajdując ten najlepszy.

Tak prezentują się uzyskane `accuracy` na zbiorze testowym dla różnych głębokości drzewa. Wynik dla każdego z tych modeli został uśredniony z 5 przebiegów dla różnego podziału danych treningowych i testowych.

Głębokość	Precyzja	Czas Trenowania	Czas Predykcji
1	0.8925	0.0199	0.0023
2	0.9084	0.0399	0.0027
10	0.9493	1.2309	0.0089
20	0.9616	6.1970	0.0145

Tak jak wspomniane zostało w sekcji „Analiza danych" możemy przyjrzeć się jak dobrze radzi sobie bardzo małe drzewko o głębokości 1. Otrzymujemy stosunkowo dobry wynik ~0.9.

4.1. Support Vector Machine

Metoda wektorów nośnych polega na minimalizacji następującego wyrażenia.

$$C \cdot \|w\|^2 + \frac{1}{m} \sum_{i=1}^{m} \max\left(0, 1 - y^{(i)}\left(w^T x^{(i)} + b\right)\right)$$

Wówczas żeby otrzymać predykcję wystarczy obliczyć wartość wyrażenia (hiperpłaszczynę)

$$y_{\text{pred}} = w^T x$$

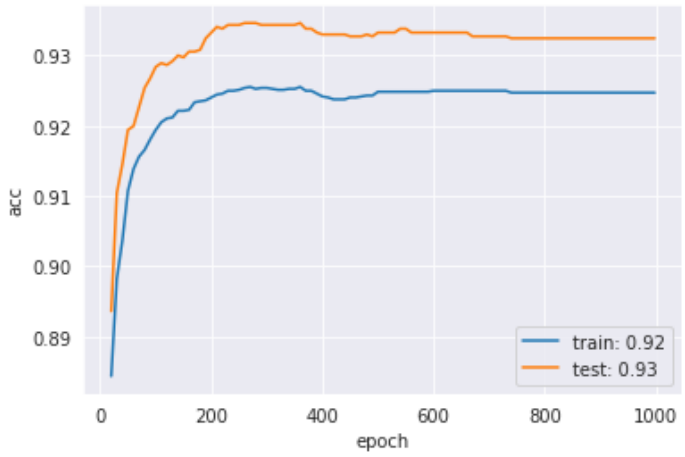
Jeśli $y_{\text{pred}}^{(i)} < 0$ wskazujemy klasę -1, a w przeciwnym wypadku klasę 1.

Stosując metodę spadku gradientu dostajemy wzór na wagi w kolejnych iteracjach:

1. $y^{(i)} \cdot y_{\text{pred}}^{(i)} \geq 1$ (wtedy max przymuje wartość 0)
   $w \leftarrow w - \lambda \cdot 2Cw$
   $b \leftarrow b$
2. $y^{(i)} \cdot y_{\text{pred}}^{(i)} < 1$
   $w \leftarrow w - \lambda \cdot 2Cw + x^{(i)} y^{(i)}$
   $b \leftarrow b + \lambda \cdot y^{(i)}$

Algorytm ten, dla hiperparametrów $\lambda = 0.0001$ oraz $C = 0.001$, daje nam dokładność **0.93** po niecałych **42** sekundach treningu (dane uśrednione z 5 przebiegów algorytmu dla róznych podziałów danych treningowych/testowych).

Poniżej prezentuje się dokładność w kolejnych iteracjach metody spadku gradientu:



5. k - Nearest Neighbour

Algorytm ten polega na znalezieniu najbliższych k sąsiadów. Na podstawie ich klas, wyliczamy prawdopodobieństwo przynależności do każdej z nich.

Algorytm w tej wersji jest bardzo prosty. Przechodzimy po wszystkich punktach i wyznaczamy odległości do zadanego punktu. Następnie wartości te sortujemy i bierzemy k najmniejszych z nich.
Nie stosujemy tu żadnego preprocessingu, a zatem czas odpowiedzi na pojedyncze zapytanie będzie dość wolny. Rzeczywiście wyznaczenie odpowiedzi na dane testowe o 3685 rzędach zajęło ok. **80 sekund**. Niemniej model sprawdza się całkiem nieźle, osiągając dokładność **0.94**.

Tak prezentują się uzyskane accuracy na zbiorze testowym dla różnych wartości k. Wynik dla każdego z tych modeli został uśredniony z 5 przebiegów dla różnego podziału danych treningowych i testowych.

k	Precyzja	Czas Trenowania	Czas Predykcji
1	0.9656	0.0	79.9399
5	0.9558	0.0	78.2672
10	0.9467	0.0	77.7900
20	0.9442	0.0	77.4020

5.1. KD - Drzewa

Podczas omawiania ostatniego miniprojektu, wspominaliśmy, że nieudane próby również warto zawrzeć w raporcie. Ta część będzie skupiać się na właśnie takiej eksploracji problemu.

5.1.1. Motywacja

Motywacją stojącą za zaimplementowanie takiej struktury był bardzo długi predykcji. Naiwna wersja tego algorytmu wyznaczała odpowiedź przechodząc za każdym razem po wszystkich punktach. Nie odciążaliśmy więc ilości obliczeń stosując preprocessing.

5.1.2. Rozwiązanie

W celu usprawnienia tego algorytmu postanowiłem zaimplementować `kd-drzewa`. Wcześniej eksperymentowałem i zaimplementowałem w wolnym czasie z zastosowaniem drzew czwórkowych, więc problem ten nie wydawał się większym wyzwaniem.

5.1.3. Trenowanie

`kd-drzewa`, w odróżnieniu od drzew czwórkowych dzielą rekurencyjnie hiper płaszczyznę na dwie części po jednej ze współrzędnych. Po dodaniu wszystkich punktów otrzymujemy więc rekurencyjny podział na (wielowymiarowe) prostokątne komórki, w których znajdują się punkty ze zbioru treningowego.
Zdecydowałem się dzielić komórkę po współrzędnej, która dzieli punkty w najbardziej zbliżony sposób (~0.5 po jednej stronie i ~0.5 po drugiej). Dodatkowo w celu zredukowania głębokości drzewa, w każdej z komórek-liści znajduje się pewna liczba punktów `capacity` (np. 100).

5.1.4. Predykcja

Wówczas, żeby otrzymać k najbliższych sąsiadów dla pewnego zadanego punktu, możemy sprawdzić kolejne komórki. Zaletą wcześniej utworzonej struktury jest to, że możemy dość szybko (w teorii) odrzucać wierzchołki drzewa, które znajdują się w całości dalej niż już znalezione punkty.
Żeby otrzymać k najbliższych sąsiadów, wystarczy użyć pewnej struktury, która umożliwia usuwanie największego elementu, dodawanie elementów i znajdowanie elementu maksymalnego. Strukturami, które umożliwiają szybkie wykonywanie takich operacji, jest np. kolejka priorytetowa lub kolejka maksimów. W mojej implementacji użyłem kolejki priorytetowej z biblioteki `heapq`.

5.1.5. Napotkane problemy

1. Pierwszym problemem, który napotkałem, był Pythonowi wyjątek mówiący o przekroczonej głębokości wywołań rekurencyjnych. Był to dość nieoczekiwany rezultat, z uwagi na fakt, że punktów w zbiorze treningowym nie znalazło się tak dużo.
   Okazuje się, że dla małej wartości `capacity` pojawia się zbiór identycznych punktów, który ma więcej elemntów niż `capcity`. Wówczas próbujemy bez skutku dzielić pewną komórkę w nieskończoność.

Problem ten rozwiązałem, pozwalając na przekraczanie `capacity` jeżeli nie istnieje żaden dalszy podział.

2. Drugi (niestety nierozwiązany) problem, to prędkość. Z nieznanych mi powodów zaimplementowana struktura nie przyspiesza szukania najbliższych elementów.
   Być może dla ogromnych zestawów danych efekt byłby zauważalny. Problem może być związany również z implementacją, choć jest ona lekką modyfikacją drzewa czwórkowego, które wydawało się być zadowalająco szybkie.
   Ostatnim moim pomysłem, co może powodować zaistniały problem, jest sam rozkład danych. Być może dwuelementowa dziedzina $\{0, 1\}$ sprawia, że duża część danych jest zgrupowana, co nie pozwala na szybkie odrzucanie kolejnych komórek.
3. Trzeci problem, który zauważyłem to nieznaczne różnice w wynikach dawanych przez dwa modele. Nie powinno to mieć oczywiście miejsca, `kNN` jest algorytmem w pełni deterministycznym.
   Przyjrzałem się pobieżnie temu problemowi. W wierszach, które sprawdziłem, różnica wynikała z wybrania różnych reprezentantów wśród równoodległych punktów. Istnieją punkty o tych samych współrzędnych z różnym zaklasyfikowaniem.

Tak prezentują się uzyskane accuracy na zbiorze testowym dla różnych wartości k oraz dla capacity $= 100$. Wynik dla każdego z tych modeli został uśredniony z 5 przebiegów dla różnego podziału danych treningowych i testowych.

k	Precyzja	Czas Trenowania	Czas Predykcji
1	0.9647	0.1572	73.4703
5	0.9537	0.1742	103.5071
10	0.9462	0.1589	114.8301
20	0.9401	0.1596	118.2375

6. Implementacja

Implementację powyższych modeli oraz użyte w raporcie wykresy można znaleźć w repozytorium na githubie: https://github.com/Marwyk2003/mpum_miniproject3