Optymalizacja hiperparametrów xgboost

Dokumentacja wstępna

Przemysław Stawczyk, Piotr Zmyślony

15 kwietnia 2020

Spis treści

1 Treść zadania

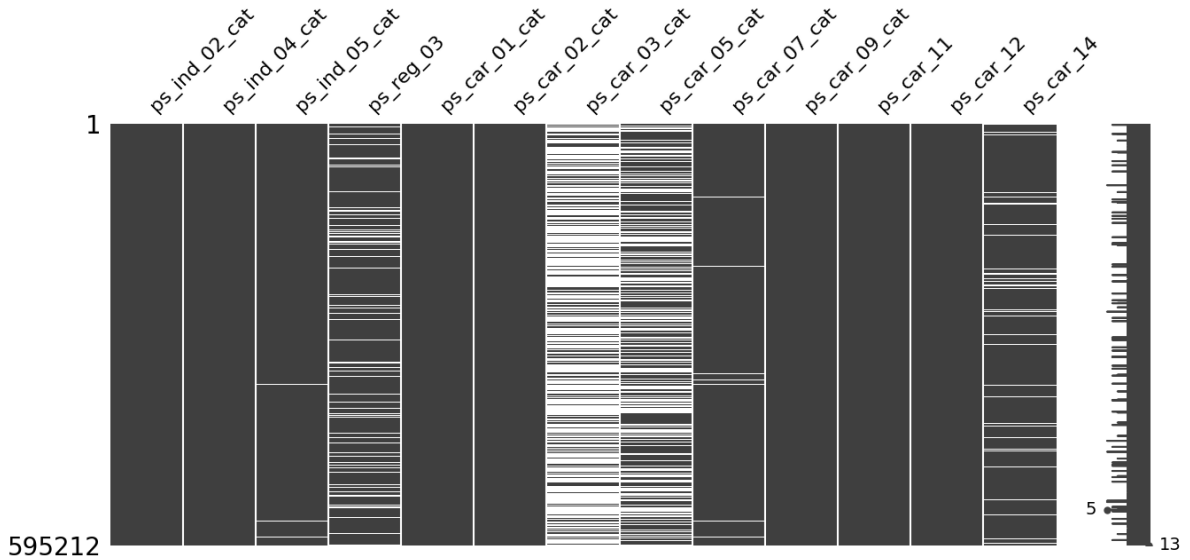
Naszym zadaniem jest przetestowanie różnych algorytmów heurystycznych/populacyjnych w kontekście problemu strojenia hiperparametrów algorytmu xgboost. Problem wyboru hiperparametrów wynika z ich bardzo dużej ilości, co często rozwiązane jest poprzez manualny dobór parametrów klasyfikatora.

Projekt zostanie zrealizowany w języku Python 3+.

2 Dane testowe

Jako dane na których będziemy trenować i testować klasyfikatory przyjęliśmy proponowany zestaw danych https://www.kaggle.com/c/porto-seguro-safe-driver-prediction. Zawiera on 57 atrybutów opisujących klientów firmy ubezpieczeniowej i jeden atrybut binarny sygnalizujący, czy w ciągu roku od zawarcia umowy, klient skorzystał z ubezpieczenia.

Rysunek 1: Brakujące atrybuty



2.1 Analiza danych

Po wstępnej analizie danych odkryliśmy, że w zbiorze danych posiadamy około 79% niekompletnych wierszy. Rysunek 1 przedstawia pokrycie niekompletnych atrybutów - jest ich jedynie 13, z czego większość jest wybrakowana w bardzo niewielkim stopniu.

Największym winowajcą jest atrybut binarny *ps_car_03_cat*, którego brakuje aż w 70% wierszy, oraz atrybut *ps_car_05_cat* (brakuje go w 44% przypadków). W końcowej wersji zdecydowaliśmy się usunąć oba z tych parametrów.

Dodatkowo, występuje znaczna dysproporcja między klasami rekordów - tylko 3% wierszy opisuje klientów, którzy skorzystali z ubezpieczenia. Stąd niezbędna będzie interpolacja danych, tak aby ilość rekordów obu klas była równa.

3 Algorytmy

Zaimplementowaliśmy następujące algorytmy:

- algorytm wspinaczkowy z tabu.
  W 2 wariantach:
  - mutacyjny z prawdopodobieństwem P mutacji jednego (losowego) z parametrów
  - z przeglądem sąsiedztwa i powracaniem
- przegląd wyczerpujący hipersiatki jako metoda bazowa

3.1 Przestrzeń poszukiwań

Trenowane modele posiadały parametry z hipersiatki, czyli iloczyn zbiorów każdego z parametrów, co w sumie daje nam 52272 dopuszczalnych rozwiązań.

nazwa parametu	zakres
liczba słabych modeli	50, 75, 100 ... 300
eta	0.1, 0.2, 0.3, 0.4
min_split_loss *gamma*	0, 1, 2, 3
max_depth	4, 5, 6 ... 14
max_delta_step	0, 1, 2
subsample	0.6, 0.8, 1
colsample_bytree	0.6, 0.8, 1

Dalej, *sąsiadem* zestawu A będziemy nazywali takie zestawy parametrów, które od zestawu A różnią się tylko jednym parametrem, większym bądź mniejszym o jeden "kwant".

3.2 Funkcja celu

Jako funkcję celu przybraliśmy *Average Precision Recall* obliczając wartość funkcji celu jako średnią arytmetyczną skuteczności przypisania predykcji. Zastosowaliśmy implementację z pakietu *scikit-learn*.
Ta sama funkcja zostanie wykorzystana do oceny jakości finalnych wytrenowanych modeli na zbiorach testowych, przy 3-krotnej walidacji krzyżowej, gdzie nasz końcowy jest średnią osiągniętych wyników AUC ROC.

3.3 Algorytm brutalny

Algorytm brutalny generuje zbiór wszystkich możliwych zestawów parametrów, które przegląda krok po kroku. Wynikiem działania jest zestaw parametrów osiągający najwyższy wynik AUC ROC.

3.4 Stabuizowany algorytm wspinaczkowy z przeglądem sąsiedztwa

Jako parametr startowy algorytmu należy podać maksymalną ilość powrotów `n_max`, które może wykonać przed zwróceniem aktualnego najlepszego wyniku. Zaczyna swoje działanie od wylosowania punktu startowego z dostępnych zbiorów parametrów (na początku przypisujemy go do `P_b` i `P_c`). Oblicza wynik dla `P_c` dalej w następujący sposób:

1. `Ustaw liczbę nawrotów n = 0`
2. `Jeśli n > n_max, przejdź do punktu ostatniego.`
3. `Wygeneruj wszystkich (nigdy nieodwiedzonych) sąsiadów P_c.`
4. `Wybierz sąsiada, który daje wynik lepszy bądź równy wynikowi P_c.`

5. Jeśli nie istnieje taki sąsiad:
   (a) n = n + 1
   (b) ustaw P_c na poprzedni P_c
6. Jeżeli istnieje:
   (a) n = 0
   (b) P_c = najlepszy sąsiad
   (c) Jeżeli wynik P_c >= P_b: P_b = P_c
7. Wróć do punktu 2.
8. Zwróć najlepszy zestaw parametrów.

3.5 Stabuizowany mutacyjny algorytm wspinaczkowy

W przypadku mutacji połączonej z tabu, problematycznym okazało się tworzenie mutantów z wykorzystaniem rozkładu normalnego, jak i mutacji w ten sam sposób wielu parametrów - tworzone były wielokrotnie już sprawdzone zestawy parametrów.

Przyjeliśmy metodę w której tworzone są wszystkie (jeszcze niezbadane) możliwe zestawy różniące się jednym parametrem w stosunku do rodzica i spośród nich losowany jest nowy mutant.

Zaczyna swoje działanie od wylosowania punktu startowego z dostępnych zbiorów parametrów - P_r. Następni sprawdza kolejne mutacje punktu początkowego przyjmując je jako punkt roboczy o ile metryka ma wyższą wartość.

1. Zainicjuj zestaw parametrów
2. Wygeneruj wszystkie (nigdy nieodwiedzone) mutantacje P_r.
3. Jeśli nie kolejnych mutacji STOP
   - Zwróć zestaw parametrów P_r.
4. Jeżeli istnieje:
   (a) Wybierz losowo mutacje P_m.
   (b) Ewaluuj mutanta.
   (c) Jeżeli wynik P_m >= P_r: P_r = P_m
5. Wróć do punktu 2.
6. Zwróć zestaw parametrów P_r.

4 Sposób mierzenia jakości rozwiązania

Jako że trenowanie znacznej liczby modeli przy użyciu adekwatnych ilości danych wejściowych jest bardzo czasochłonne, postanowiliśmy zaimplementować opcję testowania wszystkich trzech algorytmów w określonych ramach czasowych. Dzięki podaniu odpowiedniego parametru, program uruchamia każdy z nich tylko na jakiś czas, po którym algorytm musi zwrócić najlepszy dotychczasowy wynik.

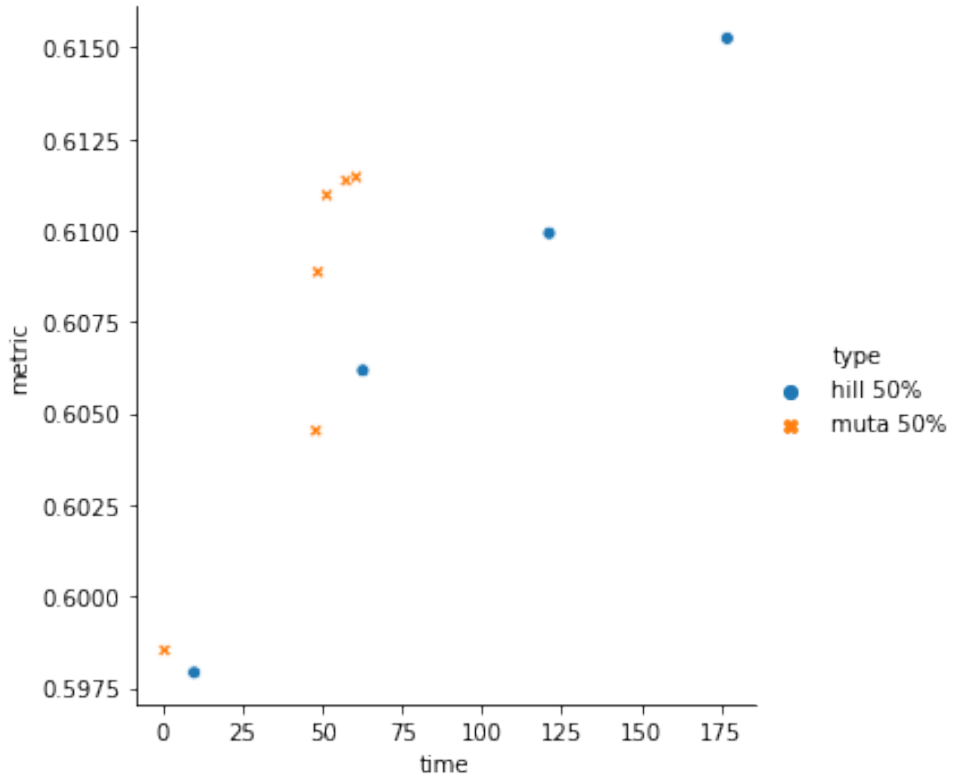
Wszystkie opcje uruchomieniowe programu można sprawdzić poleceniem:

```
python performance_test.py --help
```

5 Wyniki pomiarów

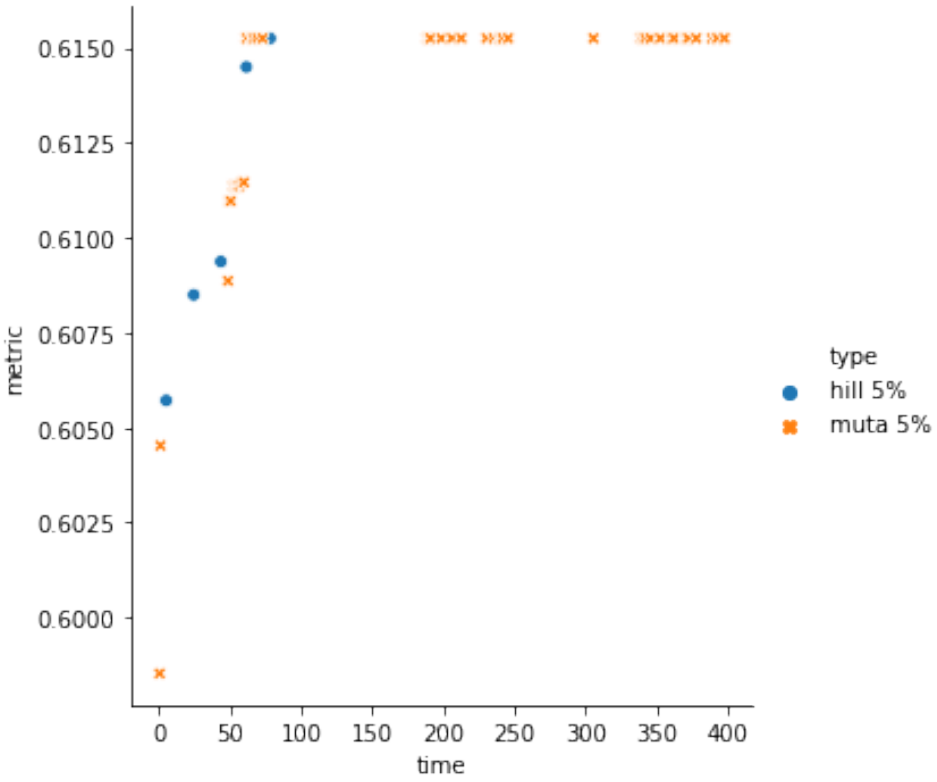
Otrzymane rezultaty dla algorytmu wspinaczkowego i algorytmu mutacyjnego są oddalone od siebie nieznacznie, ale różnice możemy zaobserwować w przebiegu ich działania. Poniższe wykresy reprezentują wszystkie zestawy parametrów, które pobiły dotychczasowy najlepszy wynik.

Rysunek 2: Wykres najlepszego dotychczasowego wyniku w zależności od czasu



Na powyższym wykresie, około 50. sekundy możemy zaobserwować taką mutację zestawu parametrów, która dała znaczny wzrost AUC ROC, a potem wyraźny jest proces znajdywania przez algorytm lokalnego optimum.

Rysunek 3: Trafność warunku stopu algorytmu z przeglądem sąsiedztwa



Na rysunku 3, między 50. a 100. sekundą, widoczne jest działanie warunku stopu dla algorytmu z przeglądem sąsiedztwa. Warunkiem stopu dla algorytmu mutacyjnego jest jedynie pokrycie odpowiedniej ilości przestrzeni rozwiązań (w tym wypadku jest to 5%).

Zastosowany algorytm z przeglądem sąsiedztwa posiadał ilość maksymalnych powrotów ustawioną na 8, co okazuje się trafne, ponieważ przez następne 300 sekund działania, algorytm z mutacjami nie potrafił znaleźć znacznie lepszego rozwiązania, podczas gdy poprzedni już skończył swoje działanie. Niewykluczone, że wartość ta może być mniejsza, co pozwoli jeszcze bardziej zaoszczędzić na czasie przetwarzania.

6 Wnioski i rekomendacje

Porównanie obu algorytmów Algorytm mutacyjny w naszych próbach typowo osiąga szybciej szczytowe rezultaty szybciej od algorytmu z przeglądem sąsiedztwa. Metryki znalezione przez te algorytmu mimo przeglądu 50% są bliskie optymalnemu [różnica poniżej 0.01 metryki], co ciekajak bedziemy poprawiac to we uruchomienie nawet dla 5% daje zbliżone rezultaty.

Testowanie Niestety komputery domowe nie są idealnym narzędziem do szybkiego trenowania modeli przez XGBoost - testowanie heurustyk było mocno ograniczone przez niewystarczającą moc sprzętu. Stąd benchmarking obu algorytmów powinny być uruchamiany na zdecydowanie mocniejszych maszynach.