

AutoML – Projekt 2 – raport

Sebastian David Botero Leonik, Bartosz Pokora, Franciszek Saliński

Listopad 2025

1 Cel projektu

Celem projektu było stworzenie miniaturowego systemu do AutoML w zadaniu klasyfikacji binarnej. Na projekt złożyły się dwa etapy:

1. Inteligentny wybór ~ 50 bazowych modeli do portfolio na bazie zewnętrznych eksperymentów.
2. Stworzenie systemu *MiniAutoML*, wybierającego najlepszy spośród modeli w portfolio – a także ensembler z nich złożonych – pod dany dataset.

2 Etap 1: Stworzenie portfolio

2.1 MementoML

W tworzeniu portfolio korzystaliśmy z danych z MementoML. W danych z Memento napotkaliśmy problemy związane z niespójnością danych.

Dla niektórych modeli, dla których są policzone wyniki, nie umieszczono konfiguracji. Problem ten był mało znaczący dla większości modeli, ale XGBoost miał braki dla ponad połowy testów. Oczyszczenie danych praktycznie zdyskwalifikowało XGBoost.

Kolejnym problemem był fakt, że dane były tworzone na podstawie eksperymentów w R, podczas gdy nasz projekt był robiony w Pythonie. Dla większości modeli obeszło się bez większych przeszkód, ale model ranger był problematyczny, jako że jedynym sensownym jego odpowiednikiem w Pythonie jest skranger, który jest przestarzały.

Do poradzenia sobie z tymi problemami utworzona została klasa Memento, która odpowiada za jakość danych. Pozwala ona również otrzymać dane w formacie, który jest bardziej przydatny w tworzeniu portfolio i którego oczekuje funkcja average_smfo.

2.2 CANE i A-SMFO

Do wybrania portfolio zaimplementowaliśmy algorytm A-SMFO, który został pokazany na wykładzie. Korzystaliśmy z pracy naukowej z sekcji IV.A i opisanych pseudokodem algorytmów Algorithm 1 CANE Optimal Sequence (dalej nazywany CANE) i Algorithm 2 Average SMFO (dalej nazywany A-SMFO)

Algorytm CANE został zaimplementowany w klasie CANE, która przyjmuje dane w postaci ramki z trzema kolumnami: config, score i dataset. Dzięki funkcji pandas.DataFrame.pivot uzyskujemy efektywnie rankingi modeli, co pozwala na wydajne liczenie metryki perf, kluczowej dla tego algorytmu.

Algorytm A-SMFO służy do dopełnienia portfolio do 50, gdyby CANE tego nie zrobił, poprzez ponowne wywołanie go dla nie wybranych jeszcze modeli. Okazał się on tu zbyteczny, gdyż CANE nie osiągał swojego warunku stopu. Został zaimplementowany w funkcji average_smfo.

3 Etap 2: Ewaluacja modeli i wybór najlepszego z nich

3.1 Preprocessing

Jako że różne sposoby przygotowania danych nie miały być przedmiotem skupienia projektu, każdy model dostaje dane przetworzone przez ten sam pre-definiowany Pipeline, na który składają się: zastąpienie braków danych medianą (lub najczęszą kategorią dla zmiennych kategorycznych), one-hot encoding zmiennych kategorycznych, a także standaryzacja wszystkich zmiennych.

Kategoryczne zmienne rozpoznajemy przede wszystkim za pomocą typu danych w pythonie. Zmienne o typie numerycznym z niską liczbą unikalnych wartości (< 10) traktujemy jako kategoryczne, natomiast kategoryczne z wysoką liczbą unikalnych wartości (> 20) encodujemy liczbami całkowitymi (OrdinalEncoder), zamiast one-hota. Oba progi mogą być modyfikowane przez użytkownika.

3.2 Kroswalidacja jako sposób ewaluacji modeli

Zdecydowaliśmy się na 5-krotną kroswalidację, jako solidną metodę ewaluacji poszczególnych modeli z portfolio na danym zbiorze danych. Kroswalidacja wszystkich 50 modeli jest wprawdzie kosztowna czasowo (do 40 minut na podanym przykładowym datasetie o wymiarach 3482×16). Nie było to jednak według nas na tyle długo, aby rezygnować z niezawodnej jakości w porównaniu do podejścia, w którym na początku przerzedzamy portfolio, np. za pomocą heurystyk lub meta-modelu patrzących na rozmiar danych. Takie techniki z definicji nie są w stanie tak dobrze dopasować się do konkretnego datasetu i skazane są na częste pomyłki, niesłuszne odrzucanie dobrych modeli. Rozważylibyśmy je jedynie w przypadku powiększania portfolio (np. do 200 konfiguracji), aby system nadal był w stanie zwrócić wynik w racjonalnym czasie.

Pozwalamy użytkownikowi samemu bilansować między jakością ewaluacji, a czasem poświęconym na fit, dodając parametr `cv_samples_limit`. Do kroswalidacji używamy próbki danych, liczącej maksymalnie tyle rekordów. Porównując na przykładowym zbiorze, ograniczając rozmiar danych w kroswalidacji do 1000 osiągnęliśmy nawet trochę lepsze Balanced Accuracy na zbiorze testowym (0.5660 bez ograniczenia i 0.5681 przy ograniczeniu). Powtórzylibyśmy eksperyment wielokrotnie dla pewności, ale sama jedna iteracja zajęła godzinę.

3.3 Ranking modeli

Optymalizowaną metrykę można wybrać spośród AUC oraz Balanced Accuracy. Po kroswalidacji modele sortowane są według zasad:

1. W pierwszej kolejności według średniego wyniku w kroswalidacji zaokrąglonego do 4 miejsc po przecinku
2. Ewentualne remisy rozstrzygane na korzyść stabilniejszego modelu – o mniejszym odchyleniu standardowym z kroswalidacji.

3.4 Ensembling

Z top 5 modeli uzyskanych w poprzednim kroku budujemy jeszcze 3 ensemble: Soft Voting, Hard Voting oraz Stacking (z liniowym meta-modelem, bez passthrough). Tak samo jak modele z portfolio, wszystkie 3 oddajemy kroswalidacji i dodajemy je na odpowiednie miejsca w rankingu.

3.5 Optymalizacja

Jako że wybraliśmy już bardzo kosztowny sposób ewaluacji modeli, chcieliśmy dokonać optymalizacji czasowych gdzie tylko się dało. Przede wszystkim ważne dla nas było, żeby nie wykonywać przypadkiem tego samego fitu wielokrotnie. Oprócz tego zaimplementowaliśmy przetwarzanie równolegle.

3.5.1 Optymalizacja w preprocessingu i kroswalidacji

Podczas kroswalidacji wykonujemy $50 \cdot 5 = 250$ fitów modeli. Gdybyśmy każdy z nich owinęli w pre-processing pipeline, wykonalibyśmy ten sam preprocessing 50 razy więcej, niż to potrzebne. Oczywiście nie zadowala nas prosty `fit_transform(X)` na samym początku, bo formalnie prowadzi do wycieku danych, a także nie będzie radził sobie np. w z niewidzianymi podczas fitu kategoriami w danych testowych. Aby zaoszczędzić niepotrzebnych obliczeń, na początku wykonujemy preprocessing dla każdego z 5 splitów i następnie w kroswalidacji korzystamy już z odpowiednich przetworzonych danych.

Ponadto zaimplementowaliśmy znane z sklearn przetwarzanie równolegle za pomocą biblioteki `joblib` (parametr `n_jobs`). Ewaluacje poszczególnych modeli są równomiernie rozdzielane między dostępne procesy.

3.5.2 Optymalizacja w ensemblingu

Chcąc kroswalidować normalnie świeżo utworzony w sklearn `VotingClassifier`, wszystkie 5 modeli bazowych fitowałoby się od zera ($5 \cdot 5 = 25$ niepotrzebnych fitów), co jest w stanie wydłużyć cały proces o parę dodatkowych minut. Aby tego uniknąć, trzymamy zaflitowane podczas kroswalidacji modele w pamięci i używamy ich ponownie, aby otrzymać predykcje na foldach zarówno w głosowaniu twardym i miękkim.

W stackingu standardowo stosuje się podejście Out-of-Fold (OOF). Nasza implementacja jest aproksymacją, tak aby także korzystać z już zaflitowanych modeli: predykcje OOF używane do treningu meta-modelu pochodzą z modeli, które widziały część danych walidacyjnych (przy treningu na innych foldach). Pełne podejście wymagałoby wewnętrznego CV ograniczonego wyłącznie do foldów treningowych, co prowadziłoby do wielokrotnego refitowania modeli bazowych. Trzeba

mieć więc na uwadze, że wynik stackingu przy kroswalidacji może być sztucznie zawyżony (choć empiryczne eksperymenty pokazały, że wcale nie znajduje się tak często na szczycie rankingu).

4 Wnioski

Podczas tworzenia – nawet tak prostego jak nasz – systemu AutoML doświadczyliśmy już trudności z porządną empiryczną weryfikacją podejmowanych decyzji oraz znalezieniem równowagi pomiędzy poziomem gwarancji jakości, a pewnych koniecznych uproszczeń na rzecz oszczędności czasowych.

Zaimplementowane techniki ensemblingu często okazywały się wartościowym uzupełnieniem pojedynczych modeli, choć ich wyższość nie była gwarantowana w każdym przypadku. Potwierdza to, że ensemble nie powinny być traktowane jako domyślnie najlepsze rozwiązanie, lecz jako kolejny kandydat podlegający tej samej procedurze ewaluacji.