WUM - Projekt - Raport

1 Cel projektu

Celem projektu było zbudowanie klasyfikatora o jak największej mocy predykcyjnej mierzonej miarą zrównoważonej dokładności *(balanced accuracy)*.

2 Pliki z rozwiązaniem

Cały kod potrzebny do rozwiązania projektu został podzielony na następujące pliki: `config.py`, `preprocessors.py`, `param_grids.py`, `model_tuner.py`, `data_exploration.ipynb`, `basic_models.ipynb`, `ensambles.ipynb`, `b_m&ens_results.ipynb`, `voting.ipynb`, `best_classifier_final.ipynb`.
Plik `config.py` zawiera zmienną `kfold` będącą wywołaniem instrukcji `KFold()` z ustawionymi parametrami `cv = 7`, `random_state = 327426` oraz `shuffle = True`. Ustawienia te są wykorzystywane we wszystkich przypadkach kroswalidacji w projekcie. Ponadto wszędzie, gdzie było to możliwe ustawiono parametr `max_iter` na 10000.
Szczegółowy opis zawartości pozostałych plików przedstawiono w kolejnych sekcjach raportu.

3 Eksploracja danych

Dysponowano sztucznie wygenerowanym zbiorem danych *artificial*, zawierającym 100 zmiennych objaśniających i obejmującym pliki: `artifical_train_data.csv` (zbiór treningowy), `artifical_train_labels.csv` (etykiety zbioru treningowego) oraz `artifical_test_data.csv` (zbiór testowy). Oznaczono je odpowiednio jako `X_train`, `y_train` i `X_test`. Zbiór treningowy zawierał 1500 obserwacji, natomiast zbiór testowy 500.
Na początku sprawdzono występowanie braków danych. Stwierdzono, że ani zbiór treningowy, ani testowy nie zawierał brakujących wartości.
Następnie sporządzono histogramy oraz wykresy pudełkowe dla wszystkich 100 zmiennych objaśniających, na podstawie których oceniono skośność rozkładów oraz obecność obserwacji odstających. Analiza wykazała brak istotnej skośności rozkładów. Choć w wielu zmiennych występowały obserwacje odstające, nie były one bardzo ekstremalne, dlatego na tym etapie nie przeprowadzono transformacji zmiennych ani nie usuwano obserwacji.
Na koniec przeanalizowano etykiety zbioru treningowego. Proporcje klas zmiennej objaśnianej `y_train` okazały się niemal zrównoważone: klasa 1 stanowiła $50{,}3\%$, a klasa 2 — $49{,}6\%$.

4 Preprocessing

W pliku `preprocessors.py` zdefiniowano siedem funkcji zwracających `pipeliny` odpowiedzialne za preprocessing danych dla poszczególnych modeli. Skład każdego z nich przedstawiono w poniższej tabeli.

preprocessor	**skład**
`preprocessor_select()`	`scaler: StandardScaler()`, `const: VarianceThreshold(threshold=0.01)` `selector: SelectFromModel(LogisticRegression(l1,saga))`
`preprocessor_PCA()`	`scaler: StandardScaler()`, `const: VarianceThreshold(threshold=0.01)` `selector: PCA()`
`preprocessor_rf()`	`scaler: StandardScaler()`, `const: VarianceThreshold(threshold=0.01)`
`preprocessor_et()`	`scaler: StandardScaler()`, `const: VarianceThreshold(threshold=0.01)`
`preprocessor_gbc()`	`scaler: StandardScaler()`, `const: VarianceThreshold(threshold=0.01)`
`preprocessor_xgb()`	`const: VarianceThreshold(threshold=0.01)`
`preprocessor_lgbm()`	`const: VarianceThreshold(threshold=0.01)`

Pierwsze dwa preprocessory wykorzystano przy budowie klasyfikatorów opartych na modelach podstawowych, natomiast pozostałe zastosowano w przypadku komitetów klasyfikatorów. Informację o tym, który preprocessor został użyty dla danego komitetu, jednoznacznie określa jego nazwa.
Dla modeli wykorzystujących `preprocessor_select()` optymalizowano parametr `selector__estimator__C`, przyjmujący wartości `np.logspace(-3, 2, 10)`. Natomiast w przypadku `preprocessor_PCA()` strojonymi parametrami były: `selector__n_components`, `selector__whiten` oraz `selector__svd_solver`, o wartościach odpowiednio `[20, 40, 60, 75, 80, 90, 95]`, `[True, False]` oraz `["full", "randomized"]`. Dla pozostałych preprocessorów nie prowadzono optymalizacji parametrów.

5 Funkcja `model_tuner()` i siatki hiperparametrów `param_grids`

Na potrzeby projektu zbudowano wiele klasyfikatorów, dlatego w celu uporządkowania kodu w pliku `model_tuner.py` zaimplementowano funkcję `model_tuner(model, pre, params)`. Funkcja ta przyjmuje model klasyfikacji (np. `SVC()`), preprocessor z pliku `preprocessors.py` oraz siatkę hiperparametrów przeznaczonych do optymalizacji, zdefiniowaną w pliku `param_grids.py`.
Funkcja konstruuje pipeline składający się z preprocessora `pre` (lub — w przypadku ustawienia `pre = None` — pomija etap preprocessingu) oraz modelu `model`. Następnie, dla tak przygotowanego pipeline'u oraz siatki parametrów `params`, funkcja `model_tuner()` wykonuje kroswalidację na danych treningowych z wykorzystaniem `GridSearchCV()`, przyjmując jako metrykę *balanced accuracy*. Parametry kroswalidacji są ustawiane na podstawie zmiennej `kfold` z pliku `config.py`. Wyłącznie na potrzeby tej funkcji zmienna `y_train` została przemapowana z wartości 1 i 2 na 0 i 1 (poprzez odjęcie 1), ponieważ część modeli akceptuje wyłącznie takie etykiety klas.
W wyniku działania funkcji zwracany jest pipeline z optymalnymi hiperparametrami, który osiągnął najwyższą średnią wartość *balanced accuracy* w kroswalidacji. Dodatkowo zwracana jest sama średnia jako wynik modelu oraz jego odchylenie standardowe.

Siatki optymalizowanych hiperparametrów dla wszystkich klasyfikatorów zdefiniowano w pliku `param_grids.py` i zebrano w słowniku `param_grids`. Umożliwia to wygodny dobór odpowiedniej siatki parametrów w pozostałych częściach projektu.

Uwaga: na końcu ostatniej strony raportu znajduje się tabela przedstawiająca siatki optymalizowaych hiperparametrów dla danego modelu oraz ich możliwe wartości. Nazwa danej siatki jednoznacznie wskazuje na to, do którego modelu została ona użyta (`params_`*nazwa modelu*). Natomiast, nazwa `params_`*nazwa modelu*`_(select/PCA)` oznacza, że dana siatka występuje w dwóch wersjach. Poza parametrami modelu optymalizowane są również parametry preprocessora `preprocessor_select()` lub preprocessora `preprocessor_PCA()`, które podane zostały w poprzedniej sekcji.

6 Modele podstawowe i komitety klasyfikatorów

Na początku w projekcie zbudowano kilka podstawowych modeli klasyfikacji i każdy z nich połączono z preprocessorem `preprocessor_select()`, a następnie z `preprocessor_PCA()`. W pliku `basic_models.ipynb` znajdują się wywołania funkcji `model_tuner()` dla wszystkich kombinacji rozważonych modeli klasyfikacji oraz podanych preprocessorów (z wybranymi odpowiednimi siatkami parametrów). Wyniki tych wywołań funkcji zapisano w postaci słownika w pliku `all_basic_grids.pkl` (nazwa elementu w słowniku sugeruje, którego klasyfikatora oraz preprocessora użyto). Poniższa tabela przedstawia wyniki dla wszystkich modeli podstawowych uszeregowane malejąco.

name	model	score	var
svm_PCA	SVC(probability=True)	0.788285	0.026171
knn_select	KNeighborsClassifier()	0.776660	0.033445
knn_PCA	KNeighborsClassifier()	0.773528	0.023255
tree_select	DecisionTreeClassifier()	0.772595	0.036183
tree_PCA	DecisionTreeClassifier()	0.748135	0.039292
qda_PCA	QuadraticDiscriminantAnalysis()	0.700844	0.022155
svm_select	SVC(probability=True)	0.686658	0.030315
qda_select	QuadraticDiscriminantAnalysis()	0.636529	0.031978
glm_en_select	LogisticRegression(penalty=elasticnet)	0.616464	0.045915
glm_l1_select	LogisticRegression(penalty=l1)	0.611789	0.033803
glm_l2_PCA	LogisticRegression(penalty=l2)	0.609361	0.051610
glm_en_PCA	LogisticRegression(penalty=elasticnet)	0.607232	0.052698
glm_l2_select	LogisticRegression(penalty=l2)	0.606852	0.035173
glm_l1_PCA	LogisticRegression(penalty=l1)	0.605043	0.056817
glm_select	LogisticRegression(penalty=None)	0.604700	0.037449
lda_select	LinearDiscriminantAnalysis()	0.604026	0.037317
glm_PCA	LogisticRegression(penalty=None)	0.602144	0.046820
lda_PCA	LinearDiscriminantAnalysis()	0.597781	0.045196

Następnie przystąpiono do budowy komitetów klasyfikatorów. Z pliku `all_basic_grids.pkl` wybrano pipeliny modeli `glm_en_select`, `svm_PCA`, `knn_select`, `tree_select` i na ich podstawie zbudowano komitety używając *Baggingu*. Dla każdego z nich w funkcji `model_tuner()` podano siatkę hiperparametrów na `params_bagging` i pominięto preprocessor. Ponadto zbudowano modele oparte o *Random forest*, *Extra trees*, *Gradient boosting*, *XGBoost* (z pakietu `xgboost`) oraz *LightGBM* (z pakietu `lightgbm`). Dla każdego z nich w funkcji `model_tuner()` ustawiono pasujący preprocessor i siatkę parametrów. W pliku `ensambles.ipynb` znajdują się wywołania funkcji `model_tuner()` dla wszystkich rozważonych modeli komitetów oraz odpowiednich preprocessorów i siatek parametrów. Wyniki tych wywołań funkcji zapisano w postaci słownika w pliku `all_ensamble_grids.pkl`. Poniższa tabela przedstawia wyniki dla wszystkich modeli komitetów uszeregowane malejąco.

name	model	score	var
et	ExtraTreesClassifier()	0.872733	0.029274
rf	RandomForestClassifier()	0.841028	0.018532
lgbm	LGBMClassifier(obj=binary)	0.829319	0.034049
tree_bagging	BaggingClassifier(tree_select)	0.825010	0.022200
xgb	XGBClassifier(obj=binary:logistic, logloss, hist)	0.824005	0.020391
svm_bagging	BaggingClassifier(svm_PCA)	0.814795	0.027604
gbc	GradientBoostingClassifier()	0.787967	0.020824
knn_bagging	BaggingClassifier(knn_select)	0.775499	0.030799
glm_en_bagging	BaggingClassifier(glm_en_select)	0.615406	0.042320

Zestawienie wyników z obu tabel zostało wygenerowane w pliku `b_m&ens_results.ipynb`.

7 Voting

Następnie wybrano cztery modele z poprzedniej sekcji: `et`, `rf`, `lgbm` oraz `xgb`. Na ich podstawie zbudowano klasyfikator z wykorzystaniem funkcji `VotingClassifier(voting=soft)`.
Jakość modelu oceniono w pliku `voting.ipynb` za pomocą funkcji `cross_val_score()`, przy ustawieniach `cv = kfold` (pobranych z pliku `config.py`) oraz `scoring = balanced_accuracy`. Przeanalizowano trzy warianty parametru `weights`: `None`, wagi proporcjonalne do wyników poszczególnych modeli oraz wektor wag `[4, 3, 2, 2]`.
We wszystkich trzech przypadkach średnia wartość miary *balanced accuracy* z kroswalidacji wynosiła około 0.84, a odchylenie standardowe około 0.02. W związku z tym model oparty na głosowaniu nie przewyższył skutecznością najlepszych klasyfikatorów opisanych w poprzedniej sekcji.

8 Najlepszy klasyfikator

Ostatecznie za najlepszy model klasyfikacji uznano model `et` oparty o metodę *Extra trees*. W wyniku kroswalidacji w modelu tym ustawiono następujące parametry: `'n_estimators': 150`, `'criterion': entropy`, `'max_depth': 20`, `'max_features': 0.5`.
Wynik modelu to **0.872733**, a odchylenie standardowe: 0.029274.
Zestawienie wyników tego modelu można znaleźć w pliku `best_classifier_final.ipynb`. Na koniec wyznaczono wektor zawierający prawdopodobieństwo przynależności do klasy 1 dla danych testowych i zapisano go w pliku `327426_artificial_prediction.txt`.

Tabela 1: Hiperparametry optymalizowane w modelach

siatka parametrów	**zmienne i możliwe wartości**

`params_tree_(select/PCA)`	`criterion: [gini, entropy]`
	`max_depth: [None, 1, 3, 5, 7, 9]`
	`min_samples_leaf: [1, 3, 5, 7, 9]`
	`splitter: [best, random]`
	`min_samples_split: [2, 4, 6, 8, 10]`

`params_glm_(select/PCA)`	`solver: [lbfgs, newton-cg, sag, saga]`
`params_glm_l2_(select/PCA)`	`solver: [lbfgs, liblinear, newton-cg, sag, saga]` `C: np.logspace(-3, 2, 10)`
`params_glm_l1_(select/PCA)`	`solver: [liblinear, saga]` `C: np.logspace(-3, 2, 10)`
`params_glm_en_(select/PCA)`	`C: np.logspace(-3, 2, 10)` `l1_ratio: np.linspace(0.001, 0.999, 10)`

`params_svm_(select/PCA)`	`kernel: [linear, poly, rbf, sigmoid]`
	`C: np.logspace(-3, 2, 10)`
	`gamma: [scale, auto]`

`params_lda_(select/PCA)`	`(brak strojenia hiperparametrów modelu)`
`params_qda_(select/PCA)`	`reg_param: [0.0, 0.01, 0.05, 0.1, 0.2, 0.5]`

`params_knn_(select/PCA)`	`n_neighbors: [5, 10, 20, 50, 80, 100, 150, 200]`
	`weights: [uniform, distance]`
	`metric: [euclidean, manhattan]`

`params_rf`	`n_estimators: np.arange(50, 500, 50)`
	`criterion: [gini, entropy]`
	`max_features: [sqrt, 0.5]`

`params_bagging`	`n_estimators: np.arange(50, 500, 50)`

`params_et`	`n_estimators: np.arange(50, 500, 50)`
	`criterion: [gini, entropy]`
	`max_depth: [None, 10, 20]`
	`max_features: [sqrt, 0.5]`

`params_gbc`	`n_estimators: np.arange(50, 500, 50)`
	`learning_rate: [0.01, 0.05, 0.1]`
	`max_depth: [2, 3, 4]`
	`min_samples_leaf: [1, 5, 9]`

`params_xgb`	`n_estimators: np.arange(50, 500, 50)`
	`learning_rate: [0.01, 0.05, 0.1]`
	`max_depth: [3, 5]`
	`subsample: [0.8, 1.0]`
	`colsample_bytree: [0.8, 1.0]`
	`min_child_weight: [1, 5, 10]`

`params_lgbm`	`n_estimators: np.arange(50, 500, 50)`
	`learning_rate: [0.01, 0.05, 0.1]`
	`num_leaves: [15, 31, 63]`
	`max_depth: [-1, 5]`
	`min_child_samples: [10, 20, 50]`
	`subsample: [0.8, 1.0]`
	`colsample_bytree: [0.8, 1.0]`