Automatyczne uczenie maszynowe - Projekt 1

Norbert Frydrysiak Bartosz Jezierski

Listopad 2024

1 Wstęp

Na zajęcia „Automatyczne Uczenie Maszynowe" na 3 roku kierunku Inżynieria i Analiza Danych na wydziale Matematyki i Nauk Informacyjnych Politechniki Warszawskiej przygotowaliśmy projekt, nad którym pracowaliśmy od początku października do piętnastego listopada 2024 roku. Celem projektu było przeanalizowanie tunowalności 3 wybranych algorytmów uczenia maszynowego na 4 zbiorach danych za pomocą 2 różnych technik losowania punktów z przestrzeni hiperparametrów.

1.1 Dane

Dane, które wzieliśmy do eksperymentu to:

1. Credit Score - Zbiór danych zawiera informacje o 1000 klientach oraz 84 cechy pochodzące z ich transakcji finansowych i obecnej sytuacji finansowej. Głównym celem jest wykorzystanie tego zbioru danych do przewidywania potencjalnych przypadków niewypłacalności. Z wszystkich danych ta ramka danych jest najbardziej skomplikowana.
2. Raisin - Zbiór do klasyfikacji dwóch rodzajów ziaren rodzynek Kecimen i Besni, uprawianych w Turcji. W ramce danych mamy łącznie 900 ziaren rodzynek, równo po 450 z każdej klasy. Z obrazów wyodrębniono 7 cech morfologicznych, które są kolumnami.
3. Alzheimer's Disease - Zbiór danych zawiera rozległe informacje zdrowotne o 2149 pacjentach. Obejmuje on szczegóły demograficzne, czynniki związane ze stylem życia, historię medyczną, pomiary kliniczne, oceny poznawcze i funkcjonalne, objawy oraz diagnozę choroby Alzheimera.
4. Salary - Zbiór danych został wyodrębniony przez Barry'ego Beckera z bazy danych spisu z 1994 roku i służy do przewidywania, czy dana osoba zarabia ponad 50 tys. dolarów rocznie. Obejmuje takie kolumny jak wiek, klasa zatrudnienia, poziom wykształcenia, stan cywilny, zawód, relacje rodzinne, rasa, płeć, dochód kapitałowy, liczba godzin pracy w tygodniu oraz kraj pochodzenia.

1.2 Wybrane algorytmy uczenia maszynowego

W naszym projekcie rozważyliśmy 4 algorytmy uczenia maszynowego z pakietu *scikit-learn*:

- LogisticRegression
- RandomForestClassifier
- KNeighborsClassifier
- GradientBoostingClassifier

Tak wybraliśmy, ponieważ byliśmy ciekawi wniosków na temat tych algorytmów oraz mają umiarkowany czas uczenia.

1.3 Techniki losowania punktów

W naszym projekcie rozważyliśmy 2 techniki losowania punktów z przestrzeni hiperparametrów:

- Random Search z pakietu *scikit-learn* za pomocą RandomSearchCV
- Bayes Search z pakietu *scikit-optimize* za pomocą BayesSearchCV

2 Rozważane przestrzenie hiperparametrów

W wybieraniu przestrzeni hiperparametrów inspirowaliśmy się wnioskami z Tunability: Importance of Hyperparameters of Machine Learning Algorithms.

2.1 Logistic Regression

Dla Random Searcha mamy siatkę:

```
    search_grid = {
    'model__C': stats.loguniform(1e-4, 1e4),
    'model__penalty': ['l1', 'l2'],
    'model__solver': ['liblinear', 'saga'],
    "model__class_weight": [None, "balanced"]
}
```

Natomiast dla Bayes Searcha rozważąliśmy siatkę:

```
    search_grid = {
    'model__C': Real(1e-4, 1e4, prior='log-uniform'),
    'model__penalty': Categorical(['l1', 'l2']),
    'model__solver': Categorical(['liblinear', 'saga']),
    "model__class_weight": Categorical([None, "balanced"])
}
```

2.2 RandomForestClassifier

Dla Random Searcha mamy siatkę:

```
    search_grid = {
    'model__n_estimators': np.arange(200, 1750, 25),
    'model__max_depth': [None] + list(np.arange(5, 110, 5)),
    'model__min_samples_split': np.arange(2, 20, 2),
    'model__min_samples_leaf': np.arange(1, 20, 2),
    'model__max_features': ['sqrt', 'log2', None],
    'model__criterion': ['gini', 'entropy', 'log_loss'],
    'model__max_samples': [None] + list(np.linspace(0.1, 1.0, 10)),
    'model__ccp_alpha': stats.loguniform(1e-4, 1e4)
}
```

Natomiast dla Bayes Searcha rozważąliśmy siatkę:

```
    search_grid = {
    'model__n_estimators': Integer(200, 1750),
    'model__max_depth': Integer(5, 200),
    'model__min_samples_split': Integer(2, 20),
    'model__min_samples_leaf': Integer(1, 20),
    'model__max_features': Categorical(['sqrt', 'log2', None]),
    'model__criterion': Categorical(['gini', 'entropy', 'log_loss']),
    'model__max_samples': Real(0.1, 1.0),
    'model__ccp_alpha': Real(1e-4, 1e4, prior='log-uniform')
}
```

2.3 KNeightboursClassifier

Dla Random Searcha mamy siatkę:

```
    search_grid = {
    'knn__n_neighbors': [i for i in range(2, 101)],
    'knn__weights': ['uniform', 'distance'],
    'knn__p': [i for i in range(1,4)],
    'knn__leaf_size': [i for i in range(1, 51, 1)]
}
```

Natomiast dla Bayes Searcha rozważąliśmy siatkę:

```
    search_grid = {
    'knn__n_neighbors': Integer(2, 100),
    'knn__weights': Categorical(['uniform', 'distance']),
```

```
    'knn__p': Integer(1,4),
    'knn__leaf_size': Integer(1,50)
}
```

2.4 GradientBoostingClassifier

Dla Random Searcha mamy siatkę:

```
    search_grid={
    'gbc__n_estimators': [i for i in range(50, 1701,1)],
    'gbc__learning_rate': np.linspace(0.01, 0.2, 5),
    'gbc__max_depth': np.arange(3, 11),
    'gbc__min_samples_leaf': [i/1000 for i in range(5,201,5)],
    'gbc__subsample': [i/10 for i in range(1,11)],
    'gbc__max_features': [None, 'sqrt', 'log2']
}
```

Natomiast dla Bayes Searcha rozważaliśmy siatkę:

```
    search_grid={
    'gbc__n_estimators': Integer(50, 1700),
    'gbc__learning_rate': Real(0.01, 0.2),
    'gbc__max_depth': Integer(3, 30),
    'gbc__min_samples_leaf': Real(5/1000,200/1000),
    'gbc__subsample': Real(0.01,1.0),
    'gbc__max_features': Categorical([None, 'sqrt', 'log2'])
}
```

3 Jak liczyliśmy tunowalność algorytmu?

Jako metrykę do klasyfikacji binarnej wzięliśmy ROC AUC. Rozważaliśmy tunowalność jako:

$$T \stackrel{\text{def}}{=} M(\theta_{opt}) - M(\theta_{default})$$

gdzie:

- θ_{opt} to jest wektor hiperparametrów optymalnych, które dawały najlepszą wartość metryki, szukanych przez BayesSearchCV lub RandomSearchCV
- $\theta_{deafult}$ to jest wektor hiperparametrów, które są dane standardowo w pakiecie *scikit-learn* dla danego algorytmu
- $M(\theta)$ jest to obliczona wartość metryki dla modelu z wektorem hiperparametrów θ

4 Wnioski z eksperymentu

Wykonywaliśmy po 100 iteracji RandomSearchCV oraz BayesSearchCV zawsze, aby znaleźć optymalny zestaw hiperparametrów. Próbowaliśmy również porównywać jak radzi sobię BayesSearch w czasie, w którym RandomSearch robił 100 iteracji. W tej sekcji spróbujemy odpowiedzieć na następujące pytania:

1. Ile iteracji oraz czasu potrzeba, aby uzyskać stabilne wyniki optymalizacji?
2. Czy występuje bias sampling?

4.1 LogisticRegression

Z racji, że nie jest to skomplikowany model z niewielką ilością hiperparametrów nie potrzeba dużo iteracji oraz czasu. RandomSearchCV trwał 4 minuty, natomiast BayesSearchCV 40 minut. Obie metody zbiegają do podobnych optymalnych hiperparametrów na wszystkich zbiorach danych, co można zobaczyć na wykresie 9. Zazwyczaj potrzeba kilku iteracji, aby otrzymać dość stabilne wyniki, ale to zależy też od skomplikowania danych. Po 4 minutach od odpalenia obu metod BayesSearch zachowywał się gorzej w znajdywaniu lepszych hiperparametrów, co można zauważyć na wykresie 13. Zauważyliśmy, że nie występuje zjawisko bias samplingu.

4.2 RandomForestClassifier

Skomplikowanie modelu, który ma sporo różnych hiperparametrów, spowodowało, że RandomSearchCV trwał 18 minut, natomiast BayesSearchCV 2 godziny i 18 minut. Widać, że tutaj już występuje zjawisko bias sampling. BayesSearch znajduje takie zestawy hiperparametrów, które są "niedostępne" dla RandomSearchCV, co można zauważyć na wykresie 10. Gdyby ograniczyć do 18 minut obie metody samplingu to tylko na zbiorze Salary BayesSearch znalazłby lepszy zestaw hiperparametrów, dla pozostałych zbiorów byłby daleko do metryk osiąganych przez najlepsze zestawy hiperparametrów od RandomSearcha, zobacz wykres 14. Jeżeli chodzi o ilość iteracji do uzyskania optymalnych wyników widać, że RandomSearch często napotyka na "ścianę" i niezbyt często znajduje coś lepszego, z wykresu 10 można zauważyć, że dla RandomSearch wystarczy często nawet tylko 20 iteracji, aby wyniki ustabilizowały się, natomiast BayesSearch ma potencjał, żeby znajdywać jeszcze więcej lepszych zestawów hiperparametrów przez więcej niż 100 iteracji.

4.3 KNeightborsClassifier

Nie jest to skomplikowany model. RandomSearch wykonał 100 iteracji w mniej niż minutę, natomiast BayesSearch w około 13 minut. Zjawisko bias samplingu nie występuje. Zarówno dla BayesSearch i RandomSearch po około 60 iteracjach otrzymujemy stabilne wyniki, co można zauważyć na wykresie 11. Przez pierwszą minutę przy bardziej skomplikowanych danych BayesSearch daje gorsze zestawy hiperparametrów, wykres 15. Z racji tego, że ten model głównie ma parametr ilości sąsiadów to nie widać zalet użycia bayesowskiej optymalizacji ponad Random Searcha.

4.4 GradientBoostingClassifier

Ten model ma podobną sytuacje co RandomForestClassifier. 100 iteracji RandomSearcha trwały 7 minut, natomiast BayesSeacha 56 minut. Po niewielkiej ilości iteracji (około 30) BayesSearch już znajduje lepsze zestawy hiperparametrów, które nie będą osiągalne dla RandomSearcha, co można zauważyć na wykresie 12. Zjawisko bias samplingu występuje. Jakby rozważyć to czasowo, to przez pierwsze 7 minut szukania hiperparametrów BayesSearch na większości ramek danych znalazł lepsze zestawy hiperparametrów niż RandomSearch, co można zauważyć na wykresie 16.

5 Podsumowanie

Optymalizacja Bayesowska jest bardzo użyteczna, przez to że inteligentnie przeszukuje przestrzeń hiperparametrów pozwala znaleźć zestawy hiperparametrów o wiele lepsze niż wyszukawanie przez losowanie. Jednakże przez to, że RandomSearch może być zrównoleglony, w wielu przypadkach wystarczy użyć RandomSearch, ale jeśli chcemy idealnych zestawów hiperparametrów i każda poprawa jakości naszych prognoz ma ogromne znaczenie to warto poświęcić czas i moc obliczeniową na BayesSearch. Zjawisko bias samplingu występuje zazwyczaj w skomplikowanych modelach z bogatą przestrzenią hiperparametrów. Ciekawe okazało się, że czasami najbardziej tunowalnym algorytmem spośród wszystkich wybranych okazuje się KNeighborsClassifier, co można zauważyć na wykresach 17 i 18.

6 Grafika

6.1 Zbieżność metody BayesSearch

Wykresy poniżej pokazują średnie wartości metryki uzyskane w kolejnych iteracjach metody BayesSearch na różnych ramkach danych.

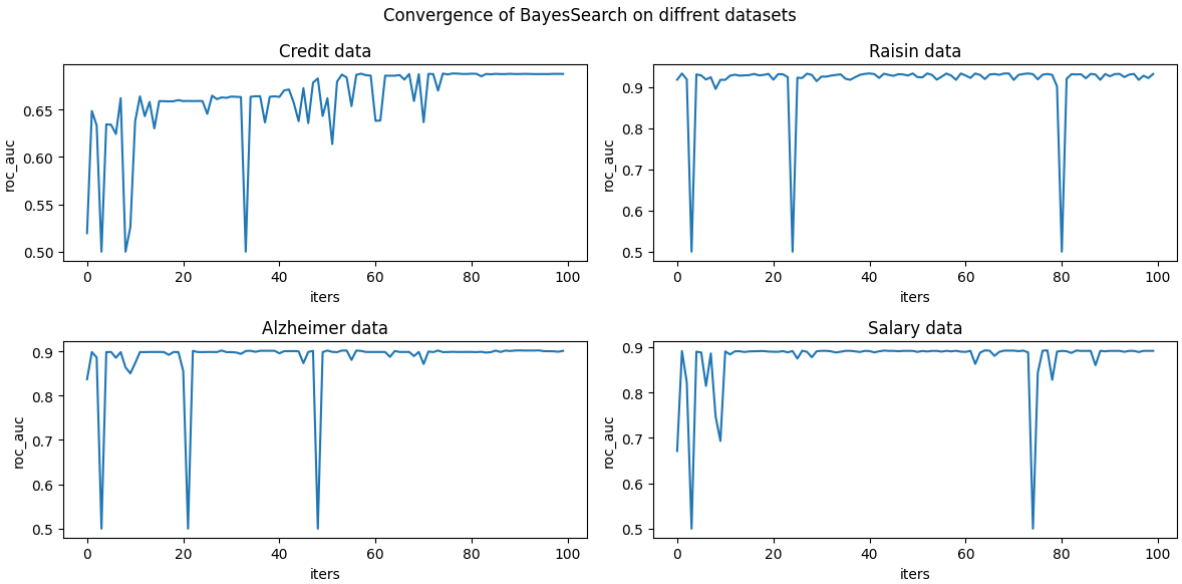

Figure 1: Wykres zbieżności dla modelu LogisticRegression.

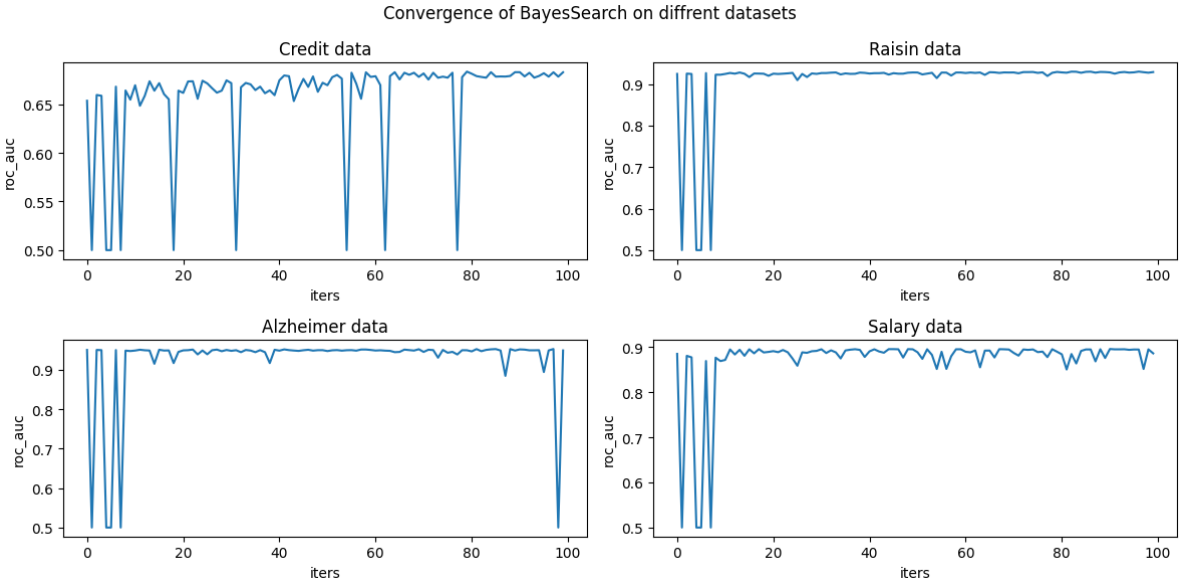

Figure 2: Wykres zbieżności dla modelu RandomForestClassifier.

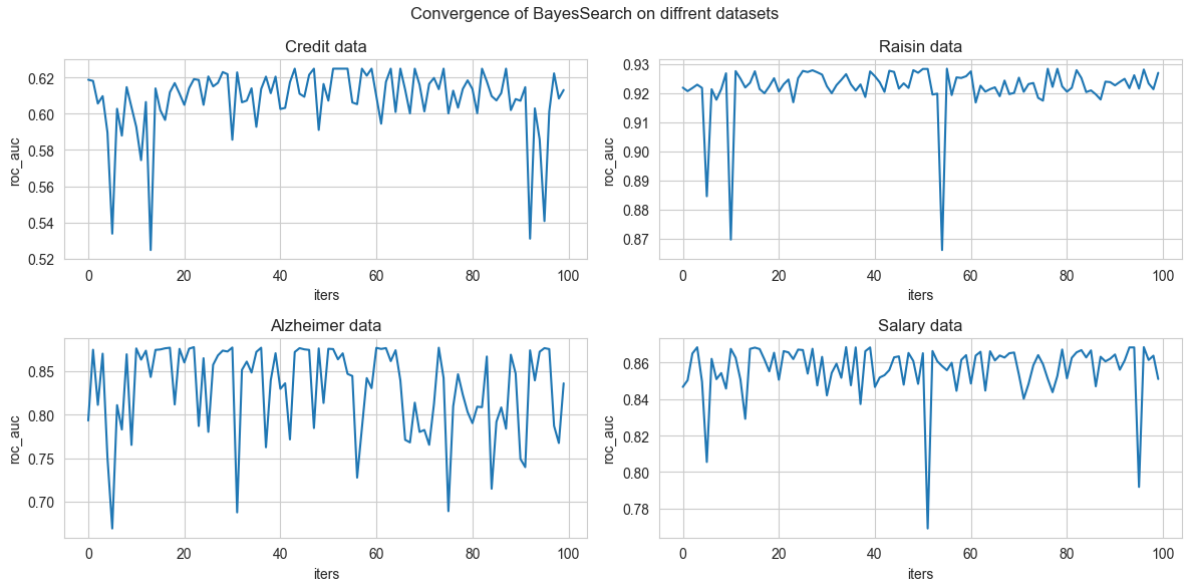

Figure 3: Wykres zbieżności dla modelu KNeighborsClassifier.

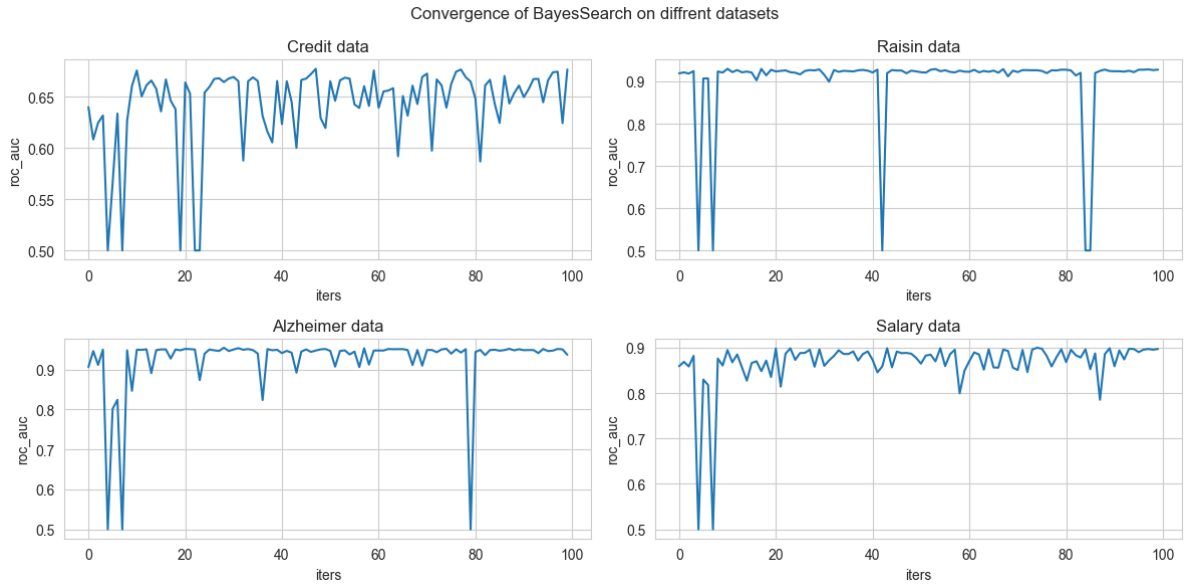

Figure 4: Wykres zbieżności dla modelu GradientBoostingClassifier.

6.2 Rozkład wartości metryki w kolejnych iteracjach

Wykresy poniżej przedstawiają w formie boxplotów rozkład metryki przesunięty o metrykę uzyskaną w przypadku użycia modelu z default'owymi hiperparametrami tzn. przedstawiony jest rozkład:

$$y = ROC\_AUC_{(CVMeanInIteration)} - ROC\_AUC_{(CVMeanWithDefaultHiperparameters)}$$

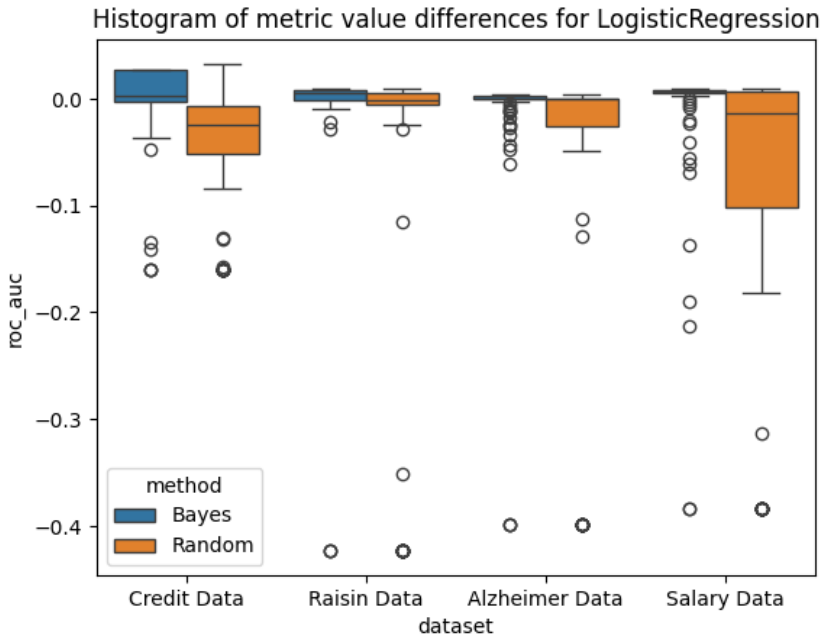

Figure 5: Wykres rozkładu metryki w kolejnych iteracjach dla modelu LogisticRegression.

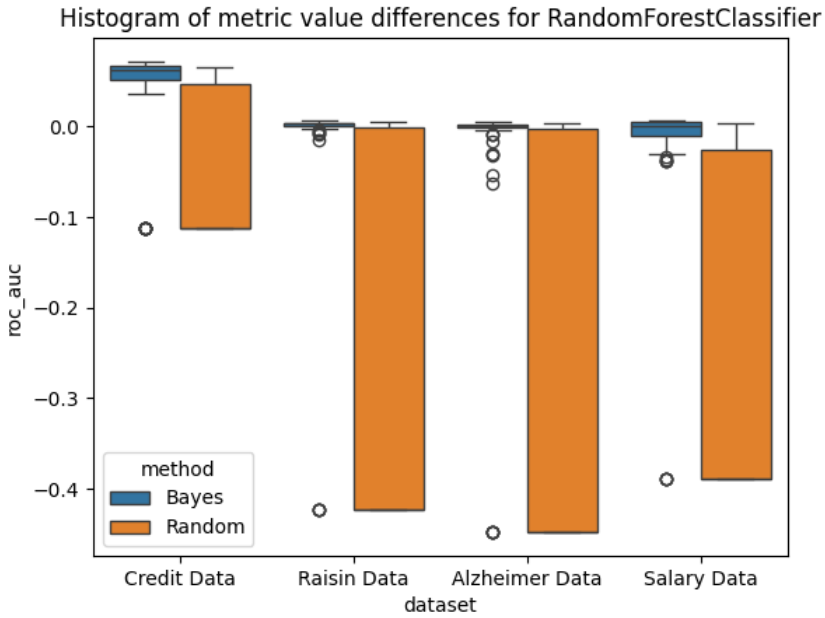

Figure 6: Wykres rozkładu metryki w kolejnych iteracjach dla modelu RandomForestClassifier.

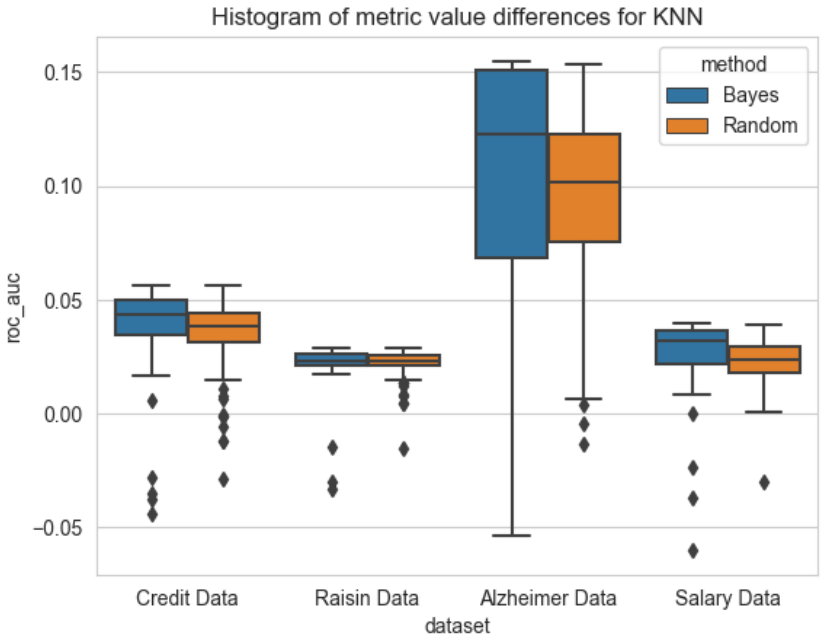

Figure 7: Wykres rozkładu metryki w kolejnych iteracjach dla modelu KNeighborsClassifier.

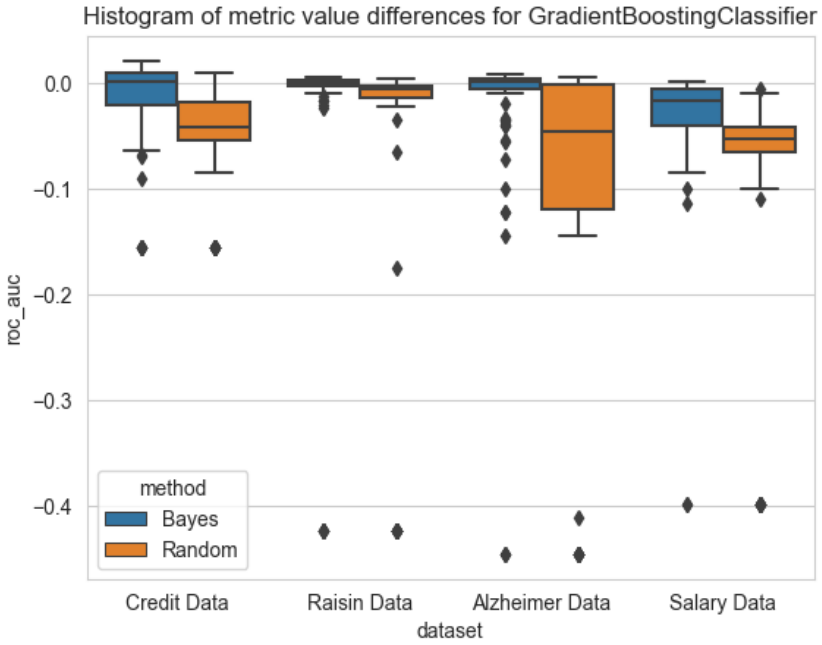

Figure 8: Wykres rozkładu metryki w kolejnych iteracjach dla modelu GradientBoostingClassifier.

6.3 Ile iteracji potrzeba dla uzyskania optymalnych wyników?

Wykresy poniżej przedstawiają maksymalną (krosswalidacyjną) średnią wartość metryki uzyskanej od pierwszej do oznaczonej na osi X iteracji tzn:

$$y = cummulativeMax(CVMeanROC\_AUC[from(1)to(iter)])$$

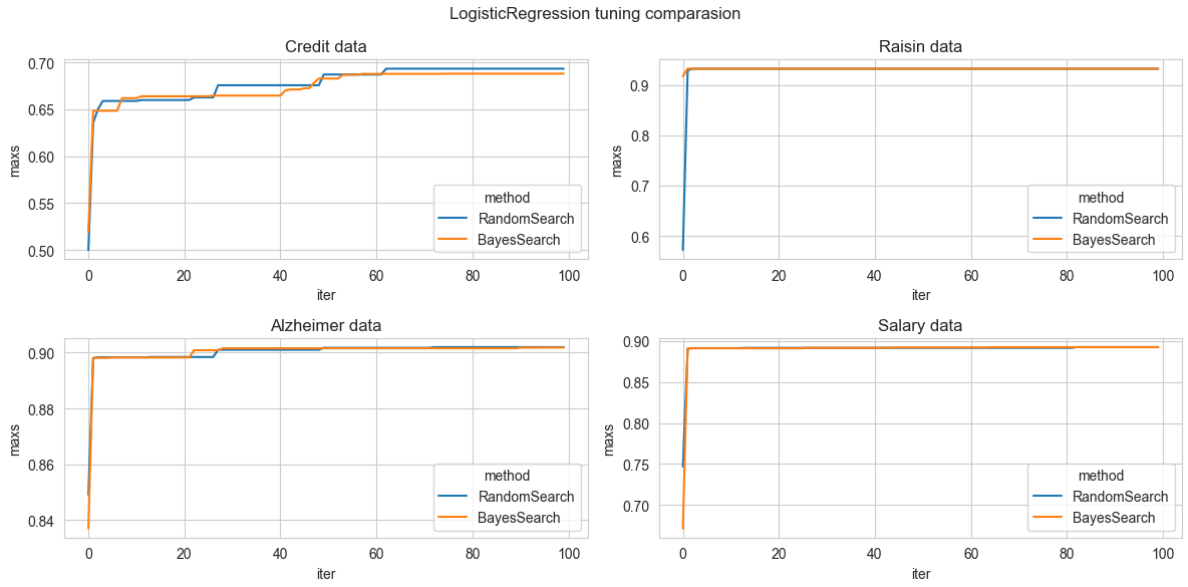

Figure 9: Wykres przedstawiający maksymalną jak dotychczas (w sensie iteracji) wartość metryki dla algorytmu LogisticRegression.

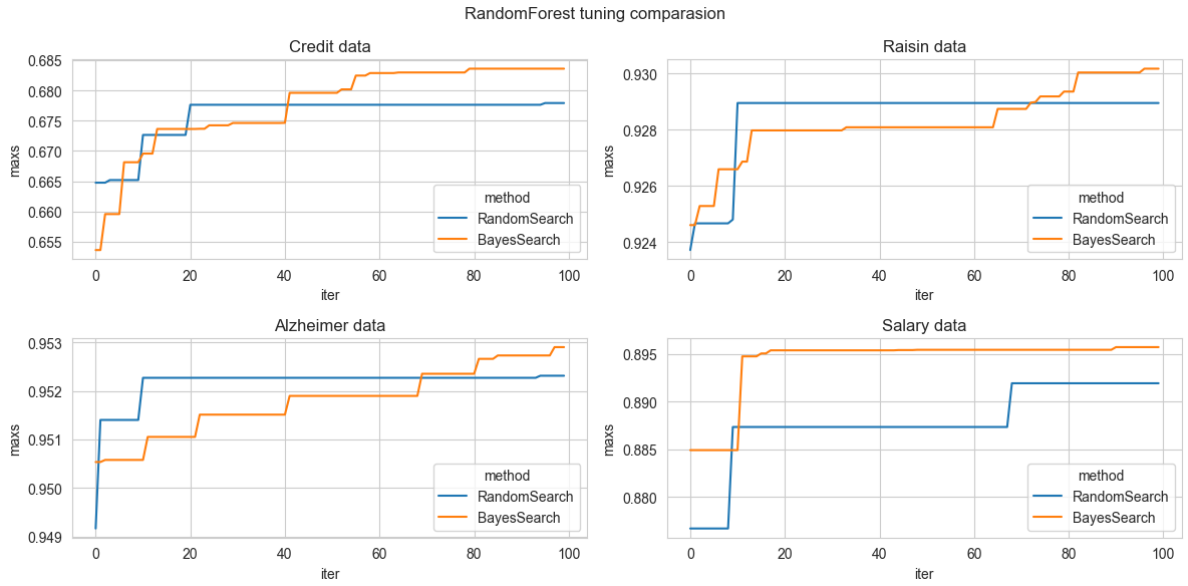

Figure 10: Wykres przedstawiający maksymalną jak dotychczas (w sensie iteracji) wartość metryki dla algorytmu RandomForestClassifier.

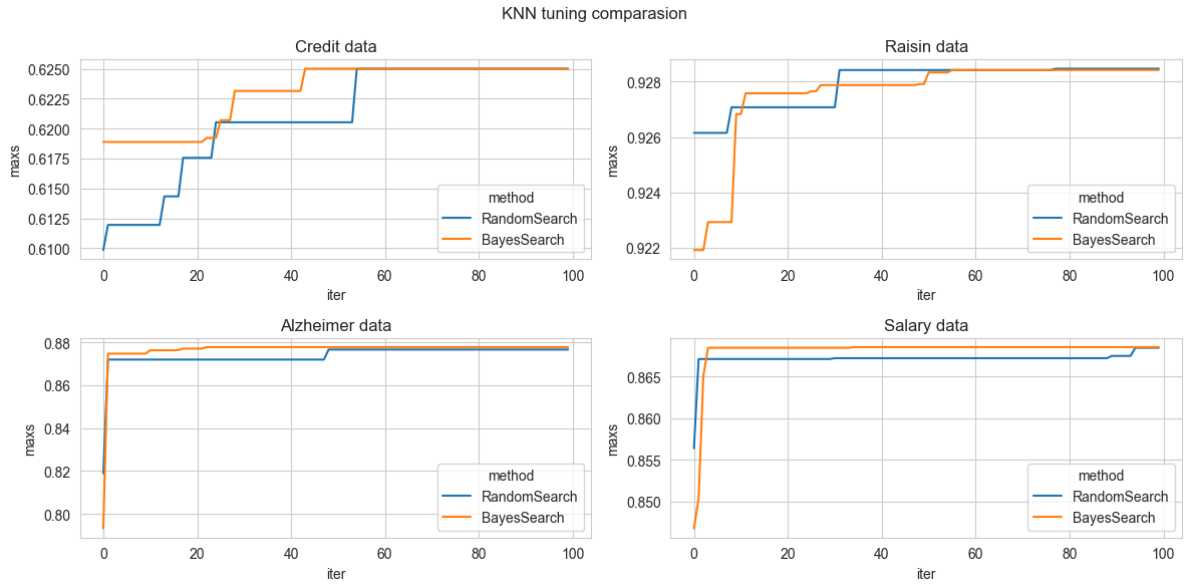

Figure 11: Wykres przedstawiający maksymalną jak dotychczas (w sensie iteracji) wartość metryki dla algorytmu KNeighborsClassifier.

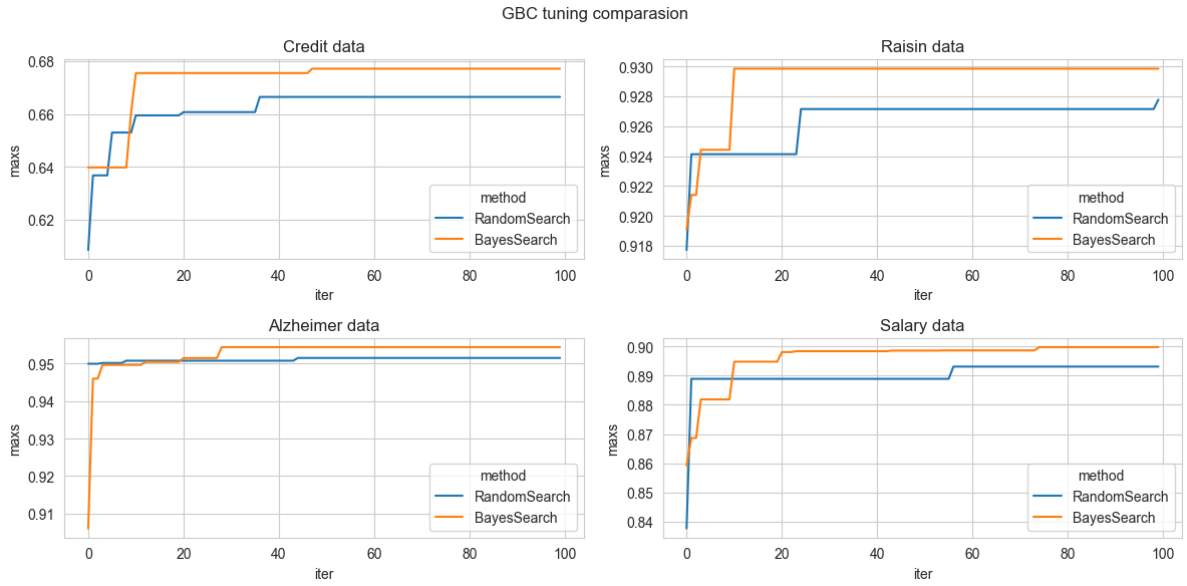

Figure 12: Wykres przedstawiający maksymalną jak dotychczas (w sensie iteracji) wartość metryki dla algorytmu GradientBoostingClassifier.

6.4 Zależność wyniku optymalizacji od czasu

Wykresy poniżej przedstawiają zależność podobną do tego co działo się w poprzedniej sekcji wykresów (tak samo liczony cummulativeMax) jednak tym razem na osi X ląduje czas. Ukazują one jaki mniejwięcej(*) wynik otrzymałaby metoda BayesSearch, gdyby dać jej tyle samo czasu, ile wykonywał się RandomSearch.

(*) Przyjęliśmy, że każda iteracja BayesSearch wykonuje się tyle samo czasu, aby móc skorzystać z wcześniej uzyskanych wyników (wykres dla metody BayesSearch jest odpowiednio rozciągnięty (o wielokrotność czasu) i ucięty w czasie zakończenia się metodyRandomSearch). Stąd na oś X można patrzeć jako na % czasu ukończenia się 100 iteracji metody RandomSearch

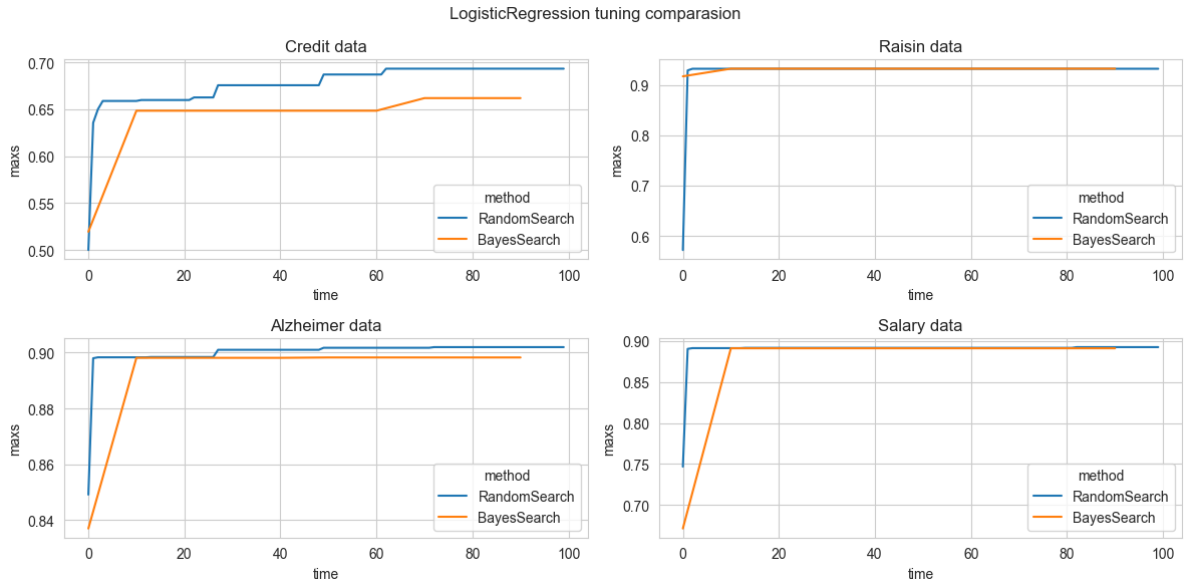

Figure 13: Wykres przedstawiający maksymalną jak dotychczas (biorąc pod uwagę czas) wartość metryki dla algorytmu LogisticRegression.

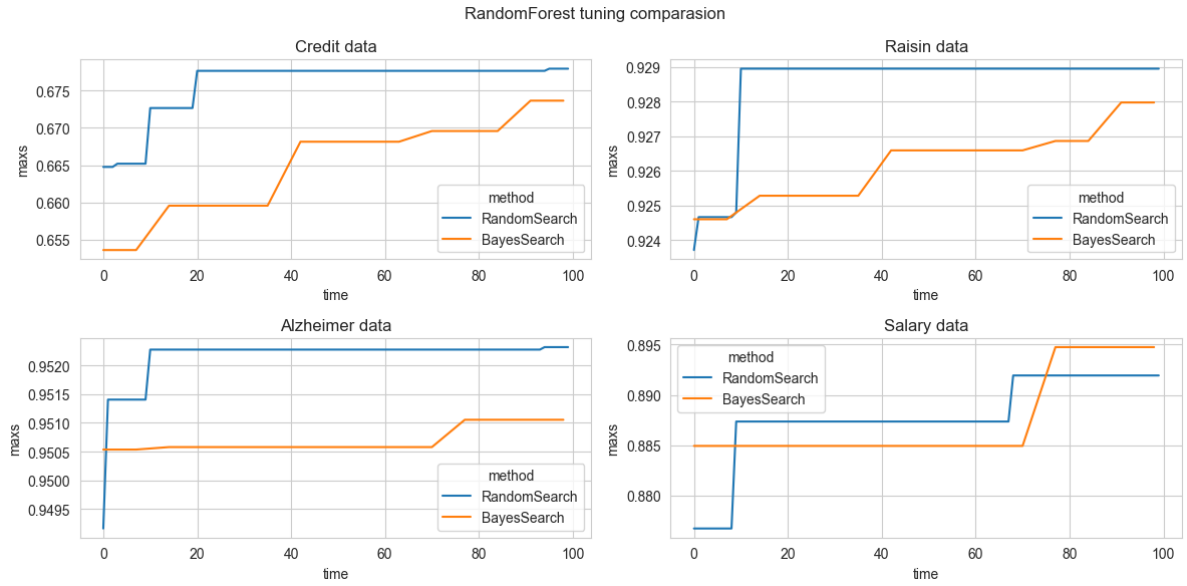

Figure 14: Wykres przedstawiający maksymalną jak dotychczas (biorąc pod uwagę czas) wartość metryki dla algorytmu RandomForestClassifier.

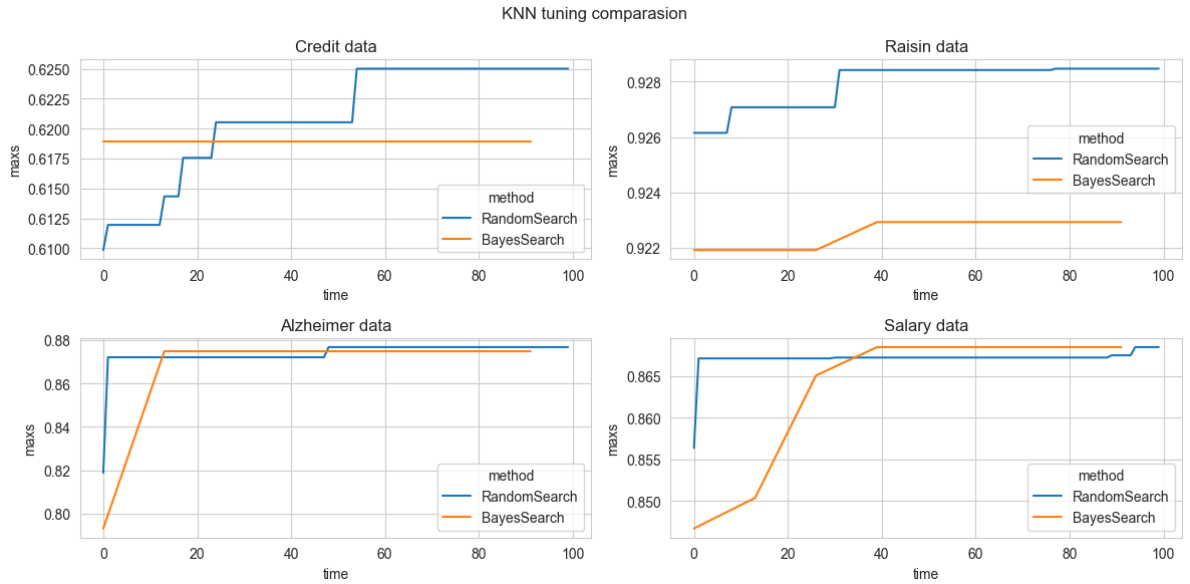

Figure 15: Wykres przedstawiający maksymalną jak dotychczas (biorąc pod uwagę czas) wartość metryki dla algorytmu KNeighborsClassifier.

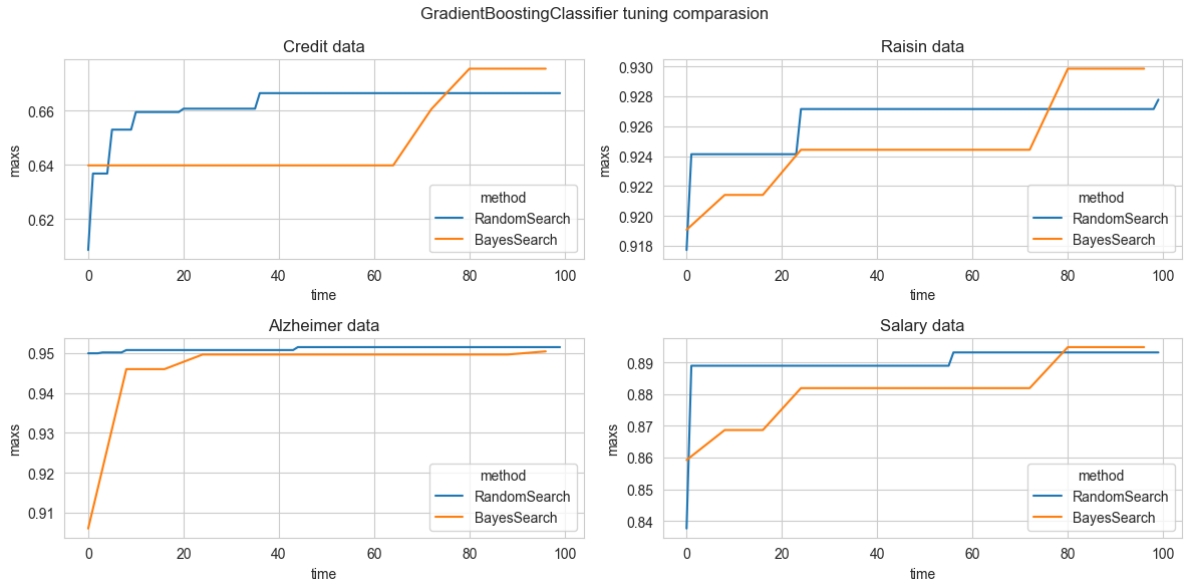

Figure 16: Wykres przedstawiający maksymalną jak dotychczas (biorąc pod uwagę czas) wartość metryki dla algorytmu GradientBoostingClassifier.

6.5 Tunowalność

Wykresy poniżej pokazują przedstawioną w sekcji 3 uzyskaną tunowalność algorytmów w zależności od użytej metody optymalizowania hiperparametrów.

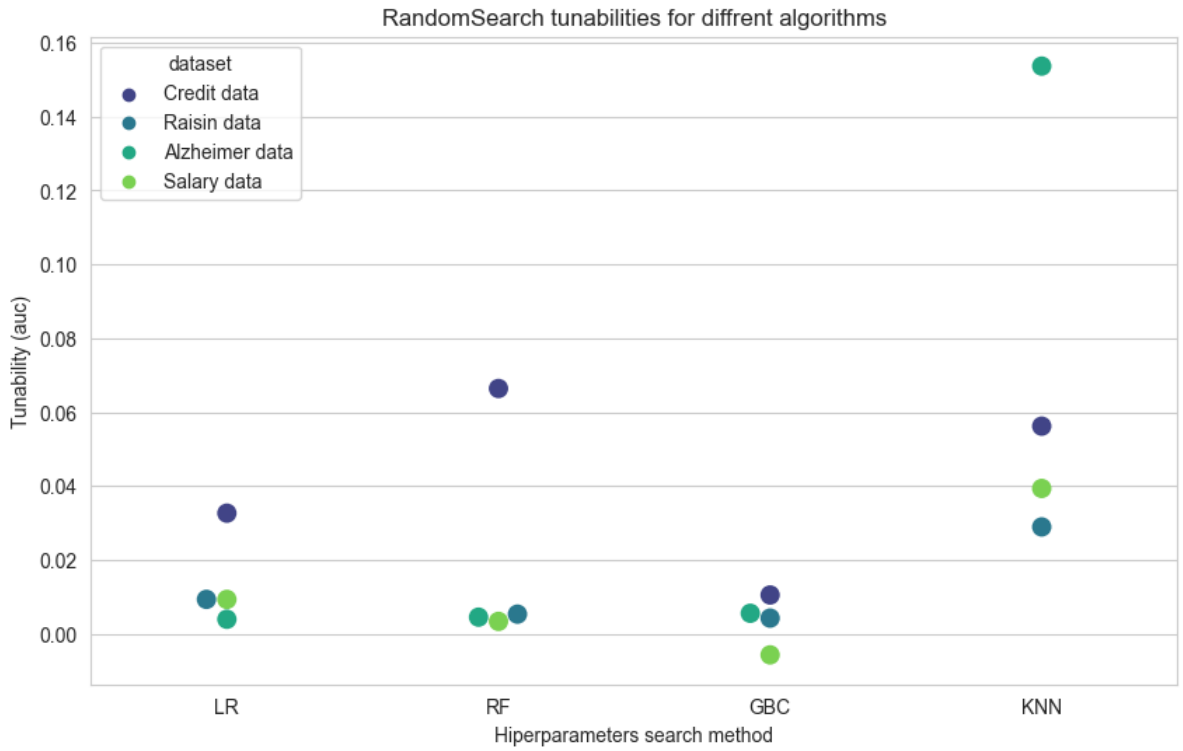

Figure 17: Tunowalność algorytmów dla metody RandomSearch.

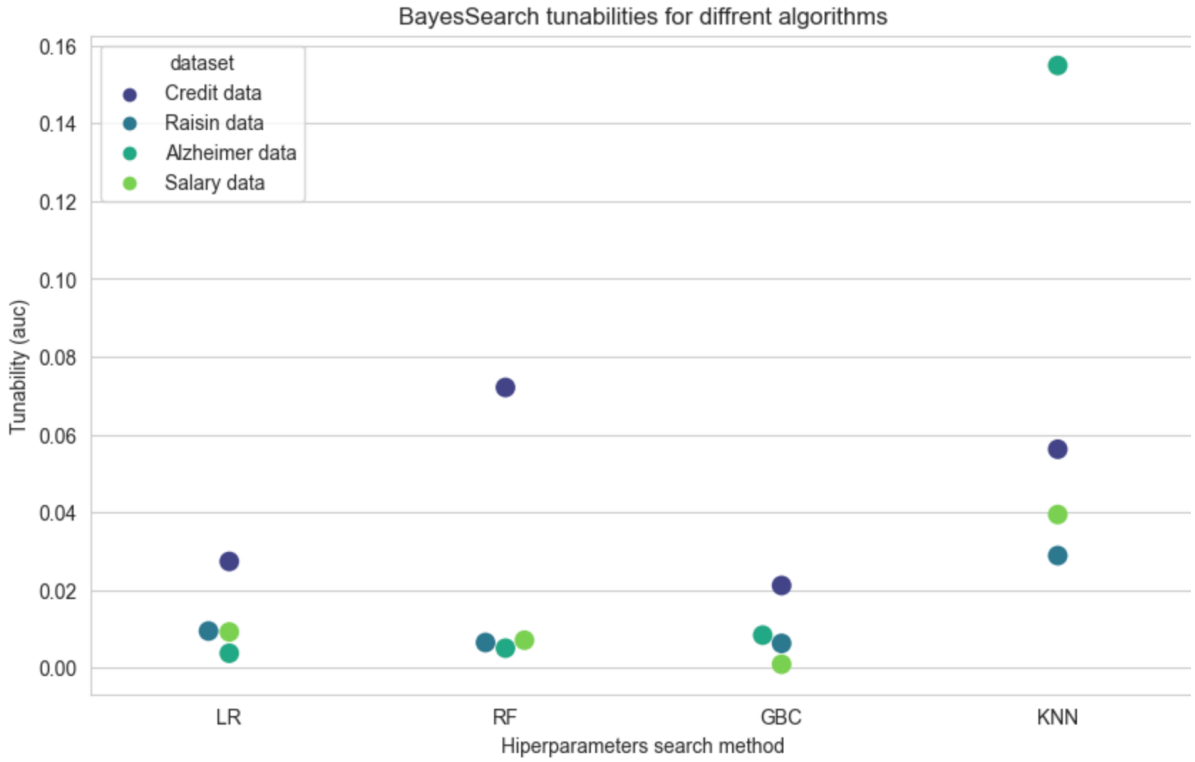

Figure 18: Tunowalność algorytmów dla metody BayesSearch.