Jan Mogielski 303899

Laboratorium 7
Modele Bayesowskie

16 czerwca 2023

Spis treści

1. Cel laboratorium

Celem laboratorium było napisanie kodu w języku Programowania Python i zaimplementowanie naiwnego klasyfikatora Bayesa. Dla sprawdzenia działania klasyfikatora sprawdzimy jak zachowuje się on pod wpływem różnego rodzaju podziału danych oraz k-krotnej walidacji krzyżowej. Klasyfikator będziemy sprawdzać na zbiorze danych Breast Cancer Wisconsin.

2. Zbiór danych

Zestaw danych składa się 569 próbek i 30 różnych cech. W składa wchodzą dwie klasy: Benign oraz Malignant (Łagodny, Złośliwy). Cechy to m.in: radius, texture, perimeter, itd. Dane te są w postaci ciągłej. Na podstawie cech określamy z jakim rakiem piersi mamy do czynienia: łagodny czy złośliwy.

3. Implementacja

Implementacja kodu składa się z części odpowiedzialnej za budowę klasyfikatora, a następnie załadowanie danych i wytrenowanie modelu.

Część odpowiedzialna za naiwny klasyfikator Bayesa jest napisana w klasie NaiveBayesClassifier, w której mamy funkcję:

— def fit - funkcja odpowiedzialna za określenie ilości unikalnych klas oraz obliczenia prawdopodobieństwa wystąpienia klas i cech,

— def calculate_class_probabilities - w tej funkcji zliczamy wystąpienia klas w zestawie danych, obliczamy prawdopodobieństwo wystąpienia danej klasy i je zwracamy,

— calculate_feature_probabilities - funkcja odpowiedzialna za obliczenie średniej i odchylenia standardowego wśród cech zawartych w zestawie danych. Dla każdej klasy sprawdzamy i liczymy wartości dla danych cech. Na końcu zwracamy słownik z wynikami,

— calculate_likelihood - funkcja odpowiedzialna za obliczenie prawdopodobieństwa wystąpienia danej cechy. W tym celu korzystamy ze wzoru:

$$P(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Rys. 1. Wzór na prawdopodobieństwo warunkowe

— predict - funkcja służąca do wykonania przewidywania na podstawie obliczeń wykonanych wcześniej. Tworzymy pustą tablicę, do której będziemy dodawać nasze wyniki. Dla każdej próbki w zestawie danych i każdej klasy zbieramy informacje na temat prawdopodobieństwa klas, cech, obliczamy prawdopodobieństwo wystąpienia danej cechy. Dla zabezpieczania kodu przed dzieleniem przez ”0” dodajemy w kodzie wartość epsilon, który jest bardzo mało, żeby nie zaburzać wyników. Wyliczamy zmienną class_score i dodajmy to do tablicy. Predykcja klasy jest na podstawie najwyższego wyniku, predykcja jest dodawana do tablicy wyników.

Następnie ładujemy zestaw danych i rozgraniczamy go na dane i etykiety, w tym celu wykorzystujemy zestaw danych wbudowywany w bibliotekę scikit-learn. Dla wygody i lepszego zrozumienia danych wypisujemy ilość próbek, ilość cech oraz ilość klas.

Definiujemy interesujące nas podziały zestawu danych za pomocą funkcji train_test_split. Dla każdego podziału danych sprawdzamy skuteczność naszego modelu powtarzając każdą próbę 20 razy w celu uśrednienia wyniku i uzyskania wiarygodnych wyników.

Identyczną metodę stosujemy dla k-krotnej walidacji krzyżowej. Na koniec tworzymy wykresy, które pokazują w jaki sposób zachowują się nasze dane w zależności od wartości train_test_split oraz k.

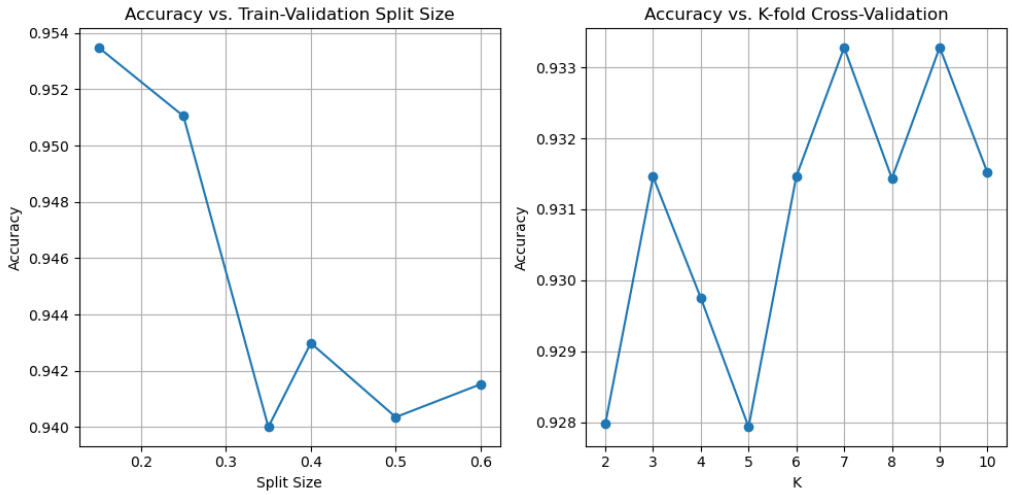
4. Wnioski

Testy zostały przeprowadzone na różnych podziałach danych i wartościach k. Obie metody korzystały z tego samego parametru random_state = 42 w celu prawdziwego porównania wyników.

Dla train_test_split korzystamy z podziałów: [0.15, 0.25, 0.35, 0.4, 0.5, 0.6], a w przypadku k-krotnej walidacji krzyżowej: [2, ..., 11].

```
(base) johnny@Johnnys-MacBook-Air WSI7 % python3 main.py
Dataset size: 569 samples, 30 features
Number of classes: 2
```

Rys. 2. Działanie programu



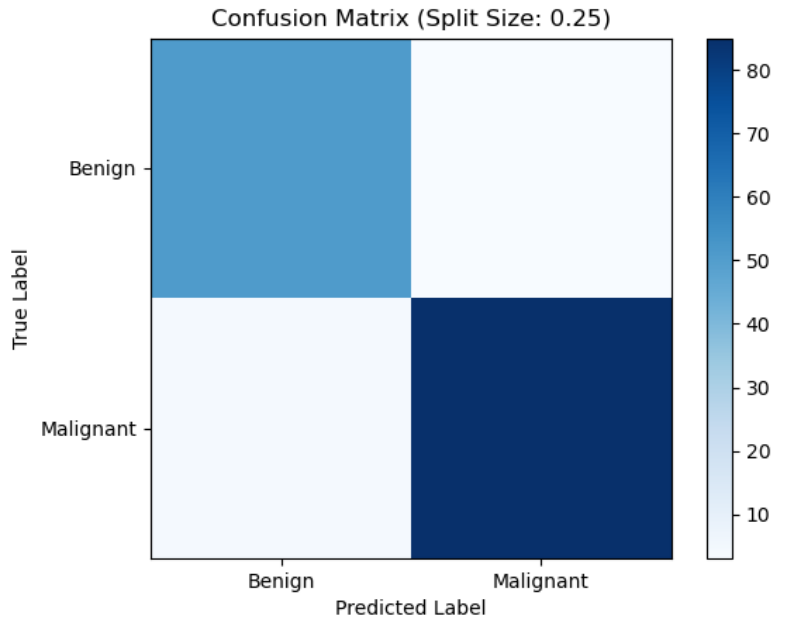
Rys. 3. Wykresy zależności wyników modelu od podziału danych

Jak widać mała wartość train_test_split skutkuje dużą skutecznością modelu, ale jest to złudny wynik ze względu na to, że model ma dużo danych do uczenia, a mało do testowania, więc siłą rzeczy będzie miał dobry wynik. Dużo bardziej reprezentatywne wyniki są dla podziałów większych niż 0.2, w ten sposób otrzymujemy dobry balans dla danych. Widzimy, że wynik modelu stabilizuje się w granicach 94%, co jest bardzo dobrym wynikiem. Jednak za duża wartość podziału doprowadzi do gorszych wyników, ze względu na za małe ilości do wytrenowania modelu.

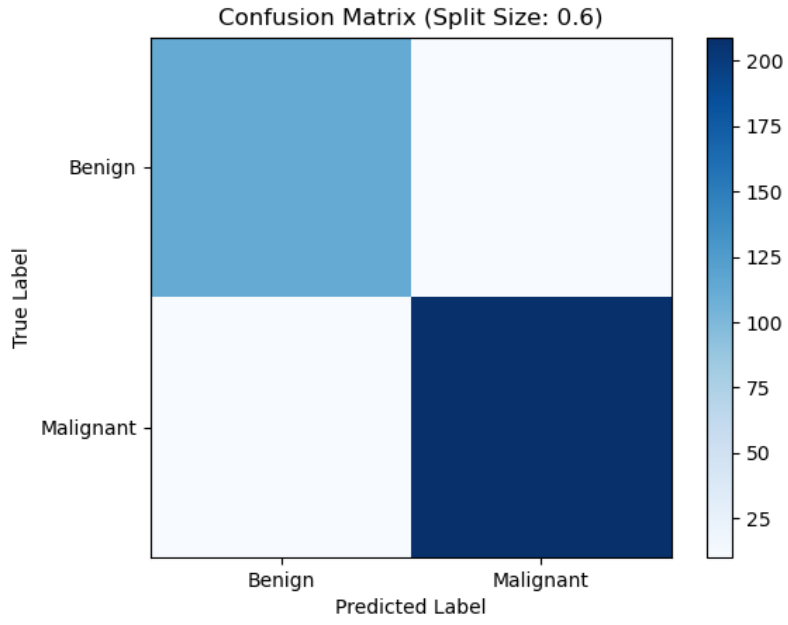
Na odwrót ma się sprawa z k-krotną walidacją krzyżową, w jej przypadku mała wartość k skutkuje dużymi różnicami w wynikach, a końcowy model jest słabo wytrenowany. Im wyższa wartość k, tym wyniki są bardziej spójne i dodatkowo dokładność modelu jest wyższa.

Average Accuracy (Validation Set): 0.9449
Average Accuracy (Cross-validation): 0.9309

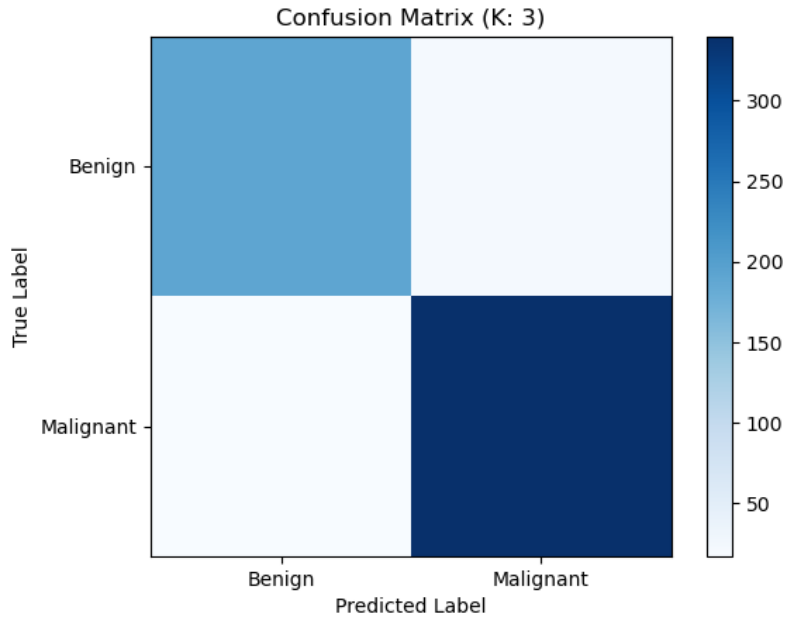
Dodatkowo w ramach test stworzyłem macierze błędów, żeby sprawdzić jak się zachowuje dany podział dla naszych danych. Pokażę przykładowe macierze błędu, ale przy tak wysokiej skuteczności są jedynie potwierdzeniem wcześniejszych obserwacji.



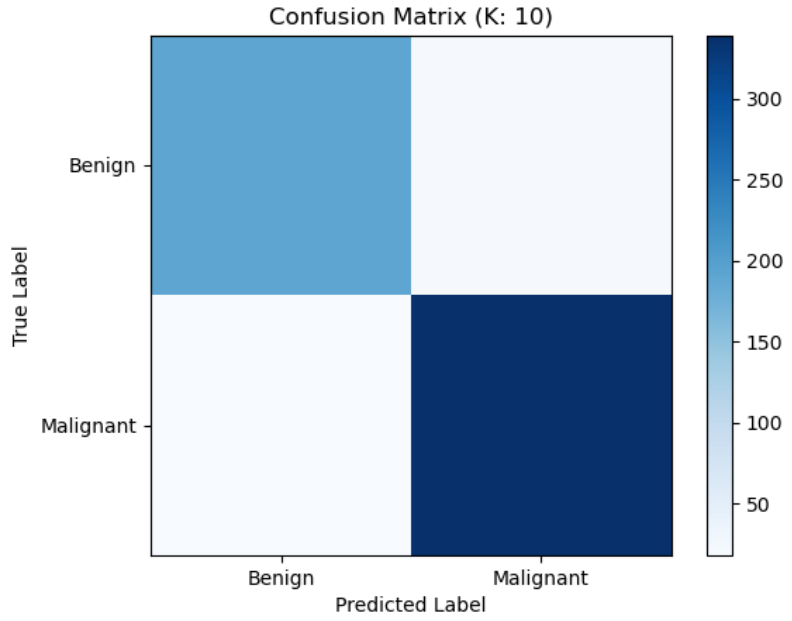
Rys. 4. Macierz błędów dla podziału train_test_split = 0.25



Rys. 5. Macierz błędów dla podziału train_test_split = 0.60



Rys. 6. Macierz błędów dla podziału k=3



Rys. 7. Macierz błędów dla podziału k=10

Macierze błędów potwierdzają poprawne działanie klasyfikatora, różnice są tak małe że gołym okiem nie da się zauważyć zmian w macierzach, należałoby dodać wartości w celu sprawdzenia różnic.

Laboratorium przebiegło pomyślnie, udało się zaimplementować naiwny klasyfikator Bayesa i uzyskać bardo dobre wyniki.