

# Uczenie maszynowe - klasyfikacja

Kamil Niemczyk

6 maja 2024

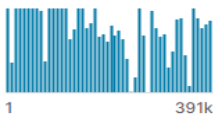
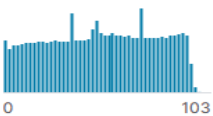

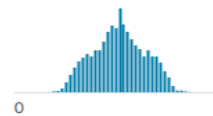
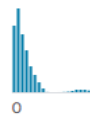
## 1 Wstęp

Zadanie polegało na przebadaniu wybranej bazy danych pod kątem klasyfikacji. Na podstawie bazy danych ze statystykami oddanych strzałów przez piłkarzy przeprowadziłem klasyfikację w celu zgadywania procentu oczekiwanej strzelonej bramki przez zawodnika.

## 2 Projekt

### 2.1 Baza danych

Baza danych oryginalna z której skorzystałem znajduje się na stronie kaggle spod adresem <https://www.kaggle.com/datasets/shushrutsharma/top-5-football-leagues?select=FullShotsData.csv>. Oryginalne dane wyglądają posiadają 20 kolumn. Każda kolumna posiada informacje na temat oddanego strzału przez zawodnika. Wśród informacji jakie można znaleźć w tej bazie danych są: id zawodnika w której został oddany strzał, minuta spotkania, rezultat strzału (czy został zablokowany, nietrafiony lub trafiony), długość boiska w skali od 0 do 1, szerokość boiska w skali od 0 do 1, procent oczekiwanej bramki, imię i nazwisko zawodnika oddającego strzał i wiele innych interesujących statystyk. Niektóre z nich są w formie liczbowej takie jak np. rok w którym odbywało się dane spotkanie lub szerokość boiska z której został oddany strzał. Reszta jest w formie tekstowej np. imię i nazwisko zawodnika lub rezultat oddanego strzału. Baza danych posiada około 300000 wierszy. Poniżej poglądowe zdjęcie jak ta baza danych wygląda.

id	minute	result	X	Y	xG
MatchID of that specific match	Minute at which the shot was taken (remember that one of the downsides of understat data is that it doesn't differentiate	Outcome of that shot. Examples →Goal, MissedShots, SavedShot, MissedShots, Goal, BlockedShot	The understat pitches classifies the pitch as 100 units by 100 units (both length and width). X represents the length. 0	Represents the width. 0 to 100 is from bottom to top	Expected that speci
		<div>MissedShots 39%</div> <div>BlockedShot 25%</div> <div>Other (105511) 36%</div>			
1 391k	0 103		0 1	0 1	0
378451	20	BlockedShot	0.875999984741211	0.602000007629395	0.019478
378458	54	MissedShots	0.878000030517578	0.43	0.031428
378464	77	MissedShots	0.858000030517578	0.679000015258789	0.060696
379954	34	MissedShots	0.830999984741211	0.655	0.127709
379956	46	MissedShots	0.943000030517578	0.5	0.445353
381490	3	Goal	0.915999984741211	0.524000015258789	0.510213
381496	21	Goal	0.888000030517578	0.579000015258789	0.119856
381504	38	Goal	0.956999969482422	0.482000007629395	0.529748
381511	71	SavedShot	0.853000030517578	0.527999992370605	0.280143

Baza danych

## 2.2 Preprocessing

Baza danych, z której korzystałem nie posiadała w sobie błędów takich jak puste wartości czy różne typy danych, aczkolwiek stwierdziłem, że wstępnie nie potrzebna jest mi tak duża ilość kolumn, więc ograniczyłem ją do kolumn: player id, situation, X, Y, shotType, home team, away team, year i xG. Następnie danym, które były wartości string przypisałem indexy i dzięki temu dane tekstowe przekształciłem w liczby. Takimi danymi, które potrzebowały takiej przeróbki były: situation, shotType, home team i away team. Następnie dane xG (procent oczekiwanej bramki) zaokrągliłem do części dziesiętnej i podzieliłem dane na dane których wyniki będę chciał zgadywać xG i dane na podstawie których będę wyciągał wnioski: player id, situation, X, Y, shotType, home team, away team i year. Kolejną potrzebną rzeczą było przekonwertowanie wartości xG z wartości liczbowej na wartość tekstową, żeby stworzyć coś na kształt etykiety. Odpowiednio dla wartości przypisałem im taką wartość tekstową np. liczba 0.2 dostała nową wartość "0.2". Dane X (czyli dane bez wartości xG) przekształciłem następnie za pomocą MinMaxScaler() na wartości przeskalowane od 0 do 1. Potem za pomocą PCA zredukowałem ilość kolumn potrzebnych. Okazało się że pierwsze 6 kolumn wystarczy, żeby zachować 94% informacji, więc ostatnie 2 kolumny, które zawierały informacje na temat roku w, którym odbywał się mecz i drużyna która w tym meczu grała na wyjeździe zostały usunięte co skutkowało usunięciem 5% informacji.

```
#PCA
pca_data = PCA().fit(X)
cumulative_variance = np.cumsum(pca_data.explained_variance_ratio_)
print(cumulative_variance) #shows how much information is in each column
n_components_95 = np.argmax(cumulative_variance >= 0.94) + 1 #shows how many columns we need to have 95%
information_loss = 1 - np.sum(pca_data.explained_variance_ratio_[:n_components_95]) #shows how much info
# print(n_components_95)
# print(information_loss)
X = PCA(n_components=n_components_95).fit_transform(X)
# print(X) #PCA dataset
```

Zobrazowanie nie wygląda fragment kodu z PCA

## 2.3 Klasyfikacja

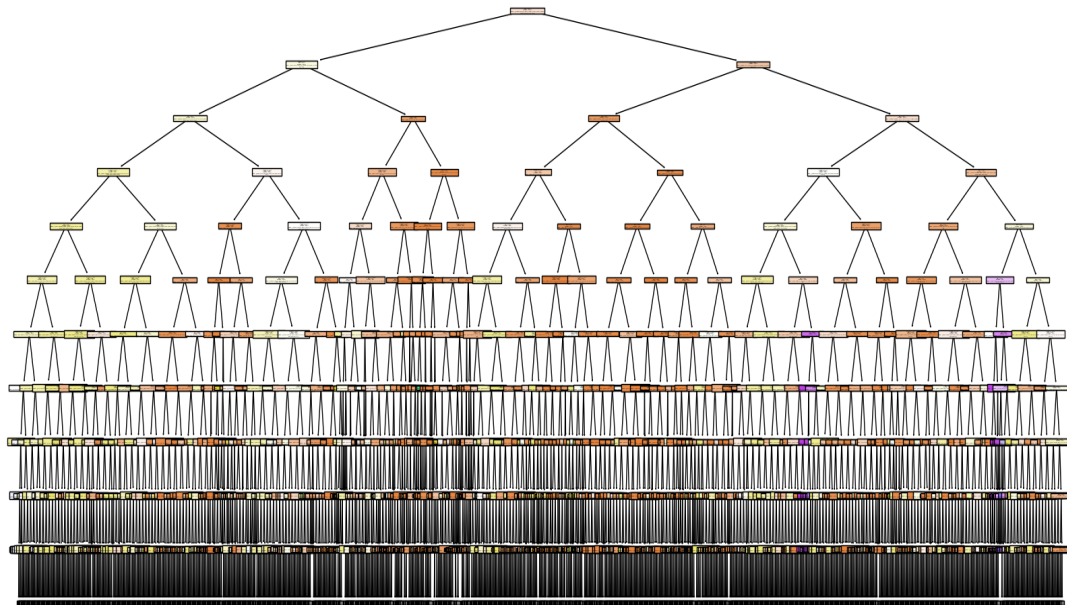
### 2.3.1 Drzewo decyzyjne

Klasyfikację zacząłem od podzielenia danych na dane treningowe jak i testowe w skali 80:20. Następnie na tych danych sprawdziłem jak wypadnie przeprowadzenie klasyfikacji za pomocą drzewa decyzyjnego.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=56)
DecisionTreeClassifier
tree = DecisionTreeClassifier() # DecisionTreeClassifier
tree.fit(X_train, y_train)
print("DecisionTreeClassifier score")
print(tree.score(X_test, y_test))
```

Fragment kodu przedstawiający budowę drzewa decyzyjnego

Wyniki można uznać za satysfakcjonujące, ponieważ za pomocą takiego drzewa można było w 68% dobrze określić wartość xG. Poniżej zobrazowanie wyglądu drzewa dla max głębokości 10.



Aktywuj system Windows  
Przejdź do ustawień, aby aktywować system Windows.

Drzewo max-depth=10

### 2.3.2 K najbliższych sąsiadów

Następnie stwierdziłem, że można spróbować wytrenować model k-NN na różnych ilościach sąsiadów.

```
# k-NN, k=3
neighbours = KNeighborsClassifier(n_neighbors=3)
neighbours_fit = neighbours.fit(X_train, y_train)
predict = neighbours_fit.predict(X_test)
correct = accuracy_score(y_test, predict)
print("k-NN, k=3")
print(correct)
print(confusion_matrix(y_test, predict))
```

Fragment kodu przedstawiający budowanie modelu dla 3 sąsiadów

Wyniki w były testowane na ilości 3, 5, 11 i nie różniły się od siebie znacznie. Wynosiły kolejno 59%, 60% i delikatnie więcej niż 60%, więc można zauważyć, że przy wzroście liczby sąsiadów delikatnie wzrasta procent poprawnych wyników. Na końcu wygenerowałem jeszcze macierz błędów każdego z modeli.

```
[ [22997 6157 32 76 75 35 15 2 0 0 0]
 [ 8648 11121 44 229 260 101 55 3 1 0 1]
 [ 325 345 5 9 11 2 2 0 0 0 0]
 [ 806 1210 5 45 40 11 8 0 1 0 0]
 [ 631 1243 6 52 65 37 15 1 0 0 0]
 [ 395 845 5 28 55 39 9 5 1 0 0]
 [ 239 598 4 14 31 24 26 3 0 0 0]
 [ 71 183 4 5 8 14 14 137 18 1 0]
 [ 38 82 2 7 4 4 3 21 567 0 0]
 [ 23 74 0 3 6 3 4 0 0 0 0]
 [ 20 36 1 1 1 7 2 1 1 0 0]]
```

Fragment kodu przedstawiający macierz błędów dla 3 sąsiadów

Na podstawie tej macierzy można zauważyć, że rzeczywiście liczby idą regularnie po przekątnej co oznacza, że wyniki są poprawne. Widać też bardzo częste określanie wartości automatycznie jako niskie (pierwsze 2 kolumny oznaczają jak często wyniki wynosiły 0.0 lub 0.1). Jako ciekawostkę można zauważyć bardzo dziwną zależność, że stosunkowo łatwo było stwierdzić, że xG wynosi 0.8 co pokazuje wynik 567, który znacznie różni się od innych wyników z tego wiersza.

### 2.3.3 Naive Bayes

Jako następny klasyfikator wybrałem Naive Bayes, który jest probabilistycznym klasyfikatorem, która zakłada niezależność między cechami.

```
# Naive Bayes
gnb = GaussianNB()
gnb_fit = gnb.fit(X_train, y_train)
predict = gnb_fit.predict(X_test)
correct = accuracy_score(y_test, predict)
print("Naive Bayes")
print(correct)
print(confusion_matrix(y_test, predict))
```

Fragment kodu przedstawiający tworzenie Naive Bayes

Wynik wynosił 51%, więc był najgorszy ze wszystkich klasyfikatorów.

```
[ [24842 4366 0 0 0 0 0 0 181 0 0]
 [15371 4874 0 0 0 0 0 0 218 0 0]
 [ 462 210 0 0 0 0 0 0 27 0 0]
 [ 1679 447 0 0 0 0 0 0 0 0 0]
 [ 1567 483 0 0 0 0 0 0 0 0 0]
 [ 1016 365 0 0 0 0 0 0 1 0 0]
 [ 705 234 0 0 0 0 0 0 0 0 0]
 [ 248 53 0 0 0 0 0 0 154 0 0]
 [ 110 29 0 0 0 0 0 0 589 0 0]
 [ 95 18 0 0 0 0 0 0 0 0 0]
 [ 55 15 0 0 0 0 0 0 0 0 0]]
```

Macierz błędów Naive Bayes

Po macierzy widać, że ten klasyfikator po prostu za każdym razem stwierdzał, że wynik wynosi 0.0, 0.1 lub 0.8.

### 2.3.4 Sieć neuronowa

Stwierdziłem, że jako ostatni klasyfikator spróbuję skorzystać z sieci neuronowej.

```
12
13 clf = MLPClassifier(hidden_layer_sizes=(30, 30, 30), max_iter=500, alpha=0.0001, solver='adam', verbose=0)
14 clf.fit(X_train, y_train)
15 # Save the trained model
16 joblib.dump(clf, 'neural_model.pkl')
17 print(clf.score(X_test, y_test))
18
19
20
```

Fragment kodu przedstawiający budowę sieci

Testowana sieć była na 3 warstwach po 100 neuronów, na 4 warstwach po 100 neuronów i na 3 warstwach po 30 neuronów. Pomiedzy tymi wynikami nie było zauważalnej różnicy. Zawsze po około 260 iteracjach sieć przestała się uczyć, ponieważ w ciągu 10 epok nie został zmniejszony średni błąd predykcji o 0.0001%. Średnia wartość predykcji po przeprowadzeniu tych 3 eksperymentów wynosiła 77% skuteczności.

## 2.4 Reguły asocjacyjne

Do przeanalizowania reguł asocjacyjnych musiałem skonwertować dane od nowa, żeby każda kolumna zawierała dane typu string.

```
9 dataset = pd.read_csv('FullShotsData.csv')
10 data = dataset[['player_id', 'situation', 'X', 'Y', 'shotType', 'h_team', 'a_team', 'xG']]
11 data = pd.DataFrame(data)
12 data = data.round(1)
13 data = data.astype(str)
14
15 data_encoded = pd.get_dummies(data)
16 data_encoded = data_encoded.astype(bool).astype(int)
17 # print(data_encoded.head())
18
19 frequent_itemsets = apriori(data_encoded, min_support=0.2, use_colnames=True)
20 # print(frequent_itemsets)
21
```

Fragment kodu

Użyłem do tego algorytmu Apriori który wykrywa zależności pomiędzy częstymi występowaniami różnych zestawów elementów.

```

pe
warnings.warn(
    support
                                itemsets
0  0.730876                      (situation_OpenPlay)
1  0.304098                      (X_0.8)
2  0.479149                      (X_0.9)
3  0.304263                      (Y_0.5)
4  0.205098                      (Y_0.6)
5  0.314264                      (shotType_LeftFoot)
6  0.515209                      (shotType_RightFoot)
7  0.504574                      (xG_0.0)
8  0.350053                      (xG_0.1)
9  0.239470                      (X_0.8, situation_OpenPlay)
10 0.340055                      (X_0.9, situation_OpenPlay)
11 0.253422                      (shotType_LeftFoot, situation_OpenPlay)
12 0.405403                      (shotType_RightFoot, situation_OpenPlay)
13 0.368045                      (xG_0.0, situation_OpenPlay)
14 0.255401                      (xG_0.1, situation_OpenPlay)
15 0.224840                      (xG_0.0, X_0.8)
16 0.205105                      (X_0.9, shotType_RightFoot)
17 0.242894                      (xG_0.1, X_0.9)
18 0.270637                      (xG_0.0, shotType_RightFoot)
19 0.213610 (xG_0.0, shotType_RightFoot, situation_OpenPlay)
PS C:\Users\kamil\Desktop\FootballPrediction>

```

### Zależności

Na podstawie tych zależności można wywnioskować, że 73% strzałów jest oddawana ze zwykłej gry a nie żadnych stałych fragmentów. Można też zauważyć dużą przewagę oddawanych strzałów prawą nogą nad oddawanymi strzałami lewą nogą. Ostatnim motywem o którym warto wspomnieć jest to, że strzały w około 78% są oddawane od około 25 metrów do 5 metrów od bramki.

## 3 Podsumowanie

No podstawie tych wyników można stwierdzić, że sieć neuronowa poradziła sobie najlepiej pod względem klasyfikacji w stosunku do innych klasyfikatorów. Najgorzej zaś poradził sobie klasyfikator Naive Bayes. Niestety reguły asocjacyjne nie wskazały nam żadnych ciekawych zależności oprócz pojedynczych, które często na siebie wpływały. Być może wyniki byłyby bardziej interesujące gdyby zamiast klasyfikować wyniki obliczałyby się ich wartość regresji.