Raport z tworzenia modelu klasyfikacyjnego

Metody Systemowe i Decyzyjne

Wiktor Kubera

Spis treści

1. Wstęp

1.1. Zdefiniowanie problemu badawczego

Problemem badawczym jest klasyfikacja **aplikacji jako płatnej lub bezpłatnej** na podstawie jej cech.

1.2. Narzędzia

1.2.1. Scraper i dane

Na początku szukałem narzędzia, które pozwoli mi pozyskać pewne dane na temat aplikacji znajdujących się na platformie google play, znalazłem **scraper**, który można wykorzystać używając pythona, tj. `https://pypi.org/project/google-play-scraper/`. Korzystając z dokumentacji, zrobiłem research na temat danych, które można pozyskać z użyciem tego narzędzia, jest ich dużo, ale sporo z nich to dane tekstowe, **dane tekstowe ciężko przerobić na liczby, także szukałem danych liczbowych.**
Udało mi się znaleźć w internecie dane - `https://www.appbrain.com/stats/android-market-app-categories`, na podstawie których mógłbym przerobić parametr tekstowy "**Kategoria**" na liczbę, tj. znalazłem zestawienie kategorii wraz z ilością aplikacji z tej kategorii oraz ilością płatnych aplikacji z tej kategorii, wówczas parametr "Kategoria" będzie liczbą aplikacji płatnych do liczby wszystkich aplikacji. Po wstępnej analizie, jako **parametry opisujące aplikację** wybrałem:

- **installs** - liczba pobrań aplikacji
- **score** - ocena aplikacji
- **ratings** - liczba ocen
- **reviews** - liczba opinii
- **categoryProb** - liczba przypisana do kategorii: $\frac{paidApps}{allApps}$

Jako **parametr predykowany** wybrałem:

- **isFree** - czy aplikacja jest darmowa czy nie

1.2.2. Narzędzia uczenia maszynowego

Skoro problem jest binarny, tzn. aplikacja może być płatna albo darmowa, to jest to problem **klasyfikacji binarnej**.
Do zbudowania modelu wykorzystam algorytm, który poznaliśmy na zajęciach - **SVM**.

2. Zdobycie danych

2.1. Problemy

Okazało się, że scraper, który jest dostępny w pythonie, jest nakładką na scraper, który został napisany w NodeJs.
Myślałem, że udostępnia identyczne funkcjonalności, co oryginalny scraper, więc zacząłem testować jego możliwości.

2.1.1. Brak możliwości wyszukiwania aplikacji po kategoriach

Scraper w pythonie nie daje możliwości wyszukiwania po kategoriach, co jest bardzo dużym problemem. Niemniej jednak poeksperymentowałem i okazało się, że jeśli **jako query dam kategorię**, to scraper zwraca mi dane aplikacji właśnie z tej kategorii.

2.1.2. Ograniczona ilość aplikacji

Okazało się, że scraper zwraca **maksymalnie 30 aplikacji**, jako rezultat zapytania, co jest słabym wynikiem, ponadto zwraca zawsze te same aplikacje oraz istnieje limit zapytań, które można wykonać w okresie pewnego przedziału czasowego.
Planowałem, aby zbiór danych zawierał dane na temat **25000-50000** aplikacji.
Z wykorzystaniem scrapera w pythonie to nie było możliwe, tak więc postanowiłem wykorzystać **oryginalny scraper napisany w NodeJs**.

2.1.3. Niezgodność kategorii

Scraper w NodeJs udostępnia inne kategorie, niż te, które znalazłem na internecie wraz z danymi potrzebnymi do wyliczenia **categoryProb**.
Scraper udostępnia **54 kategorie**, gdzie **23 kategorie nie posiadają categoryProb**.
Postanowiłem nie brać pod uwagę tych kategorii, dla których nie ma categoryProb.

2.1.4. Ograniczone wyszukiwanie

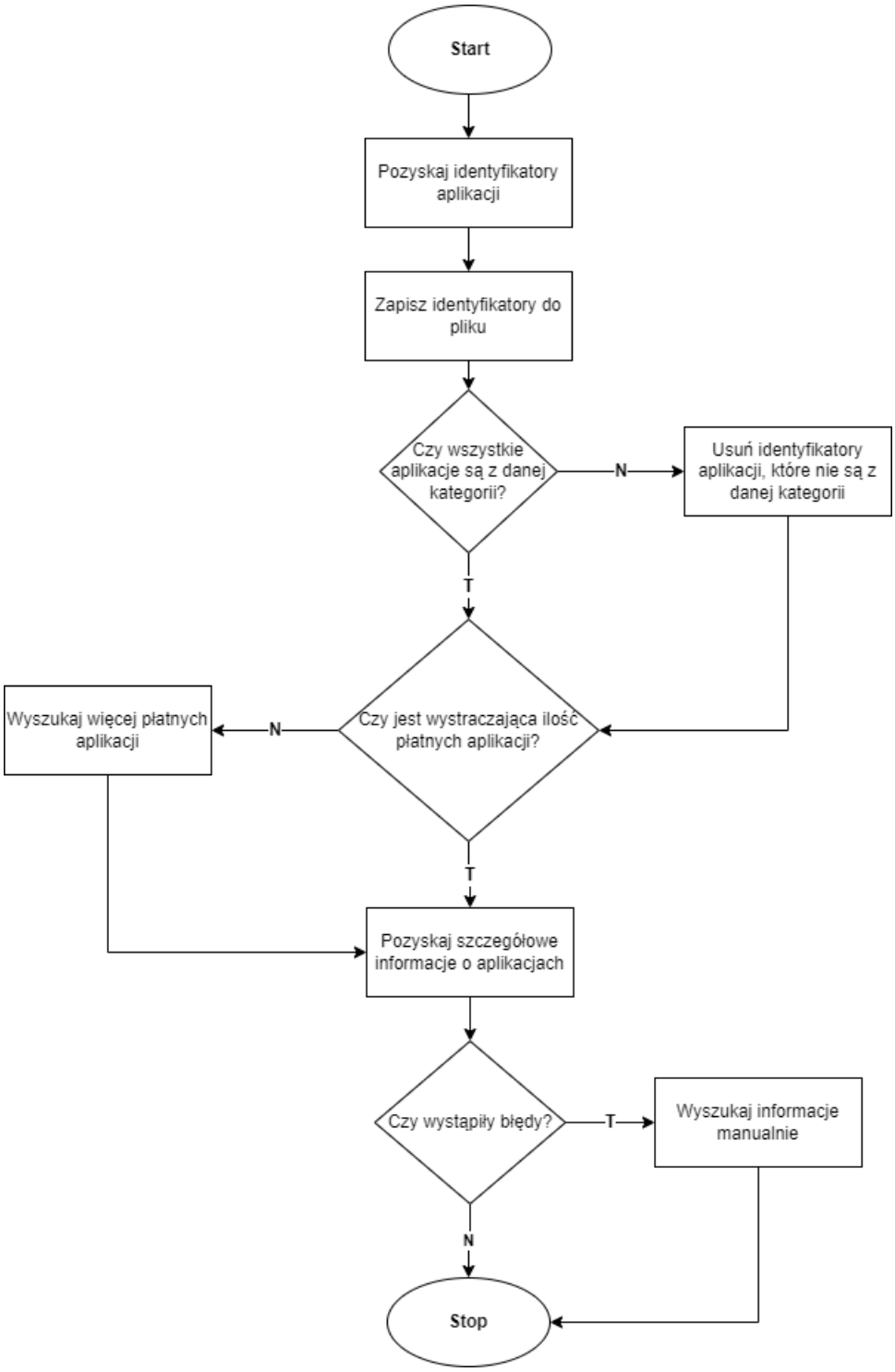
Scraper w NodeJs oferuje wyszukiwanie po kategoriach, co jest bardzo pomocnym narzędziem, niemniej jednak związane są z tym 3 problemy:

- Limit 200 aplikacji zwracanych jako wynik zapytania
- Zwracane są zawsze te same aplikacje
- Przy wyszukiwaniu na podstawie kategorii, nie można jednocześnie wyszukiwać po słowach kluczowych

Po kilku eksperymentach zauważyłem, że **wraz z kategorią mogę jednocześnie wyszukiwać aplikacji z danego kraju**, tak więc zmodyfikowałem proces wyszukiwania, tak, aby dla danej kategorii próbował wyszukiwać dla wszystkich możliwych krajów. Dzięki temu dla pierwszej kategorii udało mi się zwiększyć ilość wyszukanych aplikacji z 350 do **2000+**

2.2. Pozyskiwanie danych

Mając już opracowaną strategię, na początku pozyskuję identyfikatory aplikacji, które są z danej kategorii, zapisuję je w pliku, a następnie korzystam z metod wyszukiwania po identyfikatorze, w celu poznania szczegółowych parametrów aplikacji.
Szczegółowe informacje o aplikacjach zapisuję w osobnym pliku w formacie csv, przy tworzeniu modelu będę z niego korzystał z poziomu pythona.
Przy próbie wyszukania szczegółowych informacji na temat niektórych aplikacji występuje problem, który czasem udaje się załatwić manualnym wyszukaniem aplikacji (poza pętlą).
Po zebraniu wystarczającej ilości identyfikatorów z danej kategorii sprawdzam, czy na pewno każda aplikacja jest z podanej kategorii oraz sprawdzam, czy jest wystarczająca ilość płatnych aplikacji (gdyż jest ich zazwyczaj znacznie mniej).
Algorytm, który obrazuje schemat działania pozyskiwania danych dla danej kategorii można przedstawić za pomocą poniższego schematu blokowego.


Start
Pozyskaj identyfikatory aplikacji
Zapisz identyfikatory do pliku
Czy wszystkie aplikacje są z danej kategorii?
N
Usuń identyfikatory aplikacji, które nie są z danej kategorii
T
Czy jest wystraczająca ilość płatnych aplikacji?
N
Wyszukaj więcej płatnych aplikacji
T
Pozyskaj szczegółowe informacje o aplikacjach
Czy wystąpiły błędy?
T
Wyszukaj informacje manualnie
N
Stop

3. Przygotowanie i wyczyszczenie danych

3.1. Wyczyszczenie danych

Udało mi się zebrać dane na temat **143310** aplikacji, niemniej jednak w procesie scrapowania, kilka aplikacji zawiera braki w danych.
Aplikacji, które nie zawierają kompletu potrzebnych danych jest 120, jest to bardzo mała liczba w porównaniu do aplikacji "kompletnych", tak więc postanawiam je usunąć z pomocą skryptu.

3.2. Przygotowanie danych

Dane wczytuję z pliku csv za pomocą bilblioteki pandas.
Dane są przetrzymywane w strukturze **DataFrame**.

4. Analiza danych

4.1. Zawartość

W przygotowanym zbiorze danych znajduje się **143188** wierszy, gdzie każdy wiersz reprezentuje pojedynczą aplikację i jej parametry.
Każdy wiersz jest kompletny, tzn. nie posiada pustych parametrów.
Dane dotyczące poszczególnych parametrów są bardzo różnorodne, przedstawię je w postaci tabelki.

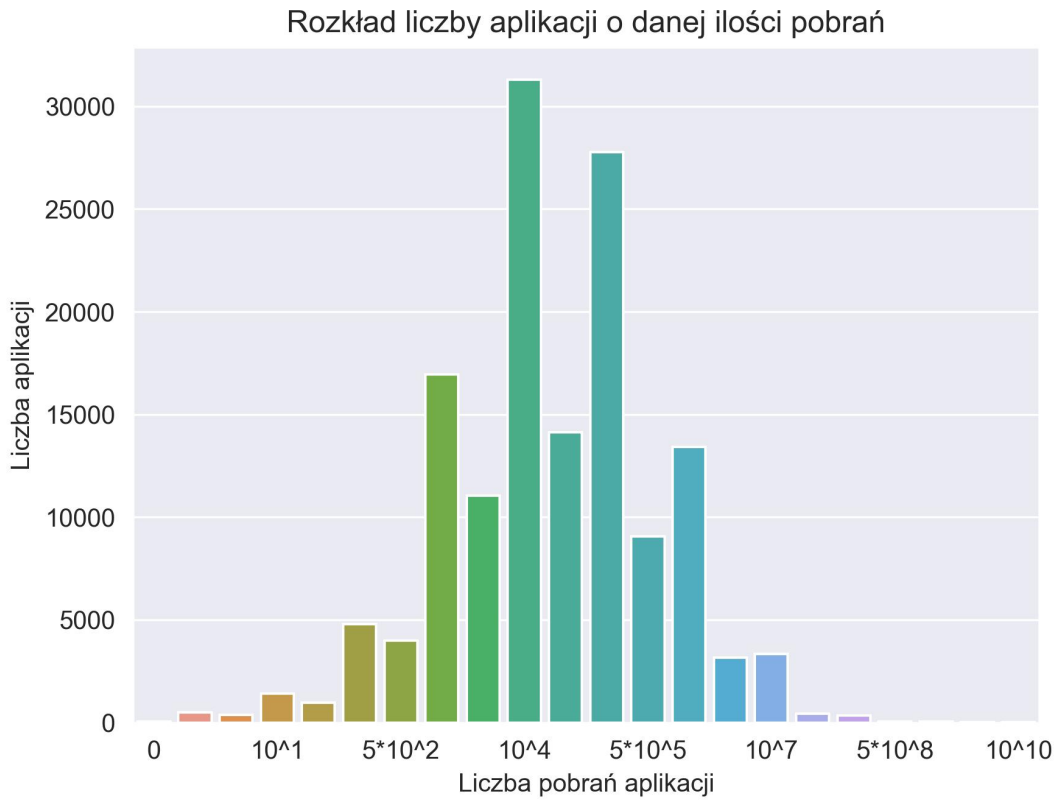
Parametr	Min	Max	Średnia	Odch. Standardowe
Installs	0	10000000000	2042198	69821755
Score	0	5	1.918	2.083
Ratings	0	176546332	30229	851576
Reviews	0	4469629	928	23845
isFree	0	1	0.9336	0.2488

Dodatkowo, **darmowych aplikacji jest 133695** a **płatnych 9494**. Natomiast zestawienie kategorii, do których należą aplikacje lepiej przedstawi wykres.

4.2. Wizualizacje

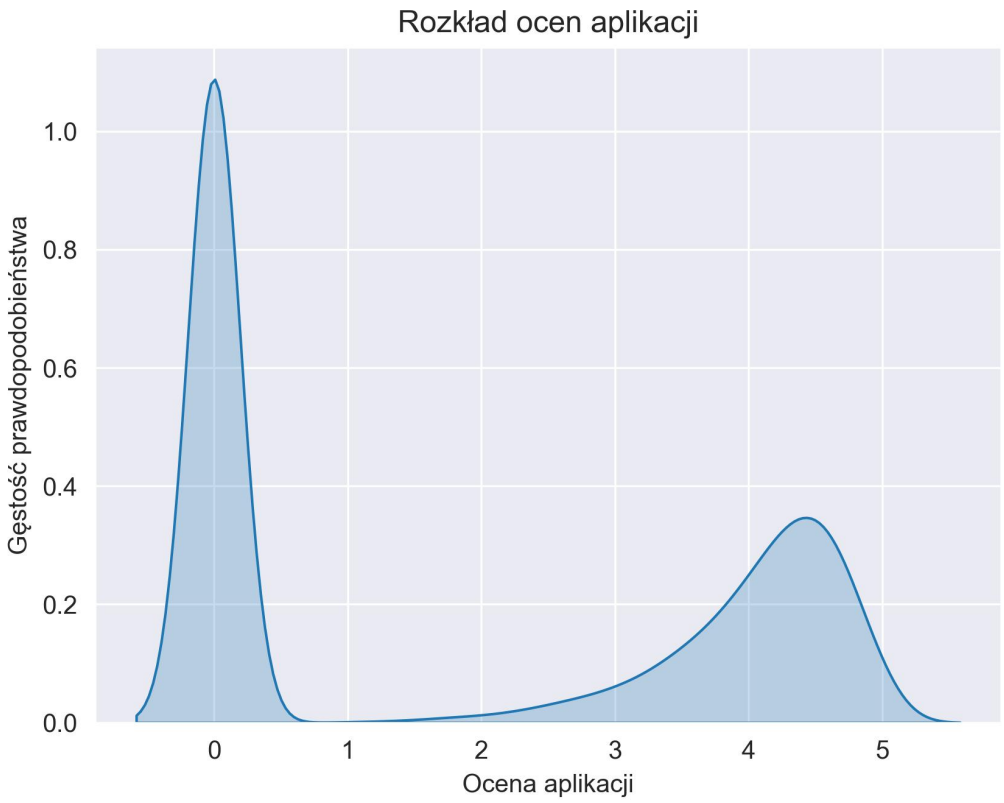
Przygotowałem kilka wykresów, które pozwolą zobrazować zawartość zbioru danych i tego, jakie dane zawiera.

4.2.1. Liczba aplikacji o danej liczbie pobrań



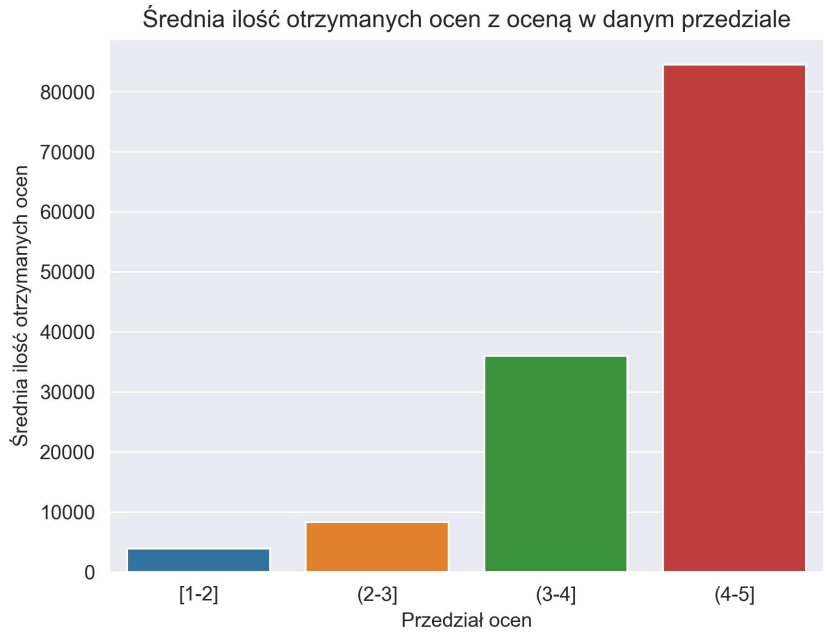
Wykres przedstawia zestawienie liczb aplikacji o danej ilości pobrań, najwięcej aplikacji ma liczbę pobrań rzędu 10^4

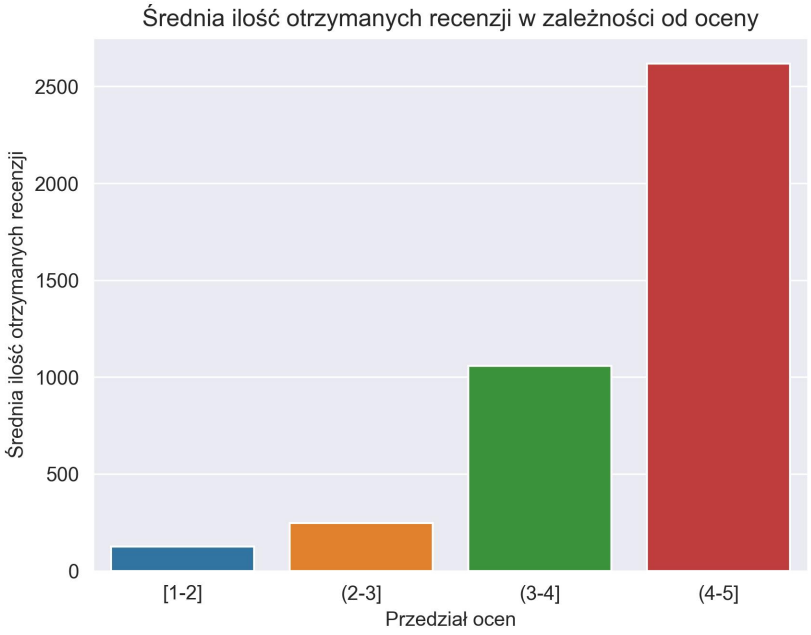
4.2.2. Rozkład ocen aplikacji



Wykres przedstawia rozkład gęstości ocen aplikacji.
Z wykresu wynika, że w zbiorze danych jest **najwięcej aplikacji, które mają ocenę 0**.
Ocena 0 jest związana z brakiem ocen, tj. nikt jeszcze nie ocenił tej aplikacji.
Jeśli chodzi o aplikacje, które mają już ocenę, to najwięcej aplikacji ma ocenę około 4.3 - 4.6.

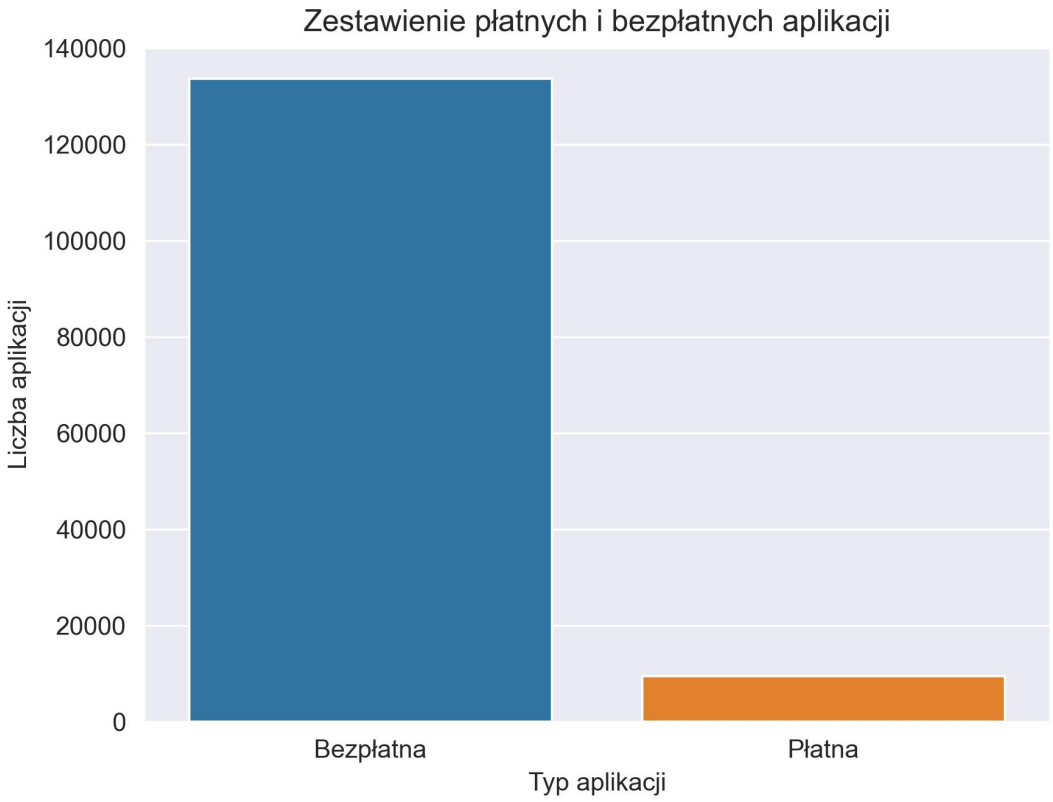
4.2.3. Związek ilości ocen oraz ilości recenzji z oceną aplikacji





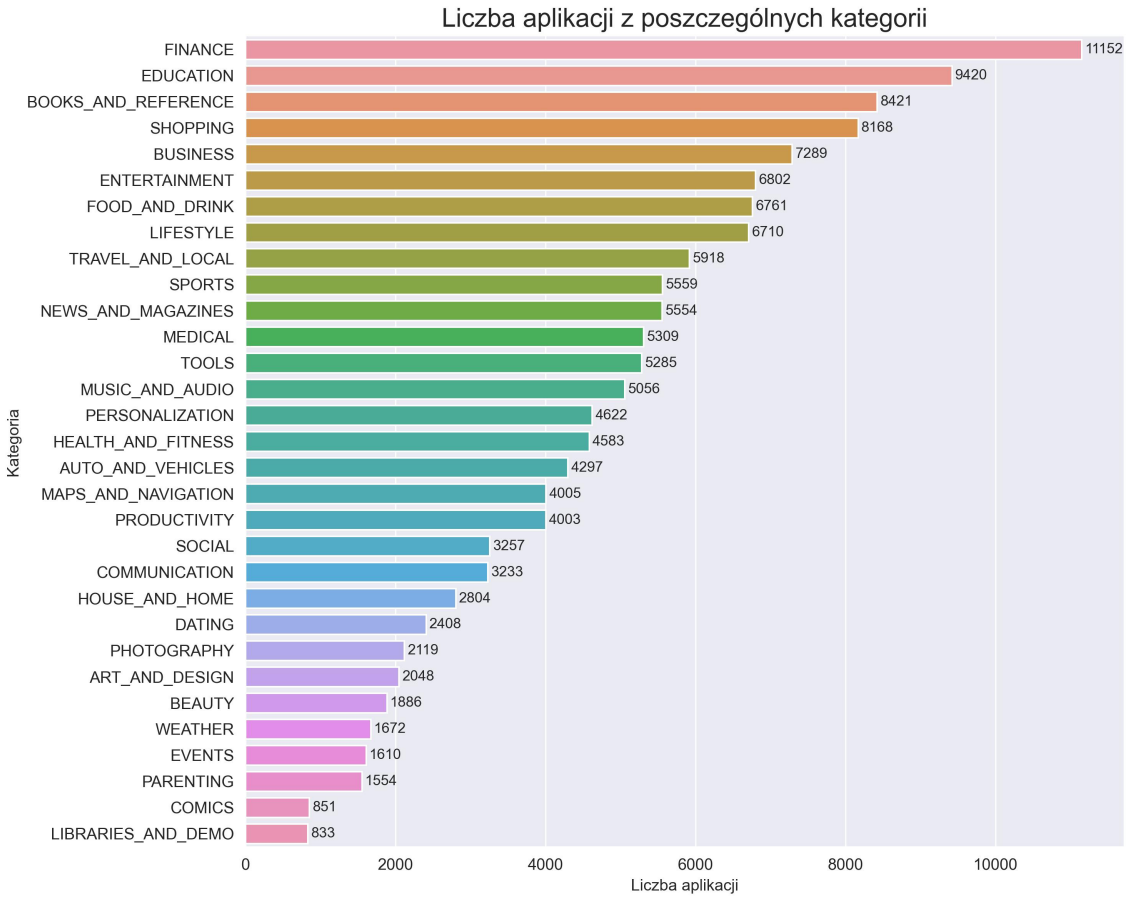
Wykresy przedstawiają średnią ilość otrzymanych ocen oraz średnią ilość otrzymanych recenzji dla aplikacji, których ocena zawiera się w zadanym przedziale.
Wykresy są bardzo podobne, tak więc można stwierdzić, że średnia ilość otrzymanych recenzji jest wprost proporcjonalna do średniej ilości otrzymanych ocen.

4.2.4. Zestawienie darmowych i płatnych aplikacji



Wykres przedstawia ilość płatnych i darmowych aplikacji, które zawierają się w zbiorze danych.
Aplikacji bezpłatnych jest o wiele więcej niż płatnych, jest to naturalne, gdyż wchodząc na platformę google play statystycznie widzi się o wiele więcej bezpłatnych aplikacji, niż płatnych.

4.2.5. Analiza przynależności aplikacji do kategorii



Wykres przedstawia liczbę aplikacji z kategorii, z wykresu wynika, że najwięcej aplikacji pochodzi z kategorii ”FINANCE” a najmniej z ”LIBRARIES_AND_DEMO”

5. Przygotowanie modelu

5.1. Podział danych na dane wejściowe i wyjściowe

Dane, które zostały załadowane z pliku csv zawierają wszystkie parametry, włącznie z danymi wyjściowymi, pierwszym krokiem będzie stworzenie listy wektorów cech oraz odpowiadającej listy danych wyjściowych 0, 1, które odpowiadają temu, czy aplikacja jest darmowa, czy nie.

5.2. Podział danych na treningowe i testowe, skalowanie

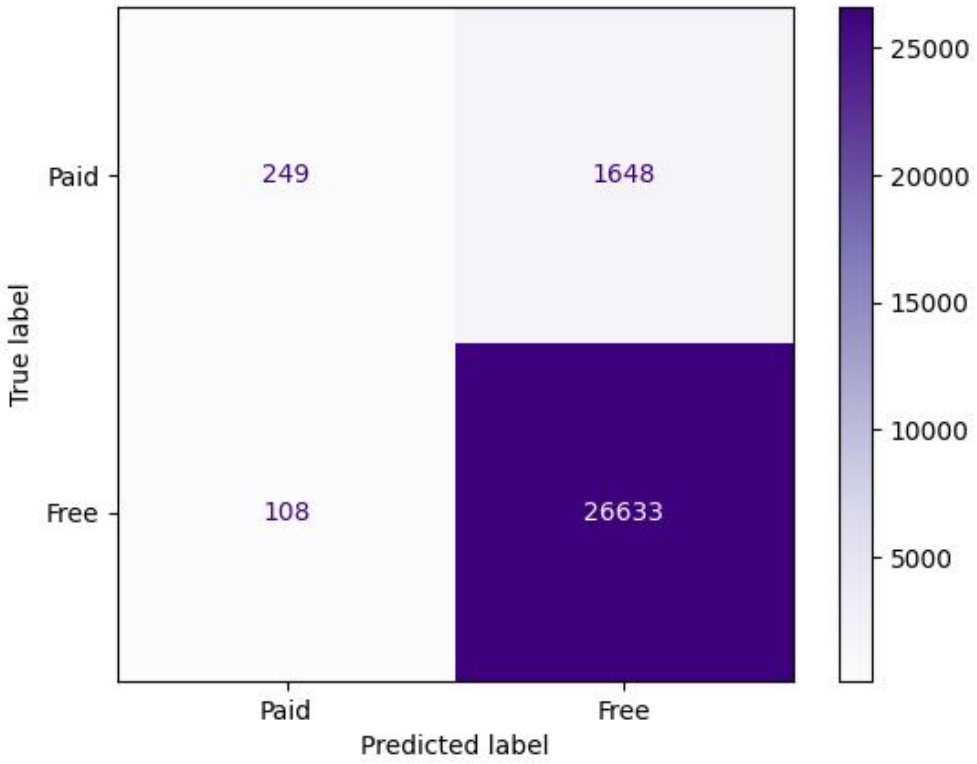
Do podziału danych używam funkcji test_train_split z bilbioteki sklearn. Zdecydowałem się na podział danych w stosunku 80:20.
Parametry zbiorów po podziale, zestawione z oryginalnym zbiorem przedstawiam w tabelce.

Typ zbioru	Wielkość	Liczba płatnych aplikacji	Liczba darmowych aplikacji
Treningowy	114551	7597 (6.632 %)	106954 (93.368 %)
Testowy	28638	1897 (6.6241 %)	26741 (93.3759 %)
Oryginalny	143189	9494 (6.6304 %)	133695 (93.3696 %)

Jak widać, podział zbioru został wykonany dobrze, tj. część procentowa aplikacji płatnych i darmowych jest bardzo podobna do oryginalnego zbioru.
Dane wejściowe muszę przeskalować, gdyż wektory cech zawierają takie parametry, jak liczba pobrań, których wartości zawierają się w zbiorze $[0; 10^{10}]$ oraz liczbę, która odpowiada kategorii, która zawiera się w zbiorze $(0; 1)$, tak więc przeskalowanie danych jest niezbędne.
Do skalowania rozważyłem dwie metody, **normalizację** lub **min-max**.
Zdecydowałem się na normalizację, gdyż w skalowaniu min-max wartości odstające mogą mieć znaczny wpływ na wynik, gdzie normalizacja jest na to mniej wrażliwa.

5.3. Model i pierwsze predykcje

Jako pierwszą instancję modelu utworzyłem model SVC, który używa jądra radialnego (rbf). Podjąłem taką decyzję, gdyż uważam, na podstawie wykresów, że oprócz zależności średniej ilości otrzymanych ocen i średniej ilości otrzymanych recenzji w zależności od oceny, nie ma więcej zależności, które by przypominały liniowe.
W ramach pierwszych predykcji, zdecydowałem się na domyślne parametry C i gamma, tj. C=1.0 i gamma="scale".
Czas, w którym model dokonuje predykcji jest dość długi, około 4 lub 5 minut, jest to spowodowane tym, że złożoność obliczeniowa SVM w zależności od ilości danych jest kwadratowa.
Jest to główny problem modelu, który ogranicza możliwości optymalizacji, które poruszę później.
Wynik pierwszych predykcji przedstawię za pomocą macierzy pomyłek.



5.4. Analiza predykcji i dobranie metryki jakości

Model dobrze przewiduje aplikacje darmowe, niemniej jednak może to być mylące, bo **aplikacji darmowych jest ponad 10 razy więcej niż płatnych**.
Tak więc przy ocenie jakości modelu trzeba się skupić bardziej na pomyłkach związanych z predykcją aplikacji płatnych, mając to na uwadzę, zdefiniuję znaczenie parametrów używanych przy obliczaniu jakości.

- **Klasa pozytywna** - aplikacja płatna (0)
- **Klasa negatywna** - aplikacja darmowa (1)
- **TP** (True positive) - jeśli model poprawnie przewidzi, że aplikacja jest płatna
- **TN** (True negative) - jeśli model poprawnie przewidzi, że aplikacja jest darmowa
- **FP** (False positive) - jeśli model przewidzi, że aplikacja jest płatna, a jest darmowa
- **FN** (False negative) - jeśli model przewidzi, że aplikacja jest darmowa, a jest płatna

Analizując znaczenie parametrów, stwierdzam, że chcę skorzystać z takiej metryki, względem której optymalizując model, **zminimalizuję FP oraz FN**.
Metryka, która mi w tym pomoże, to **F1 score**.
Poniżej zestawię parametry oraz ocenę działania modelu.

TP	**TN**	**FP**	**FN**	**F1 score**
249	26633	108	1648	**0.221**

F1 score równy 0.221 to tragiczny wynik.
Można spróbować polepszyć ten wynik, próbując używać różnych wartości parametrów C i gamma.
Można do tego użyć metod optymalizacji.

5.5. Optymalizacja modelu

Jako że nie mam jawnego wzoru funkcji jakości od parametrów C i gamma, to nie mam jak zastosować tu metod gradientowych.
Jestem zmuszony wykorzystać metody eksploracyjne, w celu poszukiwania parametrów, dla których jakość będzie jak największa.
Użyję do tego metody Grid Search, która będzie przeszukiwała siatkę parametrów w celu znalezienia jak najlepszego zestawu rozwiązań.
Siatka poszukiwań składa się z 6 różnych parametrów gamma oraz 200 różnych wartości parametru C.
Jeśli chodzi o wartości C, to są to wartości z przedziału $(10^{-2}; 10^2)$.
Ponadto wzór na kolejną wartość C opisują wyrażenia:

$$C_i = 10^{k_i}$$

$$C_{i+1} = 10^{k_i+0.02}$$

$$k_i \in [-2; 2]$$

$$k_1 = -2$$

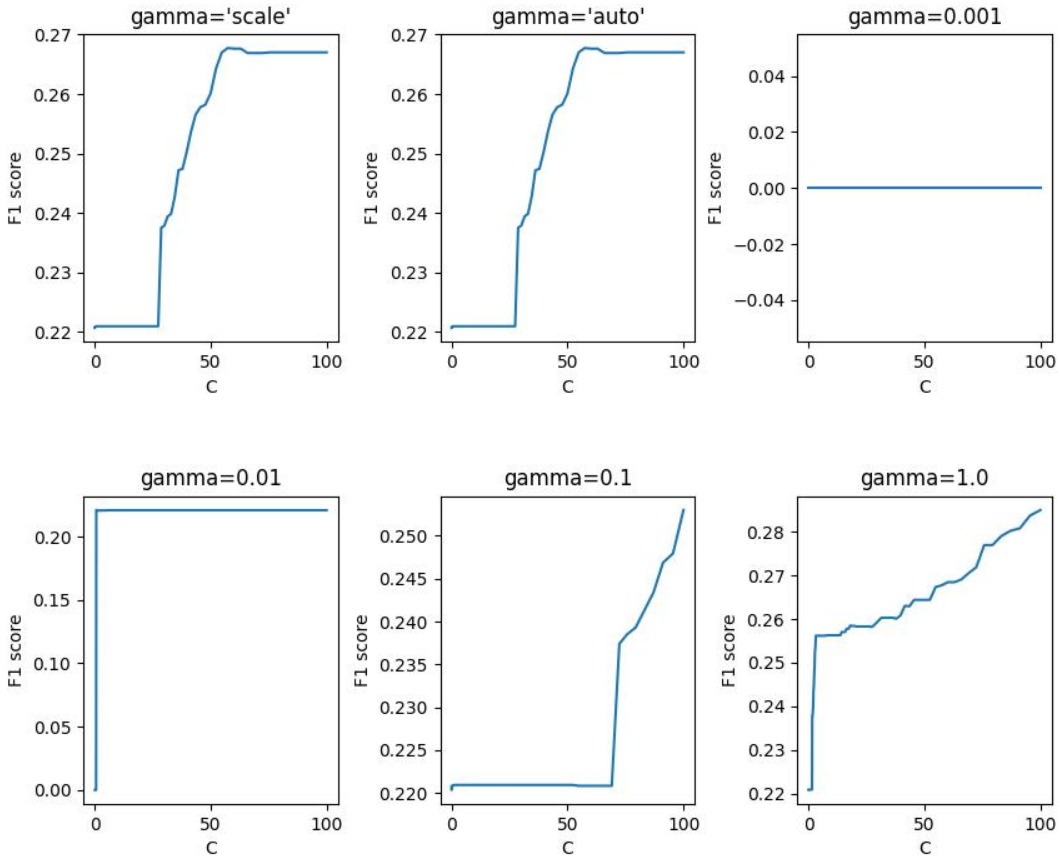
$$1 \leqslant i \leqslant 200$$

Natomiast zbiór wartości parametrów gamma to:

$$\gamma \in \{\text{"auto"}, \text{"scale"}, 0.001, 0.01, 0.1, 1.0\}$$

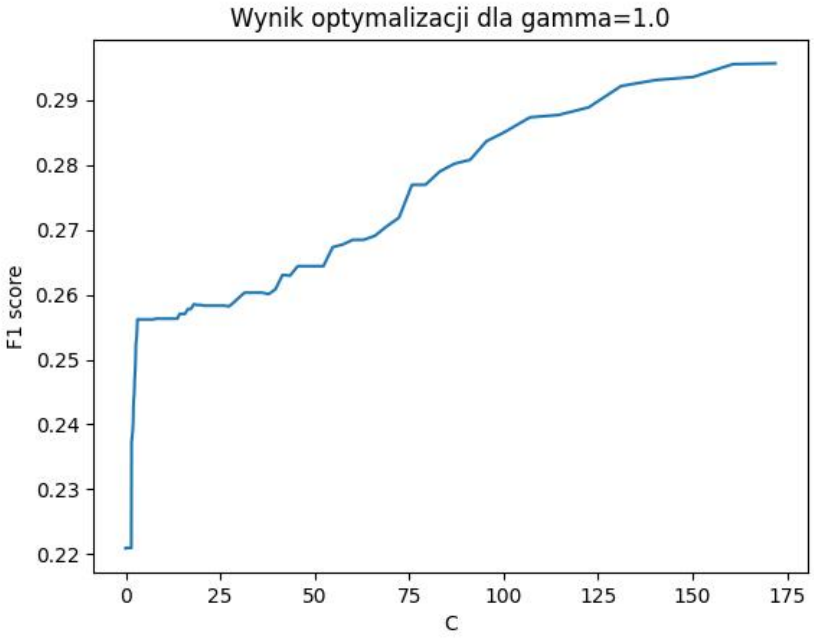
Uruchamiam optymalizator na 24 godziny (tak długo przy tak niewielkiej siatce, gdyż model działa wolno dla tak dużej liczby danych), program zapisuje wyniki w plikach tekstowych, po pracy optymalizatora przeszukam pliki w celu znalezienia parametrów, dla których f1 score jest największy.

Po skończeniu pracy optymalizatora otrzymałem dane, które opisują wpływ parametru C przy ustalonych gamma, obrazuje to wykres na następnej stronie.



Z wykresu widać, że najbardziej obiecujące parametry gamma do dalszej optymalizacji, to gamma=0.1 oraz gamma=1.0.
Jako że optymalizacja trwa bardzo długo, to wybieram jedną wartość gamma, tj. gamma=1.0, ponieważ dla tego parametru jakość osiągnęła największą jakość.

Dokonałem pomiarów jakości dla 8 kolejnych wartości parametru C, gdyż dla tak dużych wartości C, czas, w którym dokonywane są predykcje potrafi przekroczyć 20 minut.
Wykres przedstawiający wyniki ostatecznej optymalizacji jakości modelu znajduje się na następnej stronie.



Największa jakość modelu, dla której predykcje dokonywane są w sensownym czasie, to F1=0.2921997523730912, dla gamma=1.0 oraz C=131.11339374215643.
Dla tych parametrów model dokonuje predykcji w około 10 minut.
Poniżej przedstawiam zestawienie parametrów dokonanych predykcji modelu dla tak dobranych parametrówm, w porównaniu do początkowego modelu. Jak widać, mo-

Model	**TP**	**TN**	**FP**	**FN**	**F1 score**
Początkowy	249	26633	108	1648	0.221
Finałowy	354	26569	172	1543	0.292

del poprawił swoje zdolności do prawidłowego klasyfikowania aplikacji płatnych oraz zmniejszył ilość pomyłek aplikacji darmowej z płatną.
Wzrosła natomiast liczba FP, oznacza to, że model częściej rozpatruje aplikację, jako potencjalnie płatną.
Jakość modelu **wzrosła o 32.25%**, kosztem **30 godzin optymalizacji**.
Jakość ostatecznego modelu nadal nie jest dobra. Aby lepiej zoptymalizować model, potrzebna jest siatka parametrów o wymiarze większym niż 2, ale wtedy optymalizacja mogła by zająć miesiące.

6. Wnioski, komentarze

- Uważam, że dla 140000 danych, gdzie każda dana ma 5 cech, **SVM nie jest najlepszym algorytmem**, szczególnie, gdy między cechami występują skomplikowane zależności.
- Dla tak dużego zbioru danych **predykcje dokonywane przez model trwają zbyt długo**, tj. minimalnie 4 lub 5 minut.
- Optymalizacja takiego modelu jest ciężka, gdyż należy użyć do tego **przeszukiwania eksploracyjnego**, natomiast każda iteracja zajmuje bardzo dużo czasu. Ponadtdo czas klasyfikacji jest zależny od parametrów i może wynieść nawet 30 minut.
- Model by był o wiele **bardziej wydajny dla mniejszego zbioru danych**.
- Algorytmy oparte na sieciach neuronowych, zwłaszcza głębokie sieci neuronowe, mają potencjał do modelowania nieliniowych zależności między cechami i mogą osiągać dobre wyniki dla dużych zbiorów danych. Przy 140 000 danych i 6 cechach istnieje możliwość, że sieci neuronowe mogą efektywnie wykryć i wykorzystać te nieliniowe zależności w celu dokładniejszej klasyfikacji.
- **Zrównoleglenie procesu klasyfikacji** mogłoby pomóc w rozwiązaniu problemu z wydajnością.