POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

Zastosowanie informatyki w gospodarce

Temat: Detektor "Fake News"

Prowadzący:
Dr inż. Tomasz Walkowiak

Termin: Poniedziałek 15:15
Grupa B:
Tymoteusz Frankiewicz, 241255
Filip Gajewski, 236597
Jan Wawruszczak, 241344
Mariusz Wiśniewski, 241393
Szymon Wiśniewski, 241269

Spis treści

1 Wstęp i założenia

Celem naszego projektu jest opracowanie oraz zaimplementowanie rozwiązania oferującego klasyfikację wypowiedzi jako *fake news* lub wypowiedź prawdziwą.

Rozwiązanie zostanie ograniczone do klasyfikacji wypowiedzi polityków w języku polskim, tym samym zadanie będzie oryginalniejsze oraz w pewnym stopniu trudniejsze niż gdyby grupa postanowiła ograniczyć projekt do analizy tekstu w języku angielskim. Wynika to z faktu, że podobne rozwiązania już powstały. Jeśli chodzi o język polski, to już sama obróbka danych może sprawiać problemy. Jest to spowodowane brakiem dostępności do zaawansowanych narzędzi przetwarzania danych tekstowych takich jak stemmer, lematyzater, czy gotowe modele badające nacechowanie emocjonalne wypowiedzi. Pomoc w powyższych kwestiach poprzez zapewnienie odpowiednich narzędzi zaoferował prowadzący, dr inż. Tomasz Walkowiak. Projekt przewiduje stworzenie rozwiązania w formie aplikacji webowej oraz udostępnienie go publicznie na serwerze internetowym.

Baza danych wykorzystana do wytrenowania modelu została zbudowana z danych dostępnych w serwisie *demagog.org.pl*[11]. Strona ta została wybrana ze względu na jej bardzo szablonową strukturę, która ułatwi automatyczną ekstrakcję pożądanych danych. W tym celu grupa ma zamiar stworzyć program typu *crawler*, który automatycznie przejrzy wymienioną powyżej stronę oraz zapisze w formacie `.csv` pobrane dane.

1.1 Wymagania funkcjonalne

1. Możliwość wprowadzenia wypowiedzi polityka - w zadanym polu wklejana w formie tekstu.
2. Możliwość zatwierdzenia wprowadzanej wypowiedzi - zatwierdzenie poprzez element interfejsu użytkownika.
3. Możliwość automatycznego przekierowania sprawdzanej wypowiedzi do ręcznej weryfikacji w przypadku niepewności systemu.
4. Możliwość zgłoszenia przez użytkownika wątpliwości odnośnie otrzymanego rezultatu.
5. Możliwość logowania przez administratora, równocześnie pełniącego rolę edytora.
6. Wyświetlenie wyniku klasyfikacji.
7. Możliwość wyświetlenia statystyki pewności z jaką dany tekst został zaklasyfikowany jako prawdziwy lub *fake news*.

1.2 Wymagania niefunkcjonalne

1. Aplikacja zawierać będzie interfejs webowy.
2. Aplikacja będzie skonteneryzowana przy pomocy narzędzia *Docker*.
3. Część kliencka będzie wykonana w formie *Single Page Application*.

4. Część kliencka aplikacji zakodowana na podstawie autorskiego szablonu (ang. *mock*) przygotowanego w narzędziu *Figma*.
5. Dostęp do części edytorskiej po ówczesnym uwierzytelnieniu loginem oraz hasłem.
6. Autoryzacja będzie miała miejsce wykorzystaniem *JWT*.
7. Hasła w bazie danych zaszyfrowane przy pomocy określonego ukrytego klucza.
8. Komunikacja z częścią serwerową odbywać się będzie przy użyciu protokołu HTTP.
9. Długość możliwego do wprowadzenia tekstu wypowiedzi polityka będzie ograniczona do pewnej ilości znaków.
10. Wszelakie formularze będą sprawdzane, tak aby możliwe było wprowadzenie jedynie istotnych danych.
11. Umożliwiony okresowy trening modelu uzupełnionym zestawem danych.
12. Gotowa aplikacja wdrożona na serwer i upubliczniona.
13. Relatywnie krótki czas analizy tekstu przez model.

2 Pierwszy kamień milowy - przegląd literatury oraz źródeł danych

2.1 Przegląd algorytmów klasyfikacyjnych

Klasyfikacja tekstu jest bardzo istotnym i znaczącym zadaniem w przetwarzaniu języka naturalnego. Proces ten można podzielić na kilka zasadniczych etapów: wstępna obróbka tekstu źródłowego, klasyfikacja oraz ewaluacja. Wstępna obróbka tekstu oraz ewaluacja w zdecydowanej większości przypadków korzystają ze wspólnego wachlarza technik. Klasyfikację natomiast możemy podzielić na dwa zasadnicze podejścia. Na dokonywaną przy pomocy technik *shallow learning'u* oraz przy pomocy *deep learning'u*.

W shallow learning'u musi zostać najpierw dokonana ekstrakcja cech z tekstu. Dodatkowo należy stworzyć wektory odpowiadające danemu wejściu. Możemy do tego wykorzystać metody *Bag of words* lub *Term Frequency - Inverse Document Frequency*. Następnie poddany wektoryzacji tekst jest przepuszczany przez klasyfikator. Klasyfikatory w zależności od tego czy tworzą one model dzielimy na gorliwe, np. drzewa decyzyjne, naiwny klasyfikator bayesowski, lub leniwe np. k-najbliższych sąsiadów. Główną różnicą między takimi klasyfikatorami jest czas ich nauki oraz ewaluacji.

Deep learning nie wymaga od programisty wcześniejszej ekstrakcji cech z tekstu, sieci neuronowe robią to automatycznie dzięki czemu oszczędzamy wiele czasu na projektowaniu oraz ekstrakcji cech z tekstu. Powszechnie uważa się, że deep learning osiąga lepsze wyniki niż shallow learning. Niestety do poprawnego działania wymaga on znacznie większego zbioru danych. Najpopularniejszymi modelami sztucznych sieci neuronowych tutaj są perceptrony wielowarstwowe, sieci konwolucyjne oraz sieci rekurencyjne [8].

2.2 Artykuł nr 1 - Fake News Detection from Data Streams

Pierwszym z artykułów, który przeanalizowano w celu zdobycia gruntownej wiedzy na temat istniejących już rozwiązań jest *"Fake News Detection from Data Streams"*[7] opracowany przez Pawła Ksieniewicza oraz Pawła Zyblewskiego, pracowników Politechniki Wrocławskiej.

Autorzy podczas swojej pracy skupili się na wykorzystaniu metod uczenia maszynowego w celu detekcji fałszywych informacji. W przeciwieństwie do większości stosowanych obecnie podejść, wiadomości źródłowe traktowano jako dane strumieniowe, biorąc tym samym pod uwagę możliwość występowania zmiany definicji pojęć (ang. *concept drift*). Zauważają oni bowiem, iż cechy charakterystyczne informacji zaklasyfikowanych jako fałszywe mogą z czasem ulegać zmianie. Jest to spowodowane tym, że wraz z rozwojem prac dążących do rozwiązania tego problemu autorzy celowo publikujący fałszywe informacje dostrzegają, że staje się to coraz trudniejsze. Z tego względu należy spodziewać się, że cechy charakterystyczne tego typu wiadomości ulegną z czasem zmianie. Osoby zaangażowane w ich tworzenie zaczną bowiem poszukiwać rozwiązań, których celem będzie zmylenie istniejących systemów.

Na wstępie zdecydowano się zastosować istniejące już rozwiązanie, *Count Vectorizer*, w celu liczbowej reprezentacji każdego z artykułów wejściowych. Liczba atrybutów reprezentujących indywidualny tekst wyniosła 1000, a dodatkowe informacje brane pod uwagę podczas analizy to tytuł oraz data publikacji. Następnie zbiór danych uszeregowano chronologicznie

w celu symulacji zjawiska zmiany definicji pojęć.

2.2.1 Redukcja wymiarów

Jako iż liczba atrybutów zbioru uczącego wyniosła 1000 i wiele z nich okazywało wzajemne zależności statystyczne dokonano procesu ich redukcji przy wykorzystaniu istniejących metod ekstrakcji i doboru cech. Były to:

- *PCA* (ang. *Principal Components Analysis*) - analiza głównych składniowych polegająca na obrocie układu współrzędnych w celu maksymalizacji wartości kolejnych wariancji. Powstała w ten sposób nowa przestrzeń obserwacji charakteryzuje się tym, że jej początkowe czynniki wyjaśniają najwięcej zmienności.
- *Count Vectorizer* - metoda generacji cech liczbowych z tekstu polegająca na tworzeniu wektorów, których rozmiar jest równy liczbie analizowanych słów. Wartość określonej komórki wektora odpowiada liczbie wystąpień przypisanego wyrazu w analizowanym tekście.
- Selekcja cech w oparciu o test zgodności chi-kwadrat - wykorzystana w celu statystycznej oceny zależności pomiędzy cechami, której wyniki zostały wykorzystane do selekcji tych najbardziej znaczących.

2.2.2 Strategie trenowania klasyfikatora na strumieniu danych

Każda z wykorzystanych metod redukcji wymiarów została przeanalizowana za pomocą poniższych metod konstrukcji modeli klasyfikujących:

- *SEA* (ang. *Streaming Ensemble Algorithm*) - tworzy zestaw klasyfikatorów o stałym rozmiarze poprzez trenowanie klasyfikatora bazowego na każdym z obserwowanych kawałków danych. W przypadku przekroczenia wyznaczonego rozmiaru usuwany jest model generujący najgorsze wyniki.
- *OB* (ang. *Online bagging*) - algorytm zaliczany do metod grupowania (ang. *ensemble learning*). Utrzymuje on zestaw klasyfikatorów, w którym z momentem nadejścia nowych danych każdy z podstawowych estymatorów trenowany jest K razy, gdzie K pochodzi z dystrybucji poissona - $Poisson(\lambda = 1)$.
- *SM* (ang. *Single model*) - pojedynczy model. Podejście to nie jest odporne na występowanie zmiany definicji pojęć. W przeciwieństwie do metod grupowania nie jest on silnie zależny od rozmiaru kawałka danych wykorzystywanego w procesowaniu.

2.2.3 Podstawowe klasyfikatory do przetwarzania strumienia danych

- *GNB* (ang. *Gaussian Naive Bayes*) - naiwny klasyfikator bayesowski wykorzystujący rozkład normalny.

- *MLP* (ang. *Multi-layer Perceptron*) - perceptron wielowarstwowy z jedną warstwą ukrytą zbudowaną na 100 neuronach. Jako funkcję aktywacji wykorzystano *ReLU* (ang. *rectified linear unit*). Ponadto zdecydowano się na użycie stochastycznego optymalizatora opartego na metodzie gradientu.

- *HT* (ang. *Hoeffding Tree*) - oparty na kryterium podziału wykorzystującym współczynnik Giniego. Jako mechanizm predykcji zdecydowano się zastosować adaptacyjny naiwny klasyfikator bayesowski.

2.2.4 Analiza wyników

Eksperymenty przeprowadzone przy wykorzystaniu bibliotek dostępnych w języku *Python* pozwoliły na stwierdzenie, iż najbardziej efektywnym algorytmem klasyfikacji jest model perceptronu wielowarstwowego *MLP*. Pozwolił on na uzyskanie najlepszych wyników w kombinacji ze strategią trenowania klasyfikatora *OB* oraz metodą redukcji cech *PCA* .

2.3 Artykuł nr 2 – Detecting fake news with and without code

Artykuł autorstwa Favio Vázqueza stwierdza, że problem detekcji "fake newsów" jest bardzo rozmyty – często nie da się po prostu stwierdzić na podstawie analizy faktów, czy coś jest prawdą lub fałszem. Wynika to z tego, że same dodane do sieci wypowiedzi w formie np. tweet'ów nie zawierają weryfikowalnych informacji. Sprawia to, że metody oparte o przeszukiwanie zaufanych źródeł mogą być po prostu bardzo zawodne. Zauważono jednak fakt, że programowane boty wykorzystują specyficzne wzorce (skojarzone z socjotechnikami manipulacji), które pozwalają wykryć informacje fałszywe na poziomie składni.

2.3.1 Zastosowane podejścia i otrzymane wyniki

W ramach pracy podany został przykład kodu analizującego m. in. składnię tekstów oznaczonych jako fałszywe i prawdziwe w internetowym zbiorze danych portalu *Kaggle*. Podczas swoich prac autor wpisu przeprowadził kilka prostych eksperymentów badających, które słowa występują najliczniej w tekstach oznaczonych jako fałszywe – okazuje się, że większość z nich odnosi się do wyrażania opinii. W formie bezosobowej może to być "uważać", "mówić", "twierdzić". Poza tym dużo z nich odwołuje się do autorytetów ogólnych (w celu ich podważenia), a nie zdań ekspertów, np. Donalda Trumpa, Hilary Clinton, Baracka Obamy (wynika to z tego, że baza danych dotyczyła głównie tekstów ze Stanów Zjednoczonych Ameryki).

Zbadanych zostało kilka znanych technik klasyfikacji.

1. Regresja logiczna uzyskała dokładność wynoszącą 98.76%

2. Drzewo decyzyjne uzyskało dokładność wynoszącą 99.71%

3. Las losowy uzyskał dokładność wynoszącą 98.98%

2.3.2 Wnioski z pracy

Jak widać uzyskane wyniki sugerują, że analiza tekstu oraz wykorzystanie klasycznych klasyfikatorów gorliwych w połączeniu ze sobą pozwalają wytworzyć modele dające satysfakcjonujące rezultaty. Oczywiście, należy wziąć pod uwagę relatywnie prostszą składnię języka angielskiego względem języka polskiego oraz znacząco większą bazę danych, na której to podstawie można było trenować model. Mimo to wyniki zdają się być obiecujące i mogą zostać wykorzystane podczas planowania podejścia do omawianego problemu.

2.4 Artykuł nr 3 – Kelly Stahl, Fake news detection in social media

Kolejnym materiałem, który znacząco przyczynił się do budowania fundamentów wiedzy koniecznej do realizacji części projektu związanej z uczeniem maszynowym, jest artykuł autorstwa Kelly'ego Stahla, z 2018 roku, traktujący o detekcji *fake newsów* w mediach społecznościowych [18].

Czytelnik wprowadzany jest w temat, począwszy od określenia, czym właściwie są *fake newsy* oraz dlaczego są one realnym problemem. Następnie przedstawiono konkretne metody wykrywania omawianych wiadomości w formie tekstowej. Mimo że problem przedstawiony jest na kanwie mediów społecznościowych, wierzymy iż uzyskana wiedza jest uniwersalna i może być wykorzystana również dla innych typów źródeł danych.

Na przestrzeni artykułu przedstawione są dwa główne podejścia do wykrywania *fake newsów* – *Linguistic Cue* oraz *Network Analysis*. Zaproponowana została trzyczęściowa metoda, która wykorzystuje naiwny klasyfikator bayesowski (ang. *Naive Bayes Classifier*), maszynę wektorów nośnych (ang. *Support Vector Machines*) oraz analizę semantyczną (ang. *Semantic Analysis*).

W podejściach typu *Linguistic Cue* następuje wykrywanie próby oszustwa przy pomocy analizy różnych zachowań komunikacyjnych. Mechanizm detekcji polega na weryfikacji stylu konkretnej wypowiedzi. Wyszukiwane są przesłanki lingwistyczne, które mogą świadczyć o oszustwie. Autor informuje, że oszuści mają tendencję do używania większej liczby słów niż osoba, która przekazuje prawdziwą wiadomość. Mało kiedy używają oni zaimków, które dotyczą bezpośrednio ich, lecz znacznie częściej wykorzystują zaimki, które określają innych. Dodatkowo, w informacjach nieprawdziwych, często zawarte są słowa, które intensywniej oddziaływują na zmysły odbiorcy.

Wśród podejść typu *Linguistic Cue* wyszczególniono cztery różne metody. Są to:

- Data Representation – każde słowo analizowane jest osobno, aby finalnie móc wyekstrahować wskazówki mogące świadczyć o oszustwie.
- Deep Syntax – wykorzystywane jest *Probability Context Free Grammars* (*PCFG*), przy pomocy którego zdania zostają przekonwertowane w zbiór przepisanych zasad celem opisania struktury składniowej analizowanej wiadomości.
- Semantic Analysis – następuje założenie, że oszust często nie ma pełnej wiedzy na dany temat, więc wiadomość, którą przekazuje może zawierać sprzeczności, bądź nie zawierać istotnych faktów, które występują w innych źródłach dotyczących powiązanego tematu.

- Sentiment Analysis – badane są emocje zawarte w przekazywanej treści, sprawdzane jest uczucie, które może towarzyszyć czytelnikowi podczas konsumpcji konkretnej wiadomości.

Drugie podejście, *Network Analysis*, w przeciwieństwie do podejść typu *Linguistic Cue*, bazuje bezpośrednio na prawdziwości treści konkretnej wiadomości, a nie na przesłankach językowych. Wymaga dostępu do źródeł kumulujących określony typ wiedzy. Następuje tutaj tzw. sprawdzanie faktów (ang. *fact-checking*), czyli prawdziwość wypowiedzi definiowana jest na podstawie obiektywnej wiedzy w konkretnym kontekście. Wyróżnia się trzy różne systemy pozyskiwania wiedzy:

- Zorientowane na wiedzę ekspercką (ang. *expert-oriented*) – wiadomości przetwarzane są przez grupę ekspertów, którzy samodzielnie je weryfikują. Tworzenie bazy jest wymagające i czasochłonne.

- Zorientowane na wiedzę społeczną (ang. *crowdsourcing-oriented*) – wykorzystywana jest wiedza społeczeństwa, czyli zwykłych ludzi, którzy dyskutują i kolektywnie analizują treść określonej wiadomości.

- Zorientowane na obliczenia (ang. *computational-oriented*) – klasyfikacja prawdziwości określonych twierdzeń jest automatyzowana przy pomocy dostępu do zewnętrznych źródeł, np. otwartych sieci, bądź grafów wiedzy.

Na podejście typu *Network Analysis* składają się dwie główne metody:

- *Linked Data* – następuje porównanie treści wiadomości ze znanymi faktami.

- *Social Network behavior* – przeprowadzana jest analiza, która ma na celu przedstawienie zawartość dużych zbiorów tekstów poprzez zdefiniowanie najważniejszych słów, które łączą pozostałe słowa w obrębie sieci.

Autor zwraca uwagę, że oprogramowanie do wykrywania fałszywych wiadomości zazwyczaj wykorzystuje parę różnych metod podejścia lingwistycznej przesłanki, gdyż wykorzystanie jednej metody nie jest wystarczająco skuteczne. Oprócz przesłanek lingwistycznych należy również zwracać uwagę na inne aspekty, między innymi na wiarygodność źródła wiadomości, ekspertyzę, czy też jakość cytatów.

Jako metody do dalszej eksploracji zagadnienia detekcji *fake newsów* zaproponowano następujące metody:

- Naiwny klasyfikator bayesowski (*Naive Bayes Classifier*) – bazuje na teorii prawdopodobieństwa warunkowego, czyli określaniu prawdopodobieństwa danego wyniku na podstawie poprzednich zdarzeń. Nie występuje tu żadna relacja między cechami. Mimo to, klasyfikator w aspekcie przetwarzania tekstu wciąż charakteryzuje się dużą wydajnością, nawet jeśli faktyczne zależności są silne. Niestety zdarzają się przypadki, w których brakujące zależności stanowią realny problem.

- Maszyna wektorów nośnych (*SVM*) – celem jest podział zbioru na dwie grupy, tak aby margines między klasami był jak największy. Bardzo dobrze sprawdza się na małych zbiorach danych i jest elastyczna. Do wad należą natomaist: duży czas przetwarzania dużych zbiorów oraz mała efektywność na zbiorach, w których dystrybucje klas się ze sobą przeplatają.

- Semantyczna analiza (*Semantic Analysis*) – pochodna przetwarzania języka naturalnego (*NLP*), która polega na znajdywaniu połączeń między częściami tekstu. Szczególnie dobrze sprawdza się w językach, w których występują słowa z wieloma znaczeniami, gdyż potrafi je rozróżnić. Działaniem, podczas przetwarzania tekstu, przypomina funkcjonowanie ludzkiego mózgu.

Autor uważa, że kombinacja powyżej opisanych metod może przynieść pozytywny skutek w aspekcie wykrywania *fake newsów*. Zwraca uwagę, że w innych zastosowaniach połączenie *naiwnego klasyfikatora bayesowskiego* i *SVM* doprowadziło do uzyskania rezultatów lepszych niż wyniki każdego z tych algorytmów z osobna. Uzupełnienie ich metodą analizy semantycznej pozwala na uwzględnienie relacji między elementami tekstu, co równoważy słabości *naiwnego klasyfikatora bayesowskiego*.

2.5 Podsumowanie zebranej wiedzy

Zaprezentowane powyżej artykuły prezentowały bardzo nowatorskie oraz solidne pod względem jakości produkowanych wyników rozwiązania. Pierwszy opisany artykuł oferował adaptację do płynnego pojęcia jakim jest *fake news*. W naszym projekcie podejdziemy bardziej naiwnie do tego zagadnienia i założymy, że politycy w naszych wypowiedziach nie starają się podnosić swoich kompetencji w zakresie tworzenia *fake news'ów*, a ich klasyfikacja będzie dokonywana na podstawie ładunku emocjonalnego zawartego w danej wypowiedzi. W artykule nr 2 zauważono, że informacjami, które mogą świadczyć o tym czy coś jest albo nie jest fake news'em jest brak interpunkcji, wielokrotne powtarzanie konkretnych słów w formie bezosobowej oraz powoływanie się na konkretne autorytety. Nasza baza danych składa się ze stosunkowo krótkich wypowiedzi polityków, poprawnych pod względem interpunkcyjnym oraz gramatycznym, więc wnioski te nie przełożą się na naszą bazę danych. Jednakowoż planujemy sporządzić porównanie graficzne istotności niektórych cech, które często są wskazywane jako świadczące o fake news'ie. Tego typu charakterystyką może być między innymi długość tekstu.

W trzecim artykule został poruszony temat weryfikacji prawdziwości informacji. Oprócz weryfikacji automatycznej - przy pomocy odpowiedniego oprogramowania, możliwe jest również sprawdzenie ich przez grupę ekspercką. Nasza aplikacja planowo też ma zawierać podobną usługę. Interfejs webowy będzie informował użytkownika o bardzo małej pewności zaklasyfikowania podanego przez niego tekstu i następnie zgłaszał taką wypowiedź do ręcznej weryfikacji. W tym samym artykule autorzy zwracają uwagę na pozytywne skutki płynące z kombinacji kilku różnych metod klasyfikacji. Obecnie podstawowa wersja naszego rozwiązania nie przewiduje implementacji takiego podejścia. Chcemy jednak się mu przyjrzeć i być może rozważyć implementacje w kolejnych iteracjach naszej aplikacji.

Część potoku projektu odpowiedzialnego za klasyfikację już przetworzonego tekstu wejściowego będzie zrealizowana w kliku różnych podejściach po to, aby na końcu porównać ich wydajność. Planowane jest wykorzystanie jednej architektury głębokiej sieci neuronowej oraz dwóch podejść uczenia płytkiego.

W związku z tym, że nasz projekt będzie opublikowany w formie aplikacji webowej potencjalny użytkownik będzie wymagał bardzo szybkiej odpowiedzi na jego zapytanie odnośnie klasyfikacji podanej wypowiedzi polityka. Dlatego aby skrócić czas oczekiwania klasyfikator będzie zaimplementowany z wykorzystaniem algorytmów gorliwych. Najprawdopodobniej w pierwszych implementacjach projektu posłużymy się algorytmami zaproponowanymi w powyższych artykułach takimi jak las losowy i przeprowadzimy porównanie w celu weryfikacji, które z nich oferują najlepszą wydajność oraz jak prezentują się w porównaniu z uczeniem głębokim na stworzonym zbiorze danych. W implementacji projektu skupimy się na tym, aby ewaluacja modelu osiągnęła jak najwyższy współczynnik czułości (ang. *recall*) uznając, że najważniejsze jest bycie podejrzliwym w stosunku do informacji, które mogą okazać się nieprawdziwe i uznajemy, że bezpieczniejsze jest podejście, w którym część prawdziwych informacji zaklasyfikujemy jako nieprawdę niż uznać *fake news'a* za fakt.

3 Drugi kamień milowy - pozyskanie zbioru danych oraz design części klienckiej

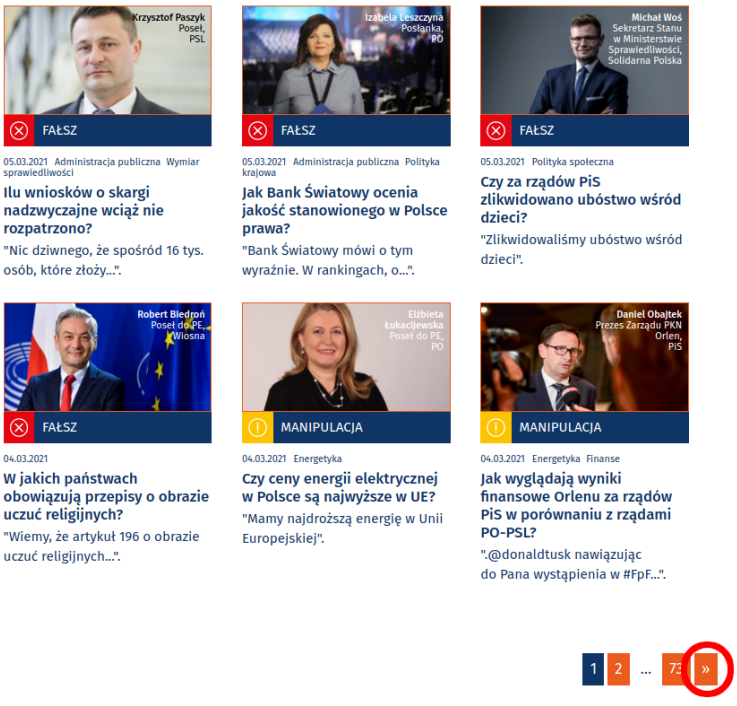
3.1 Projekt i implementacja programu zbierającego dane z sieci

3.1.1 Zarys problemu

Program do pozyskiwania danych z internetu nazywany jest w literaturze fachowej crawlerem, scraperem lub spiderem (pająkiem). W przypadku projektu, zaistniała potrzeba zebrania wypowiedzi polityków. Wypowiedzi te są zbierane i weryfikowane przez portal demagog[11].

Automatyzacja zbierania informacji jest sensowna, o ile źródło ma regularną strukturę. Oznacza to, że dane, które są przeznaczone do zebrania, muszą znajdować się za każdym razem w przewidywalnych miejscach. Jeśli dane są przechowywane na kilku podstronach, ważne jest także, aby do kolejnej podstrony można było dotrzeć w ten sam sposób.

Część portalu *demagog*, prezentująca zweryfikowane wypowiedzi, posiada następującą strukturę; strona główna, widoczna na obrazie 1. Zawiera ona listę najaktualniejszych zweryfikowanych wypowiedzi, a także link do kolejnej strony, zaznaczony na obrazie na czerwonym okręgiem. Znajduje się on zawsze w tym samym miejscu, dzięki czemu, możliwa jest nawigacja do kolejnych stron z listą zweryfikowanych wypowiedzi.



Rysunek 1: Główna strona z wypowiedziami na portalu *demagog*

Kliknięcie na tytuł, bądź obraz ilustrujący artykuł, powoduje przejście do widoku szczegółowego dla danej wypowiedzi. Widok szczegółowy, został przedstawiony na obrazie 2. Tutaj znajduje się cała wypowiedź poddana weryfikacji, autor wypowiedzi, medium, uzasadnienie oceny, oraz ocena. Ocena może przyjmować wartości: "Prawda", "Fałsz", "Manipulacja", Ńieweryfikowalne". Z punktu widzenia projektu, przydatne okazały się informacje o autorze, treść wypowiedzi i ocena.

Rysunek 2: Strona szczegółowa wypowiedzi na portalu *demagog*

Biorąc pod uwagę powyższą analizę, strona jest regularna i zasadnym jest stworzenie rozwiązania, ściągającego dane w sposób automatyczny.

3.1.2 Opis technologii

Przegląd dostępnych technologi, zdawał się wykazać, że najlepszym narzędziem jest *scrapy*[19]. Kluczowymi cechami frameworka są:

- wykonywanie zapytań w sposób asynchroniczny, tj. nie ma konieczności czekania, aż dane zapytanie HTTP się skończy.
- umożliwienie programiście kontroli nad szybkością wykonywania zapytań, w celu oszczędzenia docelowego serwera (np. ustawienie opóźnienia, ustalenie liczby jednocześnie wykonywanych asynchronicznych zapytań HTTP)
- dedykowana powłoka *scrapy*, z poziomu której można wykonywać testowe zapytania i sprawdzać rozwiązania dotyczące ekstracji danych z witryny.

Scrapy dostarcza także przydatne funkcjonalności, jak automatyczne tworzenie zapytań, czy też automatyczny eksport do kilku znanych formatów pliku (csv, json, xsm). Pozwala

to skupić się na rozwiązaniu problemów charakterystycznych dla zadania, nie przejmując się detalami implementacyjnymi zapisywania do pliku, czy obsługi zapytania.

3.1.3 Implementacja rozwiązania

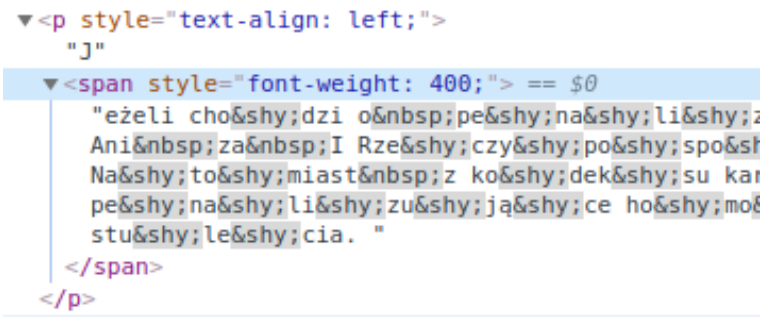
Implementację rozpoczęto od znalezienia na stronie głównej elementu prowadzącego do strony detalicznej oraz następnej strony. Okazała się być to łatwym zadaniem, gdyż odnośniki te zawsze posiadają tę samą klasę i mógły zostać łatwo zlokalizowane. W listingu 1 znajduje się to w linii 14 i 17.

Kolejnym krokiem była ekstracja informacji na stronie detalicznej. O ile, ocenę wypowiedzi i autora można zdobyć, przez podanie odpowiedniego selektora, tak zdobycie całej wypowiedzi okazało się wysoce problematyczne.

Listing 1: Implementacja crawlera

```
1  import re
2  import scrapy
3
4
5  class CitationSpider(scrapy.Spider):
6      name = 'citations'
7      start_urls = [
8          'https://demagog.org.pl/wypowiedzi/'
9      ]
10     tag_pattern = re.compile(r'<[^>]*>')
11     spaces_pattern = re.compile(r' +')
12
13     def parse(self, response):
14         detail_urls = response.css('.title-archive a')
15         yield from response.follow_all(detail_urls, callback=self.parse_detail)
16
17         next_page_url = response.css('.next')
18         yield from response.follow_all(next_page_url, callback=self.parse)
19
20     def parse_detail(self, response):
21         content = response.css('.hyphenate').get()
22         date = response.css('.date-content').get()
23
24         # remove date tags and text (if exists)
25         if date:
26             content = content.replace(date, '')
27
28         # clean all html tags
29         content = self.tag_pattern.sub('', content)
30
31         # get rid of nbsp and \n, convert ampersand code to sign, etc
32         content = content.replace('\xa0', ' ').replace('\n', ' ').replace('&amp;','&').replace(';',
               ' - ').strip()
33
34         # remove unnecessary spaces
35         content = self.spaces_pattern.sub(' ', content)
36
37         yield {
38             'content': content,
39             'author': response.css('.person-name a::text').get(),
40             'label': response.css('.ocena::text').get()
41         }
```

Struktura wypowiedzi okazała się niejednolita. Tekst występował wewnątrz różnych tagów (*span* i *p*). Po zaimplementowaniu takiego rozwiązującego ten problem, część tekstów okazała się posiadać zagnieżdżone tagi różnych typów. Pojawiały się one w mało oczekiwanych miejscach jak np. na rysunku 3. Tutaj zostały w sobie zagnieżdżone tagi typu inline, w dodatku dzieląc między siebie tekst.



Rysunek 3: Przykład nietypowego zastowsowania tagów HTML w tekscie

Podobne sytuacje stanowiły znaczny odsetek materiału, dlatego nie można było ich zignorować. Rozwiązaniem okazało się pobieranie całego tekstu, wycinanie, o ile istnieje, informacji o medium, i za pomocą wyrażenia regularnego, usuwanie wszelkich tagów HTML z jego treści. Wyrażenie regularne pasujące do tagów HTML widoczne jest w linii 10 listingu 1.

Ostatnim krokiem jest przefiltrowanie otrzymanego tekstu z symboli. Znaki te, to np. twarde spacje, czy ampersandy, które w HTML reprezentowane są przez kody. Kody te, należy zamienić na zrozumiały dla człowieka tekst. Zamiana ta odbywa się w linii 32 w listingu 1.

Uruchomienie crawlera i eksport zebranych danych jest wywoływany komendą. W komendzie podawana jest nazwa crawlera, a z podanego rozszeżenia wnioskowany jest docelowy format zapisu danych.

```
scrapy crawl citations -O citations.csv.
```

Proces zbierania danych trwa około 10 minut. Zbiór jest czysty, wolny od jakichkolwiek elementów HTML, czy znaków specjalnych. Obecne rozwiązanie wymagało jedynie manualnej poprawy w 3 rekordach ze względu na niekonsekwencje twórców portalu (wstawienie okna portalu facebook jako wypowiedź).

3.2 Design aplikacji klienckiej

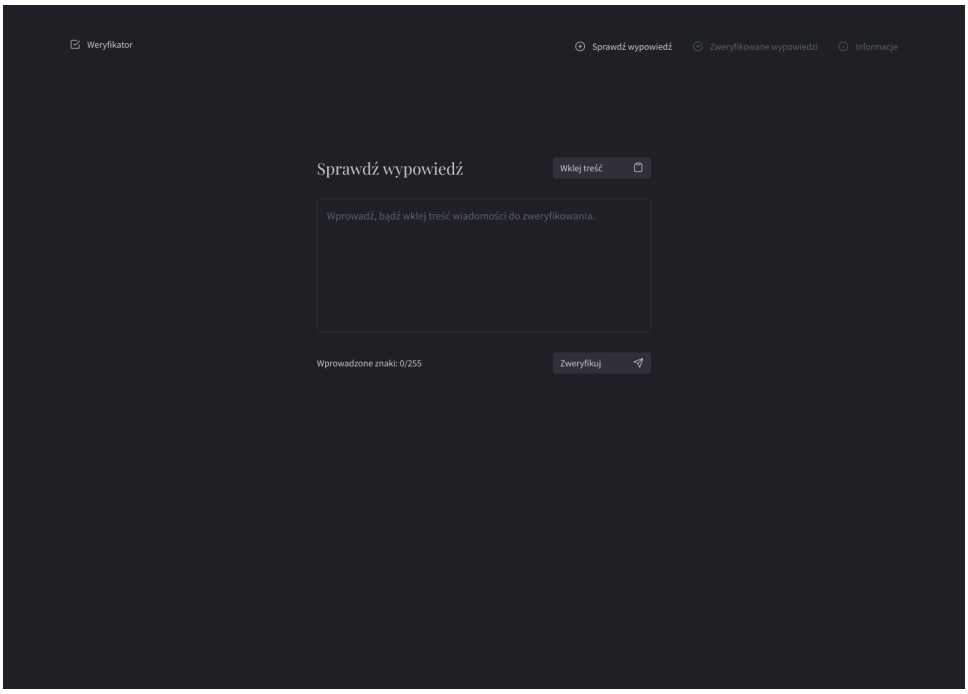
Projekt aplikacji klienckiej wykonany został przy użyciu aplikacji Figma. W ramach aplikacji wyróżniono dwa typy widoków, które charakteryzują się różnym dostępem widoczności. Widoki ogólnodostępne widzialne są dla zwykłego użytkownika Internetu. Dostęp do nich nie jest w żaden sposób limitowany. Obok widoków ogólnodostępnych, istnieją również widoki edytorskie, do których dostęp nie jest jawny. Edytorzy muszą podać poprawne dane dostępowe do swojego konta, aby uwierzytelnianie, a następnie autoryzacja przebiegały poprawnie.

3.2.1 Widoki ogólnodostępne

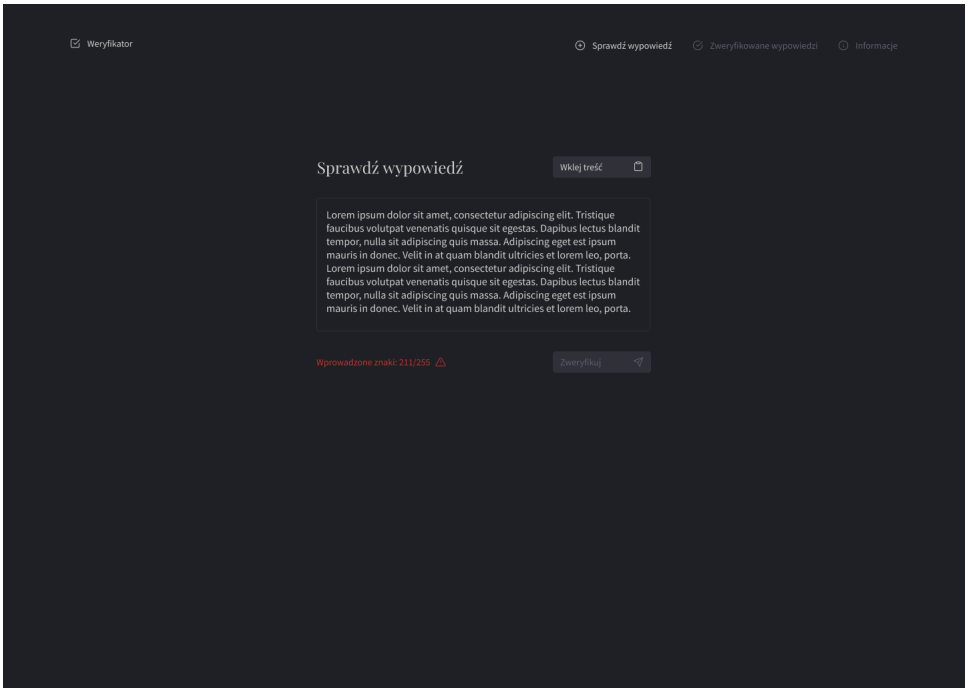
Widoki prezentowane na rysunkach 4.-11. przedstawiają części aplikacji otwarte dla wszystkich użytkowników Internetu, do których dostęp nie wymaga uwierzytelnienia. Na rysunku 4. widnieje stan startowy, w których użytkownik wprowadza, bądź wkleja wypowiedź. Limit znaków wypowiedzi zostanie ograniczony ze względu na zachowanie spójności ze zbiorem danych, na którym operuje nasza maszyna ucząca. Jeśli limit zostanie przekroczony, przycisk prowadzący do weryfikacji wypowiedzi będzie zablokowany oraz wyświetli się stosowna informacja (rys. 5.). Jeśli wprowadzona wypowiedź jest poprawna, użytkownik będzie mógł przejść do wyników poprzez naciśnięcie przycisku *Zweryfikuj* (rys. 6.).

Widok zweryfikowanej wypowiedzi składa się z kilku sekcji. W pierwszej, widoczny jest podgląd wprowadzonej wypowiedzi. Poniżej prezentowany jest wynik uzyskany po przetworzeniu wprowadzonego tekstu przez maszynę uczącą. Wynik pozytywny ukazany jest na rys. 7., negatywny na rys. 8., natomiast w przypadku, gdy maszyna ucząca nie będzie w stanie skategoryzować treści jednoznacznie, czyli z odpowiednio dużym prawdopodobieństwem, ukaże się widok z rysunku 9. Jeśli użytkownik ma wątpliwości do uzyskanego wyniku, może je zgłosić poprzez naciśnięcie przycisku *Zgłoś wynik*. Nastąpi wtedy przekierowanie do formularza zgłoszenia. Wyniki pozytywne oraz negatywne można udostępnić znajomym poprzez skopiowanie podanego linku w sekcji *Udostępnij wynik*.

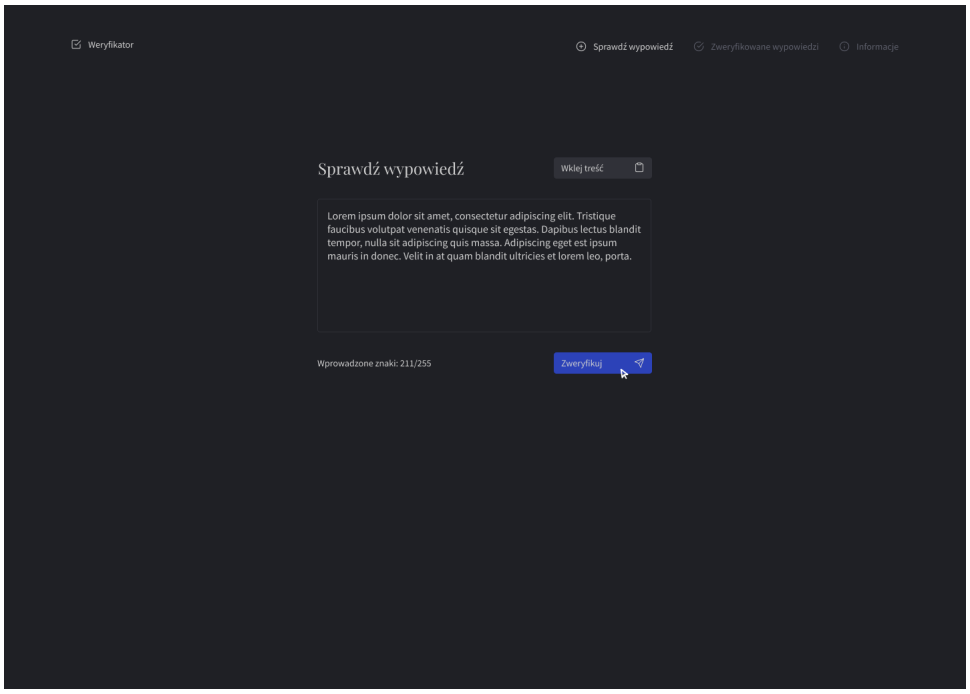
Formularz zgłoszenia (rys. 10.) składa się z dwóch zakładek. W pierwszej obecny jest faktyczny formularz, w drugiej natomiast podgląd zgłaszanego wyniku (rys. 11.). Procedurę zgłoszenia można przerwać klikając przyciski *Wróć do wyniku*, bądź *Anuluj zgłoszenie*. W formularzu użytkownik wprowadza dane, które ułatwią zespołu edytorskiemu pracę nad trafną analizą przesyłanej treści. Nie wszystkie pola są obowiązkowe. Do najistotniejszych należy wprowadzenie adresu e-mail, na który przyjdzie wiadomość z rezultatem oraz komentarz dotyczący zgłoszenia. Formularz wysyłany jest przy pomocy przycisku *Wyślij zgłoszenie*.



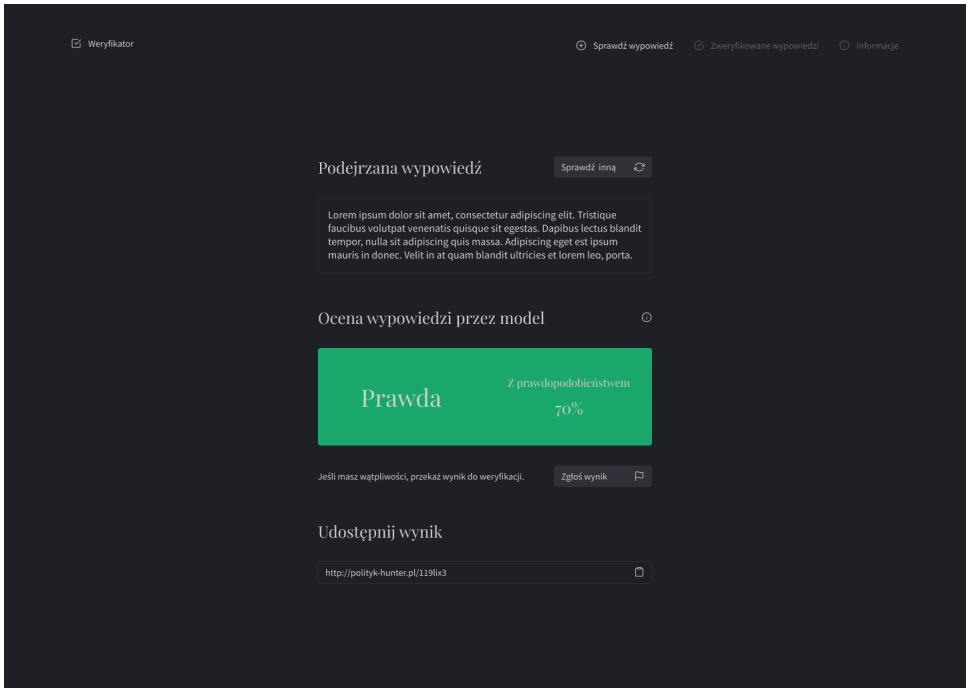
Rysunek 4: Widok wprowadzania wypowiedzi – stan domyślny



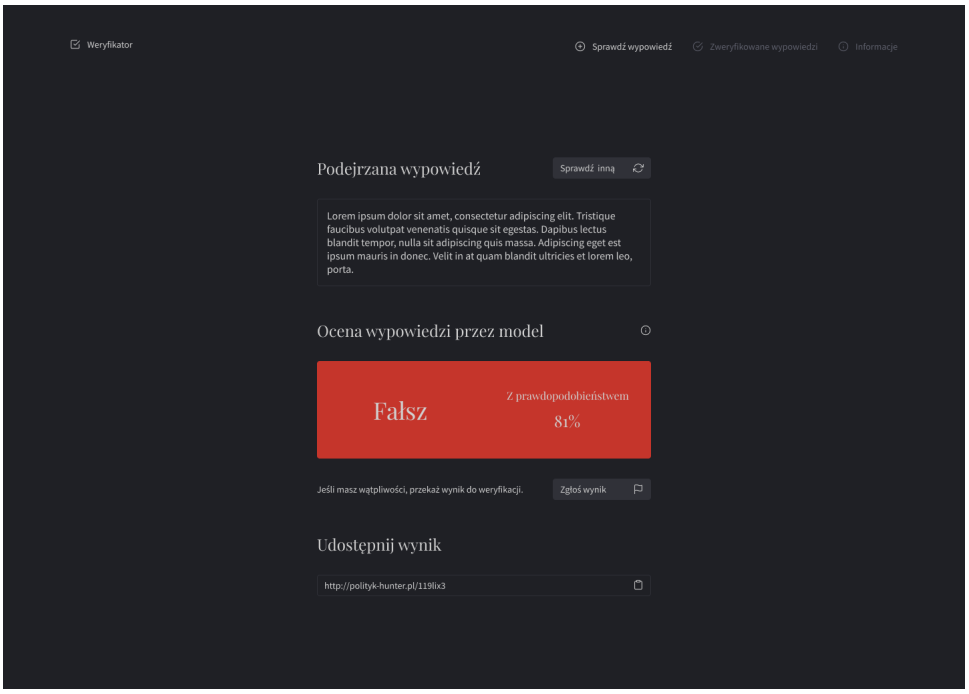
Rysunek 5: Widok wprowadzania wypowiedzi – stan błędu



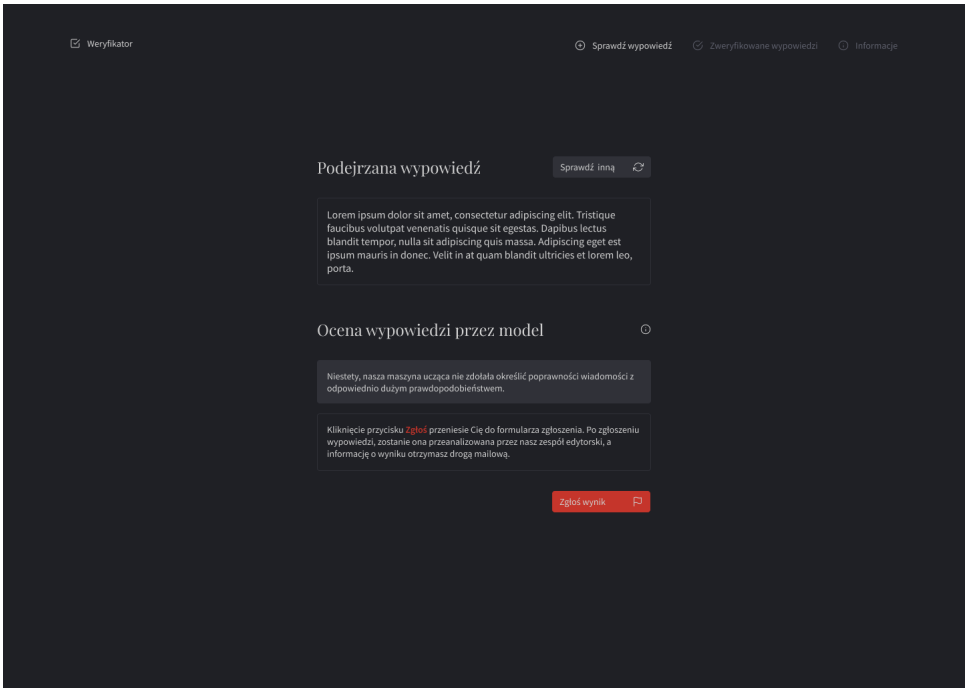
Rysunek 6: Widok wprowadzania wypowiedzi – stan przed wysłaniem



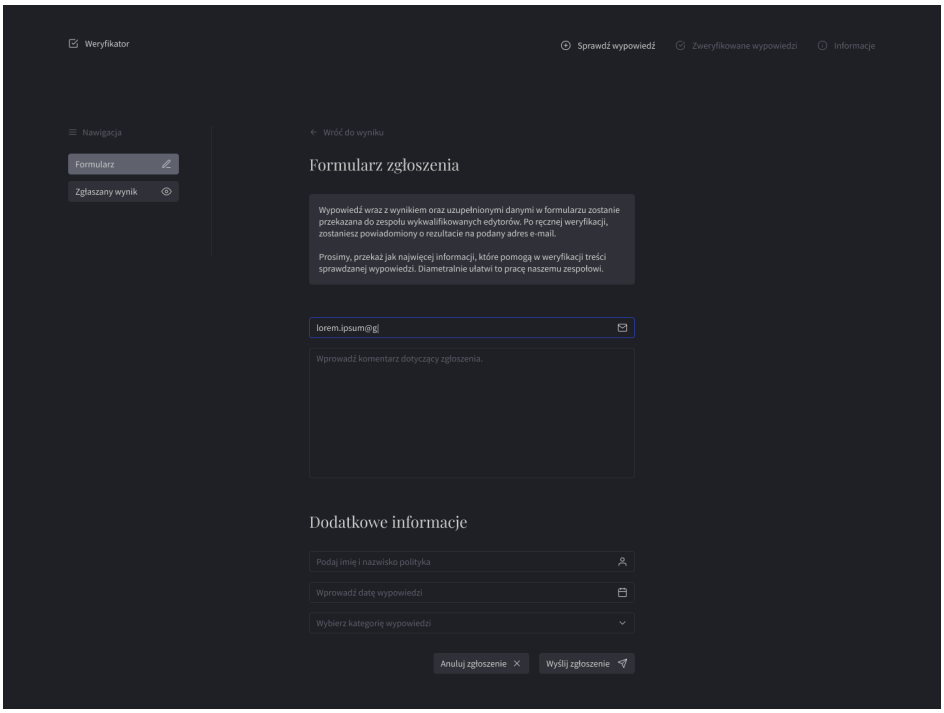
Rysunek 7: Zweryfikowana wypowiedź – wynik pozytywny



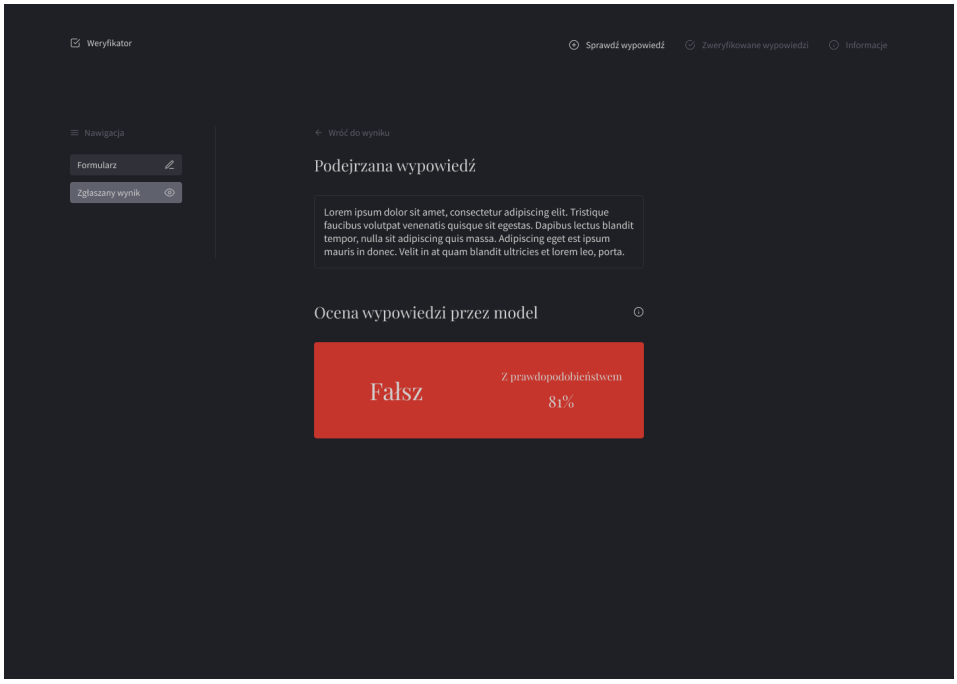
Rysunek 8: Zweryfikowana wypowiedź – wynik negatywny



Rysunek 9: Zweryfikowana wypowiedź – wynik nieokreślony



Rysunek 10: Formularz zgłoszenia



Rysunek 11: Formularz zgłoszenia – podgląd zgłaszanego wyniku

3.2.2 Widoki edytorskie

Dostęp do części edytorskiej jest ograniczony do wąskiego grona osób, które zostały uwierzytelnione oraz autoryzowane. Uwierzytelnianie następuje poprzez wypełnienie oraz wysłanie formularza logowania (rys. 12.).

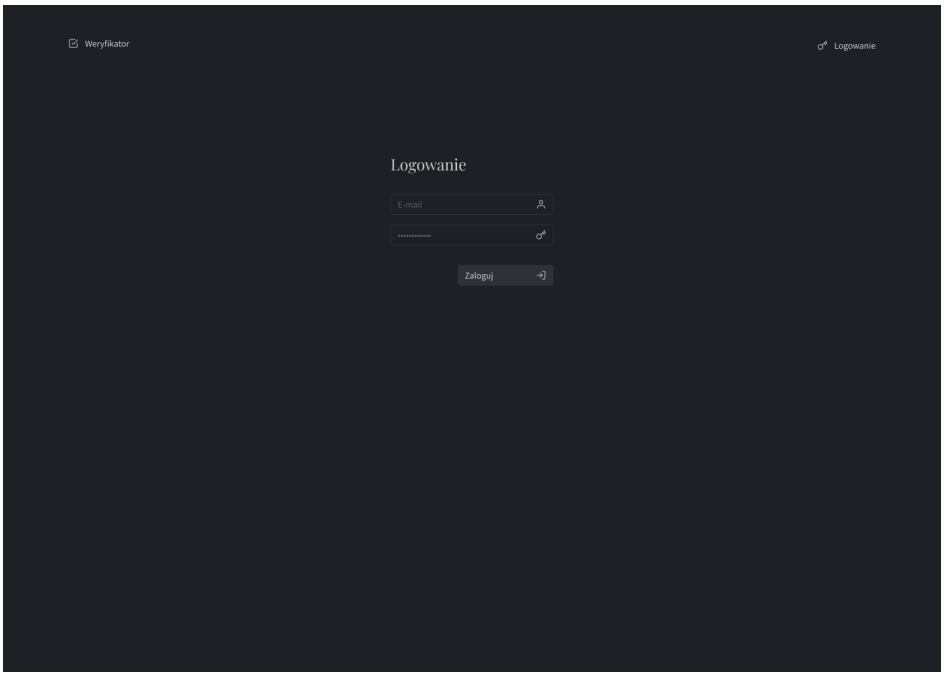
Edytor posiada dostęp do listy zgłoszeń (rys. 13.), w której przegląda wpływające raporty oraz może podejmować z nimi interakcję poprzez wciśnięcie przycisku *Zrecenzuj*. Rekordy w liście mogą być filtrowane poprzez wybieranie odpowiednich elementów w komponencie filtrowania. Widok listy zgłoszeń z wybranymi przykładowymi filtrami prezentowany jest na rysunku 14.

Widok odpowiedzi na zgłoszenie składa się z trzech głównych składowych:

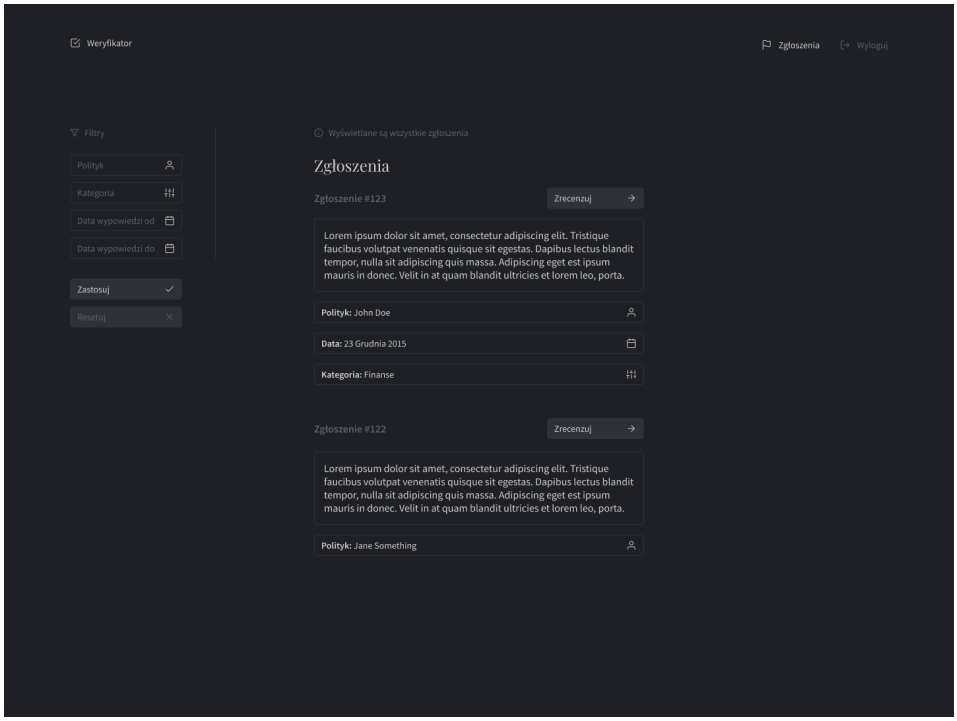
- podgląd zgłaszanego wyniku – rys. 15.,
- szczegóły z formularza zgłoszenia – rys. 16.,
- recenzja edytorska – rys 17.

Edytor może poruszać się między tymi częściami klikając przyciski *Dalej* oraz *Wstecz*, bądź wybierając odpowiednią opcję z panelu nawigacji. Edytor może również przerwać proces recenzowania, dzięki czemu powróci do listy wszystkich zgłoszeń, poprzez naciśnięcie przycisku *Anuluj*, bądź *Wróć do wszystkich zgłoszeń*.

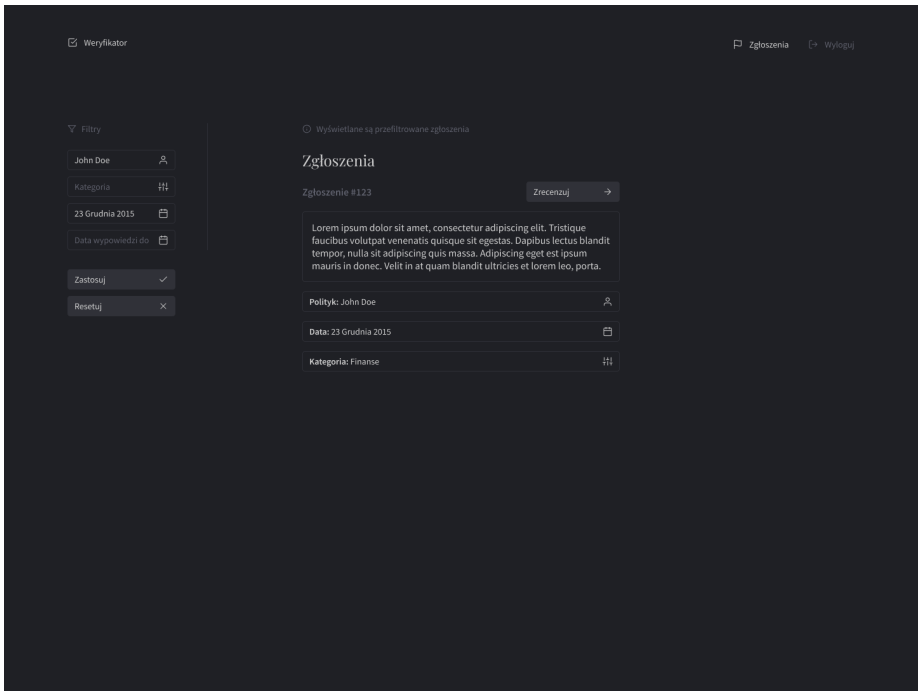
W części recenzji edytor wybiera z listy ocenę wypowiedzi – prawda (rys. 18.), lub fałsz (rys. 19.) oraz wpisuje wyjaśniający komentarz. Po zaakceptowaniu przyciskiem *Wyślij*, zgłaszający użytkownik zostanie powiadomiony drogą mailową o finalnym werdykcie.



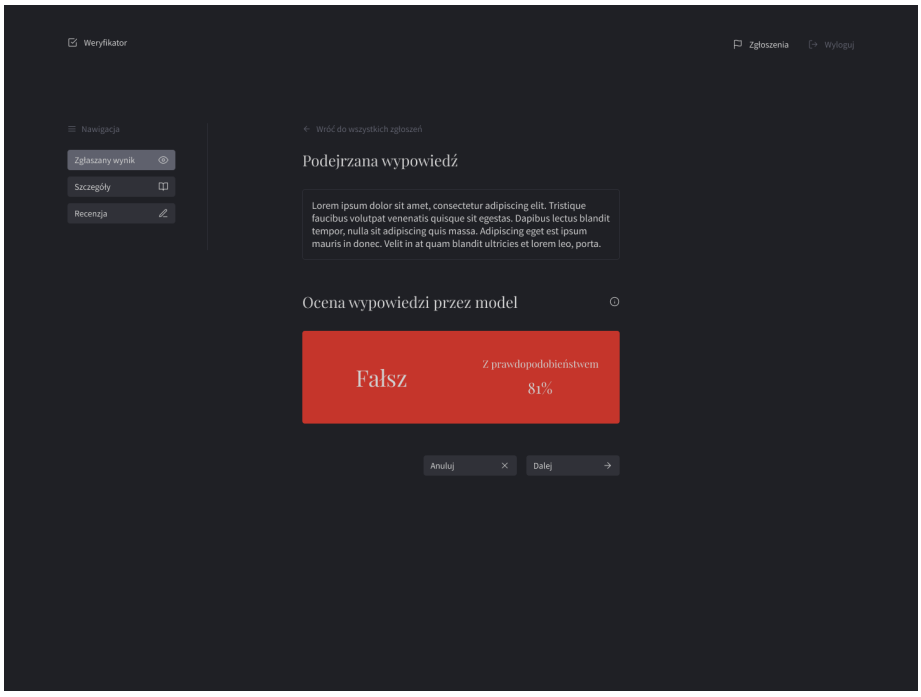
Rysunek 12: Ekran logowania do panelu edytora



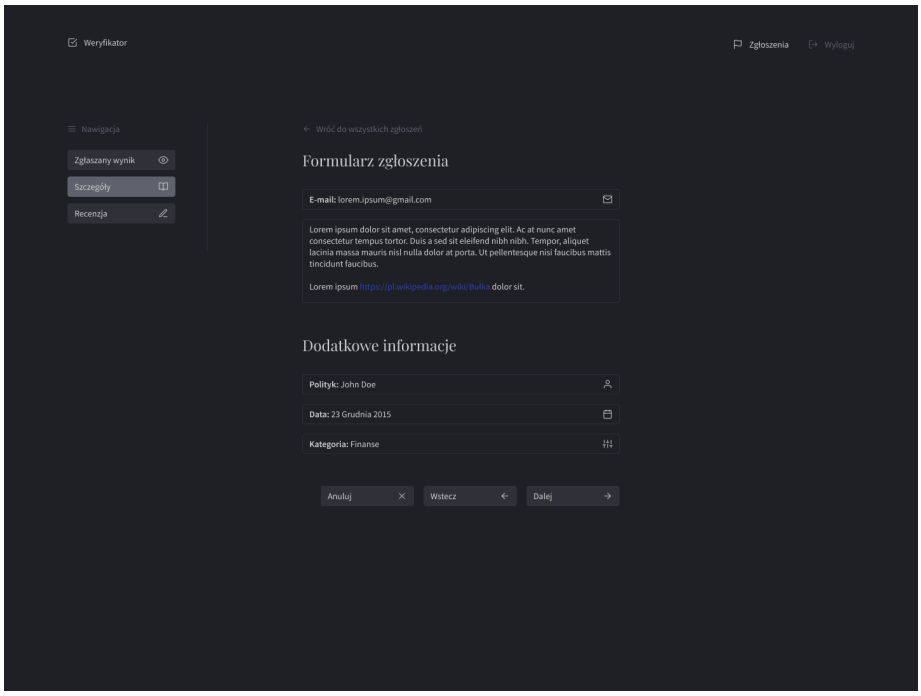
Rysunek 13: Lista otrzymanych zgłoszeń



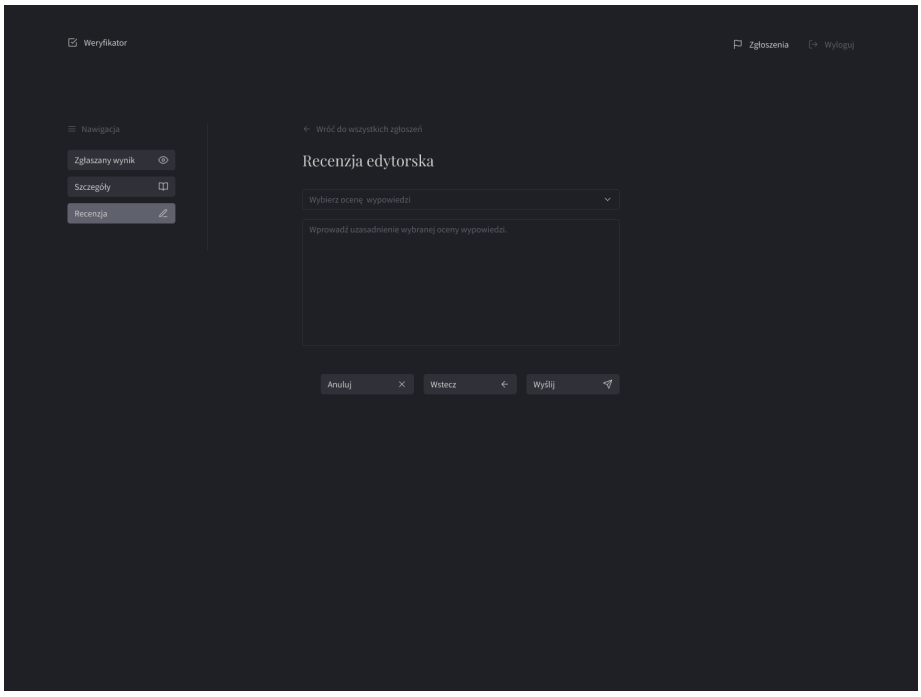
Rysunek 14: Lista otrzymanych zgłoszeń – zaaplikowane filtry



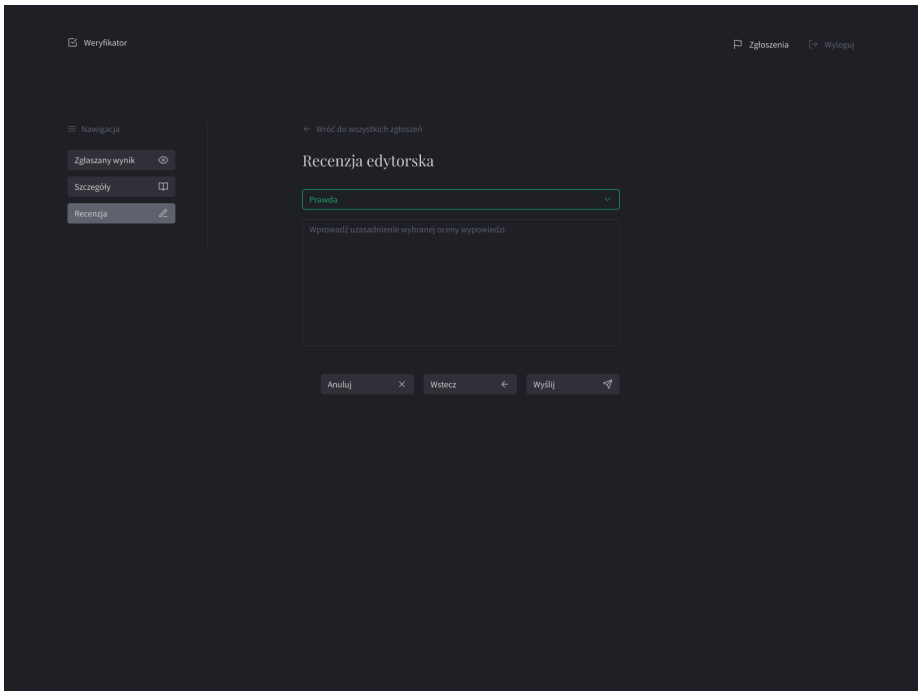
Rysunek 15: Podgląd zgłoszenia



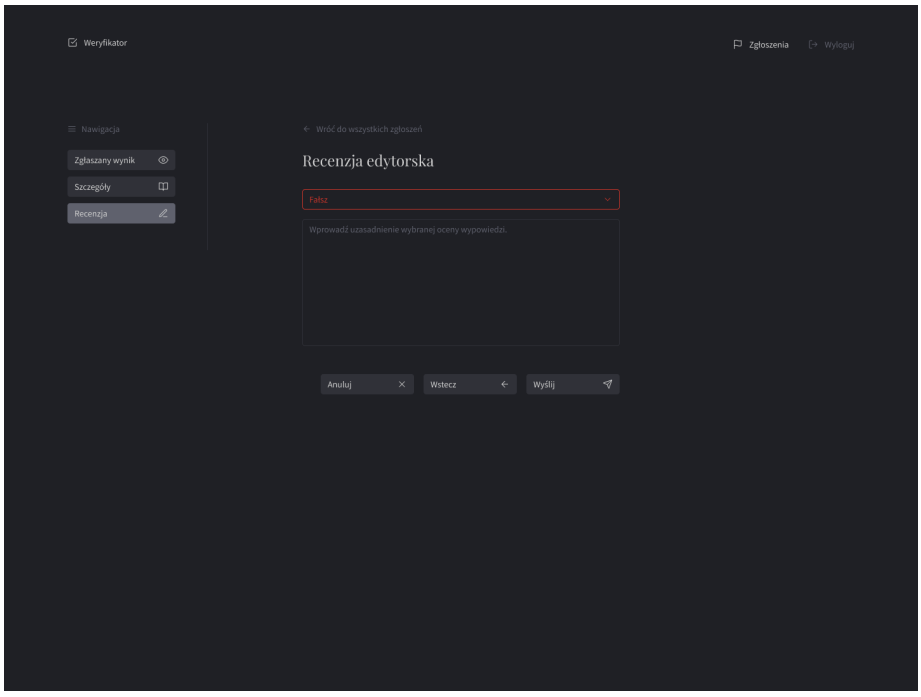
Rysunek 16: Szczegóły zgłoszenia



Rysunek 17: Recenzja edytorska – stan domyślny



Rysunek 18: Recenzja edytorska – ocena pozytywna



Rysunek 19: Recenzja edytorska – ocena negatywna

4 Trzeci kamień milowy - testy modelu oraz implementacja aplikacji webowej

4.1 Przetwarzanie tekstu

W pierwszej fazie surowy tekst został przetworzony przy pomocy funkcji znajdujących się w pliku *text_preprocessing.py*. Wśród najprostszych można wyróżnić operacje takie jak: usuwanie znaków interpunkcyjnych, zamiana wszystkich wielkich liter na małe oraz redukcja słów nieistotnych lub o niewielkim znaczeniu (w angielskiej literaturze określanych jako *stop words*). Ostatnia z wymienionych operacji dotyczy wyrazów, które w pomijalnym stopniu lub wręcz wcale nie wpływają na znaczenie całego tekstu. Są to między innymi spójniki: „i", „oraz", „lub". Oprócz tego tekst poddano lematyzacji polegającej na sprowadzeniu formy fleksyjnej każdego wyrazu do postaci słownikowej (tak zwane hasłowanie). Lematyzacja oraz usuwanie *stop words* zostało wykonane przy użyciu narzędzi *WCRFT2* [12] oraz *TermoPL* [17] dostarczanych przez serwis *Clarin-PL*.

Przykładowy tekst poddany procesowi lematyzacji:

> Na Uniwersytecie Warszawskim powstał taki raport, który jest oczywiście państwu znany, z którego wyraźnie wynika, że spółki Skarbu Państwa mają bardzo szczególne preferencje, lokując swoje zamówienia reklamowe w mediach. To jest zresztą spora kwota, bo w 2019 r. spółki Skarbu Państwa wydały prawie 1200 mln zł tylko na reklamę medialną (...) poza Internetem itd.

Przykładowy tekst po przetworzeniu:

> na uniwersytet Warszawskie powstać taki raport który być oczywiście państwo znany z który wyraźnie wynikać że spółka skarb państwo maja bardzo szczególny preferencja lokować swoje zamówienie reklamowy w media medium to być zresztą spora kwota bo w 2019 r spółka skarb państwo wydać prawie 1200 milion złoty tylko na reklama medialny ... poza Internet internet itd

4.2 Generacja i selekcja cech

Drugą fazą pracy z tekstem było wygenerowanie na jego podstawie zbioru opisujących go cech, które odgrywają bardzo ważną rolę przy problemie klasyfikacji. Tak jak w przypadku przetwarzania tekstu tak i w tym kroku można wyróżnić cechy prostsze oraz te bardziej skomplikowane. W projekcie wygenerowano proste cechy takie jak: odsetek wielkich liter w tekście, wykrzykników, znaków zapytania, cudzysłowów, czy również szeroko pojętych znaków interpunkcyjnych, a także liczba znaków tworzących dany tekst. Bardziej wyrafinowane cechy to natomiast wartość sentymentu, a także procentowy udział słów o wydźwięku negatywnym oraz pozytywnym. W celu uzyskania wartości tych cech zostało wykorzystane gotowe narzędzie *Ccl_emo* dostarczane, tak jak w powyższej sekcji, przez serwis *Clarin-PL*.

Dodatkowo w celu ekstrakcji informacji z tekstu wykorzystano metodę wektoryzacji *TF-IDF* (ang. *Term Frequency - Inverse Document Frequency*) [1]. Ma ona za zadanie liczbowe wyrażanie istotności wyrazu w dokumencie, który wchodzi w skład szerszej kolekcji dokumentów zwanej korpusem. Jest to osiągane za pomocą sprawdzenia ile razy dany wyraz pojawia

się w analizowanym dokumencie przy równoczesnym zwracaniu uwagi na liczbę jego wystąpień w pozostałych dokumentach wchodzących w skład korpusu. Do wyrażenia końcowego wzoru wykorzystywanego w tej metodzie konieczne jest przedstawienie dwóch poniższych terminów:

- ważenie częstością termów (ang. *term frequency*), czyli obliczenie częstotliwości występowania wyrazu lub wyrażenia w dokumencie za pomocą wzoru

$$tf(w,d) = log(1 + f(w,d)) \tag{1}$$

, gdzie w oznacza analizowany wyraz lub wyrażenie, d to dokument źródłowy, a *f(w, d)* to częstotliwość występowania w w d.

- odwrotna częstość w dokumentach (ang. *inverse document frequency*) wyrażana wzorem

$$idf(w,D) = log(\frac{N}{f(w,D)}) \tag{2}$$

, gdzie D oznacza korpus, *f(w, D)* - liczbę dokumentów w korpusie, które zawierają dany term w, a N liczbę wszystkich dokumentów w korpusie.

Iloczynem powyższych wzorów jest wzór wykorzystywany przez metodę *tf-idf*:

$$tfidf(w,d,D) = tf(w,d) * idf(w,D) \tag{3}$$

Przy projekcie wykorzystano modele n-gram o zasięgu od 1 do 3 wyrazów.

Aby trenowanie modeli odbywało się efektywnie konieczne było wykonanie selekcji cech i wybranie z nich tylko tych mających najbardziej znaczący wpływ na ich klasyfikację. Zbyt duża liczba cech znacząco wydłuża proces uczenia się modeli oraz może mieć negatywne skutki dla jego jakości objawiające się w postaci zjawiska przeuczenia. Wobec tego zastosowany został test zgodności chi-kwadrat służący do oceny zależności pomiędzy rozkładem częstości odpowiedzi w zakresie jednej zmiennej, w odniesieniu do drugiej zmiennej.

Test polega na weryfikacja hipotezy zerowej przez porównanie ze znanym, stablicowanym rozkładem χ^2. W ramach testu należy w tablicach statystycznych sprawdzić ile w naszym przypadku wynosi wartość krytyczna testu. Zależy to od założonego wcześniej poziomu istotności α (prawdopodobieństwo popełnienia błędu I rodzaju, czyli błędu polegający na odrzuceniu hipotezy zerowej, która w rzeczywistości jest prawdziwa). Określa również maksymalne ryzyko błędu, jakie jesteśmy skłonni zaakceptować. Wybór wartości α zależy od nas i od tego jak dokładnie chcemy weryfikować daną hipotezę, najczęściej przyjmuje się $\alpha = 0.05$ - postąpiono tak również w naszym przypadku.[6]

Test chi-kwadrat zwraca m. in. p-wartość (*p-value*, czyli tzw. graniczny poziom istotności - najmniejszy, przy którym zaobserwowana wartość statystyki testowej prowadzi do odrzucenia hipotezy zerowej. Jest to więc taki poziom istotności, przy którym zmienia się decyzja testu. P-wartość pozwala bezpośrednio ocenić wiarygodność hipotezy. Im p-wartość jest większa, tym bardziej hipoteza H_0 jest prawdziwa [6]. Mała p-wartość świadczy przeciwko hipotezie zerowej. Znajomość p-wartości pozwala przeprowadzić testowanie dla dowolnego poziomu istotności:

1. odrzucamy hipotezę zerową H_0, gdy p-wartość $\leq \alpha$,

2. nie mamy podstaw do odrzucenia hipotezy zerowej H_0, gdy p-wartość $> \alpha$

Test ten pozwolił stwierdzić, że wartościowe są 4 cechy (ich p-wartości były mniejsze niż α): odsetek znaków interpunkcyjnych (punctuation%), długość tekstu (length), wartość sentymentu (sentiment) oraz procentowy udział wyrazów nacechowanych pozytywnie w tekście (positive_words%). Wobec tego powstał nowy zbiór z wyselekcjonowanymi cechami pozwalający kontynuować pracę nad projektem.

4.3 Implementacja modelu

4.3.1 Las losowy (ang. *random forest)*

Opis metody

Pierwszym z testowanych modeli wykorzystywał metodę lasu losowego polegającą na konstrukcji wielu drzew decyzyjnych podczas procesu uczenia, a następnie generacji klasy wyjściowej w wyniku głosowania większościowego nad klasami wyznaczonymi przez poszczególne drzewa decyzyjne.

Dostosowanie hiperparametrów

Na listingu 2 przedstawione zostały wszystkie przebadane parametry przy wykorzystaniu metody *GridSearch*, która tak samo jak sam klasyfikator lasu losowego, dostarczana jest przez bibliotekę *scikit-learn*[10]. Dodatkowo skorzystano z metody pięciokrotnej warstwowej walidacji krzyżowej w celu uzyskania dokładniejszych wyników świadczących o dokładności modelu.

Przebadane parametry to:

- *max_ depth* - najdłuższa możliwa ścieżka między korzeniem a liściem drzewa,
- *max_ features* - maksymalna liczba cech dostarczana każdemu z drzew (losowość w tym algorytmie osiągana jest losowym doborem liczby cech wykorzystywanych w drzewie),
- *min_ samples_ split* - minimalna wymagana liczba obserwacji w danym węźle aby móc go dalej dzielić,
- *n_ estimators* - liczba drzew konstruowanych przez algorytm.

Listing 2: Las losowy - słownik testowanych wartości parametrów

```
rf = RandomForestClassifier(random_state=1410)

param = {
    'max_depth': [1, 2, 5, 10, 20, 40, 50, 80, 90, 100, 110, None],
    'max_features': ['sqrt', 'auto', 'log2'],
    'min_samples_split': [2, 4, 6, 8],
    'n_estimators': [10, 100, 250, 500, 1000]
}

gs = GridSearchCV(rf, param, cv=StratifiedKFold(n_splits=5, random_state=1410, shuffle=True),
    n_jobs=-1, return_train_score=True)
```

W tabeli 1 przedstawione zostały parametry odpowiadające modelom charakteryzującym się najwyższą dokładnością osiągniętą na zbiorze testowym. Ze względu na najkrótszy czas podejmowania decyzji jako najlepszy wybrano model o następujących wartościach parametrów: *max_ depth=2*, *max_ features=auto* (wartość *auto* oznacza liczbę wszystkich dostępnych cech, która w tym przypadku wynosi 4), *min_ samples_ split=6*, *n_ estimators=10*.

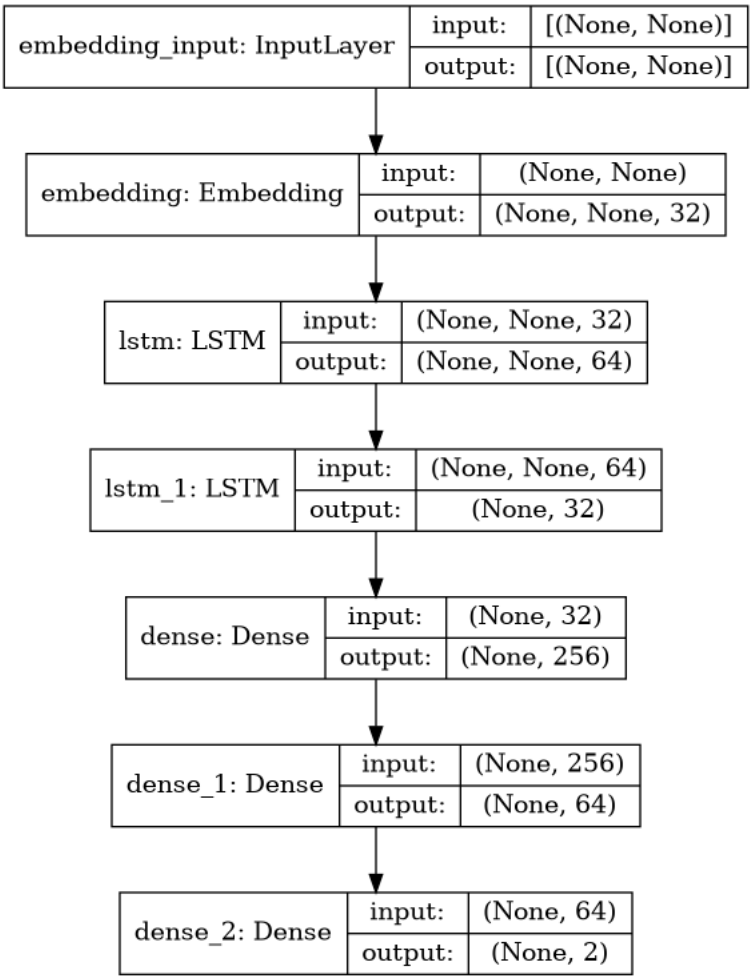
Tabela 1: Las losowy - najlepsze wyniki *GridSearch* dla podanego zestawu parametrów

mean_fit_time	mean_score_time	params	mean_test_score
0.277398	0.056399	'max_depth': 1, 'max_features': 'sqrt', 'min_samples_split': 2, 'n_estimators': 10	0.720370
1.463465	0.121349	'max_depth': 2, 'max_features': 'sqrt', 'min_samples_split': 2, 'n_estimators': 100	0.720370
0.114245	0.017376	'max_depth': 2, 'max_features': 'auto', 'min_samples_split': 6, 'n_estimators': 10	0.720370
11.284100	1.316113	'max_depth': 2, 'max_features': 'auto', 'min_samples_split': 4, 'n_estimators': 1000	0.720370
5.308424	0.367348	'max_depth': 2, 'max_features': 'auto', 'min_samples_split': 4, 'n_estimators': 500	0.720370

4.3.2 Rekurencyjna sieć neuronowa (RNN)

Opis metody

Drugi testowany model różni się podejściem od pierwszego. Las losowy zaliczamy do metod *shallow learning'u* natomiast prezentowany tutaj model, rekurencyjna sieć neuronowa należy do metod *deep learning'u*. Metody te możemy opisać w skrócie jako wykorzystujące sieci neuronowe. Rekurencyjne sieci neuronowe charakteryzują się tym, że na wejście ich neuronów podawane są oprócz standardowego wejścia ich obecne wyjście. Na rysunku 20 została przedstawiona architektura sieci użyta w projekcie, jej łączna liczba parametrów wyniosła 94,274. Rysunek został wygenerowany przez biblioteke Keras.



Rysunek 20: Model rekurencyjnej sieci neuronowej

Dostosowanie hiperparametrów

W celu dobrania optymalnych hiperparametrów wykorzystano metodę pięciokrotnej warstwowej walidacji krzyżowej, tak samo jak w poprzednim modelu.

Przebadane parametry to:

- *batch_ size* - liczba przykładów, po których sieć zmodyfikuje wagi nauronów.
- *epochs* - ile razy cały zbiór testowy zostanie rozpropagowany przez sieć.
- *loss* - funkcja określająca różnicę pomiędzy wynikiem zwróconym przez sieć a wartością oczekiwaną.
- *optimizer* - funkcja, która aktualizuje parametry modelu podczas nauki.

Listing 3: RNN - funkcja ewaluująca parametry sieci

```
def evaluate_rnn_model_params(X_tfidf_feat: pd.DataFrame, labels: pd.DataFrame):

    model = KerasClassifier(build_fn=create_model)

    param_grid = {
        'epochs': [20, 40, 60, 80],
        'batch_size': [16, 32, 64, 128],
        'optimizer': ['rmsprop', 'adam', 'SGD'],
        'loss': ['mse', 'categorical_crossentropy']
    }

    grid = GridSearchCV(estimator=model, param_grid=param_grid, cv=StratifiedKFold(n_splits=5,
        random_state=1410, shuffle=True),
                        n_jobs=-1, return_train_score=True)
    grid_result = grid.fit(X_tfidf_feat, labels)
```

W tabeli 2 przedstawiono wyniki pięciu najlepszych zestawów hiperparametrów sieci dla naszego zbioru testowego. Ze względu na bardzo zbliżone wyniki żaden z modeli znacząco nie góruje nad pozostałymi dla tego ze względu na najkrótszy czas potrzebny do nauki modelu jako najoptymalniejsze zostały wybrane parametry: batch_size: 64, epochs: 40, loss: categorical_crossentropy, optimizer: SGD

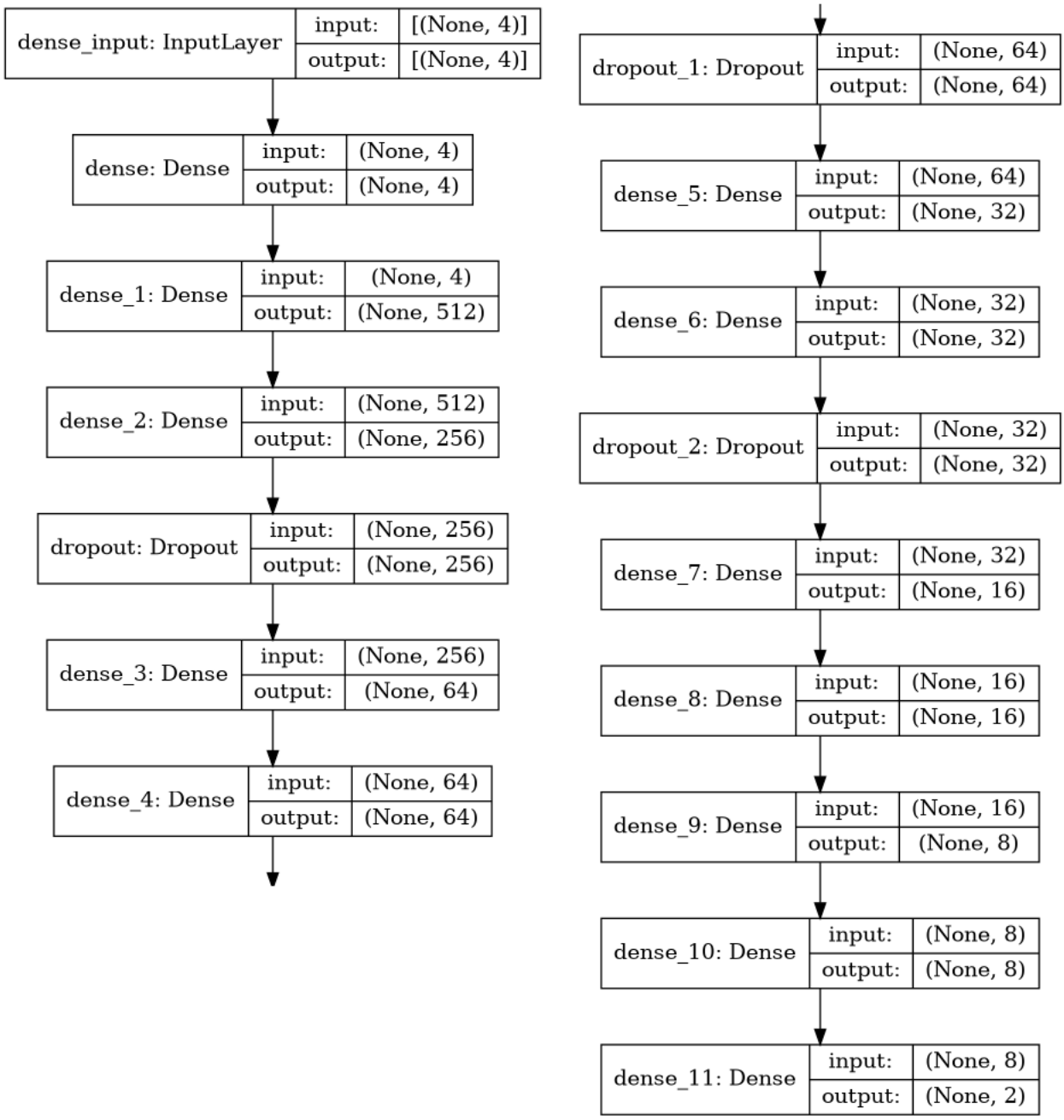
Tabela 2: Rekurencyjna sieć neuronowa - najlepsze wyniki *GridSearch* dla podanego zestawu parametrów

mean_fit_time	mean_score_time	params	mean_test_score
61.598097	0.855766	'batch_size': 128, 'epochs': 80, 'loss': 'categorical_crossentropy', 'optimizer': 'SGD'	0.720370
63.336985	2.237119	'batch_size': 128, 'epochs': 60, 'loss': 'categorical_crossentropy', 'optimizer': 'SGD'	0.720370
103.954301	2.042084	'batch_size': 64, 'epochs': 80, 'loss': 'categorical_crossentropy', 'optimizer': 'SGD'	0.720370
82.531829	1.659955	'batch_size': 64, 'epochs': 60, 'loss': 'categorical_crossentropy', 'optimizer': 'SGD'	0.720370
60.139575	1.700414	'batch_size': 64, 'epochs': 40, 'loss': 'categorical_crossentropy', 'optimizer': 'SGD'	0.720370

4.3.3 Perceptron wielowarstwowy (MLP)

Opis metody

Trzecim testowanym modelem był perceptron wielowarstwowy, najpopularniejszy i zarazem najprostszy model sieci neuronowej. Zbudowana jest ona z warstw neuronów, które są na swoich wejściach oraz wyjściach połączone z każdym neuronem odpowiednio z poprzedniej oraz kolejnej warstwy. Sieci te charakteryzują się zazwyczaj większą liczbą parametrów modelu w porównaniu z innymi typami sieci. Na rysunku 21 została zaprezentowana architektura naszego perceptronu, jego łączna liczba parametrów wyniosła 158678.



Rysunek 21: Model sieci perceptronu wielowarstwowego

Dostosowanie hiperparametrów

Tak jak w przypadku poprzednich modeli, tutaj też wykorzystano tą samą metodę walidacji krzyżowej w celu wyznaczenia optymalnych hiperparametrów.

Zostały przebadane te same parametry jak w przypadku sieci rekurencyjnej oraz dodatkowy hiperparametr *dropout_rate*. Jest to współczynnik opisujący procent losowych neuronów, które są pomijane w trakcie procesu uczenia. Służy to zmniejszeniu efektu uczenia się danych *na pamięć* przez sieć.

Listing 4: MLP - funkcja ewaluująca parametry sieci

```
def evaluate_mlp_model_params(X_tfidf_feat: pd.DataFrame, labels: pd.DataFrame):
    nmb_of_features = X_tfidf_feat.shape[1]

    model = KerasClassifier(build_fn=create_model, nmb_of_features=nmb_of_features)

    param_grid = {
        'epochs': [20, 40, 60, 80],
        'batch_size': [16, 32, 64, 128],
        'dropout_rate': [0.2, 0.25, 0.3],
        'optimizer': ['rmsprop', 'adam', 'SGD'],
        'loss': ['mse', 'categorical_crossentropy']
    }

    grid = GridSearchCV(estimator=model, param_grid=param_grid, cv=StratifiedKFold(n_splits=5,
        random_state=1410, shuffle=True),
                        n_jobs=-1, return_train_score=True)
    grid_result = grid.fit(X_tfidf_feat, labels)
```

W tabeli 3 przedstawiono wyniki dla najlepszych pięciu zestawów hiperparametrów. Ze względu na minimalnie lepszy wyniki testów w porównaniu do pozostałych zestawów dla tego modelu jako zbiór najoptymalniejszych parametrów zostały wybrane: batch_size: 16, dropout_rate: 0.25, epochs: 20, loss: categorical_crossentropy, optimizer: adam

Tabela 3: Perceptron wielowarstwowy - najlepsze wyniki *GridSearch* dla podanego zestawu parametrów

mean_fit_time	mean_score_time	params	mean_test_score
26.398941	0.933909	'batch_size': 16, 'dropout_rate': 0.25, 'epochs': 20, 'loss': 'cat_crossent', 'optimizer': 'adam'	0.720650
20.087320	0.323605	'batch_size': 128, 'dropout_rate': 0.3, 'epochs': 80, 'loss': 'cat_crossent', 'optimizer': 'SGD'	0.720370
52.307115	0.965127	'batch_size': 32, 'dropout_rate': 0.25, 'epochs': 60, 'loss': 'cat_crossent', 'optimizer': 'rmsprop'	0.720370
37.881862	0.858528	'batch_size': 32, 'dropout_rate': 0.25, 'epochs': 60, 'loss': 'cat_crossent', 'optimizer': 'SGD'	0.720370
12.113878	0.699276	'batch_size': 128, 'dropout_rate': 0.2, 'epochs': 40, 'loss': 'cat_crossent', 'optimizer': 'SGD'	0.720370

4.3.4 Wzmocnienie gradientowe (ang. *gradient boosting)*

Opis metody

Wzmocnienie gradientowe to technika uczenia maszynowego stosowana w przypadku problemów z regresją lub klasyfikacją, która tworzy model predykcyjny w postaci zbioru słabych modeli predykcyjnych, zazwyczaj drzew decyzyjnych. W ramach modelu występują następujące kroki:

1. Na początku modelujemy dane za pomocą prostych modeli i analizujemy dane pod kątem błędów. Błędy te oznaczają punkty danych, które są trudne do dopasowania w prostym modelu.

2. Następnie w przypadku późniejszych modeli skupiamy się szczególnie na tych trudnych do dopasowania danych, aby je uzyskać.

3. Na koniec łączymy wszystkie predyktory, przypisując wagi każdemu predyktorowi.

Boosting jest ogólną metodą, u podstaw której leży idea poprawy dokładności działania dowolnego algorytmu uczącego. Technika ta polega na stosowaniu sekwencji prostych modeli, przy czym każdy kolejny model przykłada większą wagę do tych obserwacji, które zostały błędnie zaklasyfikowane przez poprzednie modele. W naszym projekcie wykorzystano *GradientBoostingClassifier* z biblioteki *scikitlearn.esamble*[10]

Dostosowanie hiperparametrów

Na poniższym listingu przedstawiono część parametrów, które zostały przebadane przy pomocy metody *GridSearch* [10]. Natomiast w celu weryfikacji dokładności modelu użyta została pięciokrotna walidacja krzyżowa. Najważniejsze z testowanych w ramach optymalizacji parametrów to:

- learning_rate - parametr mówiący po każdej wyliczonej iteracji jaki krok chcemy dać do przodu. Im większy krok tym szybciej zbliżamy się do celu, ale jeśli będzie zbyt duży to możemy nie dojść do najlepszego wyniku,
- max_depth: maksymalna głębokość prostych drzew. Im głębsze drzewa tym model jest mocniejszy, ale trzeba uważać by nie przeuczyć modelu,
- n_estimators : liczba drzew, które powinny być zbudowane

Listing 5: Funkcja testująca różne parametry klasyfikatora opartego o boosting

```
1
2
3  def tuning_model_metaparameters(X_tfidf_feat: pd.DataFrame, labels: pd.DataFrame):
4      gbc = GradientBoostingClassifier(random_state=0).fit(X_tfidf_feat, labels)
5
6      param_grid = {
7          'max_depth': [1, 5, 10, 50, 100],
8          'learning_rate': [0.1, 0.3, 0.5, 0.7, 1.0],
9          'min_samples_leaf': [1, 2, 4],
10         'max_features': ['sqrt', 'auto', 'log2'],
11         'min_samples_split': [2, 4, 6, 8],
12         'n_estimators': [100, 250, 500, 1000]
13     }
14
15     print("starting grid search")
16     gs = GridSearchCV(gbc, param_grid, cv=StratifiedKFold(n_splits=5, random_state=1410, shuffle=
           True), n_jobs=-1,
17                       return_train_score=True)
18     print("testing")
19     gs_fit = gs.fit(X_tfidf_feat, labels)
20
21     print(pd.DataFrame(gs_fit.cv_results_).sort_values('mean_test_score', ascending=False))
```

Tabela 4 przedstawia najlepsze uzyskane wyniki przez wzmocnienie gradientowe dla parametrów widocznych na listingu 5 przy wykorzystaniu metody *GridSearch*.

Tabela 4: Gradient boosting - najlepsze wyniki *GridSearch* dla podanego zestawu parametrów

mean_fit_time	mean_score_time	params	mean_test_score
27.903869	0.068759	'learning_rate': 0.5, 'max_depth': 100, 'min_samples_split': 2, 'n_estimators': 500	0.610255
27.978025	0.067793	'learning_rate': 0.5, 'max_depth': 100, 'min_samples_split': 4, 'n_estimators': 500	0.610255
28.264832	0.072211	'learning_rate': 0.5, 'max_depth': 50, 'min_samples_split': 2,'n_estimators': 500	0.609977
27.861177	0.068008	'learning_rate': 0.5, 'max_depth': 50, 'min_samples_split': 4, 'n_estimators': 500	0.609977
27.351431	0.066109	'learning_rate': 0.7, 'max_depth': 50, 'min_samples_split': 4, 'n_estimators': 500	0.596247

4.3.5 Wybór modelu

Po porównaniu dokładności uzyskanych przy wykorzystaniu omówionych w tej sekcji modeli zdecydowano się ostatecznie na wybór modelu lasu losowego o parametrach: *max_depth=2*, *max_features=auto* (wartość *auto* oznacza liczbę wszystkich dostępnych cech, która w tym przypadku wynosi 4), *min_samples_split=6* oraz *n_estimators=10*.

Jedynym modelem wykazującym się większą dokładnością był *MLP* o następujących wartościach parametrów: *batch_size=16*, *dropout_rate=0.25*, *epochs=20*, *loss=cat_crossent*, *optimizer=adam*. Średnia dokładność wyniosła wówczas 0.720650, podczas gdy dla wybranego modelu lasu losowego wyniosła ona 0.720370. Przy podejmowaniu decyzji czynnikami decydującymi okazały się być: średni czas trenowania modelu oraz średni czas predykcji. Tabela 5 przedstawia porównanie tych wartości dla najlepszych wersji poszczególnych modeli.

Tabela 5: Porównanie najlepszych wyników *GridSearch* dla poszczególnych modeli

model	mean_fit_time	mean_score_time	mean_test_score
Random Forest	0.114245	0.017376	0.720370
RNN	61.598097	0.855766	0.720370
MLP	26.398941	0.933909	0.720650
Gradient Boosting	27.903869	0.068759	0.610255

4.4 Implementacja aplikacji

4.4.1 Część serwerowa obsługująca model

W celu umożliwienia komunikacji aplikacji klienckiej z modelem służącym do klasyfikacji tekstu na *fake news* oraz wypowiedź prawdziwą stworzono serwer implementujący jedną metodę *HTTP POST*. Aby uzyskać prawdopodobieństwo z jaką wiadomość została przyporządkowana danej klasie należało wysłać żądanie na *endpoint /classify/*, w którego treści powinna znajdować się informacja w formacie *json* o treści *{'statement': <treść wiadomości do klasyfikacji>}*. Przykładowa treść odpowiedzi wygląda następująco:

{"true": 0.37873372435569763, "fake": 0.62126624584198}

Listing 6 przedstawia implementację omawianego serwera przy wykorzystaniu bibliotek *fastapi*[13] oraz *uvicorn*[2].

Listing 6: Implementacja serwera umożliwiającego dostęp do wytrenowanego modelu.

```
model = rf.load_model('models/rf_model.pickle')
app = FastAPI()


class Item(BaseModel):
    statement: str


@app.post("/classify/")
async def classify_text(item: Item):
    features = create_selected_features_for_single_text(item.statement)
    prediction = rf.predict_single_instance(model, features)
    return dict(zip(['fake', 'true'], prediction))


```

```
16  def main():
17      uvicorn.run(app, port=8000)
```

4.4.2 Część serwerowa obsługująca aplikację

Ponad warstwą części serwerowej, obsługującej zapytania do modelu powstała aplikacja serwerowa obsługująca bezpośrednio kliencką aplikację webową. Istnienie dwóch aplikacji serwerowych z pewnością było do uniknięcia, jednak takie rozwiązanie miało większy walor edukacyjny - zespół miał okazję poznać nowe dla niektórych członków technologie. Postanowiono skorzystać z poniższego *stacku technologicznego*:

- Baza danych - MongoDB[9] - jest to otwarty, nierelacyjny system zarządzania danych. Dane przechowywane są jako dokumenty w formacie JSON. Wykorzystaliśmy usługę chmurową MongoDB Atlas. Jako narzędzie ODM (*Object Document Mapping*) użyliśmy Mongoose. Umożliwia ono mapowanie dokumentów w środowisku *Node.js*.

- Aplikacja serwerowa - Express.js[4] - framework działający w środowisku node.js, czyli Javascriptu po stronie serwera. Charakteryzuje się on minimalnym zakresem funkcjonalności, ale możliwością szerokiego rozszerzenia i dostosowania przez zewnętrzne biblioteki.

4.4.3 Uwierzytelnianie i autoryzacja

Obsługa procesu uwierzytelniania oraz autoryzacji wyzwalana jest z części klienckiej, a obsługuje ją część serwerowa.

Wyróżnić można widoki ogólnodostępne, do których dostęp ma każdy, bez konieczności uwierzytelniania oraz widoki edytorskie. Aby skorzystać z tych drugich, należy uprzednio się zalogować na istniejące konto.

Uwierzytelnianie i autoryzacja po stronie serwerowej wykorzystują moduł *Passport.js* - dyskretny i prosty middleware służący do uwierzytelniania w środowisku *Node.js*. Owe narzędzie daje programiście dostęp do predefiniowanych strategii, których wykorzystanie znacząco ułatwia projektowanie mechanizmów autoryzacji oraz uwierzytelniania. Są to moduły, które można dostosowywać do własnych wymagań.

W ramach realizacji projektu wykorzystano dwie strategie - *lokalną* oraz *JWT*. Pierwsza, *lokalna* (*passport-local*), służy do uwierzytelniania. Weryfikuje poprawność danych logowania wprowadzonych w części klienckiej aplikacji przez użytkownika, w tym przypadku edytora (tylko edytorzy posiadają swoje konta). Następuje wyszukanie użytkownika po swoim identyfikatorze (e-mail) w bazie danych, a następnie sprawdzenie poprawności zahashowanego hasła. Druga strategia, *JWT* (*passport-jwt*), dokonuje autoryzacji na podstawie obecności i poprawności ciasteczka *fn_access_token* w wychodzącym zapytaniu. Na podstawie payloadu tokenu z ciasteczka identyfikowany jest autoryzowany użytkownik. Ciasteczko posiada flagę *HttpOnly*, dzięki czemu można się do niego dostać wyłącznie za pomocą protokołu *HTTP*. Oznacza to, że przy użyciu języka *JavaScript*, po stronie klienckiej, nie da się go odczytać, ani zmienić. Wpływa to znacząco na poprawę bezpieczeństwa aplikacji, gdyż w pewnym stopniu blokuje

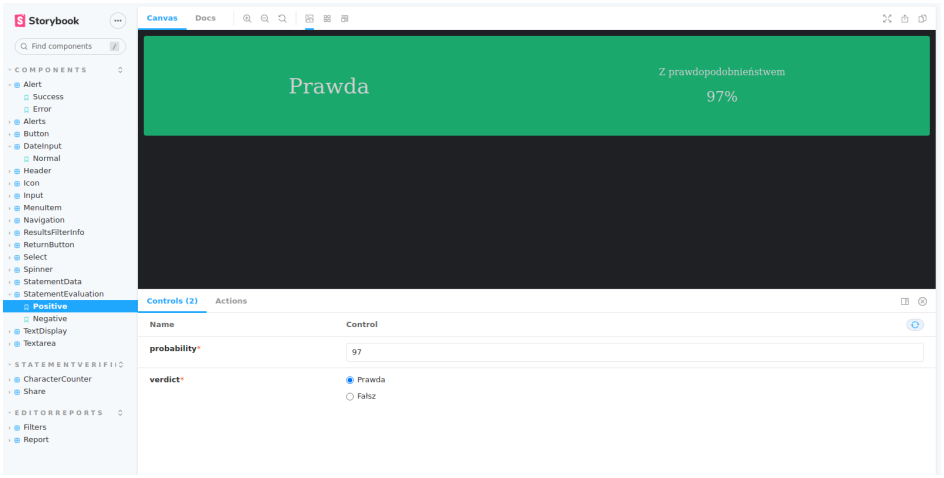
dostęp do potencjalnej kradzieży tożsamości. Służy to ochronie przed niektórymi atakami *XSS*. Jest to rozwiązanie znacznie bezpieczniejsze od przechowywania tokenu w *local-storage*.

4.4.4 Część kliencka aplikacji

Część kliencka aplikacji została stworzona w myśl wymagań jako Single Page Application. W implementacji wykorzystano bibliotekę React[14]. Proces implementacji aplikacji klienckiej został podzielony na 3 etapy:

1. implementacja komponentów składowych
2. zbudowanie z komponentów podstron aplikacji
3. integracja części klienckiej z częścią serwerową

Etap 1 wymagał dokładnej analizy projektu i określania elementów interfejsu użytkownika, które mogą być wielokrotnie używane, lub dają się rozdzielić od całości podstrony. Następnie, każdy z owych wydzielonych komponentów został zaimplementowany w wydzielonym środowisku Storybook[20]. Na rysunku 22 widoczny jest wygląd środowiska. Po lewej stronie, widoczne są istniejące komponenty. Wybierając dany komponent, widoczne są różne predefiniowane opcje jego wyglądu, w zależności od otrzymanych właściwości. Właściwości można modyfikować, i na ich podstawie, zmieniał się będzie komponent widoczny w oknie.



Rysunek 22: Środowisko Storybook

Rozwiązanie wymagało początkowo dużego wkładu pracy, jednak długofalowe korzyści, wynikające z jego użytkowania są przeważające. Mowa tu przede wszystkim o szybszym procesie rozbudowy aplikacji, przez dostęp do poglądu wyizolowanych dostępnych komponentów. Nieoceniona jest także możliwość dowolnej edycji właściwości obiektu z poziomu użytkownika. Pozwala to unaocznić działanie komponentu, bez konieczności pisania dodatkowego kodu.

Etap 2 polegał na zbudowaniu wszystkich podstron z przygotowanych wcześniej komponentów. Należało ustawić odpowiednie odstępy między sekcjami strony, a także zbudować odpowiedni dla aplikacji schemat nawigacji (za pomocą react-router[15]) i zarządzania stanem. Na tym etapie napisano obsługę wszystkich formularzy aplikacji. Wykorzystano tu bibliotekę formik[5], która redukuje w dużej mierze powtarzający się kod obsługi formularzy.

Etap 3 to przede wszystkim integracja z częścią serwerową. W związku z faktem, że posiadanie globalnego stanu w pewnych sytuacjach pozwala zaoszczędzić zasoby serwera, skorzystano z Redux[16]. Globalny stan pozwolił także na łatwe zarządzanie komunikatami informującymi o statusie operacji. Na tym etapie dokonała się integracja całości wcześniej wykonanej pracy, i zostały zastosowane poprawki np. w formie komunikatów zwracanych przez części serwerowe aplikacji, czy w modelach danych.

4.4.5 Captcha

Celem zwalczania nieporządanych zachowań, które mogłyby zagrozić stabilności serwera, wprowadzono mechanizm reCAPTCHA v3. Jest on obojętny dla użytkownika, co oznacza że działa pasywnie. Użytkownik nie odczuwa negatywnych skutków klasycznego mechanizmu captchy, takich jak np. irytacja towarzysząca wybraniu błędnego obrazka, bądź wpisaniu złej liczby i konieczności ponownej weryfikacji. Po stronie klienckiej generowany jest token, który następnie weryfikowany jest po stronie serwera przy pomocy odpowiedniego zapytania typu POST wysyłanego na serwery Google. W odpowiedzi wyodrębnić można pole *success*, które informuje, czy weryfikacja tokenu przebiegła prawidłowo, czy też nie. Na tej podstawie określa się, czy egzekucja wychodzącego zapytania powinna być kontynuowana, czy też zatrzymana.

4.5 Konteneryzacja projektu

W celu łatwego uruchomienia całości projektu wykorzystano narzędzie o nazwie *Docker*[3]. Jest to otwarte oprogramowanie służące do konteneryzacji oprogramowania, co pozwala na jego łatwe wdrażanie na różnych platformach sprzętowych, które korzystają z różnych wersji wymaganego w naszym projekcie oprogramowania np. z różnych wersji menedżera pakietów *npm*. Każdy z modułów, na które podzielić możemy finalne oprogramowanie otrzymał swój plik Dockerfile, na podstawie, którego wytwarzany jest obraz docker'owy. Obrazy są podstawami do tworzenia kontenerów, w których to, odizolowane od reszty hosta, działać będą moduły projektu. Aby uruchomić całość aplikacji wystarczy w katalogu głównym uruchomić komendę:

docker-compose up.

W ten sposób, projekt może zostać w łatwy sposób uruchomiony na dowolnej maszynie wspierającej narzędzia Docker i docker-compose.

Listing 7: docker-compose.yml

```
version: "3"

networks:
  isolation-network:
    driver: bridge

services:
  backend:
    container_name: python-server
    restart: always
    build:
      context: ./backend
      dockerfile: ./Dockerfile
    ports:
      - "8000:8000"
    networks:
      - isolation-network
  server:
    container_name: express-server
    restart: always
    build:
      context: ./server
      dockerfile: ./Dockerfile
    ports:
      - "3001:3001"
    networks:
      - isolation-network

  client:
    container_name: react-nginx
    restart: always
    build:
      context: ./client
      dockerfile: ./Dockerfile
    ports:
      - "80:80"
```

Spis rysunków

Spis listingów

Spis tabel

Bibliografia

[1] Marius Borcan. *TF-IDF Explained And Python Sklearn Implementation*. https://towardsdatascience.com/tf-idf-explained-and-python-sklearn-implementation-b020c5e83275. Czer. 2020. (Term. wiz. 15. 05. 2021).

[2] Tom Christie. *Uvicorn*. Wer. 0.13.4. 20 lut. 2021. URL: https://www.uvicorn.org.

[3] *Docker*. Wer. 20.10.02. 11 maj. 2021. URL: https://www.docker.com/.

[4] *express.js*. Wer. 4.17.1. 11 maj. 2021. URL: https://expressjs.com/.

[5] *Formik*. Wer. 2.2.8. 11 maj. 2021. URL: https://formik.org/.

[6] Sampath K. Gajawada. *Chi-Square Test for Feature Selection in Machine learning*. https://towardsdatascience.com/chi-square-test-for-feature-selection-in-machine-learning-206b1f0b8223/. Paź. 2019. (Term. wiz. 12. 11. 2020).

[7] P. Ksieniewicz i in. „Fake News Detection from Data Streams". W: *2020 International Joint Conference on Neural Networks (IJCNN)*. 2020, s. 1–8. DOI: 10.1109/IJCNN48605.2020.9207498.

[8] Qian Li i in. *A Survey on Text Classification: From Shallow to Deep Learning*. https://arxiv.org/pdf/2008.00364.pdf. Paź. 2020. (Term. wiz. 20. 03. 2020).

[9] *Mongodb*. Wer. 3.6.8. 11 maj. 2021. URL: https://www.mongodb.com/.

[10] F. Pedregosa i in. „Scikit-learn: Machine Learning in Python". W: *Journal of Machine Learning Research* 12 (2011), s. 2825–2830.

[11] *Portal Demagog.org.pl*. https://demagog.org.pl/fake_news/. (Term. wiz. 17. 03. 2021).

[12] Adam Radziszewski i Radosław Warzocha. *WCRFT2*. CLARIN-PL digital repository. 2014. URL: http://hdl.handle.net/11321/36.

[13] Sebastián Ramírez. *Fastapi*. Wer. 0.65.1. 11 maj. 2021. URL: https://fastapi.tiangolo.com.

[14] *React*. Wer. 17.0.2. 11 maj. 2021. URL: https://pl.reactjs.org/.

[15] *React router*. Wer. 5.2.0. 11 maj. 2021. URL: https://reactrouter.com/.

[16] *Redux*. Wer. 4.1.0. 11 maj. 2021. URL: https://redux.js.org/.

[17] Piotr Rychlik. *TermoPL*. CLARIN-PL digital repository. 2016. URL: http://hdl.handle.net/11321/266.

[18] Kelly Stahl. *Fake news detection in social media*. https://www.csustan.edu/sites/default/files/groups/University%20Honors%20Program/Journals/02_stahl.pdf. Kw. 2018. (Term. wiz. 21. 03. 2021).

[19] Kelly Stahl. *Scrapy | Fast and Powerful Scraping and Web Crawling Framework*. https://scrapy.org/.

[20] *Storybook*. Wer. 6.2. 11 maj. 2021. URL: https://storybook.js.org/.