POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Data Science

Inteligentne Systemy Przetwarzania Danych

Autorzy:

Lidia J. Opuchlik
Nr albumu s16478

Sandra Rawicz
Nr albumu s16536

Analiza sentymentu w recenzjach filmowych

Sentiment analysis in film reviews

Praca inżynierska

Promotor:
dr hab. Grzegorz M. Wójcik, prof. PJATK

Warszawa, wrzesień 2021

Wkład do pracy

Wkład do części pisemnej pracy inżynierskiej był następujący:

Cele i streszczenie — Sandra Rawicz
Rozdział 1 — Lidia J. Opuchlik
Rozdział 2 — Sandra Rawicz
Rozdział 3 — Lidia J. Opuchlik i Sandra Rawicz (po równo)
Rozdział 4 — Lidia J. Opuchlik

Opracowanie koncepcji oraz modelowanie było wykonywane w równym stopniu przez Lidię J. Opuchlik i Sandrę Rawicz.

Podpis promotora

.....

Spis treści

Spis rysunków

Spis tablic

Spis listingów

Wstęp

Cel

Celem ninejszej pracy było stworzenie i zbadanie przydatności klasycznych modeli machine learningowych oraz modeli wykorzystujących sieci neuronowe umożliwiających analizę sentymentu recenzji filmowych — klasyfikację czy dana recenzja ma pozytywny czy negatywny wydźwięk — z portalu IMDB (Internet Movie DataBase).

Słowa kluczowe

Przetwarzanie języka naturalnego, analiza sentymentu, recenzja filmowa, uczenie maszynowe, las losowy, maszyna wektorów nośnych, konwolucyjna sieć neuronowa, LSTM.

Streszczenie

W rozdziale pierwszym niniejszej pracy opisujemy najpierw zagadnienie sztucznej inteligencji — rys historyczny oraz podstawowe pojęcia. Następnie omawiamy pojedynczą dziedzinę sztucznej inteligencji — przetwarzanie języka naturalnego. Przybliżamy założenia i metody charakterystyczne dla tej dziedziny. Definiujemy też konstrukcje i metody wykorzystywane w zagadnieniach uczenia maszynowego, do których odwołują się kolejne rozdziały. W szczególności opisujemy problem klasyfikacji binarnej, który rozwiązujemy w ramach niniejszej pracy.

W drugim rozdziale pracy omawiamy wykorzystane przez nas algorytmy: las losowy, maszynę wektorów nośnych, konwolucyjną sieć neuronową, LSTM, oraz połączenie sieci konwolucyjnej z LSTM. Dla każdego algorytmu opisana jest jego konstrukcja i właściwości, a także szczegóły implementacji wykorzystanej w obrębie pracy.

W trzecim rozdziale pracy przedstawiamy eksperymenty aplikujące poszczególne przedstawione algorytmy do wybranego zbioru danych. Najpierw omawiamy przygotowanie zbioru danych. Następnie dla każdego z algorytmów prezentujemy implementację oraz omawiamy

sposób dobrania parametrów modelu. Pokazujemy też estymację przydatności poszczególnych algorytmów za pomocą wybranego zbioru metryk.

W czwartym, ostatnim rozdziale, znajduje się podsumowanie otrzymanych wyników i refleksje końcowe.

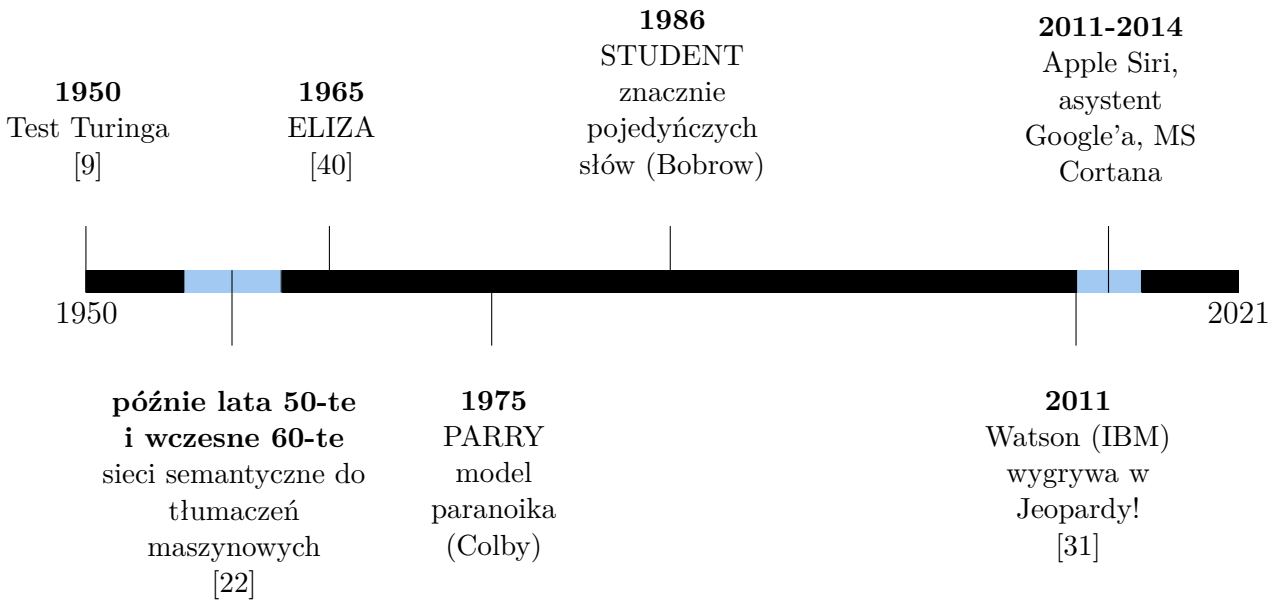
Rozdział 1

Sztuczna inteligencja

Koncepcja nowoczesnej sztucznej inteligencji ma swoje korzenie już w pracach klasycznych filozofów, którzy próbowali opisać proces ludzkiego myślenia jako mechaniczną manipulację symbolami. Wiele wieków później, w latach 40. XX wieku, zwieńczeniem tych prac było wynalezienie programowalnego komputera cyfrowego bazującego na abstrakcyjnym rozumowaniu matematycznym. Urządzenie to i stojące za nim pomysły zainspirowały naukowców do dyskusji na temat możliwości zbudowania „mózgu elektronicznego". Badania w tej dziedzinie znacznie przyspieszyły i zróżnicowały się (rozgałęziły) począwszy od lat 50-tych XX wieku, kiedy to Alan Turing — znany angielski matematyk, kryptolog — stworzył maszynę Turinga — abstrakcyjny model urządzenia służącego do wykonywania różnych algorytmów. Turing jest uważany za jednego z „ojców" informatyki i sztucznej inteligencji [38]. W ramach badań nad sztuczną inteligencją zaproponował on wykonanie testu zwanego Testem Turinga, który miał za zadanie sprawdzić czy „wypowiedzi" odpowiednio zaprogramowanej maszyny/algorytmu naśladującego sposób myślenia ludzkiego i posługującego się językiem naturalnym są odróżnialne od wypowiedzi prawdziwego człowieka. Test uznawany jest jako zaliczony, gdy sędzia (człowiek) konwersujący z kilkoma innymi obiektami (kilkoma ludźmi i maszyną) nie jest w stanie stwierdzić, czy w danej chwili rozmawia z człowiekiem, czy z maszyną symulującą człowieka. Od tego ważnego wydarzenia dynamika badań nad przetwarzaniem języka naturalnego (NLP) znacznie przyspieszyła. Ważniejsze kamienie milowe przedstawione są na osi czasu na następnej stronie. Obejmuje ona historię NLP od roku 1950 do czasów obecnych.

W ogólności sztuczna inteligencja jest definiowana jako zdolność systemu do poprawnej interpretacji danych zewnętrznych, uczenia się na podstawie takich danych oraz wykorzystywania tych informacji do osiągania konkretnych celów i zadań wykorzystując elastyczną adaptację [19]. Sztuczną inteligencję, ze względu na możliwości rozwiązywania przez nią problemów oraz umiejętność dostosowywania się do różnego rodzaju zadań, można

podzielić na 3 rodzaje: słabą, ogólną i superinteligencję.



Głównymi obszarami zastosowań sztucznej inteligencji są:

- uczenie maszynowe (automatyczne),
- rozpoznawanie i przetwarzanie obrazów — diagnostyka medyczna, tworzenie gier, śledzenie, tworzenie sztuki,
- **przetwarzanie języka naturalnego (NLP)**,
- robotyka i planowanie działań,
- tworzenie systemów eksperckich.

Obecnie świat napędzany jest nieustannie rosnącą ilością generowanych danych. W 2020 roku zostało wygenerowanych, prztworzonych, skopiowanych lub pobranych ponad 64 zettabajy danych. Jeden zettabajt danych, to 10^{21} bajtów. Odnosząc się do danych podawanych przez platformę © Statista, prognozowana ilość danych ma się mniej więcej potroić do końca 2025 roku [4]. Ze względu na tą ogromną ilość ważne jest aby stworzyć efektywne systemy umożliwiające jej wykorzstywanie.

Wg różnych źródeł [18], około 80% danych generowanych w obecnych czasach występuje w formie nieustrukturyzowanej, np. jako posty na Facebooku, tweety, opinie, recenzje wystawiane przez użytkowników na portalach internetowych, nagrania audio, wiadomości z Messengera czy Whatsappa oraz dane zbierane przez asystentów głosowych typu Alexa, Siri Cortana itd. Do danych nieustrukturyzowanych zaliczamy też filmy oraz zdjęcia i wszelką grafikę. Jednak ze względu na potrzeby tej pracy, skupimy się na danych tekstowych.

Dane nieustrukturyzowane niosą ze sobą mnóstwo użytecznych informacji. Jednak ze względu na nieodłączną złożoność ich przetwarzania i analizowania, praca z nimi stanowi wyzwanie, gdyż dane te trzeba najpierw skrupulatnie wydobyć z surowych zbiorów (nieobrobionych danych źródłowych) lub różnego rodzaju rejestratorów. Jednakże uważamy, że warto poświęcić na to czas, gdyż z biznesowego punktu widzenia nieustrukturyzowane dane stanowią potencjalną kopalnią złota.
Dla porównania, dane ustrukturyzowane czyli takie, które posiadają konkretną ściśle zdefiniowaną formę stanowią jedynie około 20% wszystkich generowanych danych, a więc zdecydowaną mniejszość. Jest to kolejny powód, dla którego warto sięgać też po dane nieustrukturyzowane, gdyż korzystanie tylko z tych ustrukturyzowanych danych powoduje, że dużo wartości umyka. Oba rodzaje wspomnianych danych wykazują istotne różnice. Poniżej wymienione są te najważniejsze.

Tablica 1.1: Porównanie danych ustrukturyzowanych i nieustrukturyzowanych.

Dane ustrukturyzowane	Dane nieustrukturyzowane
✦ o jasno zdefiniowanych typach, które można przeszukiwać wg określonych kryteriów	✦ zwykle przechowywane w ich natywnym formacie
✦ najczęściej ilościowe	✦ najczęściej jakościowe
✦ często przechowywane w hurtowniach danych	✦ często przechowywane w jeziorach danych
✦ można je łatwo przeszukiwać i analizować	✦ wymagają więcej pracy — wstępnej obróbki — do wydobycia i zrozumienia informacji, które niosą
✦ istnieją we wstępnie zdefiniowanych formatach	✦ występują w różnych formatach

Ze względu na cel niniejszej pracy, skupimy się na możliwościach przetwarzania i analizowania danych w postaci tekstu pisanego (recenzji filmów). Będziemy opierać się na koncepcji przetwarzania języka naturalnego.

1.1 Przetwarzanie języka naturalnego

Dane nie posiadające zdefiniowanej struktury — w szczególności te tekstowe i głosowe można analizować przy pomocy przetwarzania języka naturalnego. Wg Wikipedii: przetwarzanie języka naturalnego (*Natural Language Processing*, NLP) to interdyscyplinarna dziedzina, łącząca zagadnienia sztucznej inteligencji i językoznawstwa, zajmująca się automatyzacją języka naturalnego przez komputer, w tym analizą, rozumieniem, tłumaczeniem i generowaniem języka naturalnego [27].

1.1.1 Do czego wykorzystywane jest NLP?

Na pierwszy rzut oka można tego nie zauważyć, ale przetwarzanie języka naturalnego towarzyszy nam w wielu aspektach codziennego życia, i właściwie można stwierdzić, że bez jego obecności trudno byłoby nam funkcjonować.

Przetwarzanie języka naturalnego wykorzystywane jest:

1. Do analizy sentymentu — a poprzez poznawanie opinii użytkowników/klientów — dalej wykorzystywane jest np. w celu tworzenia spersonalizowanych rekomendacji i reklam, ogłoszeń. Może się to odbywać na podstawie analizy przeprowadzanych ankiet, wypełnianych formularzy, czy po prostu opinii i recenzji pisanych przez użytkowników Internetu,

2. Do automatyzacji różnych procesów:

    - kontaktów z klientami — doradztwo chatbotowe czy ekstrakcja i analiza informacji z nagrań reklamacji itp.,
    - biurowych — wyciąganie danych z faktur, dokumentów,
    - HR-owych — do usprawniania procesów rekrutacyjnych, np. analiza i wyciąganie pożądanych informacji z CV czy z portali społecznościowych. Przesłane przez aplikanta CV może nawet nie dojść do żywego rekrutera, bo zostanie już odrzucone na etapie wstępnym w związku z tym, że nie będzie posiadało pewnych elementów czy pożądanych informacji,

3. W urządzeniach Smart Home, np. asystenci głosowi tacy jak Siri, Alexa, Cortana, Google Assistant bazują na rozpoznawaniu i przetwarzaniu mowy,
4. Do tłumaczenia w czasie rzeczywistym (machine translation), np. google translate,
5. Do sprawdzania pisowni, stylistyki, słownictwa (synonimy), np. Grammarly,
6. W programowaniu — wszystkie narzędzia intellisense oparte są o NLP, np. Kite,
7. Do wyszukiwania po słowach kluczowych (np. google search zwraca bardzo trafne wyniki wyszukiwania).

Przetwarzanie języka naturalnego można podzielić na dwie główne gałęzie:

1. Rozumienie języka naturalnego,
2. Tworzenie języka naturalnego.

Gałąź pierwsza obejmuje wydobywanie informacji z tekstu (*text mining*) i analizę testu (*text analytics*) [25]. Na analizę składają się: mapowanie surowych danych z tekstów lub nagrań wyrażonych przy pomocy języka naturalnego w ich użyteczną reprezentację, czyli przekształcanie ich w wartościowe informacje, a ponadto nadawanie im odpowiedniej struktury i analiza pewnych cech i wzorców wydobywanych z tych tekstów oraz analiza aspektów językowych. Natomiast, gałąź druga polega na wytwarzaniu fraz i zdań oraz dłuższych form z wewnętrznych reprezentacji mających znaczenie w formie języka naturalnego [20]. Gałąź pierwsza jest dużo bardziej pracochłonna i dużo więcej aspektów trzeba zrozumieć żeby wykorzystać to zastosowanie. Te dwa procesy mogą być ze sobą ściśle powiązane.

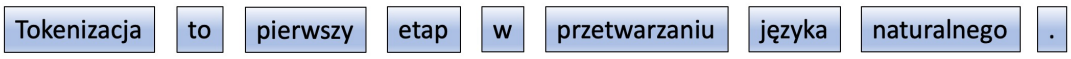
Ogólnym celem NLP jest praca z danymi w języku naturalnym, która umożliwia tworzenie modeli, wyciąganie wniosków w celu produkowania wartości biznesowej. Jednak żeby korzystać z tego typu nieustrukturyzowanych danych trzeba być zaznajomionym z technikami analizy tekstu i przetwarzania języka naturalnego oraz posiadać do tego spory wachlarz odpowiednich narzędzi.

1.1.2 Operacje na danych tekstowych

Do podstawowych operacji wykonywanych na danych tekstowych należą m. in. tokenizacja, normalizacja, usunięcie niewnoszących do znaczenia tzw. słów-stopów (*stop-words*), stemming, lematyzacja, tagowanie/oznaczanie/rozpoznawanie części mowy (w tym tworzenie N-gramów), rozpoznawanie bytów nazwanych (nazw własnych), parsowanie tekstu — chunking (parsowanie płytkie) i parsing (właściwe parsowanie). Należy zaznaczyć, że wykonywanie poszczególnych operacji jest opcjonalne i zależy od zamysłu programisty i użyteczności samej operacji. Najpopularniejszą biblioteką umożliwiającą te operacje jest biblioteka `NLTK` (*Natural Language ToolKi*t). Oprócz funkcjonalności wymienionych wcześniej, biblioteka ta zapewnia również wiele innych bardziej zaawansowanych, nieco rzadziej używanych funkcjonalności.

1.1.2.1 Tokenizacja

Jest to pierwszy etap przetwarzania tekstu polegający na rozbijaniu tekstu na mniejsze struktury — tzw. tokeny (żetony), np. zdania, wyrażenia, słowa. Należy tu wyraźnie podkreslić, że znaki przystankowe (kropka, przecinek itd.) też są tokenami. Przykładowe zdanie — *Tokenizacja to pierwszy etap w przetwarzaniu języka naturalnego.* — po wykonaniu tokenizacji zostanie rozbite na następujące tokeny:



Rysunek 1.1: Zdanie — Tokenizacja to pierwszy etap w przetwarzaniu języka naturalnego. — po wykonaniu tokenizacji.

Wygenerowane zostaje tu dziewięć tokenów:
→ każdy wyraz jest tokenem,
→ kropka również jest tokenem.

1.1.2.2 Normalizacja

Normalizacja to wykonanie procesów przekształcenia danych tekstowych, które czasami towarzyszą tokenizacji. Normalizacja, inaczej ujednolicenie, może polegać na sprowadzeniu różnych form słów mających podobne (to samo) znaczenie do wspólnej postaci, np. liczebniki znaczące to samo, ale odmienione przez rodzaje — *jeden*, *jedna*, *jedno* — należy sprowadzić do postaci *1*, gdyż mają to samo znaczenie. Pominięcie tej procedury skutkowałoby tym, że wyrazy te traktowane by były jako trzy różne słowa

mimo, że semantycznie znaczą to samo. Innym zabiegiem normalizacyjnym stosowanym w celu ułatwienia pracy z tekstem jest zamiana wszystkich znaków na znaki tej samej wielkości [11] — na wielkie lub małe litery itp.

1.1.2.3 Usuwanie słów-stopów

Słowa-stopy są to słowa nic nie wnoszące do znaczenia i zrozumienia procesowanych danych (uznawane są za szum). Słowa te (w danym języku) są bardzo powszechnie używane i pojawiają sie właściwie w każdym tekście uniemożliwiając różnicowanie. Wyeliminowanie zbioru tych słów pozwala skupić się na wartościowych treściach (kluczowych słowach umożliwiających skuteczną analizę) [12].
Do słów-stopów zalicza się rodzajniki, spójniki, przyimki itp. Najczęściej są to słowa o najwyższej częstotliwości występowania w tekście. Usuwanie można robić na podstawie dostępnych baz predefiniowanych dla danego języka lub tworzyć własne listy słów-stopów. Często tworzy się dedykowane listy słów-stopów w zależności od domeny, w której osadzony jest dany tekst. Np., żeby usprawnić analizę tekstu medycznego można usunąć takie słowa jak dr, pacjent itd., żeby analizować wypowiedzi z Twittera można usunąć hashtagi, czyli słowa zaczynające się na #, nazwy użytkowników — zaczynające się znakiem @ itp. W ogólności powinno się usuwać wszystkie słowa i frazy, które mają małą moc dyskryminującą i takie, które prowadzą do nieprawidłowości w otrzymywanych wynikach.

1.1.2.4 Inne przekształcenia

Oprócz usuwania słów-stopów można również wykonać przekształcenia dające podobny efekt np. usuwanie akcentów, znaków specjalnych, aposrtrofów, cyfr, czy czegokolwiek niepotrzebnego lub niewnoszącego do zrozumienia niesionej informacji bazując na odpowiednich wyrażeniach regularnych. Przykładem może być usuwanie tagów `HTML`, np. po pobieraniu surowych danych ze strony internetowej (*webscraping*). Przekształcenia te mogą również dotyczyć modyfikacji posiadanych danych. W języku angielskim popularne jest używanie form skrótowych *y'all, I'd, I'll*. Należy je zamienić na *you all*, *I would* oraz na *I will* itd.

1.1.2.5 Tworzenie N-gramów

Tworzenie N-gramów, np. bigramów, trigramów etc. — polega na tworzeniu połączeń słów, które mogą występować razem. Najpierw się tokenizuje zdanie do słów, a następnie z powstałych tokenów tworzy się N-gramy. Często zdarza się, że połączenia słów nabierają nowego, innego znaczenia.

Przykład:
Nowy Jork razem ma inne znaczenie niż oddzielne słowa *Nowy* i *Jork*.

1.1.2.6 Stemming

Stemming to proces usunięcia ze słowa końcówki fleksyjnej (przedrostka, przyrostka) pozostawiając tylko temat wyrazu (nieodmienny rdzeń). Może być przeprowadzany w celu zmierzenia popularności danego słowa. Różne rodzaje stemmerów — mniej i bardziej agresywne, np. `PorterStemmer` (łagodny), `LancasterStemmer` (bardziej agresywny).
Wynikiem stemmingu może być ciąg znaków, który sam w sobie nic nie znaczy, ale jest wspólny dla wszystkich słów znaczących to samo w danym tekście jak też normalnym istniejącym i funkcjonującym słowem.
Np. angielskie słowa: *connection*, *connections*, *connective*, *connected*, *connecting* poddane stemmingowi dadzą ten sam wynik, czyli słowo *connect*.

1.1.2.7 Lematyzacja

Lematyzacja to sprowadzenie słowa do jego podstawowej postaci (bazowej), czyli analiza morfologiczna. Do wykonania tego zadania potrzebny jest słownik lub rozbudowany zestaw reguł. W przypadku czasownika będzie to bezokolicznik, w przypadku rzeczownika — mianownik liczby pojedynczej. W odróżnieniu od stemmingu — forma bazowa powstała po lematyzacji zawsze jest istniejącym słowem. Warto tutaj dodać, że lematyzacja jest dużo trudniejsza dla silnie fleksyjnych języków — takich, które posiadają dużo końcówek dla różnych osób, liczb, przypadków, jak np. język polski. Jest to dużo łatwiejsze w przypadku języka angielskiego.

1.1.2.8 Rozpoznawanie części mowy

Rozpoznawanie części mowy zwane inaczej tagowaniem gramatycznym — jest to oznaczanie części mowy (POS — *Part Of Speech*), czyli rozpoznawanie i etykietowanie czy dane słowo jest rzeczownikiem, czasownikiem, przymiotnikiem, przysłówkiem, liczebnikiem itp. Wykonuje się to albo w oparciu o słownik albo wykorzystując występujące końcówki fleksyjne. Może się zdarzyć, że słowo, w zależności od kontekstu, w którym się znajduje, będzie przynależało do różnego typu części mowy .

Poniższy przykład dobrze obrazuje wspomnianą niejednoznaczność:
→ *message me* (jako czasownik) niesie znaczenie daj mi znać lub powiadom mnie,

natomiast
→ *a message* (jako rzeczownik) oznacza po prostu wiadomość.

1.1.2.9 Rozpoznawanie bytów nazwanych

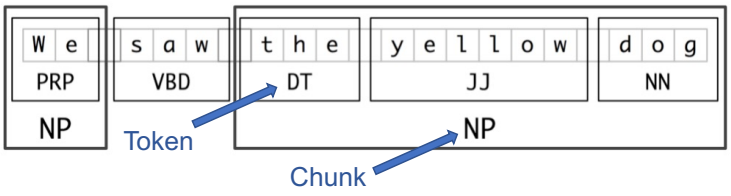
Kolejna operacja na tekście to rozpoznawanie bytów nazwanych. Jest to nic innego jak identyfikowanie/odnajdywanie nazw własnych, które w odróżnieniu od rzeczowników pospolitych, nie są łatwe do rozpoznania, np. lokalizacje, nazwy budynków, dane personalne, waluty, jednostki itp.

1.1.2.10 Parsowanie

Parsowanie to zbieranie indywidualnych „oczyszczonych" kawałków tekstu i grupowanie ich w większe wyrażenia. Z grubsza, parsowanie można podzielić na dwa rodzaje: parsowanie płytkie (*shallow-parsing*) oraz parsowanie głębokie (*deep-parsing*).

Parsowanie płytkie Parsowanie płytkie, czasami określane terminem kawałkowanie (*chunking*), ma na celu wydobycie istotnej informacji, bez potrzeby wyciągania wszystkich zależności składniowych pomiędzy słowami. Kawałkowanie bierze na wejściu pojedyncze części mowy i zwraca na wyjściu grupy wyrazów/ wyrażenia (*chunki*). Istnieje 5 głównych rodzajów wyrażeń (grup/*chunków*):

- wyrażenia rzeczownikowe (NP),
- wyrażenia czasownikowe (VP),
- wyrażenia przymiotnikowe (ADJP),
- wyrażenia przysłówkowe (ADVP),
- wyrażenia przyimkowe (PP).



Rysunek 1.2: Parsowanie płytkie.

Parsowanie głębokie Parsowanie głębokie polega na pełnym odtworzeniu drzewa składniowego zdania.

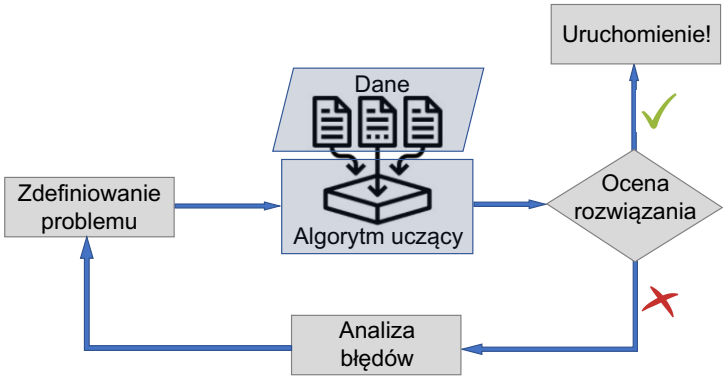
Wszystkie wyżej opisane zabiegi stosuje się po to żeby jak najbardziej uprościć budowany model (zmniejszyc jego rozmiary). Jednakże żaden algorytm nie może używać bezpośrednio słów żeby wykoywać modelowanie. Aby budować modele matematyczne, należy przekształcić słowa w wartości liczbowe (w NLP w wektory liczb). Gdy dojdzie się do tego punktu, tak naprawdę zabawa się dopiero zaczyna. Można zrobić takie rzeczy, jak wyznaczanie częstotliwości słów w tekście, mierzenie odległości między słowami, wyznaczanie wielu innych przydatnych zależności i parametrów oraz tworzenie modeli klasyfikujących, klasterujących i innych. Istnieje też wiele bibliotek, przy pomocy których można również tworzyć bardzo efektowne wizualizacje.

1.2 Podstawy modelowania

1.2.1 Definicje

1.2.1.1 Uczenie maszynowe

Uczeniem się systemu jest każda autonomiczna zmiana w systemie zachodząca na podstawie doświadczeń, która prowadzi do poprawy jakości jego działania [7].
Uczenie maszynowe to dziedzina nauki dająca komputerom możliwość uczenia się bez konieczności ich jawnego programowania [33]. Mówiąc bardziej technicznie uczenie maszynowe polega na tym, że program komputerowy uczy się na podstawie doświadczenia (E) w odniesieniu do jakiegoś zadania (T) i pewnej miary wydajności (P), jeżeli jego wydajność wzrasta wraz z nabywaniem doświadczenia [26].



Rysunek 1.3: Proces uczenia maszynowego.

Uczenie maszynowe można podzielić na trzy główne typy:

1. Uczenie nadzorowane (*supervised learning*) — maszyna uczy się generalizować na podstawie oznakowanych danych. W pierwszym etapie nastepuje trenowanie modelu: model jest trenowany na podstawie macierzy cech (*features, atributes*) i odpowiadającym wartościom na wyjściu (*label*). Polega to na dopasowaniu funkcji jak najlepiej odzwierciedlającej dane (generalizującej). Następnie wyznaczony model używa się w predykcji — na wejściu podaje się macierz z nowymi wartościami cech i wytrenowany model wnioskuje wartość wyjścia (dyskretną — klasę lub ciągłą — liczbę) wykorzystując wyuczoną funkcję.

2. Uczenie nienadzorowane (*unsupervised learning*) — ma miejsce, gdy informacja trenująca jest niedostępna (brak przynależności do klas wynikowych). De facto używany algorytm sam wyznacza klasy — znajduje opisy dla tych klas (reguły podziału: jakie klasy są obecne i co je różnicuje) — i dzieli obserwacje pomiędzy te klasy wg własnego uznania. Proces ten jest często określany mianem grupowania (*clustering*). Należy tu podkreślić, że w zależności od użytego algorytmu, klasy mogą być zdefiniowane w różny sposób — może istnieć wiele różnych sposobów dzielenia obserwacji na klasy i opisywania każdej klasy, dlatego rodzaj użytego algorytmu ma tu kluczowe znaczenie.

3. Uczenie wspomagane (*reinforcement learning*).

1.2.1.2 Model predykcyjny

Model predykcyjny opisuje zależności między zmiennymi objaśniającymi (cechami) a zmienną wynikową (targetem) [3]. Umożliwia on, w oparciu o zmienne objaśniające, domniemać jaka jest wartość targetu. Istnieje wiele rodzajów modeli. Przykładami są regresja logistyczna, regresja liniowa, drzewo decyzyjne.

Modele można podzielić ze względu na ich przeznaczenie na:

- klasyfikujące — target dyskretny (np. drzewa decyzyjne, regresja logistyczna),
- aproksymujące — target ciągły (np. regresja liniowa, sieci neuronowe),
- asocjujące — współwystępowanie wartości (np. algorytm A-Priori, sieci asocjacyjne),
- segmentujące — podział na segmenty (np. algorytm k-means, sieci Kohonena).

1.2.1.3 Klasa

Klasa — inaczej target lub zmienna wynikowa lub etykieta — jest to wartość docelowa wyznaczana przez model predykcyjny, która ma formę dyskretną (nieciągłą).

1.2.1.4 Klasyfikacja binarna

Klasyfikacja binarna — inaczej regresja logistyczna, to przykład modelu klasyfikacyjnego, w którym na podstawie danych objaśniających nadawane są przynależności do jednej z **dwóch** możliwych dyskretnych klas wynikowych. Przykładowo na podstawie pewnych określonych charakterystycznych objawów można zakwalifikować pacjenta jako zdrowego lub chorego.

1.2.1.5 Algorytm trenujący

Model może zostać wytrenowany przy pomocy różnych algorytmów trenujących. Algorytmy te mogą różnić się, np. hiperparametrami i ich wartościami oraz dochodzić do wyniku w nieco różny sposób.

1.2.1.6 Próbka

Jest to każdy „element" wykorzystywany do uczenia modelu, tzw. przykład uczący — pojedyńcza kombinacja cech (atrybutów) wraz z etykietą odpowiadającą tym cechom [15]. Mówiąc kolokwialnie, jest to jeden wiersz/ rekord/ obserwacja/ obiekt w `Dataframie`.

1.2.1.7 Atrybut

Atrybut — inaczej cecha (*feature*) — jest to jeden z aspektów danej próbki. Pojedyńczy atrybut reprezentowany jest jako pojedyńcza kolumna w `Dataframie`. Atrybuty mogą być proste (niepodzielne/ atomowe) lub złożone (można je podzielić na „mniejsze" atrybuty, np. adres zawierający miasto, ulicę, numer, kod pocztowy można rozbić na składowe), kategoryczne lub numeryczne. Atrybuty można również podzielić na niezależne (cechy wykorzystywane do wykonania predykcji) oraz zależne (wyniki predykcji — tzw. etykiety) [3].

1.2.1.8 Zbiór treningowy

Zbiór treningowy (*training set*) — jest to zbiór danych, który używany jest do uczenia się modelu (algorytmu). Na podstawie tych danych model uczy się odpowiednio klasyfikować lub aproksymować wartości wyjścia oraz buduje wszelkie zależności.

Inaczej mówiąc, model uczy się przewidywać możliwe wyniki i podejmować decyzje na podstawie przekazanych mu danych [15].

1.2.1.9 Zbiór testowy

Zbiór testowy (*testing set*) — jest to zbiór danych, którego używa się do przetestowania wcześniej wytrenowanego na danych treningowych modelu. Należy podkreslić, że dane zawarte w zbiorze testowym nie mogą być używane wcześniej do nauki modelu (dane, których model nigdy wcześniej nie widział), ponieważ nie będą się wtedy nadawać do obiektywnej oceny działania/skuteczności modelu. Zbioru tego można również użyć w procesie dobierania odpowiedniego zetawu hiperparametrów dla danego modelu [34].

Aby podzielić wyjściowy zbiór danych na treningowy i testowy można użyć popularnej metody `train_test_split` z pakietu `sklearn.model_selection`.

1.2.1.10 Hiperparametry

Zazwyczaj modele w uczeniu maszynowym posiadają parametry. Niektóre mają ich więcej drugie mniej — w zależności od złożoności danego modelu. Gdy wartość parametru obliczana jest samodzielnie przez algorytm podczas procesu trenowania to parametr taki nazywamy po prostu zwykłym parametrem, natomiast jeżeli wartość parametru zostaje podana przez modelarza, który danego algorytmu używa, to taki specjalny parametr nosi miano hiperparametru [21]. Przykładem hiperparametrów (zewnętrznych parametrów) są, np. głębokość lasu losowego (liczba drzew) czy liczba epok treningowych, natomiast zwykłymi (wewnętrznymi) parametrami mogą być wagi w sieciach neuronowych, czy współczynniki w regresji liniowej.

1.2.1.11 Funkcja kosztu (straty)

Tworzony model, czyli de facto funkcja dopasowywana do danych wejściowych modelu zwana jest powszechnie hipotezą. Dokładność funkcji hipotezy (w stosunku do zbioru uczącego) można zmierzyć za pomocą funkcji kosztu. Przykładem funkcji kosztu może być średni kwadratowy błąd (*mean squared error*). W dużym skrócie, funkcja kosztu zwraca średnią różnicę między wynikami funkcji hipotezy a rzeczywistymi danymi wyjściowymi dla każdego wejścia. Celem jest minimalizacja tej funkcji kosztu, czyli znalezienie odpowiednich optymalnych parametrów. Im mniejsza wartość funkcji kosztu, tym dokładniejsza jest funkcja hipotezy, czyli lepiej dopasowany jest nasz model [28].

1.2.1.12 Błąd

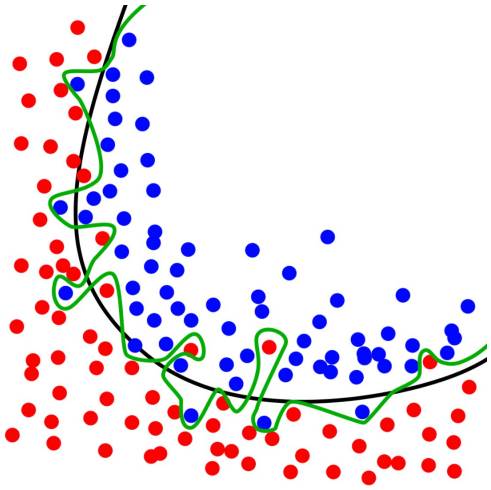
Błąd jest to różnica pomiędzy wartością otrzymaną a wartością rzeczywistą (prawdziwą) dla danej obserwacji. Technicznie, błąd można wyznaczać na wiele różnych sposobów. Najczęściej błąd podaje się jako uśrednioną wartość błędów obliczoną dla całego zbioru danych.

1.2.1.13 Walidacja — metryki sukcesu (*model evaluation metrics*)

Walidacja jest to proces oceny przydatności danego modelu. Do walidacji działania stworzonego modelu stosuje się różne metryki sukcesu. Sa to takie wielkości, które mówią jak dobrze dany model nauczył się generalizować, a co za tym idzie pozwalają nam określić przydatność danego modelu do wykonywania określonego zadania. W problemach klasyfikacyjnych często do oceny używa się macierzy omyłek, która posiada informacje o prawidłowo i nieprawidłowo sklasyfikowanych przykładach w odniesieniu do prawdziwych wartości dla danych przykładów. W oparciu o nią można później obliczyć różne metryki sukcesu. Bardzo popularną metryką jest precyzja (*accuracy*) — obliczana jako liczba dobrze sklasyfikowanych przykładów do liczby wszystkich przykładów.

1.2.1.14 Przeuczenie (*overfitting*)

Przeuczenie (*overfitting*) polega na tym, że model przesadnie dopasowuje się do konkretnych danych — tych, na podstawie których został zbudowany. Dla tych konkretnych danych zwraca on wyniki obdarzone bardzo małym błędem, natomiast nie umie on dobrze generalizować, a co za tym idzie, jeżeli nakarmimy model nowymi danymi otrzymamy znacznie gorsze wyniki. Przeuczenie następuje, gdy model statystyczny ma zbyt dużo parametrów w stosunku do rozmiaru próby, na podstawie której był konstruowany [41].



Rysunek 1.4: Przeuczenie (overfitting).

Zielona linia reprezentuje model przesadnie dopasowany (przeuczony), a czarna linia reprezentuje model uregulowany. Chociaż zielona linia najlepiej podąża za danymi treningowymi, jest zbyt zależna od tych danych i prawdopodobnie będzie miała wyższy poziom błędów w przypadku nowych danych w porównaniu z czarną linią. Czarna linia zdecydowanie lepiej generalizuje.

1.2.1.15 Wektoryzacja tekstu

Większość istniejących algorytmów trenujących modele predykcyjne wymaga na wejściu wektorów liczbowych. Istnieje wiele technik pozwalających zamienić tekst na wektor liczb rzeczywistych. Najprostszą techniką jest użycie wektoryzacji zliczeniowej (*Count Vectorization).* Wg dokumentacji `ScikitLearn`, wektoryzacja ta polega na konwersji kolekcji dokumentów tekstowych na macierz zawierającą zliczenia poszczególnych tokenów [10]. Mówiąc prościej, polega na zliczaniu słów w danym tekście (korpusie) i w rezultacie umożliwia obliczenie częstości występowania danego słowa w analizowanym tekście. Zaimplementowana jest ona w pakiecie `sklearn.feature_extract- ion.text.CountVecto`
Rozkład częstości słów można wizualizować przy pomocy wykresów używając metody `FreqDistVisualizer` z pakietu `yellowbrick.text`.

Rozdział 2

Narzędzia i algorytmy

W niniejszym rozdziale opisane są wybrane algorytmy służące do klasyfikacji binarnej — las losowy, maszyna wektorów nośnych, sieci konwolucyjne oraz rekurencyjna sieć LSTM (długoterminowa pamięć krótkoterminowa). Dla każdego algorytmu opisana jest jego konstrukcja i właściwości, a także szczegóły implementacji na potrzeby niniejszej pracy.

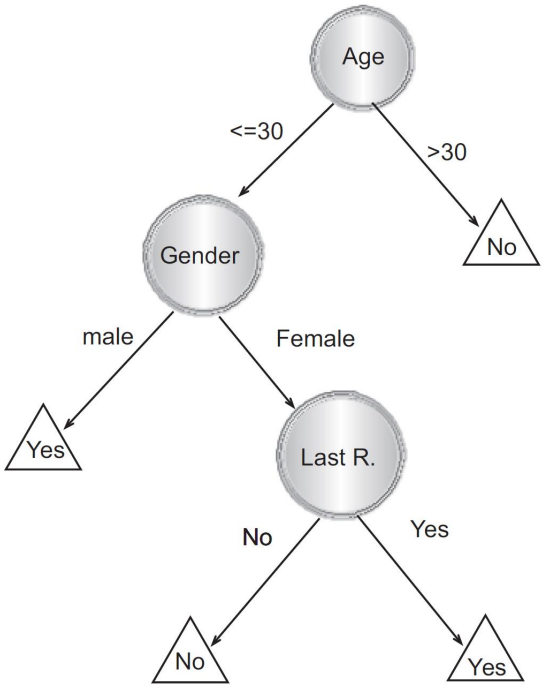
2.1 Las losowy

Las losowy to szeroko wykorzystywana metoda uczenia maszynowego, mogąca posłużyć do rozwiązania problemu klasyfikacji. Polega ona na wytrenowaniu wielu drzew decyzyjnych na losowych podzbiorach danych wejściowych. W celu uzyskania predykcji, sprawdzane są predykcje tak uzyskanych drzew — predykcja lasu to klasa wybrana przez większość drzew w lesie. Las losowy ma lepsze możliwości dostosowania się do danych wejściowych niż prostsze modele (np. liniowe), ale jest bardziej odporny na przetrenowanie (*overfitting*) niż drzewo decyzyjne. W tej sekcji opisana jest metoda drzewa decyzyjnego, następnie konstrukcja lasu losowego, potem omówione są kluczowe właściwości lasu losowego, zaś na końcu szczegóły implementacji algorytmu na potrzeby niniejszej pracy.

2.1.1 Drzewo decyzyjne

Drzewo decyzyjne to jedna z podstawowych metod wykorzystywana w data miningu dla klasyfikacji [32]. Polega ona na zbudowaniu modelu predykcyjnego, który ma postać ukorzenionego drzewa. Każdy wewnętrzny węzeł drzewa jest oznaczony którymś z atrybutów z danych wejściowych. Każda z krawędzi wychodzących z węzła do potomka jest oznaczona zakresem wartości tego atrybutu. Każdy liść drzewa jest oznaczony jedną z klas, które model ma przewidywać. Predykcja za pomocą tego

modelu polega na przejściu po drzewie od korzenia aż do któregoś z liści, po drodze wybierając odpowiednie krawędzie w zależności od wartości odpowiednich atrybutów próbki wejściowej.



Rysunek 2.1: Drzewo decyzyjne [32]

Budowa drzewa decyzyjnego odbywa się typowo za pomocą algorytmu zachłannego. W danym wierzchołku wybierany jest ten atrybut oraz takie zakresy, które w wyniku podzielenia zbioru wejściowego stworzą podzbiory możliwie najbardziej jednorodne ze względu na klasy w nich zawarte. Ściślej, jako metrykę jakości danego podziału stosuje się zwykle miarę *Information Gain* bądź *Gini Impurity* — wybór konkretnej miary jest hiperparametrem algorytmu uczącego, jednakże publikacje wskazują na to, że obie miary dają podobne wyniki [39]. Utworzone podzbiory przekazywane są rekurencyjnie w głąb drzewa w celu utworzenia kolejnych potomków. Procedura trwa do momentu gdy w każdym wierzchołku są już próbki z tylko jednej klasy, lub gdy zostanie osiągnięta maksymalna głębokość drzewa — hiperparametr algorytmu. Jak widać z konstrukcji modelu, drzewo decyzyjne ma bardzo wysoką interpretowalność — łatwo zrozumieć w jaki sposób obliczona została dana predykcja. Drzewo decyzyjne charakteryzuje się też tym, że ma możliwość idealnego dopasowania się do danych treningowych. Często skutkuje to przeuczeniem się (*overfitting*) i słabymi wynikami na danych testowych.

2.1.2 Konstrukcja lasu losowego

Las losowy budowany jest poprzez stworzenie pewnej liczby drzew decyzyjnych — liczba drzew jest hiperparametrem algorytmu. Każde z drzew jest budowane na podstawie losowo stworzonego podzbioru zbioru treningowego. Ściślej, używana jest technika o nazwie *bagging* (od ang. *Bootstrap aggregating*) [5] — dla zbioru wejściowego o rozmiarze n, dla każdego drzewa losowane jest ze zwracaniem n próbek ze zbioru. Dodatkowo, podczas budowy pojedynczych drzew, przy wybieraniu atrybutu do podziału w węźle drzewa, każdorazowo przeglądany jest jedynie losowy podzbiór atrybutów. Typowo losuje się $\sqrt{p}$ gdzie p to liczba atrybutów.
Tak zbudowany las wykorzystuje się do predykcji w następujący sposób — dla danej próbki oblicza się predykcję z każdego z drzew, a następnie sprawdza się która z klas została przewidziana najczęściej.

2.1.3 Właściwości

Las losowy ma podobną siłę wyrazu jak drzewo decyzyjne. Jest jednak dużo bardziej odporny na przeuczenie — innymi słowy ma mniejszą wariancję ale podobne obciążenie (*bias*). Dzieje się tak dzięki temu, że drzewa są trenowane niezależnie — czyni to model dużo mniej czułym na szum w zbiorze treningowym. Używanie podzbioru atrybutów podczas budowy węzłów ma za zadanie dodatkowego zmniejszenia korelacji między drzewami (w przeciwnym przypadku wszystkie stworzone drzewa będą preferowały atrybuty będące najsilniejszymi predyktorami, co sprawi, że będą one podobne) [16].
Las losowy jest też używany dla oszacowania ważności poszczególnych atrybutów — podczas konstrukcji drzew, można zaobserwować jaka była miara jednorodności atrybutów użytych w węzłach. Tak stworzony ranking ważności atrybutów może pomóc w zinterpretowaniu zbioru danych, lub w wyborze najważniejszych atrybutów do innego algorytmu.

2.1.4 Implementacja

W ramach niniejszej pracy użyłyśmy implementacji lasu losowego dla języka Python zawartej w klasie `sklearn.ensemble.RandomForestClassifier` z biblioteki `scikit-learn`. Pozwala ona na zdefiniowanie kluczowych hiperparametrów oraz na ekstrakcję ważności atrybutów.

2.1.4.1 Wybór hiperparametrów

Do znalezienia hiperparametrów modelu (liczba drzew, maksymalna głębokość pojedynczego drzewa) użyłyśmy klasy `RandomizedSearchCV` z biblioteki `sklearn` z pakietu `model_selection`, która pozwala na przeszukiwaniu przestrzeni hiperparametrów na zdefiniowanym przez nas zakresie w efektywny sposób przy użyciu techniki walidacji krzyżowej (*cross-validation*). Technika ta polega na wydzieleniu części danych treningowych, wytrenowaniu na nich modelu a następnie sprawdzeniu wyników predykcji na pozostałej części danych treningowych (zbiorze walidacyjnym).

Użyta przez nas metoda używa techniki *5-fold cross validation*, która polega na podzieleniu danych treningowych na 5 części, a następnie pięciokrotnym powtórzeniu procedury walidacji krzyżowej (tak, aby każda z 5 części raz została użyta jako zbiór walidacyjny). Wyniki wszystkich eksperymentów są zapisywane w klasie wraz z ich najistotniejszymi miarami statystycznymi (średnia wyników, odchylenie standardowe, średni czas trenowania modelu) oraz rankingowane.
`RandomizedSearchCV` jest przydatnym narzędziem, gdy chcemy przeszukać duży zbiór hiperparametrów i czas trenowania modelu jest długi. Nie sprawdza ona wszelkich możliwych kombinacji, tylko trenuje model na losowo wybranych hiperparametrach z podanych przez nas przedziałów, rozkładów lub zbiorów.

2.2 Maszyna wektorów nośnych

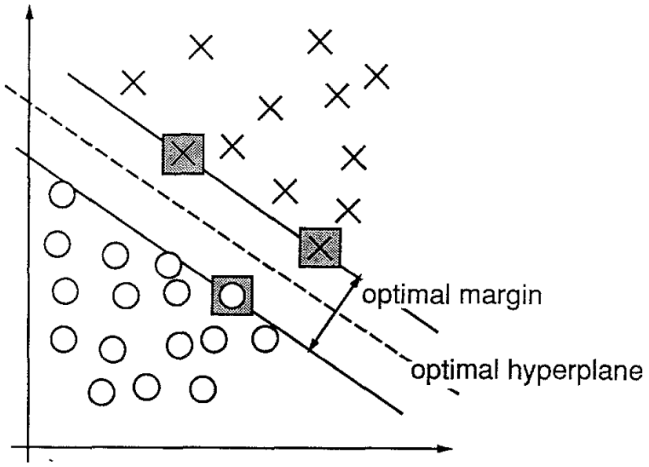
Maszyna wektorów nośnych (*Support Vector Machine*, SVM) [8], jest to jeden z klasycznych algorytmów klasyfikacji binarnej. Opiera się on na znalezieniu hiperpłaszczyzny rozdzielającej przestrzeń danych wejściowych na dwie klasy, w taki sposób by odległość punktów danych od hiperpłaszczyzny była jak największa.

W tej sekcji opisana jest konstrukcja maszyny wektorów nośnych, następnie opisane są właściwości modelu, a na końcu szczegóły implementacji algorytmu na potrzeby niniejszej pracy.

2.2.1 Konstrukcja

Dane wejściowe o p atrybutach można traktować jako zbiór punktów w p-wymiarowej przestrzeni — każdy o przypisanej klasie. Maszyna wektorów nośnych to metoda znalezienia takiej $(p-1)$-wymiarowej płaszczyzny, która będzie rozdzielać punkty z jednej klasy od punktów z drugiej klasy. W przypadku gdy istnieje wiele takich płaszczyzn, wybierana jest taka, która maksymalizuje odległość najbliższych jej punktów.

W przypadku gdy nie istnieje taka płaszczyzna, wybierana jest taka, która minimalizuje dodatkowo odległość od niej błędnie sklasyfikowanych punktów. To w jakim stosunku optymalizować te dwa cele jest hiperparametrem algorytmu (dalej oznaczony λ).



Rysunek 2.2: Przykład rozdzielonych punktów w 2-wymiarowej przestrzeni — wektory nośne oznaczone są kwadratami [8]

Wyraźmy hiperpłaszczyznę za pomocą jej wektora normalnego w oraz liczby b:

$$w^T x - b = 0 \tag{2.1}$$

Znalezienie odpowiedniej hiperpłaszczyzny sprowadza się do minimalizacji funkcji kosztu $f(w, b)$

$$\left[\frac{1}{n}\sum_{i=1}^{n}\max(0, 1 - y_i(w^T x_i - b))\right] + \lambda||w||^2 \tag{2.2}$$

gdzie n to liczba punktów, x_i to i-ty punkt danych, y_i to wartość jego klasy (1 lub -1), a λ to hiperparametr algorytmu.
Pierwszy człon wyraża karę za źle sklasyfikowane punkty (tym większą im dalej są od hiperpłaszczyzny). Drugi człon jest odwrotnie proporcjonalny do kwadratu odległości hiperpłaszczyzny od poprawnie sklasyfikowanych punktów.

2.2.2 Właściwości

Maszyna wektorów nośnych w opisanej postaci tworzy model liniowy, charakteryzuje się więc wysokim obciążeniem (*bias*) i niską wariancją. W sytuacjach gdzie klasy nie są łatwo separowalne liniowo, często stosuje się tzw. kernel trick przekształcający nieliniowo oryginalną przestrzeń punktów [1].

2.2.3 Implementacja

W ramach niniejszej pracy użyłyśmy klasycznej implementacji maszyny wektorów nośnych dla języka Python zawartej w klasie `sklearn.svm.LinearSVC` z biblioteki `scikit-learn`. Pozwala ona na zdefiniowanie hiperparametru $C = 1/\lambda$.
Do znalezienia parametru C użyłyśmy klas z biblioteki `sklearn.model_selection` - `RandomizedSearchCV` opisanej powyżej (Sekcja. 2.1.4.1) oraz `GridSearchCV` która również używa techniki walidacji krzyżowej, jednak sprawdza wszystkie możliwe kombinacje ze zdefiniowanych przez nas zbiorów.

2.3 Konwolucyjna sieć neuronowa

Konwolucyjna sieć neuronowa (*Convolutional Neural Network*) to rodzaj sztucznej sieci neuronowej. Jej konstrukcja jest inspirowana procesami biologicznymi zachodzącymi w korze wzrokowej w mózgu [6]. Typowo sieci konwolucyjne wykorzystuje się do zadań uczenia maszynowego związanego z rozpoznawaniem obrazów. Mogą być jednak stosowane także w innych dziedzinach, w szczególności w przetwarzaniu języka naturalnego.

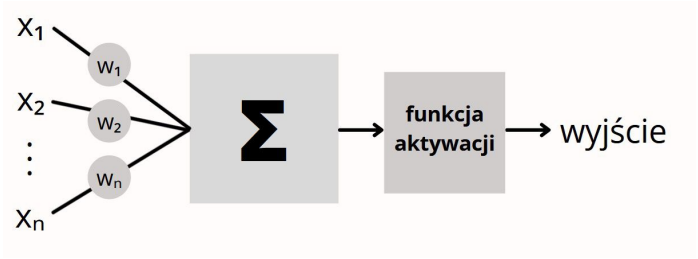
W tej sekcji najpierw opisany jest koncept klasycznej sieci neuronowej. Następnie opisana jest konstrukcja konwolucyjnej sieci, oraz zastosowanie jej w przetwarzaniu języka naturalnego. Na końcu podane są szczegóły implementacji sieci na potrzeby niniejszej pracy.

2.3.1 Sieci neuronowe

Klasyczna wielowarstwowa sieć neuronowa składa się z sekwencji warstw, w których znajduje się duża liczba węzłów obliczeniowych (tzw. neuronów). Pojedynczy neuron przyjmuje na wejściu wektor liczb rzeczywistych x, do którego następnie aplikowany jest wektor wag w (wynik uczenia sieci) przypisany do tego neuronu. W wyniku otrzymywana jest liczba:

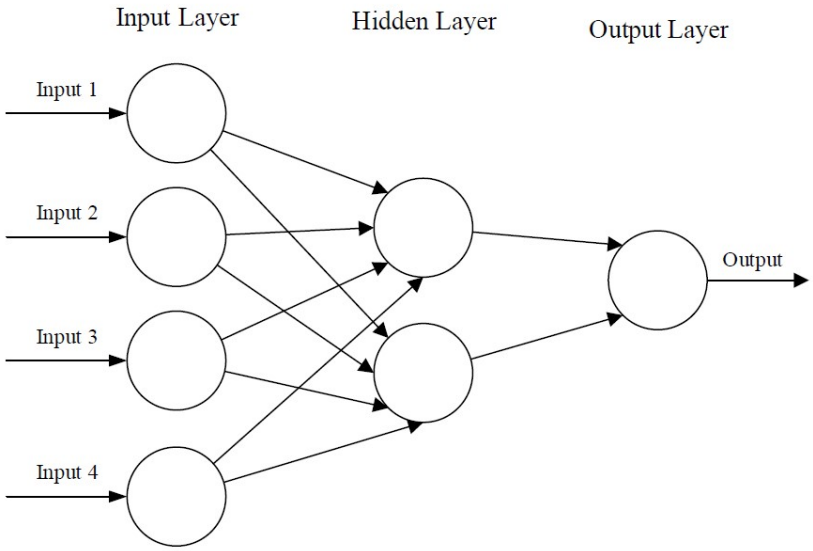
$$a = \sum_{i=1}^{n} w_i x_i \tag{2.3}$$

do której aplikowana jest funkcja aktywacji — decydująca o tym, jaki wynik zostanie przekazany do kolejnej warstwy [14]. Funkcje aktywacji odgrywają ważną rolę w sieciach neuronowych — pozwalają dodać nieliniowość — bez ich użycia, wynik wyjściowy modelu byłby kombinacją liniową danych wejściowych, z ograniczonymi możliwościami modelowania skomplikowanych typów danych (takich jak obrazy czy język naturalny) [35].



Rysunek 2.3: Schemat działania pojedynczego neuronu

Pierwsza warstwa nazywana jest warstwą wejściową (*input layer*), która przyjmuje na wejściu czyste dane w postaci wektorów. Kolejne warstwy to warstwy ukryte (*hidden layers*), do których wejściami są wyniki przekazane przez neurony poprzedniej warstwy. Ostatnią warstwą jest warstwa wyjściowa (*output layer*), która na wejściu również przyjmuje dane z poprzedniej warstwy, a której wyjcie jest predykcją modelu.



Rysunek 2.4: Wielowarstwowa sieć neuronowa [30]

Istnieje wiele różnych funkcji aktywacji, z których najpopularniejsze opisane są w [35]. W ramach niniejsze pracy wykorzystałyśmy następujące funkcje:

1. Funkcja sigmoidalna

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.4}$$

Funkcja ta, mapująca liczby rzeczywiste na przedział $(0, 1)$, jest jedną z najczęściej stosowanych w sieciach neuronowych.

2. Funkcja ReLU (od ang. *rectified linear unit*)

$$f(x) = \max(0, x) \tag{2.5}$$

Funkcja ta jest szeroko stosowana w warstawch ukrytych m.in. dzięki prostocie obliczeń oraz rozrzadzeniu aktywnych neuronów (neurony aktywują się tylko dla $a > 0$).

2.3.2 Trening sieci neuronowej

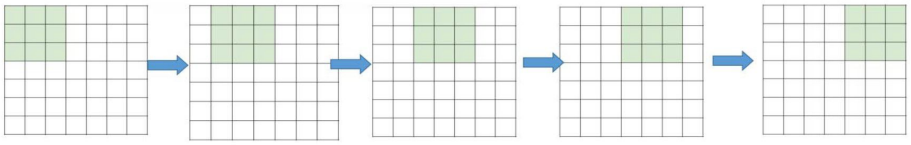
Trening — uczenie sieci neuronowej ma na celu zminimalizowanie obserwowanego błędu, czyli różnicy między predykcją sieci na danych treningowych a faktycznymi klasami. Sieć inicjalizuje się z losowymi wagami. Następnie sieć jest aplikowana do danych wejściowych i wagi są aktualizowane w celu zmniejszeniu uzyskanego błędu. Proces powtarzany jest iteracyjnie. [14]. Hiperparametrami modelu są liczba iteracji (epok) oraz ile próbek naraz używać podczas jednego kroku aktualizacji (wielkość wsadu, *batch size*).
Aktualizacja wag odbywa się za pomocą procesu zwanego propagacją wsteczną (*backpropagation*) [13]. Wagi aktualizowane są w kierunku gradientu funkcji błędu w zależności od tych wag. Innymi słowy, wartość błędu jest dzielona proporcjonalnie pomiędzy wagi które za niego odpowiadają.
W celu przeciwdziałania przeuczeniu się sieci neuronowej, można wykorzystać technikę zwaną *dropout* [36]. Polega ona na zignorowaniu podczas obliczania predykcji na potrzeby treningu niektórych neuronów z prawdopodobieństwem p będącym hiperparametrem. Dzięki temu że żaden neuron nie jest trenowany na całym zbiorze treningowym, sieć będzie miała mniejszą skłonność do przeuczenia się.

2.3.3 Sieci konwolucyjne

Sieci konwolucyjne są modyfikacją sieci neuronowych, stworzone z myślą o przetwarzaniu obrazów. Wejściem do sieci konwolucyjnej jest trójwymiarowa macierz: typowo jest to dwuwymiarowy obrazek, gdzie każdy piksel opisany jest trzema wartościami — R, G, B. Podobnie jak w przypadku sieci neuronowych, sieć konwolucyjna składa się z sekwencji warstw.

Pojedyncza warstwa konwolucyjna składa się z pewnej liczby (hiperparametr) filtrów. Każdy filtr to trójwymiarowa macierz wag. Wysokość i szerokość filtra (hiperparametry)

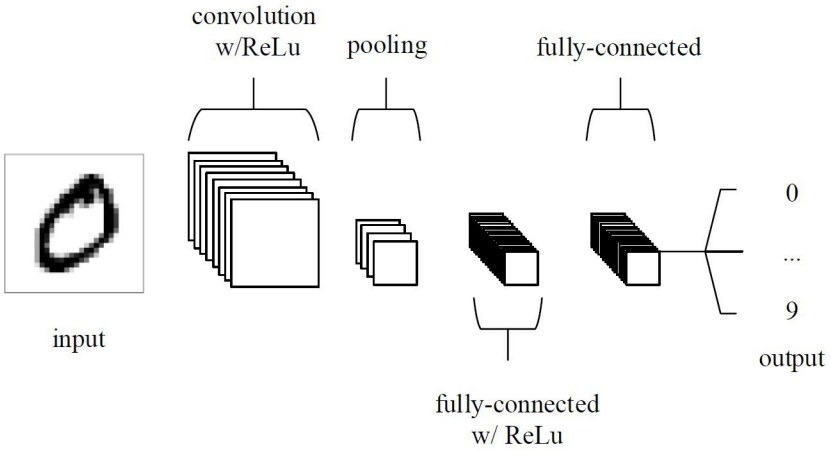
powinny być mniejsze niż odpowiednio wysokość i szerokość wejścia, natomiast głębokość musi być równa głębokości wejścia. Zaaplikowanie filtru do macierzy wejściowej polega na "przyłożeniu" go do każdej możliwej pozycji x,y wejścia, i obliczeniu iloczynu skalarnego filtra i odpowiedniego wycinka wejścia. W wyniku tej aplikacji powstaje dwuwymiarowa macierz zawierająca iloczyny skalarne, o wysokości i szerokości równych wysokości i szerokości wejścia (aby nie tracić informacji o pikselach znajdujących się na rogach obrazkach, często stosuje się technikę nazywaną *zero-padding* — jej szczegółowy opis można znaleźć w [2]. Wyniki aplikacji wszystkich filtrów z warstwy razem tworzą macierz trójwymiarową, która służy jako wejście do kolejnej warstwy.



Rysunek 2.5: Zastosowanie filtra w sieci konwolucyjnej [2]

Oprócz warstw konwolucyjnych, stosuje się też inne rodzaje warstw. Jedną z nich jest warstwa typu *pooling*. Jej zadaniem jest zmniejszenie wymiarowości danych płynących przez sieć. W tym celu segmenty wejścia są agregowane, na przykład poprzez branie maksimum z ich wartości [2].

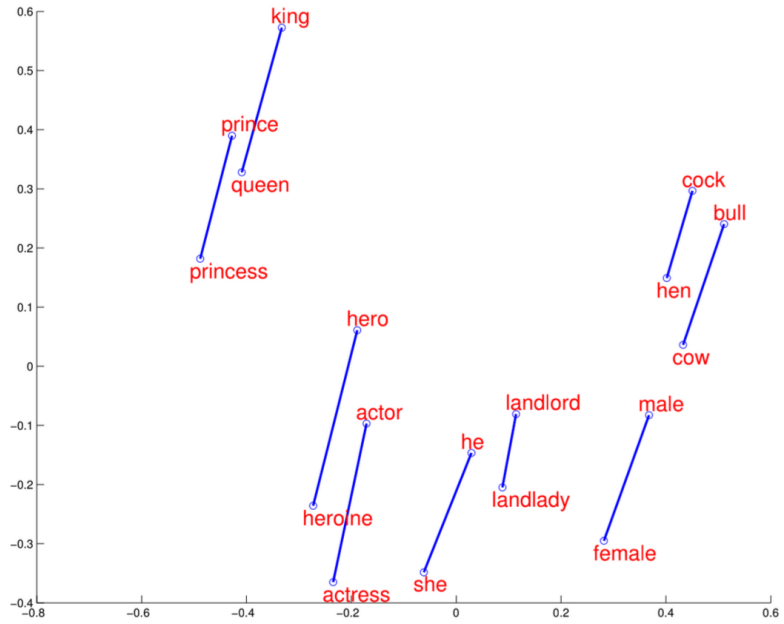
Architektura sieci konwolucyjnej typowo składa się z kilku warstw konwolucyjnych przeplatanych z warstwami *pooling*, a następnie kilku warstw klasycznych sieci neuronowych, w tym kontekście nazywanych warstwami *fully connected*. Trójwymiarowa macierz wychodząca z warstwy konwolucyjnej jest spłaszczana do jednowymiarowego wektora w celu przekazania jej do warstwy *fully connected*.



Rysunek 2.6: Pięciowarstwowa sieć konwolucyjna [30]

2.3.4 Wykorzystanie sieci konwolucyjnych w przetwarzaniu języka naturalnego

Choć sieci konwolucyjne wymyślone zostały z intencją przetwarzania obrazów, w ostatnich latach wielokrotnie były wykorzystywane do przetwarzania tekstu [24]. Wymaga to najpierw zanurzenia (*embedding*) słów — wyrażenia każdego słowa w postaci wektora liczb. Można wykorzystać do tego metodę Word2Vec [23]. Polega ona na wytrenowaniu sieci neuronowej na korpusie tekstów, która dla danego słowa zwraca wektor liczb. Sieć trenowana jest w ten sposób, by wektory słów występujących w podobnym kontekście znajdowały się blisko siebie.



Rysunek 2.7: Wizualizacja działania metody Word2Vec

W przypadku tekstu, wejściem dla sieci konwolucyjnej będzie macierz nie trójwymiarowa, a dwuwymiarowa. Zamiast dwuwymiarowego obrazu używany jest jednowymiarowy tekst. Podczas gdy w przypadku obrazów każdy piksel reprezentowany jest za pomocą trzech liczb (R, G, B), każde słowo tekstu wejściowego jest wyrażone odpowiadającym mu wektorem zanurzenia.

W przypadku obrazów, pojedynczy filtr aplikowany jest do pikseli znajdujących się blisko siebie. Intuicyjnie, każdy piksel należy interpretować w kontekście jego otoczenia. W przypadku tekstów, analogicznie filtr aplikowany jest do słów znajdujących się blisko siebie. Jedyną różnicą jest że dla obrazów otoczenie ma dwa wymiary — szerokość i wysokość, zaś dla tekstów jest to tylko jeden wymiar.

2.3.5 Implementacja

2.3.5.1 Word2Vec

W niniejszej pracy w stworzenia wektorów używanych do reprezentacji słów w nazszych modelach użyłyśmy implementacji algorytmu w języku Python zawartej w klasie `gensim.models.Word2Vec` z biblioteki `gensim`.
Zdecydowałyśmy się na własne wytrenowanie modelu na słowniku z danych treningowych, w związku z tym, że konktekst słów używanych w recenzjach filmowych często różni się od tych wykorzystywanych w różnego rodzaju artykułach (na których zazwyczaj trenowane są dostępne modele).

2.3.5.2 Sieć konwolucyjna

W ramach niniejszej pracy wykorzystałyśmy bibliotekę `keras`, która jest jedną z najczęściej używanych deep-learningowych bibliotek w języku Python. Pozwala ona na proste dodawanie kolejnych warst sieci, w celu tworzenia skomplikowanych modeli.
Do implementacji sieci konwolucyjnej użyłyśmy następujących funkcji:

- `keras.models.Sequential` budowanie modelu, do którego następnie dodawane są kolejne warstwy. Kolejne sekwencje warstw na wejściu przyjmują wyjścia z warstw poprzednich.
- `keras.layers.Embedding` pierwsza warstwa modelu, której wejściem jest lista słów treningowych oraz sposób ich wektoryzacji (wynik działania modelu Word2vec), a wyjściem słowa przetłumaczone na wektory
- `keras.layers.Conv1D` warstwa konwolucyjna aplikująca filtry o wysokości 1
- `keras.layers.GlobalMaxPooling1D` warstwa typu *pooling*, biorąca maksymalną wartość z dla każdego filtra
- `keras.layers.Dropout` warstwa wykorzystana w celu zapobiegnięcia przetrenowania sieci
- `keras.layers.Dense` kolejnymi warstwami są tzw. warstwy *fully-connected*.

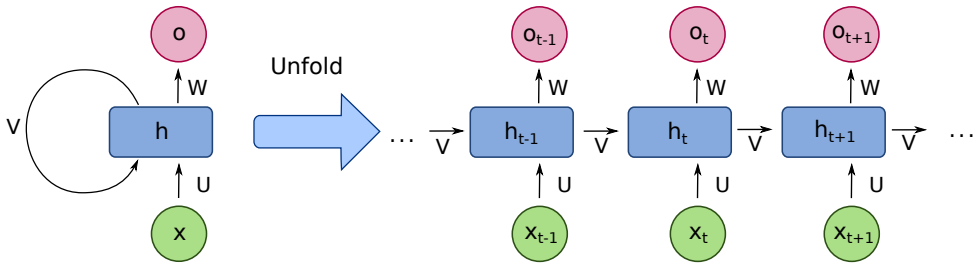
2.4 Sieć LSTM

Długoterminowa pamięć krótkoterminowa (*Long short-term memory*, dalej LSTM) to rekurencyjna sieć neuronowa. Została zaprojektowana z myślą o przetwarzaniu wejścia o zmiennej długości. Wejście do sieci przetwarzane jest sekwencyjnie — wyjście z sieci dla dotychczasowo przetworzonej sekwencji wpływa na zachowanie modelu dla dalszej części sekwencji. Dzięki temu LSTM bardzo dobrze nadaje się do przetwarzania tekstu.

W tej sekcji najpierw przybliżony jest koncept rekurencyjnej sieci neuronowej, następnie opisana jest konstrukcja sieci LSTM oraz jej zastosowanie w przetwarzaniu języka naturalnego. Na końcu podane są szczegóły implementacji sieci na potrzeby niniejszej pracy.

2.4.1 Rekurencyjna sieć neuronowa

Rekurencyjne sieci neuronowe są typem sieci neuronowych specjalizujących się w przetwarzaniu sekwencji danych [13]. W rekurencyjnej sieci neuronowej, neurony są połączone w sposób tworzący skierowany cykl. Jako wejście sieć przyjmuje sekwencję elementów — na przykład wektory reprezentujące kolejne słowa tekstu. Aplikacja sieci rekurencyjnej dla danej sekwencji polega na wprowadzeniu do sieci po kolei każdego z elementów. Otrzymane wyjście sieci — wektor stanu — jest każdorazowo wprowadzany do niej wraz z kolejnym przetwarzanym elementem sekwencji. Ponadto, wektor stanu jest przetwarzany dodatkowymi neuronami w celu otrzymania kolejnego wektora wyjściowego. Zauważmy, że tak zdefiniowana sieć rekurencyjna pozwala uzyskać na wyjściu sekwencje o zmiennej długości. Jednak w przypadku klasyfikacji interesuje nas jedynie ostatni wektor wyjściowy.



Rysunek 2.8: Schemat działania rekurencyjnej sieci neuronowej z wektorem wejściowym x, wektorem wyjściowym o, wektorem stanu h oraz macierzami wag U, V, W. Po lewej stronie przedstawiony został uproszony model sieci, po prawej widzimy aplikację sieci dla kolejnych wektorów z sekwencji wejściowej

Uczenie rekurencyjnej sieci neuronowej przebiega tak samo jak uczenie klasycznej sieci neuronowej, poprzez mechanizm propagacji wstecznej. Podczas uczenia sieci rekurencyjnej istnieje jednak duże ryzyko wystąpienia zjawiska zanikającego gradientu (*vanishing gradient problem*) [17]. Problem ten może wystąpić także w klasycznych, nie rekurencyjnych sieciach neuronowych. Polega on na tym, że aktualizacje wag docierające do wcześniejszych (bliższych wejścia) warstw sieci stają się zbyt małe by znacząco zmienić jej zachowanie — tym samym uniemożliwiając uczenie. W przypadku sieci rekurencyjnych problem jest bardziej dotkliwy, ponieważ dla przetworzenia całej sekwencji wejściowej są użyte te same wagi. Jeśli więc wagi te są mniejsze niż 1, dla wczesnych elementów sekwencji aktualizacje będą zbiegać do zera. (W przypadku wag większych niż 1, będą zbiegać do nieskończoności, co nazywane jest problemem eksplodującego gradientu). Konstrukcja LSTM opisana w następnej sekcji jest jednym ze sposobów przeciwdziałania temu zjawisku.

2.4.2 Konstrukcja LSTM

Model LSTM ma możliwość zapisania i odczytania stanu związanego z dawno przetworzonymi elementami wejściowymi. W kolejnych krokach aplikacji LSTM, oprócz dotychczasowego wyjścia przekazywany jest też wektor stanu pamięci. Po wczytaniu kolejnego elementu z sekwencji wejściowej, sieć może "zdecydować" czy odczytać informację z pamięci, czy ją nadpisać, i czy ją zresetować. Jest to zrealizowane poprzez specjalne bramki, które dla obecnego wejścia zwracają wartość 0 lub 1 (poprzez zaaplikowanie funkcji sigmoidailnej). Kolejna bramka oblicza jaką wartością nadpisać zawartość pamięci.

2.4.3 Zastosowanie LSTM w przetwarzaniu języka naturalnego

Zgodnie z jego oryginalnym przeznaczeniem, LSTM można zaaplikować do przetwarzania tekstów — ciągów słów. Ponieważ na wejściu LSTM oczekuje sekwencji wektorów, tekst należy najpierw odpowiednio przetworzyć, na przykład za pomocą Word2Vec (Sekcja. 2.3.4).

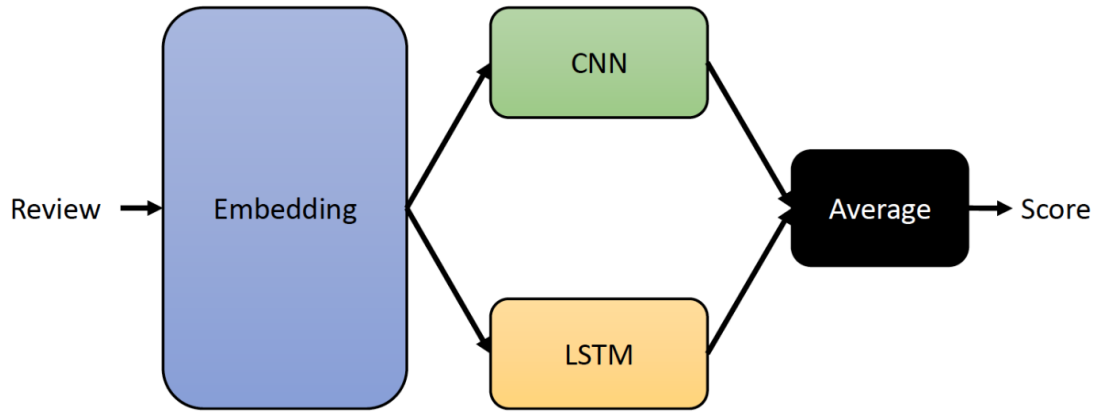
2.4.4 Implementacja

Tak jak w przypadku, wykorzystałyśmy tutaj bibliotekę `keras`, poza opisanymi wcześniej funkcjami wykorzystałyśmy także dodatkową wartwę `keras.layers.LSTM`. Implementuje ona rekurencyjną sieć LSTM, pozwala na zdefiniowanie wielkości wektora wyjściowego oraz prawdopodobieństwa *dropoutu*.

2.5 Połączenie sieci konwolucyjnej i LSTM

Częstym podejściem stosowanym w uczeniu maszynowym w celu poprawy jakości predykcji jest tzw. ensembling [29]. Polega on na połączeniu wielu wytrenowanych modeli predykcyjnych w jeden. W publikacji [24] autorzy opisują podejście w którym uśredniają predykcję z sieci konwolucyjnej oraz z modelu LSTM, uzyskując ostatecznie lepszy wynik na zbiorze testowym niż każdy z tych modeli z osobna.

W niniejszej pracy, podejście to zostało odtworzone za pomocą języka Python z wykorzystaniem bibliotek `numpy` oraz `pandas`.



Rysunek 2.9: Połączenie sieci konwolucyjnej i LSTM [24]

Rozdział 3

Analiza i modelowanie

W rozdziale tym opisane jest wykonane przez nas modelowanie. Wspieramy się tutaj wybranymi fragmentami kodu, natomiast cały kod dostępny jest w oddzielnym pliku. Wersje używanych bibliotek wylistowane są w Załączniku 4 na końcu pracy inżynierskiej.

3.1 Wstępne przygotowanie zbioru danych

W pierwszym etapie, nazwijmy wstępnym do tworzenia modeli, stworzyłyśmy notatnik w Google Colab zintegrowany z Dykiem Google i wczytałyśmy dane do `DataFrame`'a.

Listing 3.1: Wczytywanie surowych danych z Dysku Google do DataFrame'a.

```
import pandas as pd
from google.colab import drive
drive.mount('/content/drive')

data_file = "drive/MyDrive/IMDB_Dataset.csv"
data = pd.read_csv(data_file, header=0, low_memory=False)
```

Surowe dane przedstawione są na poniższym rysunku (Rysunek. 3.1).

Jak widać zbiór danych składa się z dwóch interesujących nas kolumn. Kolumna pierwsza o nazwie *review* (dtype: object) zawiera surowe treści recenzji filmów wraz z tagami w języku HTML, natomiast kolumna druga o nazwie *sentiment* zawiera informację, czy dana recenzja uznawana jest jako pozytywna czy negatywna — etykieta (dtype: object).

3	Basically there's a family where a little boy (Jake) thinks there's a zombie in his closet & his parents are fighting all the time. This movie is slower than a soap opera... and suddenly, Jake decides to become Rambo and kill the zombie. OK, first of all when you're going to make a film you must Decide if its a thriller or a drama! As a drama the movie is watchable. Parents are divorcing & arguing like in real life. And then we have Jake with his closet which totally ruins all the film! I expected to see a BOOGEYMAN similar movie, and instead i watched a drama with some meaningless thriller spots. 3 out of 10 just for the well playing parents & descent dialogs. As for the shots with Jake: just ignore them.	negative
4	Petter Mattei's "Love in the Time of Money" is a visually stunning film to watch. Mr. Mattei offers us a vivid portrait about human relations. This is a movie that seems to be telling us what money, power and success do to people in the different situations we encounter. This being a variation on the Arthur Schnitzler's play about the same theme, the director transfers the action to the present time New York where all these different characters meet and connect. Each one is connected in one way, or another to the next person, but no one seems to know the previous point of contact. Stylishly, the film has a sophisticated luxurious look. We are taken to see how these people live and the world they live in their own habitat. The only thing one gets out of all these souls in the picture is the different stages of loneliness each one inhabits. A big city is not exactly the best place in which human relations find sincere fulfillment, as one discerns is the case with most of the people we encounter. The acting is good under Mr. Mattei's direction. Steve Buscemi, Rosario Dawson, Carol Kane, Michael Imperioli, Adrian Grenier, and the rest of the talented cast, make these characters come alive. We wish Mr. Mattei good luck and await	positive

Rysunek 3.1: Przykładowe próbki z surowego zbioru danych recenzji filmowych IMDB.

Jako że model matematyczny działa na wartościach liczbowych, kolumnę etykiet słownych zamieniłyśmy na etykiety w postaci liczb, gdzie etykieta 0 oznacza recenzję negatywną, a etykieta 1 pozytywną (zmiana dtype z object na int64). Na tym etapie usunęłyśmy również tagi HTML wykorzystując pakiet `bs4`.

Listing 3.2: Zmiana etykiet z typu obiekt na typ liczbowy.

```
labels = labels.replace("negative", 0)
labels = labels.replace("positive", 1)

from bs4 import BeautifulSoup
reviews = reviews.apply(lambda text: BeautifulSoup(text).get_text())
reviews.head()
```

Przykładowa recenzja wraz z tagami HTML

Basically there's a family where a little boy (Jake) thinks there's a zombie in his closet & his parents are fighting all the time.

This movie is slower than a soap opera... and suddenly, Jake decides to become Rambo and kill the zombie.

OK, first of all when you're going to make a film you must Decide if its a thriller or a drama! As a drama the movie is watchable. Parents are divorcing & arguing like in real life. And then we have Jake with his closet which totally ruins all the film! I expected to see a BOOGEYMAN similar movie, and instead i watched a drama with some meaningless thriller spots.

3 out of 10 just for the well playing parents & descent dialogs. As for the shots with Jake: just ignore them.

Przykładowa recenzja po usunięciu tagów HTML

Basically there's a family where a little boy (Jake) thinks there's a zombie in his closet & his parents are fighting all the time.This movie is slower than a soap opera... and suddenly, Jake decides to become Rambo and kill the zombie.OK, first of all when you're going to make a film you must Decide if its a thriller or a drama! As a drama the movie is watchable. Parents are divorcing & arguing like in real life. And then we have Jake with his closet which totally ruins all the film! I expected to see a BOOGEYMAN similar movie, and instead i watched a drama with some meaningless thriller spots.3 out of 10 just for the well playing parents & descent dialogs. As for the shots with Jake: just ignore them.

Rysunek 3.2: Przykładowa recenzja przed usunięciem tagów HTML (surowa) oraz po usunięciu tagów.

Przy okazji sprawdziłyśmy, czy zbiór jest dobrze zbalansowany — czy liczba recenzji pozytywnych i negatywnych jest porównywalna. W kolejnym etapie podzieliłyśmy

zbiór danych na dane treningowe i testowe w proporcji 80:20 używając metody `train_test_split` z biblioteki `Pandas` i ponownie sprawdziłyśmy, czy po wykonanym podziale nasze podzbiory są zbalansowane (train: 20 044 recenzje pozytywne i 19 956 recenzji negatywnych oraz test: 4 956 recenzji pozytywnych i 5 044 recenzje negatywne).

Listing 3.3: Podział danych na zbiór treningowy i testowy.

```
from sklearn.model_selection import train_test_split
train_data, test_data, train_labels, test_labels = train_test_split(reviews, labels,

train_labels = train_labels.astype('int')
test_labels = test_labels.astype('int')
```

Następnie bazując na wbudowanej liście angielskich słów stop-words rozbudowanej o znaki interpunkcyjne i kilka własnych znaków (głównie cudzysłowów), usunęłyśmy wyrazy popularnie występujące a więc i nic nie wnoszące, po czym dokonałyśmy prostej wektoryzacji używając klasy `CountVectorizer`.

Listing 3.4: Wektoryzacja zbiorów danych — treningowego i testowego.

```
from sklearn.feature_extraction.text import CountVectorizer
count_vectorizer = CountVectorizer(stop_words=english_stopwords)
train_data_count = count_vectorizer.fit_transform(train_data)
test_data_count = count_vectorizer.transform(test_data)
```

3.2 Modelowanie

Po wstępnym przygotowaniu danych przyszedł czas na modelowanie. Stworzyłyśmy 4 rodzaje modeli oraz dla każdego z nich wykonałyśmy optymalizacje hiperparametrów aby otrzymać jak najlepsze wyniki. W pierwszej kolejności stworzyłyśmy model wykorzystujący układ drzew decyzyjnych — Las Losowy (*Random Forest*).

3.2.1 Las losowy

Model klasyfikujący stworzyłyśmy używając klasy `RandomForestClassifier` z pakietu `sklearn.ensemble`. Dokładny opis tego algorytmu znajduje sie w rozdziale 2 w sekcji 2.1.

Przy użyciu klasy `Randomized SearchCV` optymalizowałyśmy hiperparametry. Testowałyśmy działanie modelu dla różnych liczb estymatorów (n_estimators) wynoszących 10, 30 i 100, 300, 1000 oraz różnych głębokości drzew decyzyjnych (max_depth) wynoszących 4, 8, 16, 32, 64, None (bez limitu głębości).

Tablica 3.1: Wyniki optymalizacji dla dwóch najlepszych iteracji — z głębokością drzewa 16 i 64.

$\overline{t_{fit}}$	$\overline{t_{sc}}$	**N**	**D**	sc_1	sc_2	sc_3	sc_4	sc_5	$\overline{sc}$	$\pm$
685.14	0.05	**1000**	**64**	0.874	0.870	0.866	0.88	0.869	**0.869**	0.003
103.46	0.05	**1000**	**16**	0.859	0.855	0.854	0.854	0.868	**0.856**	0.002

sc — score, t — time (s), N — n_estimators, D — max_depth

Listing 3.5: Optymalizacja hiperparametrów RF (RandomizedSearchCV).

```
rf_params_count = {
  "n_estimators": [10, 30, 100, 300, 1000],
  "max_depth": [4, 8, 16, 32, 64, None]
}

rf_search_count = RandomizedSearchCV(RandomForestClassifier(), rf_params_count, refit
rf_search_count.fit(train_data_count, count)
rf_params_results_count = pd.DataFrame(rf_search_count.cv_results_)
rf_params_results_count.sort_values("rank_test_score").head(1)
```

Jak widać najlepszy wynik (0.87) został otrzymany dla liczby estymatorów równej 1000 oraz głębokości drzewa równej 64. Jednakże wynik otrzymany przy użyciu 4× mniejszej głębokości (16) był niewiele niższy (0.86), a średni czas trenowanie prawie 7× krótszy — jedynie 103.5 s w porównaniu do 685s. Widząc takie rezultaty, należy się zastanowić czy użycie drzew o mniejszej maksymalnej głębokości nie jest korzystniejsze. Trzeba pamiętać o tym, że drzewa decyzyjne mają tendencje do nadmiernego dopasowywania się do danych treningowych, a im głębsze drzewa tym o to przeuczenie łatwiej.

Listing 3.6: Trenowanie modelu lasu losowego dla 1000 estymatorów i głębokości 64.

```
random_forest_count = RandomForestClassifier(n_estimators = 1000, max_depth=64, verb
random_forest_count.fit(train_data_count, train_labels)
```

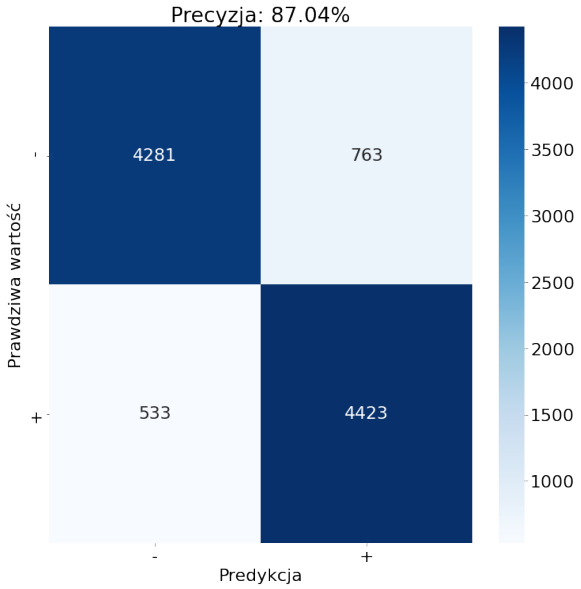
```
RandomForestClassifier(
                        bootstrap=True,
                        ccp_alpha=0.0,
                        class_weight=None,
                        criterion='gini',
                        max_depth=64,
                        max_features='auto',
                        max_leaf_nodes=None,
                        max_samples=None,
```

```
min_impurity_decrease=0.0,
min_impurity_split=None,
min_samples_leaf=1,
min_samples_split=2,
min_weight_fraction_leaf=0.0,
n_estimators=1000,
n_jobs=None,
oob_score=False,
random_state=None,
verbose=True,
warm_start=False
)
```

Trenowanie zajęło 13. 8 minuty. Następnie przy użyciu wytrenowanego modelu wykonałysmy predykcję.

Listing 3.7: Predykcja z użyciem wytrenowanego modelu lasu losowego.

```
random_forest_predictions_count = random_forest_count.predict(test_data_count)
```



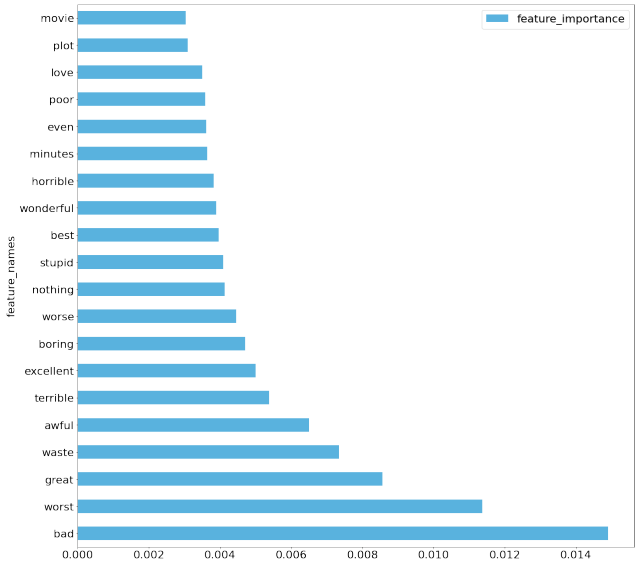
Rysunek 3.3: Wyniki klasyfikacji z użyciem lasu losowego dla n_estimators=1000 i max_depth=64. Trening modelu zajął 13.8 minuty.

Kolorem granatowym zostały oznaczone wartości przewidziane prawidłowo przez skonstruowany przez nas model — 4423 wartości zostały zaklasyfikowane prawidłowo jako pozytywne i 4281 wartości zostały zaklasyfikowane prawidłowo jako negatywne, co po zsumowaniu

daje 8704 prawidłowo sklasyfikowane wartości na 10 000 przykładów. Tak więc precyzja predykcji wynosi 87.04%. Ponadto 763 przykładów zostało błędnie sklasyfikowanych jako pozytywne i 533 błędnie jako negatywne (ćwiartki w kolorze błękitnym). Całkowity czas trenowania modelu zajął tutaj prawie 14 minut. Wydaje się nam, że jest to dosyć długi czas jak na stosunkowo mały (40 000 przykładów) zbiór treningowy.
W związku z tym, dla porównania wykonałyśmy również analogiczne modelowanie (fitowanie) dla maksymalnej głębokości drzew wynoszącej 32. Czas uległ znacznemu skróceniu. Trenowanie tego samego zbioru zajęło niecałe 4 minuty, a osiągnięta precyzja wyniosła 86.65%, a więc była tylko o 0.39% mniejsza niż dla 2× głębszych drzew. Szczegółowe wyniki przedstawione są w załączonym notatniku Google Colab. Wynik ten wyraźnie sugeruje, że należy się zawsze zastanowić nad sensem dużej głębokości drzew, gdyż w płytszymi drzewami również można osiągnąć zadowalające wyniki, a przy tym znacznie zredukować czas wykonywanych obliczeń.
Przy pomocy zaimplementowanej przez nas dedykowanej metody (Listing. 3.8) został też wyznaczony priorytet cech (*feature importance*), na bazie których następowały podziały w kolejnych węzłach drzew decyzyjnych.

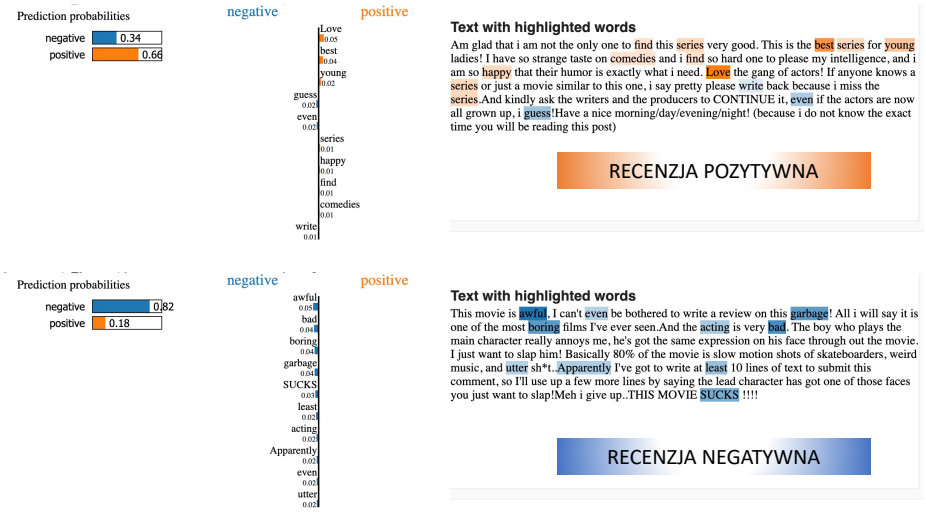
Listing 3.8: Metoda do wyznaczania i wizualizacji priorytetu cech dla drzewa decyzyjnego.

```
def plot_feature_importance(model, vectorizer):
  feature_importance = np.array(model.feature_importances_)
  feature_names = np.array(vectorizer.get_feature_names())
  data = pd.DataFrame({'feature_names': feature_names, 'feature_importance': feature

  max_data = data.nlargest(20, ['feature_importance'])

  max_data.plot(kind='barh', x='feature_names', y='feature_importance', color=blue_C
  plt.show()

plot_feature_importance(random_forest_count, count_vectorizer)
```



Rysunek 3.4: Ważność cech w drzewach decyzyjnych.

Znalazłysmy też ciekawe narzędzie — bibliotekę `LIME` — które umożliwia zwizualizowanie wydźwięków kluczowych słów, które posłużyły do sklasyfikowania danej recenzji jako pozytywnej lub negatywnej. Poszczególne słowa są w tym przypadku oznaczone albo kolorem pomarańczowym (jeżeli są pozytywne) albo niebieskim (gdy są negatywne). Intensywność koloru mówi o tym jak bardzo jwydźwięk ten jest pozytywny lub negatywny. Na poniższym rysunku (Rysunek. 3.5) znajduję się wizualizacja dla przykładowej recenzji pozytywnej (góra) oraz negatywnej (dół).

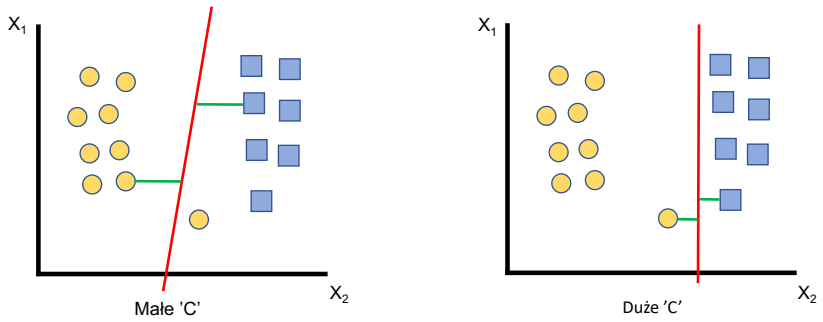


Rysunek 3.5: Wykres LIME wydźwięków kluczowych słów dla przykładowej recenzji pozytywnej (góra) oraz negatywnej (dół).

Istnieje również możliwość wykonania wizualizacji drzew decyzyjnych, co w prostu sposób pozwala interpretować i wyciągać wnioski. Jednakże głębokość naszych drzew uniemożliwia przedstawienie ich w sposób czytelny, dlatego wizualizacja została pominięta.

3.2.2 SVM

Kolejny model klasyfikacyjny stworzyłyśmy używając klasy `LinearSVC` z pakietu `sklearn.svm`. Dodatkowo przy użyciu klasy `RandomizedSearchCV` oraz `GridSearchCV` szukałyśmy optymalnego parametru 'C'. Parametr 'C' informuje o tym, jak bardzo chce się uniknąć błędnej klasyfikacji każdego przykładu ze zbioru treningowego [37]. Dla dużych wartości 'C' optymalizator wybiera hiperpłaszczyznę o mniejszym marginesie, jeśli ta hiperpłaszczyzna lepiej radzi sobie z prawidłowym sklasyfikowaniem wszystkich punktów treningowych. Przeciwnie, bardzo mała wartość 'C' spowoduje, że optymalizator będzie szukał hiperpłaszczyzny oddzielającej o większym marginesie, nawet jeśli ta hiperpłaszczyzna błędnie zaklasyfikuje większą liczbę punktów (Rysunek. 3.6). Parametr 'C' jest zasadniczo parametrem regularyzacji, który kontroluje kompromis między osiągnięciem małego błędu dopasowania modelu na danych uczących a minimalizacją norm wag w równaniu modelu.



Rysunek 3.6: Idea obrazująca parametr 'C' — płaszczyzny dla małej (lewy) i dużej (prawy) wartości parametru 'C'.

Pierwsza optymalizacja została wykonana z użyciem klasy `RandomizedSearchCV` dla: `params = {'C': scipy.stats.expon(scale=10)}`.

Listing 3.9: Optymalizacja hiperparametrów SVM (RandomizedSearchCV).

```
from sklearn.svm import LinearSVC
import scipy
from sklearn.model_selection import RandomizedSearchCV

params = {'C': scipy.stats.expon(scale=10)}
```

```
6
7  grid_count2 = RandomizedSearchCV(LinearSVC(max_iter=5000), params, refit= True, verb
8  grid_count2.fit(train_data_count, train_labels)
9  param_results = pd.DataFrame(grid_count2.cv_results_)
10 param_results.sort_values("rank_test_score").head(1)
```

Tablica 3.2: Wyniki optymalizacji dla najlepszych iteracji RandomizedSearchCV.

$\overline{t_{fit}}$	$\overline{t_{sc}}$	**C**	sc_1	sc_2	sc_3	sc_4	sc_5	$\overline{sc}$	$\pm$
10.9	0.005	**0.533**	0.865	0.865	0.868	0.863	0.867	**0.866**	0.002
18.2	0.005	**2.29**	0.861	0.862	0.864	0.856	0.861	**0.861**	0.002

sc — score, t — time (s)

Ze względu na to, że model osiągał znacząco lepsze wyniki z niskimi wartościami parametru 'C', zdecydowałyśmy się na dalszą optymalizację — tym razem przy użyciu `GridSearchCV` dla: `params = {'C': np.linspace(0, 1, num=1000)}`.

Listing 3.10: Optymalizacja hiperparametrów SVM (GridSearchCV).

```
1  from sklearn.svm import LinearSVC
2  import scipy
3  from sklearn.model_selection import GridSearchCV
4
5  params = {'C': np.linspace(0, 1, num=1000)}
6
7  grid_count2 = GridSearchCV(LinearSVC(max_iter=500), params, verbose= 3)
8  grid_count2.fit(train_data_count, train_labels)
9  param_results = pd.DataFrame(grid_count2.cv_results_)
10 param_results.sort_values('rank_test_score').head(1)
```

Tablica 3.3: Wyniki optymalizacji dla najlepszych iteracji GridSearchCV.

$\overline{t_{fit}}$	$\overline{t_{sc}}$	**C**	sc_1	sc_2	sc_3	sc_4	sc_5	$\overline{sc}$	$\pm$
0.44	0.006	**0.003**	0.889	0.895	0.894	0.889	0.891	**0.891**	0.002
0.49	0.006	**0.004**	0.889	0.894	0.894	0.889	0.891	**0.891**	0.002

sc — score, t — time (s)

Jak widać, najlepsze wyniki otrzymałyśmy dla parametru 'C', który wynosi ≈0.003, a dokładnie 0.00300300300
3003003, w związku z czym zecydowałyśmy się go użyć w dalszym modelowaniu. Poniżej znajdują się fragmenty kodu odpowiadające trenowaniu (fitowaniu) modelu `LinearSVC` (Listing. 3.11) oraz predykcji wartości wyjścia (Listing. 3.12).

Listing 3.11: Trening SVM dla C 0.003003003003003003.

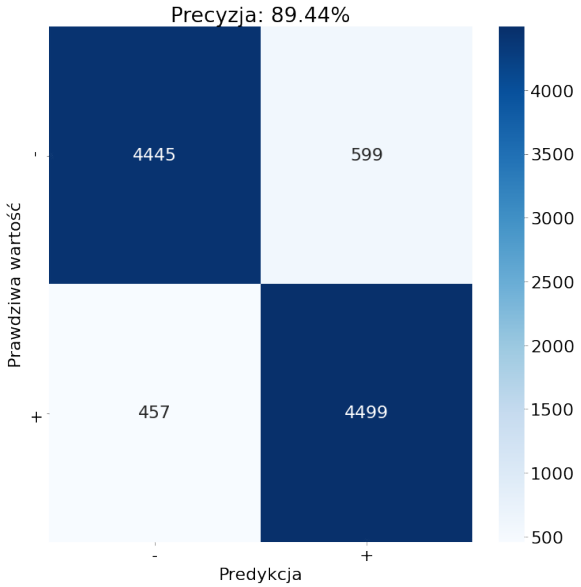
```
svc_classifier_count = LinearSVC(max_iter=500, C=0.003003003003003003)
svc_classifier_count.fit(train_data_count, train_labels)
```

```
LinearSVC(
        C=0.003003003003003003,
        class_weight=None,
        dual=True,
        fit_intercept=True,
        intercept_scaling=1,
        loss='squared_hinge',
        max_iter=500,
        multi_class='ovr',
        penalty='l2',
        random_state=None,
        tol=0.0001,
        verbose=0
        )
```

Listing 3.12: Predykcja SVM dla C 0.003003003003003003.

```
svc_predictions_count = svc_classifier_count.predict(test_data_count)
print_metrics(svc_predictions_count)
```

Na rysunku (Rysunek. 3.7) przedstawiona jest macierz omyłek (pomyłek) zestawiająca wartości prawdziwe oraz te otrzymane w wyniku predykcji wraz z zależnościami pomiędzy nimi.



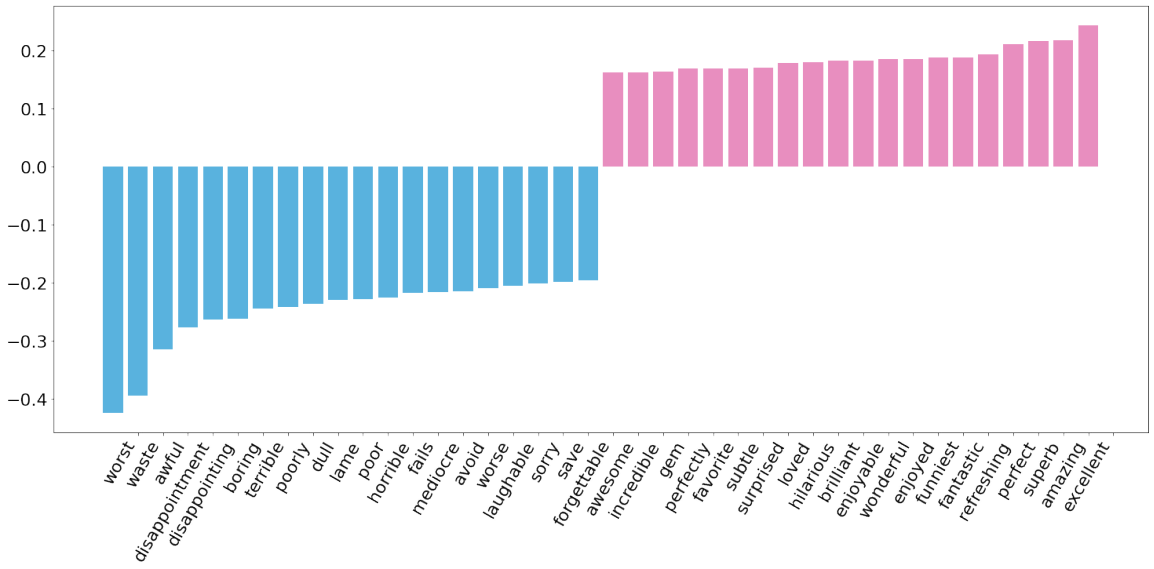
Rysunek 3.7: Wyniki klasyfikacji z użyciem maszyny wektorów nośnych dla parametru C $\approx$ 0.003.

Kolorem granatowym zostały oznaczone wartości przewidziane prawidłowo przez skonstruowany przez nas model — 4499 wartości zostały zaklasyfikowane prawidłowo jako pozytywne i 4445 wartości zostały zaklasyfikowane prawidłowo jako negatywne, co po zsumowaniu daje 8944 prawidłowo sklasyfikowane wartości na 10 000 przykładów. Tak więc precyzja predykcji wynosi 89.44%. Wynik wydaje się być logiczny. Ponadto 599 przykładów zostało błędnie sklasyfikowanych jako pozytywne i 457 błędnie jako negatywne (ćwiartki w kolorze błękitnym).
Na poniższym listingu (Listing. 3.13) przedstawiona jest implementacja metody `plot_coefficients(...)`, która umożliwia wykonanie wizualizacji najistotniejszych słów po klasyfikacji, które mają wydźwięk negatywny (kolor niebieski) lub pozytywny (kolor różowy).

Listing 3.13: Wizualizacja SVM.

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.svm import LinearSVC
import matplotlib.pyplot as plt

def plot_coefficients(classifier, feature_names, top_features=20):
  coef = classifier.coef_.ravel()
  top_positive_coefficients = np.argsort(coef)[-top_features:]
  top_negative_coefficients = np.argsort(coef)[:top_features]
  top_coefficients = np.hstack([top_negative_coefficients, top_positive_coefficients


  # create plot
```

```
    plt.figure(figsize=(15, 5))
    colors = [blue_0 if c < 0 else pink_0 for c in coef[top_coefficients]]
    plt.bar(np.arange(2 * top_features), coef[top_coefficients], color=colors)
    feature_names = np.array(feature_names)
    plt.xticks(np.arange(1, 1 + 2 * top_features), feature_names[top_coefficients], ro
    plt.show()

plot_coefficients(svc_classifier_count, count_vectorizer.get_feature_names())
```



Rysunek 3.8: Wizualizacja wydźwięku pozytywnego lub negatywnego wyrazów (SVM).

3.2.3 Sieci neuronowe

W kolejnych etapach stworzyłysmy modele sieci neuronowych a dokładniej konwolucyjnej sieci neuronowej (subsekcja 2.3.3), sieci LSTM — długoterminowej pamięci krótkoterminowej, która jest typem sieci rekurencyjnej (sekcja 2.4) oraz zamodelowałyśmy wyniki połączenia tych dwóch rodzajów sieci razem (sekcja 2.5).
Jako że użycie sieci neuronowych wymaga najpierw wyrażenia każdego słowa w postaci wektora liczb (*embedding*), zastosowałyśmy metodę Word2Vec [23]. Żeby osiągnąć pożądany rezultat zaimplementowałysmy metodą `get_lists_of_words(data)`, do której zaaplikowałyśmy nasze dane treningowe i w rezultacie otrzymałyśmy listę słów.

Listing 3.14: Stworzenie listy słów dla modelu Word2Vec.

```
import re
def get_lists_of_words(data):
    clean_reviews = data.map(lambda review: re.sub(r'([^a-z|\s])+', '', review.lower()
```

```
    return [review.split() for review in clean_reviews]

train_data_list_of_words = get_lists_of_words(train_data)
```

Następnie wytrenowałysmy model `Word2Vec` na tej stworzonej liście słów.

Listing 3.15: Trenowanie modelu Word2Vec.

```
import gensim
word2vec_model = gensim.models.Word2Vec(
        train_data_list_of_words,
        size=150,
        window=10,
        min_count=2,
        workers=10
        )

word2vec_model.train(train_data_list_of_words,total_examples=len(train_data_list_of_
```

Wytrenowane wektory słów są przechowywane w instancji KeyedVectors jako, w naszym przypadku, `word2vec_model.wv`. Ten obiekt zasadniczo zawiera mapowanie między słowami i osadzaniami/ zagłębieniami (*embeddings*).
Pakiet gensim zapewnia też wiele przydatnych metod. Na przykład dzięki metodzie `most_similar('słowo')` umożliwia znajdowanie słów najbardziej zbliżonych do podanego w parametrze słowa (Listing. 3.16, linia 1) oraz dzięki metodzie `doesnt_match ([słowa])` słowo/a, które nie pasuje/ą do reszty słów na liście (Listing. 3.16, linia 2).

Listing 3.16: Znajdowanie najbardziej zbliżonych i nie pasujących słów.

```
word2vec_model.wv.most_similar('actor')
word2vec_model.wv.doesnt_match(['good', 'nice','small', 'fine'])
```

Słowa najbardziej zbliżone do słowa actor:

```
[('actress', 0.5977960824966431),
('performer', 0.5925692915916443),
('role', 0.5761367678642273),
('comedian', 0.5694815516471863),
('performance', 0.5201245546340942),
('newcomer', 0.4978533387184143),
('actors', 0.4970739483833313),
('impersonation', 0.47669658064842224),
('thespian', 0.47024333477020264),
('foxx', 0.46994471549987793)]
```

Słowo nie pasujące do reszty:

small

Listing 3.17: Generacja wektora słów — model Word2Vec.

```
word_vectors = pd.DataFrame(word2vec_model[word2vec_model.wv.vocab], word2vec_model.
```

	0	1	2	3	4	...	145	146	147	148	149
one	-0.068556	-0.535242	-1.211615	0.509591	-1.221095		041565	1.211531	4.212045	-2.495354	-2.315618
of	2.456316	0.236809	-1.864299	2.539798	0.430902		081198	-1.271105	-0.497862	1.461353	2.070049
the	-1.027304	-0.736920	-0.374283	2.292591	0.009720		309368	-1.237591	0.496981	-0.552784	-0.906909
other	2.221614	2.065059	-2.451320	0.288963	-1.107342		096579	-0.349175	0.037189	0.119960	-3.079442
reviewers	-1.328646	1.123452	-0.757079	0.437397	-1.543197		332794	-2.574284	3.968311	-0.475326	0.407972
...	...	...	...	...	...	...	...	...	...	...	...
misleadingly	-0.095670	-0.119967	0.011220	0.059570	0.140247		083909	-0.094794	-0.045261	0.019793	0.031619
crosswords	0.041168	0.081060	-0.009382	-0.024368	-0.167992		044105	-0.046469	-0.004797	0.099627	-0.061112
kommodo	0.040535	0.114987	-0.049551	-0.066575	-0.256231		053583	-0.014575	0.129244	-0.101720	0.127185
morenos	-0.063756	0.102196	0.003643	-0.052255	0.060165		064596	-0.049735	0.160166	-0.090040	-0.023652
cattral	0.159017	0.023895	0.084336	-0.082799	-0.073952	...	071541	-0.086874	0.061408	-0.086801	0.026689

81589 rows × 150 columns

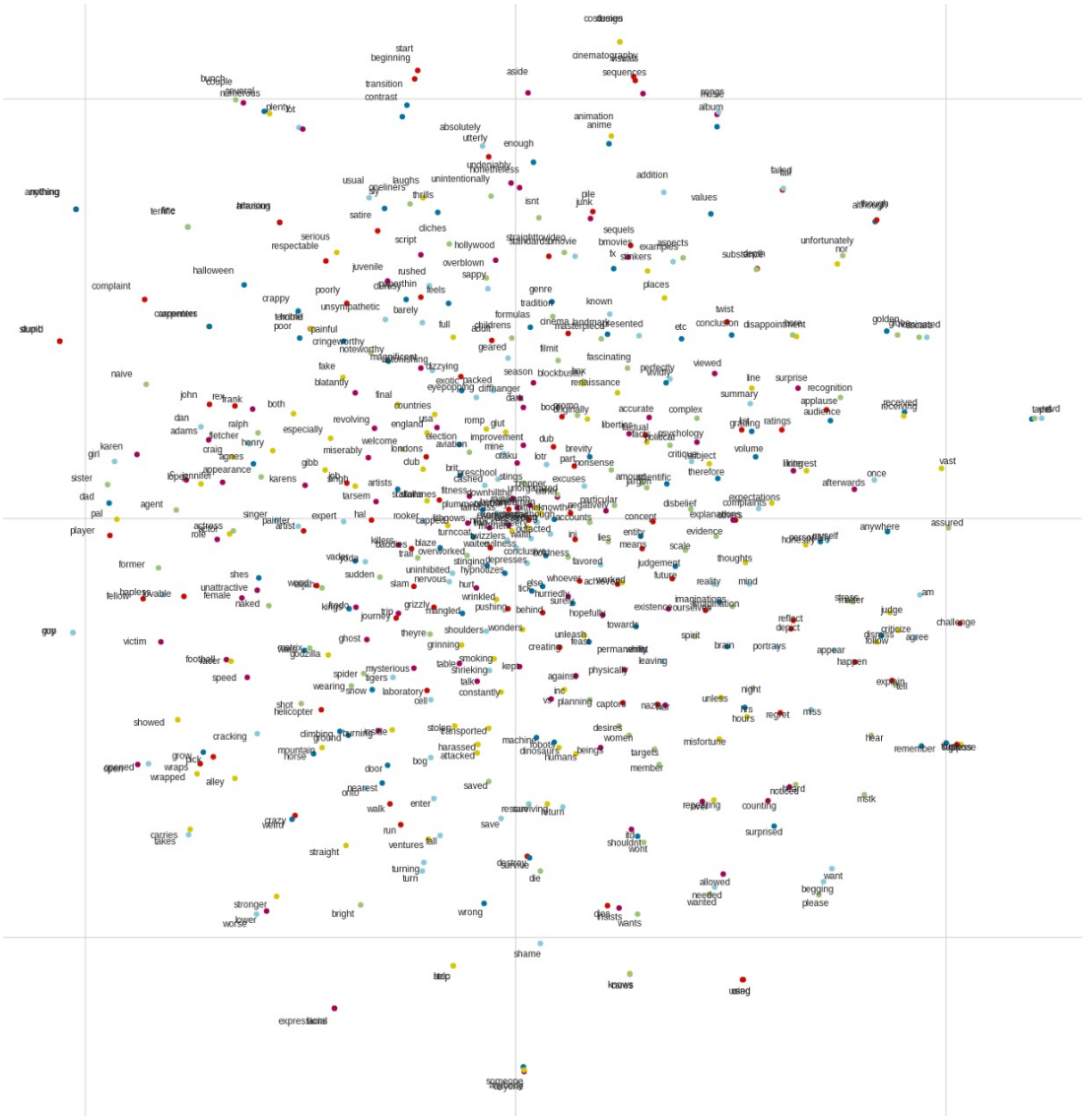
Rysunek 3.9: Wyniki modelu Word2Vec — wektory słów, dla size=150.

Rezultaty działania modelu można również wizualizować, ale najpierw należy zmniejszyć wymiarowość wektorów — w naszym przypadku ze 150 wymiarów do 2. Można to zrobić przy użyciu klasy `TSNE` z pakietu `sklearn.manifold` (Listing. 3.18), która pozwoli wygenerować odpowiednie punkty, które można później przedstawić w formie wykresu.

Listing 3.18: Zmniejszenie wymiarowości wektorów słów do punktów.

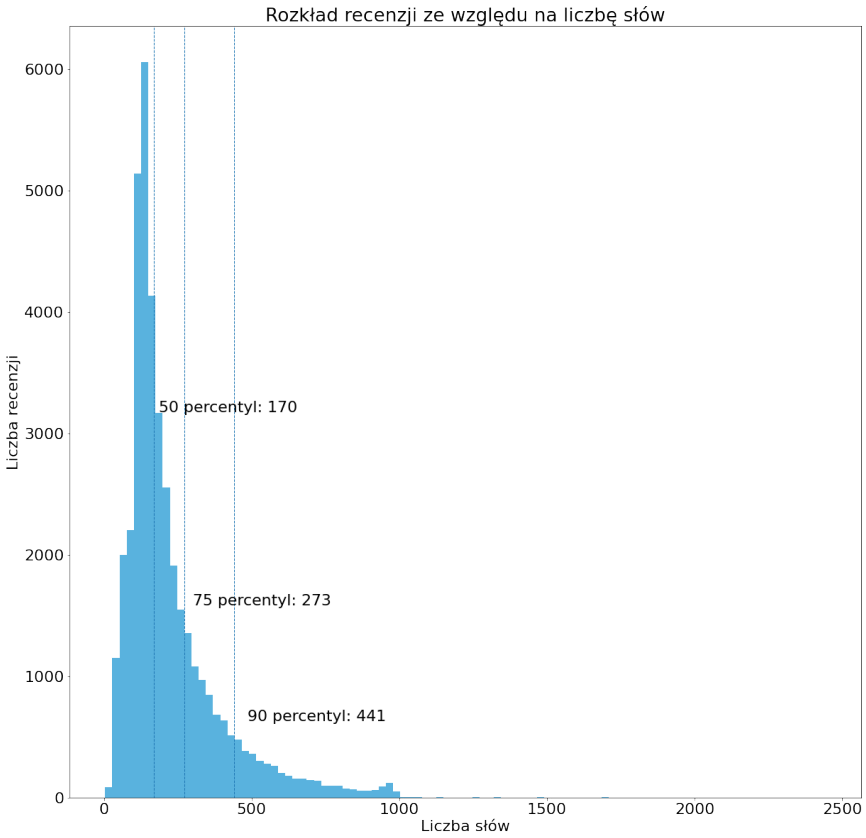
```
from sklearn.manifold import TSNE
tsne = TSNE(perplexity = 30, n_components=2, init='pca', n_iter=50000, method='exact

points = tsne.fit_transform(np.array(word_vectors)[1000:1500,:])
```

Możliwe jest wygenerowanie wykresu/ grafiki, na której zaznaczone są słowa (wszystkie lub wybrany podzbiór). Słowa o podobnym znaczeniu powinny tworzyć skupiska — znajdować się blisko siebie. Jednakże, jeżeli stworzy się taką grafikę dla zbyt dużej liczby słów, to staje się ona mało czytelna. Idea tej wizualizacji dla 500 przykładowych słów z naszego zbioru znajduje się poniżej (Rysunek. 3.10). Jak widać, przy tym powiększeniu grafika ta jest mało użyteczna. Po odpowiednim (dużym) powiększeniu można znaleźć pewne skupiska słów i zależności.



Rysunek 3.10: Wizualizacja W2V

Podobnie może być przydatny również rozkład długości recenzji (wyrażony jako liczba słów w recenzji). Taką zależnosc również można zwizualizować z zaznaczeniem wybranych percentyli (Rysunek. 3.11). W naszym przypadku łatwo zauważyć, że połowa recenzji ma 170 słów lub mniej, 25% recenzji ma długość między 171 a 273 słowa, kolejne 15% recenzji ma długość pomiędzy 274 a 441 słów, a 10% jest dłuższych niż 441 słów. Wyraźnie dominują tu recenzje krótkie. W sumie 75% recenzji jest co najwyżej średniej długości (do 273 słów).



Rysunek 3.11: Wizualizacja rozkładu liczby słów w recenzjach.

Tak więc tworząc macierze danych do trenowania modeli sieci neuronowych ustaliłyśmy 273 słowa jako górną granicę (maksymalny wymiar).

3.2.3.1 Konwolucyjna sieć neuronowa

W wyniku wykonania szeregu przekształceń (dokładna procedura znajduje się w notebooku Google Colab w części Data Preparation dla sieci neuronowych) otrzymałyśmy dwie macierze danych treningową — `train_data_matrix` i testową — `test_data_matrix`. Następnie zdefiniowałysmy funkcję `cnn_model` i przy użyciu biblioteki `keras` stworzyłyśmy model konwolucyjnej sieci neuronowej. W warstwie konwolucyjnej wykorzystałyśmy 128 filtrów o długości 5 oraz funkcję aktywacji ReLu. Następnie, wykorzystałyśmy warstwę *global max pooling* oraz *dropout* z prawdopodobieństwen 0.2. Użyłyśmy również 10 warstw ukrytych z funkcją aktywacji ReLu oraz ostatniej warstwy wyjściowej z sigmoidalną warstwą aktywacji (dokładne parametry na Listingu. 3.19). Jako funkcji straty użyłyśmy `binary_crossentropy`,a metryką sukcesu ponownie zostało `accuracy`.

Listing 3.19: Model CNN.

```
1 def cnn_model():
```

```
    embed_size=150
    model = Sequential()
    model.add(layers.Embedding(embedding_matrix_size + 1, embed_size, weights=[embeddi
    model.add(layers.Conv1D(128, 5, activation='relu'))
    model.add(layers.GlobalMaxPooling1D())
    model.add(layers.Dropout(0.2))
    model.add(layers.Dense(10, activation='relu'))
    model.add(layers.Dense(1, activation='sigmoid'))

    model.compile(loss='binary_crossentropy',
    metrics=['accuracy'])

    return model
```

```
Model: 'Sequential'
Layer (type)                 Output Shape              Param #
=================================================================
embedding_6 (Embedding)      (None, None, 150)         10894350
_________________________________________________________________
conv1d_6 (Conv1D)            (None, None, 128)         96128
_________________________________________________________________
global_max_pooling1d_6 (Glob (None, 128)               0
_________________________________________________________________
dropout_4 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_12 (Dense)             (None, 10)                1290
_________________________________________________________________
dense_13 (Dense)             (None, 1)                 11
=================================================================
Total params: 10,991,779
Trainable params: 10,991,779
Non-trainable params: 0
```

Powyższy model został skompilowany. Jak widać składa się z prawie 11 milionów trenowalnych parametrów, a następnie wytrenowany.

Listing 3.20: Trening modelu CNN.

```
cnn_model = cnn_model().fit(
      train_data_matrix,
      train_labels,
      epochs=8,
      verbose=True,
      batch_size=100
      )
```

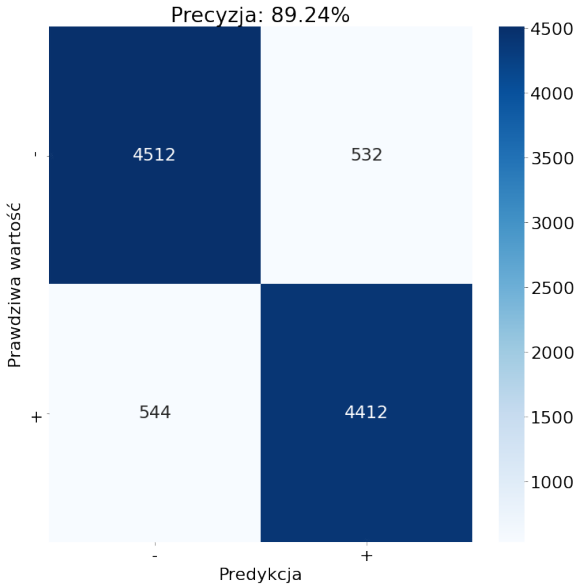
```
Epoch 1/8 - 169s 418ms/step - loss: 0.5815 - accuracy: 0.6748
Epoch 2/8 - 166s 416ms/step - loss: 0.3491 - accuracy: 0.8466
Epoch 3/8 - 166s 415ms/step - loss: 0.2954 - accuracy: 0.8750
Epoch 4/8 - 166s 414ms/step - loss: 0.2593 - accuracy: 0.8930
Epoch 5/8 - 166s 415ms/step - loss: 0.2279 - accuracy: 0.9069
Epoch 6/8 - 165s 412ms/step - loss: 0.1989 - accuracy: 0.9192
Epoch 7/8 - 164s 409ms/step - loss: 0.1729 - accuracy: 0.9316
Epoch 8/8 - 165s 412ms/step - loss: 0.1499 - accuracy: 0.9415
```

Ostatnią czynnością jaką zrobiłyśmy dla tego modelu była predykcja klas w oparciu o zbiór testowy (Listing. 3.21).

Listing 3.21: Predykcja z użyciem modelu CNN.

```
cnn_predictions = cnn_model.predict_classes(test_data_matrix)
```

Wyniki przedstawione są w macierzy omyłek (Rysunek. 3.12).
Kolorem granatowym zostały oznaczone wartości przewidziane prawidłowo przez skonstruowany przez nas model — 4412 wartości zostały zaklasyfikowane prawidłowo jako pozytywne i 4512 wartości zostały zaklasyfikowane prawidłowo jako negatywne, co po zsumowaniu daje 8924 prawidłowo sklasyfikowane wartości na 10 000 przykładów. Tak więc precyzja predykcji wynosi 89.24%. Ponadto 532 przykładów zostało błędnie sklasyfikowanych jako pozytywne i 544 błędnie jako negatywne (ćwiartki w kolorze błękitnym). Całkowity czas trenowania modelu zajął tutaj nieco ponad 22 minuty (średnio około 2 minuty i 46 sekund na 1 epokę). Jest to czas dłuższy niż dla dotychczasowych dwóch modeli, ale trzeba tu podkreślić, że model ten posiadał prawie 11 milionów trenowalnych parametrów.



Rysunek 3.12: Wyniki klasyfikacji z użyciem konwolucyjnej sieci neuronowej.

3.2.3.2 LSTM

Model LSTM został zaimplementowany podobnie jak model konwolucyjny z użyciem `Sequential()`. Zdefiniowałyśmy funkcję `model_lstsm` i przy użyciu biblioteki `keras` stworzyłyśmy model sieci neuronowej LSTM. Zamiast warstwy konwolucyjnej, używamy tutaj warstwy LSTM z wektorem wyjściowym o długości 128. Jako że jest to model rekurencyjny ustawiłyśmy dodatkowy parametr `recurrent_dropout` na wartość 0.2. Ostatnia warstwa — wyjściowa ma sigmoidalną aktywację (dokładne parametry na Listingu. 3.22). Jako funkcji straty ponownie użyłyśmy `binary_crossentropy`, a jako metryki sukcesu `accuracy`.

Listing 3.22: Model LSTM.

```
def lstsm_model():
  embed_size=150
  model = Sequential()
  model.add(layers.Embedding(embedding_matrix_size + 1, embed_size, weights=[embeddi
  model.add(layers.LSTM(128, dropout=0.2, recurrent_dropout=0.2))
  model.add(layers.Dense(1, activation='sigmoid'))
  print(model.summary())
  model.compile(loss='binary_crossentropy',
  metrics=['accuracy'])
  return model
```

```
Model: 'Sequential'
Layer (type)                 Output Shape              Param #
=================================================================
embedding_4 (Embedding)      (None, None, 150)         10894350
_________________________________________________________________
lstm (LSTM)                  (None, 128)               142848
_________________________________________________________________
dense_8 (Dense)              (None, 1)                 129
=================================================================
Total params: 11,037,327
Trainable params: 11,037,327
Non-trainable params: 0
```

Powyższy model został skompilowany. Jak widać składa się z ponad 11 milionów trenowalnych parametrów, a następnie wytrenowany.

Listing 3.23: Trening modelu LSTM.

```
lstm_model = lstsm_model().fit(
      train_data_matrix,
      train_labels,
      epochs=8,
      verbose=True,
      batch_size=100
      )
```

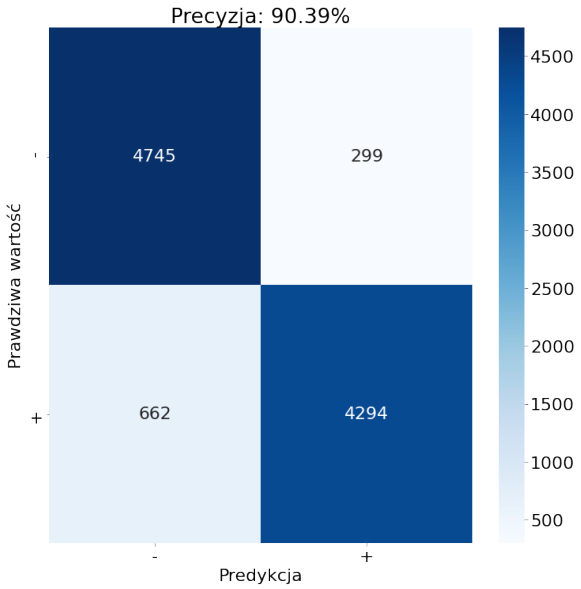
```
Epoch 1/8 - 583s 1s/step - loss: 0.5784 - accuracy: 0.6985
Epoch 2/8 - 576s 1s/step - loss: 0.4081 - accuracy: 0.8310
Epoch 3/8 - 583s 1s/step - loss: 0.3197 - accuracy: 0.8699
Epoch 4/8 - 577s 1s/step - loss: 0.2765 - accuracy: 0.8904
Epoch 5/8 - 575s 1s/step - loss: 0.2416 - accuracy: 0.9055
Epoch 6/8 - 577s 1s/step - loss: 0.2169 - accuracy: 0.9155
Epoch 7/8 - 569s 1s/step - loss: 0.1934 - accuracy: 0.9259
Epoch 8/8 - 564s 1s/step - loss: 0.1702 - accuracy: 0.9358
```

Ostatnią czynnością jaką zrobiłyśmy dla tego modelu była predykcja klas w oparciu o zbiór testowy (Listing. 3.21).

Listing 3.24: Predykcja z użyciem modelu LSTM.

```
lstm_predict_data = lstm_model.predict(test_data_matrix)
```

Poniżej przedstawiłyśmy macierz omyłek.



Rysunek 3.13: Wyniki klasyfikacji z użyciem sieci LSTM.

Kolorem granatowym zostały oznaczone wartości przewidziane prawidłowo przez skonstruowany przez nas model — 4294 wartości zostały zaklasyfikowane prawidłowo jako pozytywne i 4745 wartości zostały zaklasyfikowane prawidłowo jako negatywne, co po zsumowaniu daje 9039 prawidłowo sklasyfikowane wartości na 10 000 przykładów. Tak więc precyzja predykcji wynosi 90.39%. Ponadto 299 przykładów zostało błędnie sklasyfikowanych jako pozytywne i 662 błędnie jako negatywne (ćwiartki w kolorze błękitnym). Całkowity czas trenowania modelu zajął tutaj ponad 76 minut (średnio około 9 minut i 35 sekund na 1 epokę). Jest to czas znacznie dłuższy niż dla dotychczasowych modeli — lecz model jest również najbardziej zaawansowany i składa się z ponad 11 mln parametrów.

3.2.3.3 CNN–LSTM Ensemble

Jako ostatnią próbę polepszenia naszych predykcji, mimo, że wartości bliskie 90%, to bardzo wysokie wyniki, zastosowałyśmy połączenie działania modeli CNN i LSTSM (CNN-LSTM Ensemble).

Listing 3.25: Model stworzony przez połączenie CCN z LSTM.

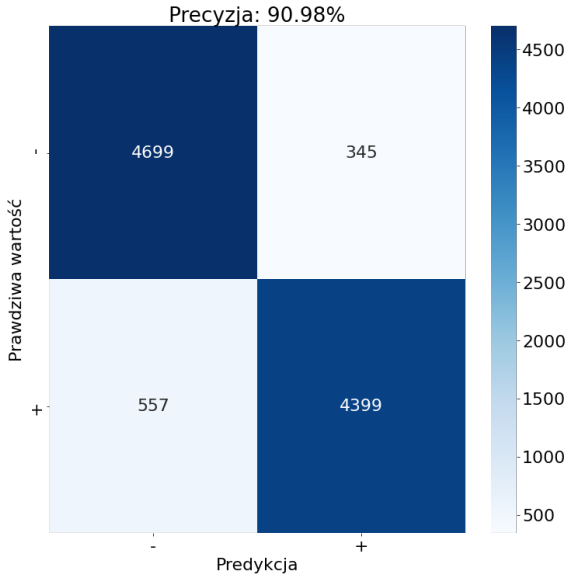
```
cnn_lstm_predictions = pd.DataFrame(cnn_predict_data, columns=['CNN'], index= range(
cnn_lstm_predictions['LSTM'] = pd.DataFrame(lstm_predict_data, index= range(0,len(te
cnn_lstm_predictions['AVG'] = cnn_lstm_predictions.mean(axis=1)
cnn_lstm_predictions['Prediction'] = np.where(cnn_lstm_predictions['AVG']>0.5, 1, 0)
cnn_lstm_predictions
```

	CNN	LSTM	AVG	Prediction
0	0.009072	0.032402	0.020737	0
1	0.000361	0.005767	0.003064	0
2	0.004610	0.006389	0.005499	0
3	0.000339	0.004691	0.002515	0
4	0.989763	0.991970	0.990867	1
...	...	...	...	...
9995	0.926858	0.958253	0.942556	1
9996	0.270218	0.949126	0.609672	1
9997	0.522854	0.008012	0.265433	0
9998	0.965067	0.995375	0.980221	1
9999	0.009403	0.005554	0.007479	0

10000 rows × 4 columns

Rysunek 3.14: Wyniki połączenia modeli CNN i LSTM.

Zgodnie z naszą intuicją, osiągnęłyśmy pożądany efekt. Precyzja osiągnięta z tego połączenie wyniosła 90.98%. Jest to najwyższy wynik wśród tych, które osiągnęłyśmy.



Rysunek 3.15: Macierz omyłek dla wyników z połączonych modeli CNN i LSTM.

Tradycyjnie kolorem granatowym zostały oznaczone wartości przewidziane prawidłowo przez skonstruowany przez nas model — 4399 wartości zostały zaklasyfikowane prawidłowo jako pozytywne i 4699 wartości zostały zaklasyfikowane prawidłowo jako negatywne, co po zsumowaniu daje 9098 prawidłowo sklasyfikowane wartości na 10 000 przykładów.

Tak więc precyzja predykcji wynosi 90.98%. Ponadto 345 przykładów zostało błędnie sklasyfikowanych jako pozytywne i 557 błędnie jako negatywne. Został tu osiągnięty najlepszy wynik jeżeli chodzi o precyzję predykcji. Jednakże minus jest tu taki, że trzeba wytrenować zarówno sieć konwolucyjną jak i rekurencyjną, co sumarycznie zajmuje znacznie więcej czasu niż zajmowało to dla prostszych modeli, które dawały nie dużo gorsze wyniki.

3.3 Porównanie wyników dla wszystkich modeli

W celu wyciągnięcia finalnych wniosków, porównałyśmy niektóre odpowiadające parametry stworzonych przez nas modeli. Te najważniejsze i decydujące zostały zestawione w poniższej tabeli (Tabela. 3.4). Modele posortowane zostały na podstawie precyzji — od najmniej wydajnego do najbardziej wydajnego.

Tablica 3.4: Porównanie czasów trenowania i precyzji wszystkich 5 modeli.

	RF (D=32)	RF (D=64)	CNN	SVM	LSTM	CNN–LSTM
czas	4 min.	14 min.	22 min.	1 sek.	76 min.	22 + 76 min.
precyzja	86.65%	87.04%	89.24%	89.44%	90.39%	90.98%

RF — Random Forest Classifier (maksymalna głębokość = 32 lub 64), SVM — Support Vector Machine Classifier, CNN — Convolutional Neural Network, LSTM — Long Short-Term Memory

Nie zawsze bardziej skomplikowany model jest lepszy. W przypadku naszego zbioru danych oraz modeli widać, że różnice w precyzjach między skrajnymi modelami wynoszą tylko 4.33%, natomiast czasy trenowania tych modeli znacznie się różnią. Pytaniem, które się tu nasuwa jest: jak by się wydłużyły te czasy gdybyśmy miały do czynienia z milionami przykładów trenujących? Należy się więc zawsze zastanowić, czy nie użyć prostszego modelu o nieco gorszej precyzji zyskując na czasie?
Godne uwagi jest działanie modelu klasyfikacyjnego wykorzystującego maszynę wektorów nośnych. Jak widać prosty liniowy model osiąglą precyzję równą aż 89.44% (czyli jedynie 1.54% gorszą od najlepszego modelu), przy czym całkowity czas trenowania wyniósł tylko 1 sekundę. Biorąc pod uwagę kompromis pomiędzy osiągniętą precyzją a czasem treningu, ten model uznałyśmy za najkorzystniejszy.

Zauważyłyśmy również, że przypadku większości recenzji wszystkie modele rozstrzygały jednogłośnie o przynależności recenzji do danej klasy. Niemniej jednak, w przypadku niektórych recenzji różne modele różnie interpretowały ich wydźwięk i werdykt już

Tablica 3.5: Porównanie jednogłośności algorytmów.

nr rec.	s	rec.	SVM	RF	CNN	LSTM	CNN–LSTM
35067	0	I thought maybe... maybe this could be good. An early appearance by the Re-Animator (Jeffery Combs); many homage's to old horror movies; the Troma label on the front this movie could be a gem! I thought wrong.Frightmare is a boring, overplayed, half assed homage to the fright films of yore. The story is an old one, young people breaking into a house, getting drunk, making love, and tampering with things that shouldn't be tampered with. The oft recycled slasher film formula is used here, this time with a thought to be dead actor named Conrad Radzoff doing the killing. In fact, the performance by the Radzoff's actor Ferdy Mayne is the only redeeming quality of this film. He does the snooty Dracula style character very well. But as for the kids, its not so good, with Combs only having a minimal part.The film lacks entertainment value, and only features one cool character, and one or two scenes that can hold your attention. I do not recommend this film unless you are desperate for something to watch, and this is the only movie left at blockbuster.	0	0	0	0	0
12196	1	Well, if you are one of those Katana's film-nuts (just like me) you sure will appreciate this metaphysical Katana swinging blood spitting samurai action flick.Starring Tadanobu Asano (Vital, Barren Illusion) & Ryu Daisuke (Kagemusha). This samurai war between Heiki's clan versus Genji's clan touch the zenith in the final showdown at Gojo bridge. The body-count is countless.Demons, magic swords, Shinto priests versus Buddhist monks and the beautiful visions provided by maestro Sogo Ishii will do the rest.A good Japanese flick for a rainy summer night.	1	1	1	1	1
49858	1	I liked this movie. That's pretty much all I can say about it. Lou Gossett did a good job, even though I'm still very disappointed in him after all the Iron Eagle movies. And even if I was smiling on the inside when the first main teenager dies (I won't give it away) it was done in a nice, fitting fashion. Pretty much everyone in this movie does a good job, so check it out! It's another one of those movies I found real cheap, so I bought it, and I recommend the same.	1	1	1	0	1
46899	0	There is an inherent problem with commenting or reviewing a film such as this. I remember feeling the same way after disliking Dogma. If you do not like a film that is odd and controversial like Mulholland Dr., you are seen as ńot getting it.Óf course for those who have already seen this film you know that the entire point is not getting it anyway.I have heard from several different sources that the unique and likable aspect of this film is a dream-like quality it has. In other words, the plot isn't structured like other films. With the case of Mulholland Dr., it seems more like an unfocused collage made by a third grade boy who procrastinated until the last second to do his art project. It doesn't make sense, it isn't supposed to, but I know it was to be a TV series at first. It appears Lynch had a stack of unused film and decided to mash it in with a bunch of new stuff. You will notice that toward the end the nudity, sex and foul language increase. All things he would not have filmed for television.For a better film not told in a traditional, linear fashion, rent The Thin Red Line from 1998. That was a great film, this is not.Rating: 2 out of ten	0	0	1	1	1

nie był jednogłośny (Tabela. 3.5). Powyżej znajdują się jedynie wybrane obrazujące to przykłady. Po analizie ustaliłysmy, że niezgodności występują w 138 przypadkach, gdzie prawidłowa wartość sentymentu była negatywna, a któryś z algorytmów przewidział wartość pozytywną, a odwrotna sytuacja miała miejsce w 99 przypadkach.

Inspirację do stworzenia modeli opartych o sieci neuronowe zaczerpnęłyśmy z publikacji [24], dlatego też osiągnięte przez nas wyniki dla odpowadających modeli porównałysmy z tymi, które otrzymali autorzy publikacji. Należy tutaj wspomnieć, że wykorzystali oni nieco inne warunki trenowania i testowania, a mianowicie podzielili wyjściowy zbiór w inny sposób — 50% recenzji użyli do trenowania a kolejne 50% do wykonania testów. U nas natomiast trening odbywał sie na 80% przykładów a testy obejmowały 20%.

Tablica 3.6: Wyniki naszego modelowania vs. wyniki opublikowane przez Minaee et al. [24]. Parametrem porównywanym jest precyzja.

Model	Nasze wyniki	Wyniki Minaee et al.
CNN	89.24%	89.3%
LSTM	90.39%	89%
CNN–LSTM	90.98%	90%

Jak widać udało nam się odtworzyć wyniki z przedmiotowej publikacji. Osiągnęłysmy wyniki porównywalne dla modelu CNN, a dla pozostałych dwóch nawet nieco lepsze niż autorzy. Jednak, tak jak wspominałyśmy, Minaee et al. mieli mniej przykładów żeby sie dobrze nauczyć a zarazem wiecej przykładów, w których ich modele miały szanse się pomylić. Osiągnięte przez nas różnice są de facto znikome, więc finalnie można ogłosić remis.

Rozdział 4

Podsumowanie

Celem ninejszej pracy było stworzenie i zbadanie przydatności klasycznych modeli machine learningowych oraz modeli wykorzystujących sieci neuronowe umożliwiających analizę sentymentu — klasyfikację czy dana recenzja ma pozytywny czy negatywny wydźwięk — recenzji filmowych z portalu IMDB (Internet Movie DataBase). Cel został przez nas osiągnięty. Zoptymalizowałyśmy, wytrenowałyśmy i wykonałyśmy predykcję dla 5 modeli klasyfikacyjnych: (1) Lasu losowego z głębokością drzew 32 (RF), (2) Lasu losowego z głębokością drzew 64 (RF), (3) Maszyny Wektorów Nośnych (SVM), (4) Konwolucyjnej sieci neuronowej (CNN), (5) Rekurencyjnej sieci neuronowej (LSTM). Ponadto zasymulowałyśmy również działanie modelu powstałego z połączenia konwolucyjnej oraz rekurencyjnej sieci neuronowej (CNN–LSTM Ensemble) z sukcesem odtwarzając wyniki opisane w publikacji naukowej pod tytułem *Deep-sentiment: Sentiment analysis using ensemble of cnn and bi-lstm models* z 2019 roku [24].

W wyniku przeprowadzonych przez nas analiz oraz symulacji okazało się, że ze względu na precyzję predykcji, najlepszy rezultat otrzymałyśmy dla połączenia sieci neuronowych CNN–LSTM, natomiast najniższą precyzję uzyskałyśmy dla klasyfikatora wykorzystującego las losowy z drzewami o głębokości 32. Z kolei biorąc pod uwagę czas potrzebny do wytrenowania modelu, najlepszy okazał się model klasyfikacyjny wykorzystujący maszynę wektorów nośnych, a najdłużej zajęło trenowanie dwóch sieci neuronowych dla połączenia CNN–LSTM (obie po 11 milionów trenowanych parametrów).
Całościowo, biorąc pod uwagę oba czynniki, zdecydowanym faworytem okazał się klasyfikator SVM — model liniowy, bardzo prosty, o małej liczbie parametrów. Osiągnął on precyzję aż 89.44%, podczas, gdy czas jego trenowania wyniósł zaledwie 1 sekundę.

Wyraźnie widać, że nie zawsze model najmniej mylący się w predykcjach, jest najlepszy, ponieważ, czas jego trenowania może być zbyt długi i mało korzystny w połączeniu z wynikami osiąganymi przez model. Należy jeszcze podkreślić, że im większy jest wejściowy zbiór danych, tym dłuższe są czasy trenowania modeli, a co za tym idzie również różnice w czasach trenowania pomiędzy poszczególnymi modelami rosną. Gdy z kolei zależy nam na łatwej interpretowalności modelu, to dobrym algorytmem jest np. las losowy, który jest przejrzysty i zawsze można podejrzeć kryteria podziału zbiorów na poszczególne klasy. W naszym przypadku, modele zbudowane z jego pomocą dawały co prawda nieco gorsze rezultaty, ale były to wartości gorsze o kilka procent.

Podsumowując, nie ma rozwiązania idealnego. Żeby osiągać optymalne wyniki, trzeba prawie zawsze iść na kompromis. Należy też zawsze podejmować próby optymalizacji hiperparametrów, gdyż czynność ta może mieć kluczowe znaczenie w dostarczeniu dobrego lub chociazby optymalnego rozwiązania. Ponadto, ważnym faktem, który zawsze trzeba wziąć pod uwagę, jest to, że niektóre algorytmy ulegają łatwemu przetrenowaniu. Wysokie wyniki predykcji modelu, ale fałszywe wyniki nie wnoszą dużo wartości dodanej, a wręcz prowadzą do tego, że wyciągami nieprawidłowe wnioski. Na koniec — warto pamiętać, że każdy model trzeba okresowo testować i dostosowywać jego działanie, gdyż dane, którymi model jest karmiony mogą się zmieniać, a wtedy może się zdarzyć, że model pogorszy swoje działanie, a w skrajnym przypadku straci swoją przydatność.

Załącznik A

Zbiór danych

Wejściowy zbiór danych pochodzi z serwisu IMDB (imdb.com). Jest to popularny zbiór danych wykorzystywany jako benchmark wielu technik przetwarzania języka naturalnego. Składa się z pięćdziesięciu tysięcy recenzji różnych filmów (w języku angielskim). Każda recenzja oznaczona jest jako pozytywna bądź negatywna. Stosunek recenzji pozytywnych do negatywnych wynosi 1:1. Plik wejściowy ma format CSV z dwiema kolumnami: review — zawiera fragment HTML z treścią recenzji, oraz sentiment — zawiera wydźwięk recenzji — odpowiednio positive lub negative. Plik z danymi załączony jest na płycie CD.

Środowisko obliczeniowe i biblioteki

Wszystkie obliczenia opisane w ramach niniejszej pracy wykonano w środowisku Google Colab używając interaktywnego narzędzia Notebook. Ponadto użyto języka Python w wersji 3.6.8. Wszystkie biblioteki wraz z wersjami zostały wyeksportowane przy pomocy komendy `freeze` do pliku `requirements.txt`, co pozwala na pełne i wierne odtworzenie stacku technologicznego, na którym pracowałysmy. Plik z bibliotekami i wersjami załączony jest na płycie CD.

Bibliografia

[1] Mark A Aizerman. Theoretical foundations of the potential function method in pattern recognition learning. *Automation and remote control*, 25:821–837, 1964.

[2] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1–6. Ieee, 2017.

[3] Algolytics. Analityka predykcyjna dla początkujących — część 1.

[4] Arne Holst at Statista. Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2025, 2021.

[5] Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[6] Rahul Chauhan, Kamal Kumar Ghanshala, and RC Joshi. Convolutional neural network (cnn) for image detection and recognition. In *2018 First International Conference on Secure Cyber Computing and Communication (ICSCCC)*, pages 278–282. IEEE, 2018.

[7] Paweł Cichosz. *Systemy uczące się*. WNT Wydawnictwa Naukowo-Techniczne, 2007.

[8] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[9] Daniel Crevier. *Ai: The Tumultuous History Of The Search For Artificial Intelligence*. Basic Books, 1993.

[10] Scikit Learn Documentation. Count vectorizer.

[11] Kavita Ganesan. https://kavita-ganesan.com/text-preprocessing-tutorial/.

[12] Kavita Ganesan. https://kavita-ganesan.com/what-are-stop-words/.

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[14] Kevin Gurney. *An introduction to neural networks*. ROUTLEDGE, 2013.

[15] Aurélien Géron. *Uczenie maszynowe z użyciem Scikit-Learn i TensorFlow. Wydanie II*. O'Reilly, 2020.

[16] Tin Kam Ho. A data complexity analysis of comparative advantages of decision forest constructors. *Pattern Analysis & Applications*, 5(2):102–112, 2002.

[17] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[18] Unstructured Data Itrex Group. https://itrexgroup.com/blog/your-unstructured-data/.

[19] Andreas Kaplan and Michael Haenlein. Siri, siri, in my hand: Who's the fairest in the land? on the interpretations, illustrations, and implications of artificial intelligence. *Business Horizons*, 62(1):15–25, 2019.

[20] Wikipedia: Natural language generation. https://tinyurl.com/wikipedia-nl-generation.

[21] Mirosław Mamczur. Czym są hiperparametry i jak je dobrać?

[22] Margaret Masterman. Semantic message detection for machine translation, using an interlingua. *International Conference on Machine Translation of Languages and Applied Language Analysis*, Conference Paper, 1961.

[23] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[24] Shervin Minaee, Elham Azimi, and AmirAli Abdolrashidi. Deep-sentiment: Sentiment analysis using ensemble of cnn and bi-lstm models. *arXiv preprint arXiv:1904.04206*, 2019.

[25] Wikipedia: Text mining. https://tinyurl.com/wikipedia-text-mining.

[26] Tom M. Mitchell. *Machine learning*. McGraw-Hill, 1997.

[27] Wikipedia: Przetwarzanie Języka Naturalnego. https://tinyurl.com/eng-nlp.

[28] Andrew Ng. Machine learning course on coursera.

[29] David Opitz and Richard Maclin. Popular ensemble methods: An empirical study. *Journal of artificial intelligence research*, 11:169–198, 1999.

[30] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.

[31] Youtube: IBM Research. https://www.statista.com/statistics/871513/worldwide-data-created

[32] Lior Rokach and Oded Z Maimon. *Data mining with decision trees: theory and applications*, volume 69. World scientific, 2007.

[33] Arthur L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959.

[34] Vahid Mirjalili Sebastian Raschka. *Python. Uczenie maszynowe. Wydanie II.* Packt Publishing, 2019.

[35] Sagar Sharma and Simone Sharma. Activation functions in neural networks. *Towards Data Science*, 6(12):310–316, 2017.

[36] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[37] StackExchange. https://stats.stackexchange.com/questions/31066/what-is-the-influence-of-c-i

[38] Alan L. Selman Steven Homer. *Computability and Complexity Theory.* Springer Verlag, 2001.

[39] Suryakanthi Tangirala. Evaluating the impact of gini index and information gain on classification using decision tree classifier algorithm. *International Journal of Advanced Computer Science and Applications*, 11(2):612–619, 2020.

[40] Joseph Weizenbaum. Eliza—a computer program for the study of natural language communication between man and machine. *Commun. ACM*, 9(1):36–45, January 1966.

[41] Wikipedia. Przeuczenie (overfitting).