Drzewo, wyrażenia regularne

Jonatan Kasperczak 341208, Radosław Kasprzak 335202

Czerwiec 2025

Polecenie

Drzewo decyzyjne w zadaniu klasyfikacji miejsc rozcięcia w sekwencji DNA. Należy użyć wyrażenia regularnego w testach. Wyrażenia regularne powinny być wczytane z pliku tekstowego. Dopasowanie wyrażenia sprawdzamy na rozważanej pozycji w sekwencji, np. jeśli rozważamy atrybut numer 3, a wyrażenie to "A.T", to w sekwencji "GGAGT" nastąpi dopasowanie, a w sekwencji "AGTGG" już nie. Więcej informacji o specyfice problemu znaleźć można w: OpisDNA. Dane do pobrania- donory, akceptory. Przed rozpoczęciem realizacji projektu proszę zapoznać się z zawartością: LINK.

1 Temat projektu

Klasyfikacja miejsc rozcięcia w sekwencji DNA przy użyciu drzewa decyzyjnego z testami opartymi na wyrażeniach regularnych.

Celem projektu jest zaimplementowanie drzewa decyzyjnego służącego do klasyfikacji miejsc rozcięcia w sekwencji DNA (ang. *splice sites*). W szczególności, projekt zakłada wykorzystanie wyrażeń regularnych jako testów decyzyjnych w węzłach drzewa.

W klasycznym drzewie decyzyjnym test sprawdza wartość pojedynczego atrybutu (np. „czy cecha $x_i > t$”). W naszym podejściu, atrybutem jest fragment sekwencji DNA (np. 3-znakowe okno), a testem jest dopasowanie do wyrażenia regularnego. Przykładowo, wyrażenie `A.T` dopasowuje się do fragmentu składającego się z litery A, dowolnego znaku, oraz T.

1.1 Główne założenia podejścia i ustalenia z konsultacji

- Dane wejściowe to sekwencje DNA o ustalonej długości, w naszym przypadku 15 i 90 nukleotydów.
- Każda sekwencja jest oznaczona etykietą binarną: 1 (miejsce rozcięcia), 0 (brak miejsca).
- Drzewo decyzyjne podejmuje decyzje w każdym węźle na podstawie:
  - wyrażenia regularnego `regex`,
  - dopasowania X-literowego fragmentu sekwencji `seq[pos:pos+X]` do regexu.
- W preprocesingu danych odrzucamy ciąg znaków od pozycji 0 do pozycji wskazanej na początku pliku, oraz dwa następne znaki, gdyż są takie same dla każdego elementu w zbiorze.
- W celu weryfikacji założenia o odrzuceniu początkowych znaków przeprowadzimy testy dopasowujące wyrażenia regularne do całego ciągu znaków.

2 Wstęp teoretyczny

Miejsca rozcięcia (ang. splice sites) występują na styku egzonów i intronów w pre-mRNA. Biologiczne znaczenie takich miejsc oraz ich konserwatywne otoczenie sprawiają, że możliwa jest ich klasyfikacja na podstawie sekwencji nukleotydów.

3 Środowisko implementacji

Projekt został zaimplementowany w języku Python 3.13. Wykorzystano następujące biblioteki:

- `re` — dopasowanie wyrażeń regularnych,

- `numpy`, `pandas`, `collections`, `typing`, `itertools` — przetwarzanie danych,
- `scikit-learn` — porównawcza implementacja drzewa decyzyjnego oraz metryki klasyfikacji,
- `joblib` - zapis drzewa decyzyjnego do pliku,
- `time` - pomiar czasu wykonywanych testów,
- `matplotlib` — wizualizacja wyników.

4 Struktura projektu

- **main.py** Główny plik uruchamiający eksperyment. Odpowiada za:
  - parsowanie argumentów linii komend (typ danych, ścieżka, wybór implementacji),
  - podział danych na train/val/test,
  - trening modelu i ewaluację,
  - zapis wyników.
- **src/data_loader.py** Parsuje pliki `.dat`, zwraca sekwencje i etykiety jako listy.
- **src/src/custom_regex_generator.py** Generuje n wyrażeń regularnych o zadanej długości z alfabetu A, C, G, T, .. Maksymalna ilość kropek definiowana jest parametrem w, ilość znaków w wyrażeniu regularnym znakiem k.
- **src/decision_tree/model.py** Własna implementacja drzewa decyzyjnego:
  - `class Node` → struktura pojedynczego węzła drzewa, zawierająca informację o indeksie cechy, progu (wartości), wartości klasy liścia i prawdopodobieństwie klas.
  - `fit()` → metoda główna budująca drzewo za pomocą funkcji `_grow_tree()`,
  - `_grow_tree()` → rekurencyjna metoda tworzenia drzewa, wybierająca najlepsze podziały na podstawie `best_split()` oraz kryteriów zatrzymania (max depth, min samples, czystość etykiet),
  - `predict()` → klasyfikuje dane, wywołując rekurencyjną funkcję `_traverse_tree()`,
  - `predict_proba()` → zwraca rozkład prawdopodobieństwa dla każdej klasy na podstawie liścia drzewa,
  - `_most_common_label()` → pomocnicza metoda wybierająca najczęstszą etykietę.
- **src/decision_tree/tree_utils.py** Zawiera funkcję `split_dataset()`, która dla zadanej cechy i progu dzieli dane na cztery części: `X_left`, `y_left`, `X_right`, `y_right`, zgodnie z warunkiem `X[:, feature_idx] <= threshold`. Działa efektywnie na poziomie macierzy numpy.
- **src/decision_tree/metrics_func.py** Funkcja `gini_impurity(y)` oblicza indeks Giniego na podstawie częstości klas w zbiorze `y`. Funkcja `best_split()` przeszukuje kombinacje cech i progów podziałowych, znajdując ten, który minimalizuje średni Gini impurity dla podziału (korzysta z `split_dataset` oraz `gini_impurity`).
- **src/model_trainer.py** Zarządza przepływem treningu i walidacji modelu. Dzieli dane, dopasowuje drzewo i liczy metryki.
- **testing/** Skrypty porównawcze:
  - `custom_vs_sklearn.py` → porównanie wyników własnego drzewa i sklearn,
  - `hyperparameter_tuning.py` → testowanie głębokości drzewa i rozmiaru regexu.
  - `full_vs_window.py` → analiza przeszukiwania całego ciągu znaków i ciągu od wyznaczonej pozycji,
  - `onehot_vs_regex.py` → porównanie klasyfikacji binarnej i klasyfikacji wykorzystującej wyrażenia regularne,
  - `regex_common.py` → skrypt liczący jakoś regexów wykorzystanych do klasyfikacji,
  - `update_regex.py` → skrypt wykorzystywany do aktualizowania listy regexów. Usuwa X wyrażeń, które są najmniej dopasowane do poprawnej klasyfikacji, dodaje Y kolejnych wyrażeń do sprawdzenia

5 Implementacja drzewa decyzyjnego

5.1 Definicja

Drzewo decyzyjne to model predykcyjny w postaci hierarchicznej struktury grafowej, w której każdy węzeł wewnętrzny odpowiada za sprawdzenie wartości jednej cechy (atrybutu), a każda gałąź reprezentuje wynik tego testu (np. „tak”/„nie” lub porównanie z progami). Proces klasyfikacji polega na rozpoczęciu od korzenia i kolejnych przesunięciach w dół drzewa—w zależności od wyniku testu w węźle—aż do dotarcia do liścia, który przypisuje klasę (lub wartość przewidywaną) dla danego przykładu. Poniżej przedstawiono opis własnej implementacji drzewa decyzyjnego, uwzględniający strukturę węzła, proces budowy oraz używane metryki i kryteria podziału.

5.2 Struktura węzła (`Node`)

Każdy węzeł drzewa jest reprezentowany przez obiekt klasy `Node`, który zawiera następujące pola:

- `feature_idx` – indeks cechy, według której wykonujemy test decyzyjny,
- `threshold` – próg kwalifikacji obserwacji jako klasy 1. Domyślnie wartość `threshold` jest równa 0.5, czyli jeśli prawdopodobieństwo przynależności do klasy 1 jest większe lub równe 0.5, przypisywana jest klasa 1.
- `left` i `right` – wskaźniki na poddrzewa (obejmują obiekty typu `Node` lub `None`),
- `value` – przewidywana klasa w węźle liściu (dla węzłów niemających dzieci wartość ta jest ustalana jako najbardziej prawdopodobna klasa),
- `proba` – wektor $[p_0, p_1]$ prawdopodobieństw przynależności do klas 0 i 1, obliczany w węźle liściu.

Węzeł jest liściem wtedy, gdy nie istnieje dalszy podział (tj. nie są już tworzone żadne dzieci), a wówczas pola `left` i `right` pozostają `None`.

5.3 Konstruktor i parametry drzewa

Klasa `DecisionTree` jest inicjowana za pomocą następujących parametrów:

- `max_depth` – maksymalna głębokość drzewa (gdy osiągnięta, węzeł staje się liściem),
- `min_samples` – minimalna liczba przykładów konieczna do podziału (jeśli węzeł zawiera mniej przykładów, staje się liściem),
- `n_feats` – parametr opcjonalny - liczba losowo wybranych cech, spośród których w danym węźle poszukujemy najlepszego podziału (domyślnie liczba wszystkich cech)

5.4 Budowa drzewa (`fit` i `_grow_tree`)

1. **Metoda `fit(X, y)`:**
   - Przyjmuje macierz cech $\mathbf{X} \in \{0,1\}^{n \times m}$ oraz wektor etykiet $\mathbf{y} \in \{0,1\}^n$.
   - Wywołuje metodę pomocniczą `_grow_tree(X, y, depth=0)` i zwraca obiekt `Node` reprezentujący korzeń drzewa.
2. **Metoda `_grow_tree(X, y, depth)`:**
   (a) **Sprawdzenie warunków stopu:**
   - Jeśli `depth` $\geq$ `max_depth` (osiągnięto maksymalną głębokość),
   - LUB jeśli liczba próbek w bieżącym węźle $|\mathbf{y}| <$ `min_samples`,
   - LUB jeśli wszystkie etykiety w bieżącym węźle są identyczne ($\#\{\text{unikalnych etykiet w } y\} = 1$),

   wówczas tworzymy *liść*:
   - Obliczamy liczności klas: counts = Counter(y).
   - Ustalamy $p_0 = \frac{\text{counts}[0]}{|y|}$, $p_1 = \frac{\text{counts}[1]}{|y|}$.
   - Tworzymy węzeł liścia:

$$\texttt{Node}\big(\,\text{value} = \arg\max(\text{counts}),\ \text{proba} = [\,p_0,\,p_1\,]\big)$$

   i zwracamy go.

(b) **W przeciwnym razie:**

i. Losujemy podzbiór indeksów cech: `feat_idxs` $\subset \{0, 1, \ldots, m-1\}$ o rozmiarze `n_feats` (jeśli m to liczba cech).

ii. Wywołujemy `best_split(X, y, feat_idxs, min_samples)`. W wyniku otrzymujemy:

$$(\texttt{best\_feat}, \texttt{best\_thr}) \quad \text{lub} \quad (\texttt{None}, \_,)$$

Zasady działania `best_split` opisano w kolejnym podrozdziale.

iii. Jeśli `best_feat` = `None` (nie znaleziono sensownego podziału), tworzymy liść (analogicznie jak w przypadku warunków stopu).

iv. W przeciwnym razie:

- Rozdzielamy dane, wywołując `split_dataset`$(X, y, \texttt{best\_feat}, \texttt{best\_thr})$, otrzymując (X_L, y_L) i (X_R, y_R).
- Rekurencyjnie wywołujemy:

$$\texttt{left} = \_grow\_tree(X_L, y_L, \texttt{depth}+1), \quad \texttt{right} = \_grow\_tree(X_R, y_R, \texttt{depth}+1).$$

- Tworzymy węzeł wewnętrzny:
`Node(feature_idx=best_feat, threshold=best_thr, left, right)`
i zwracamy go.

5.5 Wybór najlepszego podziału – funkcja `best_split`

Metoda `best_split(X, y, feat_idxs, min_samples)` (plik `src/decision_tree/metrics_func.py`) realizuje następujące kroki:

1. Inicjalizujemy zmienne:

$$\texttt{best\_feat} = \texttt{None}, \quad \texttt{best\_thr} = \texttt{None}, \quad \texttt{best\_impurity} = +\infty.$$

2. Dla każdej cechy $idx \in$ `feat_idxs` oraz dla każdego możliwego progu $thr \in \{0, 1\}$:

(a) Wywołujemy `split_dataset`(X, y, idx, thr), które zwraca:

$$(X_L, y_L, X_R, y_R),$$

gdzie

$$X_L = \{\, x \in X : x[idx] \le thr \}, \quad X_R = \{\, x \in X : x[idx] > thr \}.$$

(b) Sprawdzamy warunek minimalnej liczebności: jeśli $|y_L| <$ `min_samples` lub $|y_R| <$ `min_samples`, pomijamy ten podział.

(c) W przeciwnym razie obliczamy impurity[1]:

$$G_L = G(y_L), \quad G_R = G(y_R),$$

$$G_{\text{total}} = \frac{|y_L|}{|y|} G_L + \frac{|y_R|}{|y|} G_R.$$

(d) Jeśli $G_{\text{total}} <$ `best_impurity`, aktualizujemy:

$$\texttt{best\_impurity} = G_{\text{total}}, \quad \texttt{best\_feat} = idx, \quad \texttt{best\_thr} = thr.$$

3. Po przeszukaniu wszystkich kombinacji zwracamy (`best_feat`, `best_thr`) lub (`None`, `None`) jeżeli nie znaleziono żadnego dopuszczalnego podziału.

5.6 Miara podziału – wskaźnik Giniego

W implementacji korzystamy z funkcji `gini_impurity(y)` (plik `src/decision_tree/metrics_func.py`), która dla wektora etykiet y oblicza:

$$G(y) = 1 - \sum_{c \in \{0,1\}} p_c^2, \quad p_c = \frac{\#\{i : y_i = c\}}{|y|}.$$

Wskaźnik Giniego rośnie, gdy rozkład etykiet jest bardziej wymieszany, a maleje, gdy zestaw staje się jednorodny.

[1] Miara Giniego jest definiowana w metodzie `gini_impurity(y)`.

5.7 Predykcja

- Metoda `predict(X)` – dla każdej próbki $x \in X$ wywołuje rekurencyjnie funkcję `_traverse_tree(x, root)`, która:
  1. Jeśli bieżący węzeł jest liściem (`left` = `right` = `None`), zwraca wartość `value`.
  2. W przeciwnym razie:
     - Pobiera idx = `feature_idx` oraz thr = `threshold`.
     - Jeżeli $x[idx] \leq thr$, schodzi do `left`, w przeciwnym razie do `right`.
- Metoda `predict_proba(X)` – analogicznie przechodzi po drzewie i zwraca w węźle liścia wektor `proba` = $[p_0, p_1]$.

5.8 Procedura budowy drzewa decyzyjnego

Proces trenowania drzewa decyzyjnego w naszej implementacji przebiega według następującego schematu:

1. W funkcji `fit()` rozpoczyna się rekurencyjne budowanie drzewa, poprzez wywołanie metody pomocniczej `_grow_tree()`.
2. Dla danego węzła, metoda `_grow_tree()` sprawdza, czy spełniony jest warunek zatrzymania:
   - osiągnięto maksymalną głębokość drzewa,
   - liczba próbek w węźle jest mniejsza niż `min_samples`,
   - wszystkie etykiety są identyczne.

   Jeśli tak — tworzony jest liść drzewa, przechowujący najczęstszą klasę oraz rozkład prawdopodobieństwa klas.
3. W przeciwnym razie, losowany jest podzbiór indeksów cech (`feat_idxs`), z których wybierany będzie najlepszy podział.
4. Funkcja `best_split()` iteruje po parach (cecha, próg). Dla każdej kombinacji:
   - wywoływana jest funkcja `split_dataset()`, która dzieli dane na dwie części według warunku: `X[:, feature_idx] <= threshold`,
   - obliczany jest indeks Giniego dla podziału,
   - zapamiętywana jest kombinacja o najmniejszym impurity.
5. Jeśli nie znaleziono sensownego podziału (np. za mało próbek w podzbiorach), również tworzony jest liść.
6. W przeciwnym wypadku, na podstawie najlepszego podziału:
   - dzielone są dane,
   - rekurencyjnie wywoływana jest funkcja `_grow_tree()` dla lewej i prawej gałęzi,
   - tworzony jest węzeł drzewa z zapamiętanym indeksem cechy, progiem i wskaźnikami do dzieci.

6 Zbiory danych

Użyto dwóch zbiorów danych dostępnych na stronie przedmiotu:

- `spliceDTrainKIS.dat` — klasyfikacja miejsc donorowych,
- `spliceATrainKIS.dat` — klasyfikacja miejsc akceptorowych.

Każdy przykład to sekwencja DNA o stałej długości oraz przypisana etykieta:

- 1 — miejsce rozcięcia (pozytywna klasa),
- 0 — brak miejsca rozcięcia.

W pierwszej linii każdego zbioru danych podana została pozycja na podstawie której można zidentyfikować rozcięcie lub brak.

Zasadniczą różnicą pomiędzy wymienionymi dwoma zbiorami danych jest długość każdego elementu. Zbiór akceptorów posiada dłuższe ciągi nukleotydów, co pozwala na korzystanie z dłuższych wyrażeń regularnych w celu dopasowania.

Oba zbiory danych mają niezbalansowany rozkład klas. Dla zbioru akceptorów oraz donorów liczność klasy 1 to około 20%. Wpływa to negatywnie na proces uczenia się.

Table 1: Rozkład klas w zbiorze danych

Zbiór danych	**Klasa 0 (brak rozcięcia)**	**Klasa 1 (rozcięcie)**
Donory	4140	1116
Akceptory	4672	1116

6.1 Założenia bazowe eksperymentów

Na potrzeby dalszych badań i testów jakości klasyfikacji zaprojektowaliśmy trzy bazowe warianty eksperymentów, oparte na różnych metodach reprezentacji sekwencji DNA i sposobie dopasowywania wzorców. Każdy z tych wariantów stanowi odrębny schemat ekstrakcji cech wejściowych, na których budowane były modele klasyfikacyjne. W oparciu o te trzy założenia prowadziliśmy dalsze analizy porównawcze.

1. **Regex window** W tym wariancie cechy były tworzone poprzez dopasowanie wyrażeń regularnych (regexów) do **wyciętego okna** sekwencji DNA, obejmującego fragment od pozycji granicznej.
   Ograniczenie dopasowania do okna wynikało z podanej w zbiorze dancyh pozycji, że istotne wzorce determinujące obecność miejsca rozcięcia znajdują się w bezpośrednim sąsiedztwie tego miejsca. Było to podejście domyślne w projekcie.

2. **One-hot window**
   W tym wariancie zastosowano binarne kodowanie (one-hot) wyciętego okna sekwencji DNA. Każdy znak sekwencji w oknie został zakodowany jako wektor binarny o długości 4, reprezentujący litery `A`, `C`, `G`, `T`. Model klasyfikacyjny (drzewo decyzyjne) był budowany na podstawie tak zakodowanych wektorów cech. Analogicznie jak w pierwszym wariancie, przetwarzano tylko okno od pozycji granicznej do końca ciągu znaków.

3. **Regex full**
   W tym wariancie wyrażenia regularne były dopasowywane do **całej sekwencji DNA**, bez ograniczenia do wyciętego okna. Dla każdego regexu iterowano po sekwencji znak po znaku (przesunięcie sliding window) od początku do końca sekwencji, sprawdzając czy regex dopasowuje się w którejkolwiek pozycji. Celem tego wariantu było sprawdzenie, czy uwzględnienie pełnej sekwencji DNA (poza obszarem okna) wnosi dodatkową wartość informacyjną dla klasyfikacji.

Powyższe trzy warianty stanowiły **bazowe założenia**, na których oparte zostały dalsze eksperymenty, analizy porównawcze oraz testy wpływu różnych parametrów modelu na skuteczność klasyfikacji.

W trakcie preprocessingu dokonaliśmy następujące modyfikacje na zbiorach danych:

- W obu zbiorach zostały wycięte znaki od początku elementu do pozycji określonej w pierwszej linii pliku.
- Zaobserwowano, że dla zbioru akceptorów i donorów dwa następne znaki są takie same dla wszystkich elementów. Znaki zostały usunięte, gdyż nie wnosiły żadnej wartości dodanej do zadania.

Dane zostały podzielone na zbiór treningowy, walidacyjny oraz testowy (65/15/20).

Ważną obserwacją jest różnica w długości elementów w analizowanych zbiorach. Zbiór akceptorów zawiera elementy dłuższe niż zbiór donorów. W kontekście naszego projektu zwiększa to liczbę wymaganych testów oraz zmniejsza szanse na znalezienie optymalnego zestawu wyrażeń regularnych.

Przykład testu

Dla sekwencji `GGAGT` i testu "pozycja 1, regex = A.T":

- fragment od pozycji 1 to `GAG` → nie pasuje do `A.T`,
- zatem wynik testu to: niedopasowanie.

7 Przeprowadzone badania

7.1 Badanie 1: Porównanie własnej implementacji drzewa z `sklearn`

Cel badania: Ocena jakości klasyfikacji własnej implementacji drzewa decyzyjnego w porównaniu z `DecisionTreeClassifier` ze scikit-learn. Obie implementacje korzystają z wyrażeń regularnych.

Opis: W tym badaniu testy wykonywane są na tych samych zbiorach trenujących i testujących.

Wyniki:

Table 2: Porównanie: własne drzewo vs. `sklearn` na zbiorze donor

Model	Accuracy	Precision (klasa 1)	Recall (klasa 1)
Własna implementacja	0,90	0,76	0,77
`sklearn`	0,90	0,78	0,73

Table 3: Porównanie: własne drzewo vs. `sklearn` na zbiorze akceptor

Model	Accuracy	Precision (klasa 1)	Recall (klasa 1)
Własna implementacja	0,78	0,37	0,19
`sklearn`	0,79	0,37	0,12

Table 4: Confusion Matrix dla zbiorów donorów i akceptorów

Zbiór danych	Confusion Matrix
Donor	`TN=793 FP=36` `FN=56 TP=167`
Acceptor	`TN=883 FP=52` `FN=169 TP=54`

Dla celów interpretacji oraz weryfikacji implementacji zastosowano dwie techniki prezentacji drzewa decyzyjnego:

1. *Grafika (sklearn)* – użyto wbudowanej funkcji `plot_tree` z biblioteki scikit-learn, aby wygenerować diagram drzewa (Rys. 1).
2. *Tekst (custom)* – zaimplementowano metodę `export_text`, która drukuje strukturę drzewa w postaci wciętego tekstu w terminalu, co ułatwia szybkie prześledzenie reguł decyzyjnych własnej implementacji.

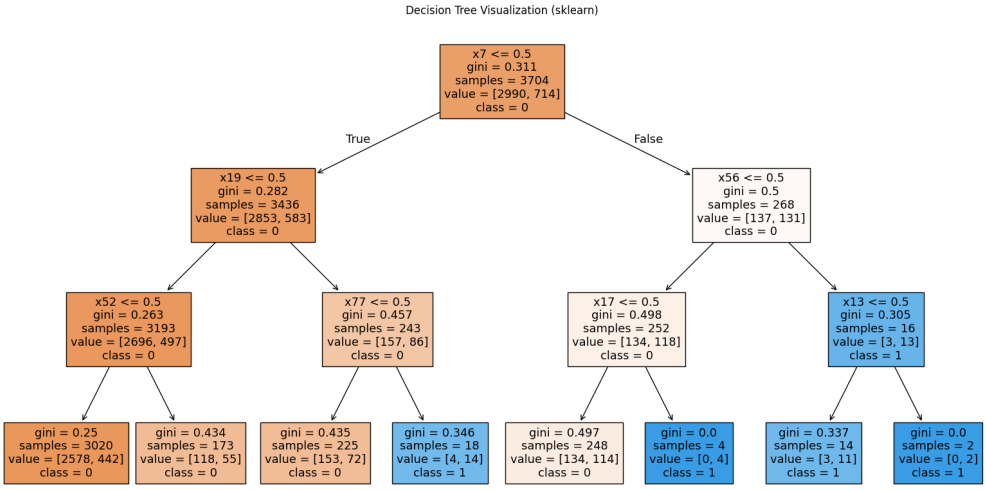

Figure 1: Graficzna wizualizacja drzewa decyzyjnego wygenerowana przez `sklearn.tree.DecisionTreeClassifier`.

Przykład tekstowej wizualizacji (custom):

```
| If x7 == 0:
   | If x19 == 0:
      | If x52 == 0:
         | Predict: 0 (proba: [0.854, 0.146])
        Else (if x52 == 1):
         Predict: 0 (proba: [0.682, 0.318])
     Else (if x19 == 1):
      If x77 == 0:
         | Predict: 0 (proba: [0.680, 0.320])
        Else (if x77 == 1):
         Predict: 1 (proba: [0.222, 0.778])
   Else (if x7 == 1):
    If x2 == 0:
      | If x17 == 0:
         | Predict: 0 (proba: [0.540, 0.460])
        Else (if x17 == 1):
         Predict: 1 (proba: [0.000, 1.000])
      Else (if x2 == 1):
       If x13 == 0:
         | Predict: 1 (proba: [0.214, 0.786])
        Else (if x13 == 1):
         Predict: 1 (proba: [0.000, 1.000])
```

Powyższe zestawienie (Rys. 1 vs. tekstowy wydruk) pozwala na szybkie porównanie i weryfikację, że własna implementacja (`custom`) zachowuje analogiczną strukturę decyzji jak klasyfikator ze scikit-learn.

Wnioski:
Własna implementacja drzewa decyzyjnego jest poprawna i można dalej jej używać do kolejnych badań.

7.2 Badanie 2: Wpływ kodowania one-hot oraz oversamplingu

Cel badania: Ocena skuteczności klasyfikatora przy binarnej klasyfikacji dla sekwencji DNA. Analizowano wpływ zastosowania oversamplingu klasy mniejszościowej na jakość klasyfikacji.

Opis: Kodowanie one-hot zastosowane w naszym projekcie polegało na binarnym zakodowaniu liter sekwencji DNA od wskazanej pozycji startowej (zgodnie z informacją w pliku danych oraz po usunięciu redundantnych informacji) aż do końca sekwencji. Każda pozycja w sekwencji była reprezentowana jako wektor binarny o długości 4, odpowiadający literom `A`, `C`, `G`, `T`. Dla każdej pozycji dokładnie jeden bit w wektorze był ustawiony na 1 (pozostałe były zerami).

Dzięki temu pełna sekwencja (od pozycji startowej do końca) była przekształcana w wektor cech, w którym każda litera zajmowała dokładnie 4 bity. Takie podejście umożliwiało klasyfikatorowi analizę informacji o konkretnych literach w określonych pozycjach sekwencji, bez wprowadzania założeń o kolejności lub podobieństwie liter.

Zastosowane kodowanie zapewniało pełną informację o strukturze sekwencji w analizowanym fragmencie, a jednocześnie pozwalało stosować standardowe metody uczenia maszynowego przystosowane do danych numerycznych.

Wyniki:

Table 5: Porównanie wyników klasyfikacji dla różnych sposobów kodowania sekwencji dla akceptorów

Kodowanie	**Accuracy**	**Precision (klasa 1)**	**Recall (klasa 1)**
One-hot	0,68	0,22	0,27
Regex-based	0,80	0,47	0,18

Table 6: Porównanie wyników klasyfikacji (one-hot, z oversamplingiem i bez) dla donorów

Wariant	**klasyfikacja**	**Accuracy**	**Precision (klasa 1)**	**Recall (klasa 1)**
Bez oversamplingu	one-hot	0,87	0,67	0,79
Z oversamplingiem	one-hot	0,85	0,62	0,81
Bez oversamplingu	regex	0,90	0,80	0,75
Z oversamplingiem	regex	0,74	0,42	0,58

Wnioski:
Dobór wyrażeń regularnych jest kluczowym czynnikiem wpływającym na wyniki. Dobranie odpowiednich regexów pozwala zmniejszyć rozmiar drzewa w porównaniu z klasyfikacją binarną zachowując bardzo zbliżoną jakośc klasyfikacji. W przypadku dobrania niedokładnych regexów i niedopasowania ich do zbioru, wyniki będą znacząco odbiegać od klasyfikacji binarnej. W dalszych badaniach wykorzystywano klasyfikacje opartną na wyrażeniach regularnych zgodnie z założeniami projektu.

Przeprowadziliśmy eksperymenty z zastosowaniem techniki oversamplingu (zwiększenia liczności klasy mniejszościowej) wykazaliśmy, że oversampling prowadzi do zauważalnego zwiększenia wartości `Recall` dla klasy 1, jednak kosztem spadku metryk `Precision` oraz ogólnej `Accuracy`.

Efekt ten jest szczególnie widoczny w przypadku modeli z wykorzystaniem cech opartych o wyrażenia regularne, gdzie oversampling pozwolił zwiększyć `Recall` z 0,37 do 0,57, ale jednocześnie spowodował spadek `Precision` z 0,83 do 0,76. W przypadku modeli z reprezentacją one-hot efekt był mniejszy, ale nadal zauważalny — `Recall` wzrósł z 0,79 do 0,81, przy niewielkim spadku `Precision`.

Otrzymaliśmy wyniki, które potwierdzają, że oversampling skutecznie zwiększa zdolność modelu do wykrywania przykładów klasy 1, jednak prowadzi również do zwiększenia liczby fałszywych alarmów (obniżenie `Precision`). W związku z tym decyzja o stosowaniu oversamplingu powinna być zależna od konkretnego zastosowania i priorytetów modelu — jeżeli celem jest minimalizacja liczby pominiętych przypadków klasy 1, stosowanie oversamplingu jest uzasadnione, natomiast gdy istotna jest wysoka precyzja klasyfikacji, należy rozważyć inne techniki zbalansowania danych.

7.3 Badanie 3: Wpływ zbioru wyrażeń regularnych do jakości klasyfikacji

Cel badania: Ocena jakości klasyfikacji przy użyciu testów opartych na wyrażeniach regularnych. Porównano najlepszy wybrany podzbiór regexów oraz słabszy wariant.

Opis: Proces szukania najlepszego zbioru wyrażeń regularnych osobno dla donorów i akceptorów przeprowadzono około 2 500 iteracji, badając blisko 15 000 wyrażeń regularnych wygeneowanch losowo przy pomocy skryptu *custom_regex_generator.py*. Wyrażenia regularne posiadały różne długości, różne ilości kombinacji kilku nukleotydów oraz wildcardów. Dodatkowym testem wykonanym na początku optymalizacji listy wyrażeń regularnych było sprawdzenie jak duży zbiór regex jest optymalny. Zwiększenie zbioru wyrażeń do 200 nie poprawiało klasyfikacji, a zwiększało czas ewaluacji dwukrotnie, zmniejszanie zbioru do 60 lub 80 wyrażeń nieproporcjonalnie osłabiało jakość klasyfikacji o 20%-50%.

- Testowano 110 regexów w każdej iteracji.
- Selekcjonowano najlepszy podzbiór 100 regexów.
- Porównanie wyników dla najlepszego zbioru i dla celowo dobranego słabszego zbioru regexów.

Wyniki:

Table 7: Porównanie najlepszych i gorszych regexów dla klasy donor

Podzbiór regexów	Accuracy	Precision (klasa 1)	Recall (klasa 1)
Najlepszy podzbiór	0,91	0,82	0,7
Słabszy podzbiór	0,81	0,69	0,16

Wnioski: Dobór odpowiednich wyrażeń regularnych ma kluczowe znaczenie dla jakości klasyfikacji. Wyniki eksperymentów jednoznacznie wskazują, że zastosowanie niedopasowanego lub zbyt ubogiego zbioru regexów prowadzi do drastycznego spadku skuteczności klasyfikatora — różnice sięgały nawet 20%-50%. Z kolei rozsądnie dobrany zbiór około 100 dobrze dopasowanych wyrażeń pozwalał osiągnąć znacznie lepsze wyniki przy akceptowalnym czasie obliczeń.

Proces selekcji regexów okazał się zatem etapem krytycznym. Nieoptymalny dobór prowadził do poważnych błędów klasyfikacji, podczas gdy odpowiednio skonstruowany zestaw regexów umożliwiał wydobycie istotnych wzorców z sekwencji DNA. Wynika z tego, że skuteczność całego modelu zależy w dużej mierze właśnie od jakości i trafności użytych wyrażeń regularnych.

7.4 Badanie 4: Porównanie pełnego ciągu i okna regexów (Window vs Full)

Cel badania:
Ocenić wpływ sposobu ekstrakcji cech — pełnego przeszukiwania sekwencji DNA (full) oraz ekstrakcji w oknie (window) — na skuteczność klasyfikacji.

Opis: Dla każdego typu danych (`donor`, `acceptor`) oraz dla obu implementacji drzewa decyzyjnego (`custom`, `sklearn`) przeprowadzono testy porównujące dwie strategie ekstrakcji cech:

- `full` — dopasowanie wyrażeń regularnych w całej sekwencji od początku,
- `window` — dopasowanie tylko od pozycji podanej przy zbiorze danych.

Porównano jakość klasyfikacji mierzona metrykami: `accuracy`, `precision`, `recall`, `F1-score`.
Wyniki:

Table 8: Porównanie: pełny ciąg vs. okno (Window vs Full)

Data type	Impl.	Acc F	Acc W	F1 F	F1 W	Prec F	Prec W	Rec F	Rec W
Acceptor	Custom	0,66	0,81	0,20	0,33	0,19	0,51	0,22	0,24
Acceptor	Sklearn	0,68	0,81	0,20	0,21	0,20	0,52	0,21	0,13
Donor	Custom	0,79	0,91	0,50	0,78	0,52	0,82	0,48	0,75
Donor	Sklearn	0,79	0,91	0,44	0,76	0,50	0,85	0,39	0,69

Wnioski:
Ekstrakcja cech ograniczona do okna wokół pozycji granicznej (`window`) daje wyraźnie lepsze wyniki niż przeszukiwanie pełnego ciągu sekwencji DNA. Dla obu typów danych (`donor`, `acceptor`) oraz dla obu implementacji (`custom`, `sklearn`) uzyskano wyższą `accuracy` oraz `F1-score` przy stosowaniu okna.

W przypadku zbioru `donor`, poprawa jest szczególnie wyraźna — np. dla drzewa `custom` `Recall` wzrósł z 0,48 do 0,75. Dla zbioru `acceptor` wzrost jakości klasyfikacji jest również zauważalny.

Wynika to z powodu trenowania zbioru wyrażeń regularnych do krótszych ciągów znaków.

7.5 Badanie 5: Testowanie hiperparametrów drzewa decyzyjnego

Cel badania:
Ocena wpływu hiperparametrów drzewa decyzyjnego na jakość klasyfikacji na danych z najlepszym podzbiorem regexów.

Opis: Celem badania jest określenie, w jaki sposób ustawienia dwóch kluczowych hiperparametrów drzewa decyzyjnego: `max_depth` (maksymalna głębokość drzewa) oraz `min_samples_leaf` (minimalna liczba próbek w liściu), wpływają na skuteczność klasyfikacji miejsc rozcięcia w sekwencjach DNA.

Dane wejściowe są reprezentowane jako macierz cech binarnych, opartych na wcześniej wyselekcjonowanym zestawie 100 regexów. Dla każdego eksperymentu budowane jest drzewo decyzyjne przy użyciu wybranej kombinacji wartości hiperparametrów.

Zakres badania:

- Parametr `max_depth`: wartości od d_{min} do d_{max} (np. 3, 5, 10, 15, 20, 25, 30).
- Parametr `min_samples_leaf`: wartości od s_{min} do s_{max} (np. 2, 5, 10, 20, 30).

Dla każdej konfiguracji zapisywane są wyniki, w celu wybrania najlepszej klasyfikacji na koniec.
Wyniki:

Table 9: Wpływ hiperparametrów na jakość klasyfikacji

max_depth	min_samples_leaf	Accuracy	Precision (klasa 1)	Recall (klasa 1)
30	2	0,80	0,42	0,17
10	10	0,80	0,48	0,12
10	2	0,80	0,42	0,11
10	20	0,81	0,51	0,09
15	20	0,81	0,51	0,09
20	20	0,81	0,51	0,09
30	20	0,81	0,51	0,09
3	30	0,81	0,52	0,08
10	30	0,81	0,49	0,08
15	30	0,81	0,49	0,08
20	30	0,81	0,49	0,08
30	30	0,81	0,49	0,08
5	30	0,81	0,49	0,08
5	10	0,81	0,56	0,08
5	20	0,81	0,54	0,08
5	5	0,81	0,51	0,06

Wnioski:
Parametry najlepszego modelu:

- Donor: `max_depth`=10, `min_samples`=2, `Accuracy`=0,91, `Precision`=0,86, `Recall`=0,69
- Akceptor: `max_depth`=30, `min_samples`=2, `Accuracy`=0,81, `Precision`=0,51, `Recall`=0,25

7.6 Badanie 6: Ocena predykcyjnej wartości poszczególnych wyrażeń regularnych

Cel badania:
Wybranie najlepszego podzbioru regexów do budowy modelu klasyfikacji miejsc rozcięcia w sekwencjach DNA.

Opis: Do oceny jakości poszczególnych regexów wykorzystaliśmy dane z pliku `scores.tsv`, który zawiera dla każdego regexu liczbę dopasowań do przykładów klasy 1 (`matches_class1`) oraz klasy 0 (`matches_class0`). Na tej podstawie dla każdego regexu obliczono następujące metryki:

- **Precision (klasa 1)**:
$$\text{Precision} = \frac{matches\_class1}{matches\_class1 + matches\_class0}$$
- **Recall (klasa 1)**:
$$\text{Recall} = \frac{matches\_class1}{total\_class1}$$
gdzie $total\_class1$ to liczba przykładów klasy 1 w zbiorze danych.
- **Score** (funkcja celu dla selekcji regexów):
$$\text{Score} = matches\_class1 - \alpha \cdot matches\_class0$$
gdzie $\alpha = 1.5$ to parametr karzący dopasowania do klasy 0,

Proces selekcji:

1. Wyfiltrowano regexy o zbyt niskiej precyzji (`Precision < 0,5`) oraz o bardzo małej liczbie dopasowań do klasy 1 (`matches_class1 < 5`).
2. Dla pozostałych regexów obliczono `Score` i posortowano je malejąco.
3. Wprowadzono mechanizm usuwania funkcjonalnych duplikatów regexów:
    - Spośród regexów odpowiadających tej samej formie kanonicznej zachowywano tylko najkrótszy.
4. Do finalnego zbioru wybierano `100` regexów o najwyższym `Score`.
5. Wyniki zapisano do pliku `best_regexes_metrics.tsv`

Table 10: Najlepiej predykujące regexy dla zbioru donorów

Regex	Precision (klasa 1)	Recall (klasa 1)	Score
`.AG`	0,76	0,64	372
`AAG.`	0,91	0,27	258
`GAG.`	0,80	0,35	241
`A.G`	0,76	0,38	220,5
`AA.`	0,74	0,34	179,5
`AAGT`	0,98	0,11	121,5
`GA.T..`	0,76	0,19	111,5
`AA.T`	0,83	0,13	104,5
`G[CA][AGT]T`	0,72	0,19	90

Table 11: Najlepiej predykujące regexy dla zbioru acceptorów

Regex	Precision (klasa 1)	Recall (klasa 1)	Score
`GTAT`	0,83	0,02	14
`GTG..A`	0,72	0,02	9,5
`GTATG.`	0,90	0,01	7,5
`GTAT[GCA]A`	0,86	0,01	4,5
`GAAAA`	0,75	0,01	4,5
`GTA...`	0,62	0,04	4
`GTGT.A`	0,83	0,00	3,5
`G.AT.A`	0,67	0,01	2,5
`GTGA[ATG][GA]`	0,65	0,01	2

Wnioski:

Analizując wykazaliśmy, że pojedyncze regexy o precyzyjnej strukturze (np. `AAG.`, `.AG`) znacząco poprawiają jakość klasyfikacji. Równocześnie wyraźnie wskazaliśmy, że zbyt ogólne wzorce negatywnie wpływają na precyzję modelu. Selekcja regexów wspomagana analizą dopasowań pozwala na skuteczne ograniczenie nadmiernego dopasowania oraz zwiększenie różnorodności cech w modelu.

Analizując najlepsze regexy dla zbiorów donorów i akceptorów wykazaliśmy, że najskuteczniejsze wzorce to te o stosunkowo krótkiej długości — zwykle nieprzekraczającej 6 znaków (litery lub grupy znaków). Regexy o długości poniżej 6 znaków charakteryzowały się wysoką precyzją oraz stosunkowo dobrym pokryciem klasy 1 (wysokim recall), co miało bezpośredni wpływ na poprawę skuteczności klasyfikacji.

Podczas testowania regexów o większej długości zaobserwowaliśmy, że liczba możliwych kombinacji znacząco rośnie, co powoduje, że znalezienie optymalnych wzorców staje się trudniejsze. Regexy dłuższe często dopasowują się do bardzo ograniczonego podzbioru sekwencji lub do pojedynczych przykładów, przez co nie poprawiają ogólnej skuteczności modelu. Ponadto dłuższe wyrażenia regularne mają tendencję do przeuczania modelu (overfitting), szczególnie gdy są dopasowane do rzadkich lub nietypowych fragmentów sekwencji DNA.

W związku z powyższym, dla naszych zbiorów najlepsze wyniki uzyskaliśmy, stosując krótkie, dobrze uogólniające się regexy, które jednocześnie zapewniają wysoką interpretowalność biologiczną modelu.

8 Wyniki badań i ich analiza

- **Poprawność implementacji:** Własna implementacja drzewa decyzyjnego działającego na cechach opartych o wyrażenia regularne jest poprawna i skutecznie klasyfikuje sekwencje DNA.
- **Znaczenie wyrażeń regularnych:** Dobór odpowiedniego zestawu wyrażeń regularnych ma kluczowe znaczenie dla jakości klasyfikacji. Eksperymenty wykazały, że odpowiednio dobrane krótkie regexy (o długości do 6 znaków) pozwalają osiągnąć bardzo dobre wyniki klasyfikacji, natomiast użycie źle dopasowanych lub nadmiernie ogólnych regexów znacząco pogarsza wyniki.
- **Proces selekcji regexów:** Proces iteracyjnego ulepszania listy regexów (przy pomocy mechanizmu oceny i selekcji) okazał się niezwykle skuteczny. Finalny podzbiór 100 regexów wybrany po kilkuset iteracjach zapewniał znacznie lepsze wyniki niż zestawy losowe lub nietrafione. Selekcja regexów jest zatem jednym z kluczowych etapów konstrukcji skutecznego modelu.
- **Kodowanie one-hot vs regexy:** Kodowanie one-hot pozwala na poprawną klasyfikację, jednak stosowanie dobrze dobranych regexów umożliwia zbudowanie modeli równie skutecznych lub lepszych, przy mniejszym rozmiarze drzewa i lepszej interpretowalności cech. Regexy pozwalają też na wprowadzenie wiedzy biologicznej (np. znanych wzorców sekwencji).
- **Wpływ oversamplingu:** Oversampling klasy mniejszościowej (klasy 1) skutecznie zwiększa czułość modelu (`Recall`), co jest szczególnie istotne w zadaniach wykrywania miejsc rozcięcia. Jednak stosowanie oversamplingu wiąże się ze spadkiem precyzji i dokładności. Dlatego decyzja o jego zastosowaniu powinna być uzależniona od priorytetów konkretnego zastosowania.
- **Hiperparametry drzewa:** Ustalono, że dla zbioru donorów umiarkowana głębokość drzewa (np. `max_depth` ≈ 10) i niski próg minimalnej liczby próbek w liściu (`min_samples_leaf` $= 2$) pozwalają na najlepszy kompromis pomiędzy precyzją a czułością. W przypadku akceptorów optymalne ustawienia były bardziej zróżnicowane i wyniki ogólnie były słabsze, co wynikało m.in. z trudniejszego charakteru zbioru i dłuższych sekwencji.

- **Biologiczna interpretacja:** Analiza najlepszych regexów wskazała, że skuteczne wzorce często odpowiadają biologicznie istotnym fragmentom sekwencji (np. motywom AG, AAG, GAG), co potwierdza sensowność stosowanego podejścia i umożliwia interpretację modelu w kontekście biologii sekwencji DNA.

9 Nabyte umiejętności

- **Praktyczne zastosowanie wyrażeń regularnych w uczeniu maszynowym** Nauczyliśmy się, jak wyrażenia regularne mogą być wykorzystane do budowy cech w klasyfikatorze, szczególnie w zadaniach związanych z analizą sekwencji biologicznych (DNA). Zobaczyliśmy, że odpowiedni dobór i selekcja regexów ma kluczowy wpływ na skuteczność modelu.

- **Znaczenie dopasowania modelu do danych**
  Przekonaliśmy się, że np. model oparty na regexach działa lepiej, gdy skupiamy się na istotnym biologicznie fragmencie sekwencji (okno), a dopasowanie do pełnej sekwencji często wnosi szum i pogarsza wyniki.

- **Budowa własnego modelu drzewa decyzyjnego**
  Nauczyliśmy się, jak zaimplementować własne drzewo decyzyjne od podstaw, jak działa proces budowania drzewa (`best_split`, `gini impurity`, `grow_tree`), oraz jak kontrolować jego parametry (`max_depth`, `min_samples`).

- **Techniki optymalizacji i selekcji cech**
  Zobaczyliśmy, jak ważny jest proces iteracyjnej selekcji cech (regexów), jak można automatycznie oceniać jakość regexów i eliminować te nieprzydatne, co znacząco poprawia skuteczność klasyfikatora.

- **Automatyzacja eksperymentów**
  Nauczyliśmy się automatyzować proces prowadzenia eksperymentów (iteracyjne uruchamianie `main.py`, `regex_common.py`, `select_top_regexes.py`) oraz jak organizować wyniki eksperymentów i ich raportowanie.