Systemy Sztucznej Inteligencji

dokumentacja projektu Drzewo decyzyjne

Mikołaj Molenda, grupa 2/3
Dominik Meisner, grupa 2/3
Bartłomiej Janoszka, grupa 2/3

30 maja 2025

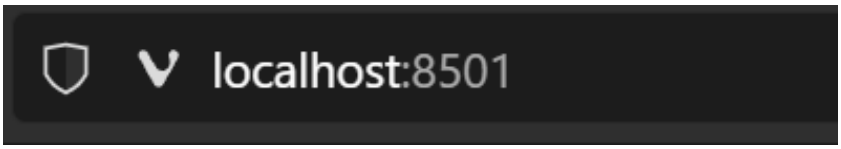
Część I

Opis programu

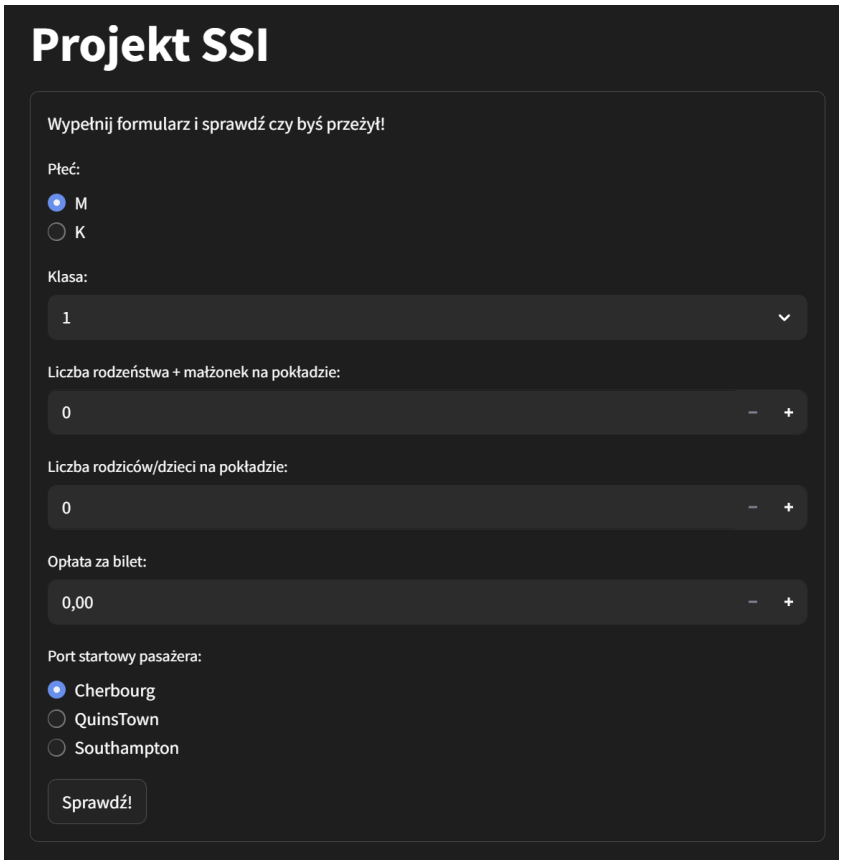
Stworzyliśmy model **DecisionTree** z warunkiem stopu opartym na **zbiorach miękkich**. Nasz model został zaprojektowany i wytrenowany pod dataset ludzi którzy przeżyli katastrofę Tytanic'a. W naszym projekcie użytkownik może wypełnić formularz odpowiednimi danymi, następnie taki formularz jest przesyłany i analizowany przez nasz wytrenowany model i udziela odpowiedzi, czy taka potencjalna osoba przeżyłaby katastrofę.

Instrukcja obsługi

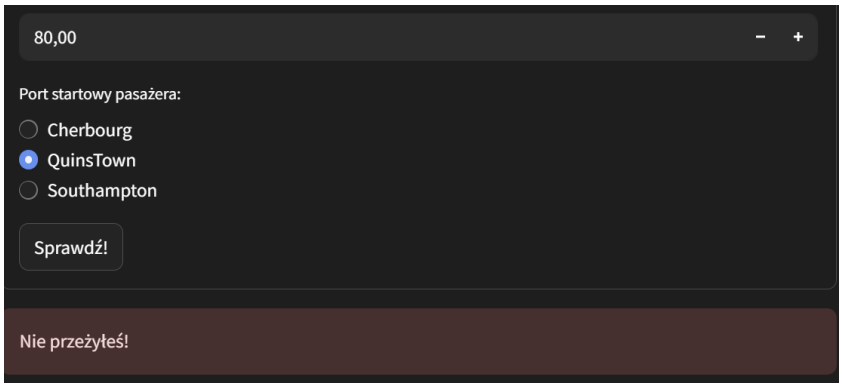
Program uruchamiany jest przy pomocy streamlit, który uruchamia aplikacje na lokalnej domenie 'localhost'. Aby skorzystać z usługi wystarczy otworzyć przeglądarkę i wyszukać localhost. Następnie wystarczy wypełnić formularz danymi i przesłać, a program wyświetli wynik "Przeżyłeś"bądź "Nie przeżyłeś", bazując na bazie treningowej. Program jest zaopatrzony w proste i czytelne UI.



Rysunek 1: Port na którym uruchomiona jest aplikacja



Rysunek 2: Wygląd zewnętrzny aplikacji (frontend)



Rysunek 3: Wyświetlanie wyniku algorytmu

Dodatkowe informacje

Algorytm nie działa w sposób probabilistyczny, co oznacza, że po otrzymaniu konkretnego przykładu (np. kobieta, klasa 1, 0 rodzeństwa, 0 rodziców, opłata za bilet: 70.00, port startowy: Cherbourg), model decyduje jednoznacznie, czy dana osoba przeżyłaby, czy nie. Decyzja podejmowana jest na podstawie porównania cech tej osoby z danymi treningowymi, bez uwzględniania rozkładów prawdopodobieństw.

Jeżeli dana instancja nie wykazuje wystarczającego podobieństwa do przypadków osób, które przeżyły, zostaje automatycznie sklasyfikowana jako osoba, która by zginęła — niezależnie od proporcji takich przypadków w zbiorze danych. Taka deterministyczna natura modelu powoduje, że dla tych samych danych wejściowych wynik klasyfikacji zawsze będzie identyczny.

Dla porównania, w przypadku modelu probabilistycznego, osoba o takich samych cechach mogłaby mieć 30% szans na przeżycie, co skutkowałoby tym, że przy 10-krotnym podaniu tych samych danych moglibyśmy otrzymać około 3 wyniki pozytywne (przeżycie) i 7 negatywnych (śmierć).

Część II

Opis działania

Nasz model do przewidywania, czy dana osoba przeżyje katastrofę Titanica, bazuje na algorytmie drzewa decyzyjnego. Różnica polega jednak na tym, że klasyfikator zbiorów miękkich decyduje, czy drzewo powinno kontynuować podział danych, czy utworzyć liść.

Algorytmy

Dla każdej klasy $c \in \mathbb{Y}$

- oblicza się średnią i odchylenie standardowe każdej cechy:

$$\mu_{cj} = \mathbb{E}[x_j | y = c], \ \sigma_{cj} = \mathrm{std}[x_j | y = c]$$

- następnie dla próbki $\mathbf{x}$ i danej klasy c, przekształcenie:

$$\tilde{x}_{cj} = \begin{cases} 1 & \text{jeśli } x_j \geq \mu_{cj} - \tau\sigma_{cj} \\ 0 & \text{w przeciwnym razie} \end{cases}$$

- łączna liczba dopasowań dla klasy c:

$$s_c(\mathbf{x}) = \sum_{j=1}^{d} \tilde{x}_{cj}$$

- predykcja:

$$\hat{y} = \arg\max_c s_c(\mathbf{x})$$

Algorithm 1: Algorytm klasyfikatora zbiorów miękkich użytego jako warunek stopu.

Data: Zbiór danych $D = \{(\mathbf{x}_i, y_i)\}_{i=1}^{N}$, gdzie $\mathbf{x}_i \in \mathbb{R}^d$ jest wektorem cech, a $y_i \in \mathbb{Y}$ jest etykietą klasy, $\mathbb{Y} = \{0, ..., C-1\}$
Result: Struktura drzewa $T : \mathbb{R}^d \to \mathbb{Y}$
Dla każdego możliwego podziału na cechę j i próg s, dane dzielone są na dwa podzbiory:

$$D_{left} = \{(\mathbf{x}_i, y_i) \in D : x_{ij} < s\},\ D_{right} = D \backslash D_{left}.$$

Aby wybrać najlepszy podział, oblicza się zysk informacji:

$$G = I(D) - (\frac{|D_{left}|}{|D|} I(D_{left}) + \frac{|D_{right}|}{|D|} I(D_{right}))$$

gdzie $I(\cdot)$ to miara niejednorodności, w klasyfikacji stosuje się m.in.:

- Indeks Giniego:

$$G(D) = 1 - \sum_{k=0}^{C-1} p_k^2$$

- Entropię:

$$H(D) = -\sum_{k=0}^{C-1} p_k \log_2 p_k$$

gdzie p_k to udział klasy k w zbiorze D.

Następnie drzewo tworzone jest rekurencyjne do momentu, gdy zostanie spełniony z jeden z warunków:

- maksymalna głębokość została osiągnięta: depth $\geq$ max_depth
- liczba próbek w węźle jest mniejsza niż min_samples_split
- brak poprawy zysku informacji: $G \leq 0$
- klasyfikator miękki osiągnął dokładność powyżej progu: $\text{accuracy}_{soft}(D) \geq$ soft_threshold

Po zbudowaniu drzewa, funkcja predykcji $T(\mathbf{x}) = \text{label}(L(\mathbf{x}))$, gdzie $L(\mathbf{x})$ to liść, do którego trafia punkt $\mathbf{x}$.

Algorithm 2: Algorytm drzewa decyzyjnego, w którym warunkiem stopu jest klasyfikator zbiorów miękkich.

Zbiór danych

Poniżej prezentujemy pierwotną strukturę danych pobranych ze strony kaggle.com. Dane te zostały przez nas poddane edycji i oczyszczeniu w celu dostosowania ich do założeń projektu oraz zwiększenia skuteczności zastosowanych algorytmów predykcyjnych.

Tabela 1: Pierwotna struktura danych

Kolumna	**Typ danych**	**Opis**
`PassangerId`	INT (PK)	Unikalny identyfikator pasażera
`Iame`	VARCHAR(100)	Imię i nazwisko pasażera
`Sex`	ENUM('male','female')	Płeć
`Age`	DECIMAL(4,1)	Wiek pasażera (np. 35.0)
`Ticket`	VARCHAR(50)	Numer biletu
`Cabin`	VARCHAR(20)	Numer kabiny (jeśli przydzielony)
`Fare`	DECIMAL(8,2)	Cena biletu
`Embarked`	ENUM('C','Q','S')	Miejsce wejścia na pokład (Cherbourg, Queenstown, Southampton)
`Pclass`	TINYINT	Klasa pasażerska (1, 2, 3)
`Survived`	BOOLEAN	TRUE – przeżył, FALSE – zginął

W celu przygotowania danych do analizy i budowy modelu predykcyjnego przeprowadziliśmy proces oczyszczania oraz selekcji zmiennych. Usunęliśmy kolumny, które nie miały istotnego wpływu na zmienną docelową "Survived" lub dla których zaobserwowana zależność była znikoma. Dotyczyło to m.in. pól takich jak PassengerId, Name czy Ticket, które były unikalne dla każdego pasażera i nie wnosiły wartości predykcyjnej.

Zmieniliśmy także format wybranych zmiennych kategorycznych, stosując kodowanie zero-jedynkowe (one-hot encoding). Zmienną Embarked, która wskazywała miejsce zaokrętowania (C, Q, S), przekształciliśmy na trzy oddzielne kolumny binarne, w których 1 oznaczało obecność danego portu, a 0 jego brak. W podobny sposób zakodowaliśmy zmienną Sex, przypisując wartość 0 dla kobiet oraz 1 dla mężczyzn.

Zrezygnowaliśmy ze zmiennej Age, mimo jej potencjalnej przydatności, ze względu na dużą liczbę brakujących wartości. Spośród 891 rekordów aż ponad 100 nie zawierało informacji o wieku. Uzyskana w ten sposób luka w danych ograniczała jakość modelu oraz skutkowała zmniejszeniem zbioru treningowego, dlatego uznaliśmy, że bezpieczniej będzie całkowicie wykluczyć tę zmienną z dalszej analizy.

Poniżej przedstawiamy ostateczną wersję przekształconej struktury danych, która została wykorzystana w dalszych etapach projektu.

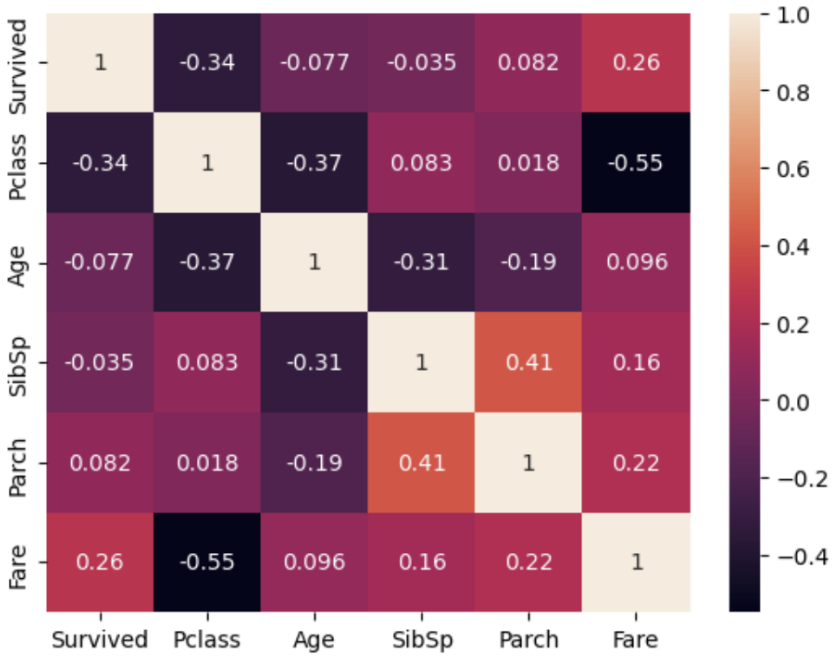
Tabela 2: użytkowa struktura danych

Kolumna	Typ danych	Opis
`id`	INT (PK)	Unikalny identyfikator pasażera
`sex`	ENUM('male','female')	Płeć
`fare`	DECIMAL(8,2)	Cena biletu
`embarked`	ENUM('C','Q','S')	Miejsce wejścia na pokład (Cherbourg, Queenstown, Southampton)
`class`	TINYINT	Klasa pasażerska (1, 2, 3)
`survived`	BOOLEAN	TRUE – przeżył, FALSE – zginął

Zależności pomiędzy zmiennymi a ich znaczenie względem zmiennej docelowej Survived zostały przeanalizowane z wykorzystaniem funkcji seaborn.heatmap, co przedstawiono na poniższej ilustracji.

```
In [5]:
sns.heatmap(
    df_train.drop(["PassengerId", "Name", "Ticket", "Cabin", "Embarked", "Sex"], axis=1).corr(),
    annot=True,
)
```

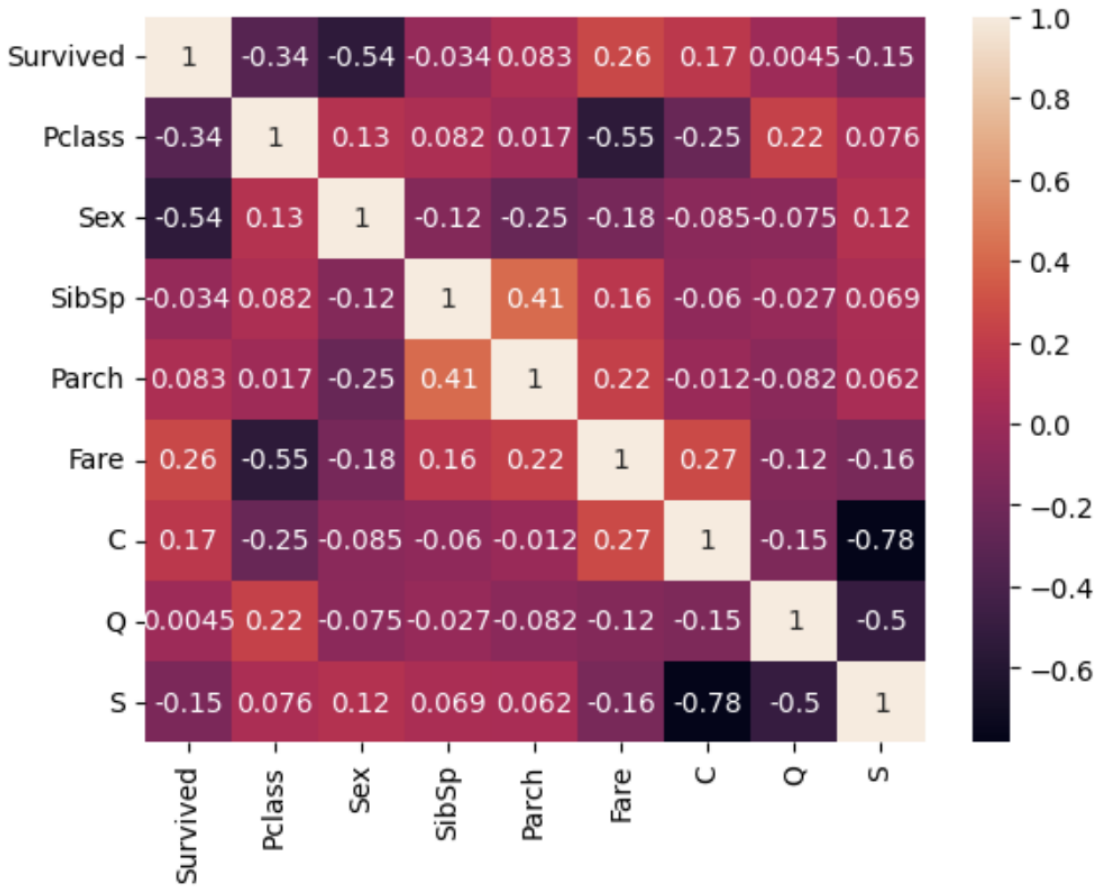
```
Out[5]: <Axes: >
```



Rysunek 4: Heatmap 1

```
In [24]: sns.heatmap(df_train.corr(), annot=True)

Out[24]: <Axes: >
```



Rysunek 5: Heatmap 2

Na podstawie powyższej mapy (heatmap) można zauważyć, że zmiennymi najsilniej skorelowanymi z prawdopodobieństwem przeżycia pasażera (Survived) są płeć (Sex, współczynnik korelacji ≈ -0.54) oraz klasa (Pclass, ≈ -0.34). Wartości ujemne oznaczają korelację odwrotną, co w tym przypadku oznacza, że niższe wartości zmiennej Sex (gdzie 0 oznacza kobiety, a 1 mężczyzn) są związane z wyższym prawdopodobieństwem przeżycia — innymi słowy, kobiety miały większe szanse na przetrwanie katastrofy.

Podobna zależność występuje w przypadku zmiennej Pclass, gdzie pasażerowie z klasy 1. (najwyższej) przeżywali częściej niż pasażerowie klasy 3. (najniższej).

Pozostałe zmienne wykazują znacznie słabszą korelację z przeżyciem. Warto wspomnieć o zmiennej Fare (cena biletu), która jest częściowo skorelowana z klasą (Pclass) – współczynnik korelacji wynosi około -0.55 – dlatego jej wpływ na Survived może być pośredni.

Zmienne takie jak PassengerId, Name, Ticket, Cabin oraz Embarked okazały się nieistotne w kontekście modelu predykcyjnego, głównie z powodu ich unikalności (np. Name, Ticket) lub dużej liczby braków (Cabin). Zmienna Age również nie wykazała silnej korelacji z przeżyciem, a dodatkowo zawierała wiele brakujących danych (891 rekordów ogółem, z czego 714 zawie-

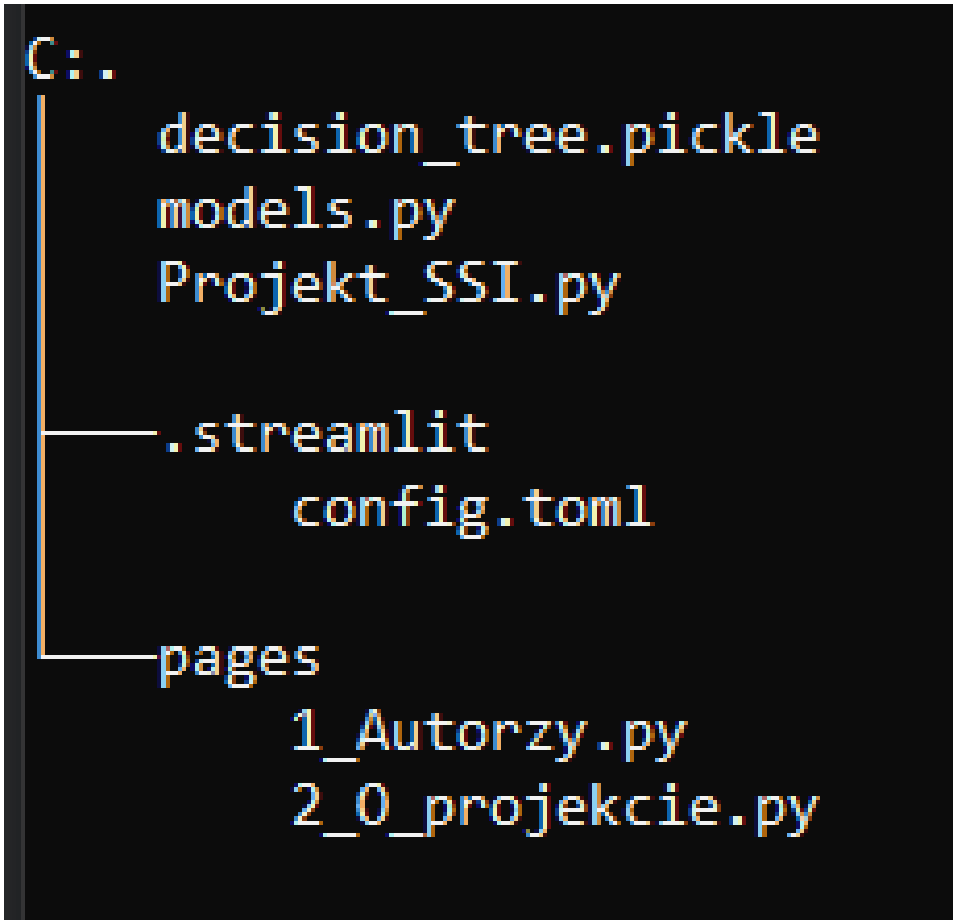
rało wartość Age, co oznacza brak danych w 177 przypadkach), co ograniczyło jej wartość analityczną.

```
: df_train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   PassengerId  891 non-null    int64
 1   Survived     891 non-null    int64
 2   Pclass       891 non-null    int64
 3   Name         891 non-null    object
 4   Sex          891 non-null    object
 5   Age          714 non-null    float64
 6   SibSp        891 non-null    int64
 7   Parch        891 non-null    int64
 8   Ticket       891 non-null    object
 9   Fare         891 non-null    float64
 10  Cabin        204 non-null    object
 11  Embarked     889 non-null    object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
```

Rysunek 6: info

Implementacja

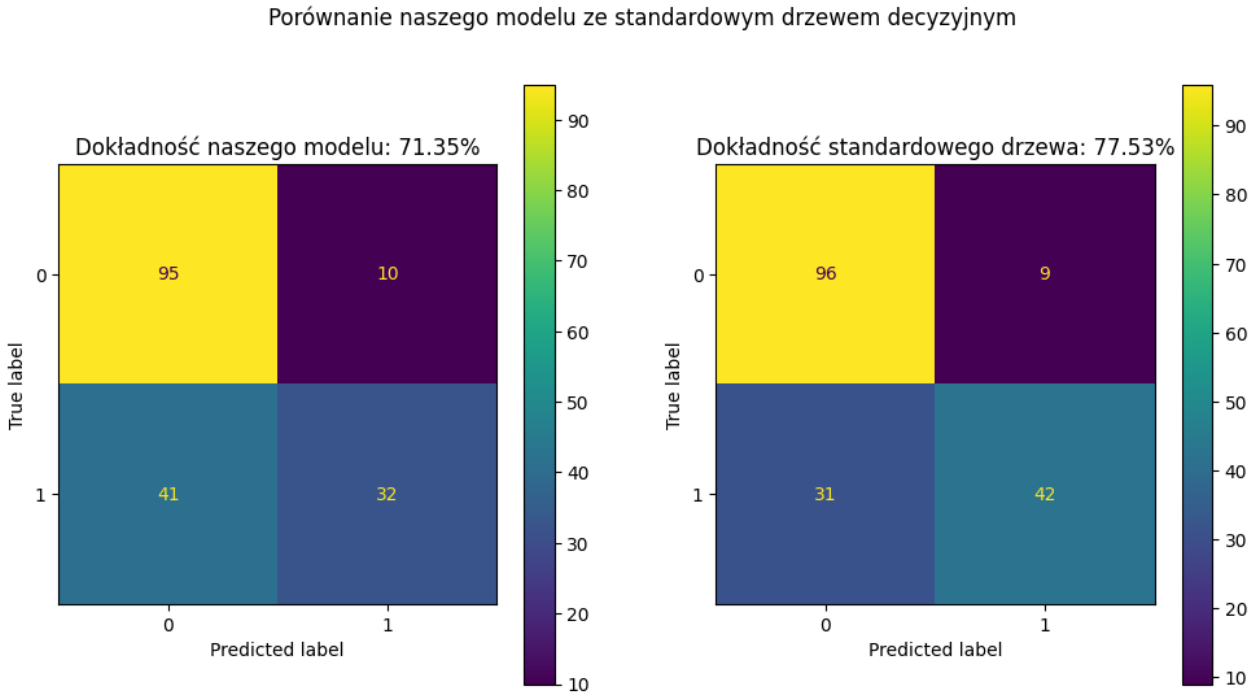


Rysunek 7: Struktura plików

Testy

Nasz model porównaliśmy ze standardowym drzewem decyzyjnym z bilbioteki *scikit-learn* z tymi samymi parametrami tj. max_dept ustawiliśmy na 100, za miarę niejednorodności posłużył nam indeks Gini'ego, a wartość min_samples_split ustawiliśmy na 2.

Dokładność naszego modelu okazała się być niższa aż o 6 pkt. procentowych od dokładności standardowego drzewa decyzyjnego. Porównując macierze pomyłek obu modeli można zauważyć, że standardowe drzewo lepiej poradziło sobie z klasyfikacją osób, które katastrofę przeżyły. Osoby, które katastrofę przeżyły, nasz model częściej klasyfikował jako osoby, które owej katastrofy nie przeżyły. Jeżeli chodzi o klasyfikację osób, które katastrofy nie przeżyły - oba modele poradziły sobie równie dobrze.



Rysunek 8: Porównanie macierzy pomyłek obu modeli

Pełen kod aplikacji

Pełen kod wraz z plikiem notatnika Jupyter, w którym trenowaliśmy model znajduje się również tutaj.

Frontend

```
import streamlit as st
import pickle as pk
import numpy as np

st.title("Projekt SSI")
with st.form(key='data_form'):
    st.text('Wype nij formularz i sprawd  czy by  prze y !')
    sex = st.radio('P  e  :',['M', 'K'])
    cls = st.selectbox('Klasa: ',['1','2','3'])
    sibsp = st.number_input('Liczba rodze stwa + ma  onek na pok adzie:', 0, 10)
    parch = st.number_input('Liczba rodzic w/dzieci na pok adzie:', 0, 10)
    fare = st.number_input('Op ata za bilet:', 0.0, 512.0)
    embarked = st.radio('Port startowy pasa era:', ['Cherbourg', 'QuinsTown','Southampton'])
    submit = st.form_submit_button('Sprawd !')

if submit:
    data = np.array([[int(cls), sex=='M', sibsp, parch, fare, embarked=='Cherbourg', embarked=='QuinsTown', embarked=='Southampton']])
    with open('decision_tree.pickle','rb') as f:
        dt = pk.load(f)
    surv = dt.predict(data)
    if surv[0] == 1:
        st.success('Prze  y  e  !')
        st.balloons()
    else:
        st.error('Nie  prze  y  e  !')
```

Plik "models.py", zawierający modele DecisionTree oraz SoftClassifier

```
import numpy as np
from typing import Optional
from dataclasses import dataclass


@dataclass
class Split:
    feature_index: int
    threshold: float
    data_left: np.ndarray
    data_right: np.ndarray
```

```
    gain: float


@dataclass
class TreeNode:
    feature_index: Optional[int] = None
    threshold: Optional[float] = None
    left: Optional["TreeNode"] = None
    right: Optional["TreeNode"] = None
    label: Optional[int] = None


class SoftClassifier:
    """
    A soft classifier that transforms features based on statistical
       thresholds
    and assigns labels by matching samples to the closest class profile.
    """

    def __init__(self, threshold=0.5):
        self.threshold = threshold
        self.feature_stats_ = None
        self.classes_ = None

    def fit(self, X, y):
        """
        Calculates mean and standard deviation of each feature per class
           .
        """
        self.classes_ = np.unique(y)
        self.feature_stats_ = {}

        for class_ in self.classes_:
            X_class = X[y == class_]
            means = np.mean(X_class, axis=0)
            stds = np.std(X_class, axis=0)
            self.feature_stats_[class_] = {"mean": means, "std": stds}

        return self

    def _transform_features(self, X, class_):
        """
        Transforms features to binary indicators based on proximity to
           class means.
        """
        means = self.feature_stats_[class_]["mean"]
        stds = self.feature_stats_[class_]["std"]
        transformed = (X >= (means - self.threshold * stds)).astype(int)
        return transformed

    def predict(self, X):
        """
        Predicts class labels based on soft matching of features to
           class profiles.
        """
```

```
63          scores = []
64          for class_ in self.classes_:
65              transformed = self._transform_features(X, class_)
66              score = np.sum(transformed, axis=1)
67              scores.append(score)
68
69          scores = np.array(scores).T
70          y_pred = self.classes_[np.argmax(scores, axis=1)]
71          return y_pred
72
73      def score(self, X, y):
74          """
75          Computes the accuracy of the soft classifier.
76          """
77          return np.sum(self.predict(X) == y) / len(X)
78
79
80  class DecisionTree:
81      """
82      A decision tree classifier with an optional stopping criterion based
            on soft classification.
83      """
84
85      def __init__(
86          self,
87          criterion: str = "gini",
88          max_depth: int = 4,
89          min_samples_split: int = 2,
90          soft_classifier: Optional[SoftClassifier] = None,
91          soft_threshold: float = 0.9,
92      ) -> None:
93          """
94          Initializes the decision tree classifier.
95
96          Args:
97              criterion (str): The function to measure the quality of a
                    split ('gini' or 'entropy').
98              max_depth (int): The maximum depth of the tree.
99              min_samples_split (int): The minimum number of samples
                    required to split a node.
100             soft_classifier (SoftClassifier): Optional soft classifier
                    to evaluate stopping condition.
101             soft_threshold (float): Minimum soft accuracy to stop
                    splitting.
102         """
103         assert criterion in [
104             "gini",
105             "entropy",
106         ], f"Invalid criterion: {criterion}. Choose 'gini' or 'entropy'.
                "
107
108         self.criterion = criterion
109         self.max_depth = max_depth
110         self.min_samples_split = min_samples_split
111         self.soft_classifier = soft_classifier
```

```
112         self.soft_threshold = soft_threshold
113
114     def _entropy(self, probabilities: np.ndarray) -> float:
115         """
116         Computes the entropy of a set of probabilities.
117
118         Args:
119             probabilities (np.ndarray): Array of probabilities for each
                class.
120
121         Returns:
122             float: The entropy value.
123         """
124         probabilities = probabilities[probabilities > 0]
125         return np.sum([-p * np.log2(p) for p in probabilities])
126
127     def _gini(self, probabilities: np.ndarray) -> float:
128         """
129         Computes the Gini impurity of a set of probabilities.
130
131         Args:
132             probabilities (np.ndarray): Array of probabilities for each
                class.
133
134         Returns:
135             float: The Gini impurity value.
136         """
137         return 1 - np.sum([p**2 for p in probabilities])
138
139     def _get_probabilities(self, y: np.ndarray) -> np.ndarray:
140         """
141         Computes the probabilities for each class label in the dataset.
142
143         Args:
144             y (np.ndarray): Array of class labels.
145
146         Returns:
147             np.ndarray: Array of probabilities for each class.
148         """
149         labels = np.unique(y)
150         probabilities = np.empty(labels.shape)
151
152         for i, label in enumerate(labels):
153             probabilities[i] = len(y[y == label]) / len(y)
154
155         return probabilities
156
157     def _calculate_gain(
158         self, parent: np.ndarray, left_child: np.ndarray, right_child:
            np.ndarray
159     ) -> float:
160         """
161         Calculates the information gain from a split.
162
163         Args:
```

```
            parent (np.ndarray): The parent node's data.
            left_child (np.ndarray): The left child node's data.
            right_child (np.ndarray): The right child node's data.

        Returns:
            float: The calculated information gain.
        """
        weight_left = len(left_child) / len(parent)
        weight_right = len(right_child) / len(parent)

        parent_probabilities = self._get_probabilities(parent)
        left_probabilities = self._get_probabilities(left_child)
        right_probabilities = self._get_probabilities(right_child)

        if self.criterion == "gini":
            return self._gini(parent_probabilities) - (
                weight_left * self._gini(left_probabilities)
                + weight_right * self._gini(right_probabilities)
            )
        else:
            return self._entropy(parent_probabilities) - (
                weight_left * self._entropy(left_probabilities)
                + weight_right * self._entropy(right_probabilities)
            )

    def _split(
        self, data: np.ndarray, feature_index: int, threshold: float
    ) -> tuple[np.ndarray, np.ndarray]:
        """
        Splits the dataset into two subsets based on the threshold of a
           feature.

        Args:
            data (np.ndarray): The dataset to split.
            feature_index (int): The index of the feature to split on.
            threshold (float): The value used to split the data.

        Returns:
            Tuple[np.ndarray, np.ndarray]: The left and right splits of
               the dataset.
        """
        mask = data[:, feature_index] < threshold
        return data[mask, :], data[~mask, :]

    def _get_best_split(self, data: np.ndarray) -> Split:
        """
        Finds the best split for the dataset based on the criterion.

        Args:
            data (np.ndarray): The dataset to find the best split for.

        Returns:
            Split: The best split for the dataset.
        """
        best_gain = float("-inf")
```

```
        best_split = None

        for feature_index in range(self.n_features_in_):
            thresholds = np.unique(data[:, feature_index])

            for threshold in thresholds:
                data_left, data_right = self._split(data, feature_index,
                    threshold)

                if data_left.size > 0 and data_right.size > 0:
                    y = data[:, -1]
                    y_left = data_left[:, -1]
                    y_right = data_right[:, -1]

                    gain = self._calculate_gain(y, y_left, y_right)
                    if gain > best_gain:
                        best_split = Split(
                            feature_index=feature_index,
                            threshold=threshold,
                            data_left=data_left,
                            data_right=data_right,
                            gain=gain,
                        )

        if best_split is None:
            return Split(
                feature_index=0,
                threshold=0,
                data_left=data,
                data_right=data,
                gain=0.0,
            )
        return best_split

    def _get_most_frequent_label(self, y: np.ndarray) -> int:
        """
        Returns the most frequent label in the dataset.

        Args:
            y (np.ndarray): The array of labels.

        Returns:
            int: The most frequent label.
        """
        values, counts = np.unique(y, return_counts=True)
        return values[np.argmax(counts)]

    def _init_tree(self, data: np.ndarray, current_depth: int = 0) ->
        TreeNode:
        """
        Recursively builds the decision tree.

        Args:
            data (np.ndarray): The dataset to build the tree from.
            current_depth (int): The current depth of the tree.
```

```
270
271         Returns:
272             TreeNode: The root node of the tree.
273         """
274         n_samples = len(data)
275         y = data[:, -1]
276         X = data[:, :-1]
277
278         if n_samples >= self.min_samples_split and current_depth <= self
                .max_depth:
279             best_split = self._get_best_split(data)
280
281             if best_split.gain > 0:
282                 if self.soft_classifier:
283                     self.soft_classifier.fit(X, y)
284                     if self.soft_classifier.score(X, y) >= self.
                            soft_threshold:
285                         return TreeNode(label=self.
                                _get_most_frequent_label(y))
286
287                 left = self._init_tree(best_split.data_left,
                        current_depth + 1)
288                 right = self._init_tree(best_split.data_right,
                        current_depth + 1)
289
290                 return TreeNode(
291                     feature_index=best_split.feature_index,
292                     threshold=best_split.threshold,
293                     left=left,
294                     right=right,
295                 )
296
297         return TreeNode(label=self._get_most_frequent_label(y))
298
299     def fit(self, X: np.ndarray, y: np.ndarray) -> None:
300         """
301         Trains the decision tree on the given data.
302
303         Args:
304             X (np.ndarray): Feature matrix (n_samples, n_features).
305             y (np.ndarray): Target labels (n_samples,).
306         """
307         self.n_features_in_ = X.shape[1]
308         data = np.concatenate((X, y.reshape((-1, 1))), axis=1)
309         self.tree_ = self._init_tree(data)
310
311     def _predict(self, x: np.ndarray, node: TreeNode) -> int:
312         """
313         Recursively predicts the label for a given sample.
314
315         Args:
316             x (np.ndarray): The input data point.
317             node (TreeNode): The current node in the tree.
318
319         Returns:
```

```
            int: The predicted label.
        """
        if node.label is not None:
            return node.label

        if x[node.feature_index] < node.threshold:
            return self._predict(x, node.left)
        else:
            return self._predict(x, node.right)

    def predict(self, X: np.ndarray) -> np.ndarray:
        """
        Predicts class labels for the input data.

        Args:
            X (np.ndarray): Feature matrix (n_samples, n_features).

        Returns:
            np.ndarray: Predicted class labels (n_samples,).
        """
        return np.array([self._predict(x, self.tree_) for x in X])

    def score(self, X: np.ndarray, y: np.ndarray) -> float:
        """
        Computes the accuracy of the classifier.

        Args:
            X (np.ndarray): Feature matrix (n_samples, n_features).
            y (np.ndarray): True class labels (n_samples,).

        Returns:
            float: Accuracy of the classifier.
        """
        return np.sum(self.predict(X) == y) / len(X)
```