Modele przewidujące chorobę symulacyjną

Daniil Shkarovskyi
Hubert Zgrzywa

1 Wstęp

1.1 Cel projektu

Celem projektu jest znalezienie modeli ML, które są ogólnie dostępne i służą do przewidywania choroby symulacyjnej. Biorąc pod uwagę brak łatwo dostępnych modeli odpowiednich do tego celu, rozpoczęliśmy konstruowanie i porównywanie kilku modeli w celu zidentyfikowania najbardziej efektywnego podejścia.

2 Import i przetwarzanie danych

2.1 Źródło danych

2.1.1 Ogólna informacja

Źródłem naszych danych jest projekt na GitHubie, w którym uczestnicy brali udział w zadaniu nawigacyjnym.

2.1.2 Opis

Uczestnicy: 53 osoby, średnia wieku 26.3 lata (26 kobiet).
Zadanie: Nawigacja w wirtualnym środowisku przy użyciu HTC Vive Pro i opaski Empatica E4.

2.1.3 Dane Zebrane

- **Head-Tracking**: Pozycje i rotacje głowy (surowe i przeskalowane).
- **Ruch**: Prędkość i rotacja (surowe i przeskalowane).
- **Biosygnały**: Odpowiedzi skórno-galwaniczne, puls objętości krwi, temperatura, tętno.
- **Cybersickness**: Wyniki z Kwestionariusza Symulatorowej Choroby (SSQ), w tym nudności, objawy okulomotoryczne i dezorientacja.

2.2 Proces Eksperymentu

Każdy uczestnik brał udział trzykrotnie w różnych dniach, co dało łącznie 159 próbek. Przed zadaniem wypełniali kwestionariusz zdrowotny oraz SSQ przed i po zadaniu, aby ocenić wpływ VR na ich samopoczucie.

2.3 Analiza Danych Bio Sygnałowych

2.3.1 Sygnał BVP (Blood Volume Pulse)

Sygnał BVP mierzy zmiany w objętości krwi w naczyniach krwionośnych, co jest często używane do oceny aktywności układu sercowo-naczyniowego. Na podstawie danych BVP można ocenić aktywność serca, w tym częstotliwość rytmu serca i zmiany w krążeniu krwi. Analiza tego sygnału może dostarczyć informacji na temat reakcji organizmu na stres czy inne bodźce emocjonalne.

2.3.2 Sygnał GSR (Galvanic Skin Response)

Sygnał GSR mierzy elektryczną przewodność skóry, która jest związana z aktywnością układu nerwowego autonomicznego. Zmiany w sygnale GSR mogą świadczyć o zmianach emocjonalnych, takich jak stres, pobudzenie czy lęk. Analiza tego sygnału może być przydatna w identyfikacji reakcji emocjonalnych na bodźce zewnętrzne, w tym na przykład stresujące doświadczenia podczas korzystania z interaktywnych środowisk wirtualnych.

2.3.3 Sygnał HR (Heart Rate)

Sygnał HR mierzy liczbę uderzeń serca na minutę. Jest to kluczowy wskaźnik stanu zdrowia serca i ogólnego poziomu aktywności fizycznej. Analiza sygnału HR może pomóc w identyfikacji zmian w rytmie serca, które mogą być związane z chorobami serca, stresem, zmęczeniem lub innymi czynnikami.

2.3.4 Sygnał TEM (Temperature)

Sygnał TEM mierzy temperaturę ciała. Zmiany w temperaturze ciała mogą być związane z reakcjami organizmu na zmieniające się warunki otoczenia, aktywność fizyczną, stres czy choroby. Analiza sygnału TEM może pomóc w monitorowaniu stanu zdrowia oraz identyfikacji nieprawidłowości w funkcjonowaniu organizmu.

3 Przegląd danych

List of data collected from one sampler

Data Type	Data
Head-Tracking Data	Raw Head Position(i.e., x, y and z) Resampled Head Position(i.e., x, y and z) Head Rotation(i.e., x, y and z) Resampled Head Rotation(i.e., x, y and z)
Motion Data	Raw Speed Resampled Speed Raw Rotation Resampled Rotation
Biosignal Data	Galvanic Skin Response(4Hz) Blood Volume Pulse(64Hz) Heart Rate() Temperature(4Hz)
CyberSickness Data	Simulator Sickness Questionnaire Score Nausea Score Oculomotor Score Disorientation Score

Rysunek 1: Struktura danych

3.1 Przekształcenie danych

Proces przetwarzania i resamplingu czasowego danych zebranych w badaniu. Dane z różnych sesji są przekształcane do jednolitej częstotliwości próbkowania za pomocą interpolacji liniowej. Skrypt definiuje również sposób przechowywania przetworzonych danych w słowniku dla każdej sesji.

```python
# Resampling function
def resample_data(df, target_fs):
    # Assuming 'Time' is in seconds and is sorted
    time_col = 'Time'
    time_new = np.arange(df[time_col].iloc[0], 238.0, 1/target_fs)  # End at 238.0 seconds
    resampled_df = pd.DataFrame({time_col: time_new})

    for col in df.columns:
        if col != time_col:
            interpolator = interpolate.interp1d(df[time_col], df[col], kind='linear', fill_value="extrapolate")
            resampled_df[col] = interpolator(time_new)

    return resampled_df

# Define the target frequency (in Hz)
target_fs = 10  # Zmniejszenie częstotliwości próbkowania do 10 Hz (krok co 0.1 sekundy)

# Dictionary to store resampled data for each session
resampled_time_series_data = {}

# Process each session
for session_name, session in data.items():
    print(f"Próbka sesji: {session_name}")

    session_data = {}
    variables_columns = {
        'BVP': ['Time', 'BVP'],
        'GSR': ['Time', 'GSR'],
        'HR': ['Time', 'HR'],
        'TEM': ['Time', 'TEM'],
        'rawHead': ['Time', 'local_X', 'local_Y', 'local_Z']
    }

    for variable_name, columns in variables_columns.items():
        if variable_name in ['BVP', 'GSR', 'HR', 'TEM']:
            df = getattr(session.Empatica, variable_name)[columns]
        else:
            df = getattr(session.Steam, variable_name)[columns]

        resampled_df = resample_data(df, target_fs)
        session_data[variable_name] = resampled_df

        print(f"Dane dla zmiennej {variable_name}:")
        print(resampled_df)
        print("\n")

    resampled_time_series_data[session_name] = session_data

# Dane przepróbkowane z zakończeniem w 238.0 sekundzie dla każdej zmiennej w każdej sesji
```

Rysunek 2: Przekształcenie danych

3.2 Złączenie szeregów

Fragment kodu prezentuje metodę scalania przetworzonych danych z różnych sesji badawczych. Kod rozpoczyna proces od danych z pulsometru, a następnie dołącza kolejne typy danych. Wykorzystuje metodykę inner join, aby zapewnić spójność czasową wszystkich danych w sesji.

```python
# Function to merge all dataframes on the 'Time' column
def merge_time_series(session_data):
    merged_df = session_data['BVP']  # Start with BVP
    for variable_name, df in session_data.items():
        if variable_name != 'BVP':  # Skip BVP since it's already the base
            merged_df = pd.merge(merged_df, df, on='Time', how='inner')
    return merged_df

# Dictionary to store merged data for each session
merged_time_series_data = {}
invalid_sessions = []  # To store sessions with empty dataframes
inv_count = 0

for session_name, session in resampled_time_series_data.items():
    merged_df = merge_time_series(session)
    if merged_df.empty:
        invalid_sessions.append(session_name)
        inv_count = inv_count + 1
    else:
        merged_time_series_data[session_name] = merged_df
```

Rysunek 3: Złączenie szeregów

3.3 Sprawdzenie nieskończonych wartości i ich usunięcie

Ten skrypt jest używany do identyfikacji i usuwania nieskończonych oraz za dużych wartości z danych sesji. Usuwa anomalie danych, które mogłyby wpłynąć na wyniki analizy, przygotowując dane do dalszego przetwarzania.

```python
# Function to check for infinity and large values
def replace_inf_and_large_values(df):
    df.replace([np.inf, -np.inf], np.nan, inplace=True)
    df.dropna(inplace=True)
    return df

# Apply the function to each session's dataframe
for session_name, df in merged_time_series_data.items():
    merged_time_series_data[session_name] = replace_inf_and_large_values(df)
```

Rysunek 4: Sprawdzenie nieskończonych wartości i ich usunięcie

3.4 Normalizacja

Fragment kodu przedstawia normalizację danych ze wszystkich sesji, co jest ważnym krokiem w przygotowaniu danych do modelowania. Używa StandardScaler do skalowania cech, co jest standardową praktyką w przetwarzaniu danych przed uczeniem maszynowym.

```python
# Normalizacja danych dla każdej sesji
scaler = StandardScaler()
for session_name, df in merged_time_series_data.items():
    if not df.empty:  # Skip empty dataframes
        df.iloc[:, 1:] = scaler.fit_transform(df.iloc[:, 1:])  # Skalowanie wszystkich kolumn poza 'Time'
        merged_time_series_data[session_name] = df
```

Rysunek 5: Normalizacja danych

```
Data for session Zunchaogroup3:
       Time        BVP       GSR         HR        TEM   local_X   local_Y  \
1       0.1   8.616123  2.210020  73.816900  34.023248  1.416288  1.714003
2       0.2   1.478357  2.207879  73.791049  34.016495  1.416142  1.713786
3       0.3   1.307004  2.205738  73.765199  34.009743  1.415972  1.713748
4       0.4 -32.756298  2.196757  73.739349  34.002991  1.414035  1.713589
5       0.5 -30.982178  2.196663  73.713498  33.996238  1.413279  1.713541
...     ...        ...       ...        ...        ...       ...       ...
2375  237.5  -5.104449  5.729392  79.996280  33.410000  1.423013  1.707277
2376  237.6 -10.676447  5.727671  79.996280  33.410000  1.423401  1.707220
2377  237.7  -9.774139  5.725949  79.996280  33.410000  1.423673  1.707373
2378  237.8  10.969138  5.724228  80.150372  33.410000  1.423786  1.707551
2379  237.9   4.868717  5.722507  80.394188  33.410000  1.424284  1.707854

       local_Z
1     0.624417
2     0.625005
3     0.625026
4     0.622013
5     0.619467
...        ...
2375  0.564543
2376  0.562017
2377  0.560143
2378  0.558363
2379  0.556625

[2379 rows x 8 columns]
```

Rysunek 6: Przykładowe dane z jednej sesji

3.5 Podział na zbiór treningowy i testowy

Ostatni rozdział dotyczy podziału danych na zestawy treningowe i testowe. Po wcześniejszym przetworzeniu i znormalizowaniu danych, wszystkie sesje są połączone w jeden duży zbiór, który zawiera zarówno cechy (dane wejściowe), jak i odpowiedzi (oceny). Następnie używając standardowej metody, dzielimy te dane na dwie części: zbiór treningowy, który posłuży do nauki modelu oraz zbiór testowy, który posłuży do sprawdzenia, jak dobrze model działa na nowych, nieznanych danych. Podział jest ustawiony tak, że 80% danych trafia do zbioru treningowego, a pozostałe 20% do zbioru testowego.

```
# Składanie wszystkich sesji w jedną macierz cech i wektor odpowiedzi
X_list = []
y_list = []

for session_name, df in merged_time_series_data.items():
    if not df.empty:  # Skip empty dataframes
        X_list.append(df.iloc[:, 1:])  # Wszystkie cechy poza 'Time'
        y_list.append(pd.Series([data[session_name].SicknessLevel.SSQ] * len(df), name='SSQ'))

X = pd.concat(X_list)
y = pd.concat(y_list)

# Podział na zbiór treningowy i testowy
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Rysunek 7: Podział danych na zbiory do trenowania modelu

4 Definiowanie modelu

Na początku zdefiniowaliśmy trzy modele regresji, czyli drzewo decyzyjne (DT), regresja liniowa (LR) i wyjaśnialna maszyna wzmacniająca (EBM) oraz dwa modele klasyfikacji: DT i EBM. Następnie stworzyliśmy dwa podstawowe modele sieci neuronowych.

```python
# Podział danych na zbiór treningowy i testowy
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Przekształcenie etykiet na klasy binarne
threshold = 90  # Przykładowy próg
y_train_binary = (y_train > threshold).astype(int)
y_test_binary = (y_test > threshold).astype(int)

# Decision Tree Regressor
dt_regressor = DecisionTreeRegressor()
dt_regressor.fit(X_train, y_train)
dt_regressor_predictions = dt_regressor.predict(X_test)
dt_regressor_mse = mean_squared_error(y_test, dt_regressor_predictions)
print("Decision Tree Regressor MSE:", dt_regressor_mse)

# Decision Tree Classifier
dt_classifier = DecisionTreeClassifier()
dt_classifier.fit(X_train, y_train_binary)
dt_classifier_predictions = dt_classifier.predict(X_test)
dt_classifier_accuracy = accuracy_score(y_test_binary, dt_classifier_predictions)
print("Decision Tree Classifier Accuracy:", dt_classifier_accuracy)

# Linear Regression
linear_regressor = LinearRegression()
linear_regressor.fit(X_train, y_train)
linear_regressor_predictions = linear_regressor.predict(X_test)
linear_regressor_mse = mean_squared_error(y_test, linear_regressor_predictions)
print("Linear Regression MSE:", linear_regressor_mse)

# Explainable Boosting Machine Regressor
ebm_regressor = ExplainableBoostingRegressor()
ebm_regressor.fit(X_train, y_train)
ebm_regressor_predictions = ebm_regressor.predict(X_test)
ebm_regressor_mse = mean_squared_error(y_test, ebm_regressor_predictions)
print("EBM Regressor MSE:", ebm_regressor_mse)

# Explainable Boosting Machine Classifier
ebm_classifier = ExplainableBoostingClassifier()
ebm_classifier.fit(X_train, y_train_binary)
ebm_classifier_predictions = ebm_classifier.predict(X_test)
ebm_classifier_accuracy = accuracy_score(y_test_binary, ebm_classifier_predictions)
print("EBM Classifier Accuracy:", ebm_classifier_accuracy)
```

Rysunek 8: Definiowanie modeli regresji i klasyfikacji

```python
# Model 1
model1 = keras.Sequential([
    layers.Dense(64, activation='relu', input_shape=[X_train.shape[1]]),
    layers.Dense(64, activation='relu'),
    layers.Dense(1)
])

# Kompilowanie modelu
model1.compile(optimizer='adam', loss='mse', metrics=['mae'])

# Trenowanie modelu
history1 = model1.fit(X_train, y_train, epochs=100, validation_split=0.2)
```

Rysunek 9: Definiowanie modelu nr. 1

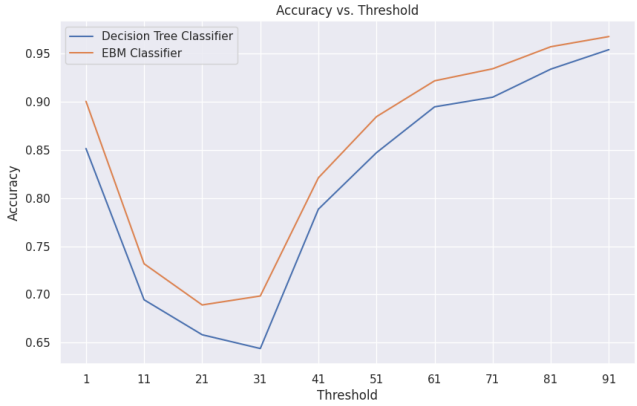
```python
# Przykładowa architektura LSTM
model_lstm = Sequential([
    LSTM(64, input_shape=(timesteps, features)),
    Dense(1)
])

# Kompilowanie modelu
model_lstm.compile(optimizer='adam', loss='mse', metrics=['mae'])

# Trenowanie modelu
history_lstm = model_lstm.fit(X_train_reshaped, y_train, epochs=150, validation_split=0.2)
```

Rysunek 10: Definiowanie modelu LSTM

5 Trenowanie modelu

5.1 Analiza wpływu parametru progowego na dokładność modeli klasyfikacyjnych

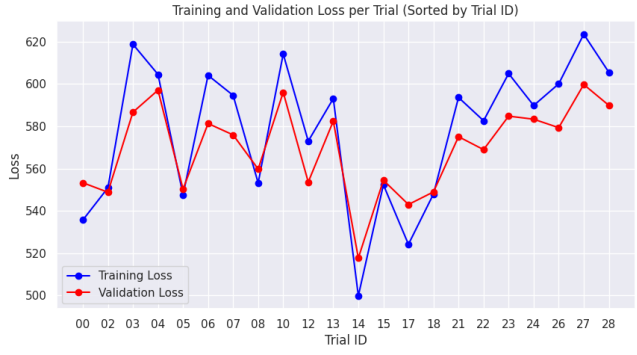
W tej części projektu skupiliśmy się na znalezieniu optymalnej wartości progowej dla klasyfikacji choroby symulacyjnej. Celem było ustalenie, przy jakiej wartości parametru threshold oba modele osiągają najwyższą dokładność (accuracy).



Rysunek 11: Wykres zależności dokładności modeli klasyfikacji od wartości parametru threshold

5.2 Strojenie hiper-parametrów w celu minimalizacji strat dla modelu nr. 1

Zdecydowaliśmy się spróbować strojenia hiper-parametrów dla stworzonego modelu sieci neuronowej. Proces ten obejmował różne konfiguracje liczby neuronów w warstwach, stopy odrzucenia czy wartości tempa uczenia się.

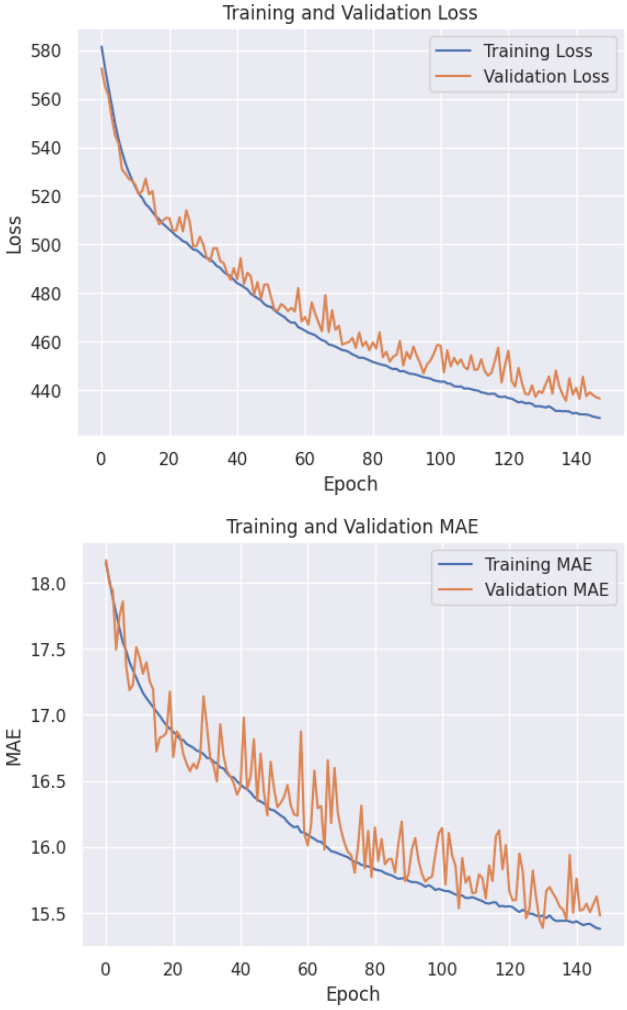


Rysunek 12: Wizualizacja efektywności strojenia hiper-parametrów

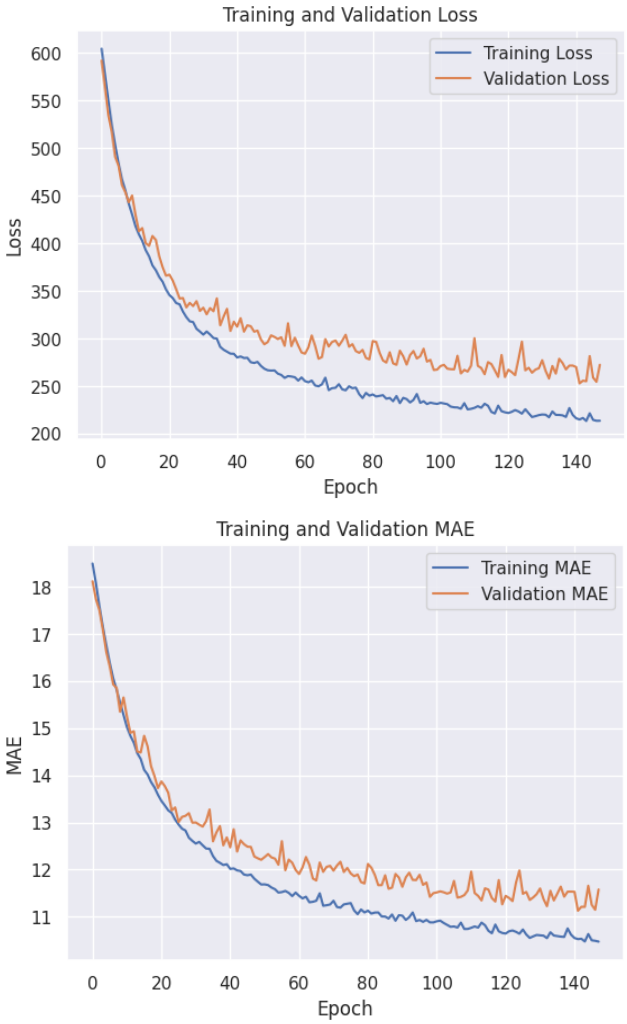
Najlepsze wyniki uzyskano w próbie nr. 14. Training Loss wyniósł ∼499.87, a Validation Loss ∼517.70. Wyższy wynik dla części walidacyjnej może wskazywać

na przetrenowanie modelu. Z tego powodu moglibyśmy użyć innych wyników, natomiast po odrzuceniu prób nr. 14 i 17 żadna nie wyróżnia się szczególnie.

5.3 Analiza postępów w uczeniu modelu sieci neuronowej



Rysunek 13: Historia trenowania modelu nr. 1

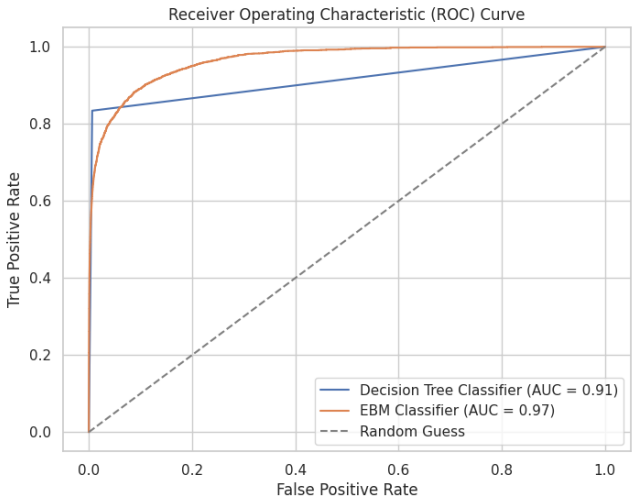


Rysunek 14: Historia trenowania modelu LSTM

Wydaje się, że oba modele mogłyby poprawić swoje wyniki przy dalszym trenowaniu. Szczególnie model nr. 1.

6 Ocena modelu

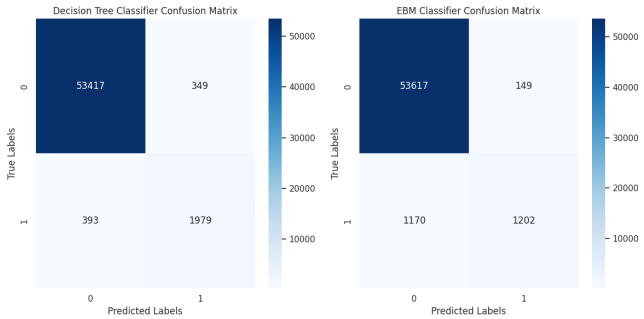
6.1 Analiza porównawcza charakterystyki operacyjnej odbiornika (ROC) dla klasyfikatorów Decision Tree i EBM



Rysunek 15: Porównanie wydajności klasyfikatorów Decision Tree i EBM na podstawie krzywej ROC

```
Decision Tree Classifier Confusion Matrix:
[[53417   349]
 [  393  1979]]

EBM Classifier Confusion Matrix:
[[53617   149]
 [ 1170  1202]]

Decision Tree Classifier Metrics:
Precision: 0.85
Recall: 0.83
F1-score: 0.84

EBM Classifier Metrics:
Precision: 0.89
Recall: 0.51
F1-score: 0.65
```

Rysunek 16: Metryki i macierze pomyłek dla klasyfikatorów Decision Tree i EBM



Rysunek 17: Macierze pomyłek na wykresach

Wyniki analizy ROC podkreślają znaczenie wyboru odpowiedniego modelu klasyfikacyjnego w kontekście choroby symulacyjnej. Klasyfikator EBM z jego wyższym AUC jest preferowanym modelem, oferującym większą pewność i mniejsze ryzyko fałszywie pozytywnej klasyfikacji. Nie jest on jednak idealny i należałoby dalej nad nim pracować.

6.2 Analiza porównawcza modeli regresji

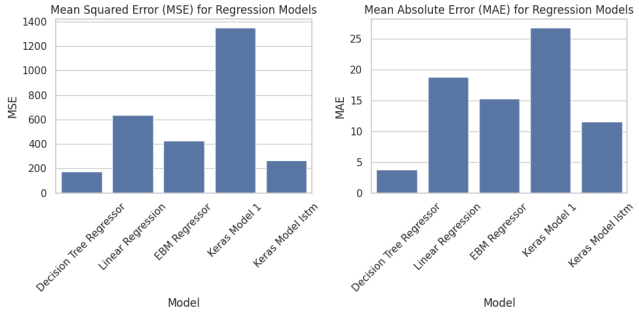
Rysunek 18: Wizualizacja dokładności modeli regresyjnych w przewidywaniu SSQ dla pierwszych 70 próbek

```
Decision Tree Regressor MSE: 178.49849992518438
Decision Tree Classifier Accuracy: 0.9867825715201824
Linear Regression MSE: 634.6190252311699
EBM Regressor MSE: 428.01760495490555
EBM Classifier Accuracy: 0.9765043286187609
```

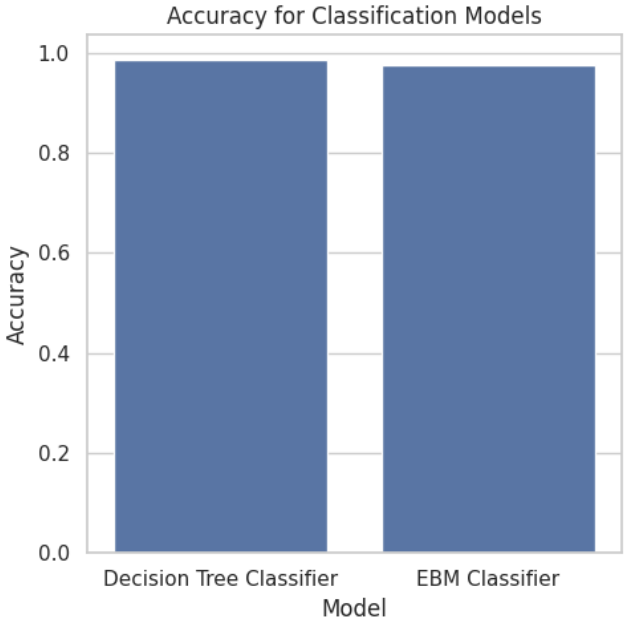
Rysunek 19: Wyniki modeli regresji

```
Decision Tree Regressor:
Mean Squared Error (MSE): 178.49849992518438
Mean Absolute Error (MAE): 3.844462574370309

Linear Regression:
Mean Squared Error (MSE): 634.6190252311699
Mean Absolute Error (MAE): 18.820834830447495

EBM Regressor:
Mean Squared Error (MSE): 428.01760495490555
Mean Absolute Error (MAE): 15.276940618268632

1755/1755 [==============================] - 3s 2ms/step
Keras Model 1:
Mean Squared Error (MSE): 441.3918230938935
Mean Absolute Error (MAE): 15.561118740227512

1755/1755 [==============================] - 12s 7ms/step
Keras Model lstm:
Mean Squared Error (MSE): 268.6220591682612
Mean Absolute Error (MAE): 11.552814803651803

Decision Tree Classifier:
Accuracy: 0.9867825715201824

EBM Classifier:
Accuracy: 0.9765043286187609
```

Rysunek 20: Porównanie modeli



Rysunek 21: Porównanie modeli



Rysunek 22: Porównanie modeli

7 Wnioski

Analiza porównawcza modeli wykazała, że wybór odpowiedniego modelu jest kluczowy w kontekście przewidywania choroby symulacyjnej. Klasyfikator EBM z jego wyższym AUC okazał się być najbardziej obiecującym modelem, jednak dalsze prace nad nim są konieczne. W przypadku modeli regresyjnych, dokładność przewidywania różniła się w zależności od zastosowanej metody, co wskazuje na konieczność dalszej optymalizacji i testowania różnych podejść.

8 Kod na Google Colab

Link do Google Colab

9 Instrukcja do uruchomienia

1. **Ładowanie pliku z GitHuba:** Proszę załadować plik o nazwie `createDataset.py` do Google Colab. Plik ten można znaleźć na tej stronie GitHub: GitHub Repozytorium.

2. **Ładowanie pliku z Google Drive:** Następnie, proszę załadować plik `raw_data2020.p` do Google Drive. Ten plik znajduje się w tym samym repozytorium GitHub, w folderze `data`.

3. **Zmiana ścieżki do pliku:** Proszę zmienić ścieżkę do pliku `data_path` w Google Drive z `/content/drive/MyDrive/AGH/6/UM/raw_data2020.p`, na taką która odpowiada lokalizacji pliku zgodnie z lokalizacją, w której umieściliście dane.

4. **Uruchamianie kodu:** Teraz możecie uruchomić wszystko krok po kroku.