Politechnika Poznańska

Wydział Automatyki, Robotyki i Elektrotechniki

Instytut Robotyki i Inteligencji Maszynowej

Projekt przejściowy

Badanie poprawności działania wirników dronów za pomoca algorytmów sztucznej inteligencji

Paweł Brzozowski, Zofia Walczak

Prowadzący:
mgr inż. Adam Bondyra
adam.bondyra@put.poznan.pl

Poznań, 14 lipca 2021

Spis treści

1 Problem badawczy **2**

2 Metodyka badania **2**
2.1 Zebranie danych podczas prób laboratoryjnych 2
2.2 Wybór odpowiedniej metody pracy 2
2.3 Wavelet 2

3 Implementacja kodu **3**
3.1 Plik main.py 3
3.2 Plik functions 5
3.3 Plik dataset_class.py 6
3.4 Plik prediction.py 8

4 Wyniki działania programu **10**

5 Dobranie odpowiednich nastaw **12**
5.1 Zmiana wielkości grupy testowej 12
5.2 Liczba drzew 13
5.3 Inny typ okna 14
5.4 Porównanie końcowe wszystkich ustawień 15

6 Podsumowanie wyników pracy **16**

7 Bibliografia **16**

1 Problem badawczy

Celem poniższego badania jest stworzenie modelu pozwalającego na detekcję uszkodzeń śmigieł bezzałogowych statków powietrznych. Praca ta jest rozwinięciem pracy inżynierskiej Pana Pawła Drapikowskiego. Weryfikacja stanu technicznego śmigieł odbywała się poprzez wykorzystanie modułu stworzonego przez Pana Pawła Drapikowskiego, który odczytywał prąd płynący przez uzwojenie silnika BLDC. Za pomocą skryptu napisanego w języku Matlab zebraliśmy dane podczas laboratoryjnych pomiarów pracy wirnika z różnymi typami śmigieł pod względem zużycia.

2 Metodyka badania

Sekcja ta poświęcona jest przedstawieniu poszczególnych kroków potrzebnych do stworzenia tego programu.

2.1 Zebranie danych podczas prób laboratoryjnych

Podczas badania laboratoryjnego do przymocowanego silnika bezzałogowego statku powietrznego podłączyliśmy śmigło będące w bardzo dobrej kondycji mechanicznej oraz śmigło, które było już zepsute.
Badania polegały na próbkowaniu sygnału generowanego przez prąd silnika a następnie zapisaniu ich do pliku csv. Zebraliśmy odpowiednio 225 plików dla śmigła zdrowego oraz 200 plików dla śmigła uszkodzonego. Każdy plik był wektorem składającym się z 1 kolumny oraz 2000 wierszy.

2.2 Wybór odpowiedniej metody pracy

Aby w sposób automatyczny móc dokonywać detekcji uszkodzeń śmigieł wirnika postanowiliśmy skorzystać z algorytmów uczenia maszynowego. Postanowiliśmy skorzystać z metody lasu losowego - (ang. Random Forest) czyli metody zespołowego uczenia maszynowego polegającego na konstruowaniu wielu drzew decyzyjnych w czasie uczenia i generowaniu klasy, która jest dominantą klas (klasyfikacja) lub przewidywaną średnią (regresja) poszczególnych drzew.

2.3 Wavelet

Przygotowanie danych do dalszej obróbki polegało na skorzystaniu z dekompozycji falkowej - każde okno wyliczone z plików .csv pocięliśmy na mniejsze fragmenty za pomocą funkcji WaveletPacket z biblioteki pywt. W naszym kodzie użyliśmy dekompozycji 4 poziomu - ostatecznie 1 okno zostało pocięte na $2^4 = 16$ pakietów. Następnie z każdej paczki było liczone odchylenie standardowe, które było przekazywane do klasyfikatora.

3 Implementacja kodu

3.1 Plik main.py

W tym pliku tworzymy obiekty które posiadają zbiory danych zebranych podczas prób badawczych z wirników których śmigła były 'zdrowe' oraz 'wadliwe'.
Poprzez określenie 'zdrowe' oraz 'wadliwe' rozumiemy śmigła, które odpowiednio są bez wad mechanicznych oraz takie, które posiadają różnego stopnia wady mechaniczne.
Do stworzonych obiektów przypisujemy początkowo dane zebrane podczas prób badawczych. Wiemy które dane zebrane zostały z jakiego rodzaju śmigieł, więc cały proces jest prosty w realizacji. Dane te zapisujemy jako zmienne poszczególnego obiektu.
Następnie za pomocą metody poszczególnego obiektu liczmy okno hamminga z danych reprezentujących pomiary. Wyliczone okno hamminga zapisujemy w formie listy jako zmienna obiektu.
Kolejno tworzymy wavelety z otrzymanych danych i również zapisujemy je jako zmienne obiektu.
Finalnie oba obiekty zostaną przekazane jako argumenty do funkcji która korzystając z algorytmów sztucznej inteligencji tworzymy model aby zbadać, czy możliwe jest wykrycie uszkodzenia wirnika.

Kod prezentujący działanie funkcji:

```
import functions
import dataset_class
import prediction

# Loading files from csv.
# Function functions.get_data_from_csv() requires type of data set [healthy or faulty] and return list of pandas DataFrame objects
# Returned data from function above is now assigned as data_from_csf parameter in dataset_class.Dataset()
healthy = dataset_class.Dataset(functions.get_data_from_csv(type="healthy"), data_type="healthy")
faulty = dataset_class.Dataset(functions.get_data_from_csv(type="faulty"), data_type="faulty")


# Now hamming window function is being computed for each item in data_from_csv parameter od Dataset()
# Elements are saved in new list (Dataset() in parameter hamming_window_list
healthy.compute_hamming_window()
faulty.compute_hamming_window()


# Creating wawelets and computing standard deviation for each list of wawelets
healthy.create_wavelet_packet()
faulty.create_wavelet_packet()


# A function that accepts healthy and faulty sets as arguments.
# With the help of sklearn, we create a model and then, using the random forest,
# we try to predict whether the given data sets represent a defective rotor or a healthy one.
# The function returns nothing, but prints the confusion matrix and displays the graphical interpretation of the results
prediction.predict_values(healthy=healthy,faulty=faulty)
```

3.2 Plik functions

Plik ten odpowiada za prawidłowe wczytanie danych w postaci plików zawartych w folderze. Funkcja ta zwraca listę wczytanych plików z rozszerzeniem csv. Dzięki temu możemy uzyskać listę danych, która będzie nam potrzebna do inicjalizacji obiektów.

Kod realizujący powyższe zagadnienia:

```
1  import pandas as pd
2  import os
3  def get_data_from_csv(type):
4      #Checing if type of files is correct [there are only 2 options as
           ↪ this was specyfied in task]
5      if type != 'healthy' or type != 'faulty':
6
7          # Loading list of files names
8          files_names = os.listdir(f'./datasets/{type}')
9          dataset_list = []
10
11         # Loading csv files from directory and appending it into the
               ↪ list
12         for file_path in files_names:
13             with open(f"./datasets/{type}/{file_path}", 'r') as file:
14                 df = pd.read_csv(file)
15                 dataset_list.append(df)
16         print(f'Downloaded {len(dataset_list)} files in {type} data
               ↪ completed')
17
18         # Returning list of all loaded files as list of DataFrame
               ↪ objects
19         return dataset_list
20
21     # In case of wrong type no files will be loaded
22     else:
23         print('Sorry your type is incorrect, please choose healthy or
               ↪ faulty')
```

3.3 Plik dataset_class.py

Plik ten opisuje klasę jaką jest dataset. Tworzymy tutaj wszystkie niezbędne metody oraz zmienne dla obiektów. W ten sposób łatwiej jest przechowywać dane oraz je obrabiać.

```
1  import statistics
2  import numpy as np
3  import pywt
4
5  class Dataset:
6      def __init__(self, data, data_type):
7          self.data_type = data_type
8          self.data_from_csv = data
9          self.hamming_window_list = []
10         self.wavelet_list = []
11         self.std_deviation_list = []
12         self.target_list = []
13
14     # calculating standard deviation and saving it into object argument
15     def standard_deviation(self):
16         for item in self.wavelet_list:
17             self.std_deviation_list.append(statistics.stdev(item))
18
19     # calculating hamming window and then appending it into object
       ↪ argument
20     def compute_hamming_window(self):
21         for item in self.data_from_csv:
22             temporary_hamming_window = []
23             current_array = np.array(item)
24             hamming = np.hamming(len(item))
25
26             hamming = hamming.tolist()
27             current_array = current_array.tolist()
28
29             len_of_lists = len(current_array)
30             for i in range(len_of_lists):
31                 tmp_compute = current_array[i][0]*hamming[i]
32                 temporary_hamming_window.append(tmp_compute)
33             self.hamming_window_list.append(temporary_hamming_window)
34         print(f'Hamming window created {[self.data_type]}')
35
36     # returning hamming window list
37     def get_hamming_window_list(self):
38         return self.hamming_window_list
39
40     # creating wavelet packet
41     def create_wavelet_packet(self):
42         for item in self.hamming_window_list:
43             item = np.array(item)
44             wp = pywt.WaveletPacket(data=item, wavelet='db1', mode='
```

```
                ↪ symmetric', maxlevel=4)
45              self.wavelet_list.append(wp['aaaa'].data)
46              self.wavelet_list.append(wp['aaad'].data)
47              self.wavelet_list.append(wp['aada'].data)
48              self.wavelet_list.append(wp['aadd'].data)
49              self.wavelet_list.append(wp['adaa'].data)
50              self.wavelet_list.append(wp['adad'].data)
51              self.wavelet_list.append(wp['adda'].data)
52              self.wavelet_list.append(wp['addd'].data)
53              self.wavelet_list.append(wp['daaa'].data)
54              self.wavelet_list.append(wp['daad'].data)
55              self.wavelet_list.append(wp['dada'].data)
56              self.wavelet_list.append(wp['dadd'].data)
57              self.wavelet_list.append(wp['ddaa'].data)
58              self.wavelet_list.append(wp['ddad'].data)
59              self.wavelet_list.append(wp['ddda'].data)
60              self.wavelet_list.append(wp['dddd'].data)
61
62          self.standard_deviation()
63          print(f'Wavelet packet created + calculated std. deviation from
                ↪ {[self.data_type]}')
64
65      # Returning standard deviation
66      def get_std_deviation_list(self):
67          return self.std_deviation_list
68
69      # Returning target. This is required to sklearn's models.
70      # Objects could be either healthy (1) or faulty (0)
71      def get_target(self):
72          num_of_targets = len(self.std_deviation_list)
73          if self.data_type == "healthy":
74              target = 1
75          elif self.data_type == "faulty":
76              target = 0
77          for i in range(num_of_targets):
78              self.target_list.append(target)
79
80          return self.target_list
```

3.4 Plik prediction.py

Plik ten definiuje funkcję, która tworzy model i korzystając z algorytmu random forest dokonuje predykcji czy dane charakteryzują wirnik 'zdrowy' czy 'wadliwy'. Dodatkowo wyświetlana jest wiadomość określająca dokładność naszego modelu a także graficzna jej interpretacja.

```
1  import numpy as np
2  from sklearn.model_selection import train_test_split
3  from sklearn.ensemble import RandomForestClassifier
4  from sklearn.metrics import accuracy_score, confusion_matrix
5  import matplotlib.pyplot as plt
6  from sklearn.metrics import plot_confusion_matrix
7
8
9  def predict_values(healthy, faulty):
10     # DATA - a collection of features from each window. In the basic
           ↪ case, we will take them out of the hamming window
11     # Target - a set of healthy or faulty features (written in binary
           ↪ as 1 or 0)
12
13     # Creating DATA and TARGET variables.
14     DATA = healthy.get_std_deviation_list() + faulty.
           ↪ get_std_deviation_list()
15     TARGET = healthy.get_target() + faulty.get_target()
16     DATA = np.reshape(DATA, (len(DATA), 1))
17
18     # Creating train and test variables and then random forest model.
19     X_train, X_test, y_train, y_test = train_test_split(DATA, TARGET,
           ↪ test_size=0.3)
20     model = RandomForestClassifier(n_estimators=20)
21     classifier = model.fit(X_train, y_train)
22     model.score(X_test, y_test)
23
24     # Predicting data and calculating it's accuracy
25     y_predict = model.predict(X_test)
26     acc = accuracy_score(y_test, y_predict)
27     print(f'accuracy: {round(acc*100, 2)} % \n')
28     cm = confusion_matrix(y_test, y_predict)
29
30     # Creating plots to visualize data
31     class_names = ['healty', 'faulty']
32     titles_options = [("Confusion matrix, without normalization", None)
           ↪ ,
33                       ("Normalized confusion matrix", 'true')]
34     for title, normalize in titles_options:
35         disp = plot_confusion_matrix(classifier, X_test, y_test,
               ↪ display_labels=class_names, cmap=plt.cm.Blues,normalize=
               ↪ normalize)
36         disp.ax_.set_title(title)
37         print(title)
38         print(disp.confusion_matrix)
39     plt.show()
```

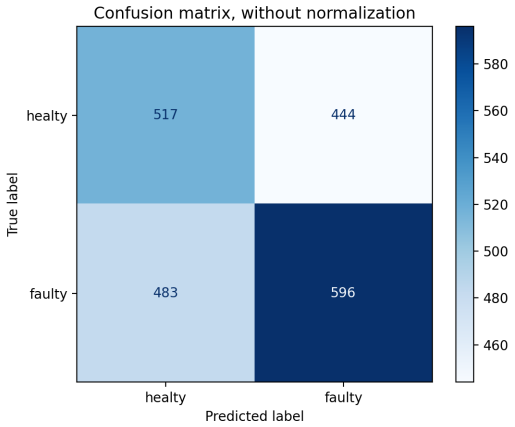
4 Wyniki działania programu

W sekcji tej opiszemy wyniki które uzyskaliśmy za pomcą stworoznego przez nas programu. Gdy program działa, kolejno wypisuje informacje, jaką prace aktualnie wykonał.

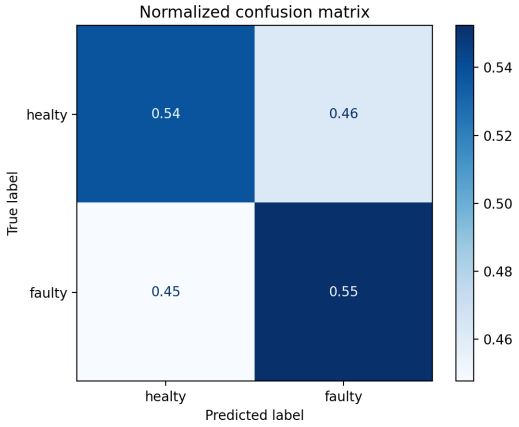
```
Downloaded 225 files in healthy data completed
Downloaded 200 files in faulty data completed
Hamming window created ['healthy']
Hamming window created ['faulty']
Wavelet packet created + calculated std. deviation from ['healthy']
Wavelet packet created + calculated std. deviation from ['faulty']
accuracy: 54.56 %

Confusion matrix, without normalization
[[517 444]
 [483 596]]
Normalized confusion matrix
[[0.53798127 0.46201873]
 [0.4476367  0.5523633 ]]
```

Rysunek 1: Informacja zwrotna podczas wykonywania programu

Program najpierw pobiera dane z folderów i zapisuje je do odpowiednich obiektów. Następnie liczone jest okno Hamminga oraz tworzone są Wavelet a także liczone jest odchylenie standardowe. 2 ostatnie operacje mają na celu wyznaczenie pewnych cech charakterystycznych, które umożliwią identyfikację sygnału.
Następnie program informuje, z jaką dokładnością model jest w stanie wskazać śmigło uszkodzone. Ostatecznie tworzone są macierze wynikowe, które dodatkowo prezentowane są w formie graficznej.



Rysunek 2: Macierz wynikowa bez normalizacji - interpetacja graficzna



Rysunek 3: Macierz wynikowa z normalizacją - interpetacja graficzna

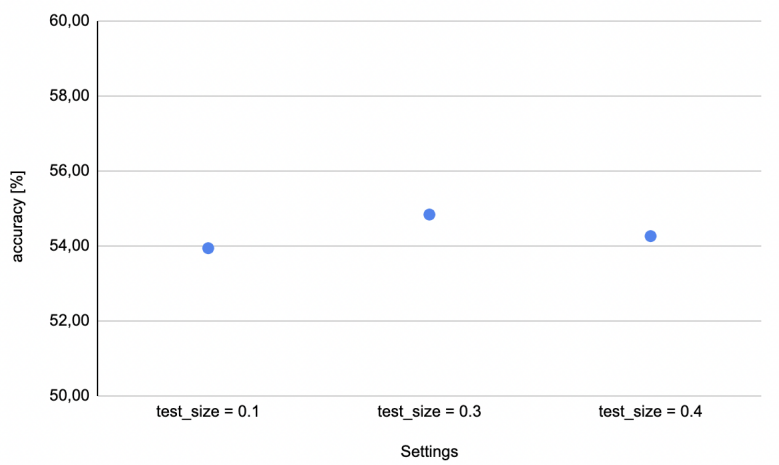
Interpretacja otrzymanych wyników nie jest zadowalająca. Średnio uzyskujemy wynik dokładności 55%. W przypadku gdy do wyboru mamy 2 możliwości i finalnie otrzymujemy dokładność modelu oscylująca w okolicach 55% nie możemy uznać naszego modelu za przydatny.

5 Dobranie odpowiednich nastaw

Postanowiliśmy zmodyfikować kod w ten sposób, aby uzyskać lepsze wyniki. Chcemy sprawdzić, czy możliwe jest zwiększenie skuteczności nastaw.

5.1 Zmiana wielkości grupy testowej

Zmiana tego parametru ma na celu zwiększenie lub zmniejszenie grupy testowej podczas tworzenia modelu. Przykładowo zapis test_size = 0.3 oznacza, że 30% danych zostanie użytych jako dane testowe a pozostałe 70% będzie służyć do wyuczenia modelu.
Dane zbieraliśmy poprzez kilkukrotne tworzenie modelu a następnie sprawdzanie jaka jest końcowa dokładność. Wyniki tego badania prezentujemy na poniższym wykresie.



Rysunek 4: Wykres przedstawiający zależność wielkości grupy testowej od uzyskanej dokładności modelu

Analiza powyższego przypadku nie wyznacza w jednoznaczny sposób która wielkość test_size jest najbardziej optymalna. Najlepszy wynik został uzyskany dla próbki o wielkości 30% i wynosił 54,84%. W przypadku modelu bazowego charakteryzującego się następującymi parametrami: test_size = 0.2, n_trees = 20, wavelet_max_level = 4 oraz sygnał przemnożony przez okno Hamminga. Dla tego podstawowego przypadku uzyskaliśmy dokładność równą 53,93% co w rzeczywistości daje nam różnicę równą około 1.5%.
Dodatkowo należy wziąć pod uwagę, że poszczególne odchyły pomiaru dla tych samych ustawień mogą sięgnąć nawet około 5 punktów procentowych.
Reasumując, wyniki badań wskazują, że w tym rpzypadku wielkość test_size nie ma dużego znaczenia w uzyskaniu znacznie lepszej skuteczności. Dlatego właśnie do dalszych badań nie zmieniamy wielkości parametru test_size i nadal przyjmuje on wartość 20%.

Poniżej prezentujemy wyniki uzyskane dla zmodyfikowanego programu. Modyfikacja polega na wielokrotnym tworzeniu modelu oraz wyświetlaniu jego skuteczności. W ten sposób chcemy przedstawić jak wyglądają średnie wahania dla modeli z tymi samymi ustawieniami (w tym przypadku wspomniane wyżej ustawienia modelu bazowego).

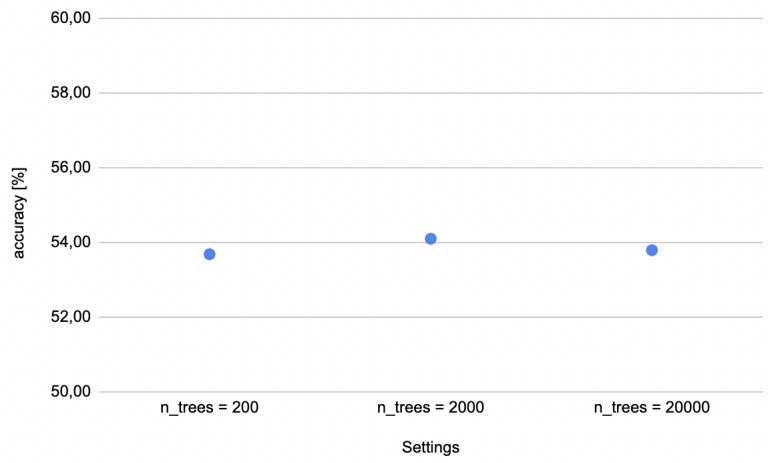
```
accuracy: 52.79 %
accuracy: 52.99 %
accuracy: 54.41 %
accuracy: 54.22 %
accuracy: 53.97 %
accuracy: 56.03 %
accuracy: 52.89 %
accuracy: 53.92 %
accuracy: 52.99 %
accuracy: 54.8 %
accuracy: 56.32 %
accuracy: 55.39 %
accuracy: 55.98 %
accuracy: 54.95 %
accuracy: 54.71 %
```

Rysunek 5: Wykres przedstawiający wahania dokładności uzyskanej dla modeli z tymi samymi ustawieniami

Wykres ten idealnie obrazuje fakt, że za każdym razem model charakteryzuje się nieco inną dokładnością. W przypadku przedstawionym na powyższym rysunku obserwujemy wahania wynoszące blisko 4 punkty procentowe.

5.2 Liczba drzew

Zmiana parametru liczby drzew polega na zwiększeniu lub zmniejszeniu liczby drzew decyzyjnych w konstruowanym modelu. W naszym przypadku tak jak to jest wspomniane powyżej zaczęliśmy obliczenia od 20 drzew. Następnie systematycznie zwiększyliśmy liczbę drzew badając jak wpływa to na uzyskaną dokładność.



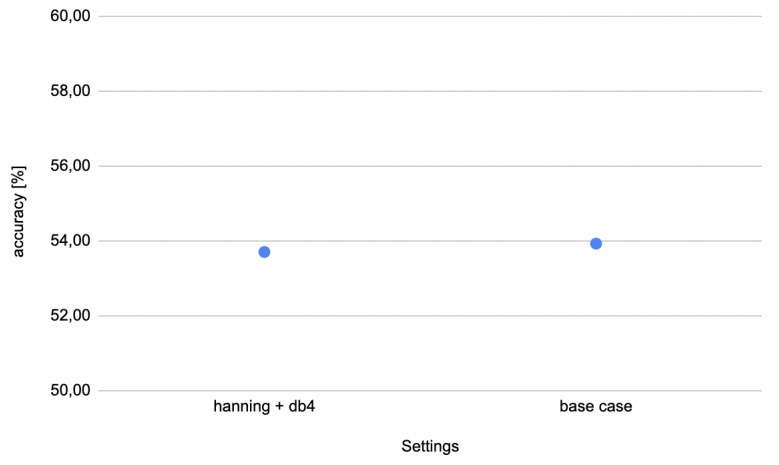
Rysunek 6: Wykres przedstawiający zależność liczby drzew od uzyskanej dokładności modelu

W tym przypadku najlepszy wynik został uzyskany dla 2000 drzew, gdzie skuteczność wynosiła 54,1%. Podobnie jak w poprzednim przypadku uzyskana dokładność jest znacząco niższa niż średnie wachania

uzyskane w obrębie 1 ustawien modelu. Pozwala nam to na stwierdzenie, że w naszym przypadku liczba drzew nie ma znaczącego wpływu na uzyskaną dokładność. Należy jednak nadmienić, że większa liczba drzew nie została sprawdzona z powodu ograniczeń sprzętowych. Nie możemy jednoznacznie wykluczyć teori, że przy zastosowaniu znacząco większej liczby drzew wynik nie ulegnie poprawie. Jednkaże mozliwośc ta jest małoprawodpobna gdyż wszytskie dotychczasowe wyniki oscylowały w okolicach podobnych wartości.

5.3 Inny typ okna

Podczas badań domyślnie skorzystaliśmy z okna Hamminga. W przypadku poszukiwań odpowiednich nastaw modelu w celu uzyskania jak najlepszej dokładności skorzystaliśmy również z okna Hanninga.

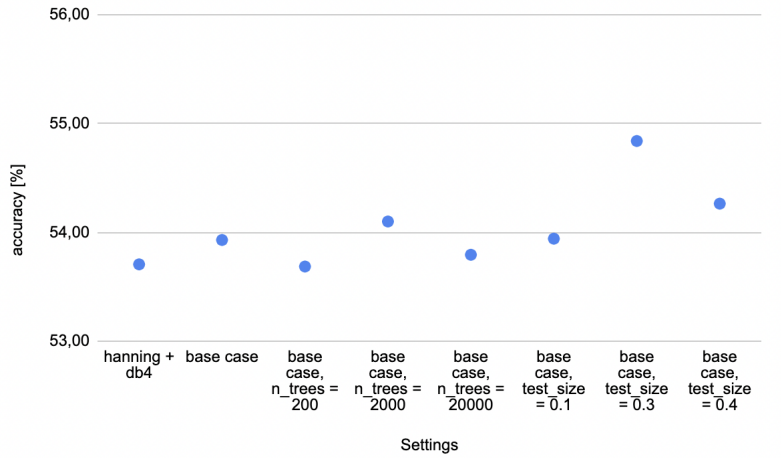


Rysunek 7: Wykres przedstawiający zależność typu okna od uzyskanej dokładności modelu

Analiza wyników również w tym przypadku nie pozwala na dokładne stwierdzenie, czy typ okna ma wpływ na uzyskana końcowa dokładność. Dokładniejsza obserwacja rysunku nr. 6 pozwala nam na stwierdzenie, że różnica pomiędzy dokładnością uzyskana za pomocą różnych typów okien jest znacząco mniejsza niż średnie wahania uzyskiwane dla modeli z tymi samymi ustawieniami.

5.4 Porównanie końcowe wszystkich ustawień

Punkt ten prezentuje zebrane wszystkie omawiane wyżej ustawienia.



Rysunek 8: Wykres przedstawiający zależność poszczegolnych ustawień od uzyskanej dokładności modelu

Dokładna analiza uzyskanych wyników jasno obrazuje, że wyniki rozrzucone są bardzo blisko siebie. Nie obserwujemy jednoznacznej różnicy która byłaby większa od średnich wahań w danym modelu z tymi samymi powiadomieniami. Niestety oznacza to, że problemem z uzyskaniem wyższej dokładności nie jest dobranie odpowiednich ustawień modelu a inny czynnik.

6 Podsumowanie wyników pracy

Badania modelu a także dobór odpowiednich ustawień nie umożliwił nam uzyskania zadowalającej dokładności. Ogólnie badając wartość średnią z wszystkich pomiarów uzyskujemy dokładność na poziomie 54,03%. Dokładność uzyskana dla naszego podstawowego modelu wynosiła 53,93%.
Specyfika dokonywanego wyboru umożliwia nam sprawne korzystanie z modelu gdyż w przypadku wyboru 1 z 2 opcji dokładność rzędu około 54% oznacza, że jest to wybór bliski losowości.

7 Bibliografia

Do jak najlepszego sformułowania tej pracy posłużyliśmy się dostępnymi stronami internetowymi, które wymieniamy poniżej:

- www.pywavelets.readthedocs.io/en/latest/
- www.scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html
- www.github.com/BrzozowskiPawel/Projekt_AI_wirnik