Wykrywanie Dna Oka

Informatyka w Medycynie – projekt

Wykonanie: Paulina Pacura, 142179

1. Przedstawienie problemu oraz wybranych metod rozwiązania

Stworzony program dzieli się na dwie części. Pierwsza to manualne przetwarzanie obrazu w celu wyodrębnienia od tła żył widocznych na zdjęciu oka. Na tak wykryte żyły nałożona zostaje maska, która następnie porównywana jest z maską ekspercką z bazy obrazów.

Druga część to budowa klasyfikatora, który na podstawie cech wyekstraktowanych z obrazu nauczy się automatycznej detekcji żył. W programie użyta została gotowa implementacja klasyfikatora (scikit-learn: **Random Forest Classifier**).

Jako interfejsu aplikacji użyty został **Jupyter Notebook** z wykorzystaniem interaktywnych widgetów dla umożliwienia modyfikowania pewnych parametrów przez użytkownika. Program napisany został w języku **Python**.

W programie wykorzystano bazę obrazów **HRF**.

Zastosowane biblioteki:

- Imblearn
- Opeccv
- Matplotlib
- Numpy
- Pandas
- Skimage
- Sklearn

2. Opis głównych funkcji programu

Użytkownik programu wybiera obraz, który ma zostać przetworzony, a następnie przy pomocy przycisków wybiera kolejne kroki działania programu. W głównym pliku programu znajdują się wyłącznie interaktywne widgety; wszystkie funkcje znajdują się w osobnym notebooku w celu uproszczenia interfejsu użytkownika. Są one podzielone na klasy: klasa ImageReader odpowiedzialna za wczytywanie obrazów, klasa ImageProcessor odpowiedzialna za „manualną" część przetwarzania obrazu oraz klasa ImageClassifier z uczeniem maszynowym.

2.1. Przetwarzanie obrazów

Manualne przetwarzanie obrazów podzielone jest na część wstępną, właściwą i końcową. W części wstępnej *initial_processing* dokonuję manipulacji kanałami obrazu oraz jego wartością gamma w celu zwiększenia kontrastu między żyłami a tłem, użyłam również rozmycia gaussowskiego.

Część właściwa wykrywania dna podzielona jest na dwie funkcje. Pierwsza: *edge_detecting* dostaje na wejściu wstępne przetworzony obraz: wywołuje wykrywanie krawędzi przy pomocy *sato* i wykonuje na tak uzyskanym obrazie operacje biblioteki skimage.morphology:*dilatation* dla uwydatnienia najmniejszych żył, które mogłyby zaginąć, *closing* by połączyć ewentualne ubytki w kanałach żył, *erosion* by zniwelować nadmierne skutki wcześniejszego pogrubienia. Tak przygotowany obraz poddawany jest *threshold'owi*. W tej funkcji wykorzystuję także maskę aby pozbyć się ewentualnych szumów z tła wokół faktycznego oka.

Druga z funkcji: *vessels_mask* wykrywania właściwego to faktyczne tworzenie maski na żyłach. Funkcja przechodzi po każdym z pikseli czarno białego już obrazka i w przypadku wykrycia bieli (czyli żyły) dodaje ten sam kolor w macierzy maski.

Końcowe przetwarzanie *fundus_final_processing* to powtórzenie operacji *erosion* oraz *closing*, a także ponowne użycie thresholdu na uzyskanym obrazie.

```
def initial_processing(self, img):
    lab = cv2.cvtColor(img, cv2.COLOR_BGR2LAB)
    l, a, b = cv2.split(lab)
    clahe = cv2.createCLAHE(clipLimit=3.0, tileGridSize=(8,8))
    cl = clahe.apply(l)
    merged = cv2.merge((cl,a,b))
    clahed_image = cv2.cvtColor(merged, cv2.COLOR_LAB2BGR)
    clahed_image[:, :, 0] = 0
    clahed_image[:, :, 2] = 0
    img=img_as_float(rgb2gray(clahed_image))
    img=gaussian(img,sigma=3)
    img=img**0.2

    mask=img_as_float(rgb2gray(self._official_mask[0]))
    img=img*mask

    return img
```

```
def edge_detecting(self,img):
    img=sato(img)
    img = (img - np.min(img)) / (np.max(img) - np.min(img))

    img=mp.dilation(img,selem=mp.disk(6))
    img=gaussian(img,sigma=3)
    img=mp.closing(img, selem=mp.disk(8))
    img=mp.erosion(img,selem=mp.disk(2))

    thresh = threshold_li(img, tolerance=0.0005)
    img = img > thresh

    mask=img_as_float(rgb2gray(self._official_mask[0]))
    mask=mp.erosion(mask, selem=mp.disk(5))
    img=img*mask

    return img
```

```
def vessels_mask_creating(self,img):
    detection = np.zeros((img.shape[0], img.shape[1], 3), np.uint8)

    for y in range(img.shape[0]):
        for x in range(img.shape[1]):
            if cv2.countNonZero(img[y, x]) > 0:
                detection[y, x] = np.array([255, 255, 255], np.uint8)

    return detection
```

```
def fundus_final_processing(self,img):
    img = np.array(img)
    img = 255*(cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) > 5).astype('uint8')

    img=mp.erosion(img,selem=mp.disk(1))
    img=mp.closing(img, selem=mp.disk(15))
    img = cv2.threshold(img, 100, 255, cv2.THRESH_BINARY)[1]

    return img
```

2.2. Uczenie maszynowe

Pierwsza część to ekstrakcja cech z wycinków wstępnie przetworzonego obrazu. Wraz z informacją pobraną z maski eksperckiej (decyzja dla środkowego piksela wycinka) wartości te umieszczane są w zbiorze danych dla klasyfikatora.

```
def split_image(self,img,expect, block_size):
    windows=[]
    decisions = []
    for r in range(0,img.shape[0] - block_size, block_size):
        for c in range(0,img.shape[0] - block_size, block_size):
            window = img[r:r+block_size,c:c+block_size]
            exp_window = expect[r:r+block_size,c:c+block_size]
            center = np.amax(exp_window[exp_window.shape[0]//2,exp_window.shape[1]//2])
            if center>50:
                center=1
            else:
                center=0
            decisions.append(center)
            windows.append(window)

    return windows,decisions
```

```
def image_features_extract(self, img, decision=None):
    data={}
    data['variance'] = np.var(img)
    img = img_as_ubyte(rgb2gray(img))
    moments = cv2.moments(img)
    data={**data,**moments}
    hu_moments = cv2.HuMoments(moments)
    for i,hu in enumerate(hu_moments):
        name = "hu"+str(i)
        data[name]=hu[0]
    data['decision']=decision
    data_for_pic = [np.var(img),moments,hu_moments]
    return data, data_for_pic
```

Zdecydowałam się na wykorzystanie algorytmu Random Forest, który polega na tworzeniu wielu drzew decyzyjnych, z których każde jest trenowane na podzbiorze danych. Pojedyncze drzewo to model, który uczy się mapować funkcje-*features* (w tym przypadku

wariancję obrazka i jego momenty centralne oraz momenty Hu) na decyzje (żyła czy tło?). Można przedstawić je jako schemat blokowy pytań prowadzących do prognozy. W przypadku Random Forest, każde drzewo uczy się na losowym podzbiorze funkcji podczas tworzenia pytań oraz ma dostęp tylko do losowego zestawu punktów danych treningowych. Zwiększa to różnorodność w lesie, prowadząc do bardziej solidnych i ogólnych przewidywań – las wydaje decyzje na podstawie większości „głosów".

W celu dobrania najlepszych zewnętrznych (ustawianych przez programistę) parametrów dla klasyfikatora ale również uniknięcia overfittingu wykorzystałam algorytm **RandomizedSearchCv**. Wykonuje on wiele iteracji ***K-Fold cross validation***, używając różnych, losowych ustawień dla modelu (korzystając ze zdefiniowanych wcześniej zakresów parametrów). Pozwala to na zawężenie zakresów dla każdego parametru, co następnie wykorzystuję w użyciu **GridSearchCV**: metody, która zamiast tworzyć losowe zestawienia, ocenia wszystkie zdefiniowane kombinacje.

```python
def find_RandomizedSearchCV_best_params(self):
    # Number of trees in random forest
    n_estimators = [int(x) for x in np.linspace(start = 200, stop = 2000, num = 10)]
    max_features = ['auto', 'sqrt']
    max_depth = [int(x) for x in np.linspace(10, 110, num = 11)]
    max_depth.append(None)
    min_samples_split = [2, 5, 10]   # Minimum number of samples required to split a node
    min_samples_leaf = [1, 2, 4]     # Minimum number of samples required at each leaf node
    bootstrap = [True, False]        # Method of selecting samples for training each tree

    random_grid = {'n_estimators': n_estimators,
                   'max_features': max_features,
                   'max_depth': max_depth,
                   'min_samples_split': min_samples_split,
                   'min_samples_leaf': min_samples_leaf,
                   'bootstrap': bootstrap }

    print("Using the random grid to search for best parameters")
    print("Random Grid options: \n")
    pprint(random_grid)

    rf = RandomForestClassifier()
    rf_random = RandomizedSearchCV(estimator = rf, param_distributions = random_grid,
                                   n_iter = 100, cv = 3, verbose=2, random_state=42,
                                   n_jobs = -1)
    # n_iter controls the number of different combinations to try
    # cv is the number of folds to use for cross validation

    X = self._X
    y = self._y
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33,stratify=y)

    rf_random.fit(X_train, y_train)
    best_random = rf_random.best_estimator_
```

```python
def find_GridSearchCV_best_params(self):
    param_grid = {
        'bootstrap': [True],
        'max_depth': [1, 10, 20, 30],
        'max_features': [2, 3],
        'min_samples_leaf': [3, 4, 5],
        'min_samples_split': [2, 8, 12],
        'n_estimators': [100, 200, 400, 800]
    }  #grid parameters based on RandomizedSearchCV best params

    rf = RandomForestClassifier()
    grid_search = GridSearchCV(estimator = rf, param_grid = param_grid,
                               cv = 3, n_jobs = -1, verbose = 2)

    X = self._X
    y = self._y
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33,stratify=y)

    grid_search.fit(X_train, y_train)
    best_grid = grid_search.best_estimator_
```

Uzyskane parametry:

```
Random best params:
{'bootstrap': True,
 'max_depth': 10,
 'max_features': 'sqrt',
 'min_samples_leaf': 4,
 'min_samples_split': 2,
 'n_estimators': 400}
```

```
Grid best params:
{'bootstrap': True,
 'max_depth': 30,
 'max_features': 3,
 'min_samples_leaf': 4,
 'min_samples_split': 2,
 'n_estimators': 200}
```

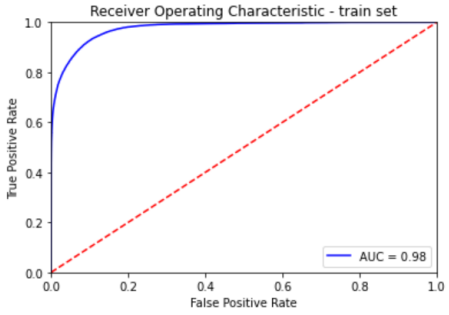
Następnie, mając już znalezione najlepsze ustawienia dla klasyfikatora następuje sprawdzenie klasyfikatora na zbiorze treningowym oraz testowym.

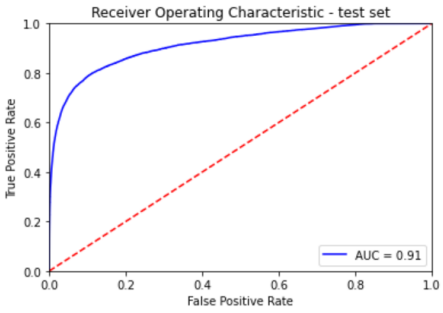
```python
@jit
def train_model(self,use_custom=False):
    X = self._X
    y = self._y

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33,stratify=y)
    display_label_distribution(y_train,y_test)

    if use_custom==True:
        bootstrap,n_estimators,criterion,max_depth,max_features,min_samples_leaf,min_samples_split = get_input_m
        rf_clf = RandomForestClassifier(bootstrap=bootstrap,n_estimators=n_estimators
                                        ,criterion=criterion,max_depth=max_depth,
                                        max_features=max_features,min_samples_leaf=min_samples_leaf,
                                        min_samples_split=min_samples_split)
    else:
        rf_clf = RandomForestClassifier(bootstrap=True,criterion='gini',
                                        max_depth=30, max_features=3,
                                        max_leaf_nodes=None, max_samples=None,
                                        min_samples_leaf=4, min_samples_split=2,
                                        n_estimators=100, n_jobs=None)

    rf_clf.fit(X_train, y_train)
```

Dla zobrazowania dokładności wykorzystałam wbudowany raport biblioteki *sklearn* oraz wykorzystałam krzywą ROC (na wykresie wypisana jest miara AUC oraz czerwoną linią zaznaczony wynik, który osiągnąłby losowy klasyfikator).

	precision	recall	f1-score	support
0	0.96	1.00	0.98	205608
1	0.95	0.62	0.75	22117
accuracy			0.96	227725
macro avg	0.95	0.81	0.86	227725
weighted avg	0.96	0.96	0.96	227725

	precision	recall	f1-score	support
0	0.95	0.99	0.97	101271
1	0.82	0.52	0.63	10893
accuracy			0.94	112164
macro avg	0.88	0.75	0.80	112164
weighted avg	0.94	0.94	0.94	112164





Po tym program wykonuje detekcję żył na wskazanym obrazie.

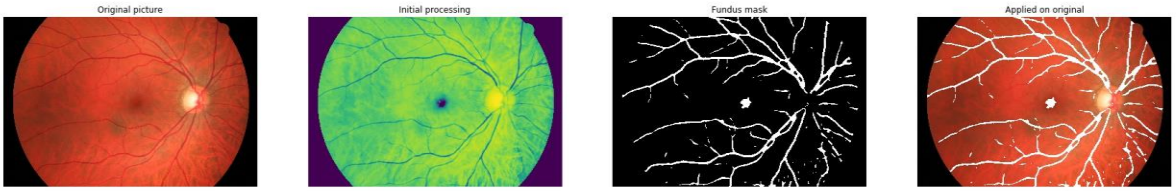
```
@jit
def vessels_mask_creating(self, img, drawing_range):
    block_size = block_size_input.value//2
    detection = np.zeros((img.shape[0], img.shape[1], 3), np.uint8)
    decisions=[]

    ###SPLITING IMAGE DRAWING INTO PARTS SO WON'T HAVE TO WAIT DAYS TO SEE RESULT
    x_start_point = drawing_range["x_start"]
    x_end_point = drawing_range["x_end"]
    y_start_point = drawing_range["y_start"]
    y_end_point = drawing_range["y_end"]

    print(x_end_point)
    maks_y = y_end_point-block_size-1

    for x in range(x_start_point + block_size, x_end_point - block_size):
        for y in range(y_start_point + block_size, y_end_point - block_size):
            window = img[x-block_size:x+block_size,y-block_size:y+block_size]

            _ ,data_set = self.image_features_extract(window)
            part_list = [data_set[0]]
            dict_list = list(data_set[1].values())
            for item in dict_list:
                part_list.append(item)
            for item in data_set[2]:
                part_list.append(item[0])
            decision = self._model.predict([part_list])
            decisions.append(decision)
            if y == maks_y:
                print(x,y)
            if decision!=0:
                detection[x, y] = np.array([255, 255, 255], np.uint8)
    display_picture(detection)
    return detection
```

3. Przedstawienie wyników działania symulacji

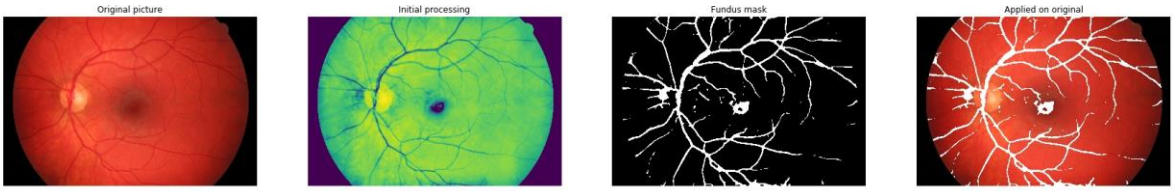
W dalszej części przedstawiam wybrane wyniki działania programu. W ramach wyników przedstawiam kolejne etapy manualnego przetwarzania obrazu i uzyskane tą metodą statystyki a następnie maski żył uzyskane przez klasyfikator oraz ich statystyki. Na końcu zestawiłam tabele z raportem miar. Pozostałe wyniki znajdują się w folderze *img/image-results*.

3.1. Manualne przetwarzanie obrazu

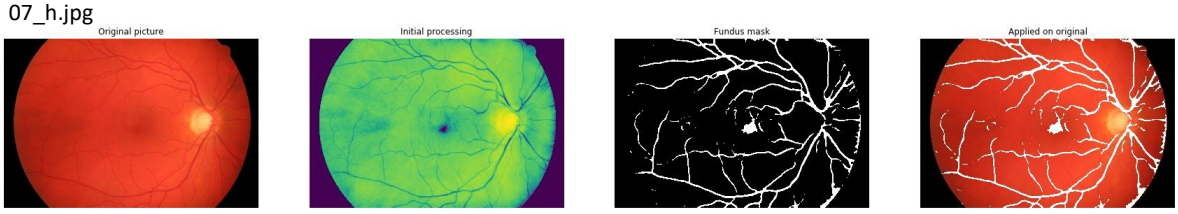
01_h.jpg



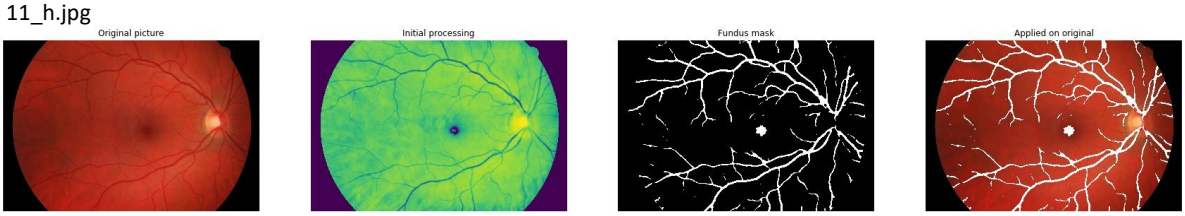
04_h.jpg



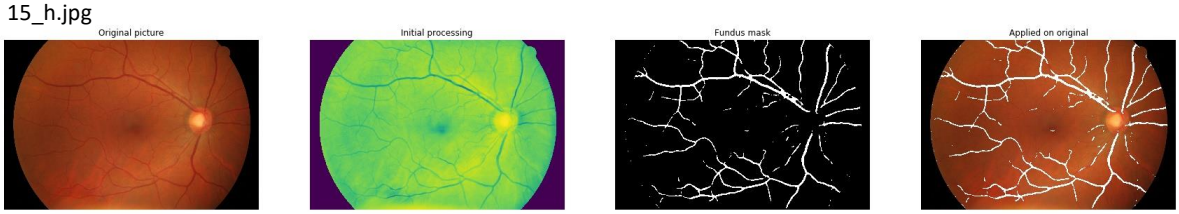
07_h.jpg



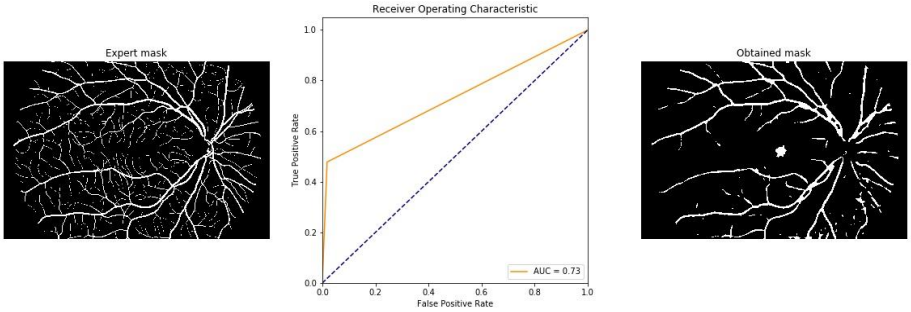
11_h.jpg



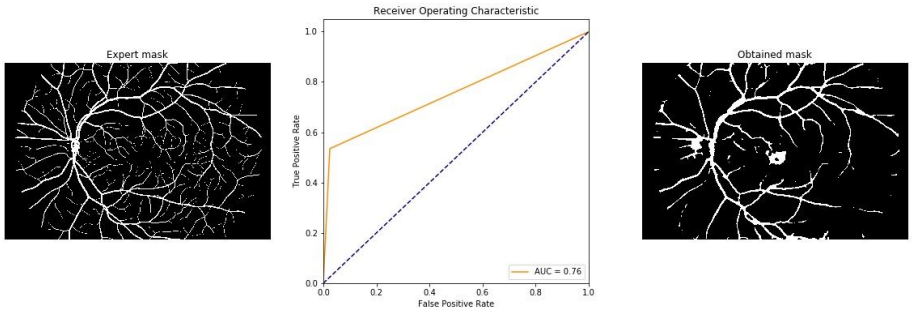
15_h.jpg



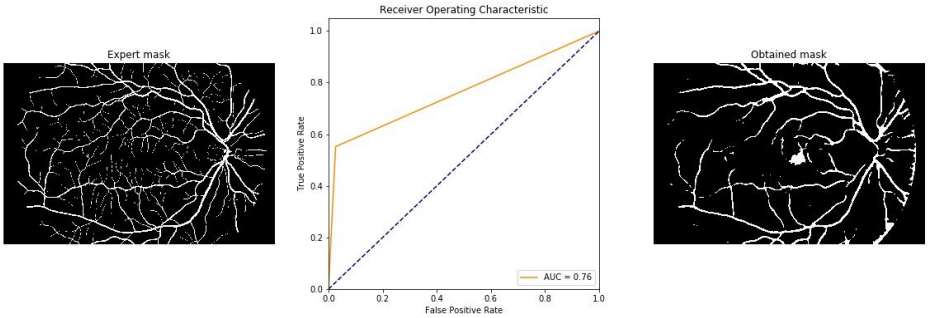
01_h.jpg



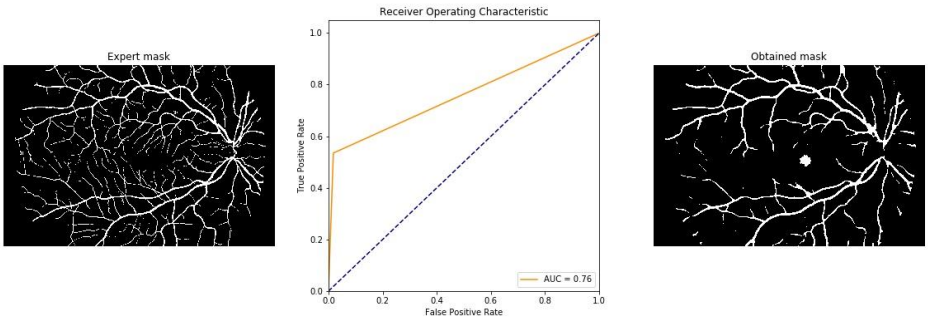
04_h.jpg



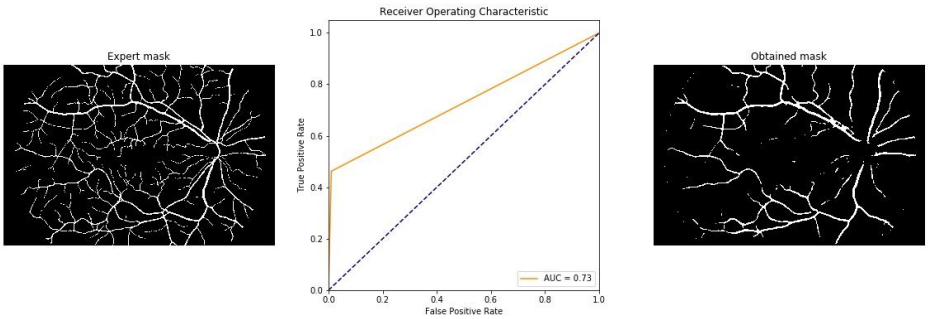
07_h.jpg



11_h.jpg

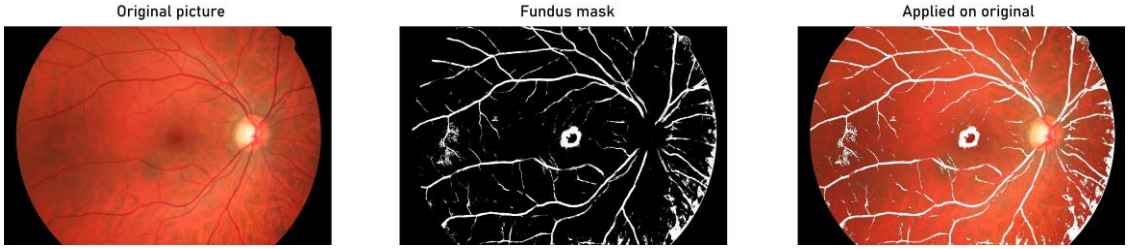


15_h.jpg

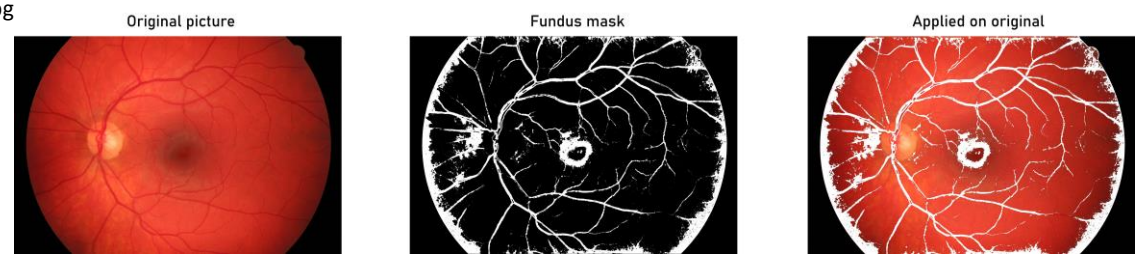


3.2. Wyniki uzyskane przez uczenie maszynowe

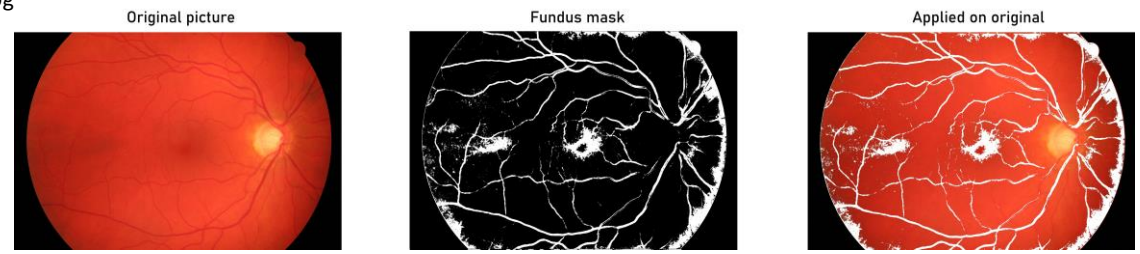
01_h.jpg



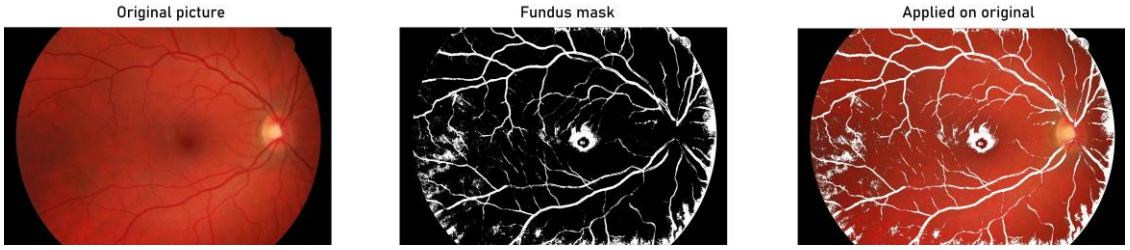
04_h.jpg



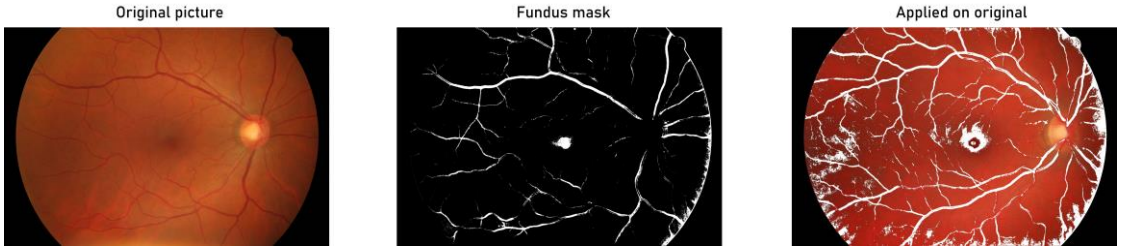
07_h.jpg



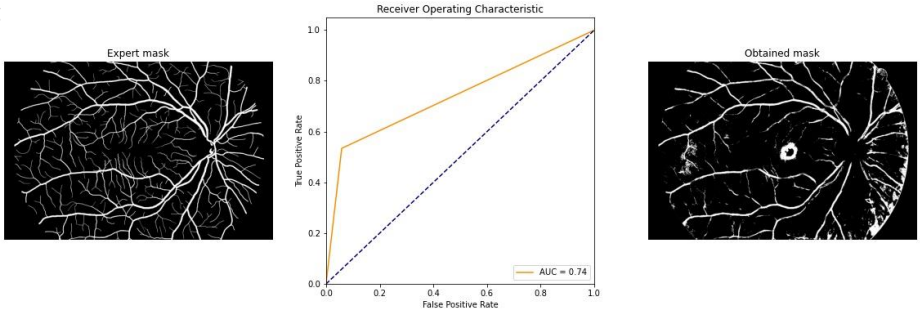
11_h.jpg



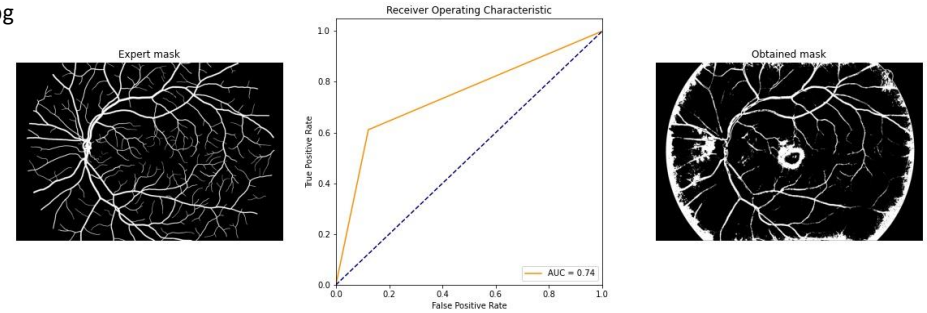
15_h.jpg



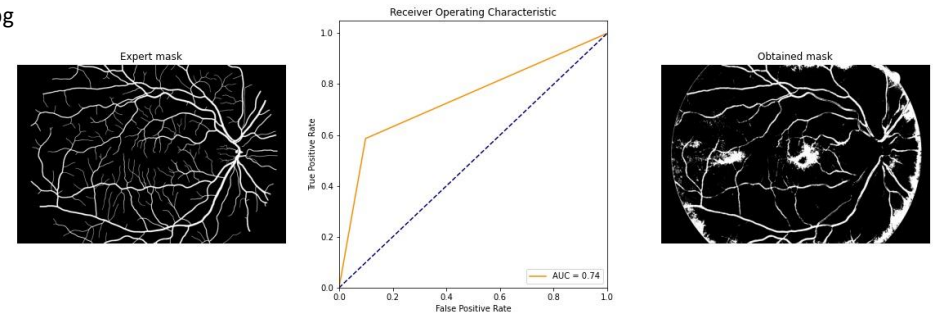
01_h.jpg



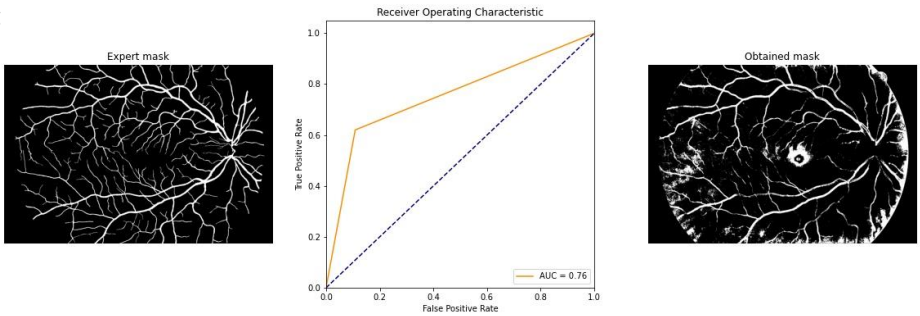
04_h.jpg



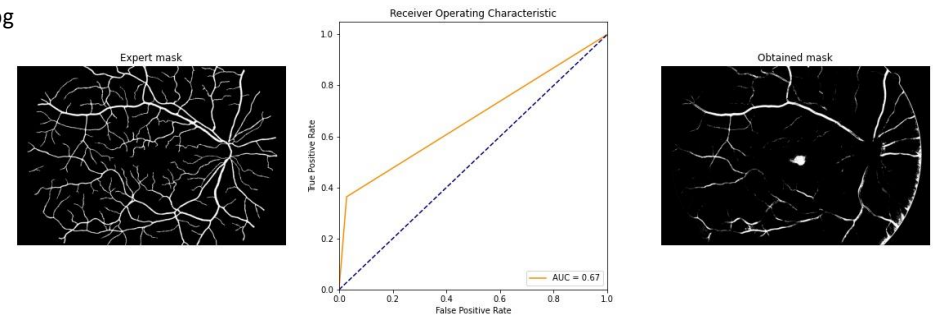
07_h.jpg



11_h.jpg



15_h.jpg



3.3. Zestawienie raportu miar

Tabela miar dla przedstawionych obrazów przetwarzanych manualnie:

	Accuracy	Sensitivity	Specificity	Geometric mean	Precision	Recall	F1
1	0.910454	0.812801	0.919328	0.864425	0.477972	0.812801	0.601958
4	0.918330	0.763376	0.933778	0.844288	0.534706	0.763376	0.628899
7	0.921864	0.744445	0.939606	0.836352	0.552092	0.744445	0.634000
11	0.924650	0.811838	0.935034	0.871261	0.534950	0.811838	0.644931
15	0.932342	0.840906	0.938132	0.888190	0.462534	0.840906	0.596802

Tabela miar dla przedstawionych obrazów przetwarzanych przez klasyfikator:

	Accuracy	Sensitivity	Specificity	Geometric mean	Precision	Recall	F1
1	0.883981	0.602065	0.924483	0.746055	0.533882	0.602065	0.565927
4	0.843597	0.42715	0.938244	0.633065	0.611195	0.42715	0.502862
7	0.862645	0.453388	0.93973	0.652735	0.586251	0.453388	0.51133
11	0.856647	0.455675	0.941156	0.654875	0.620073	0.455675	0.525312
15	0.906039	0.610893	0.926374	0.752274	0.363736	0.610893	0.455976

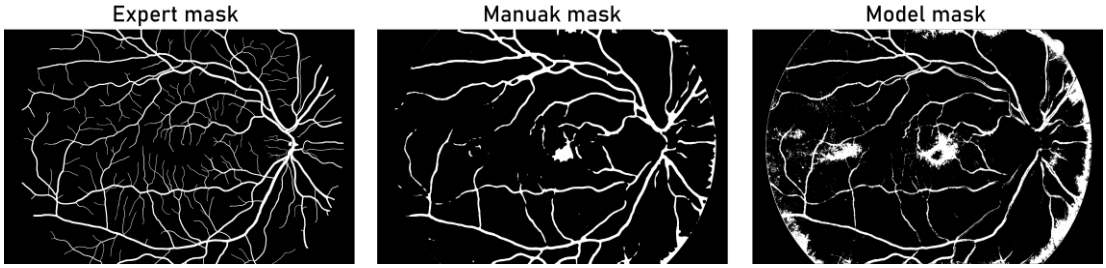
Trafność (accuracy) jest nieodpowiednią miarą dla danych niezrównoważonych (takich jak w tym przypadku, gdzie obiektów klasy negatywnej: tło jest znacznie więcej niż obiektów klasy pozytywnej: żyła, gdyż wystarczyłoby w tym przypadku wygenerowanie czarnego obrazka, by uzyskać dość wysoki poziom trafności (większość pikseli na masce eksperckiej to czarne tło). Miary, które lepiej mogą sprawdzić się przy klasyfikacji danych niezrównoważonych skupiają się na pojedynczych klasach: czułość – swoistość (sensitivity – specificity) oraz precision – recall (prezycja – czułość). Czułość oznacza, jak dobrze klasa pozytywna była rozpoznawana: odsetek prawdziwie pozytywnych. Swoistość to jakość rozpoznawania klasy negatywnej: odsetek prawdziwie negatywnych. Czułość i swoistość połączyć można we wspólną miarę: średnią geometryczną. Ta miara próbuje zmaksymalizować dokładność każdej z klas, jednocześnie utrzymując równowagę tych dokładności. Precyzja podsumowuje, ile próbek zakwalifikowanych do klasy pozytywnej faktycznie ją reprezentuje. Precyzja i czułość również mogą być połączone we wspólną miarę: F-score.

Z przedstawionych wyników wynika, że klasyfikator gorzej rozpoznawał żyły, minimalnie lepiej jednak radził sobie z wykrywaniem tła. Był też bardziej precyzyjny: częściej jeśli klasyfikował piksel jako żyłę, to faktycznie ten stan się zgadzał. Ostatecznie jednak wyniki zarówno g-mean, jak i f-score okazały się niższe niż wyniki osiągnięte przetwarzaniem manualnym. Obrazek 15_h okazał się największym problemem dla klasyfikatora, który

osiągnął w nim najgorsze rezultaty: być może ma to związek ze znaczną różnicą barw występującą między wskazanym obrazem, a wszystkimi poprzednimi.

Oprócz tych miar, w projekcie korzystałam również z krzywej ROC oraz pola pod krzywą: AUC. W ostatecznym zestawieniu uzyskane krzywe są w uproszczonej formie z powodu braku badania różnych punktów odcięcia. Wykresy przedstawiają zmienność TPR (miary rozpoznania klasy faktycznie pozytywnej) w zależności od FPR (miary błędu popełnianego na klasie negatywnej). Im większe AUC, tym lepiej: 1 oznaczałoby klasyfikator idealny, 0.5 klasyfikator losowy. Trenowany klasyfikator uzyskał tu zbliżone wyniki z przetwarzaniem manualnym, jedynie obraz 15_h spadł poniżej 0.7 AUC.

Dodatkowo, sądzę że oprócz podejścia statystycznego warto zwrócić uwagę na odbiór generowanych obrazów przez człowieka. Maski uzyskane metodą manualną wydają się „toporne" i graficzne, pozbawione detali, o małym zróżnicowaniu grubości żył. Z kolei maski generowane przez klasyfikator cechowały się zróżnicowanymi grubościami żył nadającymi bardziej zbliżony do oryginału efekt wizualny, rozpoznawane były też żyły cieńsze i mniejsze. Poniżej przykładowe zestawienie mask wygenerowanych dla obazu 07_h.jpg :



4. Wnioski

4.1. Użycie RandomizedSearchCv z uwagi na wybrany zakres parametrów okazało się bardzo czasochłonne, mimo równoległego systemu działania algorytmu. Na gorszym procesorze zajęło 10 godzin, na lepszym 2.5 godziny.

4.2. Przy ostatecznej detekcji żył na obrazie zmuszona byłam obniżyć ilość drzew w klasyfikorze w stosunku do wskazań GridSearchCV z powodu czasu, który zajmowało przetwarzanie obrazu.

4.3. Przedstawiony w raporcie klasyfikator trenowany był na części wycinków obrazu 01_h i uzyskał dla niego lepszy wynik AUC niż przetwarzanie manualne. Mimo to osiągnął niższą wartość miary sensitivity. Dla pozostałych wybranych do zestawienia obrazów ta sytuacja powtórzyła się: w większości przypadków klasyfikator uzyskał zbliżone wyniki z rezultatami przetwarzania manualnego z gorszym sensitivity, lecz wyższym specificity.

4.4. Klasyfikator użyty został do wygenerowania pięciu wybranych do tego celu obrazów (te same obrazy przedstawiono w raporcie dla przetwarzania manualnego).