Szkoła Główna Gospodarstwa Wiejskiego w Warszawie

Wydział Zastosowań Informatyki i Matematyki

Sylwester Turski

167497

Algorytm do rozpoznawania wolnych miejsc parkingowych na podstawie serii obrazów z kamery cyfrowej

Algorithm to recognize free parking places based on images from a digital camera

Praca dyplomowa inżynierska

na kierunku informatyka

Praca wykonana pod kierunkiem

Dra Pawła Hosera

Katedra Zastosowań Informatyki

Warszawa, 2017 rok

**Oświadczenie promotora pracy**

Oświadczam, że niniejsza praca\*/wskazane przez autora rozdziały pracy dyplomowej przygotowanej zespołowo\* została/zostały\* przygotowana pod moim kierunkiem   
i stwierdzam, że spełnia\*/spełniają\* warunki do przedstawienia tej pracy w postępowaniu   
o nadanie tytułu zawodowego.

Data .................................... Podpis promotora pracy ...................................................

**Oświadczenie autora pracy**

Świadom odpowiedzialności prawnej, w tym odpowiedzialności karnej za złożenie fałszywego oświadczenia, oświadczam, że niniejsza praca dyplomowazostała napisanaprzeze mnie samodzielnie i nie zawieratreści uzyskanych   
w sposób niezgodny z obowiązującymi przepisami prawa, w szczególności ustawą z dnia   
4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (Dz. U. Nr 90 poz. 631 z późn. zm.)

Oświadczam, że przedstawiona praca nie była wcześniej podstawą żadnej procedury związanej z nadaniem dyplomu lub uzyskaniem tytułu zawodowego.

Oświadczam, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Przyjmuję do wiadomości, że praca dyplomowa poddana zostanie procedurze antyplagiatowej.

Data ..................................... Podpis autora pracy .............................................

**Streszczenie**

**Inteligentny algorytm do rozpoznawania i zliczania miejsc parkingowych na podstawie serii obrazów z kamery**

Streszczenie pracy - do 1000 znaków (12)

Słowa kluczowe - do 6 (12)

**Summary** (12 bold)

**Title** (12 bold)

Summary - 1000 words max (12)

Keywords – 6 max (12)

Spis Treści

[1. Wstęp 8](#_Toc472073821)

[2. Cel i zakres pracy 8](#_Toc472073822)

[3. Przegląd piśmiennictwa 8](#_Toc472073823)

[4. Projekt rozwiązania 8](#_Toc472073824)

[4.1. Biblioteki i technologie użyte podczas pisania algorytmu 8](#_Toc472073825)

[4.2. Makieta parkingu 9](#_Toc472073826)

[4.3. Oznaczanie konturów dla miejsc parkingowych 10](#_Toc472073827)

[4.4. Badane cechy obrazu 12](#_Toc472073828)

[4.5. Klasyfikator 20](#_Toc472073829)

[4.6. Walidacja jakości klasyfikacji 23](#_Toc472073830)

[5. Omówienie i dyskusja wyników 26](#_Toc472073831)

[6. Literatura 26](#_Toc472073832)

[7. Spis obrazków 26](#_Toc472073833)

[8. Spis listingów 27](#_Toc472073834)

# Wstęp

# Cel i zakres pracy

Celem pracy jest stworzenie algorytmu, który będzie w stanie sklasyfikować miejsce parkingowe zaznaczone na zdjęciu jako wolne lub zajęte. Do algorytmu będzie dostarczany również zbiór uczący, składający się ze zdjęć parkingu wraz z oznaczeniami konturu gdzie znajdują się miejsca parkingowe i czy są one zajęte czy wolne.

Dodatkowo w zakres pracy wchodzi przygotowanie makiety parkingu i przygotowanie testowych zdjęć. Przygotowanie programu ułatwiającego oznaczanie konturów miejsc parkingowych i ich statusu. Przygotowanie programu testującego skuteczność algorytmu.

# Przegląd piśmiennictwa

# Projekt rozwiązania

## Biblioteki i technologie użyte podczas pisania algorytmu

OpenCV (Open Source Computer Vision) – Popularna biblioteka funkcji do rozpoznawania obrazów w czasie rzeczywistym. Posiada między innymi funkcje do manipulacji obrazami, wykrywania cech obrazu i uczenia maszynowego. W 1999 roku projekt OpenCV został zainicjalizowany przez firmę Intel. Biblioteka jest napisana w języku C++. Jako wrapper[[1]](#footnote-1) dla platformy .NET wybrałem bibliotekę OpenCvSharp. API tej biblioteki jest bardzo zbliżone do oryginalnego API w języku C++. Wiele klas ma zaimplementowany interfejs *IDisposable*, dzięki czemu nie trzeba się martwić zwalnianiem pamięci obiektów z poza platformy .NET. Biblioteka dodatkowo posiada API które pozwana na wywołanie łańcuchowe metod[[2]](#footnote-2).

WPF (Windows Presentation Foundation) – Framework do tworzenia interfejsu użytkownika (ang. user interface) dla platformy .NET, stworzony przez firmę Microsoft. WPF kładzie nacisk na grafikę wektorową, dzięki której większość kontrolek można skalować bez utraty jakości, czy pikselizacji. Wybór na tą technologię zapadł ponieważ można w łatwy sposób projektować wygląd aplikacji, który jest zarazem elastyczny i dopasowuje się do wielkości okna. WPF pozwana na tworzenie aplikacji przy użyciu wzorca MVVM który pozwala na separację kodu odpowiedzialnego za widok od logiki aplikacji. Nazwa MVVM pochodzi o inicjałów Model, Widok (ang. View), Model Widoku (ang. View Model). Jako model najczęściej określa się warstwę biznesową aplikacji lub warstwę dostępu do danych. Model widoku jest odpowiedzialny za komunikację widoku z modelem, poprzez przygotowanie danych z modelu dla widoku oraz udostępnia komendy które służą do interakcji przez użytkownika. Widok jest to część aplikacji która jest odpowiedzialna za prezentację wizualną aplikacji, wyświetlanie danych i obsługę interakcji użytkownika za pomocą zdefiniowanych komend zdefiniowanych w modelu widoku.

## Makieta parkingu

Pierwsza wersja makiety parkingu była zrobiona z czarnej bibuły zawiniętej i oklejonej na sztywnej podkładce. Pasy miejsc parkingowych zostały narysowane korektorem w taśmie. Faktura bibuły miała odwzorowywać fakturę asfaltu. Jako modelu samochodów zostały użyte popularne zabawkowe resoraki w kilku kolorach, i jeden resorak w kolorze czarnym, który z założenia miał zlewać się z kolorem podłoża i stanowić przypadek pesymistyczny. Zdjęcia robione były przy użyciu kamery internetowej w rozdzielczości 640x480, model ustawiany był pod różnymi obrotami względem kamery. Zdjęcia były tak dobierane aby były przypadki optymistyczne, gdzie samochód pozostaje w swoim miejscu parkingowym i takie gdzie kawałek samochodu zasłania kawałek miejsca parkingowego obok.

Takie podejście powodowało problemy. Faktura bibuły była wyraźnie widoczna ze względy na bliską odległość kamery od modelu, pomięta faktura bibuły utrudniała również rysowanie pasów korektorem w taśmie. Słabe oświetlenie sceny powodowało iż samochody o kolorze czarnym były słabo widoczne dla ludzkiego oka na obrazach. Doświetlanie sceny powodowało że samochody o kolorze czarnym były zaczynały być widoczne, a algorytm wykrywania krawędzi zaczynał wykrywać krawędzie na samochodzie, Jednakże algorytm wykrywania krawędzi znacząco więcej wykrywał pomarszczoną fakturę bibuły, a samochody kolorowe zaczynały być prześwietlone. Kamera przy słabej jakości oświetlenia zaczęła generować szum cyfrowy. Całość tych niedogodności wymagała bardzo precyzyjnego ustawienia oświetlenia sceny.

Druga wersja makiety powstała na brystolu o jednolitym ciemnoszarym kolorze. Z narysowanymi pasami korektorem w taśmie, zestaw samochodów resorków pozostał ten sam. Zdjęcia zostały wykonane aparatem telefonu komórkowego o rozdzielczości 3264x2448. Zdjęcia zostały zrobione pod różnymi kontami względem modelu, w przeciwieństwie do poprzedniego zdjęcia mają również różne nachylenie względem modelu. Są zdjęcia robione z góry, gdzie samochody idealnie się wpasowują w miejsca parkingowe, jak i zdjęcia robione pod niskim kontem, gdzie samochody zasłaniają oboczne miejsca parkingowe.

Drugie drugi model poprawiał znacząco jakość danych. Jednolita faktura brystolu eliminowała problem wykrywanych krawędzi. Dzięki sztywności brystolu nie potrzebna dodatkowych elementów usztywniających model, również rysowanie pasów korektorem w taśmie nie powodowało problemów. Dzięki zastosowaniu kamery o lepszych parametrach i lepszego źródła oświetlenia pomieszczenia, scena nie wymagała dodatkowego doświetlenia, a samochody o kolorze czarnym są wyraźnie widoczne na zdjęciach. Nie występuje również problem szumu cyfrowego.

## Oznaczanie konturów dla miejsc parkingowych

Do nauczenia algorytmu rozpoznawania zdjęć trzeba dostarczyć zdjęcia w raz z oznaczonymi miejscami parkingowymi. Przyjęta konwencja zakłada że zdjęcia z rozszerzeniami *.png, .jpg, .jpeg, .bmp* znajdują się w jednym folderze, a oznaczenia miejsc parkingowych do danego zdjęcia znajdują się w pliku o nazwie identycznej jak zdjęcie i rozszerzeniem *.json*. Pliki z oznaczeniami zawiera listę obiektów klasy *ParkingSlot* przedstawioną w Listing 1 zakodowane w formacie *JSON*.

public class ParkingSlot

{

public Contour Contour { get; set; }

public bool IsOccupied { get; set; }

}

public class Contour : List<Contour.Point>

{

public struct Point

{

public double X { get; set; }

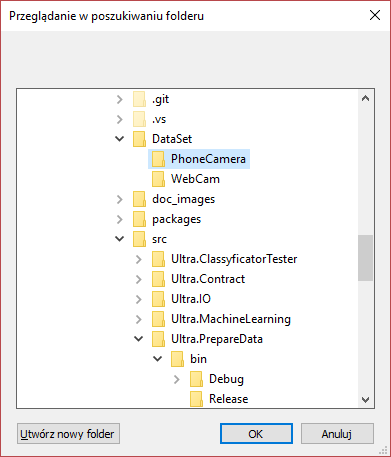
public double Y { get; set; }

}

}

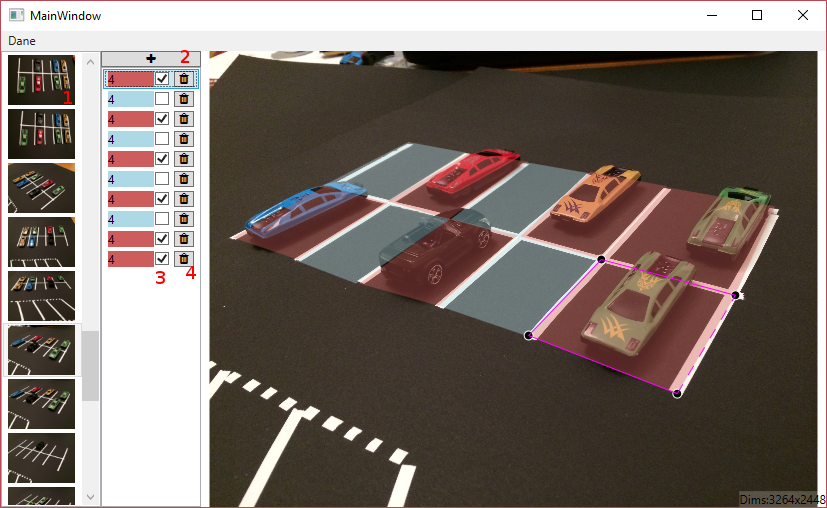
Listing 1 Klasa Parking slot i Contour

Do przygotowania danych uczących dla algorytmu powstał oddzielny program. Po otworzeniu programu do przygotowania danych pojawi się okienko (Rysunek 1) z wyborem folderu gdzie przygotowane są zdjęcia testowe, można również użyć zbiorów zdjęć dołączonych razem z pracą. Jeśli w zbiorze danych były już zapisane kontury to program wczyta je automatycznie.



Rysunek Wybieranie zbioru danych

Po wybraniu zbioru testowego ukaże nam się okno do oznaczania miejsc parkingowych. (Rysunek 2). Po lewej stronie okna jest lista do wybierania zdjęcia (oznaczone czerwoną jedynką). Możemy zmieniać aktywne zdjęcie myszką i skrótem klawiaturowym, TAB aby zmienić zdjęcie na następne i CTRL + TAB aby zmienić zdjęcie na poprzednie. Po prawej stronie od listy zdjęć znajduje się lista konturów dla obecnie zaznaczonego zdjęcia. Nowy kontur można dodać przy użyciu przycisku oznaczonym czerwonym numerem 2 lub skrótu klawiaturowego A. Punkty konturu dodaje poprzez klikanie lewym przyciskiem na obraz, punkty możemy przeciągać metodą przeciągnij i upuść (ang. drag and drop). Linią przerywaną zaznaczona jest krawędź pomiędzy pierwszym a ostatnim punktem konturu, po dodaniu nowego punktu ta krawędź zostaną zastąpiona nowymi dwiema. Aby usunąć punkt wystarczy kliknąć go prawym przyciskiem muszy. Aby usunąć kontur wystarczy kliknąć ikonkę kosza przy odpowiednim konturze. Skrót klawiaturowy D usuwa obecnie zaznaczony kontur. Do oznaczania miejsca parkingowego jako wolne lub zajęte służy pole do zaznaczenia (ang. checkbox) oznaczony czerwonym numerem 3. Alternatywnie można użyć skrótów klawiaturowych Q do oznaczenia obecnie zaznaczonego miejsca jako wolne i W do miejsca zajętego. Aby zapisać wyniki oznaczania miejsc parkingowych należy wybrać z paska menu Dane następnie Zapisz, lub opcjonalnie można użyć popularnego skrótu klawiaturowego CTRL + S.



Rysunek Okno do oznaczania konturów

## Badane cechy obrazu

Łatwo można zauważyć na zdjęciach testowych że miejsca parkingowe z samochodami zazwyczaj odróżniają się kolorem od zdjęć pustych. Bazując na tej obserwacji pierwszą badaną cechą jest badanie współczynnika pikseli z nasyceniem >100 do całości obszaru miejsca parkingowego. Listing 2 przedstawia sposób obliczania liczności pikseli z nasyceniem większym niż 100. Jako argumenty funkcji przyjmowane są kontur miejsca parkingowego w postaci listy punktów definiującej wierzchołki wieloboku, i obiekt Mat reprezentujący zawierające miejsce parkingowe. Na początku wyliczamy region zainteresowania (ROI) obramowujący kontur, będzie później używany do wycięcia kawałka obszaru w którym znajduje się miejsce parkingowe. Następnie kontur jest przekształcany w maskę w postaci macierzy, po czym maska jest przycinana do obszaru zainteresowania i zamiany schematu kolorów w skali szarości. Następnie z obrazu wejściowego wycinamy obszar zainteresowania, później wycinamy część obrazu w obszarze zainteresowania, następnie zamieniamy schemat kolorów na HSV, następnie stosujemy maskę na warstwie nasycenia, następnie stosujemy progowanie a na koniec zliczamy liczność pikseli które na warstwie nasycenia mają wartość nie zerową. Listing 4 przedstawia obliczanie liczności pikseli maski, która jest obliczana w analogiczny sposób do liczności pikseli nasyconych. Obliczanie proporcji pikseli nasyconych do całości obszaru przedstawia Listing 5. W modelu kolorów HSV zaimplementowanym w bibliotece OpenCV zdarzają się piksele o kolorze zbliżonym do czarnego które jednak posiadają wysokie nasycenie, dzieje się tak dlatego że przestrzeń kolorów HSV w bibliotece jest prezentowana jako walec. W programie skaluje nasycenie z jasnością co jest przedstawia Listing 3.

public static int CountSaturationPixels(Contour contour, Mat src)

{

var rect = GetContourRect(contour, src.Height, src.Width);

var mask = GetMask(contour, src.Size(), color: Scalar.White, background: Scalar.Black)

.Clone(rect)

.CvtColor(ColorConversionCodes.BGR2GRAY);

return src

.Clone(rect)

.CvtColor(ColorConversionCodes.BGR2HSV)

.ScaleSaturationWithValue() // returns only saturation layer

.BitwiseAnd(mask)

.Threshold(100, 255, ThresholdTypes.Binary)

.CountNonZero();

}

Listing 2 Obliczanie liczności nasyconych pikseli

public static Mat ScaleSaturationWithValue(this Mat src)

{

var mats = src.Split();

return mats[1].Mul(mats[2], 1.0/255);

}

Listing Skalowanie nasycenia z jasnością koloru

public static int CountMaskArea(Contour contour, Mat src)

{

var rect = GetContourRect(contour, src.Height, src.Width);

return GetMask(contour, src.Size(), color: Scalar.White, background: Scalar.Black)

.Clone(rect)

.CvtColor(ColorConversionCodes.BGR2GRAY)

.CountNonZero();

}

Listing 4 Obliczanie liczności pikseli w masce

public float SaturatedPixelsRatio => (float) SaturatedPixels/MaskPixels;

Listing Obliczanie współczynnika nasyconych pikseli względem obszaru



Rysunek Wyniki testów klasyfikacji z cechą współczynnika pikseli nasyconych

Rysunek 3 przedstawia wyniki walidacji dla klasyfikacji z użyciem wyżej opisanej cechy.

Cecha nie jest skuteczna przy wykrywaniu samochodów o kolorze karoserii który posiada niskie nasycenie koloru (np. białym, czarnym, szarym) dając błędnie klasyfikując miejsca zajęte jako puste (błąd II typu). Szum cyfrowy generowany przez kamerę przy słabym oświetleniu może powodować że puste miejsce może zostać zakwalifikowane jako miejsce zajęte (błąd I typu).

Tabela 1 przedstawia porównanie zdjęć na których zastawano wykrywanie krawędzi metodą *Cannego*, można zauważyć że miejsca parkingowe na których znajdują się samochody dają widocznie więcej krawędzi od pustego miejsca parkingowego.

Tabela Porównanie zdjęć po zastosowaniu algorytmu wykrywania krawędzi

|  |  |  |  |
| --- | --- | --- | --- |
|  | Puste miejsce | Kolorowy samochód | Czarny samochód |
| Zdjęcie oryginalne | C:\Users\Sylwekqaz\Documents\Visual Studio 2015\Projects\Inz\doc_images\edge\empty.png | C:\Users\Sylwekqaz\Documents\Visual Studio 2015\Projects\Inz\doc_images\edge\blue.png | C:\Users\Sylwekqaz\Documents\Visual Studio 2015\Projects\Inz\doc_images\edge\black.png |
| Po wykryciu krawędzi | C:\Users\Sylwekqaz\Documents\Visual Studio 2015\Projects\Inz\doc_images\edge\empty_edge.png | C:\Users\Sylwekqaz\Documents\Visual Studio 2015\Projects\Inz\doc_images\edge\blue_edge.png | C:\Users\Sylwekqaz\Documents\Visual Studio 2015\Projects\Inz\doc_images\edge\black_edge.png |

Na podstawie obserwacji powstała koncepcja obliczania kolejnej cechy. Wyliczany jest stosunek pikseli z krawędziami do całości obszaru. Listing 6 prezentuje algorytm obliczający tą cechę. Jako wejście przyjmowany jest kontur, czyli lista punktów reprezentujących wierzchołki wieloboku w którym znajduje się miejsce parkingowe i obiekt Mat który reprezentuje zjecie parkingu. Na początku wyliczany jest z konturu obszar zainteresowania (ROI). Następnie wyliczana jest maska z konturu i przycinana do obszaru zainteresowania. Następnie przycinamy obraz wejściowy do obszaru zainteresowania, następnie wykrywane są krawędzie bezparametrową metodą *Cannego*, następnie na zdjęciu stosowana jest maska i zliczane są piksele o wartości większej niż zero. Następnie obliczana jest liczność pikseli w masce które zostało opisane przy obliczaniu poprzedniej cechy Listing 4. Obliczanie samego stosunku pikseli z krawędziami przedstawia Listing 7. Do wykrywania krawędzi używana jest bezparametrowa implementacja metody *Cannego*, która automatycznie dobiera progi, na podstawie średniej wartości koloru, dla obrazu w skali szarości (Listing 8).

public static int CountEdgePixels(Contour contour, Mat src)

{

var rect = GetContourRect(contour, src.Height,src.Width);

var mask = GetMask(contour, src.Size(), color: Scalar.White, background: Scalar.Black)

.Clone(rect)

.CvtColor(ColorConversionCodes.BGR2GRAY);

return src

.Clone(rect)

.DetectEdges()

.BitwiseAnd(mask)

.CountNonZero();

}

Listing 6 Obliczanie liczności pikseli z krawędziami

public float EdgePixelsRatio => (float) EdgePixels/MaskPixels;

Listing Obliczanie stosunku pikseli z krawędziami do całego obszaru

public static Mat DetectEdges(this Mat src, double sigma = 0.33)

{

var graySrc = src.CvtColor(ColorConversionCodes.BGR2GRAY);

Cv2.MeanStdDev(graySrc, out var meanScalar, out var stddevScalar);

var mean = meanScalar[0];

var lower = (int) Math.Max(0, (1.0 - sigma) \* mean);

var upper = (int) Math.Min(255, (1.0 + sigma) \* mean);

return graySrc.Canny(lower, upper);

}

Listing Bezparametrowa detekcja krawędzi Cannego

Rysunek 4 Pokazuje wyniki sprawdzianu klasyfikacji.



Rysunek Wyniki walidacji dla przy użyciu cech stosunku pikseli nasyconych i stosunku pikseli z krawędziami

Koncepcja na kolejne cztery cechy powstała podczas porównania histogramów jasności i saturacji zdjęć miejsc parkingowych. Tabela 2 przedstawia histogramy dla pustego miejsca parkingowego i zajętego przez samochód o kolorze żółtym i samochód o kolorze czarnym. Na histogramach zaznaczono czerwonym kolorem wartość średniej jasności. Łatwo zauważyć że histogram niezajętego miejsca parkingowego ma bardzo małe odchylenie standardowe. Tabela 3 Przedstawia porównanie histogramów nasycenia dla zdjęć miejsc parkingowych zajętych przez czarny samochód, kolorowy samochód i pustego miejsca parkingowego. Kolorem czerwonym na histogramie zaznaczono średnią wartość nasycenia. Można zauważyć że wartość odchylenia standardowego i średniego nasycenia dla miejsca parkingowego zauważalnie różni się od miejsc zajętych przez samochody.

Tabela Porównanie histogramów jasności miejsc parkingowych

|  |  |  |  |
| --- | --- | --- | --- |
|  | Czarny samochód | Żółty samochód | Puste miejsce |
| Jasność z HSV | C:\Users\Sylwekqaz\Documents\Visual Studio 2015\Projects\Inz\doc_images\val_hist\black_car_value.png | C:\Users\Sylwekqaz\Documents\Visual Studio 2015\Projects\Inz\doc_images\val_hist\color_car_value.png | C:\Users\Sylwekqaz\Documents\Visual Studio 2015\Projects\Inz\doc_images\val_hist\no_car_value.png |
| Histogram jasności | C:\Users\Sylwekqaz\Documents\Visual Studio 2015\Projects\Inz\doc_images\val_hist\black_car_value_histogram.png | C:\Users\Sylwekqaz\Documents\Visual Studio 2015\Projects\Inz\doc_images\val_hist\color_car_value_histogram.png | C:\Users\Sylwekqaz\Documents\Visual Studio 2015\Projects\Inz\doc_images\val_hist\no_car_value_histogram.png |

Tabela Porównanie histogramów nasycenia dla miejsc parkingowych

|  |  |  |  |
| --- | --- | --- | --- |
|  | Czarny samochód | Żółty samochód | Puste miejsce |
| Nasycenie | C:\Users\Sylwekqaz\Documents\Visual Studio 2015\Projects\Inz\doc_images\sat_hist\black_car_saturation.png | C:\Users\Sylwekqaz\Documents\Visual Studio 2015\Projects\Inz\doc_images\sat_hist\black_car_saturation.png | C:\Users\Sylwekqaz\Documents\Visual Studio 2015\Projects\Inz\doc_images\sat_hist\no_car_saturation.png |
| Histogram nasycenia | C:\Users\Sylwekqaz\Documents\Visual Studio 2015\Projects\Inz\doc_images\sat_hist\black_car_saturation_histogram.png | C:\Users\Sylwekqaz\Documents\Visual Studio 2015\Projects\Inz\doc_images\sat_hist\color_car_saturation_histogram.png | C:\Users\Sylwekqaz\Documents\Visual Studio 2015\Projects\Inz\doc_images\sat_hist\no_car_saturation_histogram.png |

Obliczanie tych cech przedstawia Listing 9. Funkcja na wejście przyjmuje kontur reprezentowany poprzez listę punktów będących wierzchołkami wieloboku, i obiekt Mat reprezentujący zdjęcie parkingu. Na początku z konturu wyznaczany jest obszar zainteresowania (ROI), następnie kontur jest przekształcany na maskę i przycinany do obszaru zainteresowania. Następnie obraz wejściowy jest przycinany do obszaru zainteresowań, później zamieniany jest schemat kolorów z RGB na HSV, i przy użyciu funkcji *Split* obraz zostaje rozłożony na warstwy. Następnie kolejno warstwy nasycenia i jasności są przekazana do lokalnej funkcji *LocalMeanStdDev* gdzie wyliczana jest średnia i odchylenie standardowe z pojedynczej warstwy. Przed zwróceniem wartości są skalowane z przedziału [0-255] do przedziału [0-1].

public static ((float mean, float stddev) saturation, (float mean, float stddev) value) GetHSVColorStats(

Contour contour, Mat src)

{

var rect = GetContourRect(contour, src.Height, src.Width);

var mask = GetMask(contour, src.Size(), color: Scalar.White, background: Scalar.Black)

.Clone(rect)

.CvtColor(ColorConversionCodes.BGR2GRAY);

var layers = src.Clone(rect)

.CvtColor(ColorConversionCodes.BGR2HSV)

.Split();

(float mean, float stddev) LocalMeanStdDev(Mat area)

{

Cv2.MeanStdDev(area, out var scalarMean, out var scalarStddev, mask);

var mean = (float) (scalarMean[0] / 255);

var stddev = (float) (scalarStddev[0] / 255);

return (mean, stddev);

}

return (LocalMeanStdDev(layers[1])/\*saturation layer\*/, LocalMeanStdDev(layers[2])/\* value layer\*/);

}

Listing Obliczanie statystyk średniej i odchylenia standardowego dla saturacji i wartości



Rysunek Wyniki sprawdzianu klasyfikacji dla cech: współczynnik krawędzi, współczynnik saturacji, średnia jasność, odchylenie jasności, średnie nasycenie, odchylenie standardowe nasycenia

Rysunek 5**Błąd! Nie można odnaleźć źródła odwołania.** Przedstawia wyniki sprawdzianu klasyfikacji z użyciem poprzednio omawianych cech. Dzięki ich wprowadzeniu jest kilkuprocentowy zysk w rozpoznawaniu miejsc parkingowych.

Tabela 4 przedstawia wykresy punktowe przedstawiające rzuty 6cio wymiarowej przestrzeni parametrów. Na diagonali wypisane są nazwy współczynników. Wykresy nad główną przekątną stanowią wykresy punktowe gdzie wartości X jest reprezentowana przez cechę wypisaną na przekątnej w tym wierszu, natomiast wartości Y są reprezentują cechę wypisaną na diagonali w tej samej kolumnie. Punkty o kolorze pomarańczowym prezentują wolne miejsca parkingowe, zajęte miejsca parkingowe zaznaczone są kolorem niebieskim.

Tabela Reprezentacja przestrzeni cech dla zbioru testowego

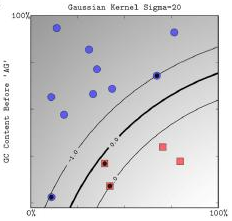
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Współczynnik krawędzi |  |  |  |  |  |
|  | Współczynnik saturacji |  |  |  |  |
|  |  | Średnia saturacja |  |  |  |
|  |  |  | Odchylenie standardowe saturacji |  |  |
|  |  |  |  | Średnia wartość |  |
|  |  |  |  |  | Odchylenie standardowe wartości |

## Klasyfikator

Jako klasyfikatora w algorytmie używam maszyny wektorów nośnych (ang. support vector machine, SVM). SVM próbuje dopasować hiperpłaszczyznę do danych testowych, tak aby rozdzielić obie klasy od siebie. Hiperpłaszczyzna jest dobierana tak aby margines (czyli odległość dowolnego punktu od hiperpłaszczyzny) był jak największy (Rysunek 6). SVM potrafi rozdzielać również dane które nie da się rozdzielić w sposób liniowy, poprzez zastosowanie funkcji jądra, która dodaje dodatkowe wymiary w przestrzeni cech. Rysunek 7 przedstawia przykładowe rozdzielenie dwóch klas przy zastosowaniu gaussowskiej funkcji jądra.

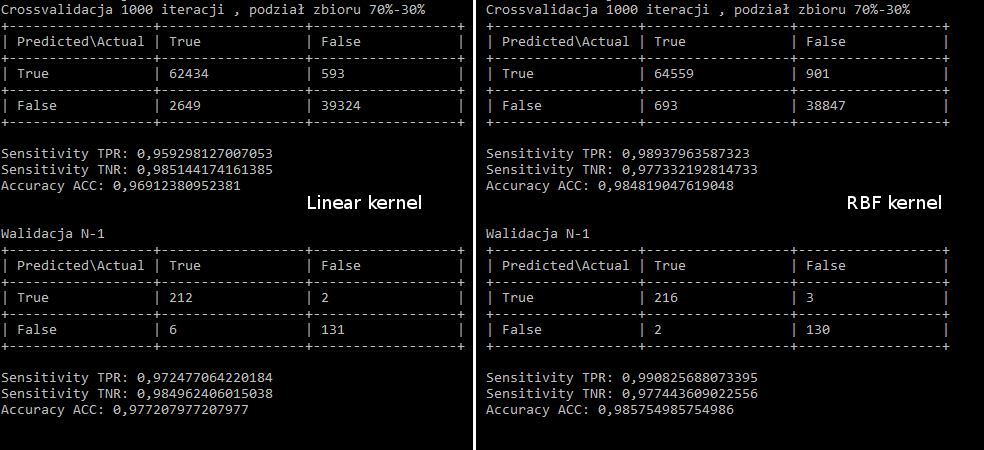


Rysunek SVM margines



Rysunek SMV z jądrem RBF

W algorytmie używana jest gaussowska funkcja jądra Dzięki podejściu nieliniowemu dostajemy kilkuprocentowy zysk w poprawności klasyfikacji (Rysunek 8).



Rysunek Różnica poprawności klasyfikacji pomiędzy jądrem RBF i liniowymmm jądrem

Listing 10 przedstawia klasę opakowującą klasyfikator SVM. W funkcji *Create* tworzony jest nowy klasyfikator i ustawiane są wartości parametrów *Gamma* i *C* które są używane w optymalizacji wyznaczania hiperpłaszczyzn. Następnie klasyfikator jest uczony zbiorem test uczącym przekazanym jako parametr wejściowy metody. Funkcja *Predict* służy do przewidywania klasy dla nowych obserwacji. Jako parametr wejściowy przyjmuje klasę przechowywującą cechy miejsca parkingowego, następnie zamienia je na wewnętrzny obiekt używany przez OpenCV i wykonywana jest klasyfikacja. Wynik klasyfikacji jest konwertowana na wartość prawda/fałsz.

public class SVMClassifier : IClassifier

{

private readonly SVM \_svm;

private SVMClassifier(SVM svm)

{

\_svm = svm;

}

public bool Predict(ImageFeatures imageFeatures)

{

return Convert.ToBoolean(\_svm.Predict(imageFeatures.ToPredictionMat()));

}

public static SVMClassifier Create(List<ImageFeatures> trainingData)

{

var svm = SVM.Create();

svm.Type = SVM.Types.CSvc;

svm.KernelType = SVM.KernelTypes.Rbf;

svm.TermCriteria = TermCriteria.Both(maxCount: 1000, epsilon: 0.000001);

svm.Gamma = 100.0;

svm.C = 1.0;

svm.Train(trainingData.ToTrainingMat(), SampleTypes.RowSample, trainingData.ToResponseMat());

return new SVMClassifier(svm);

}

}

Listing 10 Konfiguracja klasyfikatora SVM

## Walidacja jakości klasyfikacji

W programie testującym poprawność dokładność klasyfikatora wykonuje dwa sprawdzenia.

Pierwszym sprawdzianem jest 1000-krotny sprawdzian krzyżowy. Polega on na 1000-krotnym podziale prób zbiory na dwa podzbiory – uczący i testowy w proporcjach ustalonych arbitralnie 70% zbiór uczący, 30% zbiór testowy. Zbiorem uczącym jest uczony klasyfikator, a wyniki są sprawdzane na zbiorze testowym i zapisywane do macierzy błędu. Powstałe macierze błędów są sumowane do jednej macierzy. Listing 11 przedstawia k-krotną walidację, jako parametry wejściowe przyjmuje zbiór prób, ilość iteracji i proporcję w jakich ma podzielić zbiór na testowy i uczący. Do obliczania pojedynczej iteracji wykorzystywana jest funkcja *Validate* która jest przedstawia Listing 12.

public static ConfusionMatrix CrossValidation(List<ImageFeatures> observations, int iterations, double splitRatio)

{

var summaryConfusionMatrix = new ConfusionMatrix();

for (int i = 0; i < iterations; i++)

{

var tuple = observations.Shuffle().Split(splitRatio);

var iterationConfusionMatrix = Validate(tuple.Item1, tuple.Item2);

summaryConfusionMatrix += iterationConfusionMatrix;

}

return summaryConfusionMatrix;

}

Listing 11 K-krotna walidacja krzyżowa

public static ConfusionMatrix Validate(List<ImageFeatures> train, List<ImageFeatures> validation)

{

var svmClassifier = SVMClassifier.Create(train);

var confusionMatrix = new ConfusionMatrix();

foreach (var validationObservation in validation)

{

var predict = svmClassifier.Predict(validationObservation);

confusionMatrix.AddVote(actual: validationObservation.IsOccupied, predicted: predict);

}

return confusionMatrix;

}

Listing Walidacja klasyfikatora przy użyciu dowolnego zbioru testowego i uczącego

Drugim sprawdzianem jest sprawdzian *Leave-one-out* który jest odmianą k-krotnej walidacji krzyżowej. Różnicą jest to że nie dzieli on prób według proporcji, ale oddziela jedną próbę od pozostałych, pojedyncza próba służy jako zbiór testowy, a pozostałe próby jako zbiór uczący. Krotność walidacji jest równa ilości prób w początkowym zbiorze. Listing 13 przedstawia sposób obliczania tej walidacji, do obliczania pojedynczej iteracji używana jest wcześniej przedstawiona funkcja *Validate* którą prezentuje Listing 12.

public static ConfusionMatrix LeaveOneOutValidation(List<ImageFeatures> observations)

{

var confumaMatrix = new ConfusionMatrix();

for (var i = 0; i < observations.Count; i++)

{

var validation = new List<ImageFeatures> {observations[i]};

var train = observations.WithoutElementAt(i);

confumaMatrix += Validate(train, validation);

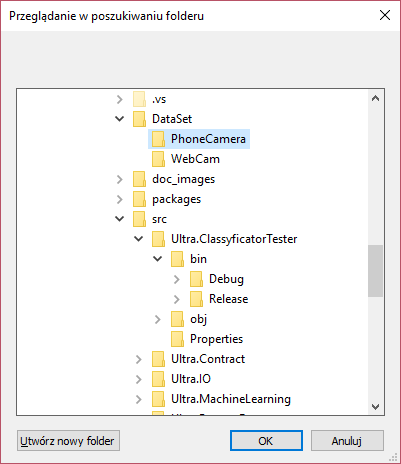
}

return confumaMatrix;

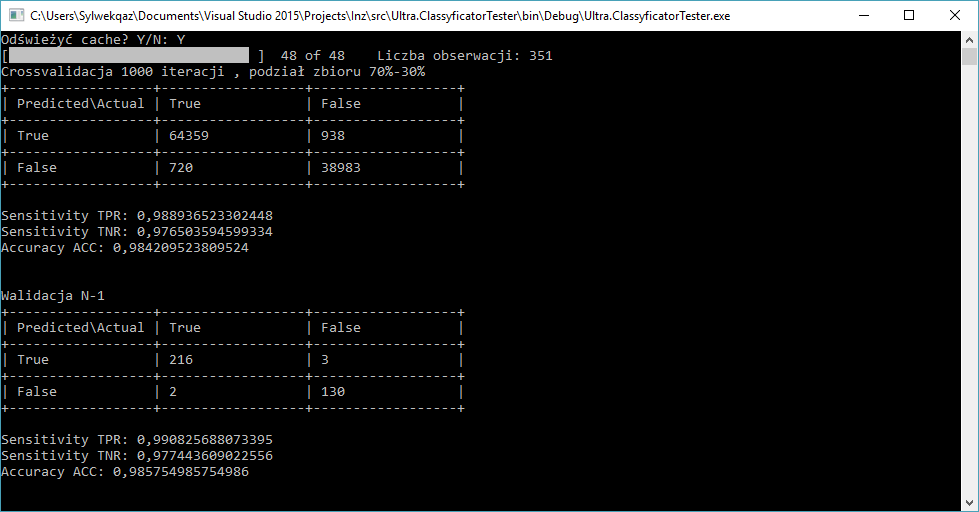
}

Listing 13 Walidacja Leave One Out

Po włączeniu programu do testowania klasyfikatora zostaniemy zapytani o wybranie folderu z przygotowanymi danymi testowymi (Rysunek 9) . Program zapisuje obliczone cechy dla prób w folderze z danymi testowymi, tworząc mechanizm pamięci podręcznej (ang. cache) dla optymalizacji obliczeń wykonywanych przy kolejnych wywołaniach na tym samym zbiorze. Pamięć podręczna dla zbioru jest zapisywana w pliku *cachedFeatures.csv*. Po wybraniu folderu z danymi zostaniemy zapytani czy chcemy aby program odświeżył pamięć podręczną zapisaną w pliku co widać w pierwszej linijce na Rysunek 10. Jeżeli wybierzemy nie, program spróbuje odnaleźć plik z pamięcią podręczną, odczytać danę i przekaże je dalej do testerów klasyfikatorów. Jeżeli wybierzemy aby program odświeżył cache, lub plik z pamięcią podręczną nie zostanie znaleziony lub jest uszkodzony, wtedy program wczyta kolejno zdjęcia wraz z oznaczeniami miejsc parkingowych i obliczy ich cechy. Następnie zapisze dane do pamięci podręcznej i zwróci próby do testerów klasyfikatorów. Z racji iż przetwarzanie może być operacją czasochłonną program wyświetla pasek postępu widoczny w drugim wierszu programu na Rysunek 10. Następnie program wypisze ilość załadowanych obserwacji i obliczy skuteczność klasyfikatorów sposób opisany wcześniej w tym rozdziale.



Rysunek Wybór folderu z danymi do programu walidującego klasyfikator



Rysunek Okno programu do walidacji klasyfikatora

# Omówienie i dyskusja wyników

# Literatura

# Spis obrazków

[Rysunek 1 Wybieranie zbioru danych 11](#_Toc472073835)

[Rysunek 2 Okno do oznaczania konturów 12](#_Toc472073836)

[Rysunek 3 Wyniki testów klasyfikacji z cechą współczynnika pikseli nasyconych 14](#_Toc472073837)

[Rysunek 4 Wyniki walidacji dla przy użyciu cech stosunku pikseli nasyconych i stosunku pikseli z krawędziami 16](#_Toc472073838)

[Rysunek 5 Wyniki sprawdzianu klasyfikacji dla cech: współczynnik krawędzi, współczynnik saturacji, średnia jasność, odchylenie jasności, średnie nasycenie, odchylenie standardowe nasycenia 19](#_Toc472073839)

[Rysunek 6 SVM margines 21](#_Toc472073840)

[Rysunek 7 SMV z jądrem RBF 21](#_Toc472073841)

[Rysunek 8 Różnica poprawności klasyfikacji pomiędzy jądrem RBF i liniowymmm jądrem 22](#_Toc472073842)

[Rysunek 9 Wybór folderu z danymi do programu walidującego klasyfikator 25](#_Toc472073843)

[Rysunek 10 Okno programu do walidacji klasyfikatora 26](#_Toc472073844)

# Spis listingów

[Listing 1 Klasa Parking slot i Contour 11](#_Toc472073845)

[Listing 2 Obliczanie liczności nasyconych pikseli 13](#_Toc472073846)

[Listing 3 Skalowanie nasycenia z jasnością koloru 13](#_Toc472073847)

[Listing 4 Obliczanie liczności pikseli w masce 14](#_Toc472073848)

[Listing 5 Obliczanie współczynnika nasyconych pikseli względem obszaru 14](#_Toc472073849)

[Listing 6 Obliczanie liczności pikseli z krawędziami 16](#_Toc472073850)

[Listing 7 Obliczanie stosunku pikseli z krawędziami do całego obszaru 16](#_Toc472073851)

[Listing 8 Bezparametrowa detekcja krawędzi Cannego 16](#_Toc472073852)

[Listing 9 Obliczanie statystyk średniej i odchylenia standardowego dla saturacji i wartości 19](#_Toc472073853)

[Listing 10 Konfiguracja klasyfikatora SVM 23](#_Toc472073854)

[Listing 11 K-krotna walidacja krzyżowa 24](#_Toc472073855)

[Listing 12 Walidacja klasyfikatora przy użyciu dowolnego zbioru testowego i uczącego 24](#_Toc472073856)

[Listing 13 Walidacja Leave One Out 24](#_Toc472073857)

Wyrażam zgodę na udostępnienie mojej pracy w czytelniach Biblioteki SGGW   
w tym w Archiwum Prac Dyplomowych SGGW

.................................................................

*(czytelny podpis autora pracy)*

1. Wrapper – biblioteka programistyczna, która opakowuje wywołania natywnych metod biblioteki w sposób ujednolicony z językiem dla którego została przygotowana. Zazwyczaj posiada konwersję z popularnych typów w języku na typy które obsługuje biblioteka [↑](#footnote-ref-1)
2. Wołanie łańcuchowe (ang. method chaining) – polega na łączeniu wywołań metod na obiekcie. Każda metoda zwraca obiekt który pozwala na wywołanie kolejnych metod w pojedynczej instrukcji (np. person.SetName("Jan Kowalski").SetAge(35);). Najczęściej osiąga się to poprzez zwracanie przez metody obiektu w kontekście którego zostały wykonane (np. return this;). Drugim stosowanym podejściem jest zwracanie kopi obiektu w kontekście którego została wywołana metoda, z zastosowanymi zmianami. Drugie podejście stosowanie jest wtedy, kiedy zależy nam na zachowaniu obiektu oryginalnego lub obiekt jest niezmienny. [↑](#footnote-ref-2)