Project Report

June 5, 2022

1 Projekt Algorytmy Uczenia Maszynowego

1.1 Cel Projektu

Celem tego projektu jest wykorzystanie algorytmów uczenia maszynowego do zrealizowania zadania klasyfikacji.

1.2 Przedstawienie problemu

W ramach projektu zostało wykonane zadanie klasyfikacji zdjęć zwierząt. Przygotowana implementacja powinna rozpoznawać zwierzęta z pięciu różnych klas, w tym przypadku będą to:

- kurczak
- owca
- koń
- pająk
- słoń

Do realizacji zadania wykorzystano zbiór danych dostępny pod linkiem: https://www.kaggle.com/datasets/alessiocorrado99/animals10

1.3 Wybrane algorytmy uczenia maszynowego

Jednym z założeń projektowych jest zrealizowanie danego zadnia z wykorzystaniem trzech podstawowych algorytmów uczenia maszynowego, w tym przypadku zdecydowano się na wybranie następujących algorytmów:

- Maszyna wektorów nośnych
- Wielowarstwowy perceptron
- K najbliższych sąsiadów

Po wykorzystaniu algorytmów uczenia maszynowego należało zaimplementować mechanizm łączenia wykorzystywanych algorytmów. W tym projekcie zdecydowano się na wykorzystanie `głosowania większościowego`.

1.4 Wczytanie oraz wizualizacja zbioru danych

Jednym z kluczowych elementów w zadania klasyfikacji jest dobranie odpowiedniego zbioru danych. W tym przypadku zaszła potrzeba usunięcia części elementów ze zbioru ze względu na ich jakość. Zdjęcia w których występowało wiele zwierząt, były bardzo zaszumione albo przedstawiały nie te zwierzęta musiały zostać usunięte. W efekcie końcowym zbiory danych zostały zmniejszone do 250 zdjęć dla każdej klasy. Mimo zmniejszenia zbioru uczącego uzyskano lepsze rezultaty.

1.4.1 Wykorzystanie techniki data augmentation(rozszerzanie danych)

W celu rozszerzenia zbioru danych wykorzystano technikę polegającą na dodaniu wcześniej wykorzystywanych obrazów poddanych różnym transformacją, między innymi:

- dodanie szumu
- rotacja obrazu
- rozmycie obrazu
- zwiększenie kontrastu
- zwiększenie korekcji gamma

Takie zabiegi wykorzystuje się w celu zwiększania różnorodności oraz w zbiorach treningowych co przekłada się na lepsze wyniki wytrenowania modelu.

[ ]:
```
from skimage.io import imread_collection
from skimage.transform import resize, rotate
from sklearn.model_selection import train_test_split
from skimage import exposure
from scipy import ndimage
from skimage import util
import matplotlib.pyplot as plt
import numpy as np
from joblib import dump

def load_images(class_name, values, labels):
    path_to_image = 'raw-img/'
    postfix = '/*.jp*'
    image_size = (64,64,3)
    i = 0
    for image in imread_collection(path_to_image + class_name + postfix):

        img = resize(image, image_size)
        values.append(img)
        labels.append(class_name)
        values.append(exposure.adjust_sigmoid(img))
        labels.append(class_name)
        values.append(exposure.adjust_gamma(img))
        labels.append(class_name)
        values.append(ndimage.uniform_filter(img,(2,2,1)))
        labels.append(class_name)
        values.append(rotate(img, -30))
        labels.append(class_name)
        values.append(rotate(img, 30))
        labels.append(class_name)
        values.append(rotate(img, -60))
        labels.append(class_name)
        values.append(util.random_noise(img))
        labels.append(class_name)
```

```
def create_data_set():
    class_names = ['spider', 'horse', 'elephant', 'chicken', 'sheep']
    values = []
    labels = []
    for class_name in class_names:
        load_images(class_name, values, labels)

    return np.array(values), np.array(labels)

def plot_data_set(x, y, unique_labels):
    fig, axes = plt.subplots(1, len(unique_labels))
    fig.set_size_inches(15,4)
    fig.tight_layout()

    for ax, label in zip(axes, unique_labels):
        ax.imshow(x[np.where(y == label)[0][0]])
        ax.axis('off')
        ax.set_title(label)

x, y = create_data_set()
unique_labels = np.unique(y)

plot_data_set(x, y, unique_labels)
```

1.5 Ekstrakcja cech

Niezbędnym elementem w wykorzystaniu algorytmów uczenia maszynowego do zadania klasyfikacji jest wyciągniecie najważniejszych informacji ze zbioru uczącego. W przypadku danych jakimi są obrazy można zrobić to na wiele sposobów, miedzy innymi:

- wykorzystać histogram
- wykorzystać operatory pozwalające uwypuklić krawędzie
- wykorzystać bardziej skomplikowane algorytmy takie jakie np. transformacja HOGa czy cechy Haar

Do realizacji tego zadania zdecydowano się na wybranie transformaty `HOG` oraz histogram barw w opisie `hsv`.

1.5.1 Transformacja HOG(Histogram zorientowanych gradientów)

Opis cech wykorzystujący histogram zorientowanych gradientów, wymaga wykonania wielu pośrednich operacji aby uzyskiwać oczekiwane rezultaty.

Wyznaczenie gradientów Pierwszym krokiem jest wyznaczanie gradientów z w pionie oraz w poziomie wykorzystując filtrując z wykorzystaniem jąder:

$$[-1, 0, 1] \text{ oraz } [-1, 0, 1]^T$$

Kierunek oraz natężenie gradientów Kolejnym krokiem jest obliczenie natężenie gradientów oraz ich kierunków zgodnie z wzorami

$$g = \sqrt{g_x^2 + g_y^2}$$

$$\theta = arctan\frac{g_y}{g_x}$$

Obliczanie histogramów gradientów Kolejnym elementem było obliczanie gradientów, aby to zrobić obraz należy podzielić na komórki a następnie dla każdej z komórek dodać wartości gradientów aby uzyskać histogram dla danej ilości kierunków.

Normalizacja gradientów Ostatnim elementem jest normalizacja gradientów. Normalizacja gradientów polega na dobraniu bloków złożonych z komórek a następnie wykonania dla niej normalizacji w taki sposób jak dla wektorów.

1.5.2 Wykorzystanie transformacji HOG

W ramach projektu wykorzystano gotową implementację z biblioteki scikt-image, która dostarcza deskryptory HOG. Przy czym najważniejsze parametry jakie zostały wykorzystane to:

- orientation — ilość kierunków gradientów branych pod uwagę
- pixels_per_cell — ilość pikseli w jednej komórce
- cells_per_block — ilość komórek w jednym bloku

Wyżej wymienione parametry zostały dobrane z wykorzystaniem przeszukania (`GridSearchCV`), tak aby dostarczyć klasyfikatorom najlepszy zestaw cech.

Aby przedstawić efekt działania transformaty HOG można wykorzystać parameter `visualize`, dzięki któremu można zwrócić wizualny efekt działania operacji co zostało przedstawione poniżej.

1.5.3 Wykorzystanie histogramu

Do wykonania zadania poza opisem związanym z kształtami wykorzystano także informację związane z kolorami/barwami zawartymi w obrazach. W tym przypadku zdecydowano się na wykorzystanie modelu `HSV`, który wykorzystuje przestrzenny model barw, gdzie kolejny elementy oznaczają:

- H - odcień

- S - nasycenie koloru
- V - moc światła białego

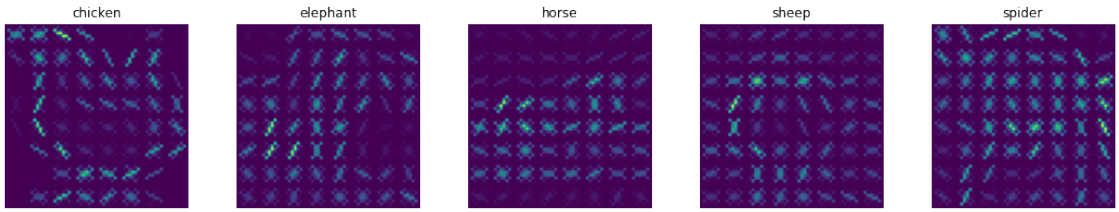
Z każdego obrazu pobrano histogram odpowiednich wartości `H`, `S` oraz `V`. Wykorzystanie cech związanych z przestrzenią barw pozwala na uzyskanie lepszych wyników klasyfikacji.

1.5.4 Podział danych na zbiór treningowy oraz testowy

Ostatnim elementem poniższego bloku jest podzielenie zbioru(z wykorzystaniem funkcji `train_test_split`) na część testową oraz część treningową w proporcji jeden do cztery. Przy podziale wykorzystano parameter **`shuffle`**, który pozwala na wstępne przetasowanie danych. Wykorzystano także parametr **`random_state`**, który pozwala na podział danych w powtarzalny sposób.

[ ]:
```
from sklearn.preprocessing import StandardScaler
from skimage.feature import hog
from skimage.exposure import histogram
from skimage.color import rgb2gray, rgb2hsv



def get_histogram(image, bins, channel):
    hist, _ = histogram(rgb2hsv(image)[:, :, channel], nbins=bins)
    return hist

def get_all_colors_histograms(image, bins = 110):
    return np.reshape((get_histogram(image, bins, 0), get_histogram(image,␣
 ↪bins, 1), get_histogram(image, bins, 2)), (bins*3))


def extract_hog_features(values, orientation = 5, pixels_per_cell=(8,8),␣
 ↪cells_per_block=(8,8)):
    scalify = StandardScaler()
    return scalify.fit_transform(
        np.array(
            [ np.concatenate(
                    (hog(rgb2gray(img), orientations=orientation,␣
 ↪pixels_per_cell=pixels_per_cell, cells_per_block=cells_per_block),␣
 ↪get_all_colors_histograms(img))) for img in values ]))

def plot_hog_features_extraction(x, y, unique_labels, orientation = 4,␣
 ↪pixels_per_cell=(8,8), cells_per_block=(8,8)):
    fig, axes = plt.subplots(1, len(unique_labels))
    fig.set_size_inches(15,4)
    fig.tight_layout()

    for ax, label in zip(axes, unique_labels):
```

```
        fd, img = hog(
            image=x[np.where(y == label)[0][0]],
            orientations=orientation,
            pixels_per_cell=pixels_per_cell,
            cells_per_block=cells_per_block,
            visualize=True,
            channel_axis=-1)
        ax.imshow(img)
        ax.axis('off')
        ax.set_title(label)

hog_train = extract_hog_features(x)
plot_hog_features_extraction(x, y, unique_labels)

x_train, x_test, y_train, y_test = train_test_split(hog_train, y, test_size=0.
 ↪2, shuffle=True, random_state=15)

del x
```



1.6 Wykorzystanie maszyny wektorów nośnych(SVM) w zadaniu klasyfikacji

Algorytm `SVM` można potraktować jako rozwinięcie modelu perceptronu. Wykorzystanie algorytmu perceptronu daje możliwość minimalizowania błędu klasyfikacji, z kolei podstawowym celem SVM jest maksymalizacja marginesu. Przy czym margines jest odległością pomiędzy hiperprzestrzenią rozdzielającą(granicą decyzyjną) a najbliższymi próbkami uczącymi(`wektorami nośnymi`). Wykorzystując algorytm SVM dążymy do uzyskania odpowiednich granic decyzyjnych, ponieważ takie modele są bardziej odporne na błędy uogólnienia, natomiast możliwie jest wystąpienie przetrenowanie.

W przypadku algorytmu ważnym parametrem jest `C`, sterując wartością `C` można kontrolować karę za niewłaściwą klasyfikację. Duże wartości `C` zwiększają kary za błędną klasyfikację, natomiast małe wartości `C` zmniejszają kary. Im większy jest parametr `C` tym bardziej dopasowujemy model do wartości testowych ograniczając ilość błędnych dopasowań jednocześnie zmniejszamy margines co może doprowadzić do przetrenowania. Małe wartości `C` zwiększają margines ty samym uogólniają model jednak wyniki klasyfikacji może być gorszy. Wartość parametru `C` została dobrana za wykorzystaniem przeszukania GridSearchCV.

Kolejnym istotnym elementem algorytmu `SVM` jest dobranie odpowiedniego jądra. Zadanie klasyfikacji dla danych rozdzielnych z wykorzystaniem liniowej hiperpłaszczyzny jest trywialnym

zadaniem z punktu widzenia algorytmu SVM. W momencie w którym dane nie są rozdzielne w sposób liniowy sytuacja jest gorsza, jednak zastosowanie odpowiedniej funkcji jądrowej pozwala na utworzenie nieliniowych kombinacji pierwotnych cech, które finalnie mogą być mapowane na przestrzenie o większej liczbie wymiarów, w których to będą liniowo separowalne. Po przeanalizowaniu dostępnych przypadków wybrano funkcje RBF(radial basis function).

[ ]:
```
from sklearn.metrics import classification_report, confusion_matrix,␣
 ↪accuracy_score
from sklearn import svm

svm_svc = svm.SVC(C=2, random_state=15)
svm_svc.fit(x_train, y_train)

test_labels_predicted = svm_svc.predict(x_test)

print('SVM percentage correct for test data: ',␣
 ↪100*accuracy_score(test_labels_predicted, y_test))
print(classification_report(y_test, test_labels_predicted))
print(confusion_matrix(y_test, test_labels_predicted))

dump(svm_svc, 'models/svm.joblib')
```

```
SVM percentage correct for test data:  89.66900702106318
              precision    recall  f1-score   support

     chicken       0.90      0.93      0.91       403
    elephant       0.89      0.91      0.90       431
       horse       0.90      0.87      0.88       394
       sheep       0.87      0.86      0.86       360
      spider       0.93      0.91      0.92       406

    accuracy                           0.90      1994
   macro avg       0.90      0.90      0.90      1994
weighted avg       0.90      0.90      0.90      1994

[[373   4  10   8   8]
 [ 10 394   7  16   4]
 [  9  20 341  16   8]
 [ 12  16  14 309   9]
 [ 11   9   7   8 371]]
```

[ ]:
```
['models/svm.joblib']
```

1.7 Wyniki Algorytmu SVM

Dla zbioru testowego wydajność jest bliska 90 procent co daje względnie dużą dokładność. Najgorsze wyniki klasyfikacji zostały otrzymane dla owcy natomiast najlepsze dla pająka, słonia oraz kurczaka. Rozważając macierz pomyłek można zauważyć iż przypadki zarówno fałszywie negaty-

wne jak i fałszywie pozytywne są rozłożone między wszystkimi klasami, najmniej pomyłek można zauważyć dla pająka. Najwięcej pomyłek można zauważyć między słoniem a koniem, jednak owca była najczęściej klasyfikowana fałszywie negatywna jak i fałszywie pozytywna.

Czas potrzebny do wytrenowania algorytmu to około 40 sekund.

1.8 Krzywa uczenia algorytmów ucznia maszynowego

Jednym z elementów pomocniczych w trakcie wykorzystywania algorytmów ucznia maszynowego jest krzywa uczenia. Przez analizę krzywej uczenia można obserwować, czy model nie został przetrenowany, jak skaluje się model oraz jakie wyniki uzyskuje. Poniżej zostało zaimplementowane wyświetlanie krzywych ucznie z wykorzystaniem walidacji krzyżowej.

[ ]:
```
from sklearn.model_selection import ShuffleSplit
from sklearn.model_selection import learning_curve

def plot_learning_curve(
    estimator,
    X,
    y,
    cv=None,
    n_jobs=None,
    train_sizes=np.linspace(0.1, 1.0, 5),
):

    _, axes = plt.subplots(1, 3, figsize=(20, 5))

    axes[0].set_title('Learning curves')
    axes[0].set_xlabel("Training examples")
    axes[0].set_ylabel("Score")

    train_sizes, train_scores, test_scores, fit_times, _ = learning_curve(
        estimator,
        X,
        y,
        cv=cv,
        n_jobs=n_jobs,
        train_sizes=train_sizes,
        return_times=True,
    )
    train_scores_mean = np.mean(train_scores, axis=1)
    train_scores_std = np.std(train_scores, axis=1)
    test_scores_mean = np.mean(test_scores, axis=1)
    test_scores_std = np.std(test_scores, axis=1)
    fit_times_mean = np.mean(fit_times, axis=1)
    fit_times_std = np.std(fit_times, axis=1)

    # Plot learning curvea
```

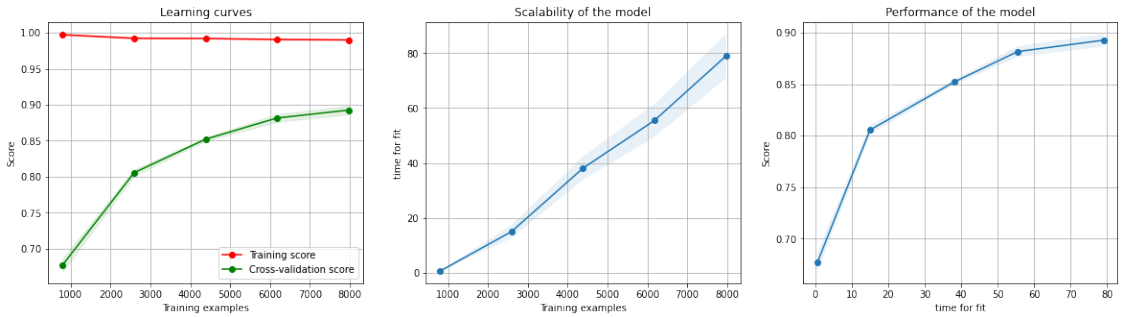
```
    axes[0].grid()
    axes[0].fill_between(
        train_sizes,
        train_scores_mean - train_scores_std,
        train_scores_mean + train_scores_std,
        alpha=0.1,
        color="r",
    )
    axes[0].fill_between(
        train_sizes,
        test_scores_mean - test_scores_std,
        test_scores_mean + test_scores_std,
        alpha=0.1,
        color="g",
    )
    axes[0].plot(
        train_sizes, train_scores_mean, "o-", color="r", label="Training score"
    )
    axes[0].plot(
        train_sizes, test_scores_mean, "o-", color="g", label="Cross-validation␣
↪score"
    )
    axes[0].legend(loc="best")

    # Plot n_samples vs fit_times
    axes[1].grid()
    axes[1].plot(train_sizes, fit_times_mean, "o-")
    axes[1].fill_between(
        train_sizes,
        fit_times_mean - fit_times_std,
        fit_times_mean + fit_times_std,
        alpha=0.1,
    )
    axes[1].set_xlabel("Training examples")
    axes[1].set_ylabel("time for fit")
    axes[1].set_title("Scalability of the model")

    # Plot fit_time vs score
    fit_time_argsort = fit_times_mean.argsort()
    fit_time_sorted = fit_times_mean[fit_time_argsort]
    test_scores_mean_sorted = test_scores_mean[fit_time_argsort]
    test_scores_std_sorted = test_scores_std[fit_time_argsort]
    axes[2].grid()
    axes[2].plot(fit_time_sorted, test_scores_mean_sorted, "o-")
    axes[2].fill_between(
        fit_time_sorted,
        test_scores_mean_sorted - test_scores_std_sorted,
```

```
        test_scores_mean_sorted + test_scores_std_sorted,
        alpha=0.1,
    )
    axes[2].set_xlabel("time for fit")
    axes[2].set_ylabel("Score")
    axes[2].set_title("Performance of the model")

    return plt

cv = ShuffleSplit(n_splits=3, test_size=0.2, random_state=15)
```

1.9 Poniżej zostały przedstawione krzywe uczenia dla algorytmu SVM

[ ]:
```
plot_learning_curve(svm_svc, hog_train, y, cv=cv, n_jobs=4)
```

[ ]:
```
<module 'matplotlib.pyplot' from
'/home/wiktor/Desktop/AlgorytmyUczeniaMaszynowego/ml_project/lib/python3.8/site-
packages/matplotlib/pyplot.py'>
```



1.10 Analiza krzywych uczenia

Obserwując krzywą uczenia można zauważyć iż razem ze wzrostem ilości wykorzystywanych danych do uczenia wynik klasyfikacji dla zbioru testowego się poprawia. Obserwując krzywe uczenia można zauważyć iż efekt dla zbioru uczącego jest cały czas bliski 100% natomiast na zbiorze testowym osiągany jest wynik bliski 90%. Różnica między wynikiem klasyfikacji dla zbioru testowego a uczącego może wskazywać na przetrenowanie, zjawisko może to występować ze względu na zbyt skomplikowany model bądź niezbyt dobre dane wejściowe. Mimo problemów osiągnięto skuteczność na poziomie 90% co jest bardzo dobrym wynikiem dla tego typu algorytmu.

Obserwując `skalowalność` modelu można zauważyć iż razem z ilością danych, czas potrzeby do wytrenowania modelu wzrasta w sposób bliski liniowemu. Taki trend sprawi iż dla większej ilości danych model uczy się dłużej.

Obserwując `wydajność` modelu można zauważyć iż razem ze wzrostem czasu wzrasta także poprawność klasyfikacji przy czym razem ze wzrostem czasu dynamika poprawy klasyfikacji maleje. Duży skok poprawy w zakresie od 0 do 20, niewielki skok poprawy w zakresie od 60 do 80.

2 Wykorzystanie perceptronu wielowarstwowego(MLP) w zadaniu klasyfikacji

Kolejnym wykorzystanym algorytmem w projekcie jest MLP - `Multi layer perceptron`. Jest to algorytm szeroko wykorzystywany w uczeniu nadzorowanym do realizacji zadania regresji jak i klasyfikacji. Algorytm opiera się o wykorzystanie warstwy wejściowej, wyjściowej oraz warstw ukrytych. Warstwa wejściowa to zbiór cech przekazywanych do algorytmy, warstwa wyjściowa to predykcje wytworzone przez model. Warstwy ukryte składają się z perceptronów, które są odpowiedzialne za naukę modelu. Działanie warstw ukrytych opiera się w wykorzystanie wielu perceptronów, do każdego z perceptronów zostaje ważona suma cech, która następnie zostaje poddawana działaniu funkcji aktywacyjne np. funkcji ReLU. Funkcja aktywacyjna wykonuje progowanie, biorąc pod uwagę funkcje ReLU daną wzorem:

$$f(x) = max(0, x),$$

dla wartości mniejszych równych 0 zwraca zero w innym przypadku zwraca daną liczbę.

Algorytm przechodząc od warstwy wejściowej do wyjściowej działa jak algorytm `Feedforward`, finalnie zwracając oczekiwaną predykcje. Następnie algorytm musi się uczyć, ucznie odbywa się poprzez wsteczną propagację. Wsteczna propagacja jest odpowiedzialna za dopasowanie wag dla kolejnych elementów warstw tak aby uzyskać najlepszy efekt uczenia. Wsteczna propagacja to zadanie optymalizacji w którym szukane są wagi, takie które minimalizują błąd uczenia.

Do wykonywanego zadani wykorzystano dostępna implementacje algorytmu w bibliotece sklearn. Wykorzystując ten można manipulować miedzy innymi parametrami:

- activation - funkcja aktywacyjna
- solver - algorytm optymalizacji wykorzystywany do propagacji wstecznej.
- alpha - regularyzacja L2

W tym przypadku jako funkcja aktywacyjna najlepiej sprawdziła się funkcja ReLU, jako algorytm optymalizacji wykorzystano `adam`.

W przypadku `MLP` najbardziej manipulowano parametrem alpha, parametr odpowiada za dodanie funkcji kary, która ogranicza wagi. Sterowanie tym parametrem umożliwia uniknięcie zjawiska przetrenowania, po przez karanie wag o dużych wartościach. Domyślnie parametr przyjmuje wartość zero, jednak w tym przypadku taka wartość dawała gorsze rezultaty. w trakcie testów sprawdzono, że najlepsze rezultaty są osiągane dla wartości równej jeden. Bo badaniach zmniejszono także warstwę ukrytą z rozmiaru (100,) do (50,) w celu uproszczenia modelu.

[ ]:
```
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import classification_report, confusion_matrix,␣
 ↪accuracy_score

mlp_classifier = MLPClassifier(alpha=1, early_stopping=True,hidden_layer_sizes␣
 ↪= (50,), random_state = 15)
mlp_classifier.fit(x_train, y_train)

labels_predicted_mlp = mlp_classifier.predict(x_test)
```

```
print('MLP percentage correct: ', 100*accuracy_score(labels_predicted_mlp,␣
 ↪y_test))
print(classification_report(y_test, labels_predicted_mlp))
print(confusion_matrix(y_test, labels_predicted_mlp))
dump(mlp_classifier, 'models/mlp.joblib')
```

```
MLP percentage correct:  86.00802407221664
              precision    recall  f1-score   support

     chicken       0.88      0.88      0.88       403
    elephant       0.87      0.86      0.87       431
       horse       0.85      0.83      0.84       394
       sheep       0.82      0.84      0.83       360
      spider       0.87      0.89      0.88       406

    accuracy                           0.86      1994
   macro avg       0.86      0.86      0.86      1994
weighted avg       0.86      0.86      0.86      1994

[[356   9  12  12  14]
 [ 11 371  16  19  14]
 [  8  20 326  24  16]
 [ 16  15  20 301   8]
 [ 15  10  11   9 361]]
```

[ ]:
```
['models/mlp.joblib']
```

2.1 Wyniki Algorytmu MLP

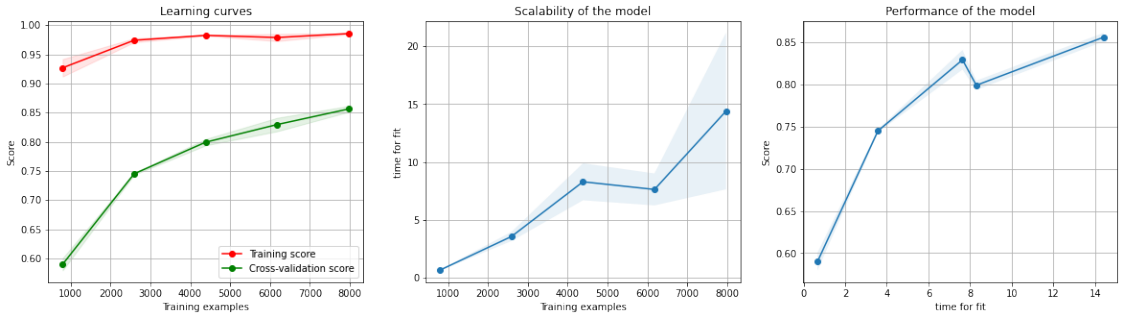
Dla zbioru testowego wydajność jest równa 86 procent co daje względnie dużą dokładność. Najgorsze wyniki klasyfikacji zostały otrzymane dla owcy natomiast najlepsze dla pająka oraz słonia. Rozważając macierz pomyłek można zauważyć iż przypadki zarówno fałszywie negatywne jak i fałszywie pozytywne są rozłożone między wszystkimi klasami, najmniej pomyłek można zauważyć dla pająka. Najwięcej pomyłek można zauważyć między owcą a koniem, jednak to owca była najczęściej klasyfikowana fałszywie negatywnie jak i fałszywie pozytywnie.

Czas potrzebny do wytrenowania algorytmu to około 9 sekund co daje nieporównywalne lepszy czas niż dla algorytmu SVM(ponad 4 razy szybciej).

2.2 Poniżej zostały przedstawione krzywe uczenia dla algorytmu MLP

[ ]:
```
plot_learning_curve(mlp_classifier, hog_train, y, cv=cv, n_jobs=4)
```

[ ]:
```
<module 'matplotlib.pyplot' from
'/home/wiktor/Desktop/AlgorytmyUczeniaMaszynowego/ml_project/lib/python3.8/site-
packages/matplotlib/pyplot.py'>
```



2.3 Analiza krzywych uczenia

Obserwując krzywe uczenia można zauważyć iż razem ze wzrostem ilości wykorzystywanych danych do uczenia wynik klasyfikacji dla zbioru testowego się poprawia. Obserwując krzywe uczenia można zauważyć, że dla zbioru uczącego rezultat klasyfikacji zaczyna się na poziomie 95% i zbliża sie do 100% natomiast na zbiorze testowym osiągany jest wynik powyżej 85%. Różnica między wynikiem klasyfikacji dla zbioru testowego a uczącego może wskazywać na przetrenowanie, zjawisko może to występować ze względu na zbyt skomplikowany model bądź niezbyt dobre dane wejściowe. Mimo problemów osiągnięto skuteczność na poziomie 85%, co jest bardzo dobrym wynikiem dla tego typu algorytmu.

Obserwując `skalowalność` modelu można zauważyć iż razem z ilością danych, czas potrzeby do wytrenowania modelu wzrasta w sposób liniowy przedziałami, finalnie osiągając wartość bliską 15 co jest dużo mniejszą wartości niż przy algorytmie `SVM`. Można także zaobserwować jeden spadek czasu klasyfikacji przy 6 tysiącach obrazów, powodem tego mogą być dodane dane, które znacznie się różniły od wcześniej wykorzystywanych nowe dane.

Obserwując `wydajność` modelu można zauważyć iż razem ze wzrostem czasu wzrasta także poprawność klasyfikacji przy czym razem ze wzrostem czasu dynamika poprawy klasyfikacji malej. W tym przypadku także można zaobserwować jeden spadek w przebiegu.

3 Wykorzystanie k najbliższych sąsiadów(KNN) w zadaniu klasyfikacji

Algorytm k - najbliższych sąsiadów. Jest innym algorytmem niż dwa poprzednie, nie uczy się on funkcji dyskryminacyjnej na podstawie danych uczących, lecz stara się zapamiętać cały zbiór próbek.

Algorytm KNN jest dość prosty, działanie algorytmu można przedstawić w trzech krokach: * wybranie ilości sąsiadów oraz metryki odległości * znalezienie k najbliższych sąsiadów * przydzielenie etykietę klasy poprzez głosowane większościowe

W przypadku tego projektu została została wybrana metryka `Mińkowskiego`, ze współczynnikiem `p = 2`, co w rezultacie daje metrykę euklidesową. Ilość sąsiadów została dobrana poprzez przeszukanie siatki, liczba wybranych sąsiadów wynosi `k=3`(Wynik dobrany na podstawie przeszukania).

```
[ ]: from sklearn.neighbors import KNeighborsClassifier

neigh = KNeighborsClassifier(n_neighbors=3)

neigh.fit(x_train, y_train)

labels_predicted_neigh = neigh.predict(x_test)
print('Knn percentage correct: ', 100*accuracy_score(labels_predicted_neigh,␣
 ↪y_test))
print(classification_report(y_test, labels_predicted_neigh))
print(confusion_matrix(y_test, labels_predicted_neigh))

dump(mlp_classifier, 'models/knn.joblib')
```

```
Knn percentage correct:  80.79237713139418
              precision    recall  f1-score   support

     chicken       0.67      0.90      0.77       403
    elephant       0.81      0.84      0.82       431
       horse       0.95      0.74      0.83       394
       sheep       0.82      0.75      0.78       360
      spider       0.89      0.80      0.84       406

    accuracy                           0.81      1994
   macro avg       0.83      0.81      0.81      1994
weighted avg       0.83      0.81      0.81      1994

[[362   6   2  19  14]
 [ 44 362   3  14   8]
 [ 45  34 290  18   7]
 [ 47  25   6 271  11]
 [ 45  20   5  10 326]]
```

```
[ ]: ['models/knn.joblib']
```

3.1 Wyniki Algorytmu KKN

Dla zbioru testowego wydajność jest bliska 81 procent co daje względnie dużą dokładność. W tym przypadku klasą najgorzej klasyfikowaną jest kurczak na poziomie 67 procent, co znacznie zaniża działanie całego klasyfikatora. Najlepiej klasyfikowaną klasą jest koń(95%), drugi jest pająk(89%).

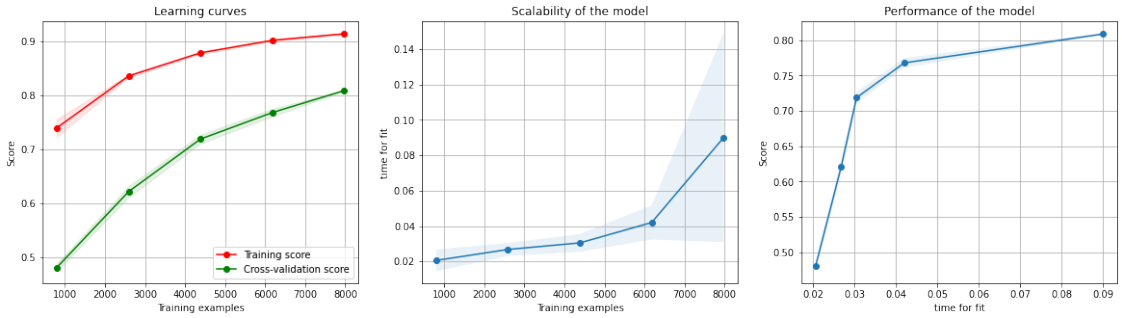
Kurczak był fałszywie pozytywnie klasyfikowany, pomyłka zachodziła dla wszystkich klas równomiernie. Pozostałe pomyłki rozkładały się równomierni.

Czas potrzebny do wytrenowania algorytmu to około ułamki sekund co daje nieporównywalne lepszy czas niż dla poprzednich algorytmów.

3.2 Poniżej zostały przedstawione krzywe uczenia dla algorytmu KNN

[ ]:
```
plot_learning_curve(neigh, hog_train, y, cv=cv, n_jobs=4)
```

[ ]:
```
<module 'matplotlib.pyplot' from
'/home/wiktor/Desktop/AlgorytmyUczeniaMaszynowego/ml_project/lib/python3.8/site-
packages/matplotlib/pyplot.py'>
```



3.3 Analiza krzywych uczenia

Obserwując krzywe uczenia można zauważyć iż razem ze wzrostem ilości wykorzystywanych danych do uczenia wynik klasyfikacji się poprawia zarówno dla zbioru testowego jak i uczącego.

Obserwując `skalowalność` modelu można zauważyć iż wielkość zbioru nie znacznie wpływa na czas nauki modelu. Finalnie zostaje osiągnięty czas poniżej 0.10 sekundy co jest niczym w porównaniu do poprzednich algorytmów.

Obserwując `wydajność` modelu można zauważyć iż razem ze wzrostem czasu wzrasta także poprawność klasyfikacji przy czym razem ze wzrostem czasu dynamika poprawy klasyfikacji malej. Ten wykres zasadniczo jest bardzo zbliżony do wykresów dla wcześniej opisywanych algorytmów.

3.4 Techniki łączeni algorytmów uczenia maszynowego

3.4.1 Głosowanie ze względu na większość głosów

Kolejnym elementem projektu było zaimplementowanie klasyfikatora umożliwiającego łącznie wcześniej wykorzystywanych algorytmów uczenia maszynowego. Zdecydowano się na wykorzystanie łączenia algorytmów z wykorzystaniem głosowania większościowego. Głosowanie większościowe polega wykorzystaniu dowolnych klasyfikatorów następnie porównaniu ich wyników i w efekcie końcowym wybranie najpopularniejszej klasyfikacji. Dodatkowym elementem jest możliwość wykorzystania klasyfikatorów z różnymi wagami co mogłoby usprawnić działanie klasyfikatora.

[ ]:
```
from sklearn.base import BaseEstimator, ClassifierMixin, clone
from sklearn.preprocessing import LabelEncoder

class VoteClassifier(BaseEstimator, ClassifierMixin):
```

```
    def __init__(self, classifiers, weights=None):
        self.classifiers = classifiers
        self.weights = weights
        self.label_encoder = LabelEncoder()


    def fit(self, x_train, y_train):
        return self

    def predict(self, x_test):
        result = np.asarray([clf.predict(x_test) for clf in self.classifiers])

        result = self.transform_labels(result)

        final_result = np.apply_along_axis(self.get_argmax_class, axis=0, arr =␣
↪result)

        return self.label_encoder.inverse_transform(final_result)

    def raise_error_when_weights_and_classifiers_have_different_length(self):
        if self.weights and len(self.weights) != len(self.classifiers):
            raise ValueError(f'Liczba klasyfikatorów musi być równa liczbie wag␣
↪dostępne wagi: {len(self.weights)}, klasyfikatory: {len(self.classifiers)}')

    def transform_labels(self, predictions):
        fitted_labels = []
        for i in range(len(self.classifiers)):
            self.label_encoder.fit(predictions[i])
            fitted_labels.append(self.label_encoder.transform(predictions[i]))

        return fitted_labels


    def get_argmax_class(self,y_value):
         return np.argmax(np.bincount(y_value, weights=self.weights))

max_target = VoteClassifier([ svm_svc, neigh, mlp_classifier ], [1, 0.8, 1])

pred = max_target.predict(x_test)
print('SVM, MLP and KNN percentage correct: ', 100*accuracy_score(pred, y_test))
print(classification_report(y_test, pred))
print(confusion_matrix(y_test, pred))
```

```
SVM, MLP and KNN percentage correct:  89.86960882647944
              precision    recall  f1-score   support

     chicken       0.89      0.93      0.91       403
```

```
    elephant       0.89      0.93      0.91       431
       horse       0.90      0.86      0.88       394
       sheep       0.89      0.84      0.86       360
      spider       0.93      0.91      0.92       406

    accuracy                           0.90      1994
   macro avg       0.90      0.90      0.90      1994
weighted avg       0.90      0.90      0.90      1994

[[376   4   7   7   9]
 [  9 402   4  12   4]
 [ 11  20 340  15   8]
 [ 13  17  19 304   7]
 [ 14  11   6   5 370]]
```

3.5 Efekty wykorzystania głosowania większościowego

Jak można zauważyć w efekcie końcowym wykorzystanie techniki głosowania większościowego pozwoliło na delikatnie poprawić wynik algorytmu `SVM`. Analizując macierz pomyłek można zauważyć iż pomyłki rozkładają się między klasami w sposób równomierny, nie da się wyróżnić jakiegoś konkretnego trendu.

W tym przypadku wagi zostały dobrane eksperymentalnie. Prawdopodobniej dokładne przeszukanie wag mogłoby skutkować delikatnie lepszym rezultatem. Niestety na takie przeszukanie zabrakło już czasu.

4 Posumowanie

W ramach projektu wykorzystano algorytmy w celu klasyfikacji zwierząt na obrazach. Kolejne algorytmy radziły sobie w następujący sposób:

- SVM - 89.67%
- MLP - 86.00%
- KNN - 80.79%

Najlepiej poradził sobie algorytm `SVM`, najgorzej natomiast `K-nn`. Co ciekawe`k-nn` działał najszybciej a `SVM` najwolniej. Mimo zauważalnych problemów które widać na krzywych uczenia finalnie osiągnięto rezultaty od 80% do 90% poprawnej klasyfikacji. Najlepszy rezultat uzyskano dla głosowania większościowego.

Projekt pozwolił na poznanie różnych algorytmów uczenia maszynowego, badania podstawowych metryk działania algorytmów a także zapoznanie się z bibliotekami dostępnymi dla języka `Python3`. W trakcie projektu poznano także ze sposobami walki z przetrenowaniem modelu(rozszerzanie zbioru danych, upraszczanie modelu).

W przyszłości projekt mógłby być rozwinięty o dopasowanie najlepszych wag dla klasyfikatora większościowego.