Praktyczne zastosowanie metod SI – Laboratorium nr 3

Kolejnym klasyfikatorem z pakietu sklearn, który wykorzystamy, jest **MLPClassifer – sieć neuronowa (uczenie nadzorowane)**

Warto zwrócić uwagę na parametry:

- hidden_layer_sizes – liczba i rozmiar warstwy/warstw ukrytych – domyślnie 1 warstwa ze 100 neuronami
- activation – funkcja aktywacji (dostępne są: identity, logistic, tanh, relu)
- solver – algorytm do optymalizacji wag:
    - lbfgs – metoda z rodziny quasi-Newton, domyślnie wybierana
    - sgd, adam – metody stochastyczne
- alpha – określa "siłę" regularyzacji l2, czyli kary za sumę kwadratów wag. Służy to do zmniejszenia złożoności modelu I zapobiega przeuczeniu. Przy dużej wartości wiele wag się wyzeruje.
- batch_size – "porcja" danych pobierana do uczenia przez optymalizatory stochastyczne. Przy lbfgs parameter jest bez znaczenia. Domyślna wartość: minimum(200, liczba_probek)
- max_iter – liczba iteracji
- tol – jeśli w kolejnych iteracjach różnica w wartości w funkcji celu jest mniejsza niż tol, uczenie jest przerywane (lub w przypadku osiągnięcia maksymalnej liczby iteracji)
- verbose – wyświetlanie komunikatów o przebiegu uczenia

Przykład użycia:

```
from sklearn.neural_network import MLPClassifier
from sklearn import datasets
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt

digits = datasets.load_digits(n_class=10) # zbiór przedstawiający cyfry
n_samples = digits['target'].shape[0]
X_train = digits['data'][:n_samples//2] # podział danych na pół
y_train = digits['target'] [:n_samples//2]
X_test = digits['data'][n_samples//2 :]
y_test = digits['target'] [n_samples//2:]

# wyświetlenie pierwszej próbki ze zbioru uczącego
plt.imshow(X_train[1,:].reshape(8, 8), cmap='gray') # dane do uczenia sieci są zapisane jako wektor,
#tutaj robimy zamianę do oryginalnych rozmiarów obrazka
#plt.show()

clf = MLPClassifier(hidden_layer_sizes=(10,), solver='lbfgs', max_iter=1000) # 1 ukryta warstwa z 10
#neuronami
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
```

ZADANIE
Proszę zrobić prostą pętlę i narysować wykres zmiany w dokładności klasyfikacji oraz czasu uczenia w zależności od liczby neuronów: 1, 3, 5, …, 25.

Można również poeksperymentować z parametrami, szczególnie z parametrem alpha.

Pomiar czasu uczenia:

```
from timeit import default_timer as timer
…
start = timer()
clf.fit(X_train, y_train)
end = timer()

print('Elapsed time: {} s'.format(end - start)) # czas w sekundach
```

Jeszcze jeden z popularnych klasyfikatorów to **regresja logistyczna** (klasyfikator binarny)

Model klasyfikatora określa funkcja z wartościami odpowiedzi z zakresu [0, 1] – prawdopodobieństwem klasy pozytywnej pod warunkiem wartości atrybutów w próbce:

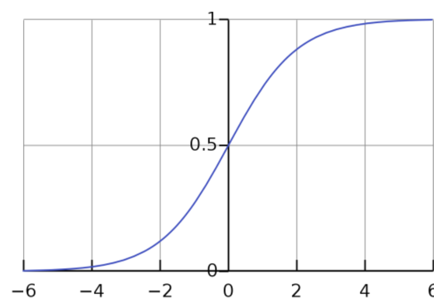
$$P(1|x_1, x_2, \ldots, x_K) = 1 / (1+\exp(-w_0 - w_1x_1 - \ldots - w_Kx_K))$$

(x_1 do x_k to atrybuty, w to wagi – podobnie jak w regresji liniowej)

Można też spotkać się z krótkim zapisem:

$$1 / (1+\exp(z))$$

Funkcję tą nazywamy funkcją logistyczną, a jej przykładowy wykres wygląda tak:



Odpowiedź klasyfikatora jest ustalana na podstawie progu decyzyjnego – jeśli prawdopodobieństwo klasy 1 jest wyższe niż próg, klasyfikator odpowiada klasą 1. W przeciwnym razie – klasą 0.

Uwaga: w wersji z sklearn domyślnie aktywowana jest **regularyzacja l2**. Regularyzacją można sterować przy użyciu parametru C – jest to odwrotność alpha z sieci neuronowej tzn. im większa wartość, tym mniejsza "siła" regularyzacji.

Szkic użycia:

```
from sklearn.linear_model import LogisticRegression
…
digits = datasets.load_digits(n_class=2)
…
clf = LogisticRegression()
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
```

Użycie regresji logistycznej dla klasyfikacji wieloklasowej wymaga podejścia **one vs. rest**. (W wersji sklearn jest obsługa więcej niż 2 klasy w wariancie one vs. rest i nie wymaga to dodatkowych ustawień)

Klasyfikacja **one vs rest**

Załóżmy teraz, że nasza implementacja regresji logistycznej radzi sobie tylko z klasyfikacją binarną. Co wtedy zrobić?

Możemy użyć klasy OneVsRestClassifier z pakietu sklearn.multiclass

Umożliwi ona wykonanie klasyfikacji wieloklasowej nawet klasyfikatorem binarnym. Przykładowo dla 3 klas wykona klasyfikację: klasa (1) 1 jako pozytywna, 2 oraz 3 jako negatywna (2) 2 jako pozytywna, pozostałe jako negatywne (3) 3 jako pozytywna, pozostałe jako negatywne.

W ten sposób uzyskamy 3 klasyfikatory – każdy dopasowany do rozpoznawania "swojej" klasy.

Przykład użycia

```
from sklearn.multiclass import OneVsRestClassifier
…
clf = OneVsRestClassifier(LogisticRegression())
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
```

Proszę sprawdzić działanie regresji logistycznej na zbiorze digits (10 klas) w dwóch wariantach:

1. "Ukryta" w implementacji sklearn obsługa wieloklasowej klasyfikacji
2. Użycie OneVsRestClassifier

ZADANIE:
Proszę napisać samodzielnie klasyfikację OneVsRest. Klasa powinna mieć metody:
__init__(model) – przekazanie wcześniej stworzonego modelu
fit(X, y) – stworzenie tyle kopii modelu i nauczenie każdej
predict(X) – odpowiedź na zasadzie – model dla *k*-tej klasy zwrócił najwyższe prawdopodobieństwo -> próbka jest klasy *k*
predict_proba(X) – zwraca prawdopodobieństwa (będzie to macierz o wymiarze liczba_próbek_do_predykcji x liczba_klas)

Modele można przechowywać w polu w klasie – jako listę albo słownik (klucz – etykieta klasy)

Przy okazji klasyfikacji warto wspomnieć o klasyfikatorze nazywanym **Zero-R** lub **dummy classifier**. Jak sama nazwa mówi, jest to chyba najprostszy klasyfikator jaki można sobie wyobrazić – nowym próbkom przypisuje zawsze tą samą klasę: tą, która była najczęstsza w zbiorze uczącym. Nie ma on zastosowania w rozwiązywaniu praktycznych problemów, ale służy jako "miernik sensowności" innych klasyfikatorów – używany przez nas klasyfikator powinien dawać lepszą niż Zero-R jakość klasyfikacji.

Użycie w sklearn:

```
from sklearn.dummy import DummyClassifier
clf = DummyClassifier(strategy='most_frequent')
```

W ramach testów klasyfikatora Zero-R można posłużyć się funkcją wykonującą automatyczny podział na zbiór uczący i testowy: **train_test_split** z pakietu **sklearn.model_selection**

Jeszcze inny prosty (ale dający lepsze efekty) klasyfikator to **OneR**. Działa w ten sposób, że wyboru klasy dokonuje na podstawie jednego, mającego na klasę największy wpływ atrybutu. Nie znalazłem implementacji w sklearn, jest ona za to dostępna w programie Weka (za pośrednictwem Javy lub z użyciem interfejsu graficznego).

Zamiast wykonywać podział na zbiór uczący i testowy, można nie "marnować" danych i każdą próbkę wykorzystać zarówno do uczenia, jak i do testowania. Służy do tego **K-krotna walidacja krzyżowa** (k-fold cross-validation). Jest to procedura oceny modelu (w naszym przypadku klasyfikatora) polegająca na podziale danych na k rozłącznych podzbiorów i dla każdego podzbioru wykonanie 2 operacji:

1. Nauczenie klasyfikatora na pozostałych k-1 podzbiorach (połączonych).
2. Testowanie klasyfikatora na rozpatrywanym podzbiorze

Miarą jakości klasyfikacja jest średnia miar z k powtórzeń.

Technika ta jest szczególnie przydatna gdy mamy mały (pod względem liczby próbek) zbiór danych.

Funkcja w scikit-learn: sklearn.model_selection.cross_val_score

```
cross_val_score(clf, X, y, cv=5) # cv – tam podajemy k
```

Większe możliwości (więcej metryk jednocześnie, czasy uczenia) daje funkcja cross_validate

---

Kolejnym zagadnieniem związanym z klasyfikacją (i nie tylko) jest procedura **Grid Search**, służąca do doboru parametrów, które nie są ustalane automatycznie w czasie uczenia modelu – dobrym przykładem jest alpha w sieci neuronowej lub C w regresji logistycznej. Zamiast ręcznie wybierać, jaka wartość parametru jest odpowiednia, można się posłużyć procedurą, która sprawdzi działanie klasyfikatora z wcześniej określonymi wartościami, wykonując dla każdej wartości parametru *k*-krotną walidację krzyżową.

Użycie:

```
from sklearn.model_selection import GridSearchCV
…
params = {'C':np.arange(0.1,0.4, 0.1)}
print(params) # słownik, można podać więcej parametrów
model = LogisticRegression(max_iter=300)
clf = GridSearchCV(model, param_grid=params)
clf.fit(digits.data, digits.target) # domyślnie z 5-krotną walidacją krzyżową
print(clf.cv_results_) # raport z uczenia
print(clf.best_estimator_) # najlepszy model
```

Przydatny parametr dla GridSearchCV: n_jobs. Ustawienie -1 zrównolegli wykonywanie funkcji na wszystkie procesory/rdzenie.