Politechnika Wrocławska
Wydział Elektroniki
Informatyka

Zastosowania informatyki w medycynie

Analiza stopnia złośliwości komórek nowotworowych wykorzystująca sieci neuronowe.

Autorzy:
Łukasz Matysiak
Kamil Machnicki

Prowadzący:
Dr inż. Bartosz Krawczyk

Termin zajęć:
Czwartek, godz. 11:15

Wrocław
29 maja 2016

Spis treści

Rozdział 1

Wstęp

1.1 Cele i założenia projektowe

Celem projektu było zapoznanie się z tematem uczenia maszynowego, a konkretnie z sieciami neuronowymi. Realizacja projektu miała umożliwić zrozumienie zasady działania oraz porównanie wydajności sieci neuronowych używających propagacji wstecznej oraz wariantu o nazwie *Extreme Learning Machines,* na podstawie analizy danych z badań cytologicznych.

1.2 Wstęp teoretyczny

Rozwój uczenia maszynowego w ostatnich latach jest bardzo ważny z punktu widzenia wielu dziedzin, jednak szczególnie ważny dla medycyny.

Komputery stają się szybsze i są w stanie przetwarzać większe ilości danych, co przekłada się na możliwość użycia ich do analizowania np. wyników badań. Dzięki rozwojowi techniki, mogą one pomagać lekarzom w diagnozowaniu, co pozwala na szybsze decyzje na temat leczenia pacjenta.

Sieci neuronowe stanowią jedno z możliwych podejść do tematu uczenia maszynowego. Są one modelowane na podobieństwo neuronów w mózgu, a zastosowanie znajdują między innymi w problemach klasyfikacji – np. klasyfikacji stopnia złośliwości raka, tak jak to miało miejsce w projekcie.

Wzrost popularności zawdzięczają pojawieniu się szybszych komputerów oraz ogromnej ilości danych dostępnych do uczenia ich, co pozwoliło na zredukowanie wpływu głównych wad sieci neuronowych – wolnego uczenia się oraz wymagania dużych zestawów danych uczących.

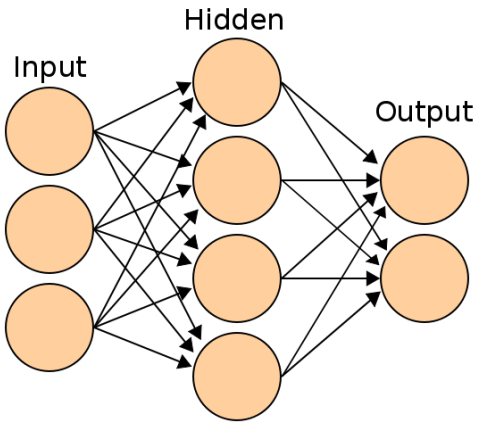
1.2.1 Budowa sieci neuronowych

Prosta sieć neuronowa może być zbudowana z trzech warstw neuronów:

- warstwy neuronów wejściowych,

- warstwy neuronów ukrytych,
- warstwy neuronów wyjściowych.

Każda z warstw odbiera dane, a następnie po obliczeniu wartości funkcji aktywacji, przesyła je do kolejnej warstwy.



Rysunek 1.1: Schemat prostej sieci neuronowej. Źródło: wikipedia.org.

Rozdział 2

Aplikacja

2.1 Wykorzystane technologie

Aplikacja implementująca sieci neuronowe na potrzeby realizacji tematu oraz wykonująca badania, zrealizowana została przy wykorzystaniu języka `Python` w wersji `3.5`.

Głównym modułem, na którym oparto implementację, był `scikit-learn`. Wykorzystano go przede wszystkim do zbudowania sieci neuronowej uczoną metodą wstecznej propagacji błędu, a także skorzystano z wielu udostępnionych tam rozwiązań, jak na przykład walidacja krzyżowa i selekcja cech. Użyto modułu w wersji deweloperskiej `0.18.dev0`[1], gdyż dopiero od tej wersji można tam korzystać z sieci neuronowych.

Jako implementację drugiego badanego algorytmu - *Extreme Learning Machines*, posłużył moduł `Python-ELM`[2]. Wyznaczony on był na oficjalnej stronie *ELM*[3] jako jedna z bazowych implementacji tegoż algorytmu. Udostępniony kod nie był, niestety, przystosowany do działania pod wersją *Pythona 3*, na potrzeby projektu przerobiono więc kod tak, aby dało się go używać.

Wszystkie wykorzystane moduły i ich przeznaczenie widnieją na poniższej tabeli:

Tabela 2.1: Wykorzystane moduły.

Nazwa	**Wersja**	**Opis**
`scikit-learn`	0.18	Sieć neuronowa uczona metodą wstecznej propagacji błędu, selekcja cech, walidacja krzyżowa, macierz błędu.
`Python-ELM`	0.3	*Extreme Learning Machines.*
`numpy`	1.11.0	Obliczenia numeryczne.
`matplotlib`	1.5.1	Tworzenie wykresów.

[1] http://scikit-learn.org/dev/documentation.html
[2] https://github.com/dclambert/Python-ELM
[3] http://www.ntu.edu.sg/home/egbhuang/elm_codes.html

2.2 Opis implementacji

Na aplikację składa się kilka modułów:

- `main.py` - główny moduł uruchamiający badania.
- `algorithms.py` - uruchamia oba algorytmy.
- `dataset.py` - importuje dane z pliku CSV.
- `grapher.py` - tworzy wykresy z wyników badań.
- `helper.py` - zawiera klasy pomocnicze do przechowywania danych.
- `consts.py` - zawiera ustawienia badań i parametry dla algorytmów.

Po uruchomieniu badania przez moduł `main.py`, w module `algorithms.py` odbywa się wykonywanie owych badań. Najpierw, na zbiorze zawierającym dane uczące `X` oraz tablicę wyników `y` wykonywana jest 10-krotna walidacja krzyżowa przy pomocy `StratifiedKFold` z modułu `sklearn.model_selection`, w wyniku której otrzymywane są indeksy elementów podzielonych na dwa podzbiory - zbiór uczący i testowy:

```
StratifiedKFold(n_folds=n_folds).split(X, y):
```

W tym momencie następuje też iteracja przez wszystkie wyniki walidacji, gdzie w każdej iteracji odbywa się selekcja `n`-cech, gdzie `n` to liczba cech, dla której aktualnie wykonywane są badania:

```
SelectKBest(k=k_best_features).fit(X, y).get_support(indices=True)
```

Na tak wyselekcjonowanych cechach odbywa się uruchomienie dwóch algorytmów - uczonego metodą wstecznej propagacji błędu (algorytm *BP*) oraz jego szybka wersja losowa *Extreme Learning Machines* (algorytm *ELM*). Najpierw tworzona jest instancja klasyfikatora algorytmu *BP* (opis parametrów użytych do badania znajduje się w rozdziale o badaniach), następnie odbywa się jego uczenie przekazując do funkcji `fit` zbiór uczący. Na końcu dokonywana jest klasyfikacja algorytmu przekazując do funkcji `score` zbiór testowy.

```
clf = MLPClassifier(...)
clf.fit(X_train, y_train)
score = clf.score(X_test, y_test)
```

Dla algorytmu `ELM` procedura wygląda bardzo podobnie:

```
elmc = ELMClassifier(...)
elmc.fit(X_train, y_train)
score = elmc.score(X_test, y_test)
```

Na końcu dla każdego algorytmu wyliczana jest macierz pomyłek, korzystając z `confusion_matrix` z modułu `sklearn.metrics`.

```
conf_matrix = confusion_matrix(y_test, clf.predict(X_test))
conf_matrix = confusion_matrix(y_test, elmc.predict(X_test))
```

2.3 Wdrożenie

W celu przygotowania aplikacji do działania, należy mając zainstalowaną wersję `Pythona 3.5` dodać katalog projektu do zmiennej środowiskowej `PYTHONPATH`. Aby tego dokonać, w katalogu projektu trzeba uruchomić komendę:

```
$ export PYTHONPATH="${PYTHONPATH}:${PWD}"
```

Kolejnym krokiem jest uruchomienie skryptu instalacyjnego `install.sh`, który to zainstaluje wymagane pakiety z pliku `requirements.txt` oraz pobierze poprawiony moduł `Python-ELM` do katalogu `modules`:

```
./install.sh
```

Teraz można uruchomić aplikację poprzez wywołanie skryptu głównego `main.py`:

```
python3 main.py
```

Działanie aplikacji powinno się zakończyć wygenerowanymi wykresami.

Rozdział 3

Badania

3.1 Przeprowadzone badania

Zgodnie z założeniami, badania przeprowadzono dla 5 różnych liczb neuronów w warstwie ukrytej: **5, 10, 20, 40, 60**.

Badania rozpoczynają się od iteracji przez wybrane liczby neuronów w warstwie ukrytej. W każdej iteracji uruchamiane są testy dla różnych liczb cech, począwszy od 1 a skończywszy na maksymalnej liczbie cech w zbiorze testowym, w tym wypadku 32. W tym momencie następuje n-krotne wykonanie uczenia i klasyfikacji, a z n-próbek wyciągane są wartości skrajne min-max, liczona jest średnia oraz odchylenie standardowe. Uzyskane w ten sposób wartości służą w późniejszym etapie do odpowiedniego zobrazowania wyników na wykresach. Na potrzeby badań uznano, że 5 powtórzeń będzie wystarczające.

Oba algorytmy uruchamiano z różnymi wartościami stałych parametrów. Oto lista niezmiennych parametrów z jakimi uruchamiano algorytm `BP`:

Tabela 3.1: Parametry algorytmu `BP`.

Parametr	**Wartość**	**Opis**
`algorithm`	`sgd`	Algorytm działania (*stochastic gradient descent*).
`max_iter`	`10000`	Maksymalna liczba iteracji.
`alpha`	`1e-6`	Parametr regularyzacji L2.
`learning_rate`	`constant`	Tempo uczenia.
`activation`	`logistic`	Funkcja aktywacji warstwy ukrytej.
`random_state`	`None`	Ziarno dla generatora liczb losowych.

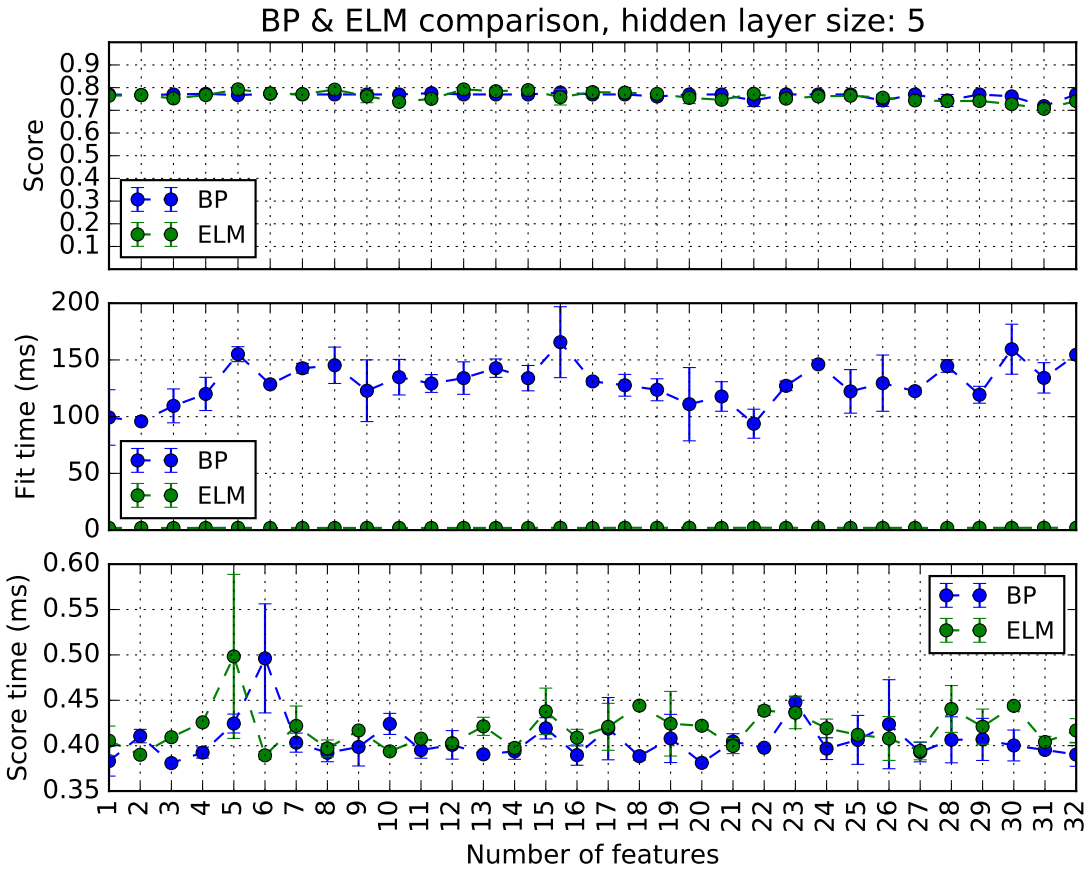
Dla algorytmu `ELM` powyższe parametry były takie same, bądź nie trzeba było ich wprowadzać, z jedną różnicą, gdzie jako funkcji aktywacji użyto funkcję `multiquadric`.

3.2 Porównanie algorytmów

Dla 5 neuronów w warstwie ukrytej zauważyć można, że oba algorytmy osgiągneły prawie takie same wyniki oscylujące w okolicy 76%, niezależne od liczby wyselekcjonowanych cech.

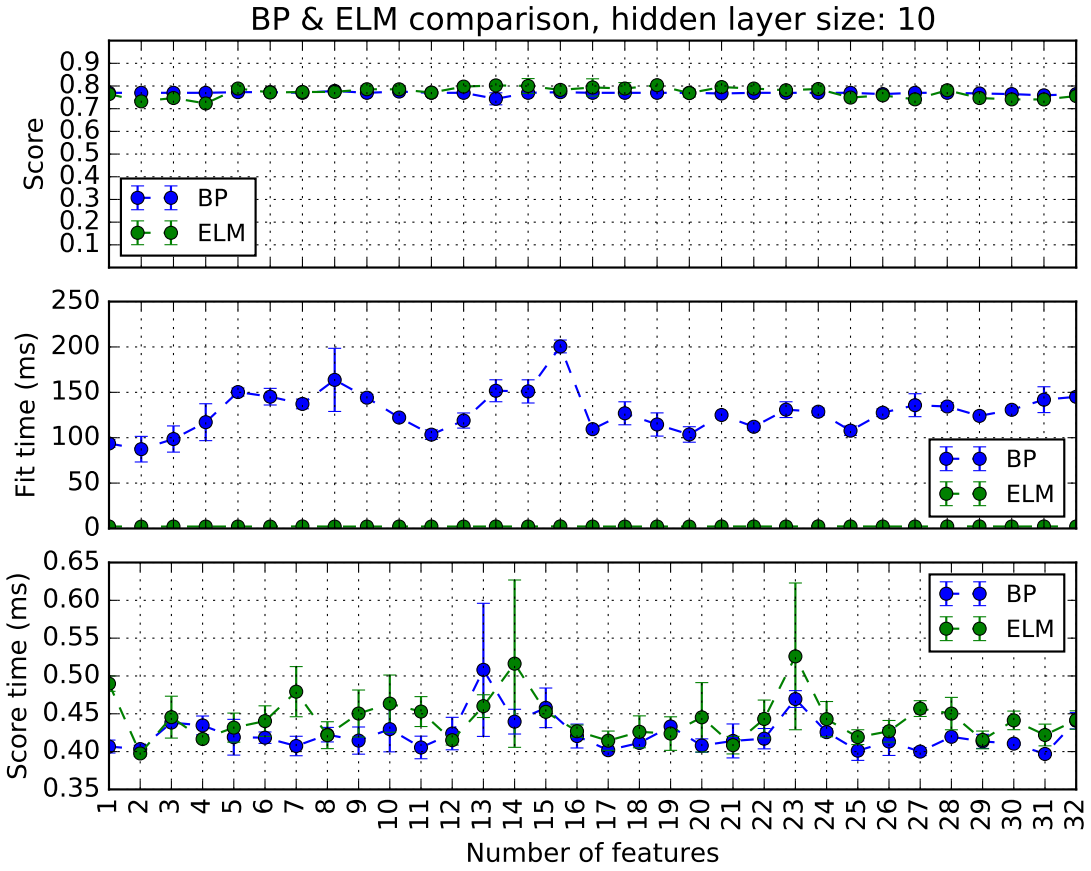
Również czasy uczenia okazały się bardzo podobne. Wystąpiły tutaj jednakże większe fluktuacje, spowodowane przede wszystkim niedokładnością w mierzeniu bardzo małych czasów (rzędu 0.5 milisekundy) oraz dużym odchyleniem standardowym sygnalizowanym na wykresie przez słupki błędu.

Największą różnicę zauważyć jednak można w czasach uczenia się, gdzie algorytm `ELM` osiągnął o wiele lepsze czasy rzędu 2 milisekund od algorytmu `BP`, którego średni czas oscylował w okolicy 130 milisekund. Zauważyć zatem można wyraźnie, że algorytm `ELM` jest średnio 65 razy szybszy od algorytmu `BP`.



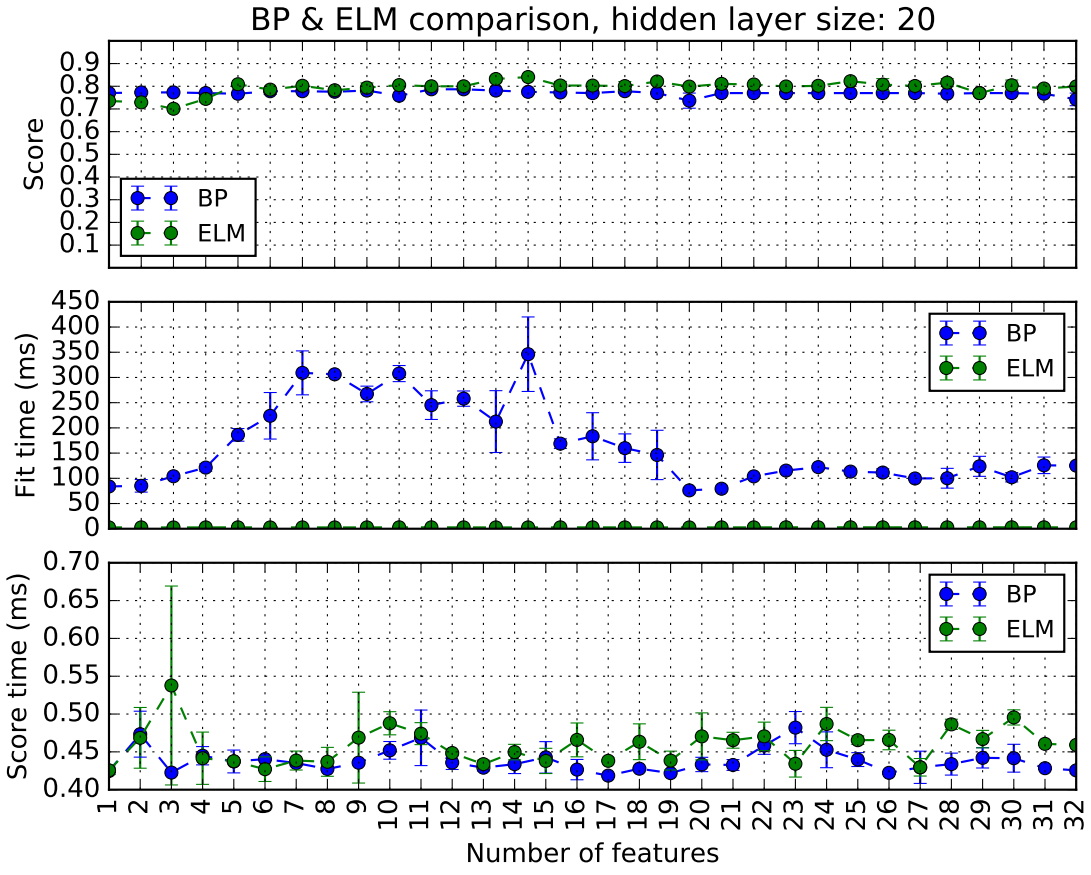
Rysunek 3.1: 5 neuronów w warstwie ukrytej - porównanie obu algorytmów

Dla 10 neuronów w warstwie ukrytej zachodzą te same zależności. Widoczny jest jednak nieznaczny wzrost czasu uczenia dla algorytmu `BP` do około 250 milisekund w okolicy 10-14 wyselekcjonowanych cech. Algorytm `ELM` pozostaje bez zmian.



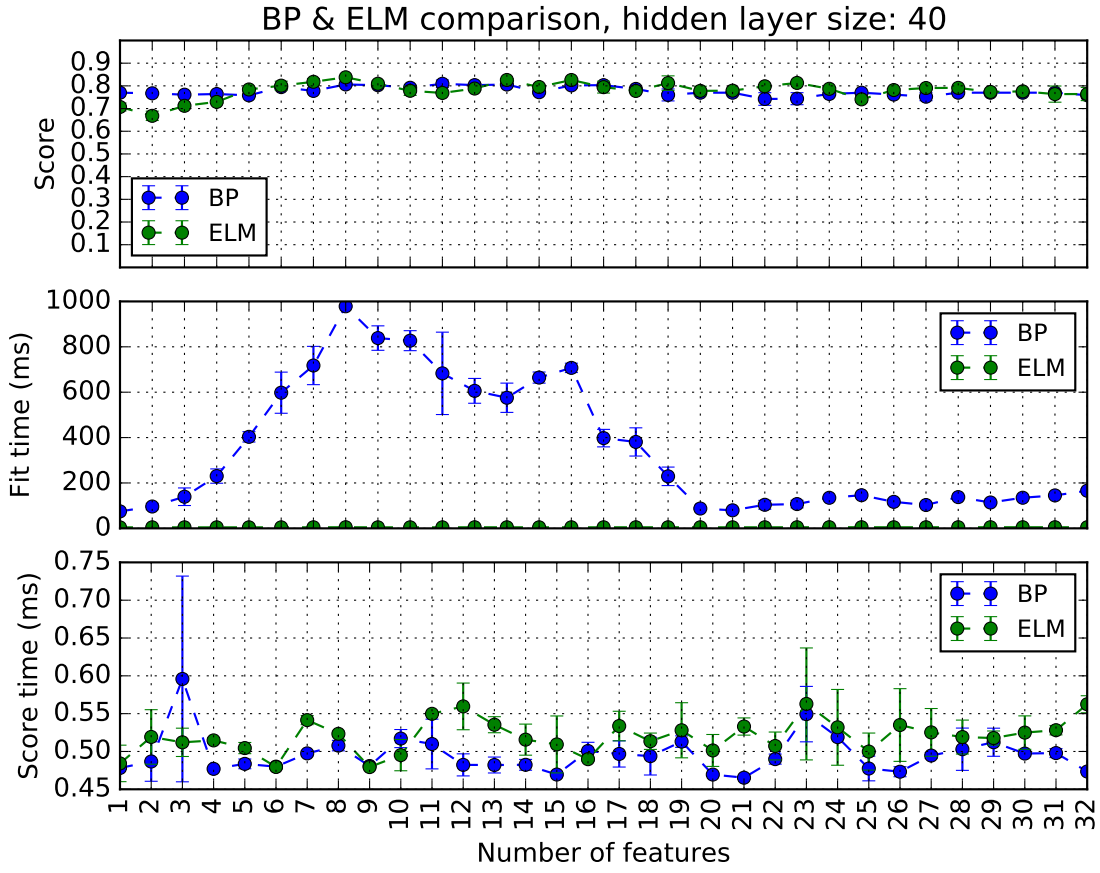
Rysunek 3.2: 10 neuronów w warstwie ukrytej - porównanie obu algorytmów

Dla 20 neuronów w warstwie ukrytej zdecydowanie zauważyć już można tendencję w czasie uczenia algorytmu BP, gdzie do około 7 wyselekcjonowanych cech czas znacząco wzrasta, później do około 16 sech pozostaje względnie stały, następnie maleje i od 22 cech utrzymuje się już na sałym, niskim poziomie rzędu 100 milisekund. Wytłumaczyć tę zależność można zjawiskiem przeuczenia, gdzie algorytm osiągnął lokalne minima, a następnie je wykorzystywał przy kolejnych uczeniach.



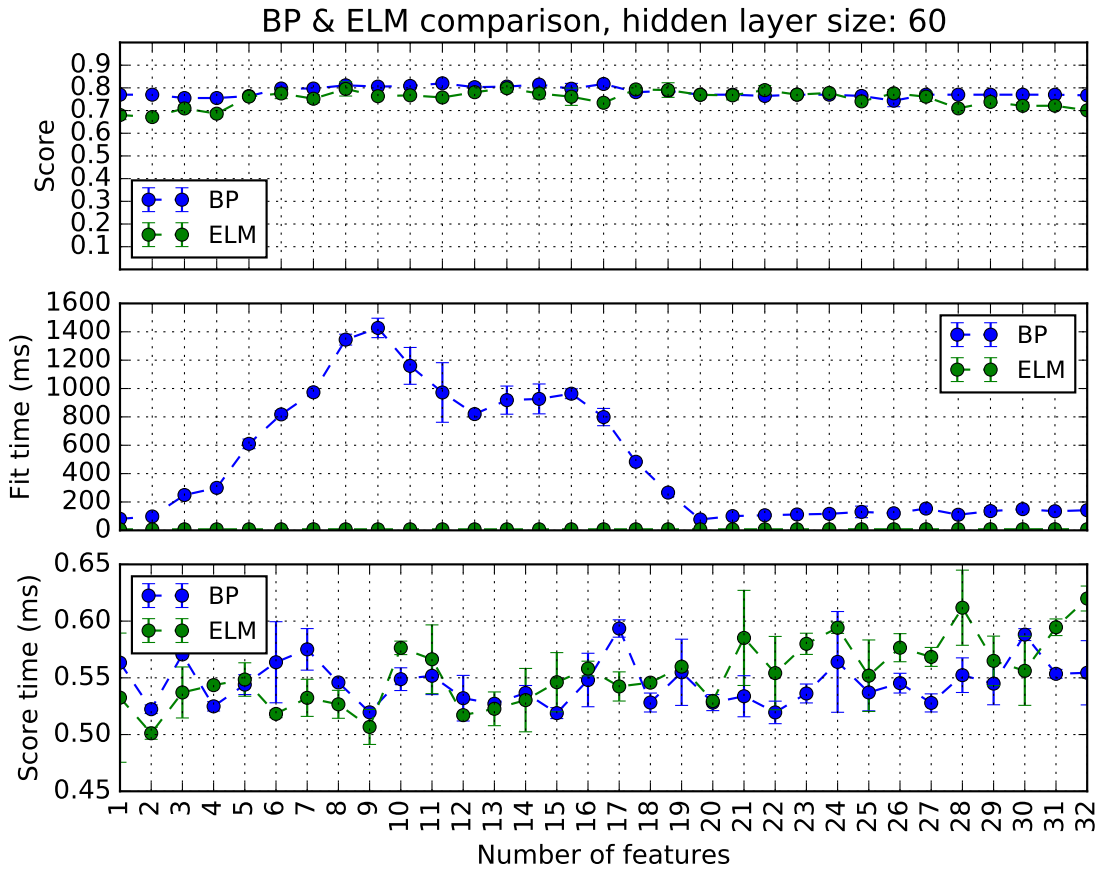
Rysunek 3.3: 20 neuronów w warstwie ukrytej - porównanie obu algorytmów

Dla 40 neuronów w warstwie ukrytej widać już bardzo wyraźnie zauważone wcześniej tendencje dla algorytmu `BP`. Reszta zależności pozostaje bez zmian.



Rysunek 3.4: 40 neuronów w warstwie ukrytej - porównanie obu algorytmów

Dla 60 neuronów w warstwie ukrytej widać, że powyższe zależności pogłębiły się znacząco i czas uczenia algorytmu BP osiągał 1.2 sekundy.

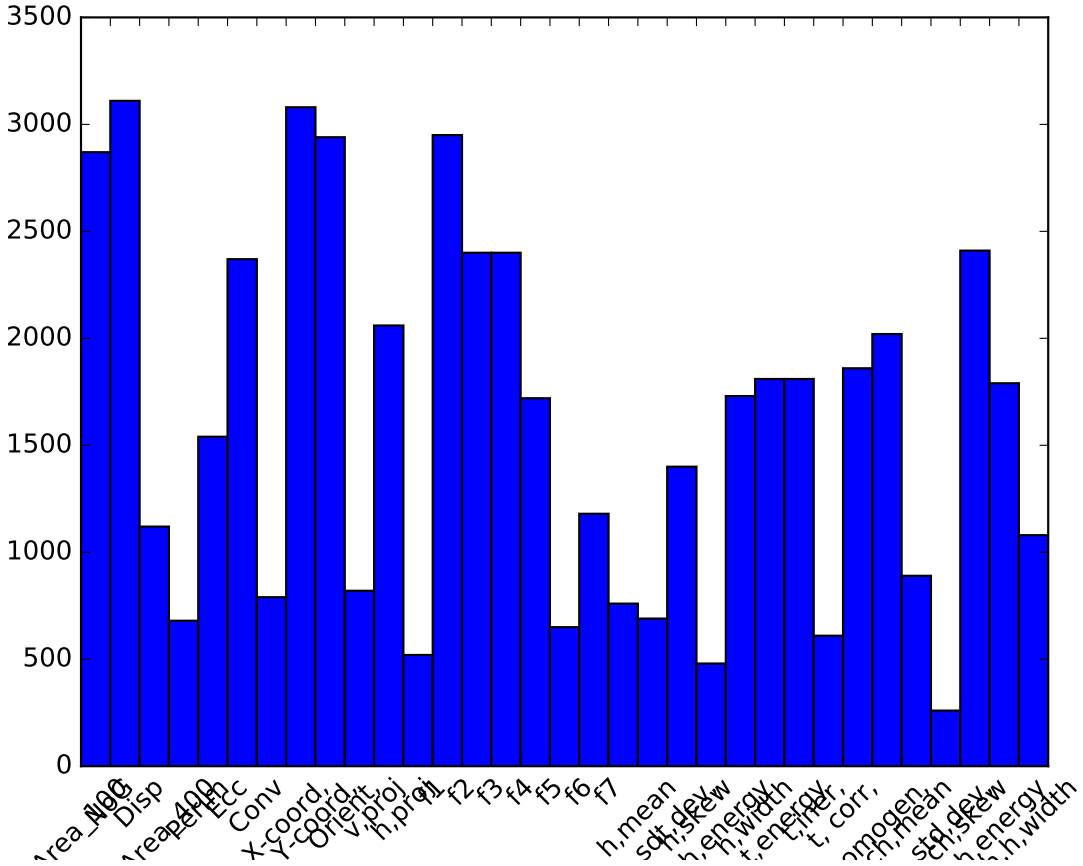


Rysunek 3.5: 60 neuronów w warstwie ukrytej - porównanie obu algorytmów

3.3 Selekcja cech

Selekcja cech polega na wybieraniu podzbioru cech w celu ograniczenia czasu uczenia, uproszczenia modelu oraz minimalizacji zjawiska przeuczenia.

W projekcie skorzystano z klasy `SelectKBest`, z modułu `sklearn.feature_selection`.



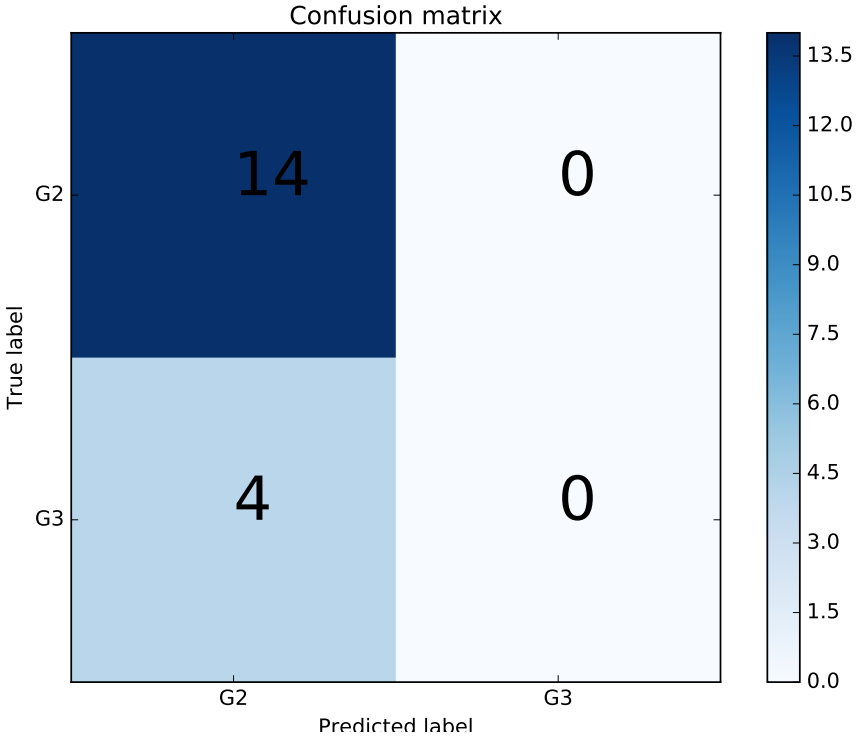
Rysunek 3.6: Częstość wybierania cech

3.4 Macierz pomyłek

Macierz pomyłek umożliwia zobrazowanie jak dobrze klasyfikator radzi sobie ze swoim zadaniem.

Dla problemu z dwiema klasami – tak jak w aktualnie rozpatrywanym problemie – macierz dzieli się na cztery części, w których zliczane są obiekty sklasyfikowane poprawnie – umieszczone na przekątnej macierzy – oraz błędnie – umieszczone poza przekątną.

W projekcie skorzystano z funkcji `confusion_matrix`, z modułu `sklearn.metrics`.



Rysunek 3.7: Przykładowa macierz pomyłek

Rozdział 4

Wnioski

- Sieci neuronowe mogą stanowić potężne narzędzie w walce z chorobą.
- Parametry sieci należy dobrać odpowiednio do problemu, aby unikać zjawiska przeuczenia.
- Dane, z którymi przyszło nam pracować nie były zbalansowane – 76% przypadków znajdowało się w klasie G2 – co powodowało problemy przy uczeniu klasyfikatora, który wszystkie przypadki klasyfikował jako G2.
- Sieci neuronowe potrzebują sporej ilości danych uczących, aby poprawnie generalizować problemy.
- ELM daje bardzo zbliżone jakościowo wyniki do sieci neuronowych ze wsteczną propagacją, jednocześnie proces uczenia jest o wiele szybszy.
- Wbrew początkowej intuicji, kilka najlepszych cech zebranych razem wcale nie musi dawać najlepszych wyników.
- Procesy selekcji cech oraz walidacji krzyżowej nie mogą być przeprowadzane osobno.

Bibliografia

[1] Scikit-learn documentation. `http://scikit-learn.org/dev/documentation.html`. Aktualne dnia: 2016-05-28.

[2] B. Krawczyk, M. Galar, Łukasz Jelen, F. Herrera. *Evolutionary undersampling boosting for imbalanced classification of breast cancer malignancy.* Elsevier, 2015.