Politechnika Warszawska

Wydział Matematyki i Nauk Informacyjnych

Komputerowe wspomaganie technik kryminalistycznych

Rok akademicki 2015/2016

Dokumentacja projektu
Identyfikacja twarzy

Autorzy:

inż. Patryk Bęza

inż. Marek Kozak

inż. Krzysztof Małaśnicki

Wykładowca:

dr inż. Magdalena Szeżyńska

14 czerwca 2016

Spis treści

Streszczenie

Niniejszy dokument powstał w ramach dokumentacji projektu zespołowego z przedmiotu *Komputerowe wspomaganie technik kryminalistycznych* w semestrze letnim roku akademickiego 2015/2016 na *Wydziale MiNI Politechniki Warszawskiej*. Ma on za zadanie udokumentować ideę działania aplikacji identyfikującą twarze, uzasadnienie wyboru zastosowanych algorytmów, sposób jej użycia, wykorzystane narzędzia i biblioteki.

1 Wstęp

W tym rozdziale przedstawiono założenia wstępne projektu pt. „Identyfikacja twarzy".

1.1 Identyfikacja twarzy

Identyfikacja twarzy to przypisanie tożsamości do dwu- lub trójwymiarowego zdjęcia twarzy. Przez ostatnie kilkadziesiąt lat opracowano wiele algorytmów, które służą do rozpoznawania i identyfikacji twarzy. Część z nich jest oparta na sieciach neuronowych, a pozostała część korzysta z klasycznych metod aparatu matematycznego, np. algebry i analizy statystycznej.

W ramach niniejszego projektu powstało rozwiązanie oparte o metody klasyczne, oparte na algebrze, a w szczególności na wyliczaniu wartości własnych i wektorów własnych macierzy, powstałej z analizowanych zdjęć twarzy, celem zamodelowania analizowanych zdjęć w przestrzeni liniowej, rozpiętej przez wyliczone wektory własne. Wykorzystana metoda identyfikacji twarzy to metoda *eigenfaces*, która opiera się na metodzie analizy głównych składowych (ang. *Principal Component Analysis* – w skrócie: PCA).

1.2 Zastosowania identyfikacji twarzy

Istnieje wiele zastosowań identyfikacji twarzy. Część z nich dotyczy zastosowań w rozrywce, takiej jak np. gry czy identyfikacja twarzy na zdjęciach serwisów społecznościowych[1]. Poza zastosowaniami identyfikacji twarzy w rozrywce, istnieją zastosowania do poważniejszych celów, np. do wspomagania technik kryminalistycznych i szeroko pojętego bezpieczeństwa danych, np.: systemy kontroli dostępu, systemy bezpieczeństwa na lotniskach, policyjne bazy danych osób poszukiwanych (np. zaginionych, poszukiwanych listami gończymi i podejrzanych).

Identyfikacja twarzy w komputerowym wspomaganiu technik kryminalistycznych jest ważnym zagadnieniem dla wszystkich służb dbających o bezpieczeństwo, które posiadają bazy danych ze zdjęciami twarzy tysięcy osób, wśród których może znajdować się zdjęcie osoby identyfikowanej, np. podejrzanej, winnej, poszukiwanej lub zaginionej. Niniejsza dokumentacja jest opisem szczegółów implementacyjnych aplikacji służącej do identyfikacji dwuwymiarowych zdjęć twarzy.

1.3 Cel aplikacji

Celem aplikacji tworzonej w ramach niniejszego projektu jest zidentyfikowanie zadanej twarzy na podstawie zbioru zdjęć osób z lokalnej bazy danych. Zakładamy, że zdjęcia są znormalizowane, tzn. wszystkie zdjęcia są monochromatyczne, mają jednakowe rozmiary, twarze sfotografowana są frontalnie, tzn. *en face* i są wycentrowane.

Jeśli osoba, której twarz podano na wejście programu nie znajduje się w bazie danych, program powinien zwrócić kilka zdjęć najbardziej podobnych do zadanego wraz z oszacowaniem ich podobieństwa w skali

[1] W przeciwieństwie do rozpoznawania twarzy, identyfikacja twarzy nie jest jeszcze stosowana na szeroką skalę na największych serwisach społecznościowych.

od 0% do 100% lub w innej mierze odległości. W obu przypadkach zdjęcia znalezionych twarzy powinny być posortowane nierosnąco względem podobieństwa do zadanej twarzy.

W przeciwnym razie, tzn. jeśli zdjęcia osoby, której twarz została poddana identyfikacji, znajdują się w bazie danych, program powinien znaleźć tę twarz i przypisać wartość jej podobieństwa bliską 100% (wartość oczekiwana = 100%) oraz ewentualnie kilka innych, podobnych twarzy, z odpowiednio mniejszym podobieństwem wyrażonym w skali od 0% do 100% lub innej mierze odległości.

Pierwszorzędnym celem aplikacji *nie* jest szybkie działanie, co można uzasadnić tym, że algorytmy mające na celu zidentyfikowanie osoby, powinny przede wszystkim nie dopuścić do przeoczenia osoby, która znajduje się w bazie danych. Tak postawione wymaganie na ogół wymusza zastosowania algorytmów, które działają względnie długo.

1.4 Zmiany założeń względem dokumentacji wstępnej

Trudności, które wynikły w czasie projektu (patrz rozdziały: 4.7, A) wymusiły częściową zmianę wymagań postawionych przed aplikacją w czasie pisania dokumentacji wstępnej. W szczególności zrezygnowano z wyświetlania zgodności twarzy znalezionej z wyszukiwaną twarzą, wyrażonej w procentach. Zamiast tego wykorzystano miarę używaną przez użytą bibliotekę *OpenCV*, nazywaną odległością (*ang. distance*). Twarze znalezione są posortowane względem tej miary rosnąco. Miarę tę można wyświetlić najeżdżając myszą na wybrane, znalezione przez aplikację zdjęcie.

Ponadto zrezygnowano z funkcjonalności znanej z takich programów jak *Gimp* i *Photoshop* pod nazwą *crop*, która pozwalałaby przycinać zdjęcie do zaznaczonych przez użytkownika wymiarów. Wiązałoby się to bowiem z trudnym do wymuszenia na użytkowniku ograniczeniem w zaznaczeniu części zdjęcia, z zachowaniem proporcji całego zdjęcia, a następnie przeskalowania zdjęcia do rozmiaru pozostałych zdjęć. Rozmiar wszystkich zdjęć z założenia musi być identyczny. Alternatywnie możnaby było pozwolić użytkownikowi zaznaczać dowolny obszar zdjęcia, a potem skalować go do zadanego wymiaru pozostałych zdjęć, ale taka funkcjonalność nie była konieczna przy testach z bazą danych, w której zdjęcia twarzy były zunifikowane – tzn. mają takie same wymiary, a twarze są wyśrodkowane na zdjęciu. Ponadto taka funkcjonalność jest sporym wydatkiem pracy, który zamiast tego został przeznaczony na rozbudowę nieplanowanej funkcjonalności, jakim było np. okno ustawień.

Zmiany nie dotyczyły tylko ograniczenia funkcjonalności, ale również jej powiększenia w niektórych aspektach. Nieplanowanym bowiem było zapisywanie nauczonego modelu do pliku XML oraz okno ustawień algorytmu, które pojawiło się w ostatecznej wersji projektu. Pozwala ono zmienić parametry obliczeń: próg akceptowalnej odległości zdjęć (*ang. threshold*), maksymalną ilość wyświetlanych, znalezionych zdjęć oraz ilość komponentów *eigenfaces*[2] używanych w trakcie obliczeń.

Ponadto, rozszerzeniem względem wymagań postawionych przed projektem, jest możliwość dodawania zdjęć do lokalnej bazy danych, tzn. dodawanie zdjęć do zakładki `All faces`, ale nie do bazy *PostgreSQL*. Umożliwia to testowanie aplikacji na dowolnym zbiorze uczącym – wystarczy wyczyścić całą[3] bazę danych przyciskiem `Clear all`, a następnie dodać wybrane zdjęcia. Co ważne, opcja dodająca zdjęcia do bazy

[2] Wartość 0 (zero) jest specjalnie traktowana i oznacza automatyczne dopasowanie ilości komponentów *eigenfaces* przez *OpenCV*.

[3] Wybiórcze usuwanie pojedynczych zdjęć twarzy z bazy lokalnej i *PostgreSQL* nie zostało zaimplementowane.

lokalnej, działa w ten sposób, że pozwala ona wybrać wiele dodawanych zdjęć naraz – wszystkim tym zdjęciom zostanie nadany ten sam `personID`. Wynika z tego konieczność dodawania zbioru zdjęć tej samej osoby za jednym razem. W przeciwnym razie zdjęcia będą miały inny `personID`, a więc będą uważane przez algorytm za zdjęcia innej osoby.

2 Algorytm

Do rozpoznawania twarzy została zastosowana *metoda analizy głównych składowych* (ang. *Principal Component Analysis* – w skrócie: PCA), zaproponowana niezależnie przez angielskiego matematyka *Karla Pearson'a* (1901) oraz ekonoma i statystyka *Harolda Hotelling'a* (1933) [**pearson**].

W niniejszym dziale przedstawiono uzasadnienie wyboru algorytmu, opis metody PCA oraz *eigenfaces*, na podstawie publikacji autorów metody *eigenfaces* – prof. *Matthew Turk'a* i prof. *Alex'a Pentland'a*[**turk**].

2.1 Analiza głównych składowych

Jednym z głównych problemów z jakim mamy do czynienia w identyfikacji twarzy ze zdjęć, jest ich duży wymiar. Dwuwymiarowe, monochromatyczne zdjęcie zdjęcie o wymiarach $p \times q$, rozpina przestrzeń $m = p{\cdot}q$–wymiarową. Zdjęcie o wymiarach $100{\times}100$ pikseli, rozpina przestrzeń 10000-wymiarową. Istnieje potrzeba zmniejszenia tej przestrzeni kosztem możliwie małych różnic między oryginalnym zdjęciem, a zdjęciem reprezentowanym w „skompresowanej"przestrzeni liniowej. Okazuje się, że do celów identyfikacji twarzy, nie wszystkie wymiary/osie są jednakowo istotne. Będziemy chcieli zachować tylko te wymiary, dla których wariancja jest największa, tak, aby skutecznie rozróżniać zdjęcia twarzy, korzystając z możliwie najmniejszej ilości wymiarów[**www:opencv**].

Główna idea metody *eigenfaces*, to przedstawienie zdjęcia jako liniowa kombinacja obrazów bazowych (patrz równanie 2.1.9), zapisywanych w postaci wektorów, nazywanych również *eigenfaces*. Dzięki temu obraz możemy przestawić w zwartej postaci wektora $[\omega_1, \ldots, \omega_M]$, o długości znacznie mniejszej od wymiaru obrazu N^2.

Załóżmy, że mamy zbiór zdjęć twarzy treningowych, pobranych z bazy danych, w postaci M wektorów $\mathbf{\Gamma_1}, \mathbf{\Gamma_2}, \ldots, \mathbf{\Gamma}_M$, z których każdy ma długość $N \cdot N$, gdzie N to długość i wysokość zdjęcia[4]. Średnia twarz tego zbioru twarzy testowych jest zdefiniowana jako następujący wektor $\mathbf{\Psi}$ o długości $N \cdot N$:

$$\mathbf{\Psi} = \frac{1}{M} \cdot \sum_{n=1}^{M} \mathbf{\Gamma_n} \tag{2.1.1}$$

i-ta twarz treningowa $\mathbf{\Gamma_i}$ różni się od twarzy średniej $\mathbf{\Psi}$ o pewien wektor $\mathbf{\Phi_i}$:

$$\mathbf{\Phi_i} = \mathbf{\Gamma_i} - \mathbf{\Psi} \tag{2.1.2}$$

Zbiór wektorów $\{\mathbf{\Phi_1}, \mathbf{\Phi_2}, \ldots, \mathbf{\Phi}_M\}$ poddajemy metodzie *analizy głównych składowych*, dzięki której

[4]Dla uproszczenia oznaczeń w opisie metody, zakładamy, że wysokość zdjęcia jest równa szerokości zdjęcia.

otrzymamy zbiór M ortonormalnych[5] wektorów $\{\boldsymbol{u_1}, \boldsymbol{u_2}, \ldots, \boldsymbol{u_M}\}$, które najlepiej obrazują rozkład wartości pikseli. k-ty wektor $\boldsymbol{u_k}$ jest wybierany tak, aby:

$$\boldsymbol{\lambda_k} = \frac{1}{M} \cdot \sum_{n=1}^{M} (\boldsymbol{u_k^T \Phi_n})^2 \tag{2.1.3}$$

był maksymalny gdzie[6]:

$$\boldsymbol{u_l^T u_k} = \delta_{lk} = \begin{cases} 1, & \text{gdy } l = k \\ 0, & \text{gdy } l \neq k \end{cases} \tag{2.1.4}$$

Równanie 2.1.4 mówi, że każde dwa, różne wektory $\boldsymbol{u_k}$, $\boldsymbol{u_l}$, gdzie $k \neq l$, są ortogonalne względem siebie. Wektory $\boldsymbol{u_k}$ i skalary λ_k to odpowiednio wektory własne i wartości własne następującej macierzy kowariancji:

$$C = \frac{1}{M} \cdot \sum_{n=1}^{M} \boldsymbol{\Phi_n \Phi_n^T} = AA^T \tag{2.1.5}$$

gdzie:

$$A = [\boldsymbol{\Phi_1 \Phi_2} \ldots \boldsymbol{\Phi_M}] \tag{2.1.6}$$

Rozmiar macierzy C jest bardzo duży – wynosi $N^2 \times N^2$. Liczenie N^2 wartości własnych i wektorów własnych byłoby bardzo czasochłonne dla typowych wymiarów zdjęć.

Jeśli ilość zdjęć w bazie danych M jest mniejsza od wymiaru przestrzeni N^2, tzn. $M < N^2$, to otrzymamy tylko $M - 1$ znaczących wektorów własnych zamiast N^2. Pozostałe wektory własne będą miały wartości własne równe 0. Na szczęście możemy znaleźć N^2-wymiarowe wektory własne najpierw licząc wektory własne mniejszej macierzy, tzn. macierzy $M \times M$. Dla 16 zdjęć o wymiarach 128×128, oznacza to, że zamiast liczyć wektory własne macierzy o wymiarach[7] $p \times q$, czyli, w rozpatrywanym wypadku 16384×16384, możemy policzyć wektory własne macierzy A^TA, która ma wymiary 16×16, a następnie zastosować odpowiednią kombinację liniową wektora $\boldsymbol{\Phi_i}$. Rozważmy zatem wektor własny $\boldsymbol{v_i}$ macierzy A^TA, taki, że:

$$A^TA\boldsymbol{v_i} = \mu_i \boldsymbol{v_i} \tag{2.1.7}$$

Po przemnożeniu obu stron lewostronnie przez A, otrzymujemy, że:

$$\underbrace{A\overbrace{A^TA}^{L}}_{C}\underbrace{\boldsymbol{v_i}}_{\boldsymbol{u_i}} = \mu_i \underbrace{A\boldsymbol{v_i}}_{\boldsymbol{u_i}} \tag{2.1.8}$$

skąd widać, że $A\boldsymbol{v_i}$ i μ_i są odpowiednio wektorami własnymi i wartościami własnymi macierzy kowariancji $C = AA^T$. Warto zauważyć, że wartości własne μ_i macierzy A^TA i wartości własne λ_i macierzy AA^T są równe, tzn. $\mu_i = \lambda_i$.

Skonstruujmy macierz $L = A^TA$ o wymiarach $M \times M$, gdzie $L_{mn} = \boldsymbol{\Phi_m^T \Phi_n}$ i znajdźmy M wektorów własnych $\boldsymbol{v_l}$ macierzy L. Wektory te wyznaczają kombinację liniową M treningowych zdjęć twarzy, która

[5] Ortonormalnych, czyli jednocześnie ortogonalnych i normalnych.
[6] δ_{lk} jest nazywana deltą lub symbolem *Kroneckera*.
[7] $128 \cdot 128 = 16384$.

w sumie tworzy wektory $\boldsymbol{u_l}$, nazywane *eigenfaces*:

$$\boldsymbol{u_l} = \sum_{k=1}^{M} \boldsymbol{v_{lk}} \boldsymbol{\Phi_k} \quad \text{gdzie:} \quad l = 1, \ldots, M \tag{2.1.9}$$

Dzięki powyższej analizie złożoność obliczeń istotnie zmalała – z rzędu liczby pikseli w zdjęciu N^2, do rzędu zdjęć w zbiorze treningowym M. W praktyce zbiór treningowy jest relatywnie mały (tzn. $M \ll N^2$) i obliczenia są względnie szybkie. W praktyce można wybrać M', takie, że $M' < M$, ponieważ wystarczy nam kombinacja liniowa, tworząca przybliżoną twarz, a nie idealnie odwzorowaną. W takim przypadku wektory *eigenfaces* rozpinają podprzestrzeń M'-wymiarową pierwotnej N^2-wymiarowej przestrzeni zdjęć. Wybór M' wektorów własnych spośród wszystkich wektorów własnych macierzy L, odbywa się na zasadzie wzięcia tych z nich, które mają największe odpowiadające wartości własne. W publikacji *M.Turka* i *A.Pentland'a* zostały z powodzeniem testowane przykłady, dla których $M = 16$, a $M' = 7$[**turk**].

Chcąc zidentyfikować osobę widoczną na zadanym zdjęciu twarzy, reprezentowanym przez wektor $\boldsymbol{\Gamma}$ o długości N^2, należy przetransformować ten wektor na „przestrzeń twarzy"(ang. *face space*), rozpiętą przez wektory $\boldsymbol{u_1}, \boldsymbol{u_2}, \ldots, \boldsymbol{u_{M'}}$ licząc współczynniki (wagi) ω_k kombinacji liniowej:

$$\omega_k = \boldsymbol{u_k^T}(\boldsymbol{\Gamma} - \boldsymbol{\Psi}) \quad \text{gdzie:} \quad k = 1, 2, \ldots, M' \tag{2.1.10}$$

Wagi ω_k tworzą wektor $\Omega^T = [\omega_1, \omega_2, \ldots, \omega_{M'}]$, który opisuje wkład każdego z wektorów *eigenface* $\boldsymbol{u_k}$ w reprezentację zadanego zdjęcia reprezentowanego w „przestrzeni twarzy"jako:

$$\begin{aligned} \sum_{k=1}^{M'} \omega_k \boldsymbol{u_k} &= [\boldsymbol{u_1^T}, \boldsymbol{u_2^T}, \ldots, \boldsymbol{u_{M'}^T}]^T [\omega_1, \omega_2, \ldots, \omega_{M'}]^T \\ &= [\boldsymbol{u_1^T}, \boldsymbol{u_2^T}, \ldots, \boldsymbol{u_{M'}^T}]^T \boldsymbol{\Omega} \end{aligned} \tag{2.1.11}$$

Tak otrzymany wektor może być użyty do porównania z wektorami dostępnymi w bazie danych np. przez obliczenie odległości dwóch wektorów miarą Euklidesową.

2.2 Uzasadnienie wyboru algorytmu

Algorytm *eigenfaces* został wybrany z kilku powodów:

1. Po zakończeniu etapu nauki modelu, model może być wykorzystywany w rozpoznawaniu twarzy w czasie rzeczywistym;
2. Jest uznawany w środowisku za dobry algorytm identyfikacji twarzy, dzięki uzyskiwanym, zadowalającym, wynikom;
3. Jest stosunkowo wiele publikacji na temat *eigenfaces*;
4. Nie wymaga dużej ilości pamięci i dużej mocy obliczeniowej;
5. W przeciwieństwie do sztucznych sieci neuronowych, algorytm korzysta z stosunkowo łatwej idei, dającej się wyrazić w krokach, z których każdy jest operacją realizowaną za pomocą prostej algebry;
6. Proces nauki jest w pełni automatyczny;

7. Jest zaimplementowany w repozytorium *OpenCV Contrib*.

Poza wyżej wymienionymi zaletami, algorytm ten ma również kilka istotnych wad, z których chyba największą jest wrażliwość na zmianę oświetlenia, zmianę skali zdjęcia i translacje. Wady te jednak nie były na tyle duże, aby przyćmiły mnogość wyżej wymienionych zalet – w szczególności chyba największej z zalet – możliwości identyfikacji w czasie rzeczywistym po przeprowadzonej fazie nauki.

3 Maszyna wirtualna

W celu uniknięcia potencjalnych problemów z zależnościami aplikacji od konkretnej konfiguracji systemu operacyjnego, w szczególności od zainstalowanych bibliotek i konfiguracji bazy danych, w ramach projektu, skonfigurowano maszynę wirtualną *VirtualBox*, na której dostarczono działającą aplikację. Na maszynie wirtualnej został zainstalowany 64-bitowy system operacyjny *Linux Debian* w najnowszej dostępnej wersji, tj. *Debian (Stretch)*, korzystający z lekkiego środowiska graficznego `Xfce`.

Maszyna wirtualna zajmuje stosunkowo niewiele miejsca, bo około `11GB`, dlatego, zrezygnowano ze zbędnych w tym przypadku, wyrafinowanych schematów partycjonowania przestrzeni dyskowej i utworzono jedną partycję systemową `ext4`.

W niniejszym rozdziale krótko opisano najważniejsze zmiany jakich dokonano względem *czystego*, tj. niezmodyfikowanego, systemu operacyjnego, pobranego z oficjalnej strony *Debiana*.

3.1 Konto administratora

Na potrzeby konfiguracji systemu stworzono konto administratora, tj. *root'a*.

Login:	`root`
Hasło:	`r00t-MiNI&PW-2015/2016`

3.2 Konto użytkownika systemu

Poza kontem administratora, utworzono konto zwykłego użytkownika o nazwie `kwtk`.

Login:	`kwtk`
Hasło:	`kwtk-MiNI&PW-2015/2016`

3.3 Konfiguracja systemu

Projekt jest napisany w Javie w wersji *8*[8], więc z natury powinien być przenośny pomiędzy wszystkimi systemami operacyjnymi, na których można zainstalować maszynę wirtualną Javy. W rzeczywistości nie jest to takie proste, ponieważ w projekcie wykorzystano biblioteki *OpenCV*, które wykorzystują mechanizm *Java-Native-Interface*. Biblioteka *OpenCV* została napisana w C++ w ten sposób, aby dało się ją skompilować dla różnych systemów operacyjnych, w tym m.in. dla *Linux'a* i *Windowsa*. W ramach

[8]Wykorzystano m.in. bibliotekę `java.time` i wyrażenia *lambda*, które nie są dostępne w poprzednich wersjach Javy. Domyślnie na *Debianie* jest jeszcze stara wersja Java 7, dlatego Javę 8 została doinstalowana.

projektu została skompilowana ta sama wersja biblioteki dla 64-bitowych systemów *Windows* i *Linux* (odpowiednio biblioteka: `opencv_java310.dll` i `libopencv_java310.so`) z drobną modyfikacją w kodzie C++ (patrz: 4.7). W trakcie pierwszych testów aplikacji przetestowano ją na obu tych systemach, jednak po dokonaniu zmian w kodzie, o których mowa w rozdziale 4.7, zrezygnowano z dalszego wspierania systemu *Windows*, ponieważ wiązałoby się to z ponowną, czasochłonną kompilacją lekko zmodyfikowanej biblioteki *OpenCV*.

Oprogramowanie wykorzystane w ramach projektu jest w całości otwartoźródłowe i *wolne*[9]. Większość zależności aplikacji jest zawarta w projekcie – np. moduł *OpenCV* i biblioteka *Hibernate*, służąca za *ORM* przy dostępie do bazy danych. Jedynym modułem, który nie można zawrzeć w postaci *Eclipse'owego* projektu jest baza danych *PostgreSQL*[10], przechowująca zdjęcia twarzy identyfikowanych osób. Wymagała ona osobnej konfiguracji, polegającej m.in. na: stworzeniu użytkownika i hasła dostępu do bazy danych, stworzeniu tabeli `faces` w bazie danych, wypełnienie jej zdjęciami twarzy identyfikowanych osób i przyznanie dostępu użytkownikowi do tabeli `faces`.

W celu łatwego przenoszenia bazy danych z maszyny *developerskiej* na maszynę wirtualną, wykorzystano *dump* bazy danych, który w przypadku eksportu bazy danych sprowadza się do wykonania komendy: `pg_dump template1 > out.dump`, a w przypadku importu bazy danych: `psql template1 < out.dump`, gdzie `template1` to nazwa bazy danych, a `out.dump` to ścieżka pliku z zrzutem bazy danych.

W przypadku chęci wygodnego konfigurowania, edytowania i *debugowania* kodu źródłowego projektu, można zainstalować dodatkowe pakiety oprogramowania, takie jak: `eclipse`, `git`, `mc`, `vim`. Każdy z tych pakietów można pobrać i zainstalować ze standardowego repozytorium *Debiana* za pomocą komendy:

`apt-get install` ***`nazwa-pakietu`***`.`

Cała opisana wyżej konfiguracja systemu *Debian* – zarówno ta obowiązkowa, jak i opcjonalna – została wykonana w ramach przygotowania maszyny wirtualnej na finalne oddanie projektu, więc aplikacja działa *out of the box* i nie wymaga dodatkowej konfiguracji.

3.4 Baza danych

Aplikacja łączy się z lokalną[11] bazą danych na, standardowym dla *PostgreSQL*, porcie 5432. Nic nie stoi na przeszkodzie, aby w konfiguracji „produkcyjnej", przenieść bazę danych na oddzielny serwer. Zmianie uległby tylko adres IP w konfiguracji *Hibernate'a* w pliku `hibernate.cfg.xml`. Dane logowania używane przez aplikację w celu połączenia się z lokalną bazą danych *PostgreSQL*:

Login:	`eigenuser`
Hasło:	`eigenfaces`
Host:	`localhost`
Port:	`5432`

Tabela `faces` powstała za pomocą następującej formuły SQL:

[9] *Wolne* w znaczeniu użytej licencji, pozwalającej na wprowadzanie zmian i nieodpłatne wykorzystywanie: źródeł, bibliotek, baz danych itd.

[10] *PostgreSQL* można zainstalować, instalując pakiet `postgres`, dostępny w standardowym repozytorium większości znanych dystrybucji *Linux'a*. Na *Debianie* i *Ubuntu* do zainstalowania wystarczy: `apt-get install postgres`.

[11] Tzn. z adresem IPv4: `127.0.0.1` (`localhost`).

```
CREATE TABLE faces (
  id SERIAL NOT NULL,
  person_id INT NOT NULL,
  image_id INT NOT NULL,
  image BYTEA NOT NULL,
  filename VARCHAR UNIQUE NOT NULL,
  filetype CHAR(16) NOT NULL,
  timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP NOT NULL,
  PRIMARY KEY (id)
);
```

Wypełnienie tabeli zostało wykonane automatycznie w Javie, po wywołaniu funkcji `addAllFacesFromPredefinedCsv()` z pliku `DatabaseConnectionManager.java`. Funkcja ta czyta plik CSV, który został wygenerowany prostym skryptem `create_csv.py`, napisanym w *Pythonie*, na podstawie folderu `YaleFacedatabaseA`, który zawiera zdjęcia twarzy. Wygenerowany plik CSV zawiera:

- ścieżkę względną zdjęcia,
- identyfikator osoby, której twarz przedstawia zdjęcie,
- identyfikator zdjęcia danej osoby[12],

Zdjęcia, które znalazły się w bazie danych można pobrać ze strony internetowej *Computer Vision Laboratory in the Computer Science and Engineering Department* Uniwersytetu Kalifornijskiego w San Diego:

`http://vision.ucsd.edu/datasets/yale_face_dataset_original/yalefaces.zip`

Notka licencyjna z pliku `Readme.txt`, rozpakowanego z wyżej wymienionego archiwum `yalefaces.zip`:

> You are free to use the Yale Face Database for research purposes. If experimental results are obtained that use images from within the database, all publications of these results should acknowledge the use of the "Yale Face Databaseąnd cite
>
> **P. Belhumeur, J. Hespanha, D. Kriegman, Eigenfaces vs. Fisherfaces: Recognition Using Class Specific Linear Projection, IEEE Transactions on Pattern Analysis and Machine Intelligence, July 1997, pp. 711-720.**
>
> Without permission from Yale, images from within the database cannot be incorporated into a larger database which is then publicly distributed.

4 Aplikacja

Aplikację zaimplementowano w języku Java w wersji 8, korzystając ze środowiska graficznego *Swing*.

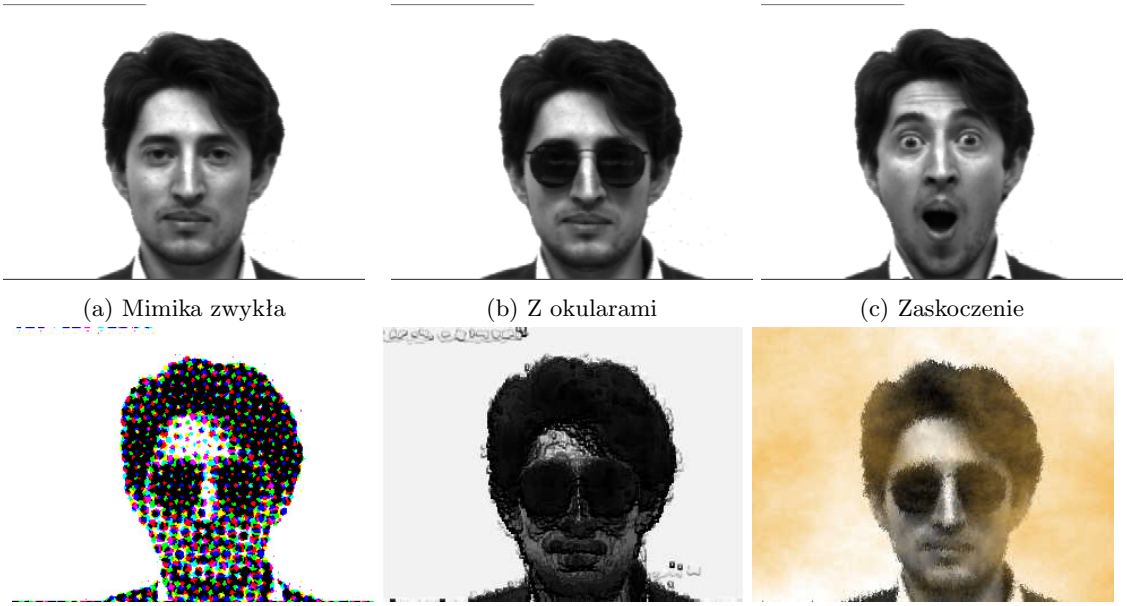
[12] Każda z 15 osób ma 11 zdjęć swojej twarzy, więc identyfikatory zdjęć danej osoby, to liczby całkowite z zakresu od 1 do 11.

4.1 Dane wejściowe

Za główne dane wejściowe aplikacji uważamy:

1. znormalizowane[13] zdjęcie identyfikowanej osoby,
2. baza danych znanych twarzy[14].

Przykładowe zdjęcia twarzy osób, które znalazły się w bazie danych, wraz ze zdjęciami testowymi, które są zmodyfikowanymi twarzami osób z bazy danych, przestawiono na rysunku 1. Twarze testowe, które stworzono modyfikując zdjęcia oryginalne w programie graficznym *Gimp*, można znaleźć w katalogu projektu `faces/Tests`, „obok" katalogu ze zdjęciami oryginalnymi, tj. `faces/YaleFacedatabaseA`.



(a) Mimika zwykła (b) Z okularami (c) Zaskoczenie

(d) Zdjęcie testowe #1 (e) Zdjęcie testowe #2 (f) Zdjęcie testowe #3

Rysunek 1: Przykładowe zdjęcia twarzy osoby oznaczonej identyfikatorem `personID = 3`

4.2 Dane wyjściowe

Za dane wejściowe aplikacji uważamy *niewielki* podzbiór zdjęć twarzy z bazy danych, podobnych do identyfikowanej twarzy, wraz z liczbą z zakresu 0% – 100%, lub inną miarą odległości, będącą odzwierciedleniem podobieństwa twarzy identyfikowanej i zwróconej jako wynik działania programu.

4.3 Wykorzystane biblioteki

Najważniejszą biblioteką wykorzystaną w ramach projektu jest biblioteka *OpenCV* i implementująca *PCA* i *eigenfaces*, biblioteka *OpenCV Contrib Faces*. Obie biblioteki zostały skompilowane z oficjalnych źródeł, tzn. źródła *OpenCV* zostały pobrane z:

[13] Przez znormalizowane zdjęcia rozumiemy zdjęcia o takich samych wymiarach z twarzą wyśrodkowaną na zdjęciu.

[14] W trakcie pisania niniejszej dokumentacji, w bazie danych znajduje się 165 zdjęć 15 osób – po 11 zdjęć twarzy każdej z osób – każde z 11 zdjęć z inną mimiką twarzy.

`https://github.com/Itseez/opencv`

`git@github.com:Itseez/opencv.git`

a *OpenCV Contrib* z:

`https://github.com/Itseez/opencv_contrib`

`git@github.com:Itseez/opencv_contrib.git`

Kompilacje biblioteki *OpenCV* i modułów *OpenCV Contrib* zostały przeprowadzone na najnowszych dostępnych wersjach tych bibliotek, dostępnych w dniu kompilacji, tj. 1 VI 2016, pobranych z oficjalnych repozytoriów *GitHub*, tzn. *OpenCV* został skompilowany z *commit'a* nr

`e1ba4399e8b4a9c3eb844992ab9e64aeae2388a2`

z modułami *OpenCV Contrib* z *commitu* nr:

`ba1d3ef99cf5e67241c5a31dbd9c344d12a2f7c5`

Niepełna lista wykorzystanych w projekcie bibliotek:

1. *OpenCV 3.1* + moduły z repozytorium *OpenCV Contrib* – implementacja m.in. *eigenfaces*,
2. *Hibernate – ORM* dla *PostgreSQL*,
3. standardowe biblioteki Javy (m.in. do logowania zdarzeń – patrz 4.6),
4. biblioteki zależne od wyżej wymienionych bibliotek.

4.4 Uruchomienie aplikacji

Aby uruchomić aplikację w trybie `debug`, należy włączyć *VirtualBox*, uruchomić maszynę wirtualną z *Debianem*, zalogować się i wybrać z menu: `Programy > Programowanie > Eclipse > Run > Debug`. W przypadku gdyby *Eclipse* pytał, który plik źródłowy uruchomić, należy wskazać `MainWindow.java`.

4.5 Autoryzacja

Po uruchomieniu aplikacji, przed wyświetleniem się okna głównego, zostaje przeprowadzona prosta autoryzacja przez wymóg podania poprawnej nazwy użytkownika oraz hasła.

Login:	`root`
Hasło:	`toor`

4.6 Logowanie zdarzeń

Aplikacja loguje różne zdarzenia podczas działania programu, np.:

- błędy krytyczne działania programu,
- ostrzeżenia – np. przy niepowodzeniu połączenia z bazą danych,
- działania użytkownika – np. wyszukiwanie w bazie danych,

- wyniki wyszukiwań w bazie danych,
- daty zalogowań i prób zalogowań do aplikacji.

Logowanie zostało zrealizowane z wykorzystaniem bardzo elastycznej paczki `java.util.logging`, która pozwala m.in. na filtrowanie logowanych komunikatów, konfigurowanie formatu logowanych wiadomości, zapisywanie zarówno do pliku jak i na standardowe wyjścia (nie tylko `stderr`) i wiele innych.

Logi z działania aplikacji zostają zapisywane do katalogu projektu `/logs` w formacie:

`yyyy-MM-dd_HH-mm-ss.log`

gdzie:

- `yyyy` – rok,
- `MM` – miesiąc,
- `dd` – dzień miesiąca,
- `HH` – godzina,
- `mm` – minuty,
- `ss` – sekundy.

Godzina w nazwie pliku logu jest ustalana względem czasu systemowego (a nie względem `UTC+00`). Logi nie są kasowane po zakończeniu programu.

4.7 Napotkane trudności

Zdecydowanie największą trudnością konfiguracji projektu, która pochłonęła sporą część całego projektu, była kompilacja modułu *OpenCV Faces* (i jego zależności), który jest implementacją algorytmu *eigenfaces*. Szczególnie dużo czasu zajęło poprawne jej skompilowanie na systemie operacyjnym *Windows 10*. Kompilacja na systemie *Linux Debian* była również czasochłonna, ale nie w takim stopniu jak na *Windows*.

Główną przyczyną trudności konfiguracji projektu był fakt, że moduł *OpenCV Faces* jest wersją rozwojową, która nie jest dołączona do standardowej instalacji *OpenCV*. Ponadto *OpenCV* i wszystkie jego moduły, w tym moduł *Faces*, jest napisany w C++ i aby włączyć *wrapper*[15] do Javy, należy przeprowadzić kompilację *OpenCV* i *OpenCV Contrib* samodzielnie, co nie byłoby dużym problemem gdyby nie to, że po drodze należy zmodyfikować kilka nieudokumentowanych miejsc w kodzie modułu *Faces*, ponieważ oficjalnie, tylko moduły z głównego wydania *OpenCV* mają wsparcie *wrapperem* Javy.

Inną trudnością związaną również z *wrapperem OpenCV Faces* do Javy, jest to, że nie wszystkie funkcje dostępne w interfejsie C++ *OpenCV Faces* są dostępne w jego *wraperze* w Javie. Jedną z istotnych funkcji niedostępnych w Javie jest przeciążona[16] funkcja `predict`, która w Javie jest dostępna tylko w jednej wersji, w której zwracane są:

[15] Przez *wrapper* rozumie się wygenerowany automatycznie, na podstawie kodu źródłowego napisanego w C++, plik *jar*, który jest interfejsem implementacji *OpenCV*, napisanej w C++ i skompilowanej do postaci binarnej **.dll* i **.so*, odpowiednio na *Windowsie* i *Linuksie*. Mechanizm ten nosi nazwę *Java Native Interface*.

[16] *Przeciążona funkcja* w sensie jednej nazwy funkcji z różnymi zestawami argumentów.

1. jeden identyfikator osoby rozpoznanej na zadanym zdjęciu twarzy,
2. miara odległości zdjęcia rozpoznawanego i tego, który wygenerował algorytm *eigenfaces*.

Taka funkcja okazała się niewystarczająca, ponieważ w projekcie potrzebowano znać nie tylko najbardziej pasującego człowieka do zadanej twarzy wraz z odległością twarzy wyrażoną w pewnej mierze przyjętej przez algorytm, ale również odległości innych, mniej pasujących twarzy. Problem ten został rozwiązany dzięki uprzejmości jednego z *developerów* modułu *OpenCV Faces*, tj. dzięki *Maksimowi Shabuninowi*, który po mailowym kontakcie, zaproponował rozwiązanie w komentarzu do pytania zamieszczonego na forum `answers.opencv.org`:

`http://answers.opencv.org/question/95181/opencv-contrib-face-module-with-java-wrapper-returning-multiple-prediction/`

Mniejszą trudnością, a bardziej uciążliwością, był fakt, że system *Windows* nie jest domyślnie przystosowany dla pracy programisty, co wiązało się z koniecznością ręcznego konfigurowania co najmniej kilku zmiennych środowiskowych i instalowania wielu zależności[17]. Niektóre z nich to: *git*, *CMake*, *Java 8 JDK*, *Python*, *ant*, *MinGW/Microsoft Visual Studio*, *Eclipse* i wiele innych.

Wyżej wymienione problemy były oczywiście jednorazowe, tzn. użytkownik końcowy nie musi się nimi przejmować, ponieważ wszystkie zależności, które mogły zostać ujęte w ramach projektu, zostały tam zawarte. Te zależności które nie mogły być zawarte w ramach projektu, lub były bardzo uciążliwe do ustawienia w ramach projektu, zostały ustawione w systemie dostarczonym na maszynie wirtualnej.

4.8 Instrukcja obsługi

Obsługa programu jest dość intuicyjna, wspierana *tooltip'ami*, które się pojawiają po najechaniu kursorem nad przyciski menu. Interfejs graficzny programu składa się z menu i dwóch kart: `All faces` i `Found faces`. Każda z kart jest podzielona pionowo na dwie części: lewą i prawą.

Po lewej stronie karty `All faces` mamy pogląd na wszystkie zdjęcia, jakie zostały wczytane z bazy danych *PostgreSQL*. Po kliknięciu na zdjęcie, po prawej stronie karty, mamy możliwość przybliżania i oddalania zdjęcia w wybranym obszarze oraz przywrócenia rozmiaru oryginalnego.

Po wybraniu z menu zdjęcia poddawanego identyfikacji, zatwierdzeniu rozpoczęcia przeszukiwania bazy danych i zakończeniu obliczeń, w karcie `Found faces` pojawiają się zdjęcia, które są najbardziej zbliżone do identyfikowanej twarzy.

Bardziej szczegółowa instrukcja obsługi aplikacji została ujęta w osobnym dokumencie.

[17] Nie dość, że na system *Windows* należało zainstalować większą ilość brakujących zależności niż na *Debianie*, to w *Windows* nie ma centralnego repozytorium pakietów, dlatego wszystkie zależności były pobierane ręcznie, tj. wchodząc na strony `www` twórców poszczególnych zależności.

Załącznik

A Podział prac

Cała praca nad projektem była prowadzona w ramach prowadzonego repozytorium kodu `git` – jest tam zarówno kod źródłowy aplikacji jak i kod LaTeX dokumentacji. Podgląd historii podziału prac i postępów prac w kolejnych tygodniach projektu można podejrzeć na stronie projektu:

`https://github.com/pbeza/komputerowe-wspomaganie-technik-kryminalistycznych/graphs/contributors`

logując się uprzednio na stronie `github.com` na tymczasowe konto:

Login:	`tutoor`
Hasło:	`--dD3AcDc$l@sH+^xORCrYpTOGr@p_hY--`

A.1 Podział pracy pisemnej

Poniższa tabela przestawia wykaz dokumentów opracowanych przez poszczególnych członków zespołu. W nawiasach podano szacowany, procentowy czas/zaangażowanie poszczególnych osób w pracę nad dokumentacją w LaTeX.

	Patryk Bęza (90%)	Marek Kozak (10%)	Krzysztof Małaśnicki (0%)
Opracowane dokumenty	Dokumentacja techniczna, czyli cały niniejszy dokument (rozdziały: 1, 2, 3, 4)	Instrukcja użytkownika (manual)	–

A.2 Podział implementacji

Poniżej wyszczególniono podział prac implementacyjnych w ramach projektu. W nawiasach podano szacowany, procentowy czas/zaangażowanie poszczególnych osób w implementację i konfigurację projektu.

- Patryk Bęza (95%):
    1. Implementacja całego programu i konfiguracja środowisk towarzyszących. W szczególności: implementacja wszystkich funkcjonalności aplikacji poza implementacją pierwszej wersji paska postępu. Szczegółowy wykaz wykonanych zadań:
    2. Kompilacja biblioteki *OpenCV Contrib* razem z *wrapperem* dla Javy na systemy operacyjne *Windows* i *Linux* (patrz rozdział: 4.7).
    3. Obsługa logiki wywołań funkcji obsługujących algorytm *eigenfaces* z biblioteki *OpenCV Contib* po stronie kodu w Javie.
    4. Kontakt z *developerem* w sprawie pomocy w dokonaniu drobnych zmian w bibliotece *OpenCV Faces* w C++ i dokonanie tych zmian (patrz rozdział: 4.7).

5. Implementacja interfejsu graficznego poza pierwszą wersją paska postępu.
6. Implementacja skryptu generującego plik CSV z listą zdjęć z bazy danych w *Pythonie*.
7. Obsługa bazy danych za pomocą *Hibernate* w Javie, w szczególności przygotowanie funkcji do automatycznego załadowania 165 zdjęć twarzy do bazy danych *PostgreSQL*.
8. Konfiguracja bazy danych *PostgreSQL*, w szczególności utworzenie tabeli ze zdjęciami twarzy i jej wypełnienie rekordami ze zdjęciami twarzy.
9. Implementacja logowania zdarzeń w programie do logów.
10. Implementacja logowania do aplikacji (patrz rozdział: 4.5).
11. Implementacja możliwości przybliżania i oddalania zdjęcia.
12. Implementacja wyświetlania szczegółów dotyczących aktualnie wybranego zdjęcia.
13. Implementacja okna ustawień programu.
14. Przygotowanie maszyny wirtualnej *VirtualBox*.

- Krzysztof Małaśnicki (5%):
  1. Dodanie pierwszej wersji *progress bar'a* i przeniesienie kilku gotowych elementów z karty *All faces* do *Found faces*.
- Marek Kozak (0%):
  1. Brak.

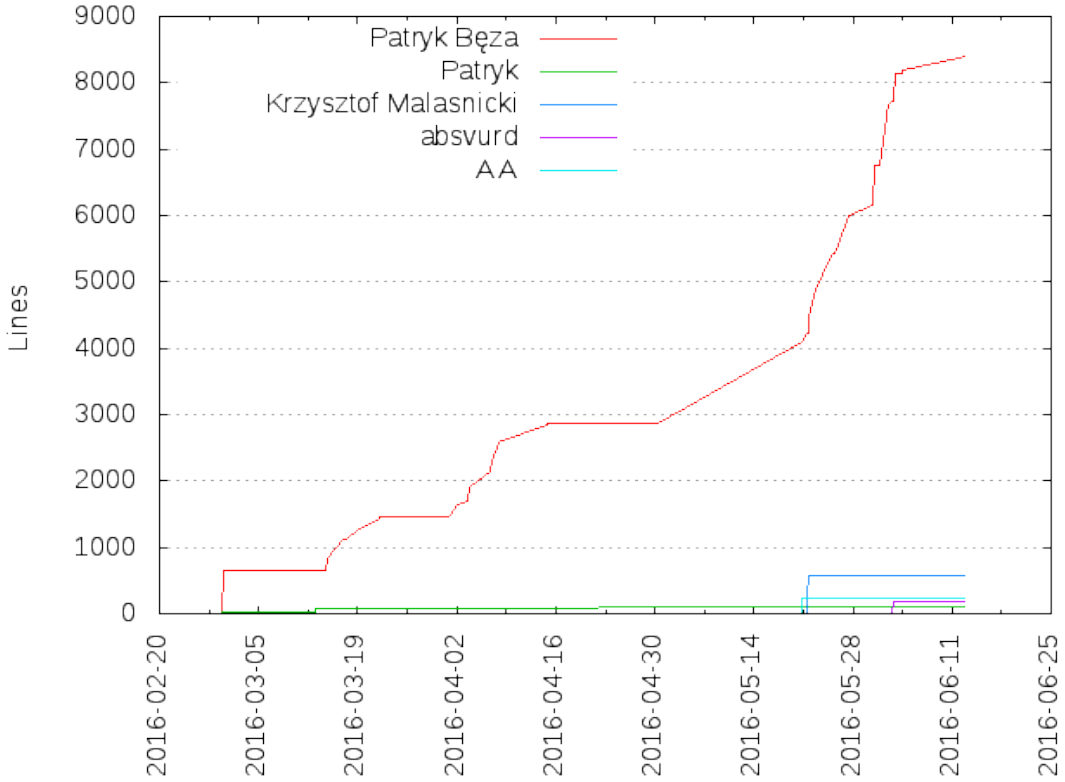
4.3 Statystyki `git stats`

Na rysunku 2 przedstawiono sumaryczną zależność liczby linii kodu i dokumentacji, wgranych do repozytorium `git` projektu, od czasu i autora. Statystyki zostały wygenerowane w pełni automatycznie w dniu 12.06.2016, przez program `git stats`, wydany na licencji `GPLv2/GPLv3` [**www:git-stats**].

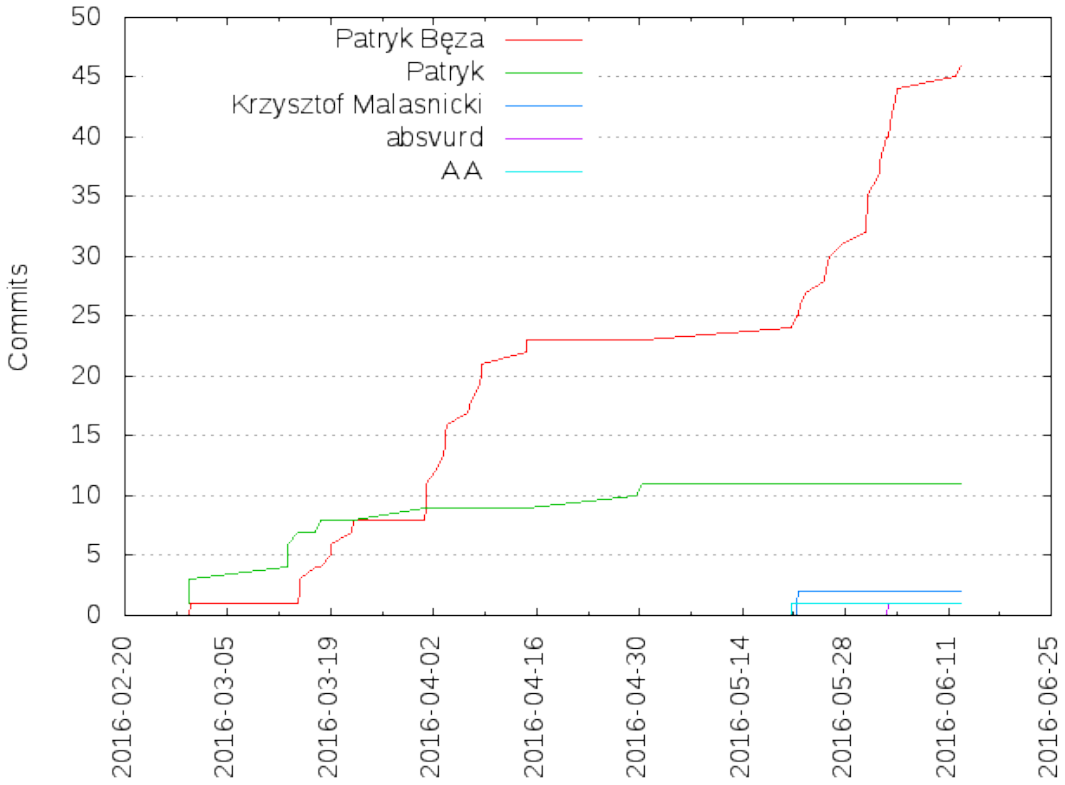
Na rysunku 2 zostało przedstawionych 5 krzywych reprezentujących aktywność poszczególnych użytkowników. Ilość krzywych nie równa się ilości autorów niniejszego projektu, co wynika z tego, że użytkownicy zmienili swoje wyświetlane nazwy użytkownika, po dokonaniu pierwszego *commitu*.

Imię i nazwisko autora	Nazwa użytkownika `git`
Patryk Bęza	`Patryk Bęza`
	`Patryk`
Krzysztof Małaśnicki	`Krzysztof Malasnicki`
	`A A`
Marek Kozak	`absvurd`

Tablica 1: Przyporządkowanie nazw użytkowników do imion i nazwisk autorów



Rysunek 2: Statystyki `git stats` ilości linii wgranych do repozytorium kodu



Rysunek 3: Statystyki `git stats` ilości *commitów* w repozytorium kodu

Author	Commits (%)	+ lines	- lines	First commit	Last commit	Age	Active days	# by commits
Patryk Bęza	46 (75.41%)	8403	4273	2016-02-29	2016-06-12	104 days, 17:08:20	25	1
Patryk	11 (18.03%)	105	30	2016-02-28	2016-04-30	61 days, 17:07:04	7	2
Krzysztof Malasnicki	2 (3.28%)	577	387	2016-05-21	2016-05-21	3:49:31	1	3
absvurd	1 (1.64%)	182	0	2016-06-02	2016-06-02	0:00:00	1	4
A A	1 (1.64%)	223	234	2016-05-20	2016-05-20	0:00:00	1	5

Rysunek 4: Tabela `git stats` ze statystykami dotyczącymi aktywności *commitów*