گزارش پروژه‌ی ریاضیات مهندسی

سید نوید کرمی‌نژاد

۹۴۳۱۰۷۰

بهمن ۱۳۹۶

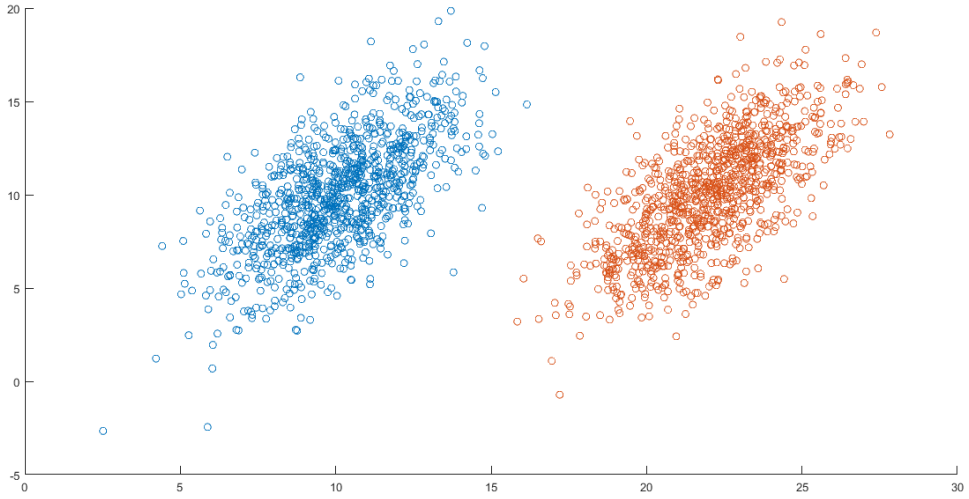
- **کد تمامی قسمت‌ها در نرم‌افزار متلب پیاده‌سازی شده‌است.**

قسمت اول

در این قسمت، هدف اجرای PCA(Principle Component Analysis) روی مسئله‌ای دو کلاسه است که داده‌های آن به صورت تصادفی به کمک توزیع گوسی تولید شده‌اند.
برای تولید داده‌های تصادفی با توزیع گوسی نیاز به بردار میانگین و ماتریس کوواریانس است که در فرض مسئله داده شده‌است.
دستورات لازم برای تولید نمونه‌های تصادفی را در زیر مشاهده می‌کنید:

```
mu1 = [10;10];
mu2 = [22;10];
sigma = [4 4;4 9];
data1 = mvnrnd(mu1,sigma,1000);
data2 = mvnrnd(mu2,sigma,1000);
```

R1,R2 دو نمونه‌ی تصادفی ۱۰۰۰ تایی است که در شکل زیر روی نمودار مشخص شده‌اند:



نقاط قرمز رنگ داده‌های data1 و نقاط آبی رنگ داده‌های data2 هستند.

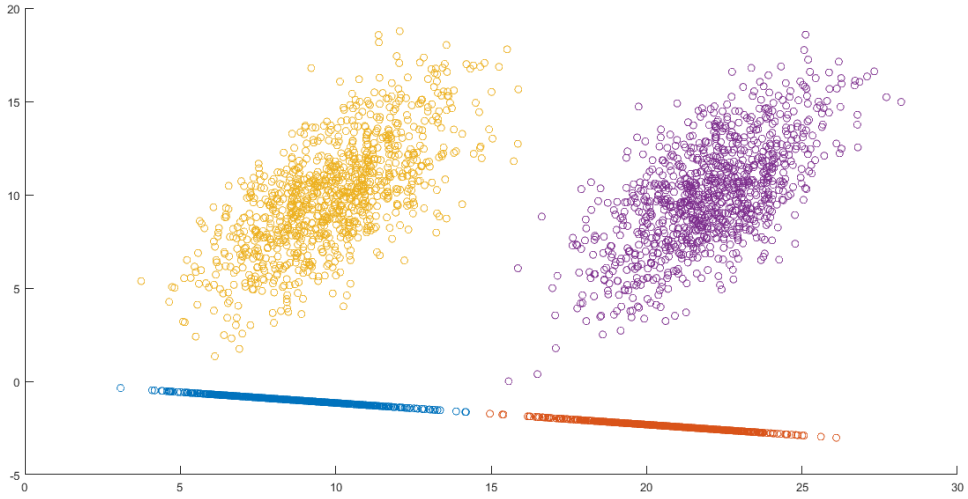
حال با ادغام این دو نمونه، به کمک تابع pca، خطی را که PCA داده‌ها را روی آن تصویر می‌کند، می‌یابیم.
ورودی و خروجی‌های تابع pca به صورت زیر است.

```
[coeff,score,latent] = pca(matrix);
```

که ماتریس coeff همان خطی که داده‌ها روی آن تصویر می‌شود را به ما می‌دهد که در زیر آمده‌است:

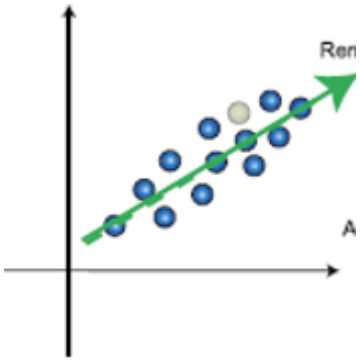
```
data = [data1;data2];
[pcaV s l] = pca(data);
for i = 1:2000
    dots(i) = dot(data(i,:),pcaV(1,:));
end
project = dots' * pcaV(1,:);
projectC1 = project(1:1000,:);
projectC2 = project(1001:2000,:);
```

که در آن projectC1,projectC2 همان تصویر دو کلاس روی خط PCA هستند که در شکل زیر نشان داده شده‌است:



خط قرمز تصویر شده‌ی داده‌های بنفش و خط آبی تصویر شده‌ی داده‌های زرد هستند. در واقع می‌توان گفت خطی که نقاط قرمز و آبی را به هم وصل می‌کند همان خط بخش **الف** است.

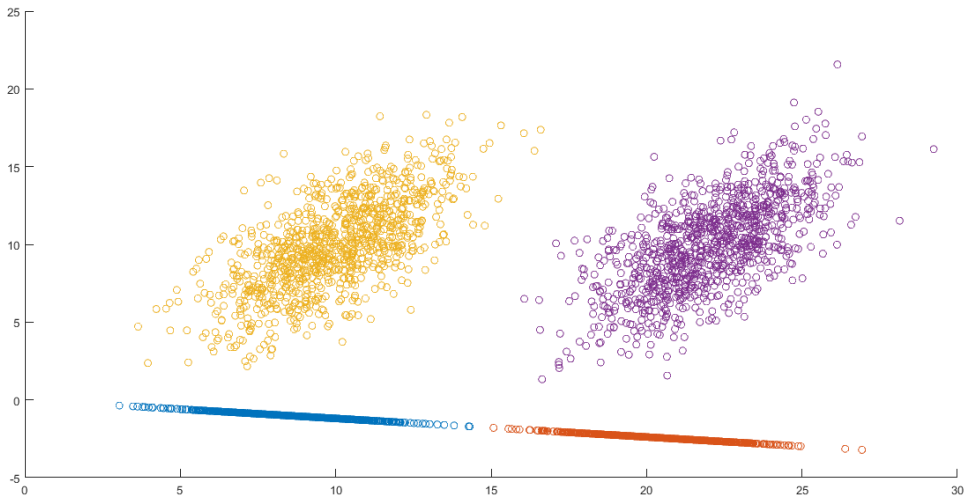
تحلیل نتیجه‌ی به‌دست آمده: با توجه به تعاریفی که از مفهوم PCA وجود دارد، انتظار داشتم که خطی که قرار است داده‌ها روی آن تصویر شود، خطی مشابه شکل صفحه‌ی بعد شود اما احتمال می‌رود به علت داشتن بردار میانگین و ماتریس کوواریانس به جای یک مقدار بودن آن‌ها و همچنین دو کلاسه بودن مسئله، شرایط تغییر کند.



در نهایت برای بازسازی داده‌ها نیاز به رابطه‌ی زیر داریم:

```
recontruct = score * coeff' + mean(matrix);
```

که ماتریس reconctruct همان ماتریس داده‌های بازسازی شده‌است که نتیجه‌ی حاصل از اجرای این دستور در مسئله ما به شکل زیر است:



خطای بازسازی یا همان min square error به روش زیر محاسبه می‌شود:

```
mse = sum((matrix - recontruct).^2);
```

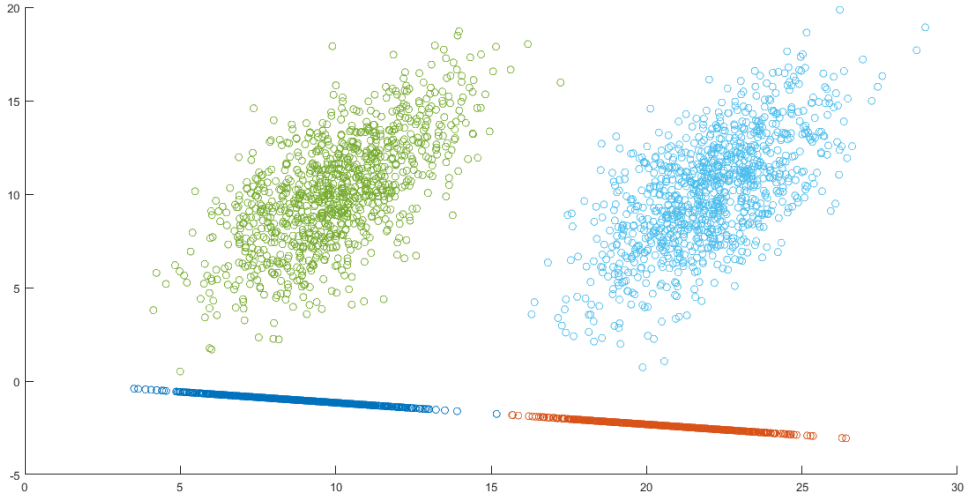
که خطای محاسبه شده در مسئله برابر است با:

```
mse =

   1.0e-26 *

    0.3061    0.0580
```

و در نهایت شکل کلی از مسئله (داده‌های اولیه، داده‌های تصویرشده و داده‌های بازسازی شده) در زیر مشاهده می‌شود:



ملاحظه می‌شود که میزان خطای بازسازی به حدی کم است که با چشم غیر مسلح تفاوت داده‌های اصلی و بازسازی شده مشاهده نمی‌شود.

قسمت دوم

- در این قسمت هر بخش به صورت فایلی جداست. که با اجرای فایل main تمامی فایل‌های بخش‌های مختلف اجرا خواهند شد.

در این بخش، هدف به‌کارگیری تجزیه مقدار منفرد (SVD) برای شناسایی چهره‌ی انسان است.
در ابتدا لازم است که عکس‌های ذخیره شده در فایل faces\images، در پروژه بارگذاری شود که از دستورات زیر استفاده شده‌است.

```
1) str = fileread('faces\train.txt');
2) temp = split(str);
3) temp = char(temp);
4) length = size(temp);
5) counter = length(1) - 1;
6) trainPics = zeros(540,2500,'uint8');
7) for i = 1:counter
8)    if mod(i,2) == 1
9)        imgData = imread(temp(i,1:28));
10)        trainPics(round(i/2),1:2500) = imgData(:)';
11)    end
12) end
```

خط اول متن موجود در فایل train را خوانده که در آن آدرس هر عکس و برچسب متناظر با آن قرار دارد.
خط دوم و سوم هر خط زیر رشته را که با فاصله از هم جدا شده‌اند را درون یک ماتریس از جنس char قرار می‌دهد. ردیف‌های فرد ماتریس حاوی آدرس عکس‌ها و ردیف‌های زوج برچسب متناظر با هر ع
کس را دارد.
کافی است ماتریسی از مسیر عکس‌ها به نام trainPics بسازیم و به کمک تابع imread عکس موجود در آدرس مورد نظر که ۵۰*۵۰ پیکسل است، خوانده شود.
خط ۱۰ این ماتریس را به آرایه ۲۵۰۰ تایی تبدیل کرده تا قابلیت ذخیره شدن به فرمت خواسته‌شده در مسئله را داشته باشد.
با اجرای این مراحل و اجرای دستور زیر یکی از عکس‌های موجود به دلخواه در زیر نشان داده شده‌است:

```
imshow(reshape(trainPics(50,1:2500)',[50,50]));
```

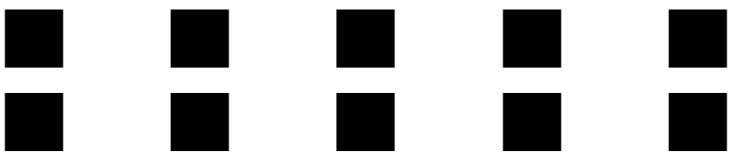
ج) برای این بخش کافی است میانگین تمامی درایه‌های ماتریس ذخیره شده را حساب کنیم. در نتیجه ماتریسی به دست می‌آید که تنها یک عکس دارد و مشابه دستور بالا می‌توان آن را رسم کرد:

د) در این قسمت باید ماتریس حاصل از قسمت قبل را از ماتریس تصاویر کم کرده و لذا ماتریسی جدید حاصل می‌شود که یکی از تصاویر آن به شکل زیر است:

ه) به کمک تابع svd که در زیر ساختار کلی آن مشخص شده‌است می‌توانیم ماتریس‌های U، Σ و V را محاسبه کنیم:

```
[U,S,V] = svd(pics);
```

تابع Eigenface تعریف شده که در واقع این ماتریس‌ها را به عنوان خروجی برمی‌گرداند. کافی است ماتریس V را ترانهاده کرده و ۱۰ تصویر اول آن را که در واقع همان ۱۰ سطر اول ترانهاده‌ی V است را نشان دهیم:



همانطور که مشخص است تمامی ۱۰ عکس اول تماما سیاه هستند.

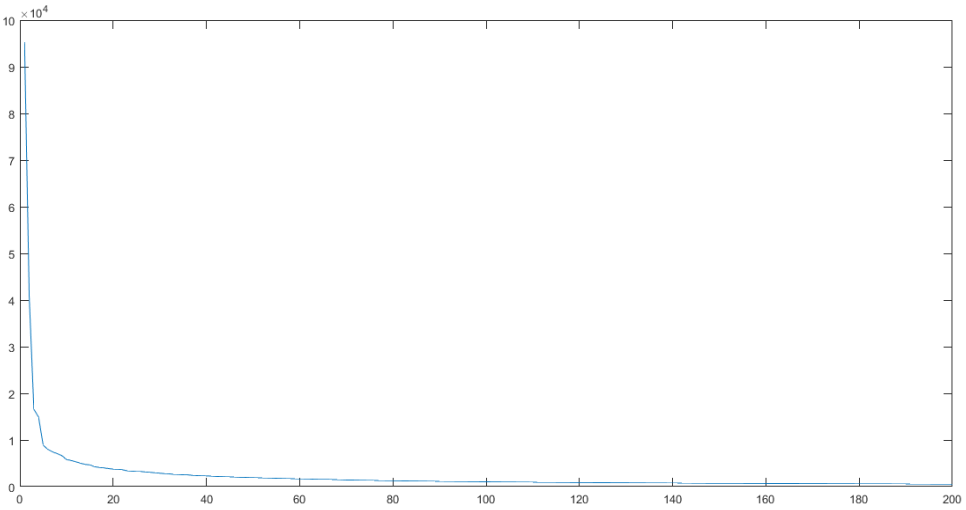
و) در این قسمت هدف محاسبه‌ی تقریب مرتبه‌ی پایین (k) است. یک روش استفاده از دستورات زیر است:

```
Ak = U(:,1:k)*S(1:k,1:k)*V(:,1:k)';
error = norm(A-Ak);
```

اما تابع error = diag(sigma) در متلب پیاده‌سازی شده که خطای تقریب مرتبه‌ی r را به ازای تمامی rهای ممکن در یک آرایه نگه‌داری می‌کند. لذا برای این بخش کافیس ۲۰۰ عنصر اول این آرایه را به روش زیر نشان دهیم:

```
plot(errorsTrain(1:200,1));
```

که شکل حاصل از آن به صورت زیر است:



ز) براساس نکته‌ی گفته شده در صورت بخش کافیست تابعی مشابه تابع زیر که با نام EigenFaceFeature در پروژه آمده‌است ایجاد شود:

```
function [F] = EigenfaceFeature(r,A,B)
%// r = low-rank , A = Vtranspose , B = Pics
    T = A(1:r,:)';
    F = double(B) * double(T);
end
```

ح) خواسته‌ی اول این بخش همان استفاده از تابع تعریف شده در قسمت قبل است:

```
[Ftrain] = EigenfaceFeature(10,VtrainTranspose,trainPics);
```

که باید یک مدل رگرسیون منطقی با Ftrain آموزش داده‌شود و روی Ftest امتحان شود. به این معنی که می‌خواهیم بررسی کنیم به احتمال چند درصد عکس‌هایی که در Ftest دارای برچسب i هستند به درستی تشخیص داده می‌شوند که مربوط به عکسی با برچسب i در Ftrain است.