Aprendizagem Automática

Aula Prática

MNIST - Imagens de Dígitos Manuscritos

Pré-Processamento
Análise em Componentes Principais
e Análise em Discriminantes Lineares

$\mathbb{G}$. Marques

Dígitos Manuscritos

1. Pré-Processamento
   - Médias e variâncias
   - Pré-processamento com `scikit-learn`. Exemplos com `Iris` dataset
   - Base de dados MNIST
   - Matrizes de covariância coeficientes de correlação dos dígitos MNIST
   - Remoção de dimensões supérfluas
2. Análise em Componente Principais (PCA)
   - Questões práticas
   - Analise da variância total
   - Transformação e reconstrução
   - Normalização da variância
3. Análise em Discriminantes Lineares (LDA)
   - Discriminantes de Fisher, 2 classes
   - LDA: discriminantes de Fisher multi-classe

Pré-Processamento

Quando se lida com dados reais é habitualmente necessário proceder a um passo de pré-processamento dos mesmos antes se de aplicar técnicas de aprendizagem automática. Dados como texto, imagem, áudio, e muitos outros, apresentam por vezes valores omissos, pontos espúrios, variáveis com escalas e valores médios distintos, etc. Na maioria dos casos, utilizar dados com estas característica influencia negativamente o desempenho dos algoritmos. Por exemplo, as máquinas de suporte vetorial (SVMs) do `scikit-learn` são particularmente sensíveis ao escalamento dos dados.

Pré-Processamento com `scikit-learn`

Existem diversos algoritmos de pré-processamento de dados no sub-módulo `preprocessing` do `scikit-learn`, entre os quais se destacam os seguintes:

- `StandardScaler`: reescala os dados de modo a todas as dimensões ficarem com média zero e variância unitária $\left(\mu = 0, \sigma^2 = 1\right)$.
- `RobustScaler`: usa a mediana e primeiro e último quartil dos dados em vez da média e variância. Mais robusto a pontos espúrios (outliers).
- `MinMaxScaler`: usa translações e escalamentos de modo a todos os valores de cada dimensão dos dados estarem no intervalo $[0, 1]$.
- `Normalizer`: cada vetor é reescalado de modo a ter norma 1. Os dados são projetados num circulo no caso de dados 2D e no caso 3D, numa esfera. Aplicável a certo tipo de dados (esparsos como no caso de texto), e para a métrica de distância de cosseno.

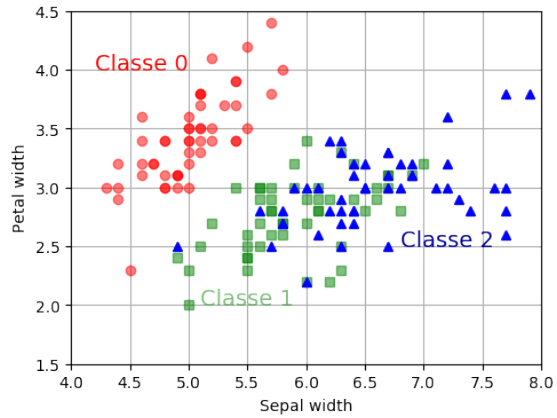
ATENÇÃO: os algoritmos em `scikit-learn` assumem que os dados estão em `np.arrays` de $N \times d$ (com N o nº de pontos e d a dimensão de cada vetor).

Pré-Processamento com `scikit-learn`

Dados do `Iris` dataset.
Carregar dados e visualizar as duas 1ª dimensões

```
>>> from sklearn.datasets import load_iris
>>> D=load_iris()
>>> X=D['data'][0:2,:].T
>>> plt.plot(X[0,:],X[1,:],'.')
```

dados originais.

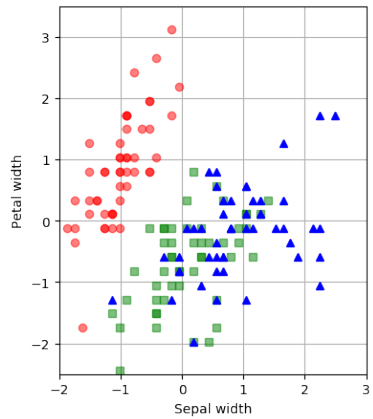


Reescalar com `StandardScaler`

```
>>> import sklearn.preprocessing as pp
>>> sc=pp.StandarScaler()
>>> sc.fit(X.T)
>>> Xs=sc.transform(X).T
>>> plt.plot(Xs[0,:],Xs[1,:],'.')
```

média e variância dos dados reescalados

```
[-1.690-15 -1.637e-15 -1.482e-15 -1.623e-15]
[ 1.  1.  1.  1.]
```



dados reescalados

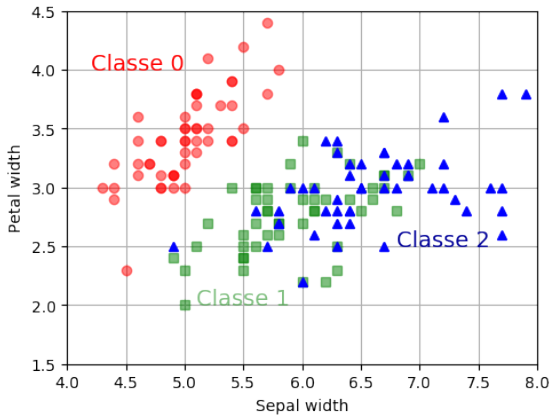
Pré-Processamento com `scikit-learn`

Dados do `Iris` dataset.

Carregar dados e visualizar as duas 1ª dimensões

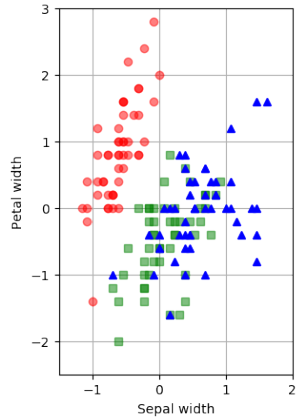
```
>>> from sklearn.datasets import load_iris
>>> D=load_iris()
>>> X=D['data'][0:2,:].T
>>> plt.plot(X[0,:],X[1,:],'.')
```

dados originais.



Reescalar com `RobustScaler`

```
>>> sc=pp.RobustScaler()
>>> sc.fit(X.T)
>>> Xs=sc.transform(X).T
>>> plt.plot(Xs[0,:],Xs[1,:],'.')
```



dados reescalados

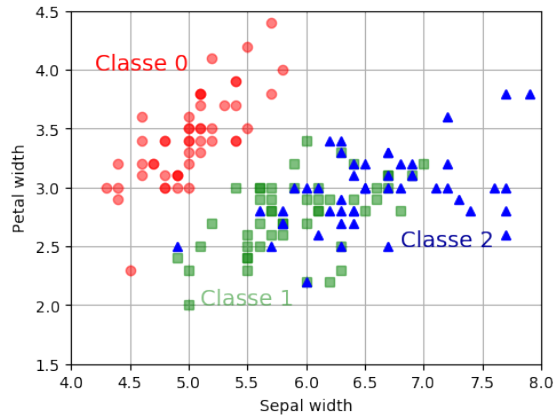
Pré-Processamento com `scikit-learn`

Dados do `Iris` dataset.

Carregar dados e visualizar as duas 1ª dimensões

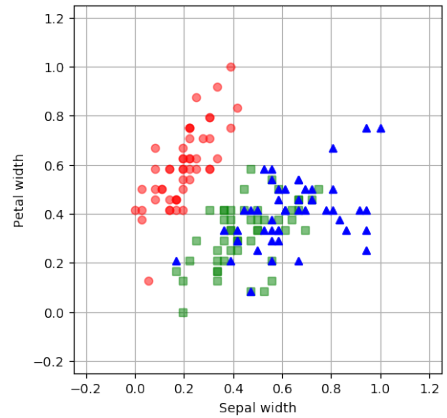
```
>>> from sklearn.datasets import load_iris
>>> D=load_iris()
>>> X=D['data'][0:2,:].T
>>> plt.plot(X[0,:],X[1,:],'.')
```

dados originais.



Reescalar com `MinMaxScaler`

```
>>> sc=pp.MinMaxScaler()
>>> sc.fit(X.T)
>>> Xs=sc.transform(X).T
>>> plt.plot(Xs[0,:],Xs[1,:],'.')
```

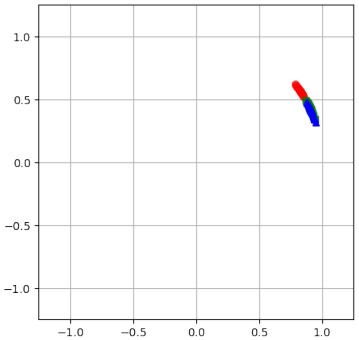


dados reescalados

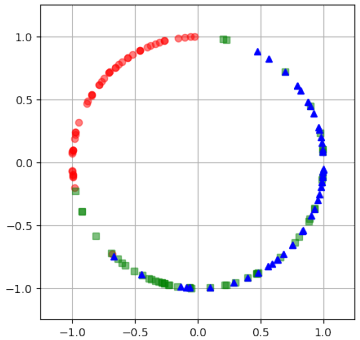
Pré-Processamento com `scikit-learn`

Dados do `Iris` dataset processados com `Normalizer`.

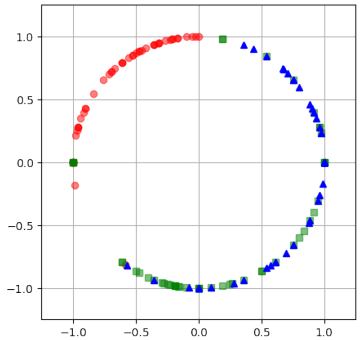
- As figuras são o resultados do processamento dos dados originais e também dos processados com as técnicas descritas anteriormente.



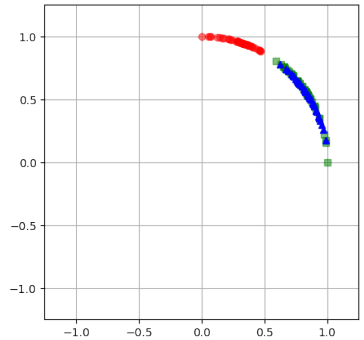
originais



`Standard`

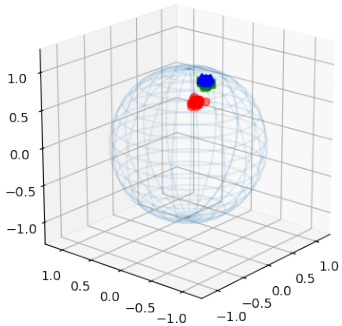


`Robust`

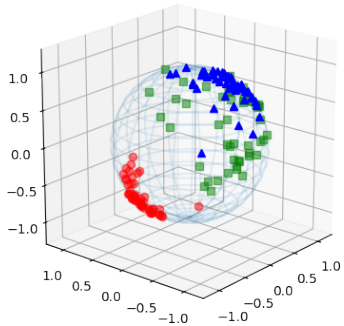


`MinMax`

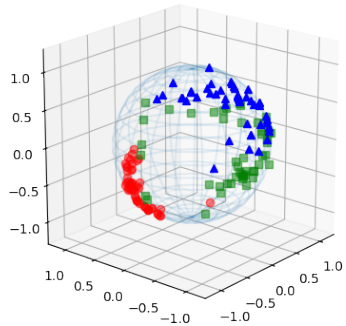
- O mesmo processamento com dados a 3 dimensões.



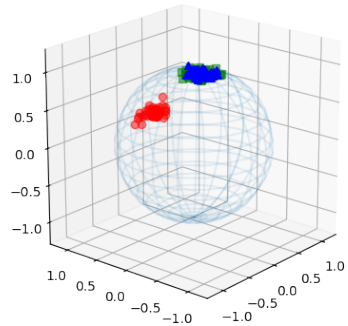
originais



`Standard`



`Robust`



`MinMax`

Pré-Processamento com `scikit-learn`

Dados do `Iris` dataset processados com `Normalizer`.

A função `Normalizer` não é aplicável a todo o tipo de dados

- Limitações:
  - Os dados têm que estar distribuídos à volta da origem.
  - Dados pertencentes a uma só classe não podem estar distribuídos à volta da origem
- Vatagens:
  - Aplicável a dados esparsos (ex: dados de texto)
  - Normalização útil quando se usa a distância de cosseno:

    $$\mathcal{D}_{\cos}(\mathbf{x}, \mathbf{y}) = 1 - \frac{\mathbf{x}^\top \mathbf{y}}{\|\mathbf{x}\| \|\mathbf{y}\|}$$

    Os dados normalizados têm norma unitária.
    A equação anterior simplifica-se: $\mathcal{D}_{\cos}(\mathbf{x}, \mathbf{y}) = 1 - \mathbf{x}^\top \mathbf{y}$

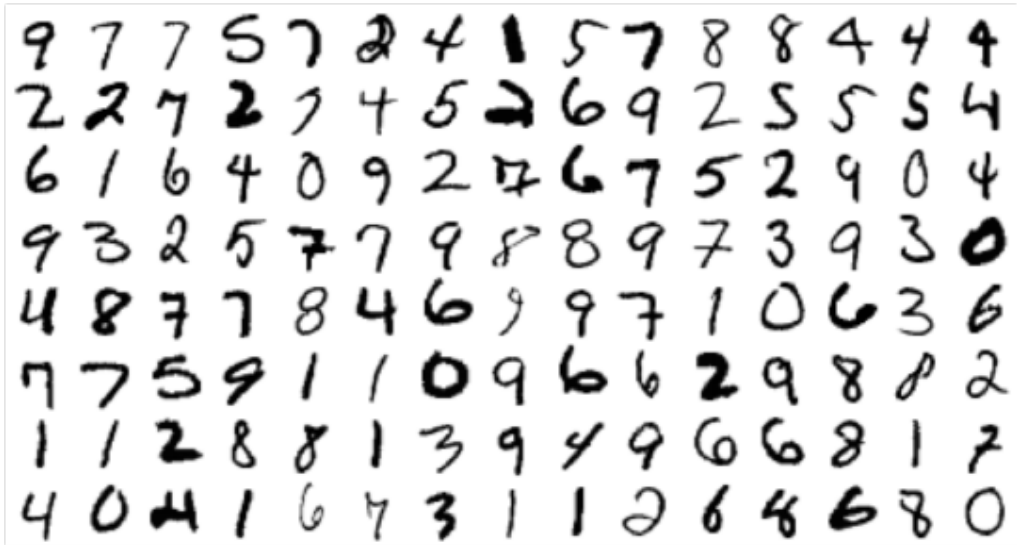
Pré-Processamento com `scikit-learn`

Os três primeiros “scalers” também apresentam algumas limitações.
Por exemplo, não são boa opção para o conjunto MNIST.

- Limitações:
  Preservam dimensões supérfluas, o que é prejudicial para vários classificadores.
- Abordagem:
  Analisar primeiro a matriz de covariância dos dados ou a matriz de coeficientes de correlação dos dígitos do conjunto MNIST.

MNIST

Base de dados com mais de 70000 imagens de dígitos manuscritos, pré-processados de modo a terem aproximadamente o mesmo tamanho e estarem centrados na imagem. No meio académico, esta base de dados é sobejamente conhecida e é muito utilizada para testar novos métodos de classificação. Para obter a informação completa sobre o MNITS, e sobre o desempenho de vários algoritmos de classificação avaliados com esta base de dados, ver: http://yann.lecun.com/exdb/mnist/.



MNIST

Dados disponibilizados: `MNISTsmall.p`

- Da base de dados MNIST foi selecionado um conjunto de 15 mil dígitos: 10 mil para treino e 5 mil para teste.
- Dígitos armazenados em ordem crescente (primeiro são 1500 “0s”, depois 1500 “1s” e assim por diante).
- Cada dígito é uma imagem em tons de cinzento `(uint8)` de 28×28 pixeis.
- As imagens estão representadas vetorialmente: vetores de $784 = 28^2$ dimensões. As primeiras 28 correspondem aos pixeis da 1ª coluna, as segundas 28 dimensões aos da 2ª coluna, e por aí em diante.
- Variáveis do dicionário armazenado em `MNISTsmall.p`
  - `X`: Matriz (np.array) de 784×15000 com todos os dígitos (treino e teste)
  - `trueClass`: np.array de 15000 índices das classes (inteiros de `0` a `9`)
  - `foldTrain`: np.array de 15000 boolenanos. Os “`True`” são os dados de treino
  - `foldTest`: np.array de 15000 booleanos. Os “`True`” são os dados de teste

MNIST

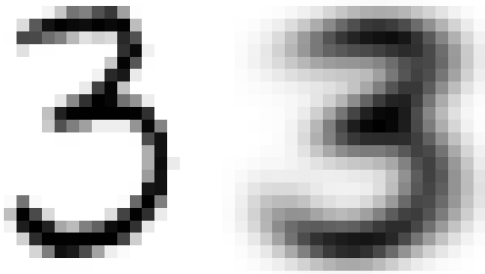
Leitura e Visualização:

- Leitura (módulo `pickle`)

```
>>> D=pickle.load(open('MNISTsmall.p','rb'))# D do tipo "dictionary"
# train3: dígitos "3s" de treino
>>> train3=D['X'][:,(trueClass==3)& foldTrain] # np.array de 784×1000
```

- Visualizações

```
# ex: ver o oitavo exemplo do dígito 3 de treino
>>> I3_8=np.reshape(train3[:,7],(28,28))
# ver o negativo
>>> plt.imshow(255-I3_8, \
interpolation='none',cmap='gray')
# calcular o dígito médio e visualizar
>>> I3_m=np.mean(train3,1)
```



3 (nº 8) 3 médio

Matrizes de Covariância

- Covariância é uma medida de correlação entre variáveis (entre pixeis)
- Correlação é uma mediada de dependência (mas no sentido estatístico, descorrelação não implica independência)
- Matrizes apresentam padrões repetidos (de 28 em 28 pixeis)
- Valores influenciados pela escalamento dos dados
- Há variâncias com valores nulos! ($\sigma_i^2 = 0$)
- **Coeficiente de Correlação**: $\rho_{ij} = \dfrac{\text{cov}(x_i, x_j)}{\sigma_i \sigma_j}$
  - Grau de correlação entre duas variáveis: valores entre $[-1, +1]$
  - Máxima correlação: valores de ρ perto de ± 1
  - Dados **descorrelacionados**: valores de ρ perto de 0
  - Independente da escala dos dados
  - Em Python: `np.corrcoef()`
    Atenção: quando $\sigma_i = 0$ ou $\sigma_j = 0$, $\rho_{ij} \Longrightarrow$ indefinido (`nan` em Python)

Matrizes de Covariância

- Dígitos representados em vetores de 784 dimensões:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{784} \end{bmatrix} \quad \boldsymbol{\mu}_{\mathbf{x}} = \begin{bmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_{784} \end{bmatrix} \quad \boldsymbol{\Sigma}_{\mathbf{x}} = \begin{bmatrix} \sigma_1^2 & \text{cov}(x_1, x_2) & \cdots & \text{cov}(x_1, x_{784}) \\ \text{cov}(x_2, x_1) & \sigma_2^2 & & \text{cov}(x_2, x_{784}) \\ \vdots & \cdots & \ddots & \\ \text{cov}(x_{784}, x_1) & \cdots & & \sigma_{784}^2 \end{bmatrix}$$

784×1 784×1 784×784

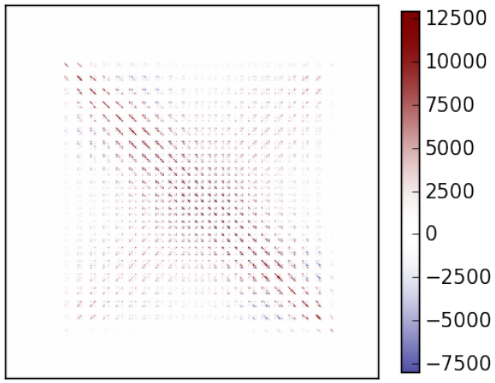
$$\boldsymbol{\mu}_{\mathbf{x}} = \mathbb{E}\{\mathbf{x}\} \approx \frac{1}{N} \sum_{n=1}^{N} \mathbf{x}_n \qquad \boldsymbol{\Sigma}_{\mathbf{x}} = \mathbb{E}\{(\mathbf{x} - \boldsymbol{\mu}_{\mathbf{x}})(\mathbf{x} - \boldsymbol{\mu}_{\mathbf{x}})^\top\} \approx \frac{1}{N-1} \sum_{n=1}^{N} (\mathbf{x} - \boldsymbol{\mu}_{\mathbf{x}})(\mathbf{x} - \boldsymbol{\mu}_{\mathbf{x}})^\top$$

$$\text{cov}(x_i, x_j) = \text{cov}(x_j, x_i) \approx \frac{1}{N-1} \sum_{n=1}^{N} (x_i - \mu_i)(x_j - \mu_j)$$

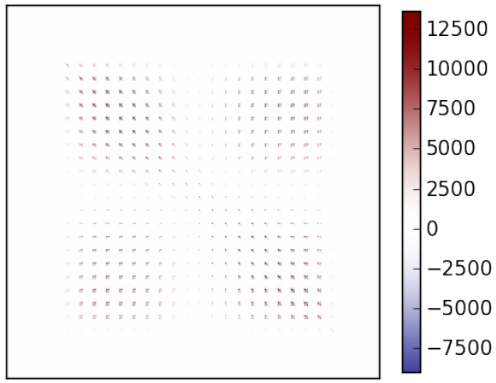
- Duas maneiras de calcular $\boldsymbol{\Sigma}_{\mathbf{x}}$ em Python (ver 1ª aula prática)
  1. `>>> C=np.cov(X)` # `X` matriz de $784 \times N$
  2. `>>> Xn=(X.T-np.mean(X,1)).T` # tirar média
     `>>> C=np.dot(Xn,Xn.T)/(N-1)`
- **Atenção:** antes de processar, converter dados para `float`!

Matrizes de Covariância

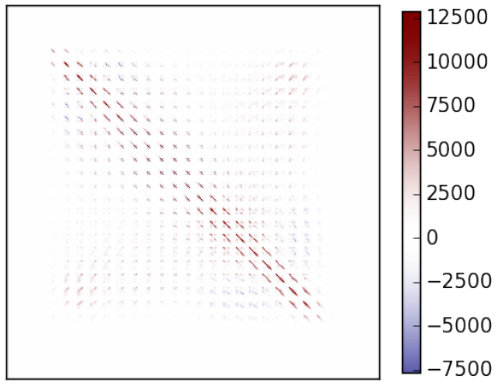
Visualizações das matrizes de covariância



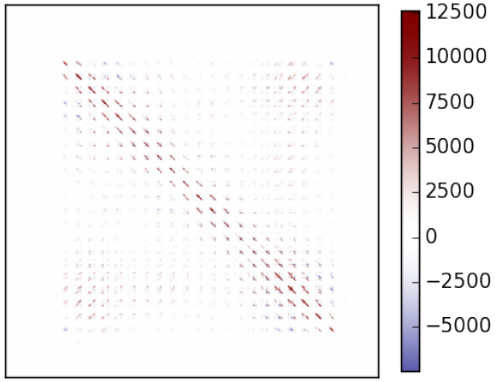
0



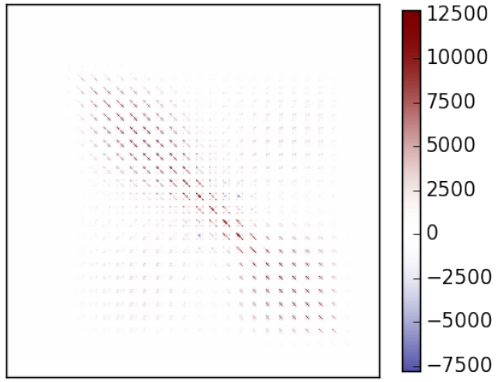
1



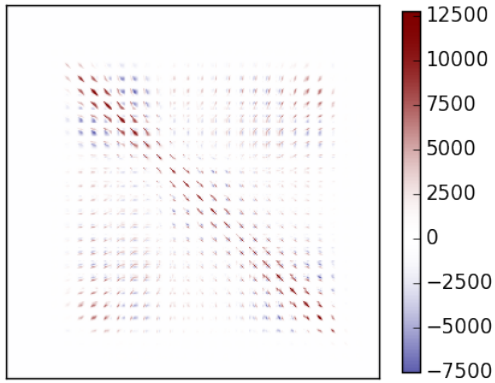
2



3



4



5

. . .

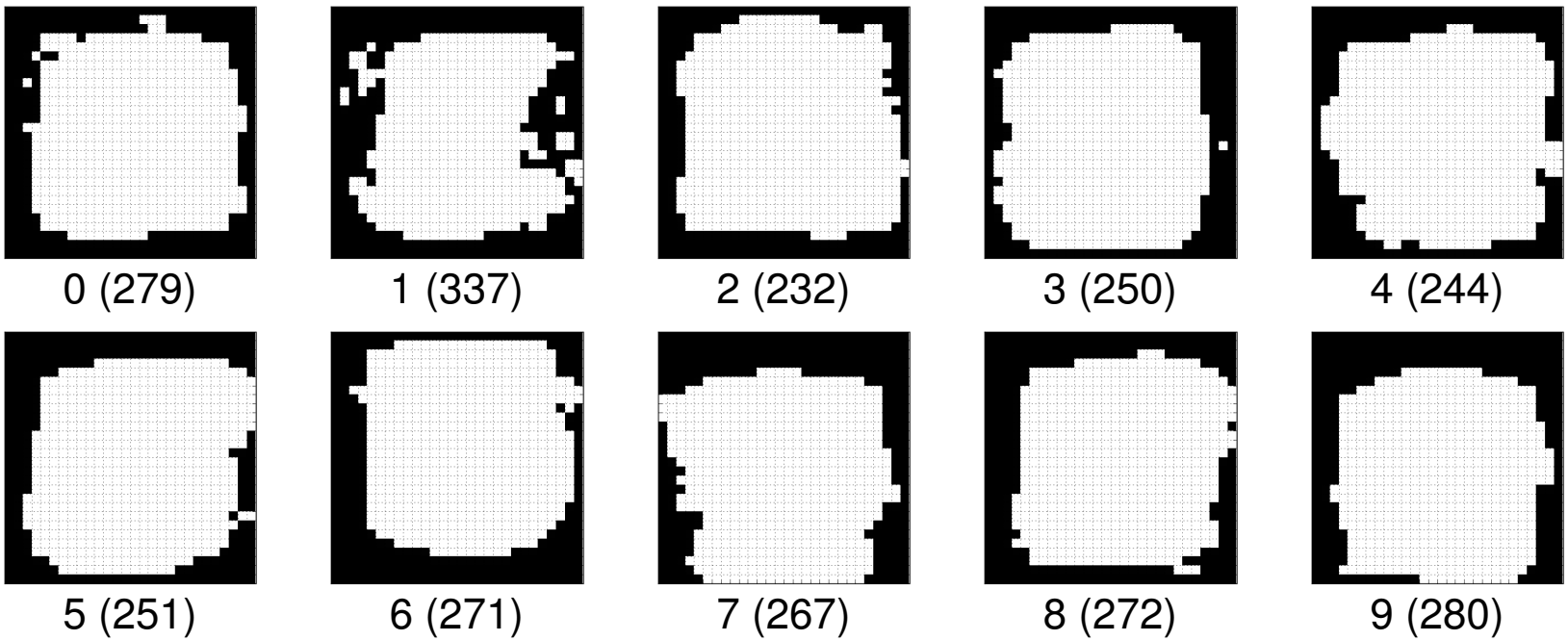
Remoção de Dimensões Supérfluas

- Para cada conjunto de dígitos, ver quais dimensões têm $\sigma_i^2 = 0$

```
# Conjunto dos dígitos "3" de treino
>>> C3=np.cov(train3) # C3 matriz de 784×784
>>> idx3_0=np.diag(C3)!=0 # índice (True quando σ_i^2 ≠ 0)
```

- Fazer imagens de 28×28 com píxeis inactivos a preto

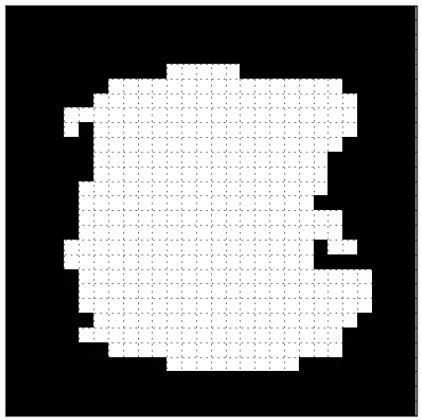
```
>>> I3_0=np.reshape(idx3_0,(28,28)) # I3_0 imagens com pixeis inactivos
>>> plt.imshow(I3_0,cmap='gray',interpolation='none') # mostrar
```



Remoção de Dimensões Supérfluas

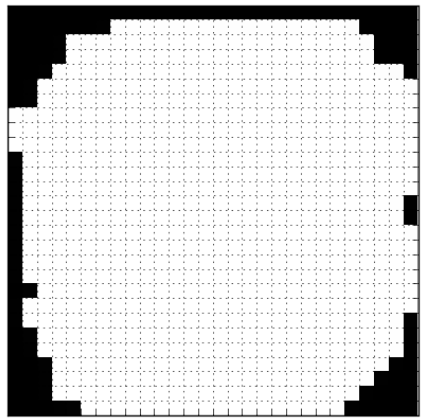
- Quais dimensões remover?

AND



431 pixeis “off”

OR



111 pixeis “off”

- Fazendo o “OR” não deitamos fora nenhuma dimensão com $\sigma_i^2 \neq 0$.
  Desvantagem: deve haver dimensões que variam muito pouco e são desnecessárias.
- Fazendo o “AND” reduzimos a dimensão dos dados e só usamos as que variam em todas as imagens. Desvantagem: deve haver dimensões com informação que são descartadas.
- Melhor: usar PCA

Análise em Componentes Principais (PCA)

- PCA é um método não supervisionado - não há classes.

  Aplicar a todos os dados (de treino), independentemente das classes.
  - $\mathbf{x}$ dados originais (vetores de $d \times 1$ com $d = 784$)
  - $\mu_{\mathbf{x}}$ média de $\mathbf{x}$
  - $\mathbf{\Sigma_x}$ matriz de covariância de $\mathbf{x}$
  - $\mathbf{\Sigma_x} = \Gamma \Delta \Gamma^\top$ decomposição em vetores e valores próprios
  - Γ matriz de vetores e Δ matriz diagonal de valores próprios
- Transformação: $\mathbf{y} = \mathbf{W}^\top \mathbf{x}$
  - Matriz de transformação $\mathbf{W}$ composta pelas colunas de Γ
  - Escolher $k \leq d$ colunas de Γ (as dos k maiores valores próprios)
  - Matriz de covariância, $\mathbf{\Sigma_y}$, de $\mathbf{y}$ é uma matriz diagonal
  - Os elementos de $\mathbf{\Sigma_y}$ são os k primeiros valores próprios de Δ
- PCA projecta os dados nas direcções de maior variância (as componentes principais)
- Os dados projectados são descorrelacionados (a matriz de covariância é diagonal)
- Maneira intuitiva de escolher o número de componentes:

$\Longrightarrow$ Escolher tantas quanto necessário para perfazer uma percentagem pré-definida da variância total

Análise em Componentes Principais (PCA)

- Obter valores e vetores próprios. Usar todos os dados de treino!

```
# X matriz com todos os dígitos de treino 784×N
>>> Cx=np.cov(X)
>>> (v,W)=np.linalg.eig(Cx) # v valores próprios, W vetores próprios
```

- Questões de Python:
  - Os valores próprios, v, são complexos devido a erros de arredondamento

```
>>> np.imag(v).max() # verificar valor máximo imaginário
>>> v=v.real# converter para real
```

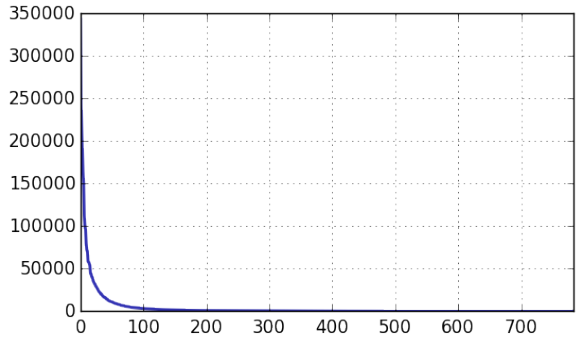
  - Os valores próprios não vêm ordenados

```
>>> idx=np.argsort(-v)# -v para ordem decrescente
>>> v=v[idx]# ordenar valores
>>> W=W[:,idx]# ordenar vetores
>>> W=W[:,v>=1e-10]# remover componentes com v≈ 0
>>> W=W.real# tirar parte imaginária (verificar se é desprezável)
```

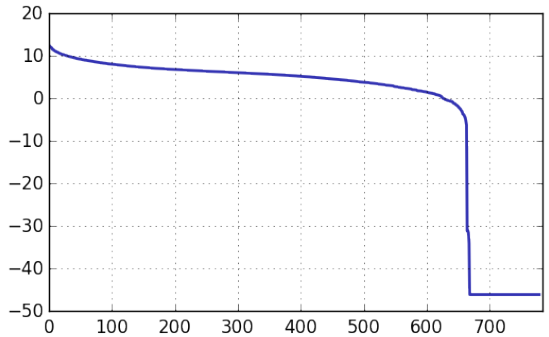
Análise em Componentes Principais (PCA)

- Analise da variância dos dados projetados (todas as componentes)
- **Lembrete:** Matriz de covariância é igual à dos valores próprios:

$$\mathbf{\Sigma_y} = \Delta = \begin{bmatrix} \delta_1 & 0 & 0 & \cdots & 0 \\ 0 & \delta_2 & 0 & \cdots & 0 \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ 0 & \cdots & 0 & 0 & \delta_d \end{bmatrix} \quad \text{com } \delta_1 \geq \delta_2 \geq \ldots \geq \delta_d$$



valores próprios



logaritmo dos valores próprios

- Conjunto de treino (N=10000): 111 valores próprios com $\delta_i = 0$
- Arredondando: 120 valores próprios com $\delta_i \leq 10^{-10}$
- Ver `np.finfo(float).eps` para obter precisão do Numpy

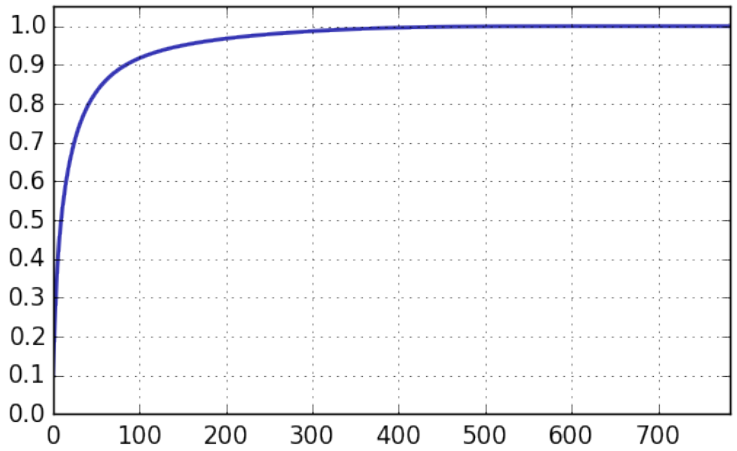
Análise em Componentes Principais (PCA)

- Normalizar os valores próprios pela soma de todos

```
>>> v=v/np.sum(v)
```

- Visualizar a soma cumulativa dos valores próprios δ_i: $\Lambda_n = \frac{\sum_{i=1}^{n} \delta_i}{\sum_{j=1}^{784} \delta_j}$

```
>>> L=np.cumsum(v)
>>> plt.plot(L)
```



- Número de componentes que contêm 95% da variância

```
>>> np.sum(L<=0.95) # 150 componentes
```

- Número de componentes que contêm 99% da variância

```
>>> np.sum(L<=0.99) # 325 componentes
```

Análise em Componentes Principais (PCA)

Projetar nas componentes principais:

- Matriz de transformação, **W** composta pelos *k* primeiros vetores próprios

```
>>> W=W[:,0:k] # ex: escolher k=150
```

- Tirar média aos dados antes de transformar

```
>>> mx=np.mean(X,1) # X matriz com todos os dígitos
>>> Xn=X-mx[:,np.newaxis]
```

- Projectar dígitos nas *k* componentes principais

```
>>> Y=np.dot(W.T,Xn)
```

Análise em Componentes Principais (PCA)

Reconstrução:

- Fazer transformação inversa

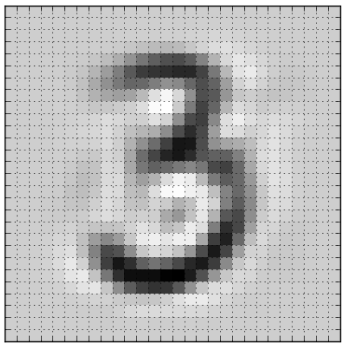
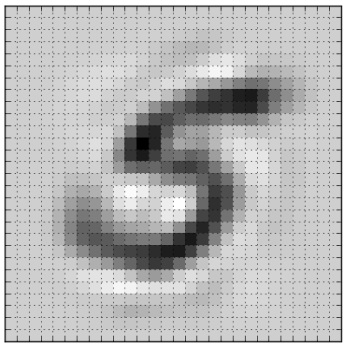
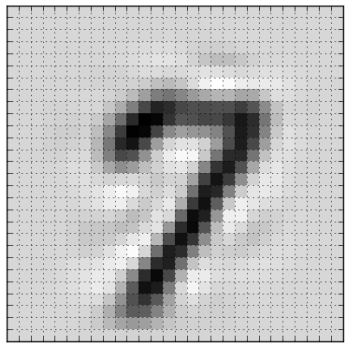
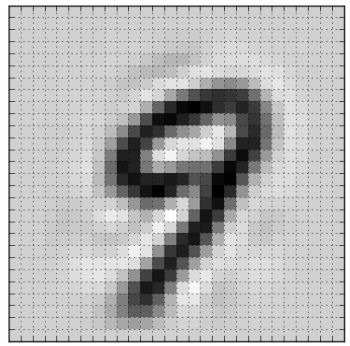
```
>>> Xr=np.dot(W,Y)
>>> Xr=Xr+mx[:,np.newaxis]# repor a média
```

- Normalizar amplitudes dos dígitos reconstruídos

(para terem valores entre $[0, 255]$)

```
>>> Xr=Xr-Xr.min()
>>> Xr=255.0*Xr/Xr.max()
```

Alguns exemplos de dígitos reconstruídos ($k = 50$)

Análise em Componentes Principais (PCA)

Reconstrução:

- Fazer transformação inversa

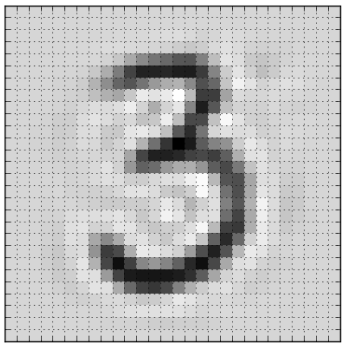
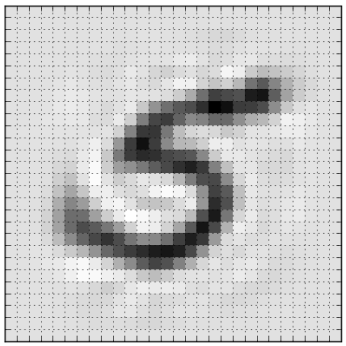
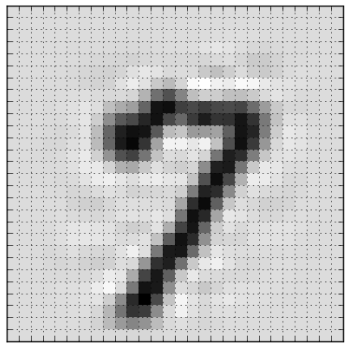
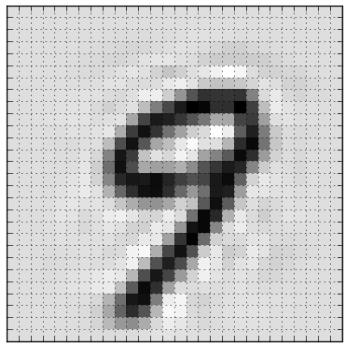
```
>>> Xr=np.dot(W,Y)
>>> Xr=Xr+mx[:,np.newaxis]# repor a média
```

- Normalizar amplitudes dos dígitos reconstruídos

(para terem valores entre $[0, 255]$)

```
>>> Xr=Xr-Xr.min()
>>> Xr=255.0*Xr/Xr.max()
```

Alguns exemplos de dígitos reconstruídos ($k = 100$)

Análise em Componentes Principais (PCA)

Reconstrução:

- Fazer transformação inversa

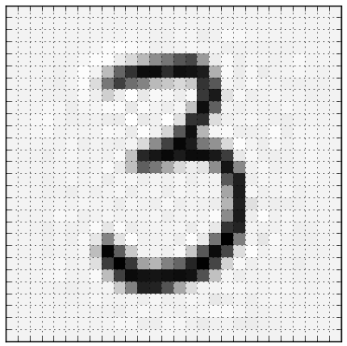
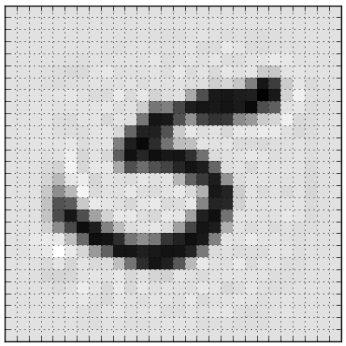
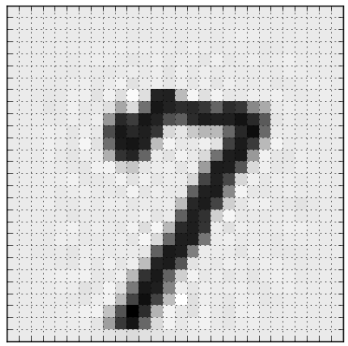
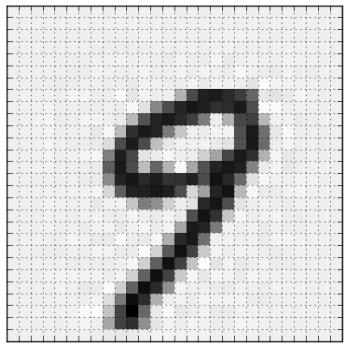
```
>>> Xr=np.dot(W,Y)
>>> Xr=Xr+mx[:,np.newaxis]# repor a média
```

- Normalizar amplitudes dos dígitos reconstruídos

(para terem valores entre $[0, 255]$)

```
>>> Xr=Xr-Xr.min()
>>> Xr=255.0*Xr/Xr.max()
```

Alguns exemplos de dígitos reconstruídos ($k = 300$)

Análise em Componentes Principais (PCA)

Reconstrução:

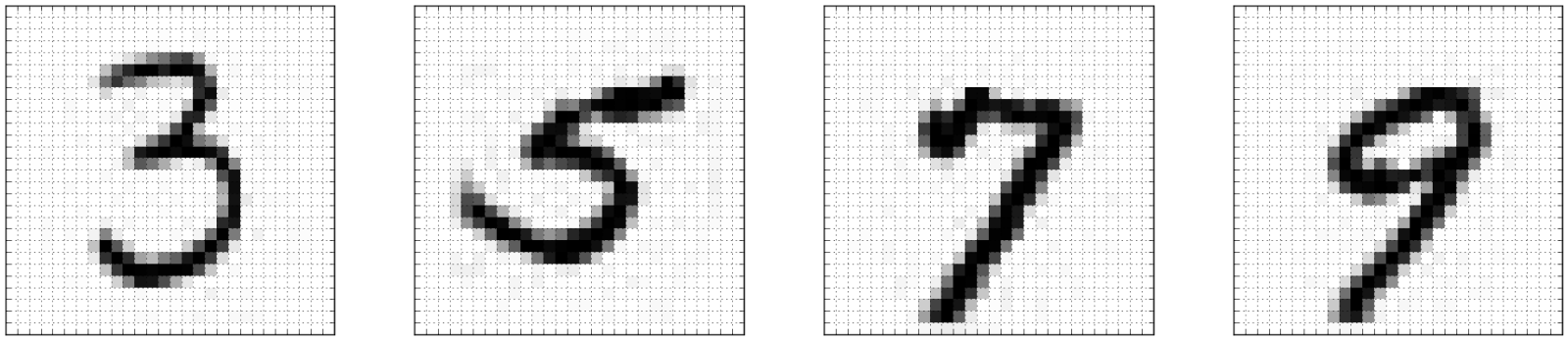
- Fazer transformação inversa

```
>>> Xr=np.dot(W,Y)
>>> Xr=Xr+mx[:,np.newaxis]# repor a média
```

- Neste caso específico, uma melhor abordagem é saturar os valores dos dígitos (valores $\leq 0 \rightarrow 0$, valores $\geq 255 \rightarrow 255$)

```
>>> Xr=np.clip(Xr,0,255)
```

Alguns exemplos de dígitos reconstruídos ($k = 300$)



Análise em Componentes Principais (PCA)

Normalizar variância dos dados projetados:

- É comum em dados reais, diferentes variáveis terem valores com escalas muito diferentes.

Ex: Dígitos de treino projetados, $\sigma_1^2 = \mathbb{E}\{(y_1 - \mu_{y_1})^2\} \approx 343000$

$$\sigma_{100}^2 = \mathbb{E}\{(y_{100} - \mu_{y_{100}})^2\} \approx 3250$$

$$\sigma_{300}^2 = \mathbb{E}\{(y_{300} - \mu_{y_{300}})^2\} \approx 450$$

- Classificadores podem dar demasiada relevância às variáveis com maior escala e ignorar as outras
- Solução: Normalizar os y_i de modo a que $\sigma_i^2 = 1.0$
- Projetar nas componentes principais
  `Xn` – dados sem média
  `W` – matriz de transformação

```
>>> Y=np.dot(W.T,Xn)
```

- Normalizar dimensões de `Y` para terem $\sigma_i^2 = 1.0$

```
>>> Yn=np.dot(np.diag(np.std(Y,axis=1)**-1),Y)
```

Análise em Componentes Principais (PCA)

Normalizar variância dos dados projetados:

- É comum em dados reais, diferentes variáveis terem valores com escalas muito diferentes.

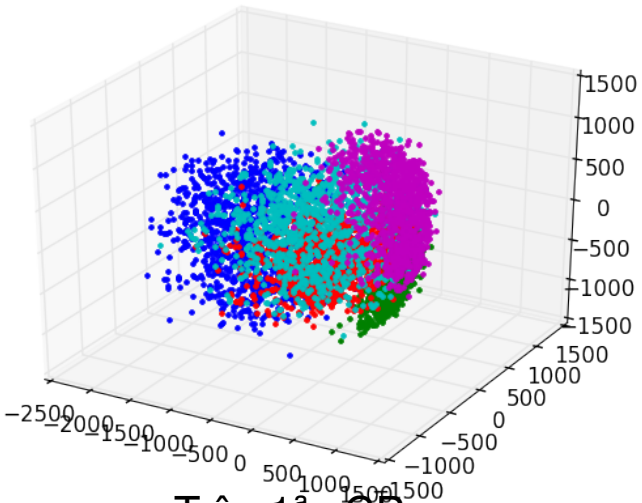
Ex: Dígitos de treino projetados, $\sigma_1^2 = \mathbb{E}\{(y_1 - \mu_{y_1})^2\} \approx 343000$

$$\sigma_{100}^2 = \mathbb{E}\{(y_{100} - \mu_{y_{100}})^2\} \approx 3250$$

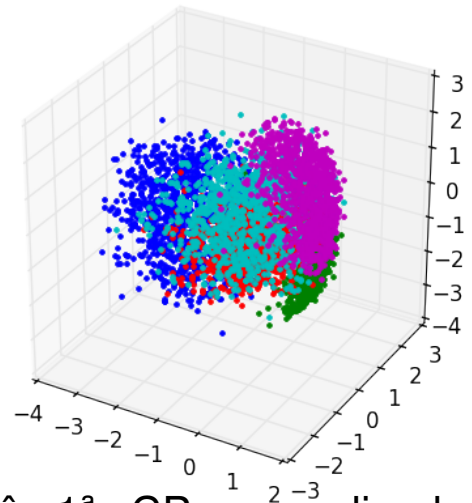
$$\sigma_{300}^2 = \mathbb{E}\{(y_{300} - \mu_{y_{300}})^2\} \approx 450$$

- Classificadores podem dar demasiada relevância às variáveis com maior escala e ignorar as outras
- **Solução:** Normalizar os y_i de modo a que $\sigma_i^2 = 1.0$

Dados por classe (dígitos de 0 a 4)



Três 1ªs CPs



Três 1ªs CPs normalizadas

Análise em Componentes Principais (PCA)

Comandos em `scikit-learn`:

- Carregar módulo PCA:
  ```
  >>> from sklearn.decomposition import PCA
  ```
- Instanciar objecto da classe PCA:
  ```
  >>> pca=PCA() # guardar todas as componentes
  ```
  Atenção: por omissão, `PCA` mantém todos as componentes principais. Para reduzir a dimensão dos dados é necessário especificar quantas componentes se pretende guardar.
  ```
  >>> pca=PCA(n_components=3) # guardar só 3 componentes
  ```
- Estimação do modelo para os dados
  (atenção: recebe matrizes de $N \times d$, N nº pontos, d dimensão dos dados)
  ```
  >>> pca.fit(X) # X matriz de dados
  ```
- Projectar dados nas componentes principais
  ```
  >>> Y=pca.transform(X)
  ```

Análise em Componentes Principais (PCA)

Comandos em `scikit-learn`:

- Ver parâmetros da classe `PCA`:

  `n_components, copy, whiten,svd_solver, tol, iterated_power,random_state`

- Ver métodos associados à classe:

  `fit()`
  `fit_transform()`
  `get_covariance()`
  `get_params()`
  `get_precision()`
  `inverse_transform()`
  `score()`
  `score_samples()`
  `set_params()`
  `transform()`

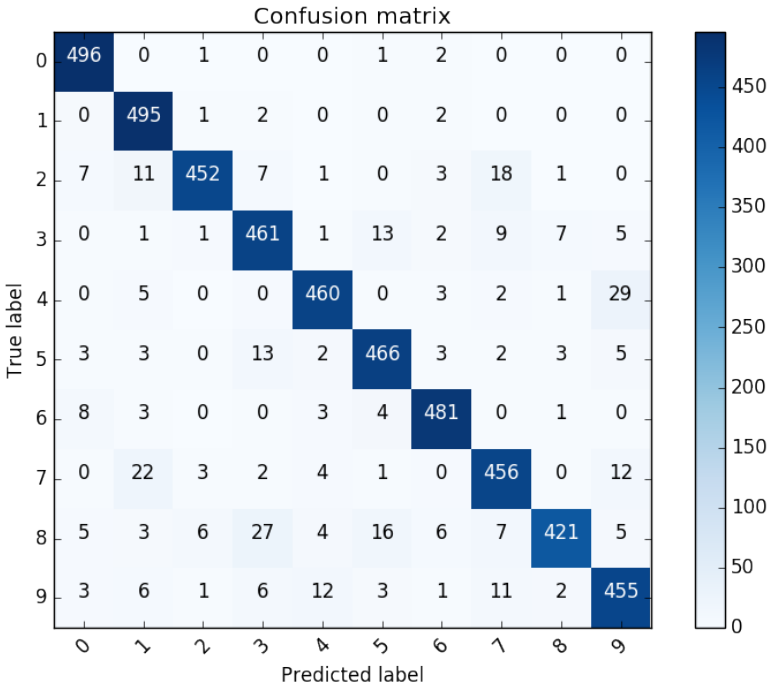
Análise em Componentes Principais (PCA)

Classificador dos k-Vizinhos Mais Próximos (k-NN)

Dígitos Manuscritos - dados em bruto

- $\mathcal{X}$, conjunto de dígitos - dados em bruto ($d = 784$).
  - Nº de pontos treino: 1000 pts por classe
  - Nº de pontos teste: 500 pts por classe

Classificação ($k = 1$):



$$\text{Prob.Erro} = \frac{357}{5000} = 7.140\%$$

Análise em Componentes Principais (PCA)

Classificador dos k-Vizinhos Mais Próximos (k-NN)

Dígitos Manuscritos - dados processados com PCA

- Calcular PCA só com conjunto de treino, e aplicar transformação ao conjunto de teste
- Comandos de Python (usando `scikit-learn`)

```
>>> from sklearn.decomposition import PCA
>>> pca=PCA(n_components=nPCs)
```

  - Instanciado objecto `pca`: irá guardar `nPCs` componentes principais

  Nota: assuma que os dados de treino estão guardados numa matriz `Xtrain` de 784×10000 e os dados de teste na matriz `Xtest` de 784×5000

- Calcular média dos dados de treino e remover-la

```
>>> mx=np.mean(Xtrain,axis=1)
>>> Xtrain=Xtrain-mx[:,np.newaxis];Xest=Xtest-mx[:,np.newaxis]
```

- Estimar PCA e projetar dados de treino e teste

```
>>> pca.fit(Xtrain.T)
>>> Ytrain=pca.transform(Xtrain.T).T
>>> Ytest=pca.transform(Xtest.T).T
```

Análise em Componentes Principais (PCA)

Classificador dos k-Vizinhos Mais Próximos (k-NN)

Dígitos Manuscritos - dados processados com PCA

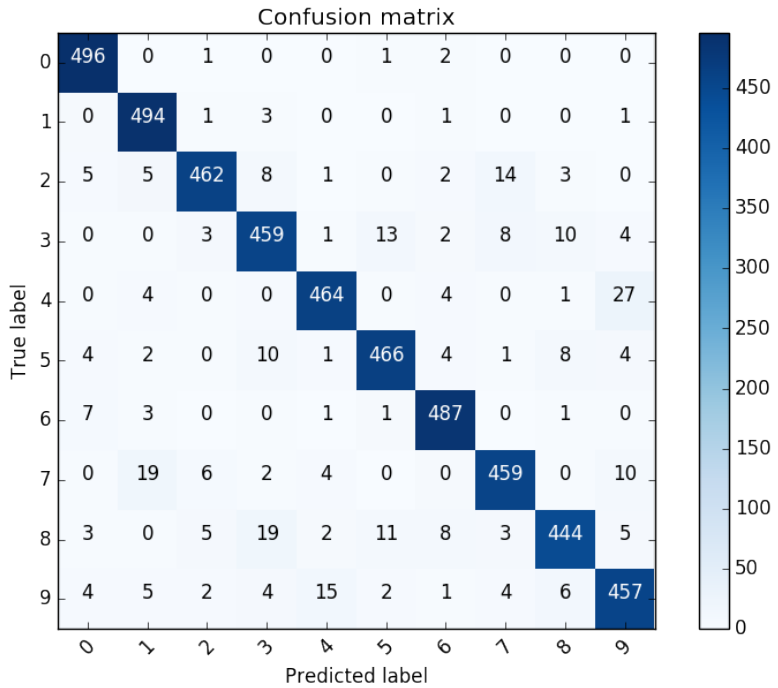
- Importar e instanciar classificador kNN (usando `scikit-learn`)

```
>>> from sklearn.neighbors import KNeighborsClassifier
>>> kNN=KNeighborsClassifier(n_neighbors=1,weigths='uniform')
```

- Treinar e classificar

```
>>> kNN.fit(Ytrain.T,trainClasses)# trainClasses classes dos dígitos
>>> resultados=kNN.predict(Ytest.T)
```

Classificação (k = 1) com `nPCs`=50:



$$\text{Prob.Erro} = \frac{312}{5000} = 6.240\%$$

Análise em Componentes Principais (PCA)

Classificador dos k-Vizinhos Mais Próximos (k-NN)

Dígitos Manuscritos - dados processados com PCA

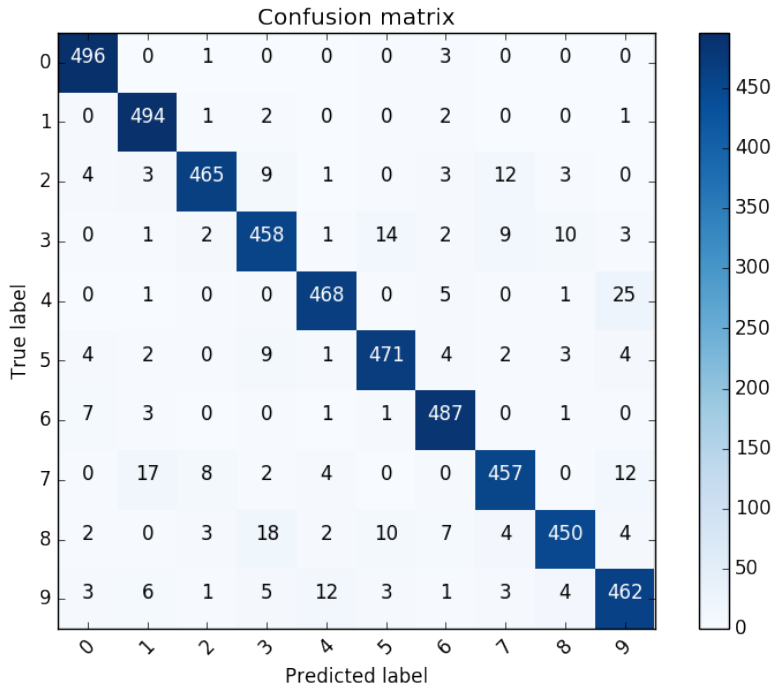
- Importar e instanciar classificador kNN (usando `scikit-learn`)

```
>>> from sklearn.neighbors import KNeighborsClassifier
>>> kNN=KNeighborsClassifier(n_neighbors=1,weigths='uniform')
```

- Treinar e classificar

```
>>> kNN.fit(Ytrain.T,trainClasses)# trainClasses classes dos dígitos
>>> resultados=kNN.predict(Ytest.T)
```

Classificação (k = 1) com `nPCs`=40:



$$\text{Prob.Erro} = \frac{293}{5000} = 5.860\%$$

Análise em Discriminantes Lineares (LDA)

- Generalização do método dos discriminantes de Fisher para mais que duas classes
- Objectivo: Encontrar uma projecção de modo a maximizar a variância inter-classe (entre classes) e minimizar variância intra-classe (dentro da mesma classe)

Discriminantes de Fisher (2 classes):

- $\Omega = \{\varpi_1, \varpi_2\}$, com $N_i = |\mathbf{x} \in \varpi_i|$ e $i = 1, 2$
- Dados projetados num recta $y = \mathbf{w}^\top \mathbf{x}$
  cada vetor $\mathbf{x}$ a d dimensões é convertido num escalar y

$$y = \mathbf{w}^\top \mathbf{x} = \begin{bmatrix} w_1 & w_2 & \cdots & w_d \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}$$

Análise em Discriminantes Lineares (LDA)

- Generalização do método dos discriminantes de Fisher para mais que duas classes
- Objectivo: Encontrar uma projecção de modo a maximizar a variância inter-classe (entre classes) e minimizar variância intra-classe (dentro da mesma classe)

Discriminantes de Fisher (2 classes):

- $\Omega = \{\varpi_1, \varpi_2\}$, com $N_i = |\mathbf{x} \in \varpi_i|$ e $i = 1, 2$
- Dados projetados num recta $y = \mathbf{w}^\top \mathbf{x}$ de modo a maximizar a função:

$$\mathcal{J}(\mathbf{w}) = \frac{\mathbf{w}^\top (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)^2 \mathbf{w}}{\mathbf{w}^\top \boldsymbol{\Sigma}_\Omega \mathbf{w}}$$

$$\text{para } \boldsymbol{\mu}_i = \frac{1}{N_i} \sum_{\mathbf{x} \in \varpi_i} \mathbf{x}$$

$$\text{e para } \boldsymbol{\Sigma}_\Omega = \boldsymbol{\Sigma}_1 + \boldsymbol{\Sigma}_2 \text{ com } \boldsymbol{\Sigma}_i = \frac{1}{N_i - 1} \sum_{\mathbf{x} \in \varpi_i} (\mathbf{x} - \boldsymbol{\mu}_i)(\mathbf{x} - \boldsymbol{\mu}_i)^\top$$

Análise em Discriminantes Lineares (LDA)

- Generalização do método dos discriminantes de Fisher para mais que duas classes
- Objectivo: Encontrar uma projecção de modo a maximizar a variância inter-classe (entre classes) e minimizar variância intra-classe (dentro da mesma classe)

Discriminantes de Fisher (2 classes):

- $\Omega = \{\varpi_1, \varpi_2\}$, com $N_i = |\mathbf{x} \in \varpi_i|$ e $i = 1, 2$
- Dados projetados num recta $y = \mathbf{w}^\top \mathbf{x}$ de modo a maximizar a função:

$$\mathcal{J}(\mathbf{w}) = \frac{\mathbf{w}^\top (\mu_1 - \mu_2)^2 \mathbf{w}}{\mathbf{w}^\top \mathbf{\Sigma}_\Omega \mathbf{w}}$$

- Solução: $\mathbf{w} = \mathbf{\Sigma}_\Omega^{-1}(\mu_1 - \mu_2) = (\mathbf{\Sigma}_1 + \mathbf{\Sigma}_2)^{-1}(\mu_1 - \mu_2)$

Análise em Discriminantes Lineares (LDA)

Exemplo: Discriminantes de Fisher (2 classes)

- Objectivo: Separar imagens de "0_s" e de "1_s"

```
>>> D=pickle.load(open('mnist_small.p','rb'))
>>> y,f=D['trueClass'],D['foldTrain']
>>> X0,X1=D['X'][:,(y==0)& f],D['X'][:,(y==1)& f]
>>> X=np.hstack((X0,X1)) # dados numa matriz de 784×2000
```

- Projecção LDA calculada com dados de treino (2000 imagens total)

$$y = \mathbf{w}^T\mathbf{x} \text{ com } \mathbf{w} = (\boldsymbol{\Sigma}_0 + \boldsymbol{\Sigma}_1)^{-1}(\boldsymbol{\mu}_0 - \boldsymbol{\mu}_1)$$

```
>>> m0,m1=(np.mean(X0,axis=1),np.mean(X1,axis=1)) # vetores de média
>>> C0,C1=(np.cov(X0),np.cov(X1)) # matrizes de covariância
```

Transformação

```
>>> m=m0-m1;m=m[:,np.newaxis] # dif. entre médias: vetor de 784×1
>>> w=np.dot(np.linalg.inv(C0+C1),m)
LinAlgError:Singular Matrix
```

- Problema: $(\boldsymbol{\Sigma}_0 + \boldsymbol{\Sigma}_1)$ é uma matriz singular (o inverso dá erro no Python)

Análise em Discriminantes Lineares (LDA)

Exemplo: Discriminantes de Fisher (2 classes)

- Solução: pré-processar dados com PCA
  (Guardar componentes com valores próprios $> 10^{-10}$)

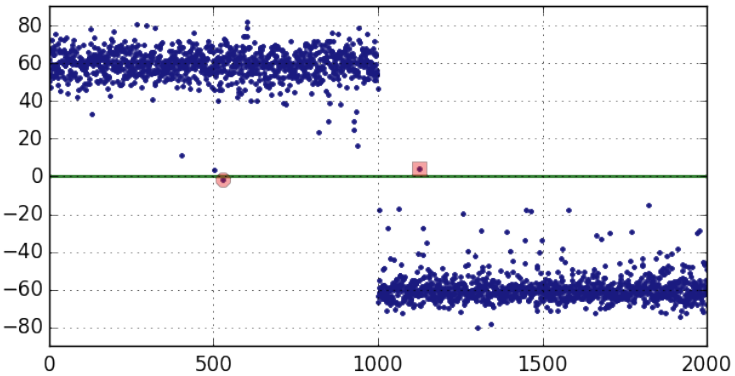
```
>>> Xn=(X.T-np.mean(X)).T # tirar média dos dados
>>> (v,V)=np.linalg.eig(np.cov(Xn)) # v valores próprios, V vetores próprios
>>> v=v.real;V=V.real# converter para real
>>> idx=np.argsort(-v)# -v para ordem decrescente
>>> v=v[idx];V=V[:idx]# ordenar valores e vetores
>>> V=V[:,v>=1e-10]# remover componentes com v≈ 0
>>> Y=np.dot(V.T,Xn)# projetar dados - 500 dimensões
```

- Usar os dados processados com PCA (os `Y`) para estimar $\mathbf{w}$.
  $\mathbf{w} \longrightarrow$ vetor de 500 dimensões
- Limiar óptimo (a meio): $\lambda = \frac{\mathbf{w}^T \boldsymbol{\mu}_0 + \mathbf{w}^T \boldsymbol{\mu}_1}{2}$

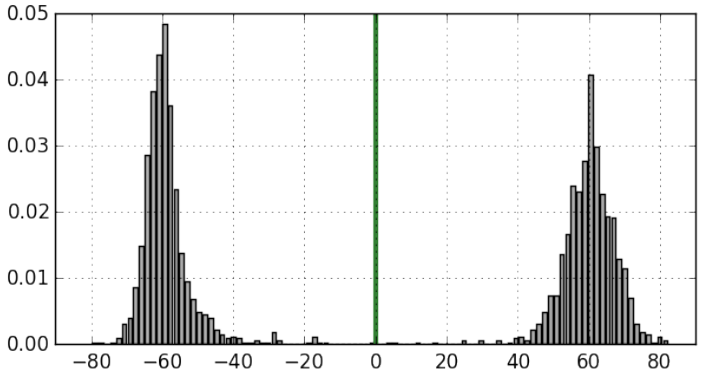
Análise em Discriminantes Lineares (LDA)

Exemplo: Discriminantes de Fisher (2 classes)

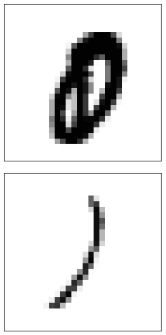
- Resultados – dados de treino



dados projetados



histograma


2 erros

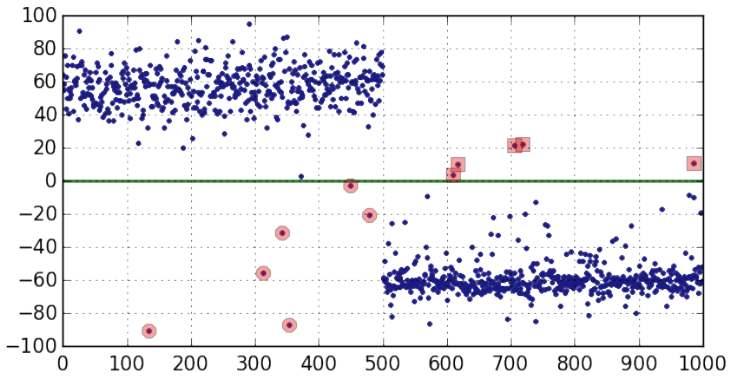
Análise em Discriminantes Lineares (LDA)

Exemplo: Discriminantes de Fisher (2 classes)

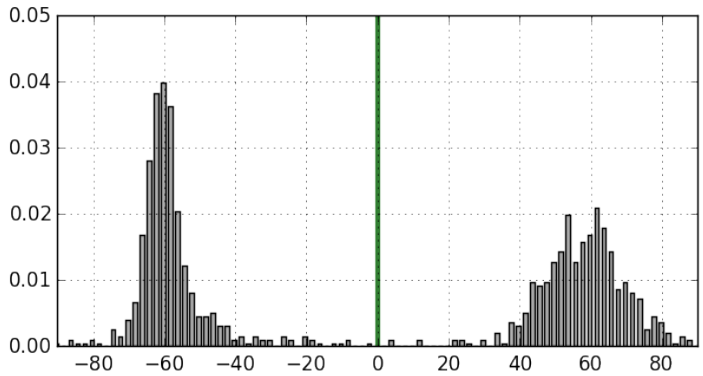
- Resultados – dados de teste

  Tirar média e aplicar transformação (vetor **w** - calculado com os dados de treino)

  Não re-calcular a transformação de Fisher!

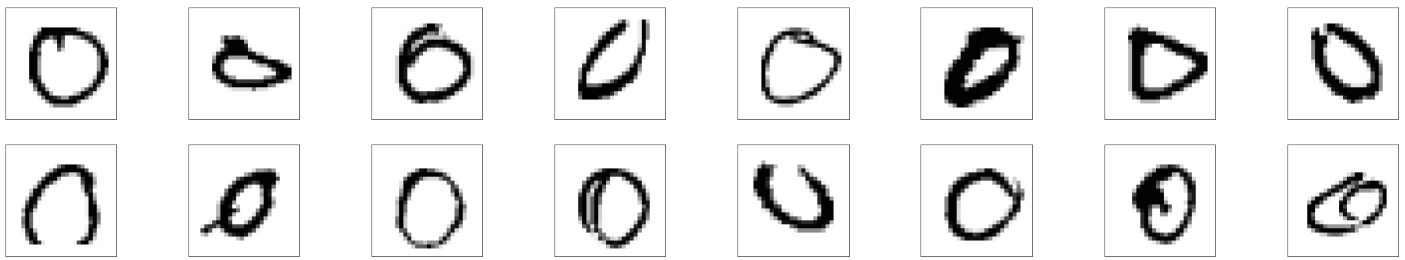


dados projetados



histograma

Total de 23 erros — 16 erros nas imagens dos 0_s



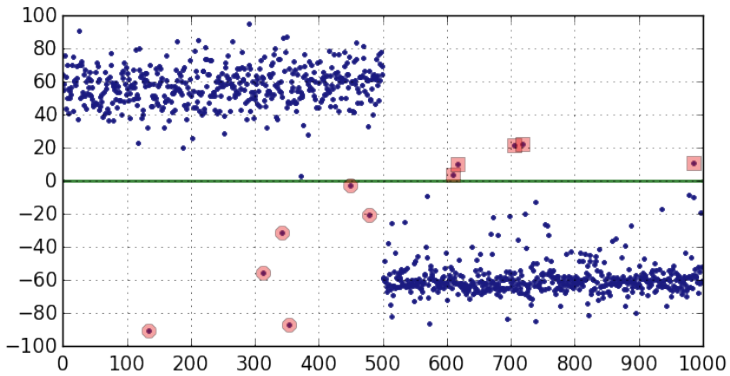
Análise em Discriminantes Lineares (LDA)

Exemplo: Discriminantes de Fisher (2 classes)

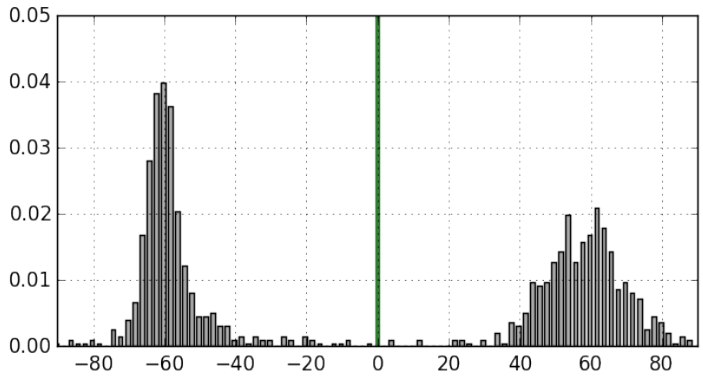
- Resultados – dados de teste

  Tirar média e aplicar transformação (vetor **w** - calculado com os dados de treino)

  Não re-calcular a transformação de Fisher!



dados projetados



histograma

Total de 23 erros — 7 erros nas imagens dos 1_s

Análise em Discriminantes Lineares (LDA)

LDA: Discriminantes de Fisher (*c* classes)

- $\Omega = \{\varpi_1, \ldots, \varpi_c\}$, com $N_i = |\mathbf{x} \in \varpi_i|$ e $i = 1, \ldots, c$
- Dados projetados num sub-espaço $\mathbf{y} = \mathbf{W}^\top \mathbf{x}$
  cada vetor $\mathbf{x}$ a d dimensões é noutro vetor $\mathbf{y}$ com o máximo de $c - 1$ dimensões

$$\mathbf{y} = \mathbf{W}^\top \mathbf{x} = \begin{bmatrix} w_{11} & w_{21} & \cdots & w_{d1} \\ \vdots & & \ddots & \vdots \\ w_{1(c-1)} & w_{2(c-1)} & \cdots & w_{d(c-1)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}$$

Análise em Discriminantes Lineares (LDA)

LDA: Discriminantes de Fisher (*c* classes)

- $\Omega = \{\varpi_1, \ldots, \varpi_c\}$, com $N_i = |\mathbf{x} \in \varpi_i|$ e $i = 1, \ldots, c$
- Dados projetados num sub-espaço $\mathbf{y} = \mathbf{W}^\top \mathbf{x}$ de modo a maximizar a função:

$$\mathcal{J}(\mathbf{W}) = \frac{\mathbf{W}^\top \mathbf{S}_\mu \mathbf{W}}{\mathbf{W}^\top \mathbf{\Sigma}_\Omega \mathbf{W}}$$

para $\mathbf{S}_\mu = \mathbf{S}_1 + \cdots + \mathbf{S}_c$ e $\mathbf{\Sigma}_\Omega = \mathbf{\Sigma}_1 + \cdots + \mathbf{\Sigma}_c$

- $\mathbf{S}_i = (\boldsymbol{\mu}_i - \boldsymbol{\mu}_\mathbf{x})(\boldsymbol{\mu}_i - \boldsymbol{\mu}_\mathbf{x})^\top$, com $\boldsymbol{\mu}_\mathbf{x} = \frac{1}{N} \sum_{\forall \mathbf{x}} \mathbf{x}$ e com $\boldsymbol{\mu}_i = \frac{1}{N_i} \sum_{\mathbf{x} \in \varpi_i} \mathbf{x}$
- $\mathbf{\Sigma}_i = \frac{1}{N_i - 1} \sum_{\mathbf{x} \in \varpi_i} (\mathbf{x} - \boldsymbol{\mu}_i)(\mathbf{x} - \boldsymbol{\mu}_i)^\top$
- $\mathbf{S}_\mu$ e $\mathbf{\Sigma}_\Omega$ matrizes de $d \times d$

Análise em Discriminantes Lineares (LDA)

LDA: Discriminantes de Fisher (*c* classes)

- $\Omega = \{\varpi_1, \ldots, \varpi_c\}$, com $N_i = |\mathbf{x} \in \varpi_i|$ e $i = 1, \ldots, c$
- Dados projetados num sub-espaço $\mathbf{y} = \mathbf{W}^\top \mathbf{x}$ de modo a maximizar a função:

$$\mathcal{J}(\mathbf{W}) = \frac{\mathbf{W}^\top \mathbf{S}_\mu \mathbf{W}}{\mathbf{W}^\top \mathbf{\Sigma}_\Omega \mathbf{W}}$$

- Solução: $\mathbf{W}$ matriz em que as colunas são os $c-1$ vetores próprios da matriz $\left(\mathbf{\Sigma}_\Omega^{-1} \mathbf{S}_b\right)$ associados aos $c-1$ valores próprios mais elevados.

Análise em Discriminantes Lineares (LDA)

Exemplo: Discriminantes de Fisher (*c* classes)

- Objectivo: Separar imagens de "0_s", "1_s", "2_s", "3_s" e "4_s"
- Dados pré-processados com PCA (removidas 180 dimensões supérfluas)
  (Este passo não é obrigatório, mas evita possíveis erros numéricos)
- Projecção calculada com dados de treino (5000 imagens total)
  Projeção: $y = \mathbf{W}^\top \mathbf{x}$
  $\mathbf{W}^\top$: matriz de 4×604

- Necessário calcular matrizes $\mathbf{S}_\mu$ e $\mathbf{\Sigma}_\Omega$
  - $\mathbf{S}_\mu = \mathbf{S}_0 + \mathbf{S}_1 + \mathbf{S}_2 + \mathbf{S}_3 + \mathbf{S}_4$

Cálculo de $\mathbf{S}_0$:

```
>>> mTot=np.mean(X,axis=1) #média global dos dados
>>> m0=np.mean(X[:,trueClass==0],axis=1) # m0: média da classe 0
>>> S0=np.dot((m0-mTot),(m0-mTot).T)
```

  - Repetir este processo para as matrizes das restantes classes
  - Somar todas as matrizes:

```
>>> STot=S0+S1+S2+S3+S4
```

Análise em Discriminantes Lineares (LDA)

Exemplo: Discriminantes de Fisher (*c* classes)

- Objectivo: Separar imagens de "0_s", "1_s", "2_s", "3_s" e "4_s"
- Dados pré-processados com PCA (removidas 180 dimensões supérfluas)
  (Este passo não é obrigatório, mas evita possíveis erros numéricos)
- Projecção calculada com dados de treino (5000 imagens total)
  Projeção: $y = \mathbf{W}^\top \mathbf{x}$
  $\mathbf{W}^\top$: matriz de 4×604
- Necessário calcular matrizes $\mathbf{S}_\mu$ e $\mathbf{\Sigma}_\Omega$
  - $\mathbf{\Sigma}_\Omega = \mathbf{\Sigma}_0 + \mathbf{\Sigma}_1 + \mathbf{\Sigma}_2 + \mathbf{\Sigma}_3 + \mathbf{\Sigma}_4$

Cálculo de $\mathbf{\Sigma}_0$:

```
>>> C0=np.cov(X[:,trueClass==0]) #matriz de covariância da classe 0
```

  - Calcular as matrizes de covariância das restantes classes
  - Somar todas as matrizes:

```
>>> CTot=C0+C1+C2+C3+C4
```

Análise em Discriminantes Lineares (LDA)

Exemplo: Discriminantes de Fisher (*c* classes)

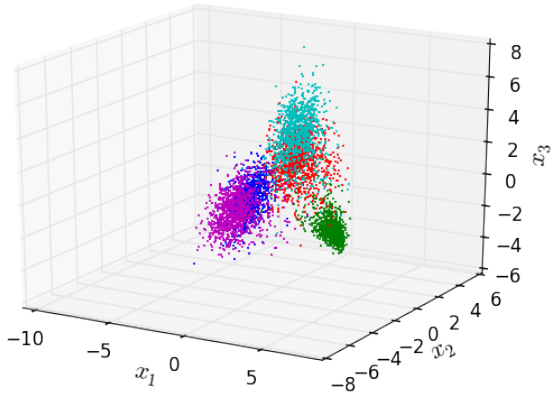
- Objectivo: Separar imagens de "0_s", "1_s", "2_s", "3_s" e "4_s"
- Dados pré-processados com PCA (removidas 180 dimensões supérfluas)
  (Este passo não é obrigatório, mas evita possíveis erros numéricos)
- Projecção calculada com dados de treino (5000 imagens total)
  Projeção: $y = \mathbf{W}^\top \mathbf{x}$
  $\mathbf{W}^\top$: matriz de 4×604

- Estimar projeção LDA (matriz $\mathbf{W}$)
  - Calcular matriz $\mathbf{M} = \left(\mathbf{\Sigma}_\Omega^{-1} \mathbf{S}_\mu\right)$
    ```
    >>> M=np.dot(np.linalg.inv(CTot),STot)
    ```
  - Calcular vetores e valores próprios da matriz $\mathbf{M}$
    ```
    >>> v,V=np.linalg.eig(M)
    ```
  - Limpar, e escolher os 4 primeiros vetores próprios
    ```
    >>> v=v.real
    >>> idx=np.argsort(-v) # ordenar por ordem decrescente
    >>> W=V[:,0:4].real # quatro 1º vetores próprios
    ```
  - Projetar dados:
    ```
    >>> Y=np.dot(W.T,X)
    ```

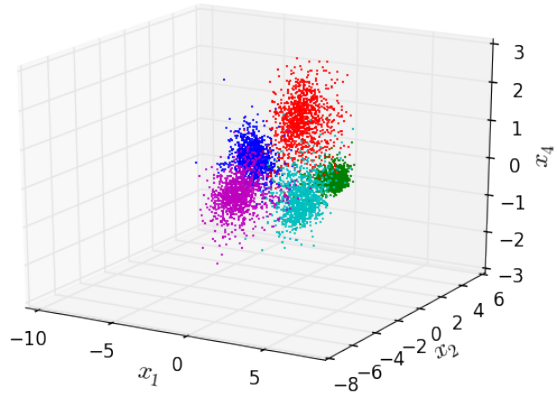
Análise em Discriminantes Lineares (LDA)

Exemplo: Discriminantes de Fisher (*c* classes)

- Objectivo: Separar imagens de "0_s", "1_s", "2_s", "3_s" e "4_s"
- Dados pré-processados com PCA (removidas 180 dimensões supérfluas)
  (Este passo não é obrigatório, mas evita possíveis erros numéricos)
- Projecção calculada com dados de treino (5000 imagens total)
  Projeção: $y = \mathbf{W}^\top \mathbf{x}$
  $\mathbf{W}^\top$: matriz de 4×604



três 1ªs dimensões dos dados projetados



1ª, 2ª e 4ª dimensão dos dados projetados

Análise em Discriminantes Lineares (LDA)

Comandos em `scikit-learn`:

- Carregar módulo LDA:
```
>>> from sklearn.discriminant_analysis import \
LinearDiscriminantAnalysis
```
- Instanciar objecto da classe LDA:
```
>>> lda=LinearDiscriminantAnalysis()
```
- Estimação do modelo para os dados
(atenção: recebe matrizes de $N \times d$, N nº pontos, d dimensão dos dados)
```
>>> lda.fit(X,trueClass) # X matriz de dados, trueClass classe dos dados
```
- Transformar dados com projeção LDA
```
>>> Xlda=lda.transform(X)
```

Análise em Discriminantes Lineares (LDA)

Comandos em `scikit-learn`:

- Ver parâmetros da classe `LinearDiscriminantAnalysis`:
  `solver, shrinkage, priors, n_components, store_covariance, tol`
- Ver métodos associados à classe:
  `decision_function()`
  `fit()`
  `fit_transform()`
  `get_params()`
  `predict()`
  `predict_log_proba()`
  `predict_proba()`
  `score()`
  `set_params()`
  `transform()`