ITA - PEDS

Machine Learning

- MSc. Adriano Henrique Rossette Leite (contato@adrianohrl.tech (mailto:contato@adrianohrl.tech))
- MSc. Túlio Lima S. M. Silva (tulio.madereira-silva@itau-unibanco.com.br (mailto:tulio.madereira-silva@itau-unibanco.com.br))
- Prof. Dr. Carlos Henrique Quartucci Forster (forster@ita.br (mailto:forster@ita.br))

Classificação da base de dados MNIST

O banco de dados MNIST de dígitos escritos à punho (disponível em: http://yann.lecun.com/exdb/mnist/ (http://yann.lecun.com/exdb/mnist/), acessado em: 12/07/2018) possui um conjunto para treinamento de 60000 exemplos e um conjunto para teste de 10000 de exemplos. Este é um subconjunto de um conjunto maior disponível em NIST. Os dígitos estão normalizados pelo tamanho e centralizados em um tamanho fixo de imagem.

Imagens Digitais

As imagens digitais podem ser representadas computacionalmente como matrizes. Cada elemento, denominado *pixel*, possui um valor numérico associado à intensidade de uma componente de cor particular. Desta forma, o número de linhas e de colunas da matriz equivalem, respectivamente, à altura e à largura da imagem. Assim, a intensidade de um dado *pixel* na imagem pode ser acessado a partir do número de linha e de coluna desejados. Como cada *pixel* possui tamanho de 1 *byte* (que se equivale à 8 *bits*), as intensidade de cores são valores compreendidos entre 0 e 255.

Existem diversos modelos de cores que podem ser utilizados para representar imagens coloridas. O modelo RGB, por exemplo, representa uma imagem digital colorida a partir de três componentes de cores vermelho (R), verde (G) e azul (B). Neste caso, a imagem digital é composta por três matrizes bidimensionais, uma matriz para cada componente de cor.

Uma imagem digital em níveis de cinza é representada por apenas uma matriz bidimensional, onde seus *pixels* possuem valores entre 0 e 255. Por um lado, quanto mais próximo de 0, o *pixel* se aproxima da cor preta e, por outro, quanto mais próximo de 255, o *pixel* se aproxima mais da cor branca. Os valores intermediários deste intervalo variam o nível de clareamento da cor cinza.

Outro tipo de imagem digital existente é a imagem digital binária. Este tipo de imagem é representado por matrizes lógicas, onde seus *pixels* podem ser pretos (valor de intensidade igual à 0) ou brancos (valor de intensidade igual à 1).

A representação de imagens digitais da base MNIST são em níveis de cinza.

Importação dos arquivos da base de dados MNIST

Primeiramente, faz se necessário definir as funções que importarão os dados da base MNIST corretamente. Para isso, a função *readMNIST* deve ser utilizada. Essa função possui apenas um único parâmetro de entrada, *filenames*. Este parâmetro de entrada é um dicionário cujas palavras-chaves são:

- *images*: identifica o caminho para o arquivo que contém as imagens 28 *pixels* e 28 *pixels* dos dígitos entre 0 e 9;
- e *labels*: identifica o caminho para o arquivo que contém os rótulos de cada imagem 28 *pixels* por 28 *pixels* domínio está necessariamente entre 0 e 9. Em ambos casos, considera-se que o formato do arquivo a ser importado segue a formatação IDX.

In [1]:
```
import struct as st
import numpy as np

def readMNIST(filenames):
    imgs = None
    with open(filenames['images'], 'rb') as imgs_file:
        imgs_file.seek(0)
        _, nimgs, nrows, ncols = st.unpack('>IIII', imgs_file.read(4 * 4))
        imgs = np.fromfile(imgs_file, dtype=np.uint8).reshape(nimgs, nrows * nc
        imgs_file.close()
    lbls = None
    with open(filenames['labels'], 'rb') as lbls_file:
        lbls_file.seek(0)
        _, nlbls = st.unpack('>II', lbls_file.read(4 * 2))
        lbls = np.fromfile(lbls_file, dtype=np.uint8)
        lbls_file.close()
    return imgs, lbls
```

Visualização de Imagens Digitais no *Python*

Sabendo que uma imagem digital é representada Note que as imagens importadas não são armazenadas como uma lista de imagens,

In [2]:
```
import matplotlib.pyplot as plt

height = 28
width = 28

def imshow(image, label='', cmap='gray', xlabel='', ylabel=''):
    plt.imshow(255 - image.reshape(height, width), cmap)
    plt.title(label)
    plt.xticks([])
    plt.xlabel(xlabel)
    plt.yticks([])
    plt.ylabel(ylabel)
```

Estrutura do projeto

Este projeto foi organizado da seguinte forma: sua pasta raiz possui três sub-pastas, *datasets*, *references* e *src*. A pasta *datasets* contém as bases de imagens e respectivos rótulos do MNIST para as fases treinamento e teste. A pasta *reference* contém materiais pertinentes para o estudo sobre o assunto deste projeto. A pasta *src* contém este arquivo Jupyter Notebook.

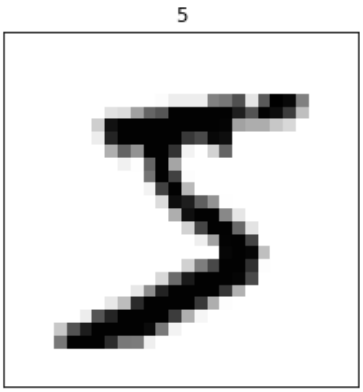
Importação das bases de treinamento e teste do MNIST

A seguir, é feita a leitura dos arquivos da base MNIST tanto das imagens e dos rótulos do treinamento quanto do teste, conforme descrito anteriormente.

In [3]:
```
train_filenames = {'images': '../datasets/train-images.idx3-ubyte', 'labels': '
test_filenames = {'images': '../datasets/test-images.idx3-ubyte', 'labels': '..

train_imgs, train_lbls = readMNIST(train_filenames)
test_imgs, test_lbls = readMNIST(test_filenames)
labels = set(train_lbls)
```

A fim de exemplificar a utilização do método de visualização das imagens carregadas, a primeira imagem do conjunto de treinamento é selecionada para ser visualizada, conforme se segue.

In [4]:
```
index = 0
image = train_imgs[index]
label = train_lbls[index]

imshow(image, label)
```



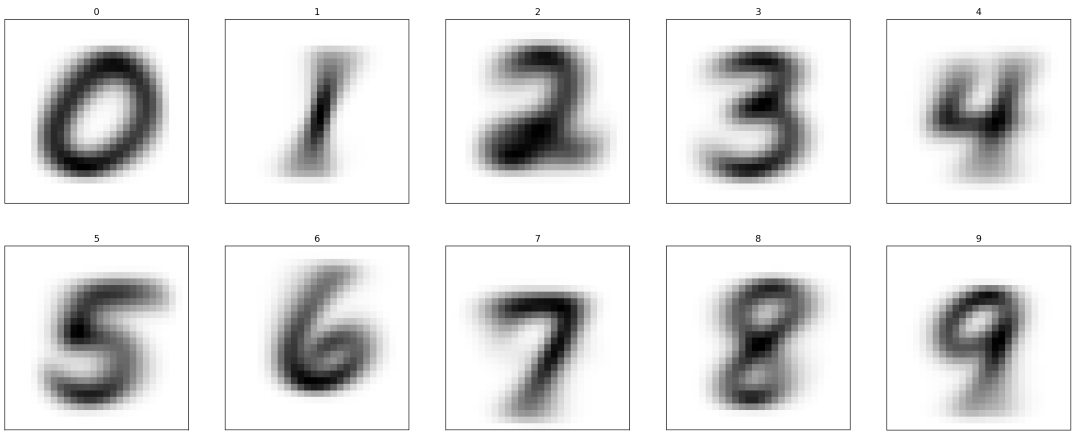
Cálculo das Imagens Médias para cada Rótulo

Foram calculadas as imagens médias para cada um dos rótulos.

In [5]:
```
mean_train_imgs = np.zeros((len(labels), width * height))
for i in range(len(labels)):
    mean_train_imgs[i, :] = np.mean(train_imgs[train_lbls == i, :], axis=0)
```

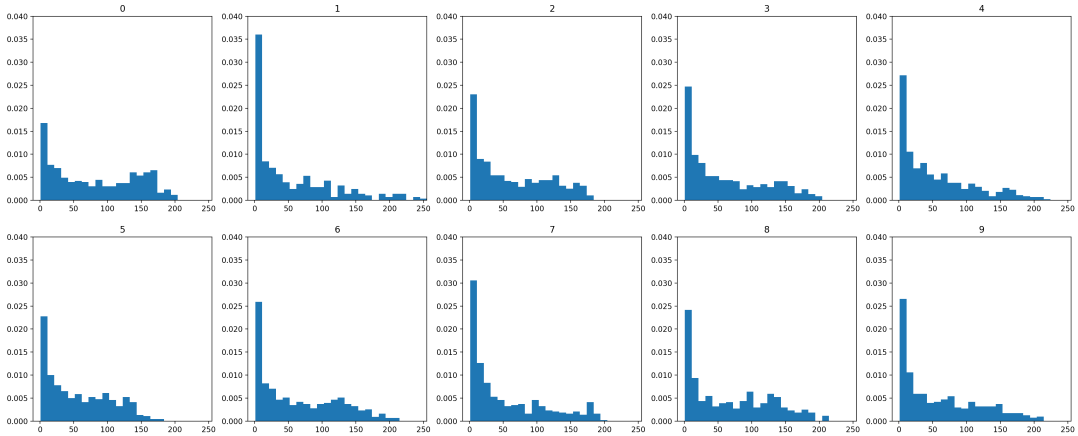
Visualizando as imagens médias de cada rótulo temos:

In [6]:
```
plt.figure(figsize=(25, 10), dpi=200)
for i in range(len(labels)):
    plt.subplot(2, len(labels) / 2, i + 1)
    imshow(mean_train_imgs[i, :], i)
plt.show()
```



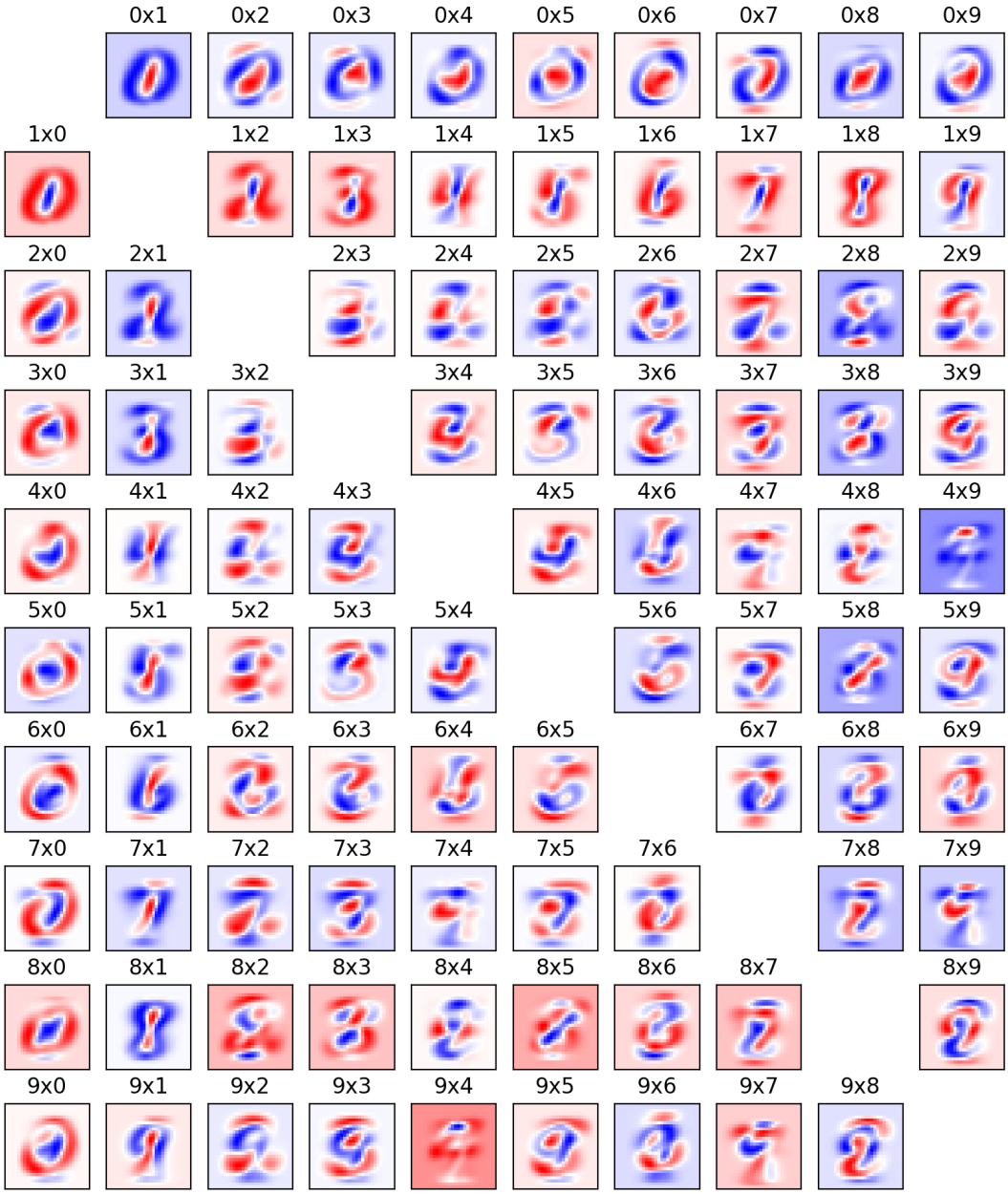
Seus respectivos histogramas são dados a seguir.

In [7]:
```
plt.figure(figsize=(25, 10), dpi=200)
for i in range(len(labels)):
    plt.subplot(2, len(labels) / 2, i + 1)
    plt.hist(mean_train_imgs[i, :], bins=25, range=(1, 255), density=True)
    plt.axis([-10, 255, 0, 0.04])
    plt.title(i)
plt.show()
```



Enfim, a comparação dois a dois das médias de cada rótulo.

In [8]:
```
plt.figure(figsize=(10, 12), dpi=200)
for i in range(len(labels)):
    for j in range(len(labels)):
        if i != j:
            plt.subplot(len(labels), len(labels), len(labels) * i + j + 1)
            image = mean_train_imgs[i, :] - mean_train_imgs[j, :]
            imshow(image, str(i) + 'x' + str(j), cmap='bwr')
plt.show()
```



Redução de Dimensionalidade

A seguir, serão discutidos os efeitos provocados pela aplicação dos algoritmos PCA e LDA ao conjunto de dados MNIST.

PCA

O PCA (do inglês, *Principal Component Analysis*), é um método de estatística multivariada muito utilizado para reduzir a dimensão do conjunto de dados mantendo a maior quantidade possível de informação nos dados transformados. O primeiro passo é obter uma matriz de covariância para dos dados, então supondo que temos um conjunto com p variáveis temos:

$$\Sigma = \begin{bmatrix} Var(X_1) & Cov(X_1, X_2) & \dots & Cov(X_1, X_p) \\ Cov(X_2, X_1) & Var(X_2) & \dots & Cov(X_2, X_p) \\ \vdots & \vdots & \ddots & \vdots \\ Cov(X_n, X_1) & Cov(X_p, X_2) & \dots & Var(X_p) \end{bmatrix}$$

onde:

$$Cov(X_i, X_j) = \frac{\sum X_i X_j - \sum X_i \cdot \sum X_j}{p}.$$

Observe que como $Cov(X_i, X_j) = Cov(X_j, X_i)$ a matriz Σ é simétrica, logo:

$$\Sigma = QDQ^T$$

sendo:

$$Q = [q_1, q_2, \cdots, q_p],$$

$$Q^T = \begin{bmatrix} q_1^T \\ q_2^T \\ \vdots \\ q_p^T \end{bmatrix}$$

e

$$D = \begin{bmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_p \end{bmatrix}$$

onde q_i é o autovetor, da matriz de covariâncias, correspondente ao auto valor λ_i, $i = 1, 2, \cdots, p$. Podemos então reescrever $\Sigma = \lambda_1 q_1 q_1^T + \lambda_2 q_2 q_2^T + \cdots + \lambda_p q_p q_p^T$, obserque que $q_i q_i^T$ são sempre matrizes de posto 1, logo, se ordenarmos de maneira decrescente os autovalores teremos os autovetores de maior peso na formação de Σ. Definindo $\lambda_{(i)}$ como o i-ésimo maior autovetor e $q_{(i)}$ como seu autovalor correspondente, tal que $q_i \geq q_j, \forall i, j = 1, \cdots, p$, a matriz de componentes será:

$$C_p = [q_{(1)}, q_{(2)}, \cdots, q_{(p)}].$$

Dada uma matriz de dados p-dimensionais, $\mathbf{X}_{n \times p}$, que desejamos reduzir para k dimensões basta multiplicarmos pela matriz formada pelas k primeiras colunas da matriz obtida anteriormente:

$$C_k = [q_{(1)}, q_{(2)}, \cdots, q_{(k)}],$$

In [5]:
```
from sklearn.decomposition import PCA

pca = PCA(n_components=10, svd_solver='auto')
pca = pca.fit(train_imgs)
print('Variância explicada pelo PCA: %.2f%%' % (100 * sum(pca.explained_varianc
```

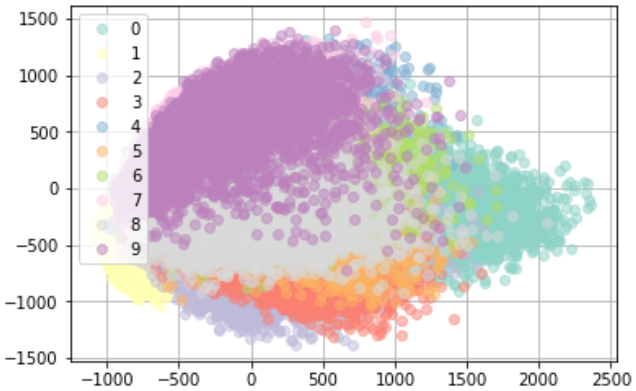
```
Variância explicada pelo PCA: 48.81%
```

Para que ambos os dados de treinamento e de teste sejam transformados pelo PCA já treinado, são necessário executar os seguintes comandos.

In [6]:
```
train_imgs_pca = pca.transform(train_imgs)
test_imgs_pca = pca.transform(test_imgs)
```

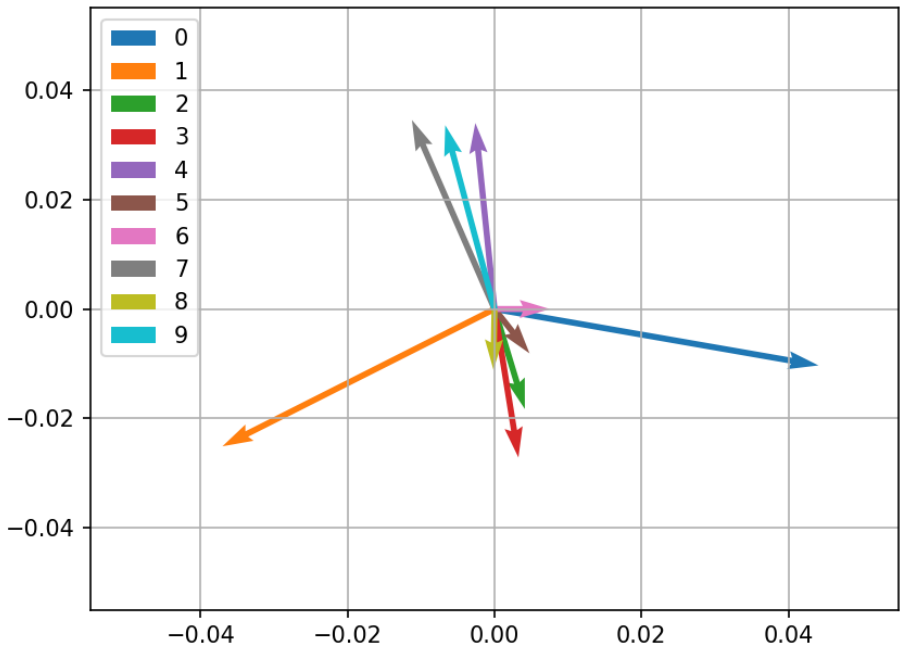
Ilustrando os 2 principais componentes providos pelo PCA, temos:

In [7]:
```
fig = plt.figure()
for i in range(len(labels)):
    i_train_imgs_pca = train_imgs_pca[train_lbls == i, :]
    plt.plot(i_train_imgs_pca[:, 0], i_train_imgs_pca[:, 1], marker='o', linest
plt.legend(labels, loc='upper left')
plt.grid(True)
plt.show()
```



Aplicando o PCA nas imagens médias de cada rótulo, ao considerar apenas os seus dois componentes principais pode-se compará-los vetorialmente, conforme visto na figura seguinte.

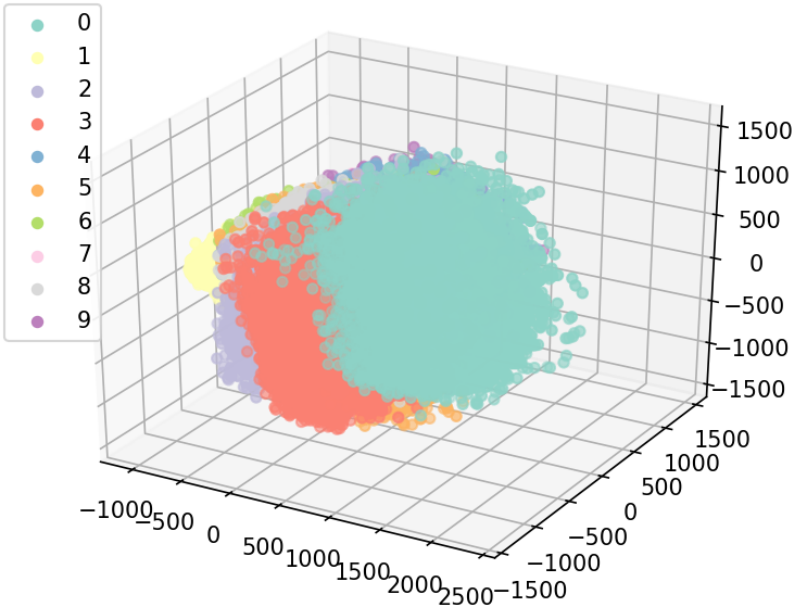
In [14]:
```
mean_train_imgs_pca = pca.transform(mean_train_imgs)

from mpl_toolkits.mplot3d import Axes3D
%matplotlib notebook

origin = [0, 0]
X, Y = zip(origin)
fig = plt.figure()
for i in range(len(labels)):
    U = (mean_train_imgs_pca[i, 0],)
    V = (mean_train_imgs_pca[i, 1],)
    plt.quiver(X, Y, U, V, color=plt.cm.tab10.colors[i], scale=2500)
plt.legend(labels, loc='upper left')
plt.grid(True)
plt.show()
```



Verifica-se que os dois componentes principais separam bem apenas as imagens que representam o algarismo 1. Os demais algarismos possuem certa similaridade.

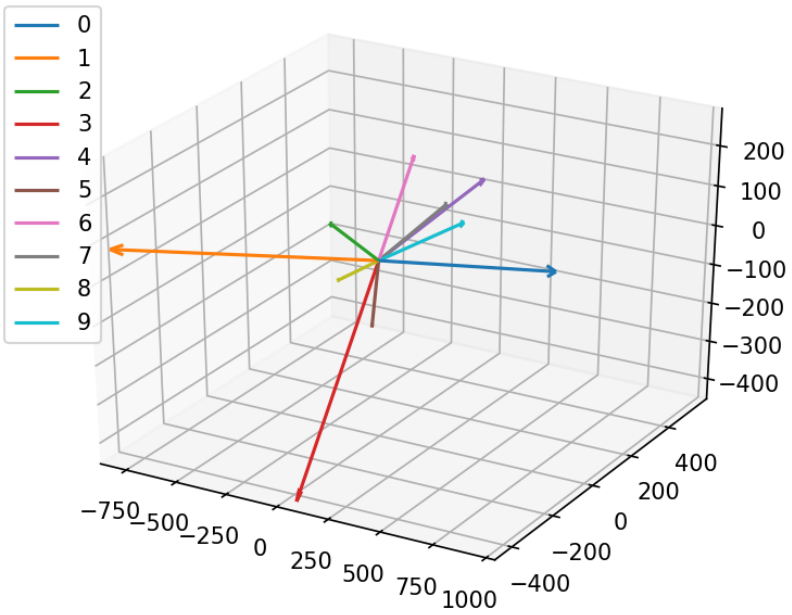
Semelhantemente, as três componentes principais providas pelo PCA podem ser visualizadas a seguir.

In [15]:
```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
for i in range(len(labels)):
    i_train_imgs_pca = train_imgs_pca[train_lbls == i, :]
    ax.scatter(i_train_imgs_pca[:, 0], i_train_imgs_pca[:, 1], i_train_imgs_pca
plt.legend(labels, loc='upper left')
plt.show()
```



Comparando, agora, as imagens médias com os 3 componentes principais, tem-se:

In [16]:
```
origin = [0, 0, 0]
X, Y, Z = zip(origin)
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
for i in range(len(labels)):
    U = (mean_train_imgs_pca[i, 0],)
    V = (mean_train_imgs_pca[i, 1],)
    W = (mean_train_imgs_pca[i, 2],)
    ax.quiver(X, Y, Z, U, V, W, color=plt.cm.tab10.colors[i], arrow_length_rati
ax.set_xlim([min(mean_train_imgs_pca[:, 0]), max(mean_train_imgs_pca[:, 0])])
ax.set_ylim([min(mean_train_imgs_pca[:, 1]), max(mean_train_imgs_pca[:, 1])])
ax.set_zlim([min(mean_train_imgs_pca[:, 2]), max(mean_train_imgs_pca[:, 2])])
plt.legend(labels, loc='upper left')
plt.show()
```



Através desta comparação vetorial, verifica-se que há uma separação bem melhor quando se considera as três principais componentes ao invés de apenas duas.

LDA

O LDA (acrônimo para *Linear Discriminant Analysis*) obtém a matriz de projeção $\mathbf{W}_{lda}$ que maximiza a separação entre as classes e minimiza a variabilidade dentro de cada classe. A matriz intra-classe, que quantifica a variabilidade dentro de cada classe, é definida como:

$$S_w = \sum_{i=1}^{g}(N_i - 1)S_i = \sum_{i=1}^{g}\sum_{j=1}^{N_i}(x_{i,j} - \bar{x}_i)(x_{i,j} - \bar{x}_i)^T$$

onde, $x_{i,j}$ é a j-ésima observação n-dimensional da i-ésima classe, N_i o número de observações da i-ésima classe e g a quantidade total de classes. É importante ressaltar que $\bar{X}_i$ e S_i são estimadores não viciados para as médias e as matrizes de covariância.

A matriz inter-classe, que quantifica a separação entre as classes, é definida como:

$$S_b = \sum_{i=1}^{g} N_i(\bar{x}_i - \bar{x})(\bar{x}_i - \bar{x})^T.$$

O objetivo da LDA é encontrar a matriz $\mathbf{W}_{lda}$ que maximiza a razão entre os determinantes das matrizes inter-classe e intra-classe:

$$W_{lda} = \arg\max_W \frac{|W^T S_b W|}{|W^T S_w W|}.$$

O critério de Fisher descrito acima é maximizado quando a matrix de projeção W_{lda} é composta pelos autovetores de $S_w^{-1}S_b$.

In [8]:
```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

lda = LinearDiscriminantAnalysis(n_components=8)
ldafit = lda.fit(train_imgs, train_lbls)
print('Variância explicada pelo LDA: %.2f%%' % (100 * sum(ldafit.explained_vari
```

```
/home/adrianohrl/anaconda3/lib/python3.6/site-packages/sklearn/discriminant_an
alysis.py:442: UserWarning: The priors do not sum to 1. Renormalizing
  UserWarning)

Variância explicada pelo LDA: 97.32%

/home/adrianohrl/anaconda3/lib/python3.6/site-packages/sklearn/discriminant_an
alysis.py:388: UserWarning: Variables are collinear.
  warnings.warn("Variables are collinear.")
```

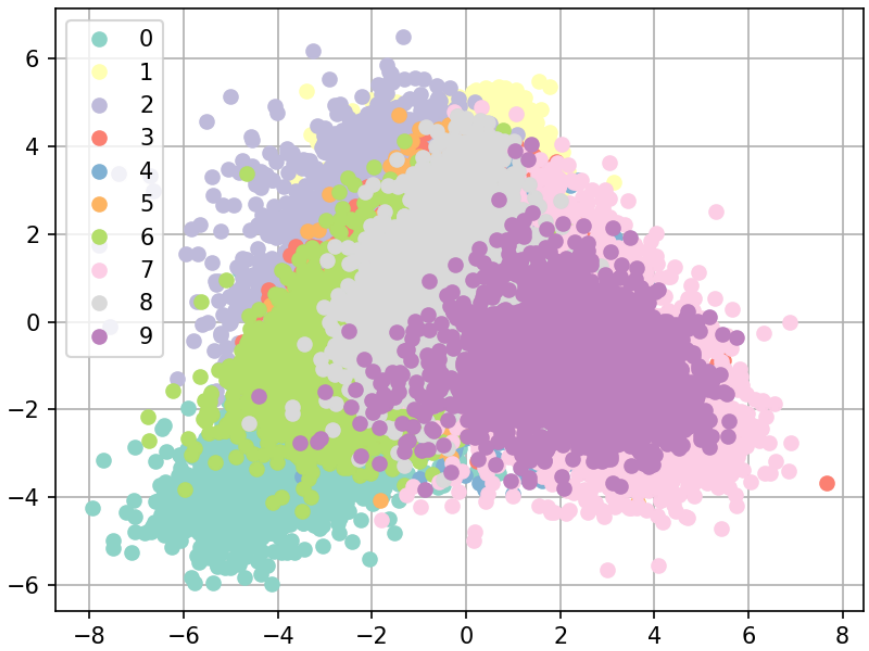
Para que ambos os dados de treinamento e de teste sejam transformados pelo LDA já treinado, são necessário executar os seguintes comandos.

In [9]:
```
train_imgs_lda = ldafit.transform(train_imgs)
test_imgs_lda = ldafit.transform(test_imgs)
```

Ilustrando os 2 principais componentes providos pelo LDA, temos:

In [143]:
```
fig = plt.figure()
for i in range(len(labels)):
    i_train_imgs_lda = train_imgs_lda[train_lbls == i, :]
    plt.plot(i_train_imgs_lda[:, 0], i_train_imgs_lda[:, 1], marker='o', linest
plt.legend(labels, loc='upper left')
plt.grid(True)
```

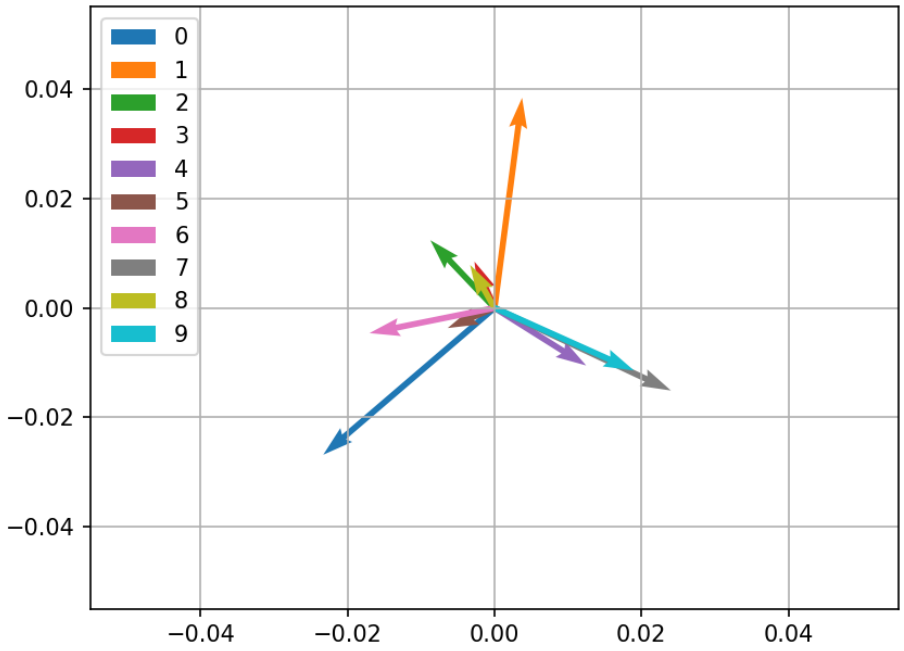
```
<IPython.core.display.Javascript object>
```



Aplicando o LDA nas imagens médias de cada rótulo, ao considerar apenas os seus dois componentes principais pode-se compará-los vetorialmente, conforme visto na figura seguinte.

In [154]:
```
mean_train_imgs_lda = ldafit.transform(mean_train_imgs)

origin = [0, 0]
X, Y = zip(origin)
fig = plt.figure()
for i in range(len(labels)):
    U = (mean_train_imgs_lda[i, 0],)
    V = (mean_train_imgs_lda[i, 1],)
    plt.quiver(X, Y, U, V, color=plt.cm.tab10.colors[i], scale=15)
plt.legend(labels, loc='upper left')
plt.grid(True)
plt.show()
```

```
<IPython.core.display.Javascript object>
```

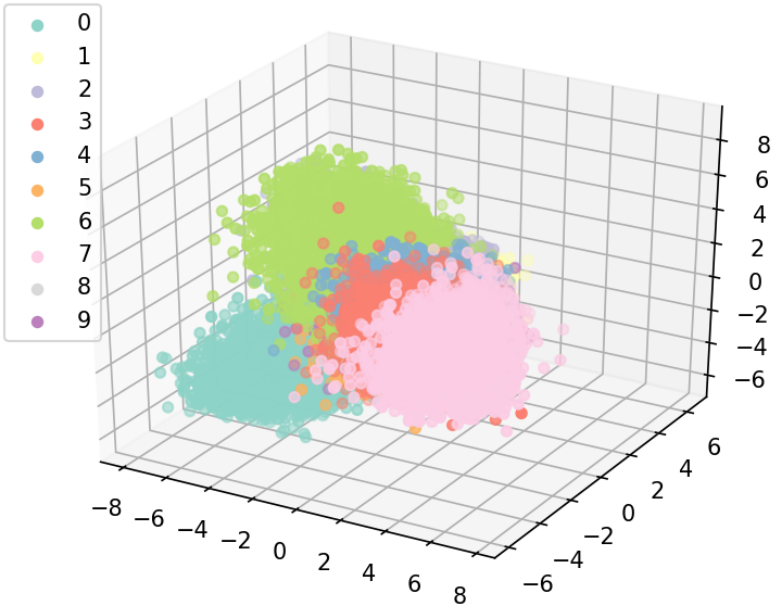


Verifica-se que os dois componentes principais orindos do LDA separam bem apenas as imagens que representam o algarismo 1 também. Os demais algarismos possuem certa similaridade.

Semelhantemente, as três componentes principais providas pelo LDA podem ser visualizadas a seguir.

In [158]:
```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
for i in range(len(labels)):
    i_train_imgs_lda = train_imgs_lda[train_lbls == i, :]
    ax.scatter(i_train_imgs_lda[:, 0], i_train_imgs_lda[:, 1], i_train_imgs_lda
plt.legend(labels, loc='upper left')
plt.show()
```

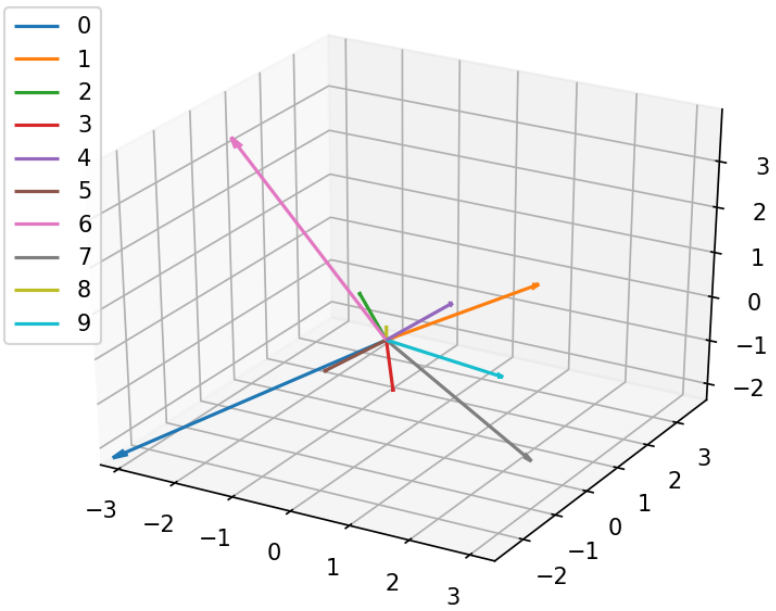
```
<IPython.core.display.Javascript object>
```



Verifica-se uma melhor separação dos dados ao utilizar as três componentes principais do LDA do que quando com o PCA. E isso é reforçado quando comparamos as imagens médias de cada rótulo veritorialmente, conforme mostra a figura a seguir.

In [187]:
```
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
for i in range(len(labels)):
    U = (mean_train_imgs_lda[i, 0],)
    V = (mean_train_imgs_lda[i, 1],)
    W = (mean_train_imgs_lda[i, 2],)
    ax.quiver(X, Y, Z, U, V, W, color=plt.cm.tab10.colors[i], arrow_length_rati
ax.set_xlim([min(mean_train_imgs_lda[:, 0]), max(mean_train_imgs_lda[:, 0])])
ax.set_ylim([min(mean_train_imgs_lda[:, 1]), max(mean_train_imgs_lda[:, 1])])
ax.set_zlim([min(mean_train_imgs_lda[:, 2]), max(mean_train_imgs_lda[:, 2])])
plt.legend(labels, loc='upper left')
plt.show()
```

```
<IPython.core.display.Javascript object>
```



Modelagem

Neste momento, deseja-se criar modelos que possam ser utilizados para que, dada uma imagem, este modelo possa classificá-la em um dos rótulos treinados.

Para isso, foram utilizados dois métodos supervisionados: *Árvore de Decisão* e *Random Forest*, os quais serão explanados a frente.

Árvore de Decisão

Árvores de decisão são modelos estatísticos que utilizam um treinamento supervisionado para a classificação e previsão de dados. Assim, este classificador necessita do conjunto de entrada e das suas respectivas saídas.

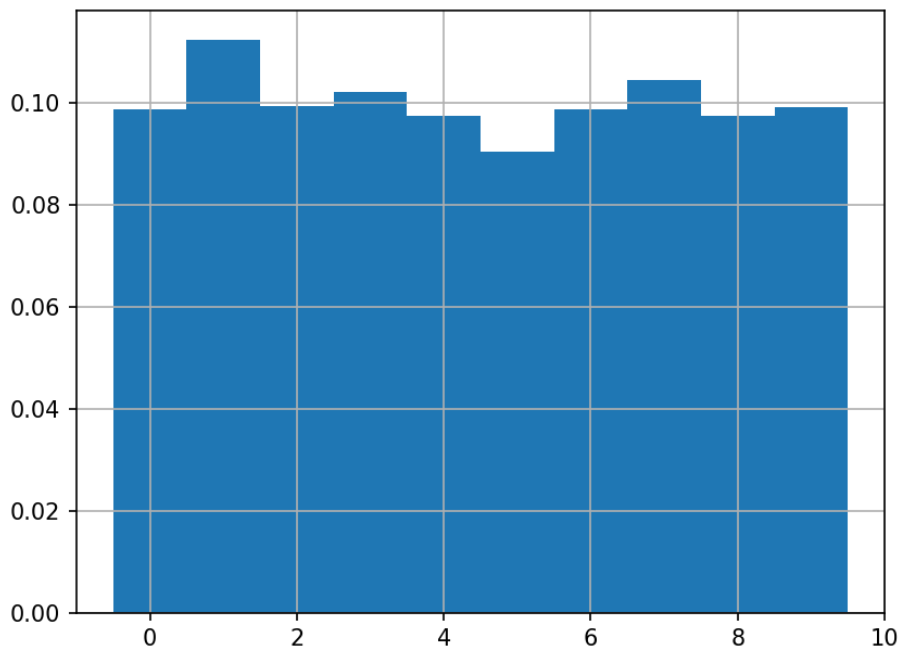
Estes modelos utilizam a estratégia de dividir para conquistar, isto é, um problema complexo é decomposto em sub-problemas mais simples e recursivamente esta técnica é aplicada a cada sub-problema.

Na árvore de decisão onde cada nó de decisão contém um teste para algum atributo, cada ramo descendente corresponde a um possível valor deste atributo, o conjunto de ramos são distintos, cada folha está associada a uma classe e, cada percurso da árvore, da raiz à folha corresponde uma regra de classificação.

Árvores de decisão podem criar árvores viesadas se o grupo de treino não estiver balanceado. Podemos verificar isso através de um histograma dos rótulos do conjunto de treino.

In [186]:
```
fig = plt.figure()
plt.hist(train_lbls, density=True, bins=range(11), align='left')
plt.grid(True)
plt.show()
```

```
<IPython.core.display.Javascript object>
```



Logo, o método Árvore de Decisão pode ser aplicado a este conjunto de dados de treinamento, pois este se encontra balanceado.

A biblioteca *sklearn* do *Python* disponibiliza o classificador de Árvore de Decisão. Para utilizar este método, são necessários as seguintes importações:

In [10]:
```
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import cross_val_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import recall_score
```

A seguir, é mostrado como se dá a utilização desta biblioteca. Para se utilizar o classificador Árvore de Decisão do *sklearn* corretamente, faz-se necessário executar duas etapas: (1) a instanciação do classificador conforme o critério desejado e (2) o treinamento do classificador. Após estas duas etapas, a árvore de decisão treinada está pronta para classificar.

Contudo, a fim de validar o modelo gerado, o conjunto de teste é classificado pelo modelo. Após esta etapa de predição, os respectivos rótulos do conjunto de teste são comparados com o resultado predito. A partir desta comparação, pode-se verificar a eficácia do modelo classificador a partir de uma matriz de confusão. Cada linha da matriz de confusão aponta para o rótulo real enquanto cada uma de suas colunas aponta para o rótulo predito.

Assim, o classificador Árvore de Decisão será aplicado à base de dados MNIST em formas diferentes.

Primeiramente, foi utilizado o classificador com o seu critério padrão, o de Gini. Assim que o classificador é instanciado, o conjunto de treinamento do MNIST de imagens com seus respectivos rótulos são utilizados para treiná-lo. Em seguida, o conjunto de dados de treinamento do MNIST são preditos pelo modelo treinado. E, então, a matriz de confusão é gerada.

In [11]:
```
tree = DecisionTreeClassifier()
treefit = tree.fit(train_imgs, train_lbls)
treepred = treefit.predict(test_imgs)
print(confusion_matrix(test_lbls, treepred))
```

```
[[ 916    1    9    8    5   12   12    4    7    6]
 [   0 1091   12    5    1    4    6    5    9    2]
 [  16   10  874   35   14   12   13   24   26    8]
 [   8    9   34  857    5   45    4    7   23   18]
 [   5    2   11    4  858    7   14    9   18   54]
 [  16    7    4   39    9  749   26    4   21   17]
 [  16    4   13   10   15   15  849    1   24   11]
 [   2   14   26   17    8    3    0  929    8   21]
 [  11    7   29   43   20   28   15   10  785   26]
 [  14    3    8   19   34   12    5   19   24  871]]
```

Analisando esta matriz de confução, verifica-se que:

In [13]:
```
print('Acurácia: %.2f%%' % (100 * treefit.score(test_imgs, test_lbls)))
i = 0
for recall in recall_score(test_lbls, treepred, average=None):
    print('Recall para ' + str(i) + ': %.2f%%' % (100 * recall))
    i += 1
```

```
Acurácia: 87.79%
Recall para 0: 93.47%
Recall para 1: 96.12%
Recall para 2: 84.69%
Recall para 3: 84.85%
Recall para 4: 87.37%
Recall para 5: 83.97%
Recall para 6: 88.62%
Recall para 7: 90.37%
Recall para 8: 80.60%
Recall para 9: 86.32%
```

Desta vez, o conjunto de treinamento do MNIST com seus respectivos rótulos são utilizados para treinar um classificador Árvore de Decisão com o critério de entropia. A partir do resultado da predição dos dados de teste do MNIST pelo novo classificador treinado, foi gerada uma nova matriz de confusão.

In [14]:
```
tree = DecisionTreeClassifier(criterion='entropy')
treefit= tree.fit(train_imgs, train_lbls)
treepred = treefit.predict(test_imgs)
print(confusion_matrix(test_lbls, treepred))
```

```
[[ 919    1    7    5    3   18    9    5    6    7]
 [   0 1097    5    4    3    9    1    3   11    2]
 [  12    5  911   21    9   13   14   19   21    7]
 [   5    8   30  857    4   42    6   14   20   24]
 [   7    4   16    5  861    7   14    8   16   44]
 [  13    6    3   43   10  748   20    1   26   22]
 [  17    4    8    7   22   20  861    6   10    3]
 [   2   10   34   15   14    6    2  922    8   15]
 [  11    8   21   29   20   21   13   10  817   24]
 [   8    8    9   19   35   15    5   16   21  873]]
```

In [15]:
```
print('Acurácia: %.2f%%' % (100 * treefit.score(test_imgs, test_lbls)))
i = 0
for recall in recall_score(test_lbls, treepred, average=None):
    print('Recall para ' + str(i) + ': %.2f%%' % (100 * recall))
    i += 1
```

```
Acurácia: 88.66%
Recall para 0: 93.78%
Recall para 1: 96.65%
Recall para 2: 88.28%
Recall para 3: 84.85%
Recall para 4: 87.68%
Recall para 5: 83.86%
Recall para 6: 89.87%
Recall para 7: 89.69%
Recall para 8: 83.88%
Recall para 9: 86.52%
```

Neste momento, utilizou-se um modelo Árvore de Decisão para classificar imagens da base MNIST a partir dos dados transformados pelo algoritmo PCA. Foi utilizado o critério de entropia para instanciar o classificador. Com as predições, foi gerada a seguinte árvore de confusão.

In [16]:
```
tree = DecisionTreeClassifier(criterion='entropy')# critério Entropia
treefit_pca= tree.fit(train_imgs_pca, train_lbls)
treepred_pca = treefit_pca.predict(test_imgs_pca)# prevendo valores do grupo te
print(confusion_matrix(test_lbls, treepred_pca))# contruindo matrix de confusão
```

```
[[ 853    1   10   12    7   43   29    4   10   11]
 [   3 1105    2    4    1    1    6    3    7    3]
 [  12    8  884   29   18   12   22   14   28    5]
 [  14    7   16  816    2   61    6   10   63   15]
 [   2    0   15    4  771   12   17   15   15  131]
 [  24    5    6   45    9  716   13    4   54   16]
 [  27    2   20    1   16   15  863    0    6    8]
 [   1   12   30   13   20    5    4  870   12   61]
 [  17    0   37   78   14   48    7   16  722   35]
 [   5    5    4   19  123   28    5   48   29  743]]
```

Analisando esta matriz de confução, verifica-se que:

In [17]:
```
print('Acurácia: %.2f%% (com PCA)' % (100 * treefit_pca.score(test_imgs_pca, te
i = 0
for recall in recall_score(test_lbls, treepred, average=None):
    print('Recall para ' + str(i) + ': %.2f%%' % (100 * recall))
    i += 1
```

```
Acurácia: 83.43% (com PCA)
Recall para 0: 93.78%
Recall para 1: 96.65%
Recall para 2: 88.28%
Recall para 3: 84.85%
Recall para 4: 87.68%
Recall para 5: 83.86%
Recall para 6: 89.87%
Recall para 7: 89.69%
Recall para 8: 83.88%
Recall para 9: 86.52%
```

Verifica-se que houve uma redução na acurácia do classificador ao considerar apenas os componentes principais.

Desta vez, um classificador Árvore de Decisão utilizando o critério de entropia foi treinado com os dados de treinamento do MNIST transformados pelo algoritmo LDA. Ao utilizar o conjunto de teste do MNIST também transformado pelo LDA, pode-se compara os rótulos reais com os preditos, obtendo a seguinte matriz de confusão.

In [18]:
```
tree = DecisionTreeClassifier(criterion='entropy')
treefit_lda= tree.fit(train_imgs_lda, train_lbls)
treepred_lda = treefit_lda.predict(test_imgs_lda)
print(confusion_matrix(test_lbls, treepred_lda))
```

```
[[ 928    0    5    4    3   14   14    3    6    3]
 [   0 1068   11    9    2    3    1    4   35    2]
 [  14   14  872   39   11   13   20   13   35    1]
 [   4    5   32  845    4   57    4   15   34   10]
 [   2    5   11    2  862    3   13    9   13   62]
 [  13    4    9   63   15  673   16   15   72   12]
 [  21    3   18    2   22   20  861    1   10    0]
 [   1   14   23   23    5    6    1  890    9   56]
 [  11   25   24   32   20   63   16   10  744   29]
 [   6    5    6   13   82   17    1   60   18  801]]
```

Analisando esta matriz de confução, verifica-se que:

In [19]:
```
print('Acurácia: %.2f%% (com LDA)' % (100 * treefit_lda.score(test_imgs_lda, te
i = 0
for recall in recall_score(test_lbls, treepred, average=None):
    print('Recall para ' + str(i) + ': %.2f%%' % (100 * recall))
    i += 1
```

```
Acurácia: 85.44% (com LDA)
Recall para 0: 93.78%
Recall para 1: 96.65%
Recall para 2: 88.28%
Recall para 3: 84.85%
Recall para 4: 87.68%
Recall para 5: 83.86%
Recall para 6: 89.87%
Recall para 7: 89.69%
Recall para 8: 83.88%
Recall para 9: 86.52%
```

Verifica-se que houve uma redução na acurácia do classificador ao considerar apenas os componentes principais. Entretanto, a acurácia quando utilizando o LDA foisuperior do que quando utilizando o PCA.

Random Forest

O método *Random Forest* é um algoritmo supervisionado de aprendizado. Este algorítmo cria um conjunto de Árvores de Decisão e combina seus resultados para aumentar sua acurácia.

A biblioteca *sklearn* do *Python* disponibiliza o classificador de *Random Forest*. Para utilizar este método, é necessária a seguinte importação:

In [20]:
```
from sklearn.ensemble import RandomForestClassifier
```

Semelhantemente à Árvore de Decisão, para se utilizar um classificador *Random Forest* é necessário instanciar um classificador escolhendo o critério desejado e, em seguida, faz-se necessário treinar o modelo classificador com um conjunto de entradas e suas respectivas saídas.

A eficácia do modelo pode ser obtida a partir da comparação entre os valores reais do conjunto de teste com os valores preditos a partir do classificador.

Logo, este classificador foi utilizado diversas vezes em configurações distintas para se especializar no problema da base MNIST.

Primeiramente, este classificador foi utilizado com o conjunto de treino do MNIST com seus respectivos rótulos. A partir do modelo treinado, o conjunto de teste foi predito por ele e, então, o seu resultado de predição foi comparado em uma matriz de confusão com os rótulos reais.

In [21]:
```
forest = RandomForestClassifier()
forestfit = forest.fit(train_imgs, train_lbls)
forestpred = forestfit.predict(test_imgs)
print(confusion_matrix(test_lbls, forestpred))
```

```
[[ 965    1    0    0    2    3    4    1    3    1]
 [   1 1116    2    5    1    3    2    1    3    1]
 [   8    0  981    8    7    2    7    9    8    2]
 [   1    0   19  945    0   21    1   12    9    2]
 [   2    1    2    2  940    2    8    2    3   20]
 [  12    3    3   31    8  809    9    1   11    5]
 [   9    4    2    2   11   11  919    0    0    0]
 [   1   10   22    1    2    2    0  969    6   15]
 [   4    1   11   21    9   20    5    5  886   12]
 [   7    7    6   17   25    5    1   11    5  925]]
```

Analisando esta matriz de confução, verifica-se que:

In [22]:
```
print('Acurácia: %.2f%%' % (100 * forestfit.score(test_imgs, test_lbls)))
i = 0
for recall in recall_score(test_lbls, forestpred, average=None):
    print('Recall para ' + str(i) + ': %.2f%%' % (100 * recall))
    i += 1
```

```
Acurácia: 94.55%
Recall para 0: 98.47%
Recall para 1: 98.33%
Recall para 2: 95.06%
Recall para 3: 93.56%
Recall para 4: 95.72%
Recall para 5: 90.70%
Recall para 6: 95.93%
Recall para 7: 94.26%
Recall para 8: 90.97%
Recall para 9: 91.67%
```

Nota-se que a acurácia do classificador *Random Forest* é superior à acurácia do classificador Árvore de Decisão para o problema MNIST.

Semelhantemente, a mesma configuração foi realizada com a exceção de que o critério escolhido para a geração do modelo foi a entropia.

In [23]:
```
forest = RandomForestClassifier(criterion='entropy')
forestfit = forest.fit(train_imgs, train_lbls)
forestpred = forestfit.predict(test_imgs)
print(confusion_matrix(test_lbls, forestpred))
```

```
[[ 970    0    0    0    0    1    4    1    3    1]
 [   0 1123    2    4    0    1    2    0    2    1]
 [  13    0  982    4    2    1    8   16    6    0]
 [   3    0   15  948    1   15    1   10   15    2]
 [   2    1    8    3  926    1    7    2    8   24]
 [   8    1    4   21    6  829   10    3    7    3]
 [  13    2    3    0    8    4  927    0    1    0]
 [   3    7   24    4    8    0    0  973    0    9]
 [   8    0   10   17    5   16    9    3  896   10]
 [   5    6    5   10   21    4    1    9    6  942]]
```

Analisando esta matriz de confução, verifica-se que:

In [24]:
```
print('Acurácia: %.2f%%' % (100 * forestfit.score(test_imgs, test_lbls)))
i = 0
for recall in recall_score(test_lbls, forestpred, average=None):
    print('Recall para ' + str(i) + ': %.2f%%' % (100 * recall))
    i += 1
```

```
Acurácia: 95.16%
Recall para 0: 98.98%
Recall para 1: 98.94%
Recall para 2: 95.16%
Recall para 3: 93.86%
Recall para 4: 94.30%
Recall para 5: 92.94%
Recall para 6: 96.76%
Recall para 7: 94.65%
Recall para 8: 91.99%
Recall para 9: 93.36%
```

Houve uma pequena redução na acurácia do classificador *Random Forest* nesta configuração.

Novamente, o classificador *Random Forest* foi utilizado para classificar a base MNIST. Entretanto, apenas os principais componentes oriundos do PCA foram utilizados desta vez.

In [25]:
```
forest = RandomForestClassifier()
forestfit = forest.fit(train_imgs_pca, train_lbls)
forestpred = forestfit.predict(test_imgs_pca)
print(confusion_matrix(test_lbls, forestpred))
```

```
[[ 924    0    9    4    1   24   14    2    1    1]
 [   0 1116    4    3    0    2    3    0    6    1]
 [  19    2  949   13    9    5   10    9   16    0]
 [   4    1   18  902    1   27    3    9   39    6]
 [   2    0    9    2  839    6   13   10    2   99]
 [  24    1    8   44   12  767    8    2   18    8]
 [  18    6    7    1    7   17  900    0    1    1]
 [   2   13   22    6    8    2    0  929    7   39]
 [  11    2   26   62   10   39    5   12  789   18]
 [   5    9    5   13  105   11    5   38   13  805]]
```

Analisando esta matriz de confução, verifica-se que:

In [26]:
```
print('Acurácia: %.2f%% (com PCA)' % (100 * forestfit.score(test_imgs_pca, test
i = 0
for recall in recall_score(test_lbls, forestpred, average=None):
    print('Recall para ' + str(i) + ': %.2f%%' % (100 * recall))
    i += 1
```

```
Acurácia: 89.20% (com PCA)
Recall para 0: 94.29%
Recall para 1: 98.33%
Recall para 2: 91.96%
Recall para 3: 89.31%
Recall para 4: 85.44%
Recall para 5: 85.99%
Recall para 6: 93.95%
Recall para 7: 90.37%
Recall para 8: 81.01%
Recall para 9: 79.78%
```

Verifica-se que houve uma redução na acurácia do classificador ao considerar apenas os componentes principais. Porém, a acurácia do classificador nesta configuração (*Random Forest* com PCA) foi superior à acurácia de um classificador Árvore de Decisão com LDA.

Dando continuidade aos experimentos, o conjunto de treino do MNIST transformado pelo LDA foi utilizado para treinar um classificador *Random Forest* com o critério de entropia.

In [27]:
```
forest = RandomForestClassifier(criterion='entropy')
forestfit = forest.fit(train_imgs_lda, train_lbls)
forestpred = forestfit.predict(test_imgs_lda)
print(confusion_matrix(test_lbls, forestpred))
```

```
[[ 954    0    3    3    1    8    7    3    1    0]
 [   0 1102    5    3    1    4    3    1   16    0]
 [  14    7  928   17   11    7   11   13   21    3]
 [   3    3   24  882    0   47    1   18   25    7]
 [   0    2    6    2  916    2   10    4    9   31]
 [  11    3    5   51   10  746   12   13   35    6]
 [  16    4   13    2   12   23  884    0    4    0]
 [   3   13   19   11   10    0    1  927    3   41]
 [   8   23   14   32   15   47   14   12  800    9]
 [   9    3    3   10   73    7    2   36   12  854]]
```

Analisando esta matriz de confução, verifica-se que:

In [28]:
```
print('Acurácia: %.2f%% (com LDA)' % (100 * forestfit.score(test_imgs_lda, test
i = 0
for recall in recall_score(test_lbls, forestpred, average=None):
    print('Recall para ' + str(i) + ': %.2f%%' % (100 * recall))
    i += 1
```

```
Acurácia: 89.93% (com LDA)
Recall para 0: 97.35%
Recall para 1: 97.09%
Recall para 2: 89.92%
Recall para 3: 87.33%
Recall para 4: 93.28%
Recall para 5: 83.63%
Recall para 6: 92.28%
Recall para 7: 90.18%
Recall para 8: 82.14%
Recall para 9: 84.64%
```

Verifica-se que houve uma redução na acurácia do classificador ao considerar apenas os componentes principais. Contudo, a acurácia do classificador *Random Forest* com LDA é maior do que com o PCA. Na verdade, este aumento na acurácia foi muito pequeno.

KNN

O método KNN (do inglês, K Nearest Neighbors) utiliza toda o conjunto de dados de treinamento na representação do modelo. Por esse motivo, recomenda-se que o conjunto de treinamento seja melhorado constantemente, removendo dados errados e *outliers*, para que este permaneça sempre consistente.

A eficiência deste algoritmo pode aumentar significamente através do armazenamento dos dados usando estruturas de dados copmlexas como árvores k-d para fazer com que a busca por padrões na etapa de predição seja eficiente.

A predição de novas entradas é realizada através da busca através de todo o conjunto de treinamento pelas k instâncias (vizinhos) mais similares e, então, resumindo a variável de saída através dessas k instâncias.

Para determinar quais são as k instâncias dentro do conjunto de treinamento que são mais similares à nova entrada é utilizada a distância euclidiana. O cálculo de Hamming, Manhattan e Minkowski também são bastante utilizados para este propósito.

A biblioteca sklearn do Python disponibiliza o classificador de KNN. Para utilizar este método, é necessária a seguinte importação:

In [29]:
```
from sklearn.neighbors import KNeighborsClassifier
```

De igual modo, um classificador KNN também é definido pelas etapas de instanciação e treinamento. Após esta última etapa, o classificador pode ser validado a partir da predição do conjunto de teste e da comparação dos seus valores reais com os valores preditos através de uma matriz de confusão.

Assim, um classificador KNN foi treinado com o conjunto de treinamento das imagens e seus respectivos rótulos da base MNIST. Em seguida, o conjunto de teste do MNIST foi utilizado para comparar seus valores reais com os valores preditos pelo modelo treinado.

In [*]:
```
knn = KNeighborsClassifier()
knnfit = knn.fit(train_imgs, train_lbls)
knnpred = knnfit.predict(test_imgs)
print(confusion_matrix(test_lbls, knnpred))
```

Analisando esta matriz de confução, verifica-se que:

In [*]:
```
print('Acurácia: %.2f%%' % (100 * knnfit.score(test_imgs, test_lbls)))
i = 0
for recall in recall_score(test_lbls, knnpred, average=None):
    print('Recall para ' + str(i) + ': %.2f%%' % (100 * recall))
    i += 1
```

Este foi o modelo que obteve a maior acurácia. Entretanto, a etapa de predição é bastante demorada.

Conclusão

O LDA apresentou uma melhor transformação do *dataset* do que o PCA, pois houve uma separação maior entre as classes.

O modelo KNN apresentou a melhor acurácia. Entretanto, sua função de predição é demasiadamente demorada. Para aumentar sua velocidade, pode-se aplicar uma transformação pelo PCA ou LDA ao conjunto de dados para que apenas os seus componentes principais sejam utilizados. Desta forma, o número de dimensões será diminuido.

O modelo *Random Forest* apresentou uma boa acurácia. Sua acurácia foi melhor do que o modelo do classificador Árvore de Decisão treinado.