APRENDIZADO DE MÁQUINA - TRABALHO 02

Jefferson Colares

Análise e conclusões sobre o código Python disponibilizado para este trabalho.

Professor:

Eduardo Bezerra

Rio de Janeiro,

Julho, 2018

Sumário

1 Agrupameto **2**

1.1 Implementando K-means 2

1.2 Encontrando centróides mais próximos 2

1.3 Atualização dos centróides 4

1.4 K-means aplicado ao conjunto de dados de exemplo 6

1.5 Inicialização aleatória 8

2 Redução de Dimensionalidade **10**

2.1 Conjunto de dados de exemplo 10

2.2 Implementando o PCA 10

2.3 Redução de Dimensionalidade com PCA 12

2.3.1 Projetando os dados nos componentes principais 12

2.3.2 Reconstruindo uma aproximação dos dados 12

2.3.3 Visualizando as projeções 14

3 Detecção de Anomalias **15**

3.1 Distribuição Gaussiana 15

3.2 Estimativa de parâmetros para uma gaussiana 15

3.3 Selecionando ϵ 17

Parte 1 - Agrupameto

1.1 Implementando K-means

O K-means consiste em duas etapas, executadas repetidamente:

1. identificar e atribuir a cada item o centroide mais proximo.
2. mover o centroide para o centro (media) dos itens a ele atribuídos.

1.2 Encontrando centróides mais próximos

Nessa item do exercício, é solicitado completar o código da função **find_closest_centroids()**, que localiza o centroide mais próximo a cada ponto do conjunto de dados. Para isso, criei uma nova função distancia() que, dadas as coordenadas do ponto e de um centroide, calcula a distância euclidiana entre eles. O código da função é o que segue:

```
def distancia(x, centroide):
        dx = x[0] - centroide[0]
        dy = x[1] - centroide[1]
        d = np.sqrt(dx **2 + dy **2)
        return(d)
```

Minha implementação da função **find_closest_centroids()** percorre todos os pontos da array X, calculando a distância entre o ponto e cada um dos K centroides. O número do centroide de menor distância é armazenado na array idx, na posição correspondente à do ponto em X.

```
def find_closest_centroids(X, centroids):
        K = np.size(centroids, 1)
        idx = np.zeros((len(X), 1), dtype=np.int8)
        ###################################################
        # SEU CODIGO AQUI
        K = len(centroids)
        dist = np.zeros((len(X), 1))
        for i in range(len(idx)):
                #obtem a distancia entre o ponto e o primeiro centroide:
                dist[i] = dist_nova = distancia(X[i],centroids[0])
                idx[i] = 0
                #executa os passos abaixo para os K centroides:
```

```
            for j in range(K):
                #calcula a distancia entre o ponto e o proximo centroide:
                dist_nova = distancia(X[i],centroids[j])
                #se a nova distancia obtida for menor que a atual
                if dist_nova < dist[i]:
                    #armazena a nova menor distancia:
                    dist[i] = dist_nova
                    #armazena o id do centroide atual em idx
                    idx[i] = j
    ######################################################
    return idx
```

A execução da função **main()** fornece como resultados:

- a lista dos clusters atribuídos aos 3 primeiros elementos de X:

```
Cluster assignments for the first, second and third examples: [0 2 1]
```

- a figura abaixo mostrando através de cores a atribuição dos elementos de X a cada cluster

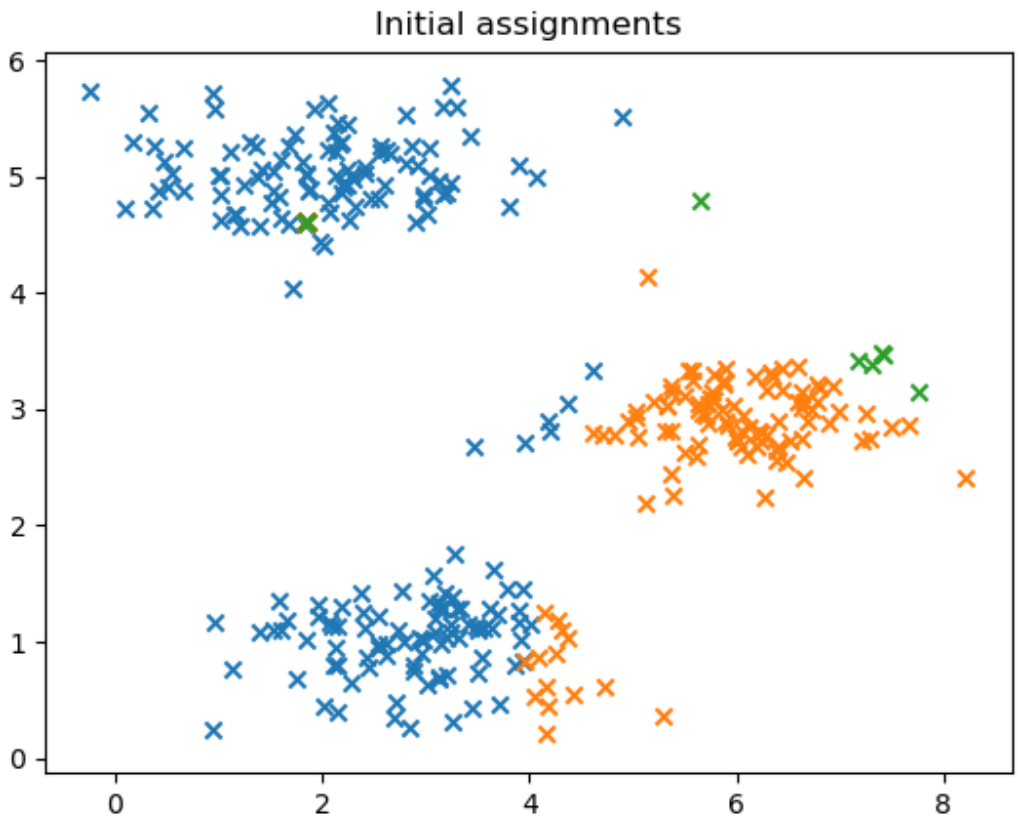


Figura 1.1: Atribuições iniciais.

1.3 Atualização dos centróides

A segunda parte do algoritimo calcula a média dos pontos atribuídos a cada cluster. Para fazer isso, é necessário passar por todos os pontos, somando seus valores caso pertençam a um ou outro cluster e calcular as médias. Para evitar o uso de loops e tornar o processamento mais rápido, a separação dos elementos de cada cluster e o cálculo da média foram feitos através de operações vetorizadas. Conforme pode ser visto no código abaixo, um único loop foi utilizado. A separação dos elementos pertencentes a cada cluster e o cálculo das respectivas médias foram realizadas usando arrays do Numpy.

```
def compute_centroids(X, idx, K):
        centroids = np.zeros((K,np.size(X,1)))
        ##################################################
        # SEU CODIGO AQUI
        for i in range(K):
                #concatena a atribuicao de clusters e X
                idxX = np.concatenate( (idx ,X ), axis = 1)
                #cria uma array booleana que identifica os elementos de X que pertencem ao cluste
                ind = (idxX[:,0] == i)
                #cria uma array contendo apenas os elementos de X que pertencem ao cluster k
                C = (X[ind])
                #calcula a media de x e y
                meanx = np.mean(C[:,0], axis =0 )
                meany = np.mean(C[:,1], axis =0 )
                # atribui as medias obtidas \'as coordenadas do centroide
                centroids[i] = np.array([meanx, meany])
        ##################################################
        return centroids
```

O script main.py executa alternadamente as duas funções implementadas acima (**find_closest_centroids()** e **compute_centroids()**). O resultado após 10 repetições é exibido na tela:

```
Centroids after the 1st update:
[[1.95399466 5.02557006]
 [3.04367119 1.01541041]
 [6.03366736 3.00052511]]
```

Como resultado, também é gerado o gráfico a seguir:

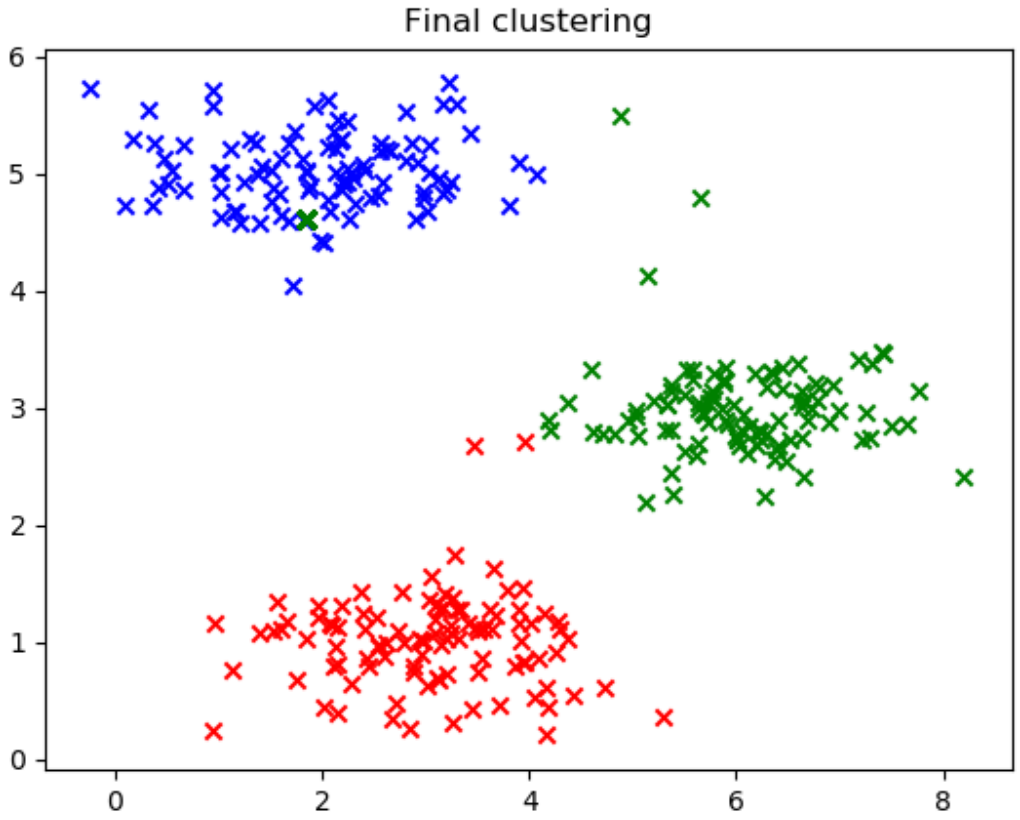


Figura 1.2: Agrupamento final.

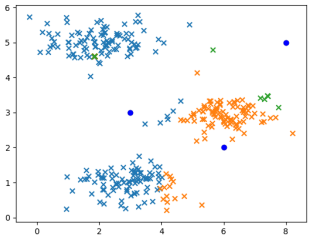
1.4 K-means aplicado ao conjunto de dados de exemplo

O K-means é um algoritimo iterativo. As funções implementadas acima precisam ser executadas sucessivamente até que os centroides estejam posicionados centralizadamente em relação aos pontos e que os pontos estejam atribuidos aos clusteres mais proximos. A quantidade de repetições é determinada pela função run_kmeans(), bem como a plotagem dos resultados parciais. A função é reproduzida abaixo, com meus comentários sobre seu funcionamento:

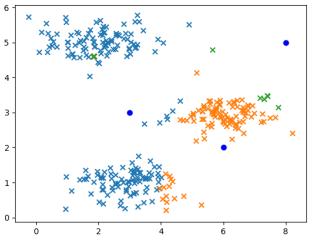
```
def run_kmeans(X, initial_centroids, max_iters, plot_progress=False):
    #inicializacao de variaveis. K representa a quantidade de clusters e e
    # obtido atraves da contagem dos centroides recebidos como parametro
    K = np.size(initial_centroids, 0)
    centroids = initial_centroids
    previous_centroids = centroids
    #o loop a seguir e' executado ate que seja atingido o numero max_iters, recebido como para
    for iter in range(max_iters):
        # primeira fase: atribuir a cada ponto o id do centroide mais proximo.
        # para isso, a funcao find_closest_centroids e chamada.
        idx = find_closest_centroids(X, centroids)

        #se o flag plot_progress tiver sido ativado na chamada da funcao, o trecho
        #de codigo abaixo e' repetido a cada iteracao, gerando varios graficos que
        #mostram o progresso do algoritimo na busca dos clusters corretos.
        if plot_progress:
            plt.scatter(X[np.where(idx==0),0],X[np.where(idx==0),1], marker='x')
            plt.scatter(X[np.where(idx==1),0],X[np.where(idx==1),1], marker='x')
            plt.scatter(X[np.where(idx==2),0],X[np.where(idx==2),1], marker='x')
            plt.plot(previous_centroids[:,0], previous_centroids[:,1], 'yo')
            plt.plot(centroids[:,0], centroids[:,1], 'bo')
            plt.show()
        #a posicao dos centroides obtida nessa iteracao sao salvos na
        #variavel previous_centroids:
        previous_centroids = centroids
        #novos centroides sao calculados a partir da posicao dos elementos a eles atribuid
        #para essa atividade, e chamada a funcao compute_centroids()
        centroids = compute_centroids(X, idx, K)
    #o resultado retornado pela funcao e a lista de centroides
    # a lista de centroides atribuidos a cada elemento de X:
    return (centroids, idx)
```

É interessante notar que a função armazena as médias anteriores e as utiliza para plotar circulos vermelhos no gráfico, que nos ajudam a perceber o progresso do algoritimo em relação a iteração anterior.

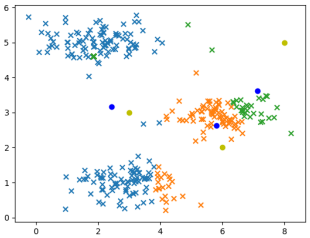
O progresso do K-means após cada iteração pode ser visto nas imagens geradas pela função **run_kmeans()**:



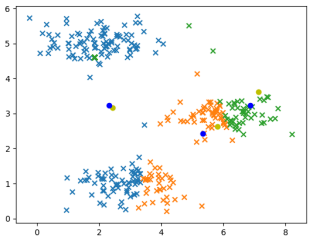
(a) Iteração 1



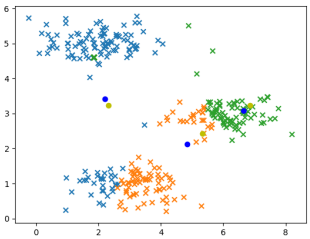
(b) Iteração 2



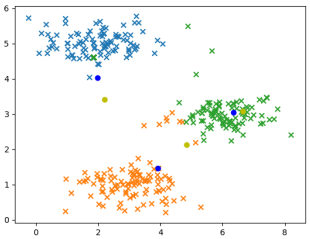
(c) Iteração 3



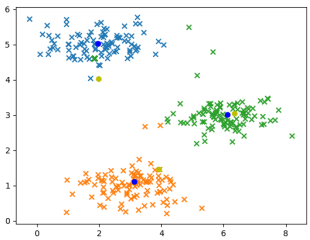
(d) Iteração 4



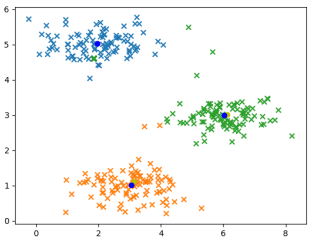
(e) Iteração 5



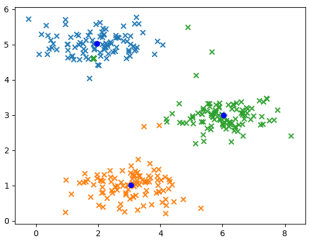
(f) Iteração 6



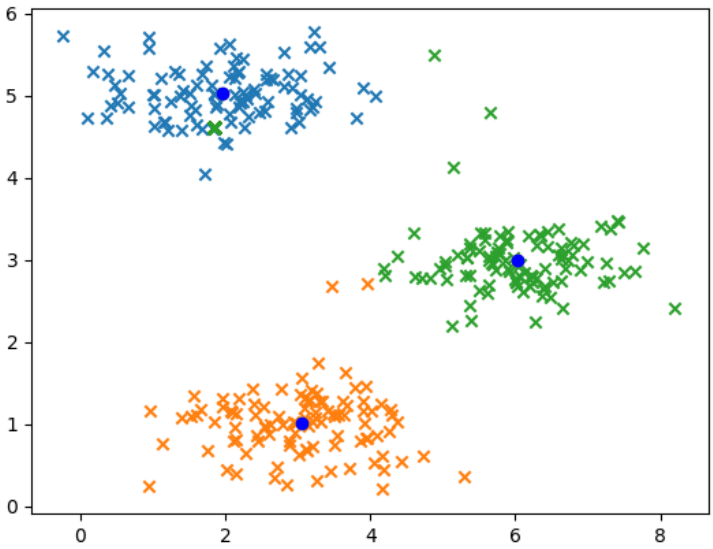
(g) Iteração 7



(h) Iteração 8



(i) Iteração 9



(j) Iteração 10

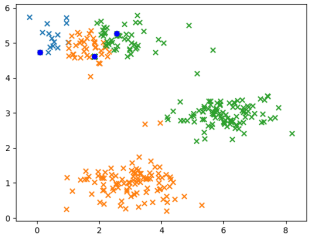
Figura 1.3: Progresso do K-means a cada iteração.

1.5 Inicialização aleatória

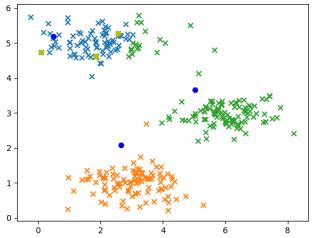
Em geral, a inicialização dos centroides é realizada de forma aleatória. A função **kmeans_init_centroids()**, que nos foi fornecida, faz essa inicialização. Para demonstrar a inicialização aleatória de centroides, fiz uma cópia do código da função **main()**, substituindo a linha de inicialização manual dos centroides por outra que faz a chamada à função mencionada acima.

```
# Fixed seeds (i.e., initial centroids)
initial_centroids = np.array([[3, 3], [6, 2], [8, 5]])

# Inicializar K centroides aleatorios
initial_centroids = kmeans_init_centroids(X, K)
```

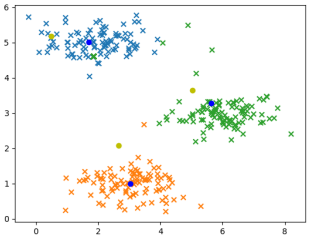
As figuras a seguir mostram o progresso do K-means em uma execução a partir de centroides iniciados aleatoriamente:



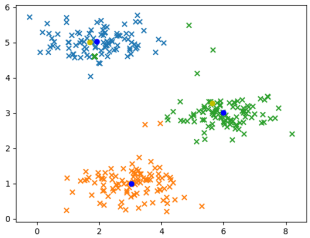
(a) Iteração 1



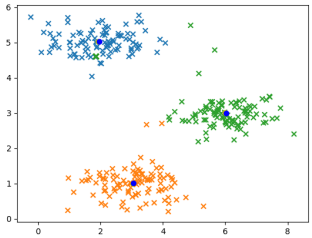
(b) Iteração 2



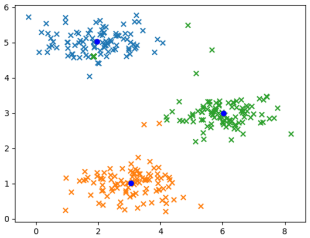
(c) Iteração 3



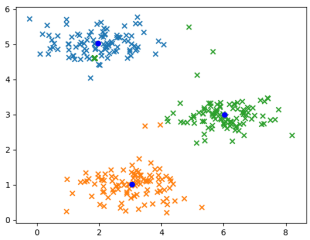
(d) Iteração 4



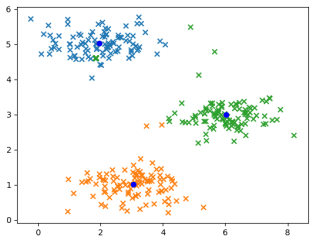
(e) Iteração 5



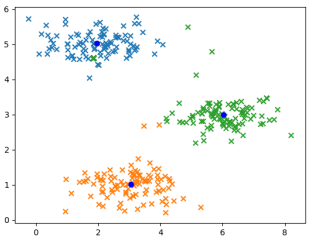
(f) Iteração 6



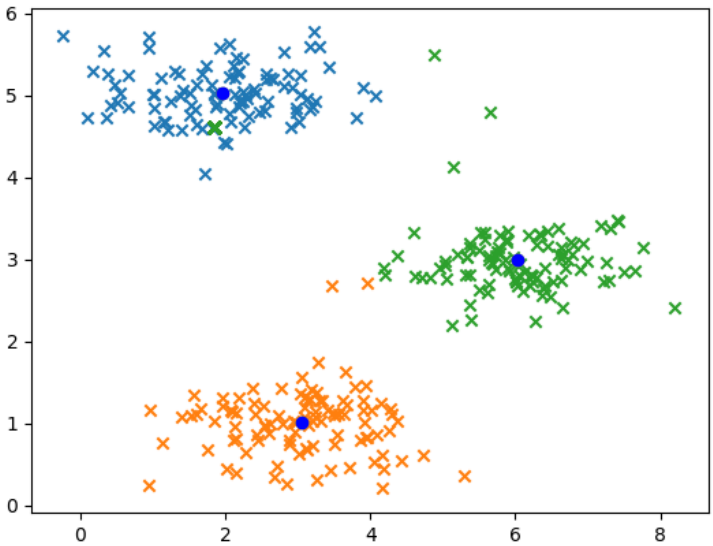
(g) Iteração 7



(h) Iteração 8



(i) Iteração 9



(j) Iteração 10

Figura 1.4: Progresso do K-means após inicialização aleatória.

Parte 2 - Redução de Dimensionalidade

2.1 Conjunto de dados de exemplo

A execução do código fornecido retorna um gráfico onde podemos visualizar o conjunto de dados de treinamento:

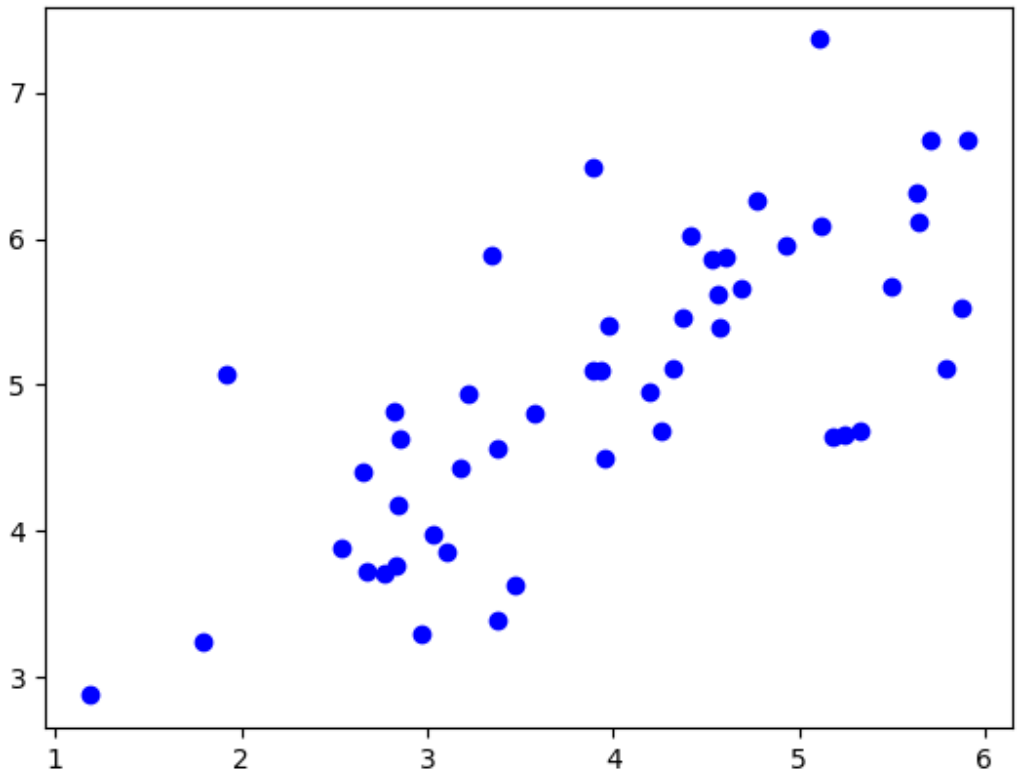


Figura 2.1: Conjunto de dados de treinamento.

2.2 Implementando o PCA

Antes de executar o PCA, deve-se fazer a normalização dos dados e, opcionalmente, *feature scaling*. A normalização é feita pelo método *z-score*, já implementado na função **normalize_features()**.

Atendido esse pré requisito, o primeiro passo do PCA é calcular a matriz de covariância dos dados. Essa matriz é determinada pela equação $\Sigma = 1/m(X)(X^T)$ e implementada através das linhas de código abaixo, na função **pca()**:

```
m = len(X)
sigma = X * X.T / m
```

Sendo X uma matrix 50x2, o resultado de sua multiplicação por X^T, sigma, será uma matriz 2x2.

Após obter a matriz Σ, o próximo passo é fazer sua decomposição utilizando a técnica SVD (Singular Value Decomposition). As linhas seguintes da função **pca()** executam essa tarefa utilizando a função *svd()* do Numpy e encerram retornando os valores de U e S, obtidos na decomposição:

```
# Decomposicao SVD da matriz sigma:
U, S, V = np.linalg.svd(sigma)
return (U, S)
```

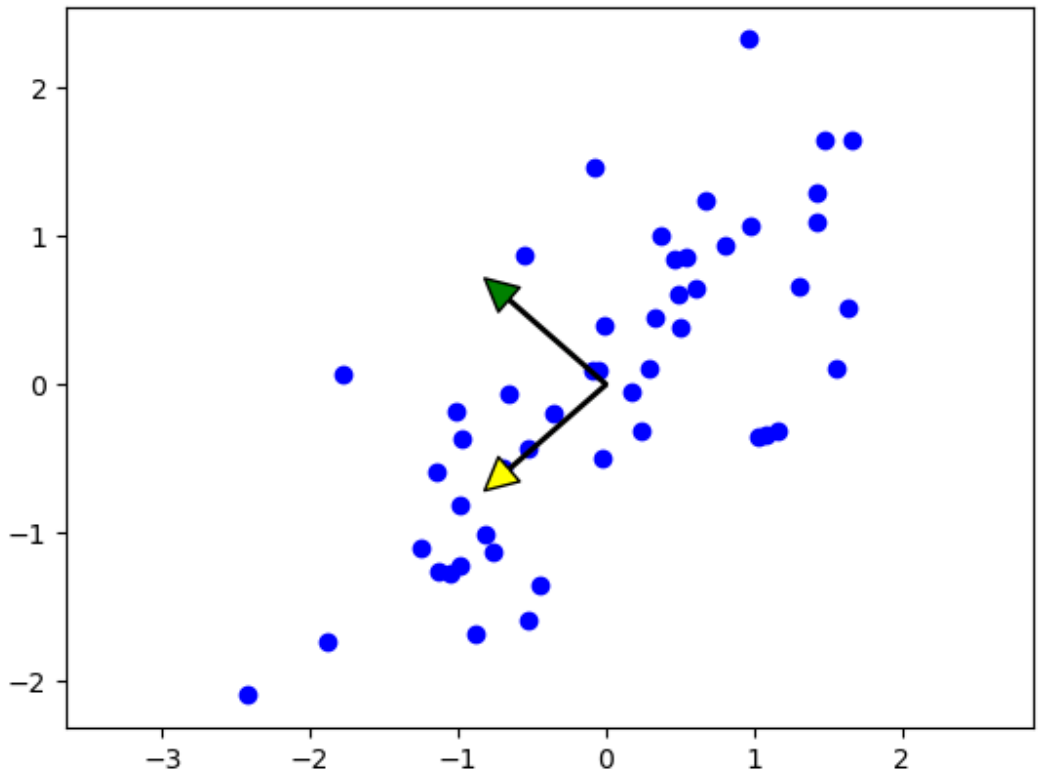


Figura 2.2: Representaçao dos componentes principais obtidos a partir das duas primeiras colunas da matriz decomposta U. Para reduzir os dados a uma dimensão, projetaremos todos os pontos sobre o primeiro vetor(amarelo)

2.3 Redução de Dimensionalidade com PCA

2.3.1 Projetando os dados nos componentes principais

A matriz U, obtida na seção anterior, será agora utilizada para fazer a redução de dimensionalidade de X (em nosso caso, de duas dimensões para uma). Isso é feito multiplicando-se a matriz X por uma nova matriz contendo apenas as primeiras K colunas de U, chamada U_reduce.

Em nosso caso, queremos reduzir X, que tem duas dimensões, para uma nova matriz Z com única dimensão. Isso é obtido multiplicando X (tamanho 50x2) por U_reduce (tamanho 2X1).

Em outros casos, como por exemplo, para reduzir um conjunto X com 10 dimensões para três dimensões, seguindo o mesmo procedimento visto acima, teríamos uma matriz U de 10x10, mas utilizaríamos apenas as 3 primeiras colunas.

O trecho de código abaixo mostra como foi feita essa implementação na função **project_data()**:

```
def project_data(X, U, K):
        # seleciona apenas as primeiras K colunas de U:
        U_reduce = U[:, 0:K]
        # Inizializa Z com o comprimento igual a X, mas com apenas K colunas:
        Z = np.zeros((len(X), K))
        # Multiplica cada linha de X por U_reduce
        for i in range(len(X)):
                x = np.matrix(X[i,:])
                projection_k = np.dot(x, U_reduce)
                Z[i] = projection_k
        # Retorna a nova matriz com dimensoes reduzidas Z:
        return Z
```

No exemplo fornecido é utilizado um loop que multiplica cada linha de X por U_reduce, mas o mesmo resultado pode ser obtido apenas multiplicando as matrizes, de forma vetorizada, como fiz no meu código:

```
        # Multiplica cada linha de X por U_reduce
        Z = X.dot(U_reduce)
```

2.3.2 Reconstruindo uma aproximação dos dados

A redução de dimensionalidade elimina as dimensões originais e as substitui por outras novas. Nesse processo, uma parte dos dados é inevitavelmente perdida. Entretanto, é possível reconstruir aproximadamente os dados originais.

Em nosso exercício, reduzimos X de duas dimensões (uma matriz mxn) para uma dimensão apenas (um vetor Z de tamanho m). Ao reconstruir os dados, conseguimos levar Z de volta a

um espaço bi-dimensional, como pode ser visto no código da função **recover_data()**:

```
def recover_data(Z, U, K):
        # Inicializa a matriz X_rec, com as dimensoes do conjunto de dados original:
        X_rec = np.zeros((len(Z), len(U)))
        # Executa o loop abaixo para cada um dos elementos do vetor Z:
        for i in range(len(Z)):
                # Inicializa v com o valor da iesima linha de Z:
                v = Z[i,:]
                # Multiplica v por cada linha da matriz U e
                # acumula os resultados ne matriz X_rec
                for j in range(np.size(U,1)):
                        recovered_j = np.dot(v.T,U[j,0:K])
                        X_rec[i][j] = recovered_j
        print("Valor do primeiro exemplo ")
        print("do conjunto de dados reconstruido:")
        print(X_rec[0])
        # Encerra a funcao, retornando a matriz X_rec
        return X_rec
```

Observe que para reconstruir a matriz original, executamos o procedimento inverso ao que utilizamos para reduzir a dimensionalidade, ou seja, em vez de multiplicar X por U para obter Z, agora multiplicamos Z por U para obter X. A diferença é que a quantidade de colunas de U que utilizamos na reconstrução (K) agora é igual a quantidade de colunas do conjunto de dados original.

A reconstrução também pode ser computada de forma vetorizada, como na versão abaixo da função recover_data():

```
def recover_data(Z, U, K):
        # Inicializa a matriz X_rec, com as dimensoes do conjunto de dados original:
        X_rec = np.zeros((len(Z), len(U)))
        # Armazena em U_rec as K primeiras colunas de U:
        U_rec = U[:,0:K]
        # Executa operacao matricial de reconstrucao:
        X_rec = np.dot(Z,U_rec.T)
        print("Valor do primeiro exemplo ")
        print("do conjunto de dados reconstruido:")
        print(X_rec[0])
        # Encerra a funcao, retornando a matriz X_rec
        return X_rec
```

2.3.3 Visualizando as projeções

O gráfico abaixo mostra os dados originais e também quão próximos a eles podem ficar os dados reconstruídos (cruzes vermelhas).

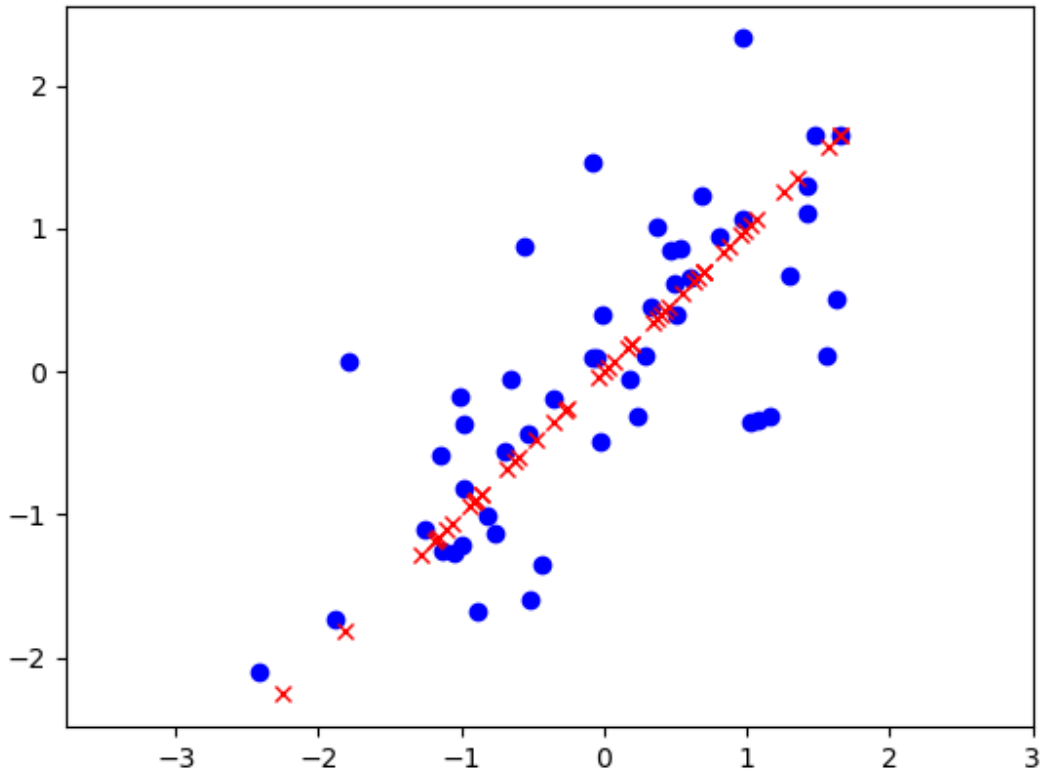


Figura 2.3: Dados originais e dados reconstruídos no espaço bidimensional.

Parte 3 - Detecção de Anomalias

Visualização do conjunto de dados do exercício, com 307 exemplos:

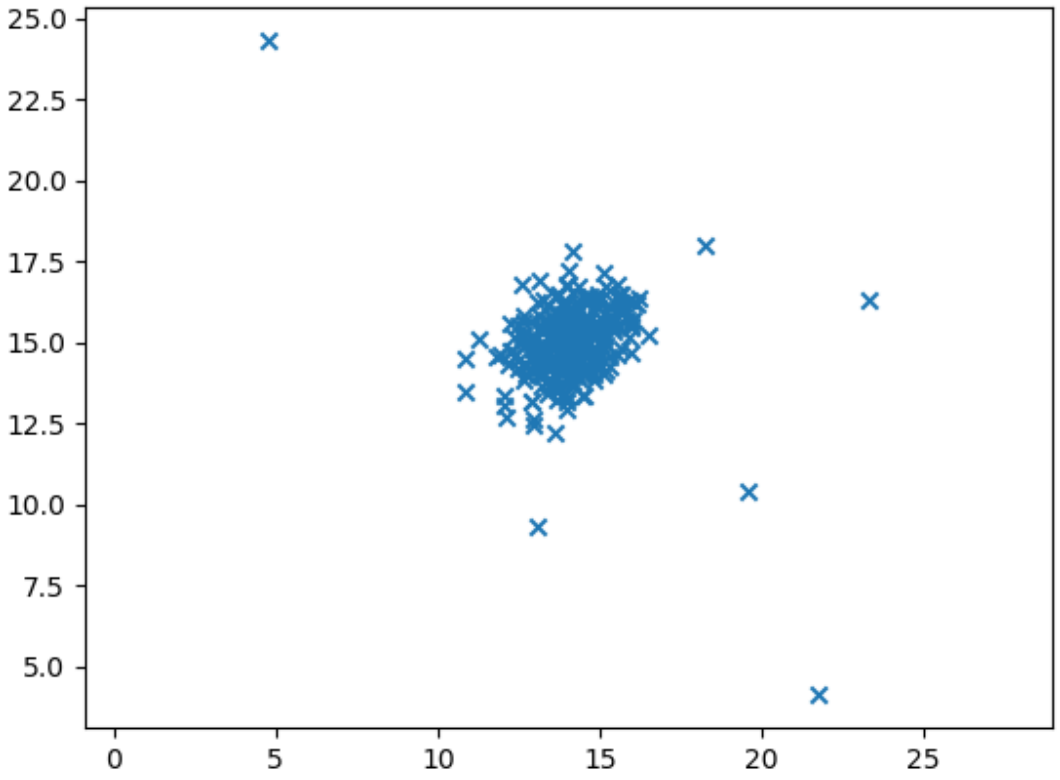


Figura 3.1: Conjunto de dados.

3.1 Distribuição Gaussiana

3.2 Estimativa de parâmetros para uma gaussiana

O objetivo dessa seção é obter os valores da média (μ) e variância(σ) para cada dimensão da matriz de dados X. Como de costume, dei preferência à utilização de cáclulos vetorizados. Com isso, ficamos com apenas um loop que é utilizado para executar os cálculos para cada uma das dimensões de X. O trecho de código abaixo mostra minha implementação da função

estimate_gaussian_params():

```
def estimate_gaussian_params(X):
        # Inicializa a variavel feat com o numero de dimensoes (colunas) em X:
        feat = np.size(X,1)
        # Inicializa m com o numero de exemplos (linhas) existentes em X:
        m = len(X)
        # Inicializa a array sigma2 com o mesmo compromento de X:
        sigma2 = np.zeros(feat)
        # Inicializa a array mu, com a mesm largura (colunas) de X:
        mu = np.zeros(feat)
        for i in range(feat):
                # Calcula a media da coluna
                mu[i] = np.sum(X[:,i]) / m
                # Calcula a variancia da coluna
                sigma2[i] = sum( np.power(X[:,i]-mu[i],2) )/m
        # retorna os valores calculados
        return (mu, sigma2)
```

O código fornecido na função main() produz o gráfico abaixo:

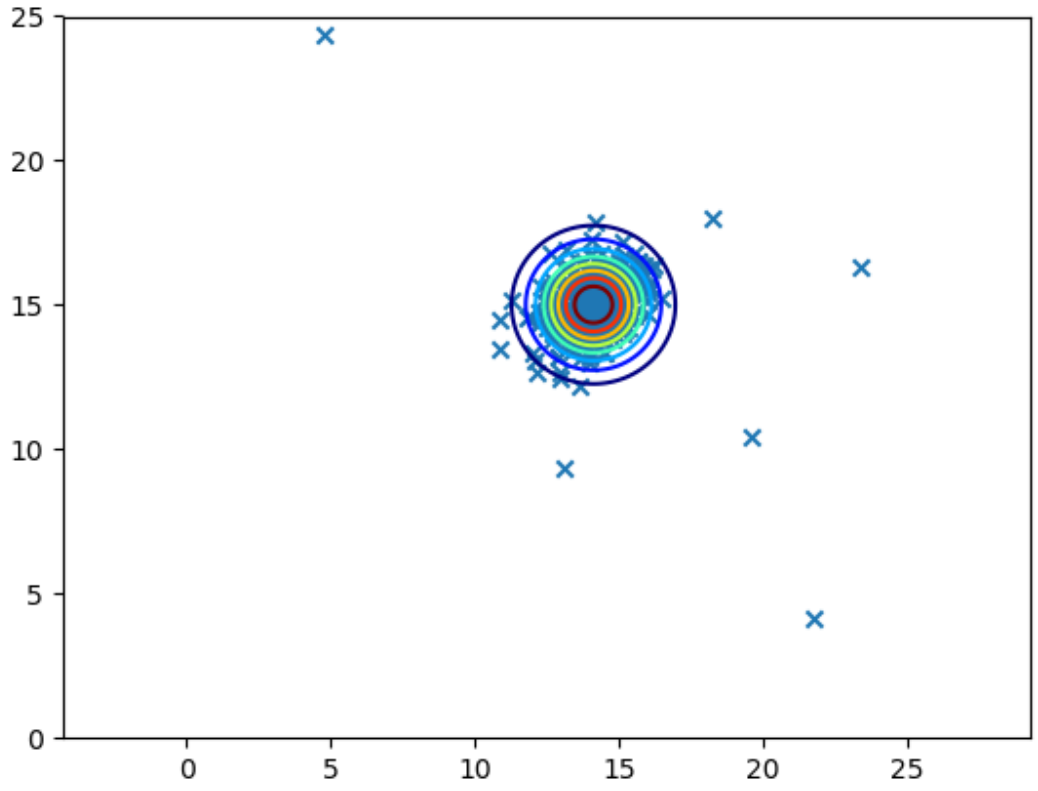


Figura 3.2: Conjunto de dados.

3.3 Selecionando ϵ

São considerados anômalos os exemplos que possuem probabilidade $P(x; \mu, \Sigma)$ inferior a ϵ. Para obter o modelo ideal, precisamos testar diversos valores de ϵ sobre os dados de um dataset de validação cruzada.

O cálculo da probabilidade é realizado pela função **main()** utilizando o pacote Stats, do SciPy, como pode ser visto no trecho de código a seguir:

```
# Calcula a desnsidade de probabilidade para cada coluna de X e armazena em pval
pval = np.zeros((Xval.shape[0], Xval.shape[1]))
pval[:,0] = stats.norm.pdf(Xval[:,0], mu[0], stddev[0])
pval[:,1] = stats.norm.pdf(Xval[:,1], mu[1], stddev[1])
```

A função **select_epsilon()** calcula o valor ideal de epsilon. A chamada da função passa como parâmetros um unico valor correspondente à multiplicação de todas as features X (porque elas todas precisam ser tratadas em conjunto) e os rótulos y, conforme abaixo:

```
# Chama a funcao select_epsilon(), armazena o resultado e imprime
epsilon, _ = select_epsilon(np.prod(pval, axis=1), yval)
```

A função select_epsilon() faz repetidas previsões para diferentes valores de epsilon. A cada execução, eça calcula scores F1, até encontrar aquele que consegue se ajustar melhor ao conjunto de validação cruzada.

```
def select_epsilon(pval, yval):
    best_epsilon_value = 0
    best_f1_value = 0
    step_size = (pval.max() - pval.min()) / 1000
    novof1 = 0
    print('step size: ' + str(step_size))
    for epsilon in np.arange(pval.min(), pval.max(), step_size):
        # armazena flag true quando o valor predito e menor que epsilon
        preds = pval < epsilon
                ########################
                # SEU CODIGO AQUI :
                # Dentro deste loop, voce deve implementar logica para
                # definir corretamente os valores das variaveis
                # best_epsilon_value e best_f1_value.
                ########################
                # chama a funcao f1 para calcular o valor de F1
        novof1 = f1(preds,pval, yval)
                # Se encontrado um valor melhor de FI, armazena.
        if novof1 > best_f1_value:
            best_f1_value = novof1
```

```
            best_epsilon_value = epsilon
    return best_epsilon_value, best_f1_value
```

Eu criei uma nova função com o nome **f1()** que faz o cálculos o FScore. Ela recebe como parâmetros três arrays, uma com os valores das predições obtidos com o epsilon atual (preds), uma com o valor do produto dos exemplos (pval) e os rótulos (yval).

As ativiades executadas pela função estão comentadas no código abaixo:

```
def f1(preds, pval ,yval):
    # inicializa as variaveis
    # TP - true positives
    tp = 0
    # FP - false positives
    fp = 0
        # FN - false negatives
    fn = 0
    # Executa os comandos do loop para cada uma das linhas do conjunto de validacao cruzada:
    for i in range(len(pval)):
        # Faz a contagem de TPs, FPs e FNs
        if preds[i] == True and int(yval[i]) == 1:
            tp = tp + 1
        if preds[i] == True and int(yval[i]) == 0:
            fp = fp + 1
        if preds[i] == True and int(yval[i]) == 1:
            fn = fn + 1
    # Faz os calculos da precisao, revocacao e do Fscore:
    if tp + fp > 0:
        prec = tp / (tp + fp)
    else:
        prec = 0
    if tp + fn > 0:
        rec = tp / (tp + fn)
    else:
        rec = 0
    if prec + rec > 0:
        f1 = 2 * prec * rec / (prec + rec)
    else:
        f1 = 0
    return f1
```

Após obter o valor ideal de epsilon, a função **main()** imprime novamente o gráfico inicial, desta vez destacando em vermelho os exemplos com probabilidade inferior, conforme a imagem abaixo.

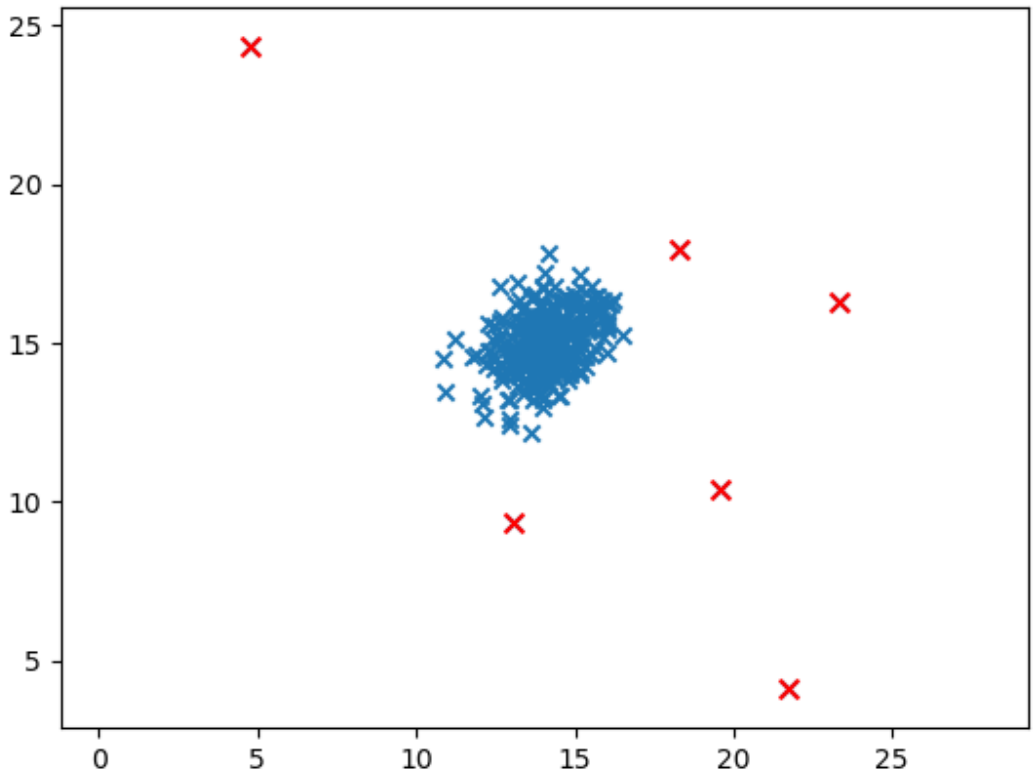


Figura 3.3: Anomalias detectadas.