Data Mining

Tamás Budavári - budavari@jhu.edu
Class 5

- Regularization
- Principal Component Analysis
- Lagrange multipliers
- Explained variance

In [1]:
```
%pylab inline
```

```
Populating the interactive namespace from numpy and matplotlib
```

Continued from Last Lecture

Linear Regression

- A linear combination of known $\phi_k(\cdot)$ **basis** functions

> $$f(t;\boldsymbol{\beta}) = \sum_{k=1}^{K} \beta_k \, \phi_k(t)$$
>
> It's a dot product (https://en.wikipedia.org/wiki/Dot_product#Definition) with $\boldsymbol{\beta} = (\beta_1, \dots, \beta_K)^T$

- Evaluated at all data points $x = (x_1, x_2, \dots, x_N)$

> $f(x;\boldsymbol{\beta}) = X\boldsymbol{\beta}$ where $X_{ik} = \phi_k(x_i)$

Method of Least Squares

- At the optimum

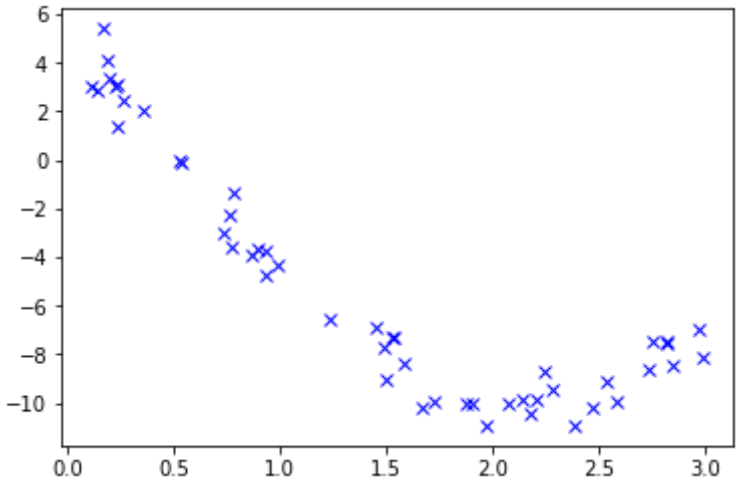
> $\hat{\boldsymbol{\beta}} = (X^T X)^{-1} X^T \boldsymbol{y}$ (c.f. **Lecture Note 04**)

- Hat matrix

> $$H = X\,(X^T X)^{-1} X^T$$
>
> $$\hat{\boldsymbol{y}} = X\hat{\boldsymbol{\beta}} = X(X^T X)^{-1} X^T \boldsymbol{y} = H\boldsymbol{y}$$

In [2]:
```
# Generate a dataset with errors
x = 3 * random.rand(50)           # uniform between 0 and 3
eps = 1 * random.randn(x.size)    # add normal noise
y = 10*cos(x+1) + eps;

# Plot the data
plot(x, y, 'bx');                 # 'b' for color(blue) and 'x' for marker
```

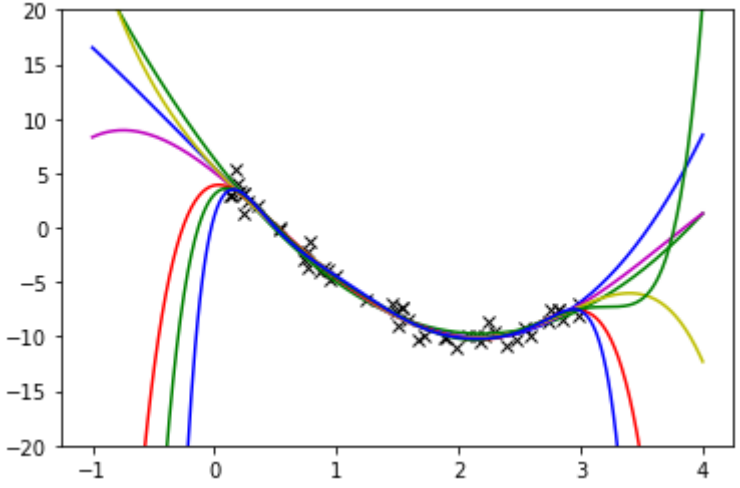


In [3]:
```
# Function to construct X matrix as [1, X, X^2, ..., X^n]
def poly(x,n):
    X = np.zeros((x.size,n+1));
    for i in range(X.shape[1]):
        X[:,i] = x**i
    return X

# Show data in black
plot(x,y,'kx');
ylim(-20, 20);                  # control the range of y-axis to be (-20, 20)

xx = np.linspace(-1,4,500)      # grid on x
color = 'yrgbm' * 5             # color sequence


for n in range(2,9):
    X = poly(x,n)                       # design matrix for fitting
    bHat = linalg.pinv(X).dot(y)        # estimate beta
    yy = poly(xx,n).dot(bHat)           # prediction
    plot(xx,yy,'-',c=color[n]);         # plot to compare the truth and prediction
```



Regularization

Penalize large coefficients in β

- **Ridge regression** uses L_2

  > $$\hat{\beta} = \underset{\beta}{\operatorname{argmin}} |y - X\beta|_2^2 + \lambda |\beta|_2^2$$
  >
  > or even with a constant matrix Γ
  >
  > $$\hat{\beta} = \underset{\beta}{\operatorname{argmin}} |y - X\beta|_2^2 + \lambda |\Gamma\beta|_2^2$$

- **Lasso regression** uses L_1

  > $$\hat{\beta} = \underset{\beta}{\operatorname{argmin}} |y - X\beta|_2^2 + \lambda |\beta|_1$$
  >
  > L_1 yields sparse results

Different geometric meanings!

- **Note**

  > You may think of this as one application of bias-variance tradeoff into regression. We want to find a model that could have a good performence and in the meantime not too complex. After introducing some regularization into $\hat{\beta}$, it will be composed of two part where first term would measure bias and second term would measure variance. And λ (strength) could be used to balance bias (accuracy) and variance (complexity). If λ is very small, then it will not care much about the complexity (second term) and may give us a relatively complex and accurated model. On the other hand, if λ is very large, then it will not care much about the accuracy (first term) and give us a relatively simple model with a bad performance.

- More about Bias–variance tradeoff (https://en.wikipedia.org/wiki/Bias%E2%80%93variance_tradeoff)

Linear Combinations

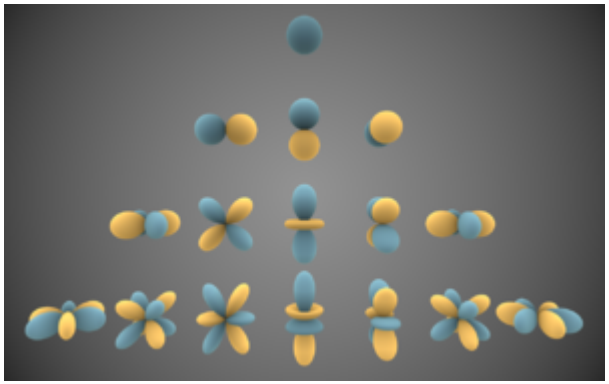
- Coefficients mix a given set of basis vectors, functions, images, shapes, ...

$$f(x;\beta) = \sum_k \beta_k \phi_k(x)$$

> Fourier series

> Discrete Cosine Transform (JPEG)

> Spherical Harmonics



- What is a good basis like?

Principal Component Analysis

Statistical Learning

Output Type	Supervised	Unsupervised
Discrete	Classification	Clustering
Continuous	Regression	Dimensionality Reduction

Directions of Maximum Variance

- Let $X \in \mathbb{R}^N$ be a continuous random variable with $\mathbb{E}[X] = 0$ mean and covariance matrix C. What is the direction of maximum variance?

For any vector $a \in \mathbb{R}^N$, we have

$$\mathbb{V}\mathrm{ar}[a^T X] = \mathbb{E}\left[(a^T X - \mathbb{E}[a^T X])(a^T X - \mathbb{E}[a^T X])^T\right] = \mathbb{E}\left[(a^T X - 0)(a^T X - 0)^T\right] = \mathbb{E}\left[(a^T X)(X^T a)\right] = \mathbb{E}\left[a^T (X X^T)\, a\right]$$

Note that

$$C = \mathbb{E}\left[(X - \mathbb{E}[X])(X - \mathbb{E}[X])^T\right] = \mathbb{E}\left[(X - 0)(X - 0)^T\right] = \mathbb{E}\left[X X^T\right]$$

Then we have

$$\mathbb{V}\mathrm{ar}[a^T X] = a^T\, \mathbb{E}\left[X X^T\right]\, a = a^T C\, a$$

We have to maximize this such that $a^T a = \|a\|^2 = 1$.

Constrained Optimization

- **Lagrange multiplier**: extra term with new parameter λ

$$\hat{a} = \arg\max_{a \in \mathbb{R}^N} \left[a^T C\, a - \lambda\,(a^T a - 1) \right]$$

- Partial derivatives vanish at optimum

$$\frac{\partial}{\partial \lambda} \rightarrow \quad \hat{a}^T \hat{a} - 1 = 0 \quad \text{(duh!)}$$

$$\frac{\partial}{\partial a_k} \rightarrow \quad ?$$

- More about Lagrange multiplier (https://en.wikipedia.org/wiki/Lagrange_multiplier)

With Indices

$$\max_{a \in \mathbb{R}^N} \left[\sum_{i,j} a_i C_{ij} a_j - \lambda \left(\sum_i a_i^2 - 1 \right) \right]$$

- Partial derivatives $\partial / \partial a_k$ vanish at optimum

$$\sum_{i,j} \frac{\partial a_i}{\partial a_k} C_{ij} a_j + \sum_{i,j} a_i C_{ij} \frac{\partial a_j}{\partial a_k} - 2\lambda \left(\sum_i a_i \frac{\partial a_i}{\partial a_k} \right)$$

$$= \sum_{i,j} \delta_{ik} C_{ij} a_j + \sum_{i,j} a_i C_{ij} \delta_{jk} - 2\lambda \left(\sum_i a_i \delta_{ik} \right)$$

$$= \sum_j C_{kj} a_j + \sum_i a_i C_{ik} - 2\lambda \, a_k$$

$$= 0$$

With Vectors and Matrices

- Write the equation above with indices as

> $C\hat{a} + C^T\hat{a} - 2\lambda\hat{a} = 0$
>
> Note that C is symmetric, i.e. $C = C^T$, then we have
>
> $C\,\hat{a} = \lambda\,\hat{a}$

- Eigenproblem !! (quick review (https://en.wikipedia.org/wiki/Eigendecomposition_of_a_matrix#Fundamental_theory_of_matrix_eigenvectors_and_eigenvalues))

Result

- The value of maximum variance is

> $\hat{a}^T C\,\hat{a} = \hat{a}^T\lambda\,\hat{a} = \lambda\,\hat{a}^T\hat{a} = \lambda$
>
> the largest eigenvalue λ_1 of C.

- The direction of maximum variance is the corresponding eigenvector a_1

> $Ca_1 = \lambda_1 a_1$

- This is the **1st Principal Component**

2nd Principal Component

- Direction of largest variance uncorrelated to 1st PC

> Given a_1, we want to maximize $\mathbb{V}\mathrm{ar}[a^T X]$ such that $a^T a = 1$ and $\mathbb{C}\mathrm{ov}[a^T X, a_1^T X] = a^T C a_1 = 0$
>
> $$\hat{a} = \arg\max_{a \in \mathbb{R}^N} \left[a^T C\, a - \lambda\,(a^T a - 1) - \lambda'(a^T C\, a_1) \right]$$

- Partial derivatives vanish at optimum

> $$2C\,\hat{a} - 2\lambda\,\hat{a} - \lambda' C a_1 = 0$$

Result

- Multiply the equation above by a_1^T

> $$2a_1^T C\hat{a} - 2a_1^T \lambda \hat{a} - a_1^T \lambda' C a_1 = 0$$
>
> $$0 - 0 - \lambda' \lambda_1 = 0 \quad \Rightarrow \quad \lambda' = 0$$

- Still just an eigenproblem

> $$C\,\hat{a} = \lambda\,\hat{a}$$

- Solution λ_2 and a_2, where λ_2 is the second-largest eigenvalue and a_2 is the associated eigenvector.

Quick Review of Matrix Decomposition

In [4]:
```
from scipy import linalg
```

- **Eigendecomposition** (more (https://en.wikipedia.org/wiki/Eigendecomposition_of_a_matrix))

> Only for diagonalizable matrices (https://en.wikipedia.org/wiki/Diagonalizable_matrix)
>
> A square matrix A is called diagonalizable if there exists an invertible matrix P such that $P^{-1}AP$ is a diagonal matrix.

In [5]:
```
# Eigendecomposition
A = np.array([[1, 2], [3, 4]])
print('Original Matrix: \n', A, '\n')

eigenvalues, eigenvector = linalg.eig(A)
print('Eigenvalues: \n', eigenvalues, '\n')
print('Eigenvector: \n', eigenvector)
```

```
Original Matrix:
 [[1 2]
 [3 4]]

Eigenvalues:
 [-0.37228132+0.j  5.37228132+0.j]

Eigenvector:
 [[-0.82456484 -0.41597356]
 [ 0.56576746 -0.90937671]]
```

- **Singular-value decomposition** (more (https://en.wikipedia.org/wiki/Singular_value_decomposition))

> The generalization of the eigendecomposition
>
> For example, a symmetric $n \times n$ matrix with positive eigenvalues to any $n \times m$ matrix
>
> $X_{n \times m} = UWV^{\top}$ where
>
> - $U_{n \times n}$, $U^{\top}U = I$
> - $W_{n \times m}$, diagonal
> - $V_{m \times m}$, $V^{\top}V = I$
>
> More generalized statement (https://en.wikipedia.org/wiki/Singular_value_decomposition#Statement_of_the_theorem)

In [6]:
```
# Example
n, m = 5, 3
A = np.random.randn(n, m)
print('Original Matrix: \n', A, '\n')

# SVD
U, s, Vh = linalg.svd(A)
print('Singular values: \n', s, '\n')
print('Left-singular vectors: \n', U, '\n')
print('Right-singular vectors: \n', Vh)
```

```
Original Matrix:
 [[-1.11923878 -0.75099072 -0.22806693]
 [ 1.42047089  0.46163772 -0.01221367]
 [-0.13676406 -0.53157936 -0.44999876]
 [ 0.65507513  0.60338112 -0.36001278]
 [-0.97670832 -1.08267532  1.93307017]]

Singular values:
 [ 2.89956307  1.62198158  0.59235923]

Left-singular vectors:
 [[-0.35533327 -0.55066852 -0.16363605  0.39389217 -0.62335756]
 [ 0.41656871  0.49873175 -0.57911445  0.05894697 -0.48876257]
 [-0.04691975 -0.32481689 -0.76937247  0.04794572  0.54594846]
 [ 0.32594378  0.08287577  0.18125689  0.88833888  0.25473955]
 [-0.76926456  0.57935862 -0.11428748  0.223449    0.09790063]]

Right-singular vectors:
 [[ 0.67620816  0.52201985 -0.51984402]
 [ 0.52874351  0.14747065  0.83587242]
 [-0.51300373  0.8400879   0.17629375]]
```

- **A little bit more about `scipy.linalg.eig` (https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.eig.html) and `scipy.linalg.svd` (https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.svd.html)**

PCA

- Spectral decomposition or eigenvalue decomposition or eigendecomposition

> Let $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_N \geq 0$ be the eigenvalues of C and $e_1, \ldots, e_N$ the corresponding eigenvectors
>
> $$C = \sum_{k=1}^{N} \lambda_k \left(e_k \, e_k^T \right)$$
>
> Consider $C\, e_l = \sum_k \lambda_k \, e_k \left(e_k^T e_l \right) = \lambda_l \, e_l$ for any l

- Matrix form

> With diagonal Λ matrix of the eigenvalues and an E matrix of $[e_1, \ldots, e_N]$
>
> $$C = E \, \Lambda \, E^T$$

- The eigenvectors of largest eigenvalues capture the most variance

> If keeping only $K < N$ eigenvectors, the best approximation is taking the first K PCs
>
> $$C \approx \sum_{k=1}^{K} \lambda_k \left(e_k \, e_k^T \right) = E_K \Lambda_K E_K^T$$

New Coordiante System

- The E matrix of eigenvectors is a rotation, $E E^T = I$

> $Z = E^T X$

- A truncated set of eigenvectors E_K defines a projection

> $Z_K = E_K^T X$
>
> and
>
> $X_K = E_K Z_K = E_K E_K^T X = P_K X$

Detour: Projections

- If the square of a matrix is equal to itself

> $P^2 = P$

- For example, projecting on the e unit vector (https://en.wikipedia.org/wiki/Unit_vector)

> Scalar times vector
>
> $r' = e \left(e^T r \right) = e \beta_r$
>
> Or projection of vector r
>
> $r' = \left(e e^T \right) r = P r$

Again

- The eigenvectors of largest eigenvalues capture the most variance

$$C \approx C_K = \sum_{k=1}^{K} \lambda_k \left(e_k \, e_k^T \right) = \sum_{k=1}^{K} \lambda_k \, P_k$$

- And the remaining eigenvectors span the subspace with the least variance

$$C - C_K = \sum_{l=K+1}^{N} \lambda_l \, P_l$$

Samples

- Set of N-vectors arranged in matrix $X = [x_1, x_2, \dots, x_n]$ with average of 0
  This is NOT the random variable we talked about previously but the data matrix!

> Sample covariance matrix is
>
> $$C = \frac{1}{n-1} XX^T = \frac{1}{n-1} \sum_i x_i x_i^T$$

- Singular Value Decomposition (SVD)

> $$X = UWV^T$$
>
> where $U^T U = I$, W is diagonal (thus $W = W^T$), and $V^T V = I$

- Hence

> $$C = \frac{1}{n-1} XX^T = \frac{1}{n-1} UWV^T (UWV^T)^T = \frac{1}{n-1} UWV^T VWU^T = \frac{1}{n-1} UW^2 U^T$$
>
> So, if $C = E\Lambda E^T$ then we have
>
> $$E\Lambda E^T = \frac{1}{n-1} UW^2 U^T$$
>
> $E = U$ and $\Lambda = \frac{1}{n-1} W^2$

Intution about PCA

Consider a continuous random variable $X \in \mathbb{R}^p$ be with $\mathbb{E}[X] = 0$ mean and covariance matrix C.

Let $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_p \geq 0$ be the eigenvalues and $e_1, \ldots, e_p$ be the corresponding eigenvectors of C.

- The **goal** of PCA is to seek a different representation of X under a new coordinate system

> The new coordinate system is define by $e_1, \ldots, e_p$ such that $e_k^T e_k = 1$ and $e_k^T e_j = 1$ for $k \neq j$.
>
> For the new representation, we have $X = (e_1^T X)e_1 + \cdots + (e_p^T X)e_p$. Note that each $e_k^T X$ is a (random) scalar representing the project of X on the axis e_k. In other words, instead of representing $X = (x_1, \ldots, x_n)$ under the classic Cartesian coordinate system, we can now represent $X = (e_1^T X, \ldots, e_p^T X)$ under the new coordinate system.
>
> Under the new coordinate system
>
> - e_1 is chosen such that $\mathbb{V}\mathrm{ar}(e_1^T X)$ is maximized and $\|e_1\|^2 = 1$ to ensure a proper coordinate system
> - e_2 is chosen such that $\mathbb{V}\mathrm{ar}(e_2^T X)$ is maximized and $\|e_2\|^2 = 1, e_1^T e_2 = 0$ to ensure a proper coordinate system (note that $Cov(e_1^T X, e_2^T X) = 0$ is equivalent to $e_1^T e_2 = 0$)
> - etc.

- When the random variable X is replaced by data points $x_1, \ldots, x_p$

> - Define $X = [x_1, x_2, \ldots, x_n]$, where each x_i is a p-dimensional column vector for $i = 1, \ldots, n$ and $C = \frac{1}{n-1} XX^T$.

Random Sample from Bivariate Normal

- See previous lecture

In [7]:
```
from scipy.stats import norm
```

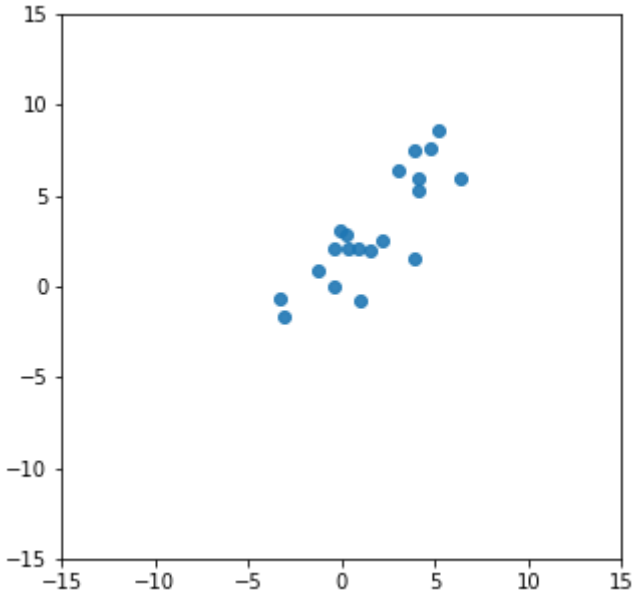
In [8]:
```
# Generate multiple 2-D (column) vectors
S = norm.rvs(0,1,(2,20))

# Scale axis 0
S[0,:] *= 4

# Rotate by 45 degrees
f = +pi/4
R = array([[cos(f), -sin(f)],
           [sin(f),  cos(f)]])
X = R.dot(S)

# Shift
X += np.array([[1],[3]])

# Plot the points
figure(figsize=(5,5));
xlim(-15,15);
ylim(-15,15);
plot(X[0,:],X[1,:],'o',alpha=0.9);
```



In [9]:
```
# Subtract sample mean (centering)
avg = mean(X, axis=1).reshape(X[:,1].size,1)
X -= avg

# Sample covariance matrix
C = X.dot(X.T) / (X[0,:].size-1)
print ("Average\n", avg)
print ("Covariance\n", C)
```

```
Average
 [[ 1.64099666]
 [ 3.16929267]]
Covariance
 [[ 7.364622    6.90883924]
 [ 6.90883924  9.21679239]]
```

In [10]:
```
# Eigendecomposition of covariance matrix
L, E = np.linalg.eig(C)
print("eigenvalues:", E)
print("eigenvectors:", L)
```

```
eigenvalues: [[-0.75261388 -0.65846211]
 [ 0.65846211 -0.75261388]]
eigenvectors: [  1.32007642  15.26133797]
```

In [11]:
```
# SVD of covariance matrix
E, L, E_same = np.linalg.svd(C)
print("unitary vectors", E)
print("singluar values:", L)
# note that eigenvalues and singular values are the same since C is positive (semi-)definite matrix
```

```
unitary vectors [[-0.65846211 -0.75261388]
 [-0.75261388  0.65846211]]
singluar values: [ 15.26133797   1.32007642]
```

In [12]:
```
# Check EE^T = I
E.dot(E.T)
```

Out[12]:
```
array([[  1.00000000e+00,  -2.22044605e-16],
       [ -2.22044605e-16,   1.00000000e+00]])
```

In [13]:
```
# Check E^T and E^(-1) are very close
np.allclose( E.T, np.linalg.inv(E) )
```

Out[13]:
```
True
```

In [14]:
```
# SVD of original data matrix
U, W, V = np.linalg.svd(X)
print("E = U:", U)
print("Lambda:", W**2 / (X[0,:].size-1))
```

```
E = U: [[-0.65846211 -0.75261388]
 [-0.75261388  0.65846211]]
Lambda: [ 15.26133797   1.32007642]
```

In [15]:
```
# Alternatively
U, W**2 / (X.shape[1]-1)
```

Out[15]:
```
(array([[-0.65846211, -0.75261388],
        [-0.75261388,  0.65846211]]), array([ 15.26133797,   1.32007642]))
```

In [16]:
```
# Check UU^T = I and VV^T = I
[ np.allclose( U.dot(U.T), np.eye(U.shape[0]) ),
  np.allclose( V.dot(V.T), np.eye(V.shape[0]) )  ]
```

Out[16]:
```
[True, True]
```

In [17]:
```
from sklearn import decomposition
```

In [18]:
```
# Another way to do PCA
pca = decomposition.PCA(n_components=X.shape[0])
pca.fit(X.T) # different convention: row vs col !!!

# E and Lambda
print("E = U:", pca.components_.T)
print("Lambda:", pca.explained_variance_)
```

```
E = U: [[-0.65846211  0.75261388]
 [-0.75261388 -0.65846211]]
Lambda: [ 15.26133797   1.32007642]
```