lecture4-RISE

November 4, 2019

1 Python Programming for Machine Learning

Lecture 4

- Numerical instability
  - Rounding, Underflow, Overflow
- Linear Algebra
  - Trace Operator
  - Linear Regression
  - Singular Value Decomposition

1.1 Rounding

[1]:
```
# Import the required packages
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
```

Let's start with a weird experiment:

[2]:
```
a = np.power([10], np.arange(4)).astype('float32')
print(a)
```

```
[   1.   10.  100. 1000.]
```

[3]:
```
# Add and subtract the same (huge) number
huge_number = 1e9
print((a + huge_number) - huge_number)
```

```
[   0.    0.  128. 1024.]
```

Now, let's repeat the experiment with higher precision (`float64`):

[4]:
```
a = np.array([1,10,100,1000],dtype='float64')
print(a)
```

```
[   1.   10.  100. 1000.]
```

[5]:
```
huge_number = 1e9
# it seems to work now
print((a + huge_number) - huge_number)
```

```
[   1.   10.  100. 1000.]
```

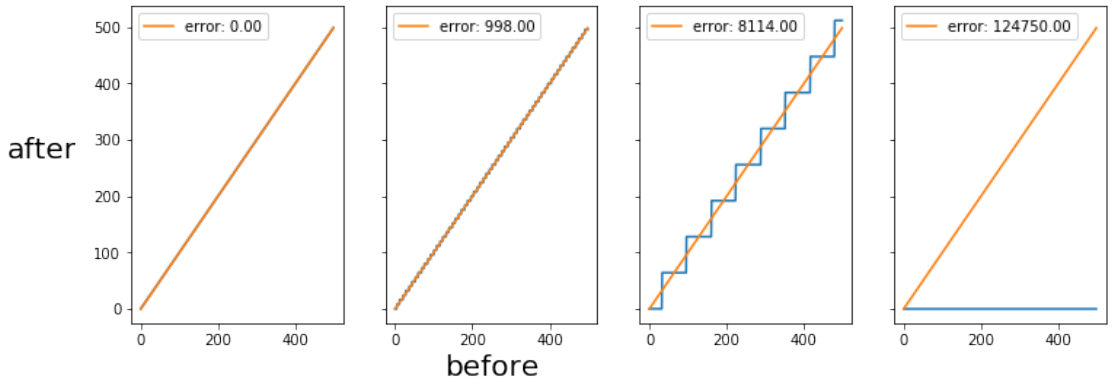
We can also reach the limits of (`float64`):

[6]:
```
huge_number = 1e18
print((a + huge_number) - huge_number)
```

```
[   0.    0.  128. 1024.]
```

1.1.1 Understanding of the rounding effect

[7]:
```
# We plot all numbers before and after application of the addition and␣
 ↪substraction:
a = np.arange(500).astype('float32')
huge_numbers = [1e7,1e8, 1e9,1e10]
f, axis = plt.subplots(1, len(huge_numbers), sharex=True, sharey=True, figsize␣
 ↪= (12,4))
for i,num in enumerate(huge_numbers):
    ax = axis[i]
    b = (a + num) - num
    ax.plot(a, b)
    ax.plot(a, a, label=f'error: {np.sum(np.abs(a-b)):.2f}')
    ax.legend()
axis[1].set_xlabel('before', fontsize=20)
_=axis[0].set_ylabel('after', rotation=0, labelpad=35, fontsize=20)
```

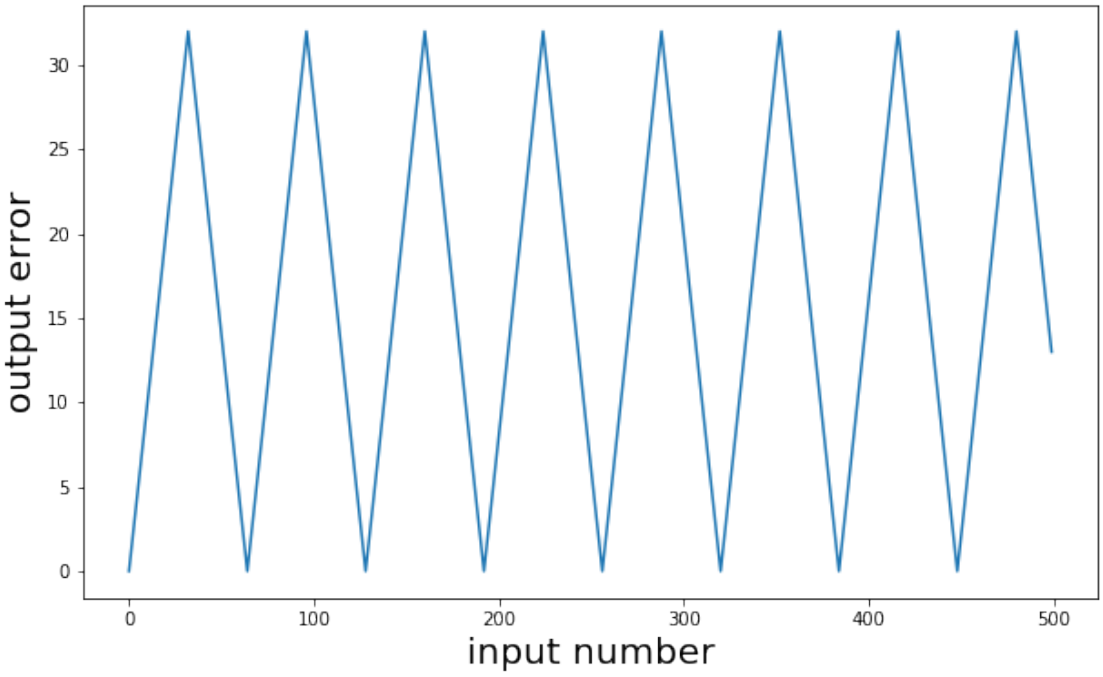


1.1.2 Comments

- The `float32` and `float64` number representations have a certain budget of bits to represent real numbers. Therefore, they allocate precision where it is important (e.g. for small numbers).
- The smaller the precision, the less memory is used and therefore the more efficient (computationally), but also the more careful we should be about potential loss of precision.
- Unlike typical observed data, error is not random-looking, but very structured:

1.2 Rounding error plot

[10]:
```
num = 1e9
a = np.arange(500).astype('float32')
b = (a + num) - num
error = np.abs(a - b)
plt.figure(figsize=(10,6))
plt.plot(a, error)
plt.xlabel('input number', fontsize=20)
_=plt.ylabel('output error', fontsize=20)
```



1.3 Overflow

Overflow is a frequently encountered problem when implementing machine learning algorithms.

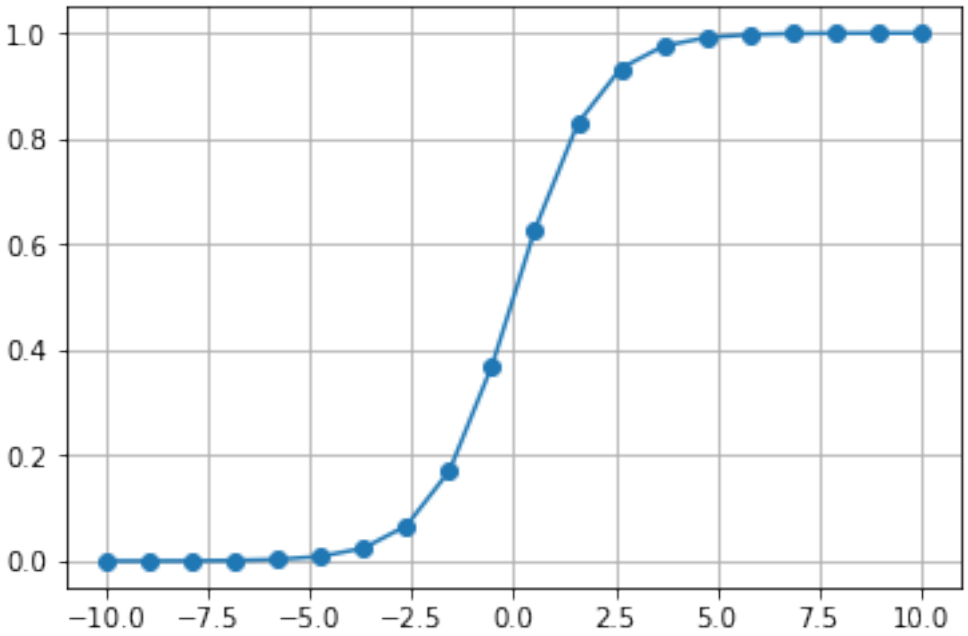
[11]:
```
a = np.array([1, 10, 100, 1000], dtype='float32')

print(np.exp(a))
```

```
[2.7182817e+00 2.2026465e+04           inf           inf]
```

1.3.1 The sigmoid function

$$\sigma(x) = \frac{e^x}{1+e^x}$$

[15]:
```
sigmoid = lambda x: np.exp(x) / (1 + np.exp(x))
x = np.linspace(-10,10,20)
plt.plot(x,sigmoid(x),'o-')
plt.grid()
```



1.3.2 Compute sigmoid function for some values

[16]:
```
x = np.array([-1e5,-1e4,-1e3,-1e2,1e1,1e0,1e1,1e2,1e3,1e4,1e5]).astype(np.
↪float32)
sigmoid(x)
```

```
/home/sdogadov/anaconda3/envs/ML/lib/python3.7/site-
packages/ipykernel_launcher.py:1: RuntimeWarning: invalid value encountered in
true_divide
  """Entry point for launching an IPython kernel.
```

[16]:
```
array([0.000000e+00, 0.000000e+00, 0.000000e+00, 3.783506e-44,
       9.999546e-01, 7.310586e-01, 9.999546e-01,          nan,
                nan,          nan,          nan], dtype=float32)
```

Where does the nan come from?

[17]:
```
print(np.exp(1000))
print(float('inf') / float('inf'))
```

```
inf
nan
```

1.3.3 Stable sigmoid function

Let's rewrite the sigmoid function in a different way

$$\sigma(x) = \frac{e^x}{1+e^x} = \frac{e^{-x}e^x}{e^{-x}(1+e^x)} = \frac{1}{1+e^{-x}}$$

[18]:
```
sigmoid = lambda x: 1 / (1 + np.exp(-x))
print(sigmoid(x))
print(1/float('inf'))
```

```
[0.        0.        0.        0.        0.9999546 0.7310586 0.9999546
 1.        1.        1.        1.       ]
0.0
```

Here, we still get an overflow. But this time, we are lucky since $1/\text{inf} = 0.0$, which is the desired result for large negative inputs.

1.3.4 The sigmoid function

The sigmoid function can be written in yet another way using $\tanh(x)$ function:

$$\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$$

$$\sigma(x) = \frac{e^x}{1+e^x} = \frac{1}{2}\left(\frac{2e^x}{1+e^x}\right) = \frac{1}{2}\left(\frac{e^x-1+1+e^x}{1+e^x}\right) = \frac{1}{2}\left(\frac{e^x-1}{e^x+1}+1\right) = \frac{1}{2}\left(\tanh\left(\frac{x}{2}\right)+1\right)$$

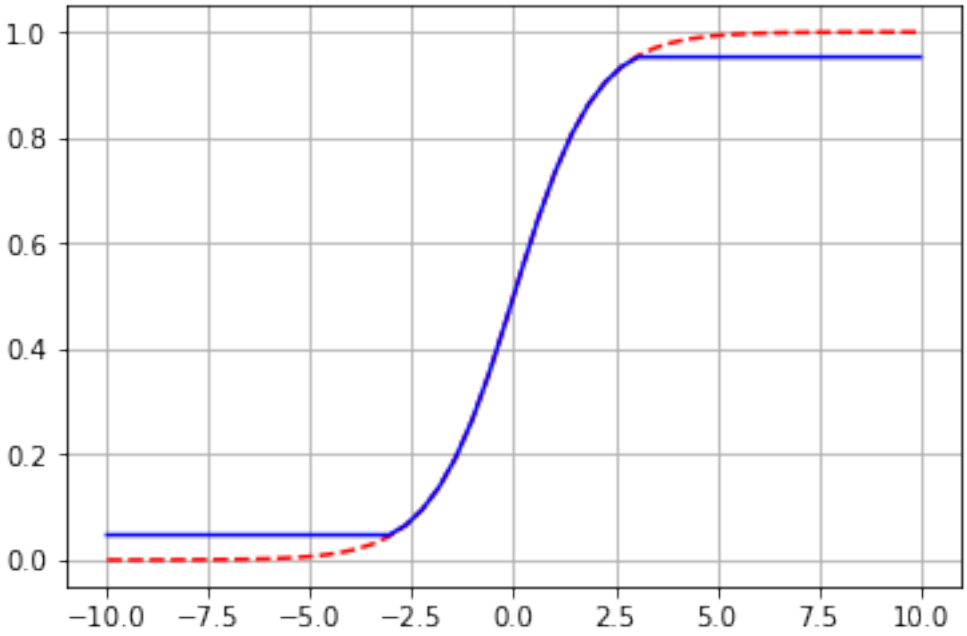
[19]:
```
sigmoid = lambda x: 0.5 * ( np.tanh(0.5 * x) + 1 )
# And there is no runtime warning this time.
print(sigmoid(x))
```

```
[0.        0.        0.        0.        0.9999546 0.7310586 0.9999546
 1.        1.        1.        1.       ]
```

1.3.5 Clipped sigmoid function

Suppose we cannot find a stable function definition. The sigmoid function can alternatively be approximated to avoid the overflow:

[23]:
```
x = np.linspace(-10,10,50)
plt.plot(x, sigmoid(x), 'r', ls='--')
plt.plot(x, sigmoid(np.clip(x, -3, 3)), 'b')
plt.grid()
```



The numpy clip function prevents the input from going outside a certain interval. This effectively avoids overflow in the exponential, but also causes a small approximation error.

1.3.6 Another source of overflow: normalizing probability distribution constant

Many probability functions can be written this way:

$$p(x) = \frac{1}{Z} \exp(f(x))$$

- Example of such functions: Gaussian, Multinomial, Dirichlet distribution.
- Machine learning algorithms often use these distributions, because their parameters can be learned easily. For example, the mean parameter of a Gaussian distribution can be estimated by computing the empirical mean of the data, and the scale parameter can be learned by computing the empirical standard deviation.
- On the other hand, these probability functions have a risk of overflow due to the exponential function.

1.3.7 Normalizing constant for discrete random variable disctributions

[24]:
```
# Let p(x) be a discrete distribution with function values
f = np.array([1.0, 8.0, 10.0, 0.1, 3.5, 2.3], dtype='float32')

# The normalization factor is the sum of these function values
# after application of the exponential function
Z = np.exp(f).sum()
f = np.exp(f)/Z
print(f.sum())
```

```
1.0
```

1.3.8 Check different (higher) function values

[25]:
```
f[2] = 100
Z = np.exp(f).sum()
#Even taking the logarithm of `Z` won't solve the overflow.
print(np.log(Z))
```

```
inf
```

Note: This problem will be studied in the homework.

1.4 Linear Algebra

Many machine learning techniques are based on linear algebra.

- Trace operator and matrix norm
- Solving the system of linear equations
- Linear regression
- Principal component analysis.

1.4.1 Trace operator

$$tr(X) = \sum_{i=1}^{N} X_{ii}, \quad X \in \mathbb{R}^{(N,M)}$$

Usefull Property:

$$tr(ABC) = tr(CAB) = tr(BCA)$$

[29]:
```
import numpy.random as rnd

X = rnd.normal(0,1,(50,50))
assert np.diag(X).sum() == np.trace(X)
```

[30]:
```
A = rnd.normal(0,1,(10,20))
B = rnd.normal(0,1,(20,10))
C = rnd.normal(0,1,(10,10))

assert np.allclose(np.trace(A @ B @ C),np.trace(C @ A @ B))
assert np.allclose(np.trace(C @ A @ B),np.trace(B @ C @ A))
```

1.4.2 Matrix L2 norm with trace operator

$$||X||_2 = \sqrt{tr(XX^\top)}$$

[32]:
```
import numpy.linalg as la

X = rnd.normal(0,1,(50,150))

assert np.allclose(la.norm(X), np.trace(X @ X.T)**.5)
```

1.4.3 Solving the system of linear equations

Like

$$3x_0 + x1 = 9, \quad x_0 + 2x_1 = 8$$

or

$$AX = B, \quad A = [[3,1],[1,2]] \quad B = [9,8]$$

[33]:
```
A = [[3,1],[1,2]]
B = [9,8]
X = la.solve(A,B)

assert np.allclose(A @ X, B)
```

1.4.4 Linear regression

The model assumes that the data is generated as following:

$$y_n = \beta_1 x_n + \beta_2 + \epsilon_n, \quad n = \overline{1..N}, \quad \epsilon_n \sim \mathcal{N}(0, \sigma^2)$$

or

$$y = \beta_1 \hat{X} + \beta_2 \mathbb{1}_N + \mathcal{E}, \quad \text{where} \quad y, \hat{X}, \mathcal{E} \in \mathbb{R}^{(N)}$$

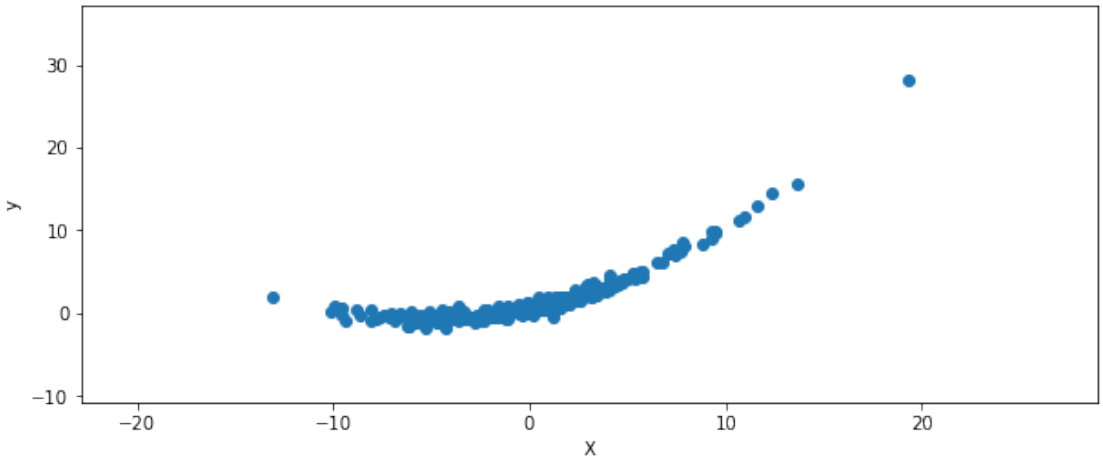
1.4.5 Linear regeression matrix form

$$y = [\beta_1, \beta_2] \times [\hat{X}, \mathbb{1}_N]^\top + \mathcal{E}$$

$$y = \beta \times X^\top + \mathcal{E}, \quad \text{where} \quad \beta = [\beta_1, \beta_2] \in \mathbb{R}^{(2)}, \quad \text{and} \quad X = [\hat{X}, \mathbb{1}_N] \in \mathbb{R}^{(N,2)}$$

Task Find the best linear fit of a labeled dataset.

[34]:
```
# Create a dataset
import numpy.random as rnd
rnd.seed(42)
N = 250 # number of data points
X = np.random.normal(0, 5, size = (N, 1))
X_ones = np.ones_like(X)
X = np.c_[(X, X_ones)] # column-wise concatenation

# Create targets (outputs) and make them depend on X in some way
y = 0.5 * X[:, 0] + 0.05 * X[:, 0] ** 2 + 0.5
sigma2 = 0.5
Eps = np.random.normal(0, sigma2, (N)) # random noise
y += Eps # add noise to the targets

plt.figure(figsize = (10,4))
plt.scatter(X[:, 0], y)
plt.xlabel('X')
plt.ylabel('y')
plt.margins(0.3)
```



1.4.6 Split dataset randomly into *train* and *test* datasets

[35]:
```
# set split ratio
split_ratio = 0.8 # 80 % for train and 20 % for test
idx = np.arange(N) # create all indexes
rnd.shuffle(idx) # shuffle them
split_idx =int(split_ratio*N)
# create train and test indexes
tr_idx = idx[:split_idx]
te_idx = idx[split_idx:]
```

```
# create Train dataset
X_tr = X[tr_idx]
y_tr = y[tr_idx]
# create Test dataset
X_te = X[te_idx]
y_te = y[te_idx]
```

[36]:
```
# Or use a function from sklearn package
from sklearn.model_selection import train_test_split
XX_tr, XX_te, yy_tr, yy_te = train_test_split(X,y,test_size=1-split_ratio,␣
↪random_state=42)
```

The solution of the linear regression model is given by:

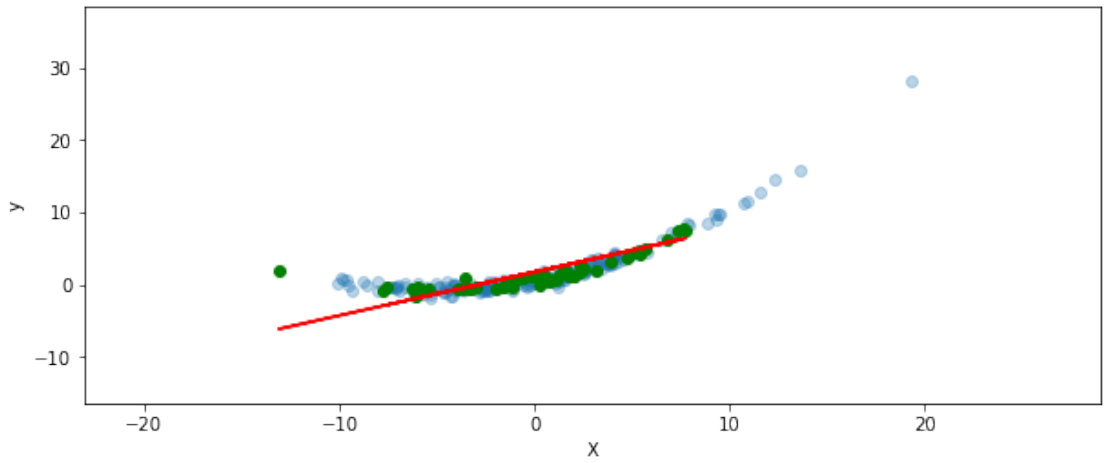
$$\boxed{\beta = (X_{\text{tr}}^\top X_{\text{tr}})^{-1} X_{\text{tr}}^\top y_{\text{tr}}}$$

And the prediction for new "test" points by:

$$\hat{y}_{\text{te}} = X_{\text{te}}\beta$$

1.4.7 Compute parameters of the linear regression model

[37]:
```
import numpy.linalg as la

beta = la.inv(X_tr.T @ X_tr) @ X_tr.T @ y_tr
y_te_predict = X_te @ beta

plt.figure(figsize = (10,4))
# Plot the data and the prediction
plt.scatter(X_tr[:, 0], y_tr, alpha=.3)
plt.scatter(X_te[:, 0], y_te, color='g')
plt.xlabel('X')
plt.ylabel('y')

plt.plot(X_te[:, 0],y_te_predict, '-', color='r')
plt.xlabel('X')
plt.ylabel('y')
plt.margins(0.3)
```



Compute the *root mean square error* (RMSE) for the predicted outputs:

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{n=1}^{N}(\hat{y}_{\text{te}_n} - y_{\text{te}_n})^2}$$

[38]:
```
rmse = np.square(y_te_predict - y_te).mean()**0.5

f"RMSE: {rmse:0.2f} where y_tr std: {np.std(y_tr):0.3f}"
```

[38]:
```
'RMSE: 1.62 where y_tr std: 3.534'
```

1.4.8 Principal component analysis (PCA)

PCA is a technique widely used for applications such as dimensionality reduction - lossy data compression - feature extraction - data visualization

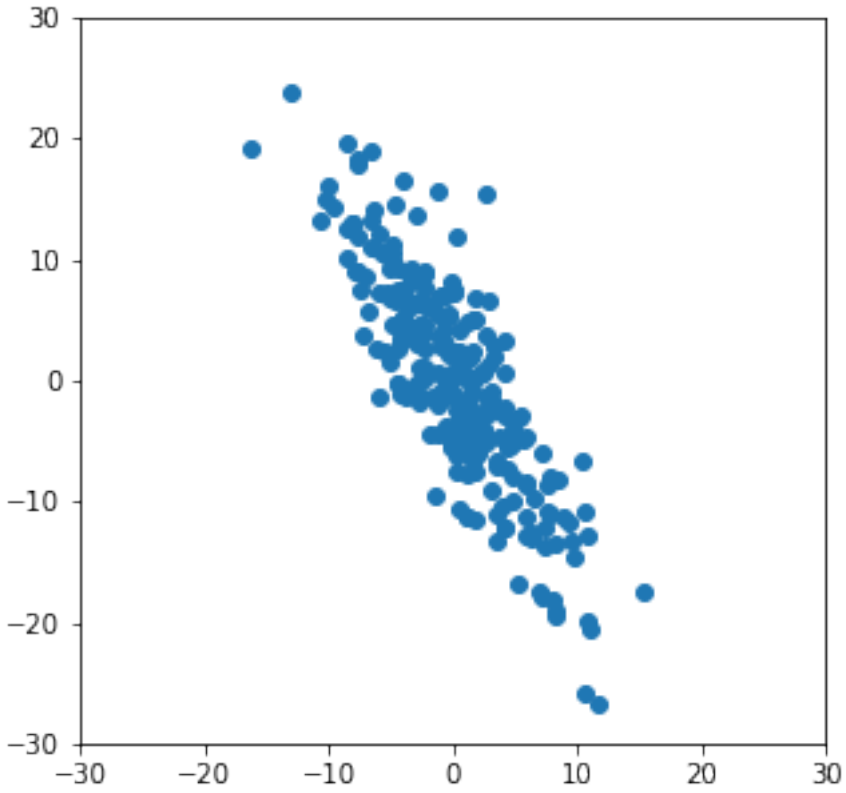
There are two commonly used definitions of PCA: - Orthogonal projection onto lower dimensional linear space such that the variance of projected data is maximized.

- Linear projection that minimizes the average projection cost, defined as the mean squared distance between the data and their projections.

1.4.9 Create a random dataset with correlated feature dependency

[39]:
```
rnd.seed(42)
N = 250
M = np.random.normal(0, 5, (N, 2))

# create some correlation between dimensions
M[:, 1] -= 1.5 * M[:, 0]
```

```
# Plot the centered dataset
plt.figure(figsize=(5, 5))
plt.scatter(*M.T)

_=plt.axis([-30, 30, -30, 30])
```



1.5 Singular value decomposition (SVD)

The Singular-Value Decomposition, or SVD for short, is a matrix decomposition method for reducing a matrix to its constituent parts in order to make e.g. certain subsequent matrix calculations like matrix inversion simpler.

1.5.1 Find the principal components via SVD

[40]:
```
# PCA only applies to centered data, so we center the data
M -= M.mean(axis=0)

U, Sigma, V = la.svd(M , full_matrices = False)
print("U shape: ", U.shape)
print("Sigma shape:", Sigma.shape)
print("V shape:", V.shape)
```

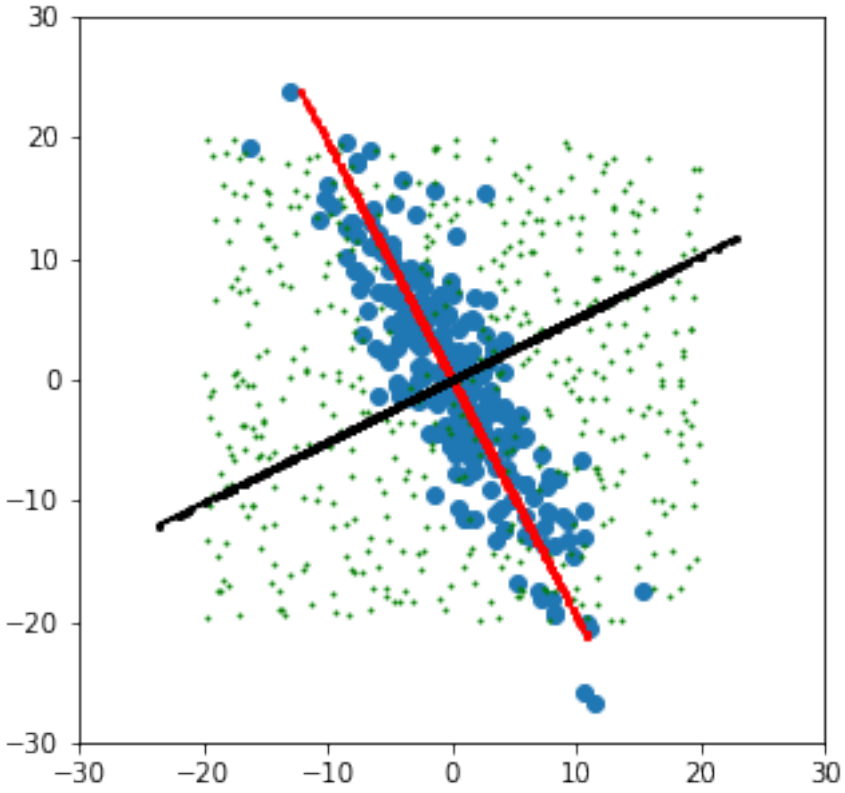
```
X = U @ np.diag(Sigma) @ V.T # SVD reconstruction

assert np.allclose(np.sum(M - X) , np.zeros_like(M))
```

```
U shape:  (250, 2)
Sigma shape: (2,)
V shape: (2, 2)
```

1.5.2 Project any pointa to the axis representing highest data variance

[42]:
```
HAT1 = np.outer(V[0], V[0]) # 1st component
HAT2 = np.outer(V[1], V[1]) # 2d component

Mtest = rnd.uniform(-20, 20, (500, 2))
# Project some test data onto the fisrt principal compenent
MtestPCA1 = Mtest @ HAT1
# Project some test data onto the second principal compenent
MtestPCA2 = Mtest @ HAT2
# Plot the original data and the projected test data
plt.figure(figsize=(5, 5))
plt.scatter(*M.T)  # equivalent to M.T[0], M.T[1]
plt.plot(*Mtest.T, 'o', color='g', ms=1)
plt.plot(*MtestPCA1.T, 'o-', color='r', ms=2)
plt.plot(*MtestPCA2.T, 'o-', color='k', ms=2)
plt.axis([-30, 30, -30, 30])
```

[42]:
```
[-30, 30, -30, 30]
```


30
20
10
0
−10
−20
−30
−30
−20
−10
0
10
20
30