Orthogonal Matching Pursuit

Group 5

Mekha Rajesh

Sharan Ravula

Md Fayek Sharaf

Shane Wojcicki

Linear Algebra
$Ax = b$

Linear Algebra

Matrix equation → $\mathbf{Ax} = \mathbf{b}$

Equation # 1 →
Equation # 2 →
Equation # 3 →

$$\begin{bmatrix} 1 & 5 & 3 \\ 2 & 8 & 6 \\ 4 & 9 & 7 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 10 \\ 12 \\ 6 \end{bmatrix}$$

Three unknowns

Linear Algebra

Matrix equation →

$$\mathbf{Ax} = \mathbf{b}$$

Equation # 1 →
Equation # 2 →
Equation # 3 →

$$\begin{bmatrix} 1 & 5 & 3 \\ 2 & 8 & 6 \\ 4 & 9 & 7 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 10 \\ 12 \\ 6 \end{bmatrix}$$

Three equations →

$$1x_1 + 5x_2 + 3x_3 = 10$$
$$2x_1 + 8x_2 + 6x_3 = 12$$
$$4x_1 + 9x_2 + 7x_3 = 6$$

Linear Algebra

$$\mathbf{Ax} = \mathbf{b}$$

$$\begin{bmatrix} 1 & 5 & 3 \\ 2 & 8 & 6 \\ 4 & 9 & 7 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 10 \\ 12 \\ 6 \end{bmatrix}$$

(# of equations) = (# of unknowns)

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -4 \\ 4 \\ -2 \end{bmatrix}$$

```
x = np.linalg.solve(A,b)
```

Linear Algebra

What if !!!

(# of equations) ≠ (# of unknowns)

A^+ is pseudoinverse

$$\mathbf{A}^{+}\boldsymbol{x} \approx \mathbf{b}$$

$$\begin{bmatrix} 1 & 5 & 3 \\ 2 & 8 & 6 \\ 4 & 9 & 7 \\ 3 & 2 & 4 \\ 10 & 5 & 7 \\ 5 & 1 & 1 \\ 6 & 3 & 8 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 10 \\ 12 \\ 6 \\ 4 \\ 24 \\ 8 \\ 20 \end{bmatrix}$$

Overdetermined system

Linear Algebra

What if !!!

(# of equations) ≠ (# of unknowns)

$$\begin{bmatrix} 1 & 5 & 3 \\ 2 & 8 & 6 \\ 4 & 9 & 7 \\ 3 & 2 & 4 \\ 10 & 5 & 7 \\ 5 & 1 & 1 \\ 6 & 3 & 8 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 10 \\ 12 \\ 6 \\ 4 \\ 24 \\ 8 \\ 20 \end{bmatrix}$$

$$\mathbf{A}^{+}\boldsymbol{x} \approx \mathbf{b}$$

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1.46919879 \\ -0.16811539 \\ 1.19239733 \end{bmatrix}$$

```
x = np.linalg.lstsq(A,b)
```

Solve using Least Squares

Orthogonal Matching Pursuit algorithm

$$\mathbf{A}^{+}\boldsymbol{x} \approx \mathbf{b}$$

$$y \approx \mathbf{A}^{+}\boldsymbol{x}$$

Column signal or Signal

Dictionary or measuring matrix or sensing matrix

Paper notation →

$$x \approx D\gamma$$

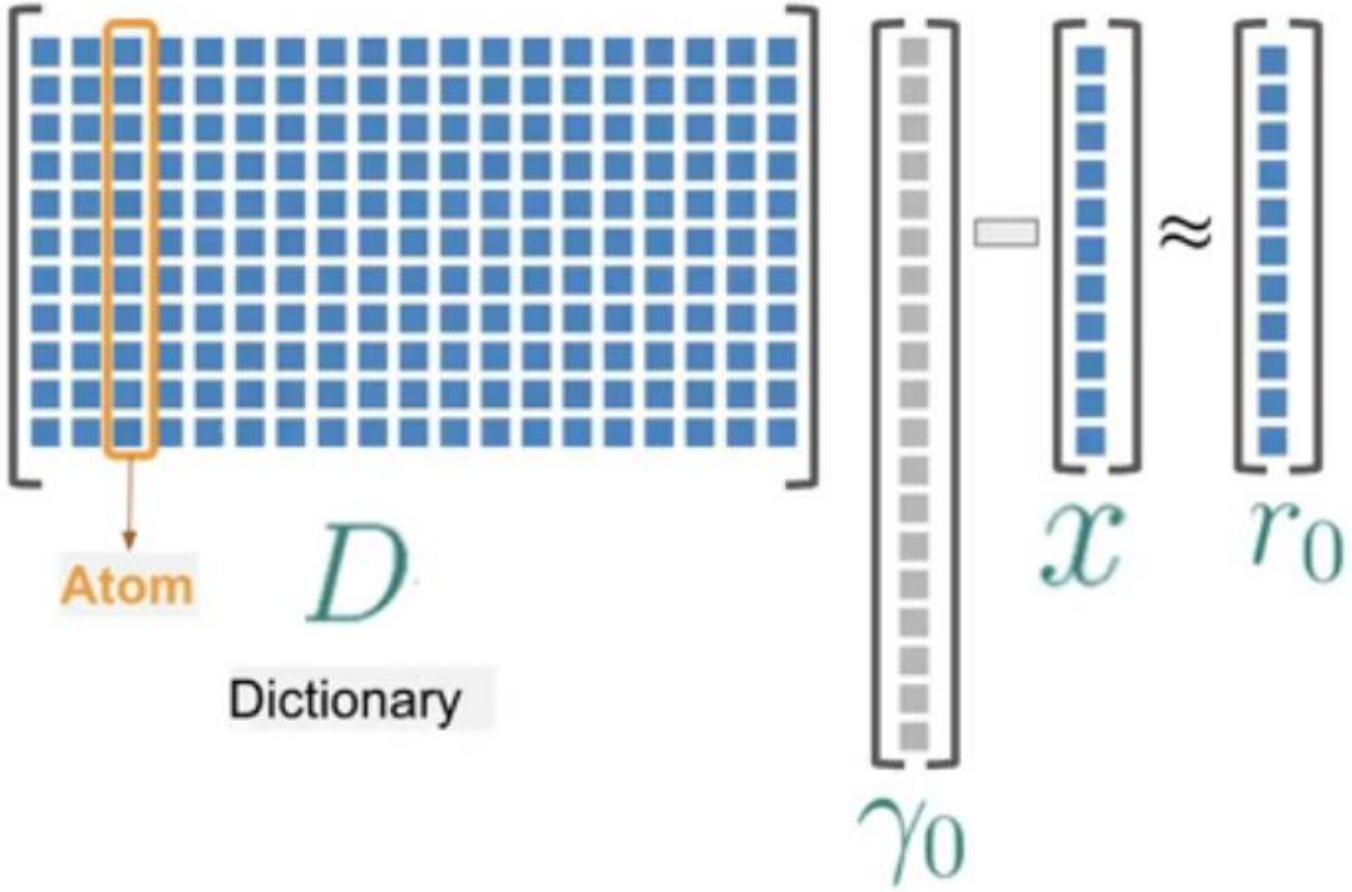
Sparse approximation or sparse representation

Scikit-learn notation →

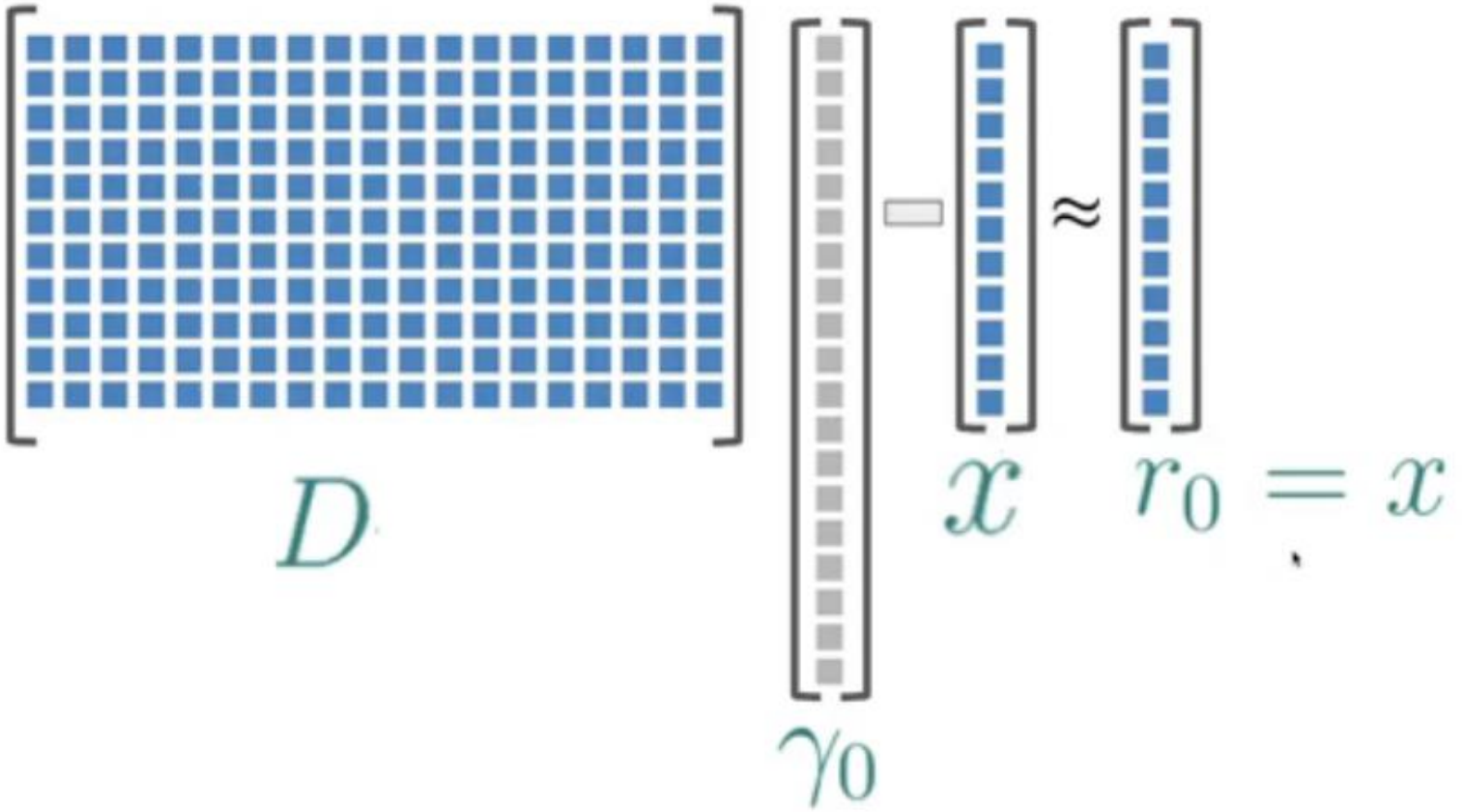
$$y \approx Xw$$

Ron Rubinstein et.al. Efficient Implementation of the K-SVD Algorithm using Batch Orthogonal Matching Pursuit.
Source: https://www.cs.technion.ac.il/~ronrubin/Publications/KSVD-OMP-v2.pdf

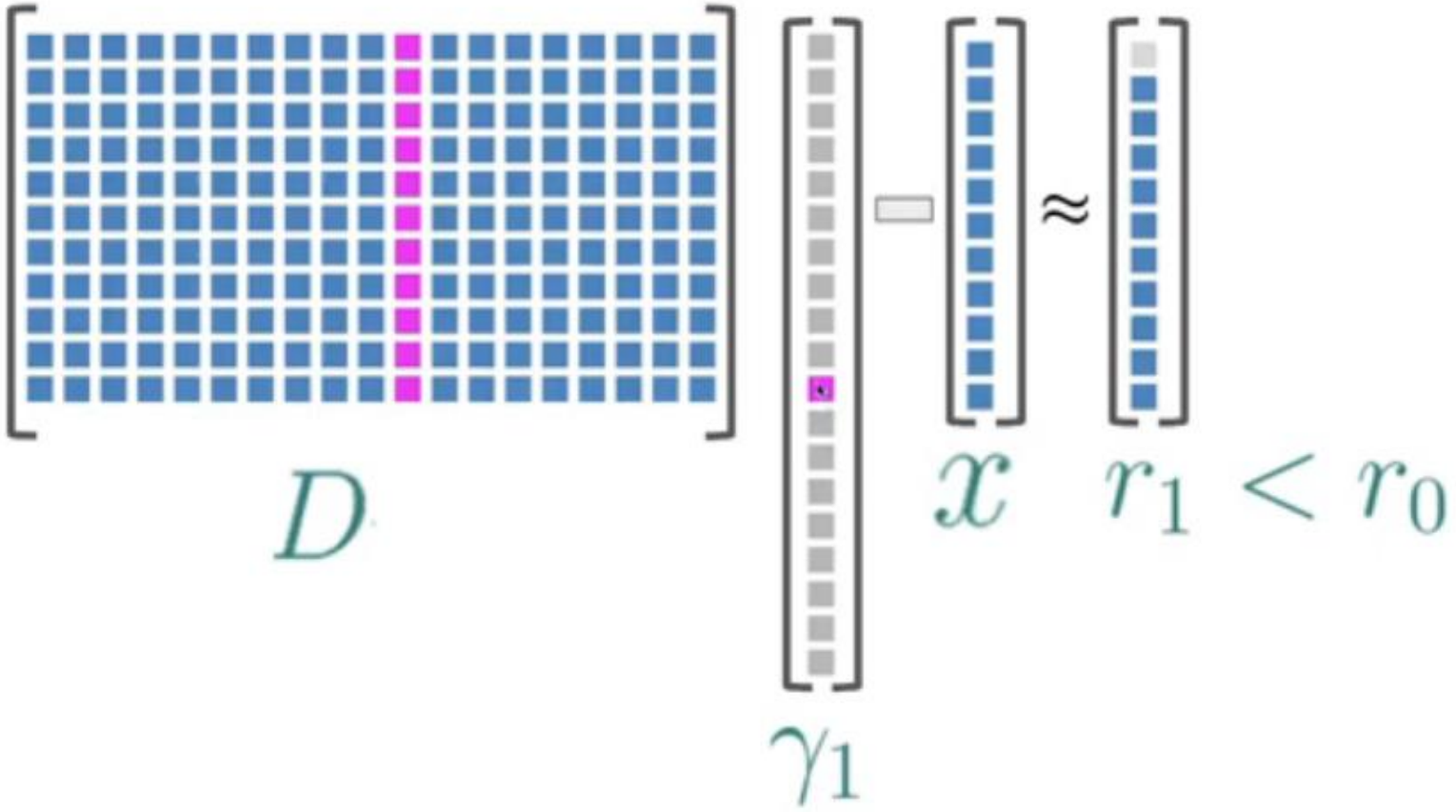
Orthogonal Matching Pursuit algorithm



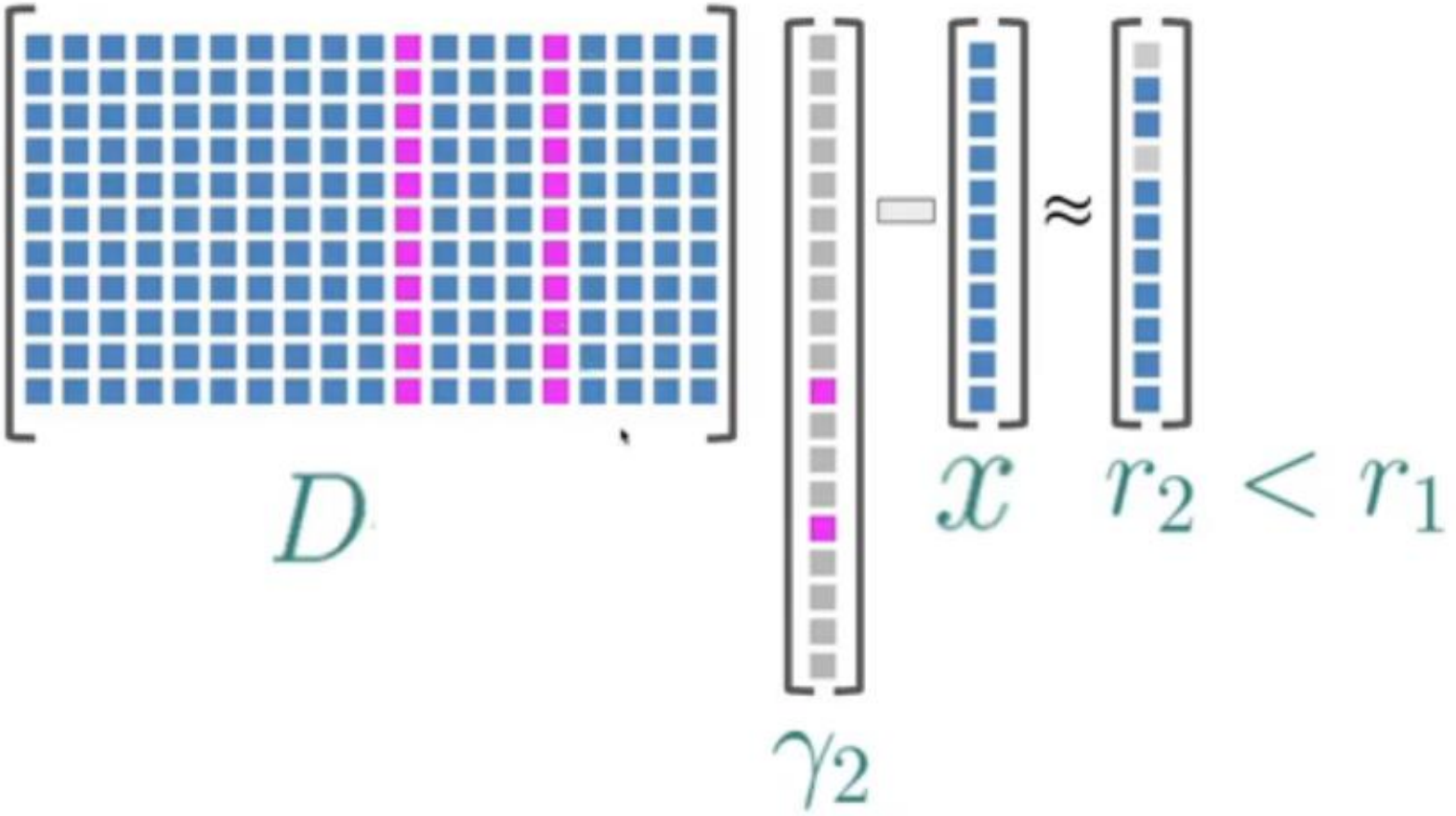
Orthogonal Matching Pursuit algorithm



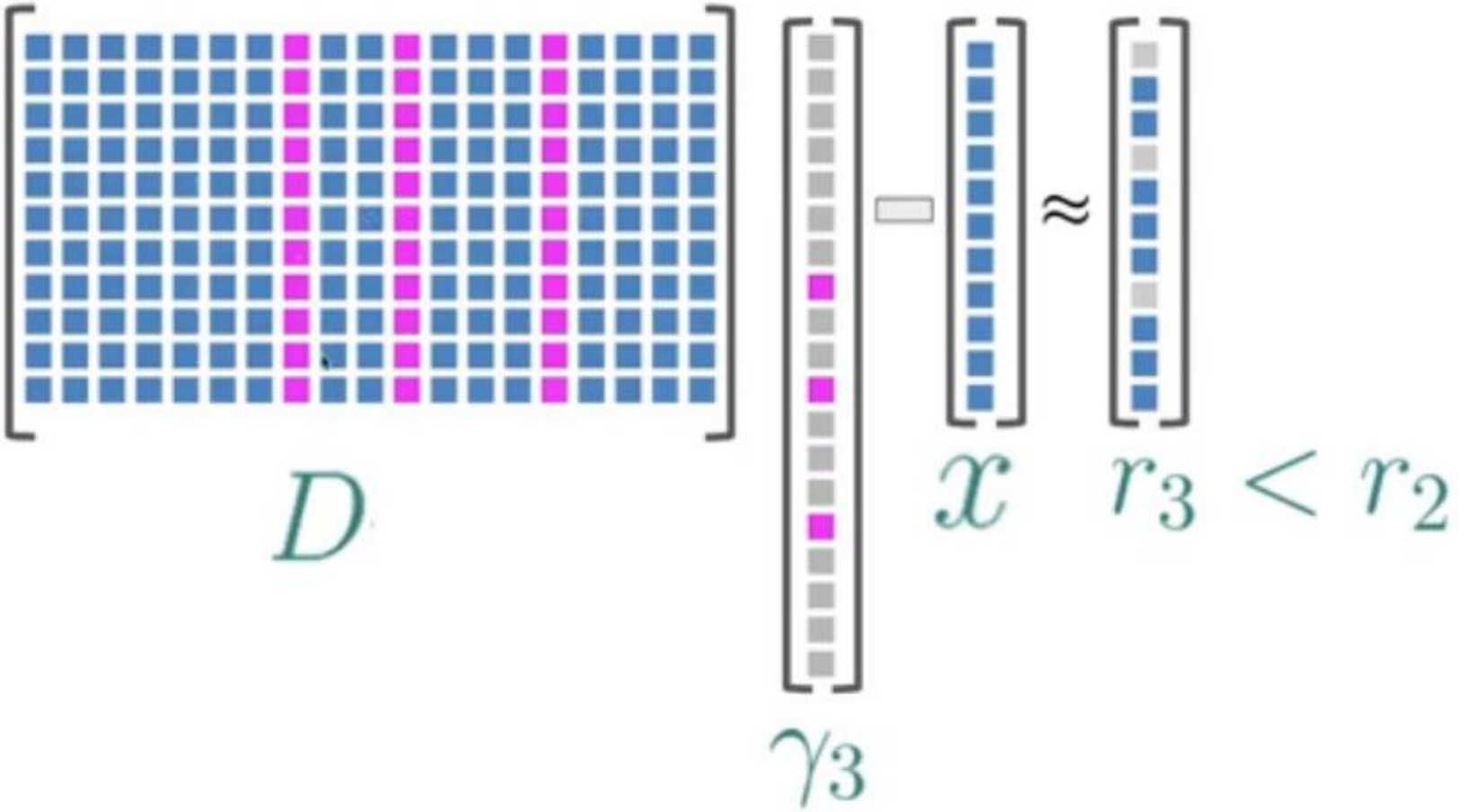
Orthogonal Matching Pursuit algorithm



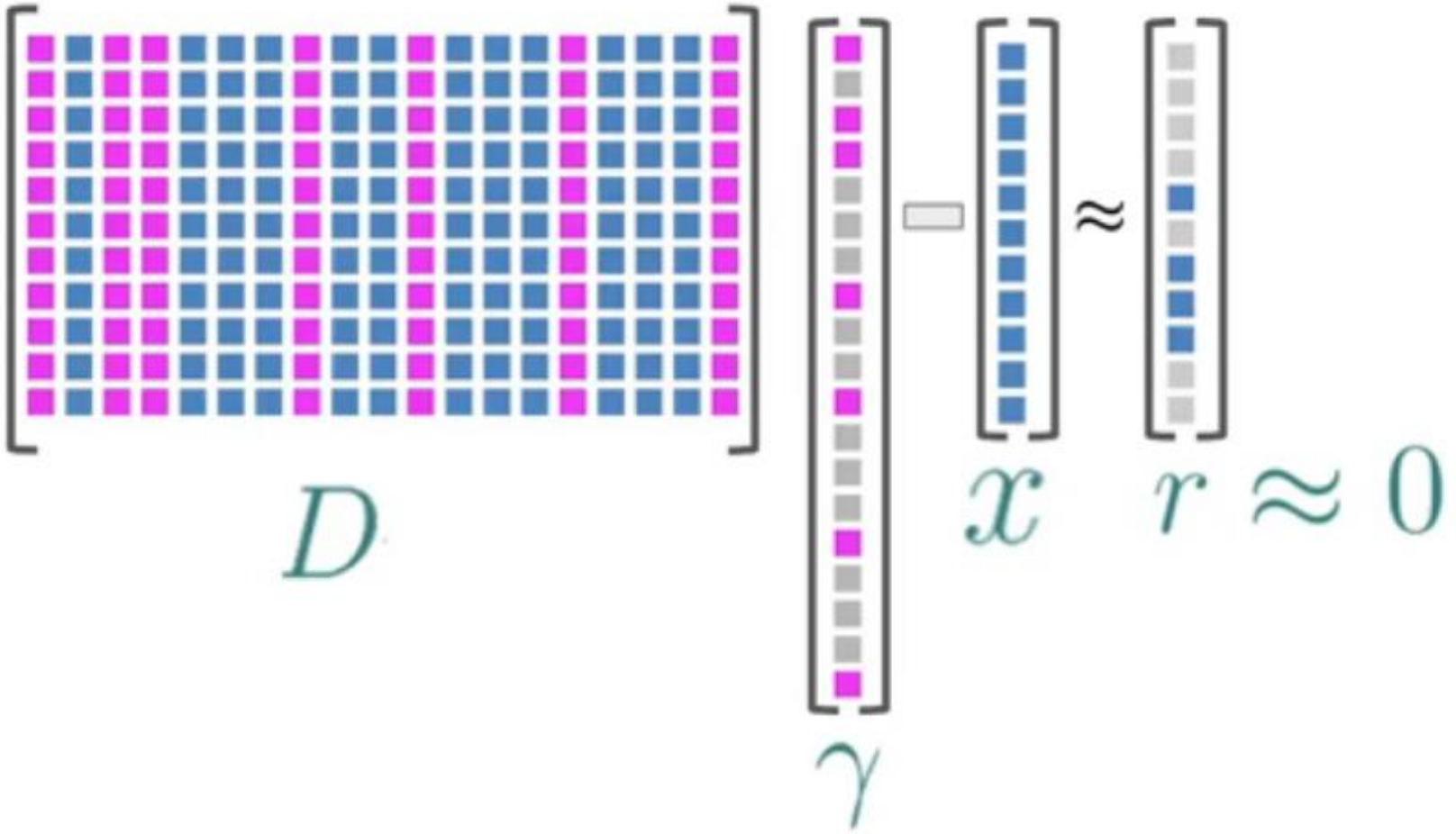
Orthogonal Matching Pursuit algorithm



Orthogonal Matching Pursuit algorithm



Orthogonal Matching Pursuit algorithm



Coding & Graphs Using OMP (Orthogonal matching pursuit)

Orthogonal Matching Pursuit Using Scikit-learn

Below code uses scikit-learn to apply Orthogonal Matching Pursuit (OMP) for sparse signal recovery. It generates a random dictionary matrix and a sparse signal, then observes the signal. OMP is applied with a specified number of expected non-zero coefficients. The results include the estimated coefficients and the support set (indices of non-zero coefficients), providing information about the original and estimated sparse signals.

```python
from sklearn.linear_model import OrthogonalMatchingPursuit
import numpy as np

# Step 1: Generate or Load Data
# Let's create a dictionary (matrix Phi) and a sparse signal (vector x)
# Dictionary matrix with 10 atoms of dimension 20
Phi = np.random.randn(10, 20)
x = np.zeros(20)
# Sparse signal with non-zero coefficients at indices 2, 5, and 8
x[[2, 5, 8]] = np.random.randn(3)
# Observed signal
y = np.dot(Phi, x)

# Step 2: Apply Orthogonal Matching Pursuit
# Specify the expected number of non-zero coefficients
omp = OrthogonalMatchingPursuit(n_nonzero_coefs=3)
omp.fit(Phi, y)

# Step 3: Get Results
coefficients = omp.coef_  # Estimated coefficients
# Indices of non-zero coefficients
support_set = np.where(coefficients != 0)[0]

# Display Results
print(&quot;Original Sparse Signal:\n&quot;, x)
print(&quot;\nEstimated Sparse Signal:\n&quot;, coefficients)
print(&quot;\nIndices of Non-Zero Coefficients (Support Set):&quot;, support_set)
```

Output:

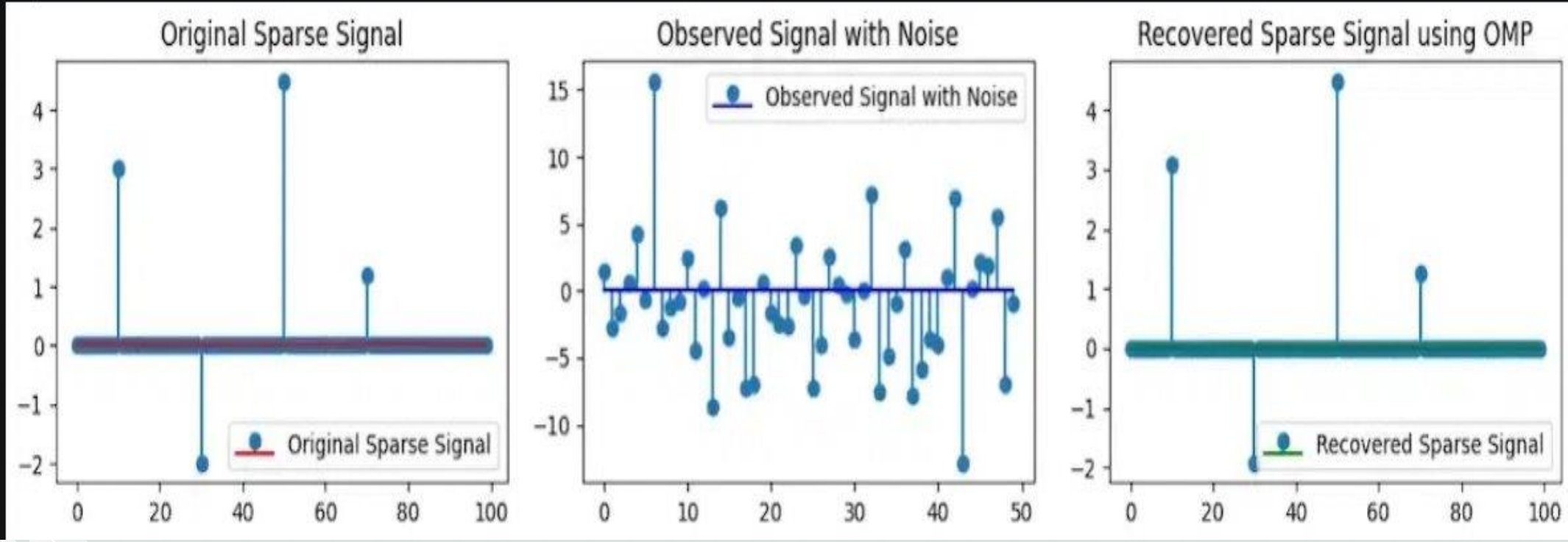
```
Original Sparse Signal:
 [ 0.          0.         -0.60035931  0.          0.          0.06069191
  0.          0.         -0.65530325  0.          0.          0.
  0.          0.          0.          0.          0.          0.
  0.          0.        ]

Estimated Sparse Signal:
 [-0.14809913  0.          0.          0.          0.          0.
  0.          0.         -1.03167779  0.          0.         -0.16831767
  0.          0.          0.          0.          0.          0.
  0.          0.        ]

Indices of Non-Zero Coefficients (Support Set): [ 0  8 11]
```

Sparse Signal Recovery using OMP (Orthogonal Matching Pursuit)

Below code illustrates sparse signal recovery using Orthogonal Matching Pursuit (OMP). It generates a sparse signal with non-zero coefficients, creates an observed signal by multiplying it with a random matrix (dictionary), and applies OMP for signal recovery. The recovered and true sparse signals are then plotted for visual comparison.

```python
from sklearn.linear_model import OrthogonalMatchingPursuit
import numpy as np
import matplotlib.pyplot as plt

# Set random seed for reproducibility
np.random.seed(42)

# Step 1: Generate a sparse signal
signal_length = 100
sparse_signal = np.zeros(signal_length)
sparse_signal[[10, 30, 50, 70]] = [3, -2, 4.5, 1.2]  # Non-zero coefficients

# Step 2: Generate a measurement matrix (dictionary)
measurement_matrix = np.random.randn(50, signal_length)

# Step 3: Create the observed signal with noise
noise_level = 0.5
observed_signal = np.dot(measurement_matrix, sparse_signal) + noise_level * np.random.randn(50)

# Step 4: Apply OMP for signal recovery
omp = OrthogonalMatchingPursuit(n_nonzero_coefs=4)
omp.fit(measurement_matrix, observed_signal)
recovered_signal = omp.coef_

# Step 5: Plot the results
plt.figure(figsize=(12, 3))

# Original Sparse Signal
plt.subplot(1, 3, 1)
plt.stem(sparse_signal, basefmt='r', label='Original Sparse Signal')
plt.title('Original Sparse Signal')
plt.legend()

# Observed Signal
plt.subplot(1, 3, 2)
plt.stem(observed_signal, basefmt='b', label='Observed Signal with Noise')
plt.title('Observed Signal with Noise')
plt.legend()

# Recovered Sparse Signal
plt.subplot(1, 3, 3)
plt.stem(recovered_signal, basefmt='g', label='Recovered Sparse Signal')
plt.title('Recovered Sparse Signal using OMP')
plt.legend()

plt.tight_layout()
plt.savefig('omp.png')
plt.show()
```

Output:

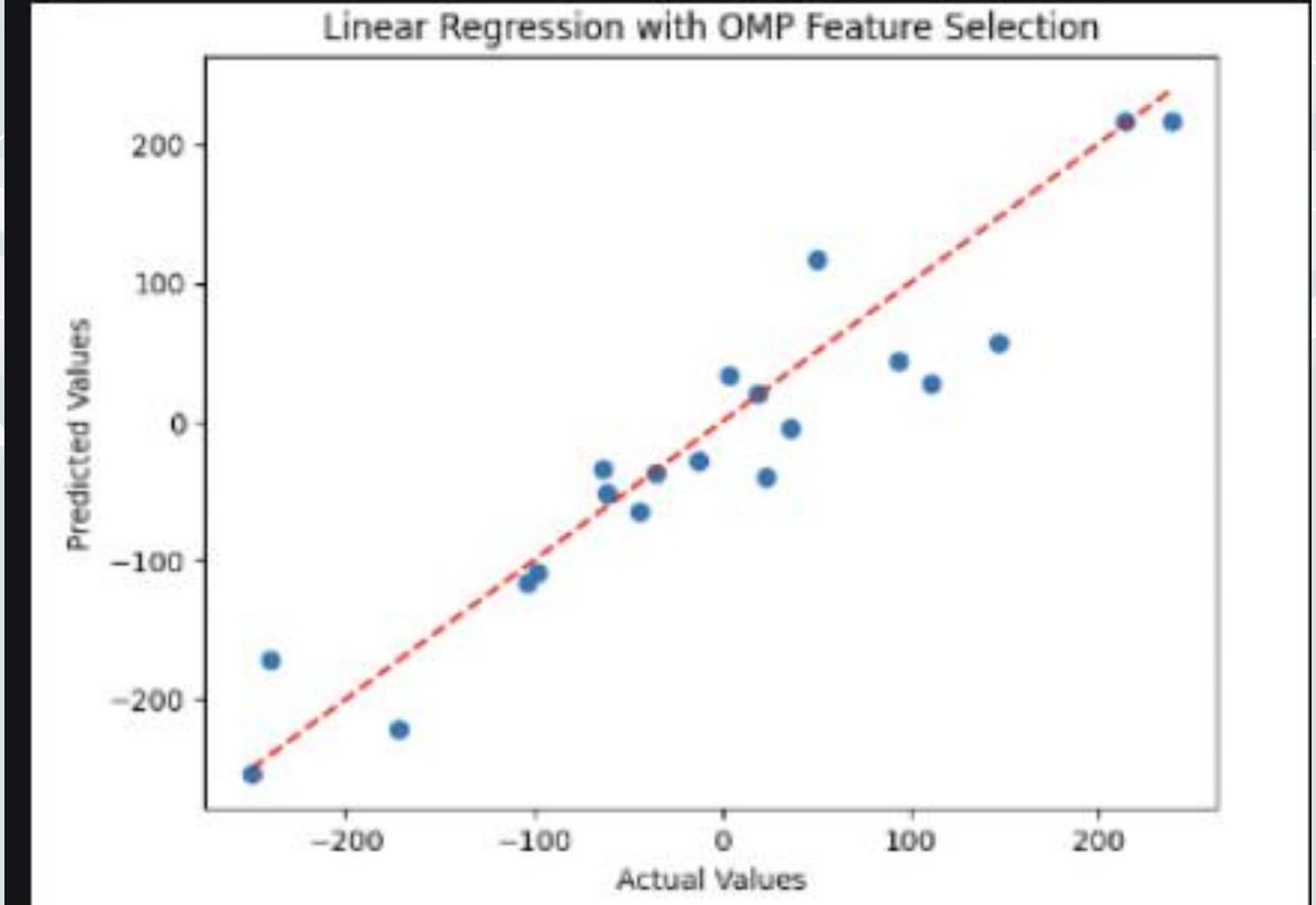

Original Sparse Signal
Original Sparse Signal
Observed Signal with Noise
Observed Signal with Noise
Recovered Sparse Signal using OMP
Recovered Sparse Signal

OMP (Orthogonal Matching Pursuit) for Feature Selection in Linear Regression

Below code showcases a regression workflow with scikit-learn. It generates synthetic data, splits it, applies Orthogonal Matching Pursuit for feature selection, trains a linear regression model, and evaluates its performance on a test set. The script concludes with a scatter plot visualizing the accuracy of the regression model.

```python
from sklearn.linear_model import OrthogonalMatchingPursuit
from sklearn.datasets import make_regression
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
import numpy as np
import matplotlib.pyplot as plt

# Generate synthetic data for regression
X, y = make_regression(n_samples=100, n_features=20, noise=5, random_state=42)

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Apply OMP for feature selection
omp = OrthogonalMatchingPursuit(n_nonzero_coefs=5)
omp.fit(X_train, y_train)

# Train a linear regression model using the selected features
selected_features = np.where(omp.coef_ != 0)[0]
X_train_selected = X_train[:, selected_features]
X_test_selected = X_test[:, selected_features]

lr = LinearRegression()
lr.fit(X_train_selected, y_train)

# Evaluate the model on the test set
y_pred = lr.predict(X_test_selected)

# Plot the predicted vs actual values
plt.scatter(y_test, y_pred)
plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], linestyle='--', color='red')
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.title('Linear Regression with OMP Feature Selection')
plt.show()
```

Output:


Linear Regression with OMP Feature Selection
Predicted Values
Actual Values
200
100
0
−100
−200
−200
−100
0
100
200

How Does OMP Work

Psuedo Code

Algorithm 1 ORTHOGONAL MATCHING PURSUIT

Input: Dictionary $\mathbf{D}$, signal $\underline{x}$, target sparsity K *or* target error ϵ
Output: Sparse representation $\underline{\gamma}$ such that $\underline{x} \approx \mathbf{D}\underline{\gamma}$
Init: Set $I := (\,)$, $\underline{r} := \underline{x}$, $\underline{\gamma} := \underline{0}$
while (*stopping criterion not met*) **do**
    $\hat{k} := \underset{k}{\text{Argmax}} \; |\, \underline{d}_k^T \, \underline{r} \,|$
    $I := (\, I, \, \hat{k} \,)$
    $\underline{\gamma}_I := (\mathbf{D}_I)^+ \underline{x}$
    $\underline{r} := \underline{x} - \mathbf{D}_I \underline{\gamma}_I$
end while

The problem we are solving is:

$$\mathbf{y} = \mathbf{X}\mathbf{w}$$

- $\mathbf{y} \in \mathbb{R}^N$: Observed vector (generated by $\mathbf{X}\mathbf{w}$).
- $\mathbf{X} \in \mathbb{R}^{N \times L}$: Sensing matrix with N rows and L columns.
- $\mathbf{w} \in \mathbb{R}^L$: Unknown sparse vector (has k_0 nonzero entries).

Step 1: Initialization

- **Residual**: Initialize with the observed vector:

$$\mathbf{r}_0 = \mathbf{y}$$

- **Solution estimate**:

$$\theta = \mathbf{0}$$

- **Support set**:

$$S = \emptyset$$

```
[7]  #Orthogonal Matching Pursuit (OMP)##

[3]  import numpy as np

     L = 20 # dimension of the unknown vector w
     k0 = 3 # assume w is k0-sparse
     w = np.zeros(L)
     rgn = np.random.RandomState(0)

[5]  # randomly choose k0 entries, and randomly assign values
     w[rgn.randint(0,L,k0)] = rgn.normal(loc=0.0,scale=1.0,size=k0)

[6]  N = 20 # dimension of the sensing matrix
     X = rgn.normal(loc=0.0,scale=1.0,size=(N,L))
     y = X.dot(w)

[8]  ##The main algorithm##

[9]  def corr(x,y):
         return abs(x.dot(y))/np.sqrt((x**2).sum())

[10] err_tol = 0.0001
     theta = np.zeros(L)
     error = y
     S = [] # the support
     ii = 0

     while np.linalg.norm(error) > err_tol:
         # find the column has maximum correlation with the error
         corrs = [ corr(x,error) for x in X.T]
         ind = np.argmax(corrs)
         S.append(ind)
         #print(S)
         X_active = X[:,S]
         # LS estimate using active support
         theta_tilde = np.linalg.inv(X_active.T.dot(X_active)).dot(X_active.T).dot(y)

         # insert estimated theta into the correct location
         theta[S] = theta_tilde

         # update the error vector
         error = y-X.dot(theta)
         ii+=1
```

At each iteration, compute the correlation between the columns of $\mathbf{X}$ and the current residual:

$$c_j = \frac{|\mathbf{x}_j^\top \mathbf{r}_t|}{\|\mathbf{x}_j\|}$$

- $\mathbf{x}_j$: The j-th column of $\mathbf{X}$.
- c_j: Correlation of column $\mathbf{x}_j$ with the residual.
- Select the column with the maximum correlation:

$$j^* = \arg\max_j c_j$$

Update the support set:

$$S = S \cup \{j^*\}$$

With the support set S, extract the active columns of $\mathbf{X}$:

$$\mathbf{X}_S = [\mathbf{x}_{j_1}, \mathbf{x}_{j_2}, \ldots, \mathbf{x}_{j_t}]$$

Solve the least squares problem:

$$\hat{\theta}_S = (\mathbf{X}_S^\top \mathbf{X}_S)^{-1} \mathbf{X}_S^\top \mathbf{y}$$

Here:

- θ_S: Coefficients corresponding to the support set S.
- Update the solution θ:

$$\theta[S] = \hat{\theta}_S$$

Update the residual using the current estimate:

$$\mathbf{r}_{t+1} = \mathbf{y} - \mathbf{X}\theta$$

This ensures that the algorithm works iteratively to refine the solution.

The final sparse estimate θ gives the reconstructed sparse vector:

$$\hat{\mathbf{w}} = \theta$$

If we want to see the output of our OMP

Initially it was theta

Our Model(OMP) recovers the initial values very near.

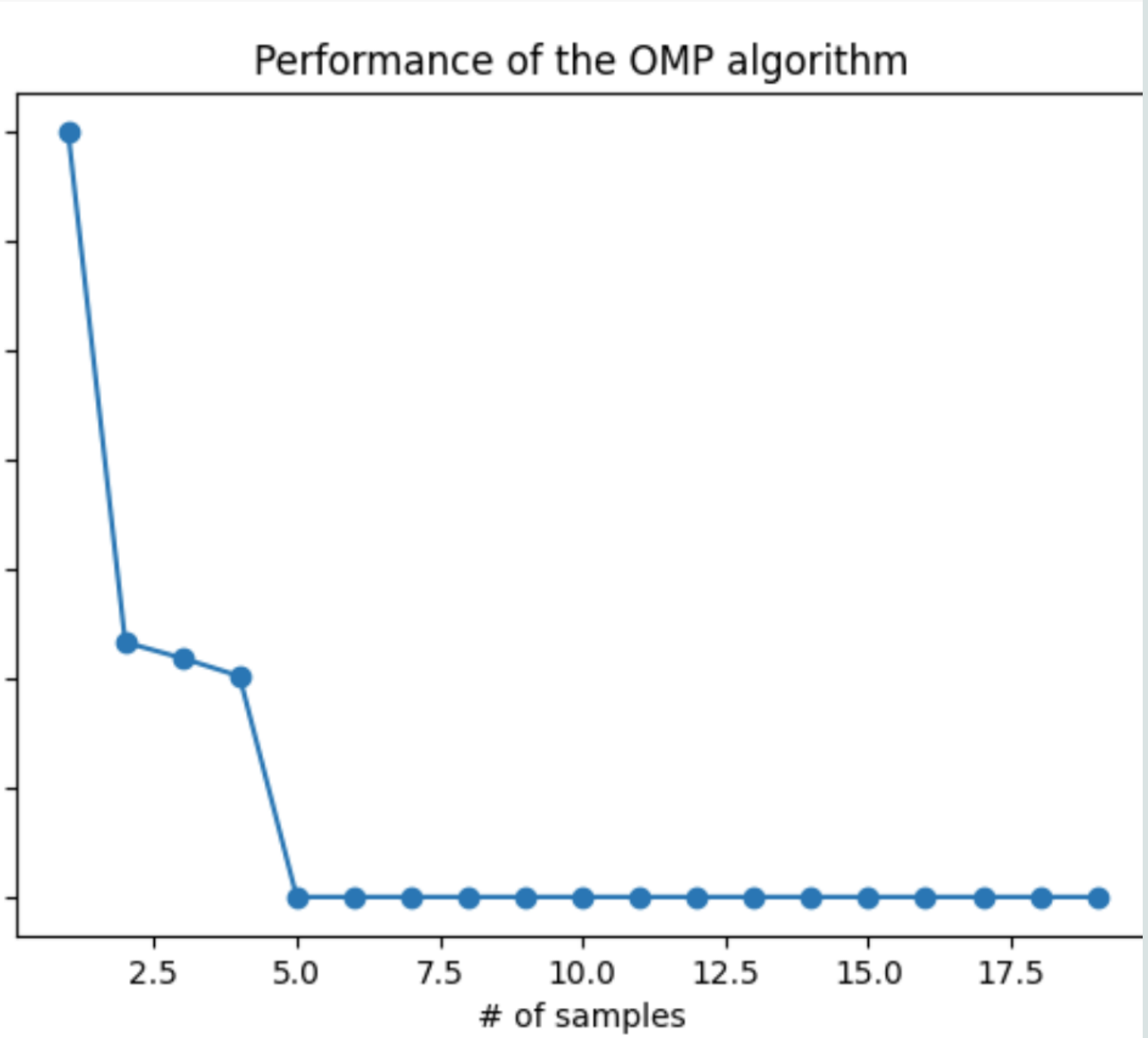
We can think of it as Labels. (Machine Learning)

```
theta
```

```
array([0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 1.76405235,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.97873798, 0.40015721])
```

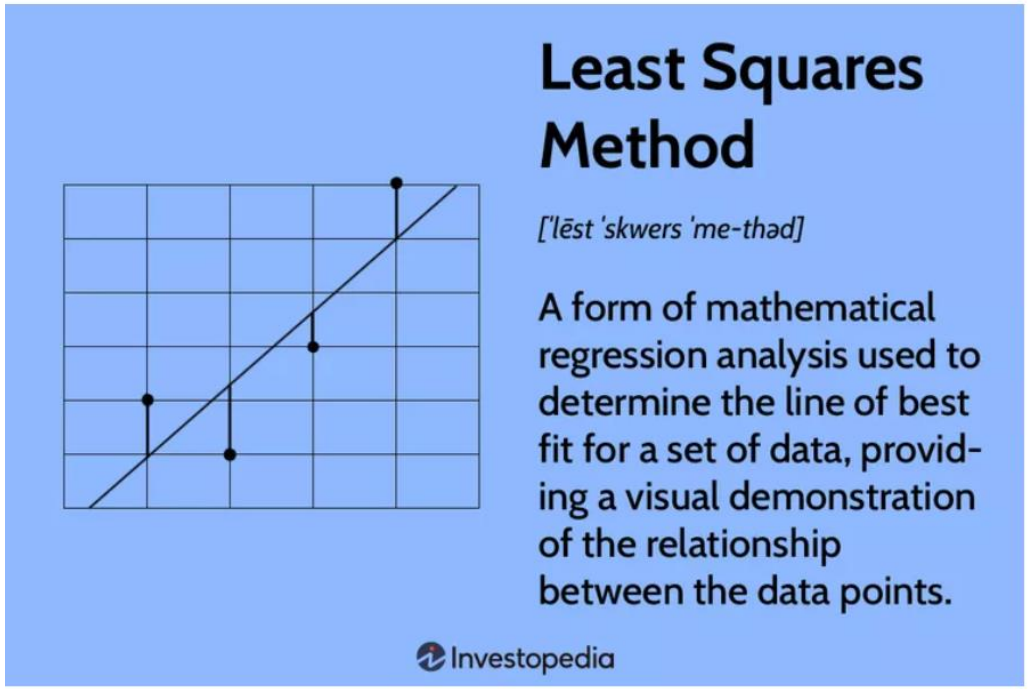
```
[12] w
```

```
array([0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 1.76405235,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.97873798, 0.40015721])
```



Efficiency of Our Algorithm.

As we can see the numbers of samples/number of iteration increases our model algorithm/model started showing up some low error



How They are related?

Y = mx + b

From given data try to approximate label.

Fitting Data to a model

Least Square Method

Minimize the Error/ Loss function.

5.2.1 Orthogonal matching pursuit

The orthogonal matching pursuit (OMP) algorithm [7] is a methodology to solve NP-hard sparsity problems. OMP (illustrated in Algorithm 1) builds a sparse solution to a given signal by iteratively building up an approximation; the vector $\boldsymbol{y}$ (usually patches in image synthesis) is approximated as a linear combination of a few atoms (the columns d_j) of D (representative patterns), where the active set of atoms to be used is built atom by atom in a greedy fashion. The support L is defined as the indices of non-zero components in $\boldsymbol{x}$. At each iteration, a new atom that best correlates with the current residual is added to the active set. Initially, the sparse coefficient is equal to zero $x = 0$ and iteratively a k-term approximation to the signal $\boldsymbol{y}$ is approximated by maintaining a set of active atoms (initially empty), and expanding the set by one additional atom at each iteration. Each iteration stops until a pre-defined number of atoms in the active set is reached by the sparsity level. The entire process stops when a residual error threshold (ϵ) is reached. The term D^+ stands for matrix pseudo-inversion which can be computed as $D^+ = (D^T D)^{-1} D^T$.

Field of Apply

Machine Learning Linear Regression

We will talk about it in later Slides.

Real-world Applications of the Algorithm

RECOVERING DATA

FROM NOISY MEASUREMENTS

Given a set of data which was collected, but is noisy, the OMP algorithm can be used to reconstruct the data to denoise it. The algorithm would be given the data collected (Y) and a measurement matrix (Φ) specific to the use case (i.e. sampling pattern, fourier transform, etc.), and attempts to solve Y = Φx, where x is the corrected data.

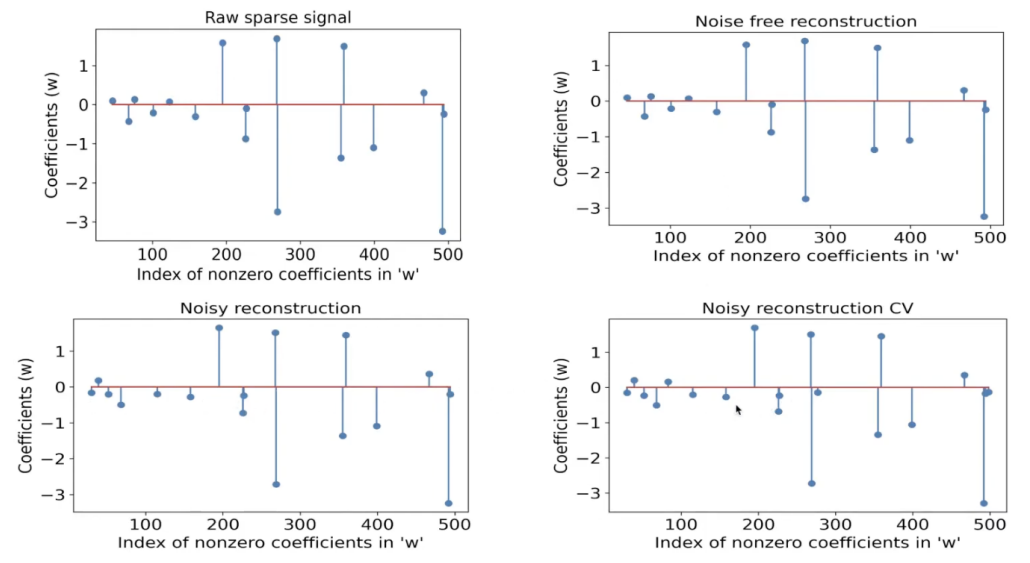
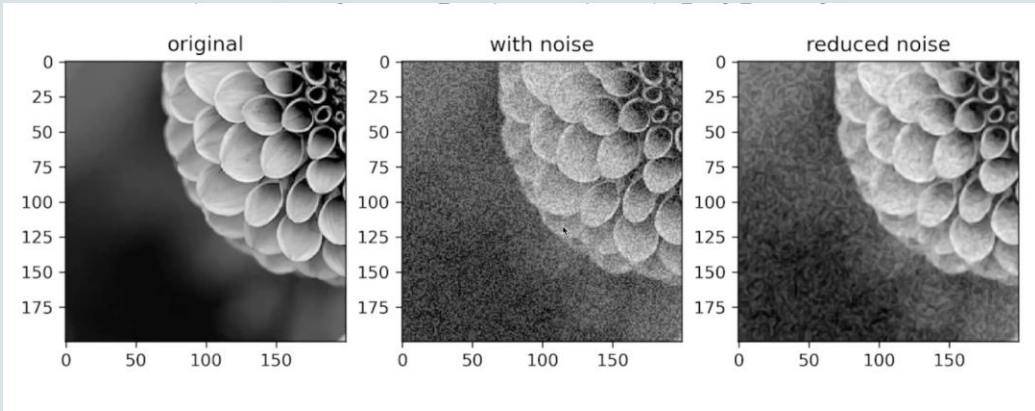


IMAGE DENOISING

Image denoising follows a similar process to correct a noisy image. The process works because noise typically does not have a sparse representation in most transform domains, so it is naturally ignored by the OMP algorithm. The algorithm receives the image (Y) and a transform (Φ) used for the denoising process (commonly a DCT or wavelet transform, or a learned dictionary for the sensor).



My Image De-noising Program in Python

- I hid some helper functions as their implementation was not relevant to the example.
- The image is divided into its three red, green, and blue color channels which are processed separately.
- Each channel (Y) is then divided into small, overlapping patches.
- Each patch is approximated as a sparse combination of basis vectors from a dictionary (Φ). The dictionary code is not important; I used OpenCV's python library to generate it, I'm not sure how it works.
- After each patch is processed, the denoised patches are placed back into their original positions in the image. Overlapping patches are averaged to ensure a smooth reconstruction.
- Now each channel is denoised, and the channels are combined to create the processed image.

```python
# Orthogonal Matching Pursuit (OMP) for a single vector y.
def omp(y, D, sparsity):
    residual = y.copy()
    selected_atoms = []
    x = np.zeros(D.shape[1], dtype=np.float32)

    for _ in range(sparsity):
        correlations = D.T @ residual
        atom_index = np.argmax(np.abs(correlations))
        selected_atoms.append(atom_index)

        D_omega = D[:, selected_atoms]
        a = np.linalg.lstsq(D_omega, y, rcond=None)[0]

        residual = y - D_omega @ a
        if np.linalg.norm(residual) < 1e-6:
            break

    for i, idx in enumerate(selected_atoms):
        x[idx] = a[i]
    return x

# Extract patches from an image.
def patchify(img, patch_size, step=1):…

# Reconstruct an image from patches and their positions by averaging.
def unpatchify(patches, positions, image_shape):…

# Denoise an image using OMP and a DCT dictionary.
def denoise_image_omp(image, patch_size=8, sparsity=10):
    D = generate_dct_dictionary(patch_size)
    patches, positions = patchify(image, patch_size, step=1)
    denoised_patches = []
    for patch in patches:
        y = patch.flatten()
        x = omp(y, D, sparsity)
        patch_recon = (D @ x).reshape(patch_size, patch_size)
        denoised_patches.append(patch_recon)
    denoised_patches = np.array(denoised_patches)
    denoised_img = unpatchify(denoised_patches, positions, image.shape)
    return denoised_img

def ensure_folder(folder_path):…

# Save a single channel image in the corresponding color.
# channel_float: (H, W) float image in [0, 1].
# channel_idx: 0 for blue, 1 for green, 2 for red.
def save_color_channel_image(channel_float, img_shape, channel_idx, save_path):
```

A picture of my cat I took on my phone at a higher quality, then reduced to 200×200 pixels.*

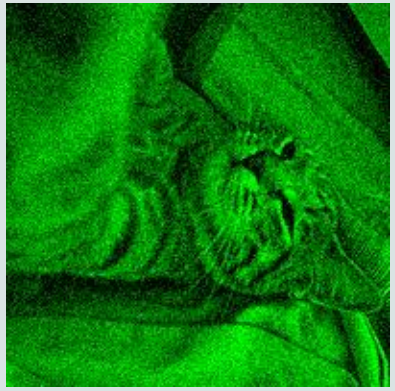
The image with 5% added noise from Photoshop's 'Add Noise' filter, using Gaussian distribution.

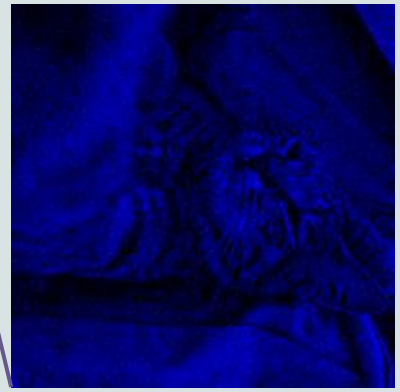
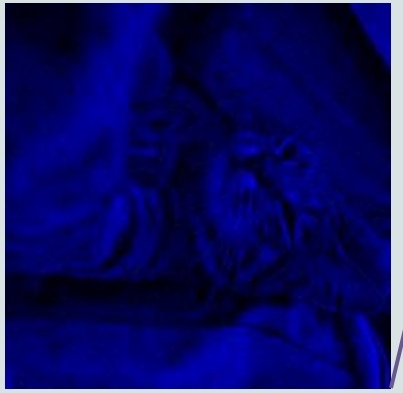
The image after passing through the denoising Python program.

*The size was reduced to save processing power on my laptop when running the algorithm. It took a few minutes to run the whole thing.








Noisy Image

First Pass

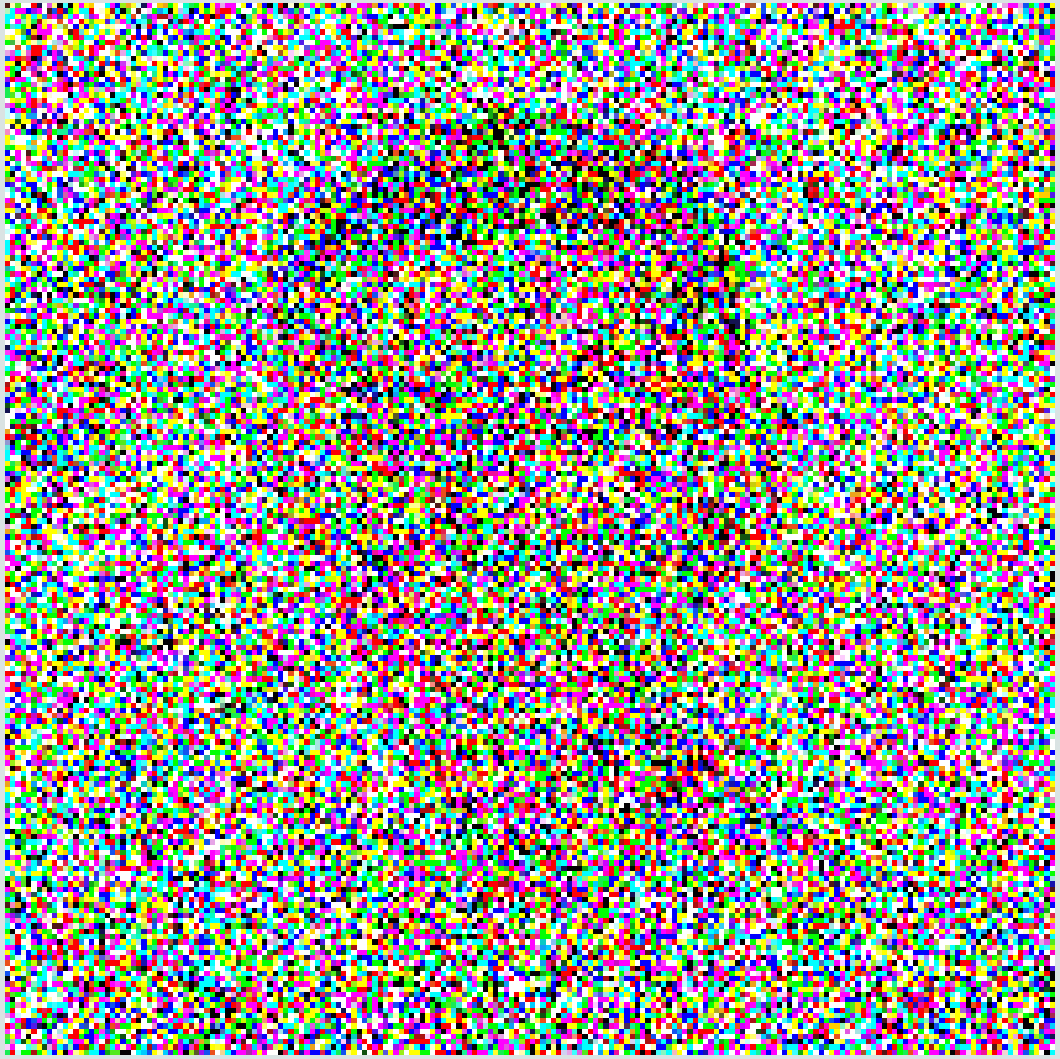
Second Pass

Third Pass

Second Pass

Original Image

Third Pass



200% Gaussian Noise, 10 Iterations

200% Uniform Noise, 5 Iterations

Reference

- Coding:1. https://github.com/stonemason11/Machine-Learning-Algorithms-in-Python/blob/master/OMP.py
- Coding: 2. https://youtu.be/jBY1s1rjRzs?si=MUQxkM5eEckNXk4i
- Graphs: https://youtu.be/sK6jTVo0Bhk?si=qouU6sF1PVnseWL0
- Theory : https://www.sciencedirect.com/topics/engineering/orthogonal-matching-pursuit
- Rubinstein, R., Zibulevsky, M., & Elad, M. (2010). Efficient implementation of the K-SVD algorithm using batch orthogonal matching pursuit. *IEEE Transactions on Signal Processing, 57*(12), 5636–5646. https://doi.org/10.1109/TSP.2009.2039178