Computer Assignment 3

Mean Squared Error

Problem 1

Two applications of autoencoders are:

1. **Image preprocessing**: When working with images, it is useful to preprocess on the dataset to fit them in the context that we are working on.

2. **Data compression**: Compression is a usefull operation specially when it comes to working with images. Compressed images are much smaller files than the original ones and they are easier to share and work on.

Problem 2

When using autoencoders, there may be hidden or unobserved variables and representations learned by the model, because the input data is mapped to a lower-dimensional space and is in a compressed form, when going back to higher-dimentional space, the model may construct and learn representations from this transition that are not actually present in the original data.
With smaller latent space dimensions, there is more chanec that such errors happend in the model.

Problem 3

Imports and constants

In [1]:
```
import numpy as np, random
import tensorflow as tf
import matplotlib.pyplot as plt
import matplotlib
from keras.datasets import mnist
from scipy import stats

green = '#57cc99'
red = '#e56b6f'
blue = '#22577a'
yellow = '#ffca3a'

bar_width = 0.5
line_thickness = 2

def set_seed(seed):
    np.random.seed(seed)
    random.seed(seed)
set_seed(810109203)
```

```
2024-01-16 03:34:00.188025: I tensorflow/core/platform/cpu_feature_guard.cc:182] This TensorFlow binary is optimized t
o use available CPU instructions in performance-critical operations.
To enable the following instructions: SSE3 SSE4.1 SSE4.2, in other operations, rebuild TensorFlow with the appropriat
e compiler flags.
```

Part A

In [2]:
```
(_, _) , (test_images, _) = mnist.load_data()
test_images = test_images.reshape(test_images.shape[0] , -1)
test_images = test_images.astype('float32') / 255.0
```

Part B

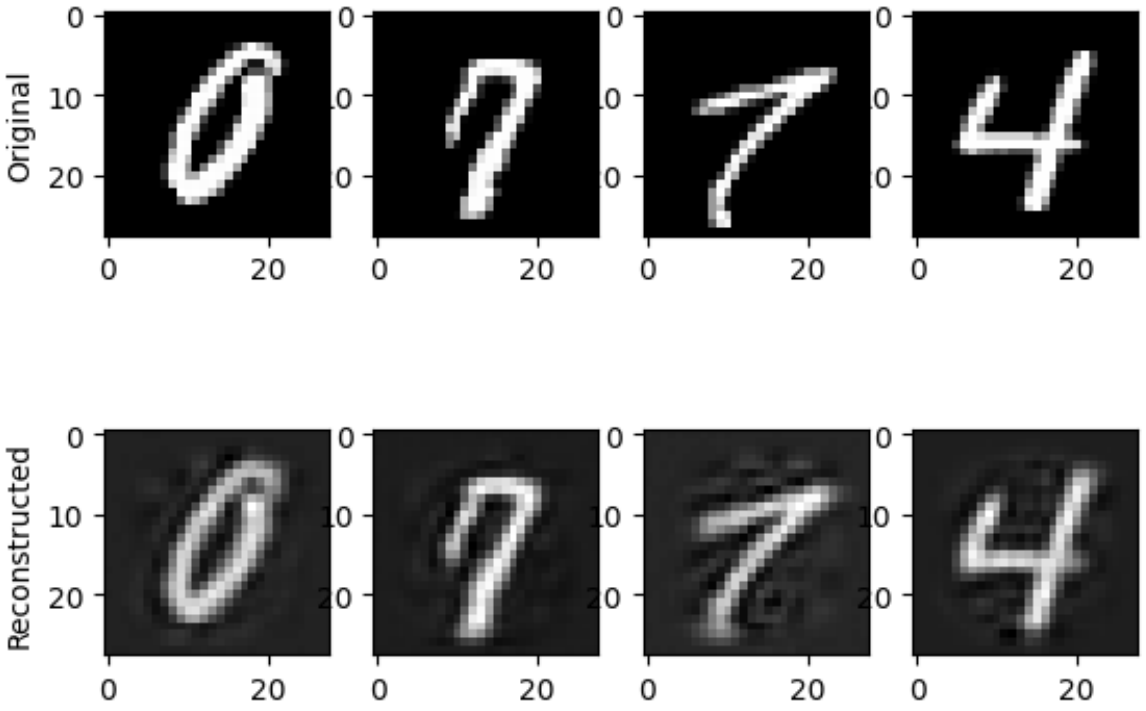
In [3]:
```
autoencoder = tf.keras.models.load_model('mnist_AE.h5')
reconstructed_images = autoencoder.predict(test_images)
```

```
2024-01-16 03:34:04.967290: I tensorflow/core/common_runtime/process_util.cc:146] Creating new thread pool with defaul
t inter op setting: 2. Tune using inter_op_parallelism_threads for best performance.
313/313 [==============================] - 6s 17ms/step
```

Part C

In [4]:
```
n = len(test_images)

choices_cnt = 4
rands = [np.random.randint(n) for _ in range(choices_cnt)]
```

```
cmap = matplotlib.colormaps.get_cmap('grey')
fig, axs = plt.subplots(2, choices_cnt)

for i in range(choices_cnt):
    rand_num= rands[i]
    axs[0, i].imshow(test_images.reshape(n, 28, 28)[rand_num], cmap=cmap)
    axs[1, i].imshow(reconstructed_images.reshape(n, 28, 28)[rand_num], cmap=cmap)

axs[0, 0].set_ylabel('Original')
axs[1, 0].set_ylabel('Reconstructed')
plt.show()
```



Part D

The MSE (Mean squared error) for two matrices A and B is:

$$MSE = \frac{1}{n}\sum_{i,j}(a_{ij} - b_{ij})^2$$

To calculate and plot the MSE for each pair of images, we'll use the following code:

In [5]:
```
mse = np.sum((test_images - reconstructed_images) ** 2, axis=1) / n

plt.hist(mse, density=True, alpha=0.7, label='MSE', color=red, edgecolor='black', bins=100)
plt.title('Mean Squared Error (MSE)')
plt.ylabel('Density')
plt.xlabel('MSE')
plt.grid(True)
plt.legend()
plt.show()
```

Part E

To calculate mean and variance of `mse`:

In [6]:
```
mu = np.mean(mse)
var = np.var(mse)
sigma = var ** 0.5
```

The calculated mean and variance for the `mse` are:

$\mu \simeq 4.61 \times 10^{-4}$

$\sigma^2 \simeq 3.34 \times 10^{-8}$

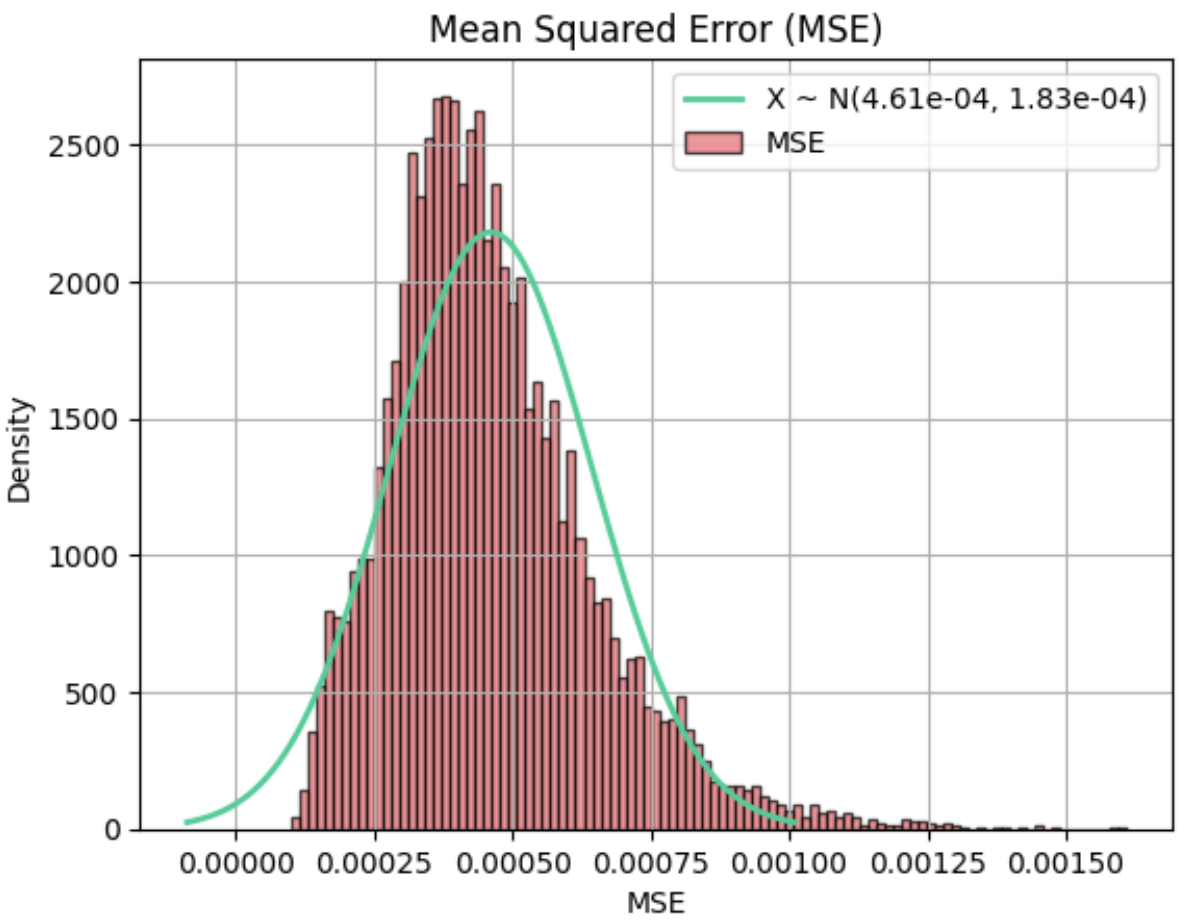
$\sigma = \sqrt{\sigma^2} \simeq 1.83 \times 10^{-4}$

Let X be a random variable from a normal distribution with calculated parameters:

$X \sim N(4.61 \times 10^{-4}, 1.83 \times 10^{-4})$

The probability mass function of X plotted with the `mse` density histogram together will be like this:

In [7]:
```
x = np.linspace(mu - 3*sigma, mu + 3*sigma, 100)
plt.plot(x, stats.norm.pdf(x, mu, sigma), label="X ~ N({:.2e}, {:.2e})".format(mu, sigma), lw=line_thickness, color=c

plt.hist(mse, density=True, alpha=0.7, label='MSE', color=red, edgecolor='black', bins=100)
plt.title('Mean Squared Error (MSE)')
plt.ylabel('Density')
plt.xlabel('MSE')
plt.grid(True)
plt.legend()
plt.show()
```



Which don't look so coinciding. Let's calculate the p-value using the Kolmogorov–Smirnov test:

In [8]:
```
ks_statistic, p_value = stats.kstest(mse, cdf='norm', args=(mu, sigma))
```

The calculated p-value is around 4.53×10^{-43}, which is an extremely small number and shows that our MSE doest not come from normal distribution.

Regression & Least Squares

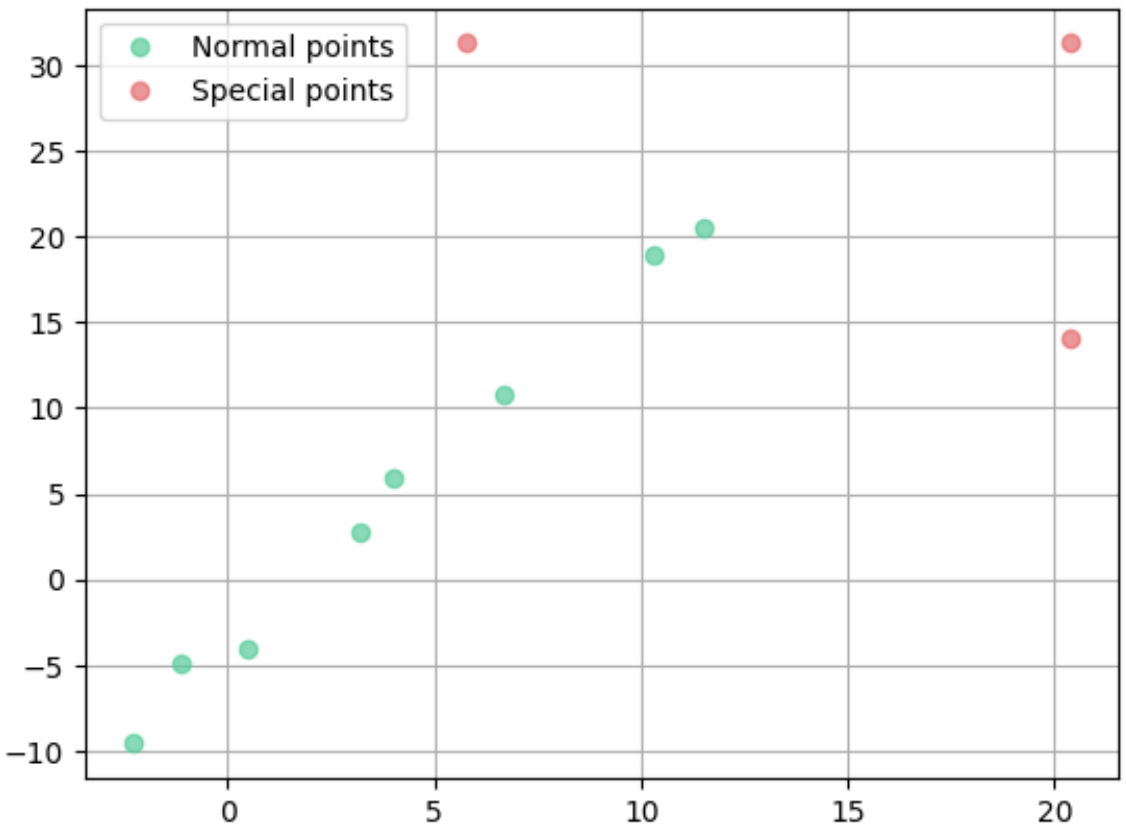
Problem 1

1. **Outlier points**: Outlier points are points that their y value does not follow the general range of the data and it is much lower or higher. It causes the resulting line to tilt to the outlier point and increases the SSR (Sum of squares residual).
2. **High leverage points**: High leverage points are like outlier points, except that they differ from the general trend of data in x value.
3. **Outlier and high leverage points**: These are points that are outlier and high leverage at the same time. So there is a chance that this point

accurs close to the regression line for normal points and doesn't effect the SSR.

Here are the input points:

In [9]:
```
points = [(-2.3, -9.6), (-1.1, -4.9), (0.5, -4.1), (3.2, 2.7), (4.0, 5.9), (6.7, 10.8), (10.3, 18.9), (11.5, 20.5)]
special_points = [(5.8, 31.3), (20.4, 14.1), (20.4, 31.3)]
outlier, high_leverage, both = special_points

x_reg = [point[0] for point in points]
y_reg = [point[1] for point in points]

x_spe = [point[0] for point in special_points]
y_spe = [point[1] for point in special_points]

plt.plot(x_reg, y_reg, 'o', color=green, label='Normal points', alpha=0.7)
plt.plot(x_spe, y_spe, 'o', color=red, label='Special points', alpha=0.7)
plt.legend()
plt.grid(True)
plt.show()
```



Problem 2

R^2 or **Coefficient of Determination** is a measure to assess how well a model explains and predicts future outcomes, in this case, how well the line fits the points. R^2 is calculated using the following formula:

$$R^2 = 1 - \frac{SSR}{SST}$$

Let the line equation be $y = mx + c$, SSR (sum of squares residual) and SST (sum of squares total) are:

$$SST = \sum_{i=1}^{n}(y_i - \bar{y})^2$$

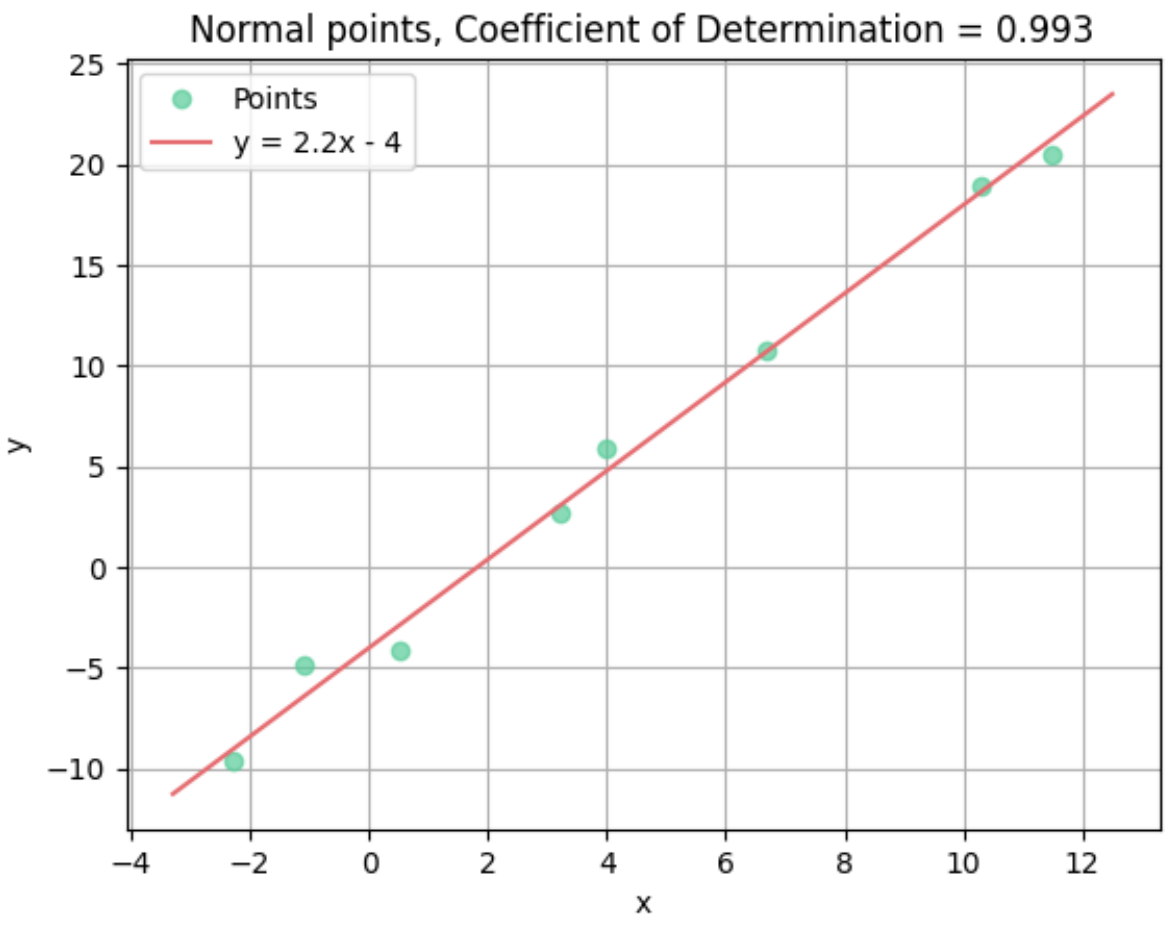
$$SSR = \sum_{i=1}^{n}(y_i - \text{y-pred})^2$$

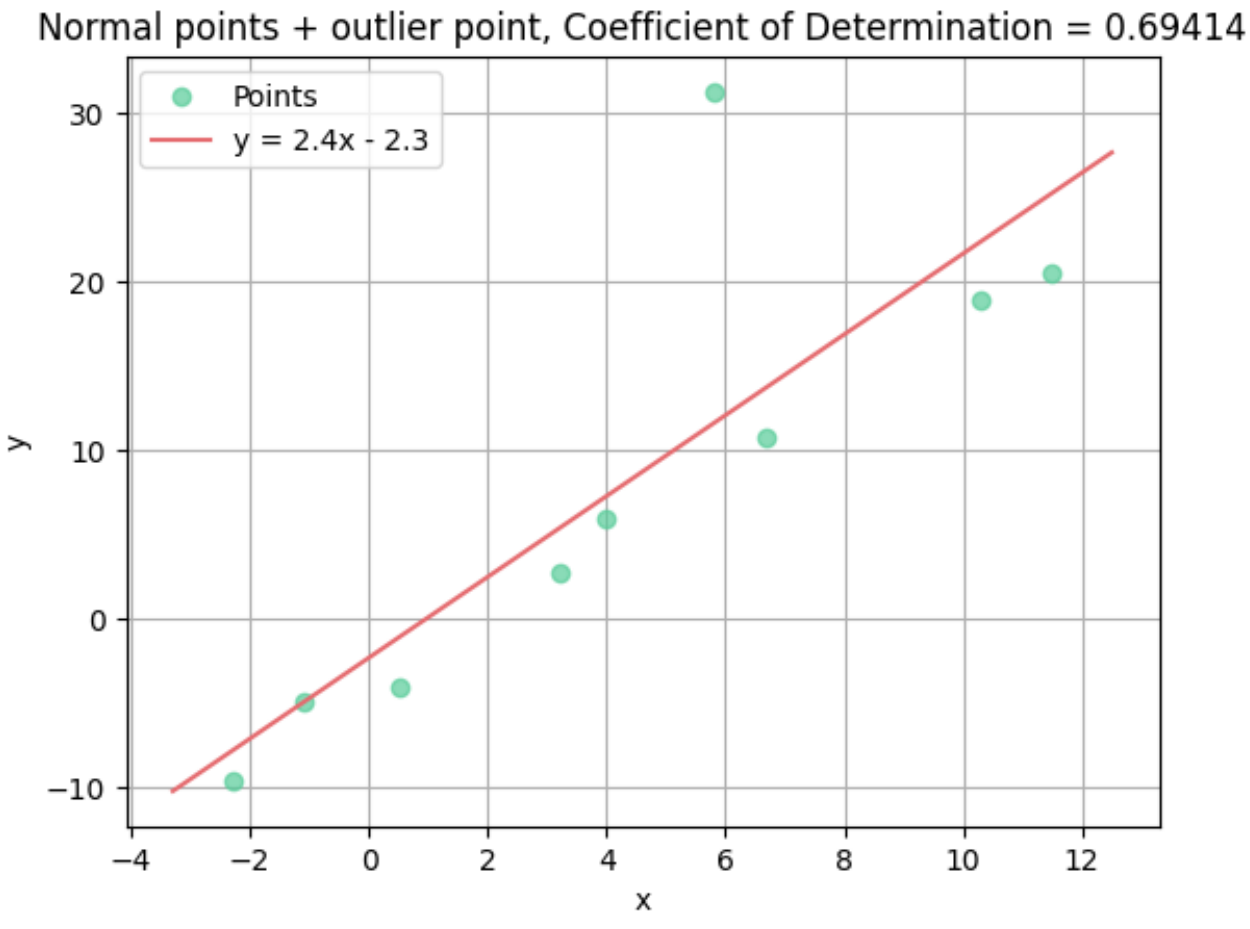
Where $\text{y-pred} = mx + c$.

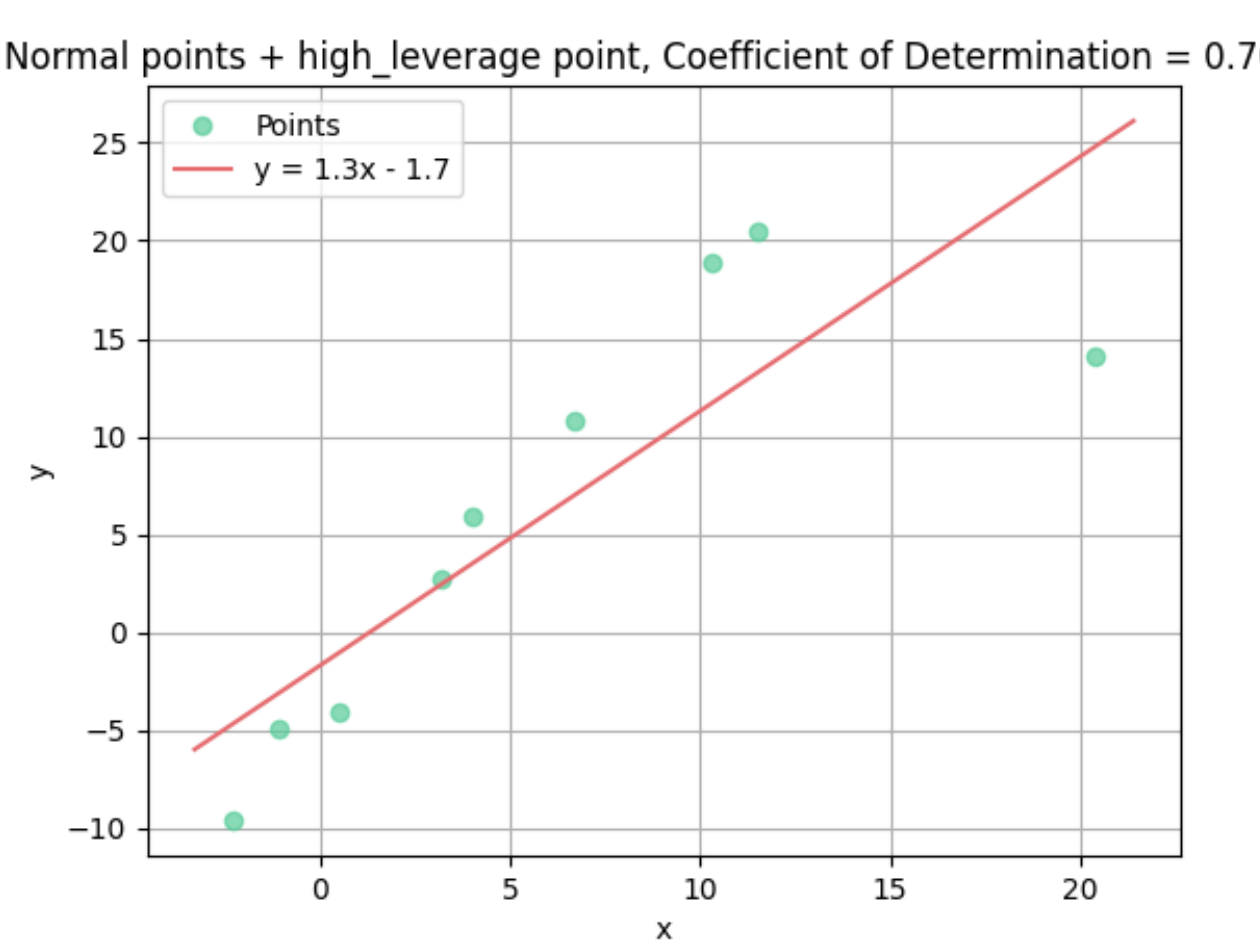
Porblem 3

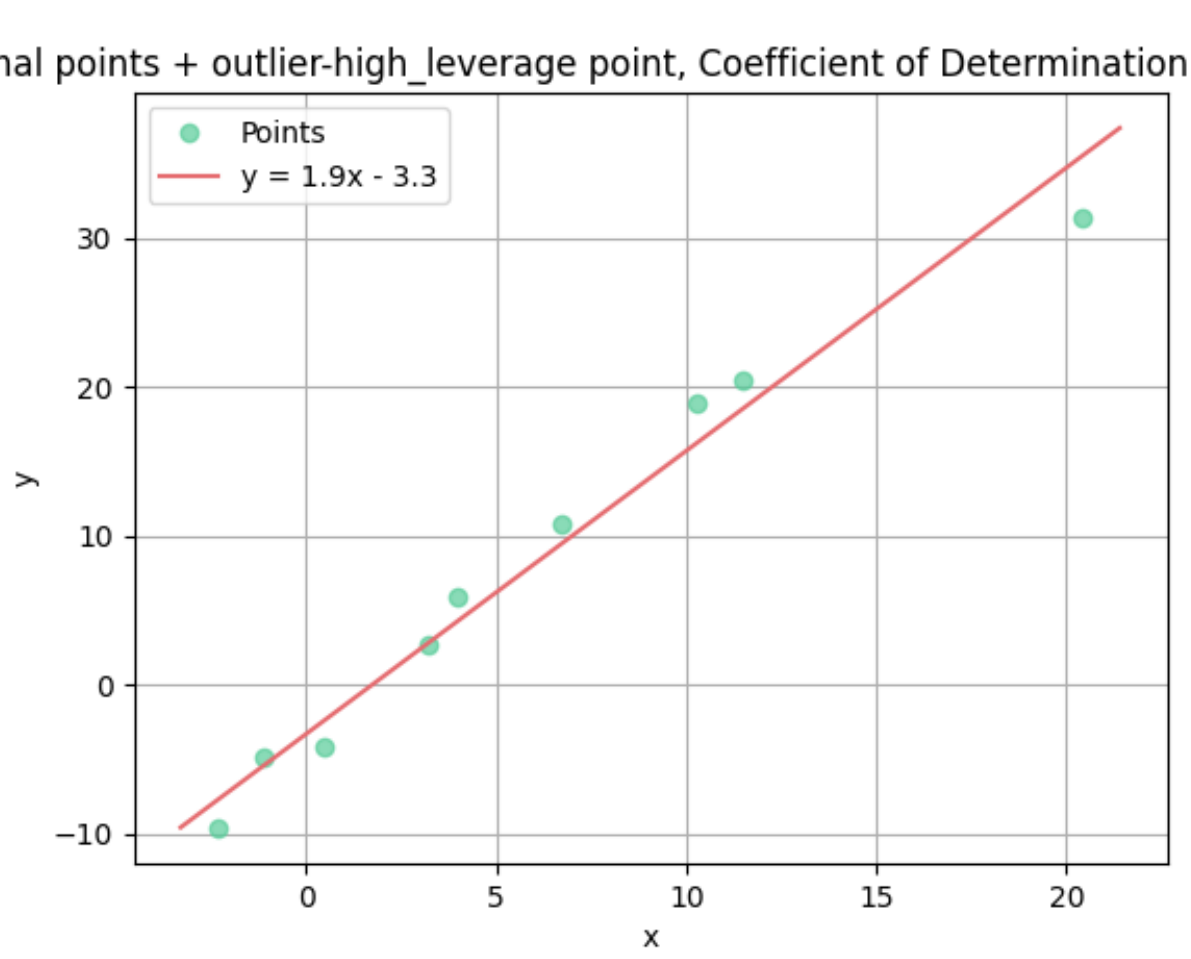
In [10]:
```
def p2l_y_dist(point_x, point_y, m, c):
    y_pred = m * point_x + c
    return abs(point_y - y_pred)

def calc_err(inp_points, m, c):
    squares_error = 0
    for point in inp_points:
        squares_error += p2l_y_dist(point[0], point[1], m, c) ** 2
    return squares_error

def linear_regression(inp_points, l = -12, r = 12, steps = 0.10):
    ans = (0, 0, int(1e9))
    for m in range(int(l / steps), int(r / steps)):
        m *= steps
        for c in range(int(l / steps), int(r / steps)):
            c *= steps
            error = calc_err(inp_points, m, c)
            if error < ans[2]:
```

```
                ans = (m, c, error)
    return ans

points_sets = [
    points,
    points + [outlier],
    points + [high_leverage],
    points + [both]
]

titles = [
    'Normal points',
    'Normal points + outlier point',
    'Normal points + high_leverage point',
    'Normal points + outlier-high_leverage point',
]

for i in range(len(points_sets)):
    points_set = points_sets[i]
    x_pts = [point[0] for point in points_set]
    y_pts = [point[1] for point in points_set]

    x = np.linspace(np.min(x_pts) - 1, np.max(x_pts) + 1, 100)

    sst = np.sum((y_pts - np.mean(y_pts)) ** 2)
    m, c, ssr = linear_regression(points_set)
    y = m*x + c

    r_squared = 1 - ssr / sst

    line_eq_format = "y = {:.2g}x - {:.2g}".format(m, -c) if c < 0 else "y = {:.2g}x + {:.2g}".format(m, c)

    plt.figure()
    plt.plot(x_pts, y_pts, 'o', color=green, alpha=0.7, label='Points')
    plt.plot(x, y, color=red, label=line_eq_format)
    plt.title(titles[i] + ", Coefficient of Determination = {:.5g}".format(r_squared))
    plt.xlabel('x')
    plt.ylabel('y')
    plt.grid(True)
    plt.legend()
    plt.show()
```









As you can see, the ressult R^2 for set of normal points and outlier/high leverage point is lower than the R^2 for normal points.

Porblem 4

We can use a regression model that gives weights to points based on how close they are to the general points of the data. The closer the point to other points, the higher its weight. Using this technique, outlier and high leverage points are assigned lower weights and as a result, their effect on the regression line is reduced.

Central Limit Theorem & Sampling

Problem 1

To replace the N/A values in our dataframe, we can use that column's mean. As a result, the mean of our data doesn't change and no outlier data is created.

In [11]:
```
import pandas as pd

df = pd.read_csv('FIFA2020.csv', encoding="ISO-8859-1")

df['dribbling'].fillna(df['dribbling'].mean(), inplace=True)
df['pace'].fillna(df['pace'].mean(), inplace=True)
```

Problem 2

To calculate minimum, maximum, Q1, Q3, median and mean, we can use the following function:

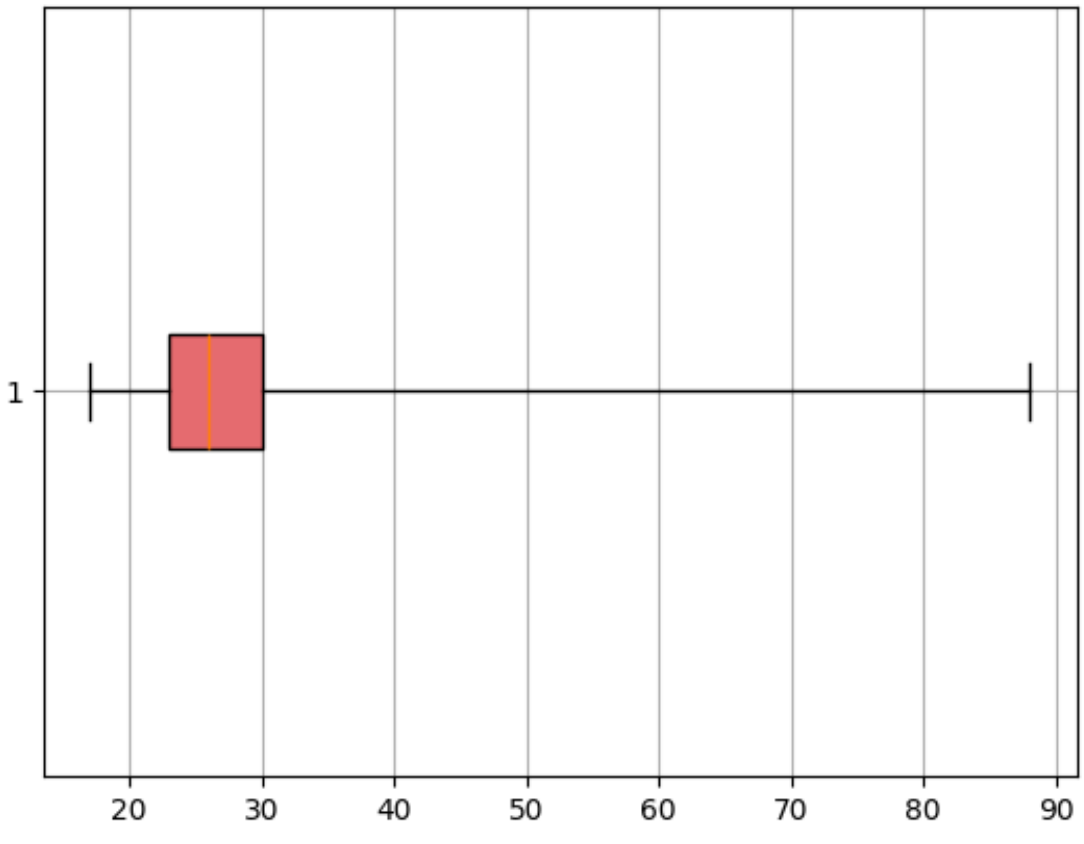
In [12]:
```
age = df['age']

q0 = np.min(age)
q1 = np.percentile(age, 25)
median = np.median(age)
mean = np.mean(age)
q3 = np.percentile(age, 75)
q4 = np.max(age)
```

The output is as follows:

Parameter	Value
Minimum	17
Q1	23
Median	26
Mean	26.68
Q3	30
Maximum	88

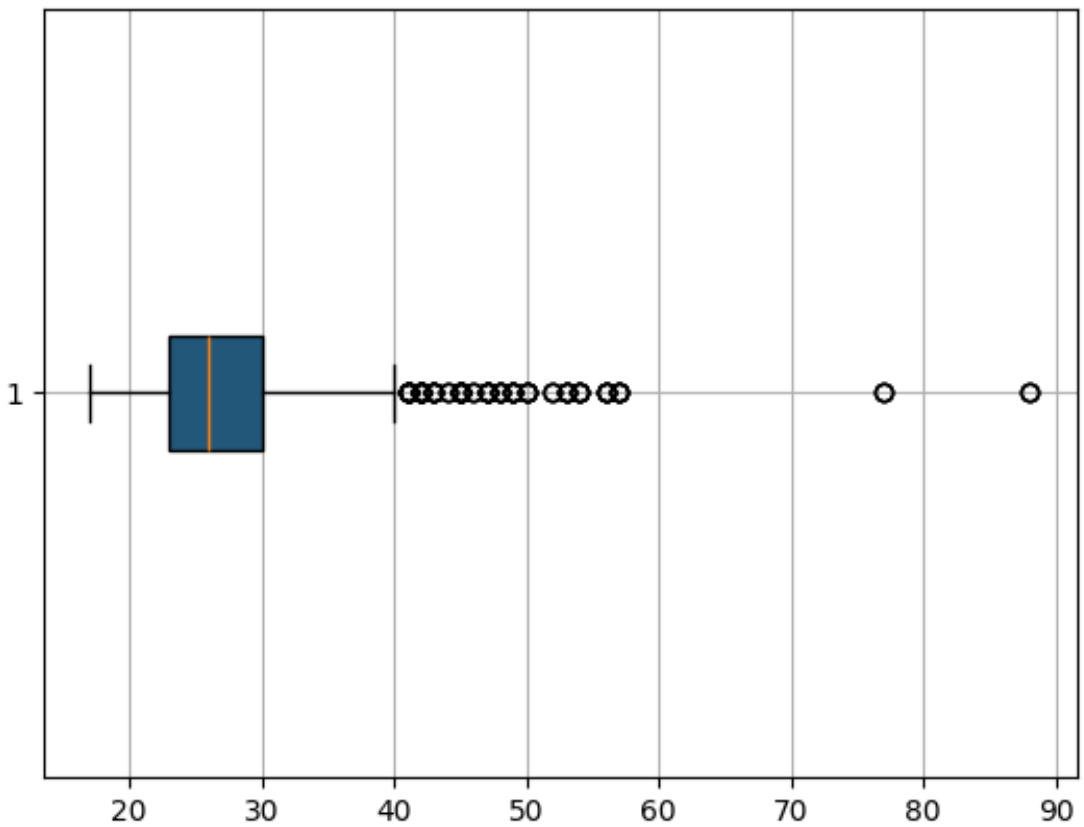
And the boxplot:

In [13]:
```
bplot = plt.boxplot(age, vert=False, showfliers=True, whis=[0, 100], patch_artist=True)
bplot['boxes'][0].set_facecolor(red)

plt.grid(True)
plt.show()
```



As you can see, the boxplot that is inlcuding outlier values is not much symmetrical, let's draw another one that ignores outliers:

In [14]:
```
bplot = plt.boxplot(age, vert=False, patch_artist=True)
bplot['boxes'][0].set_facecolor(blue)

plt.grid(True)
plt.show()
```



In this boxplot, we have ignored outlier data and we can see as a result, the box is more symmetrical than then previous one.

Problem 3

Part A

To calculate asked parameters, we can use the following code:

In [15]:
```
n = 100
weight_samples = np.random.choice(df['weight'], n, replace=False)

mean = np.mean(weight_samples)
var = np.var(weight_samples)
sigma = var ** 0.5
```

The results are:

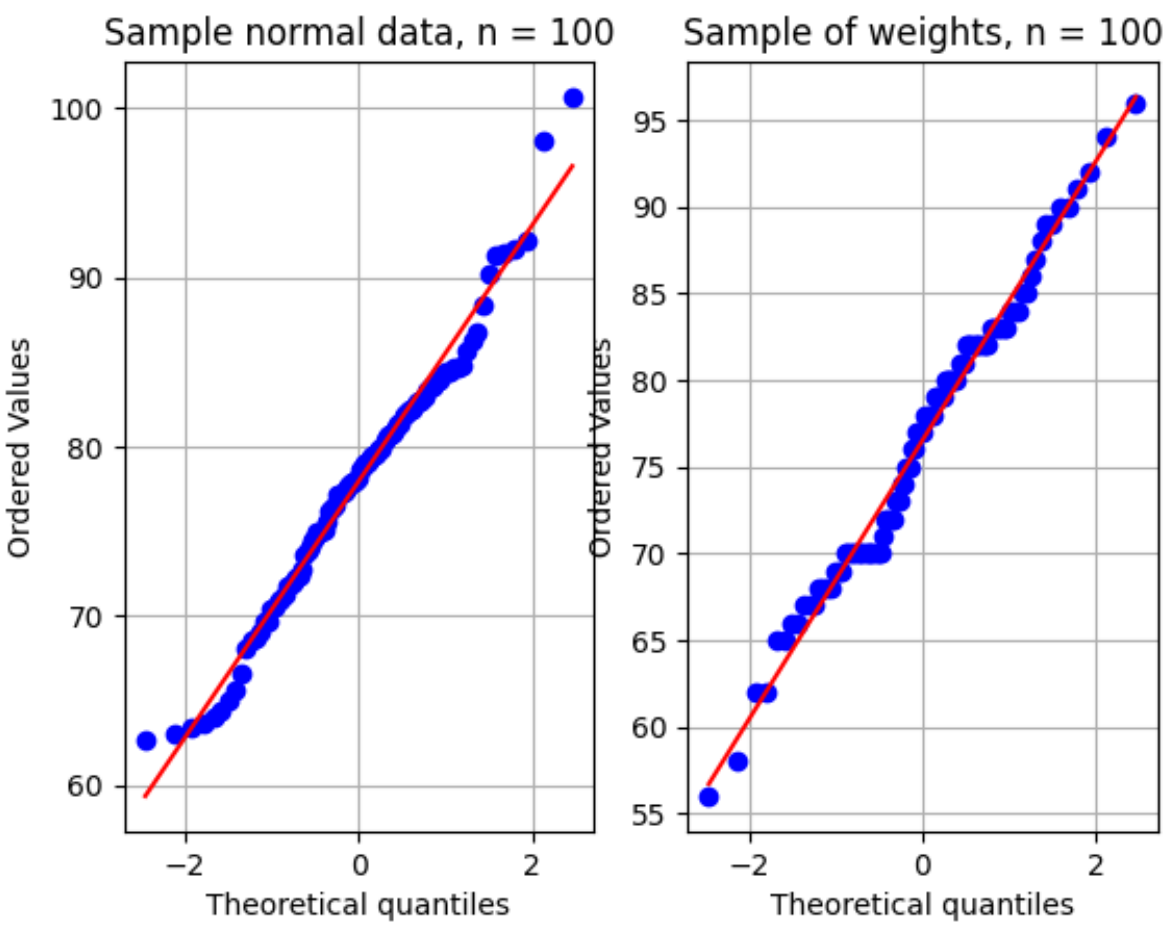
Parameter	Value
μ	76.39
σ^2	55.17
σ	7.42

Part B

A Q-Q plot, or quantile-quantile plot is the technique to see how well a sample data matches another data (or distribution) and it is often used to see if a data came from a theoretical distribution, such as a normal distribution.

Part C

In [17]:
```
def compare(n):
    weight_samples = np.random.choice(df['weight'], n, replace=False)

    mean = np.mean(weight_samples)
    var = np.var(weight_samples)
    sigma = var ** 0.5

    sample_normal_data = np.random.normal(mean, sigma, n)

    fig, (ax1, ax2) = plt.subplots(1, 2)

    stats.probplot(sample_normal_data, dist="norm", plot=ax1)
    stats.probplot(weight_samples, dist="norm", plot=ax2)

    ax1.set_title("Sample normal data, n = {}".format(n))
    ax2.set_title("Sample of weights, n = {}".format(n))
    ax1.grid(True)
    ax2.grid(True)
    plt.show()

    statistic, p_value = stats.shapiro(sample_normal_data)
    statistic, p_value = stats.shapiro(weight_samples)

compare(100)
```



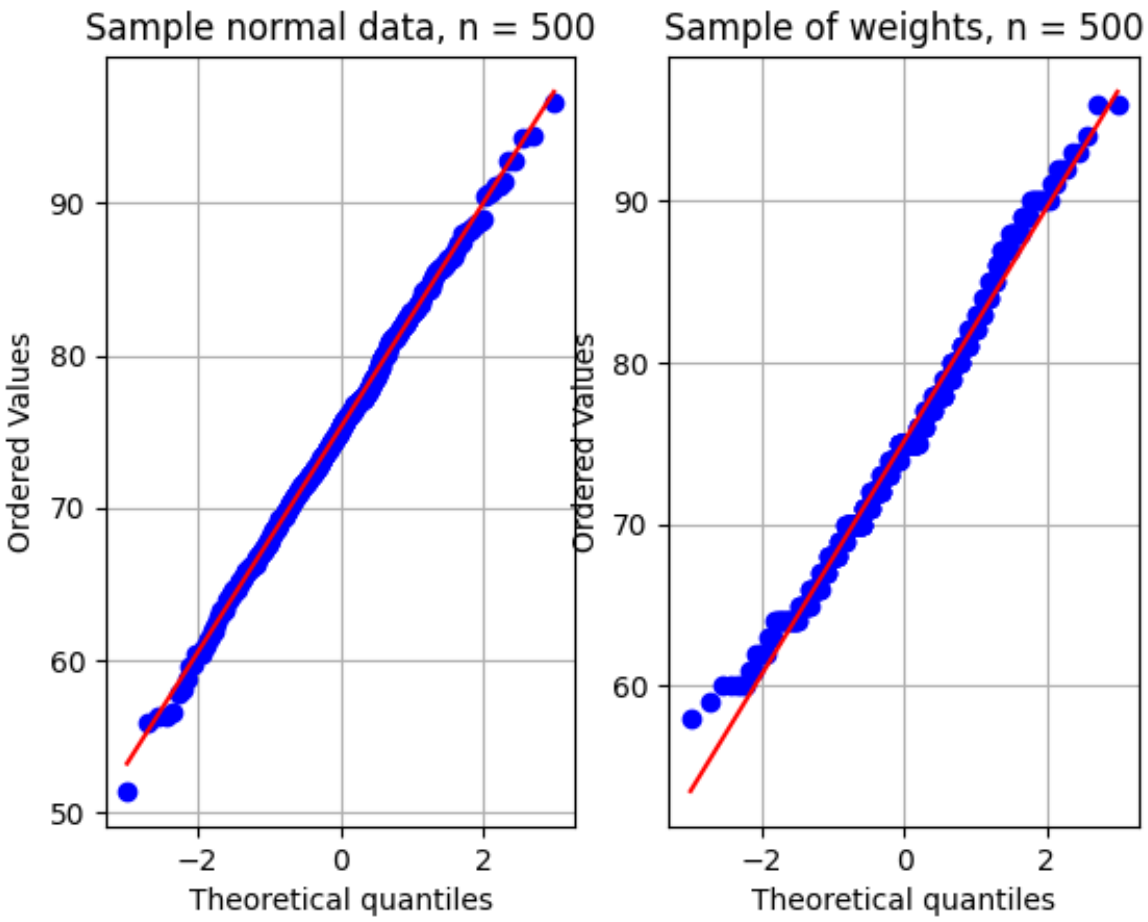
Part D

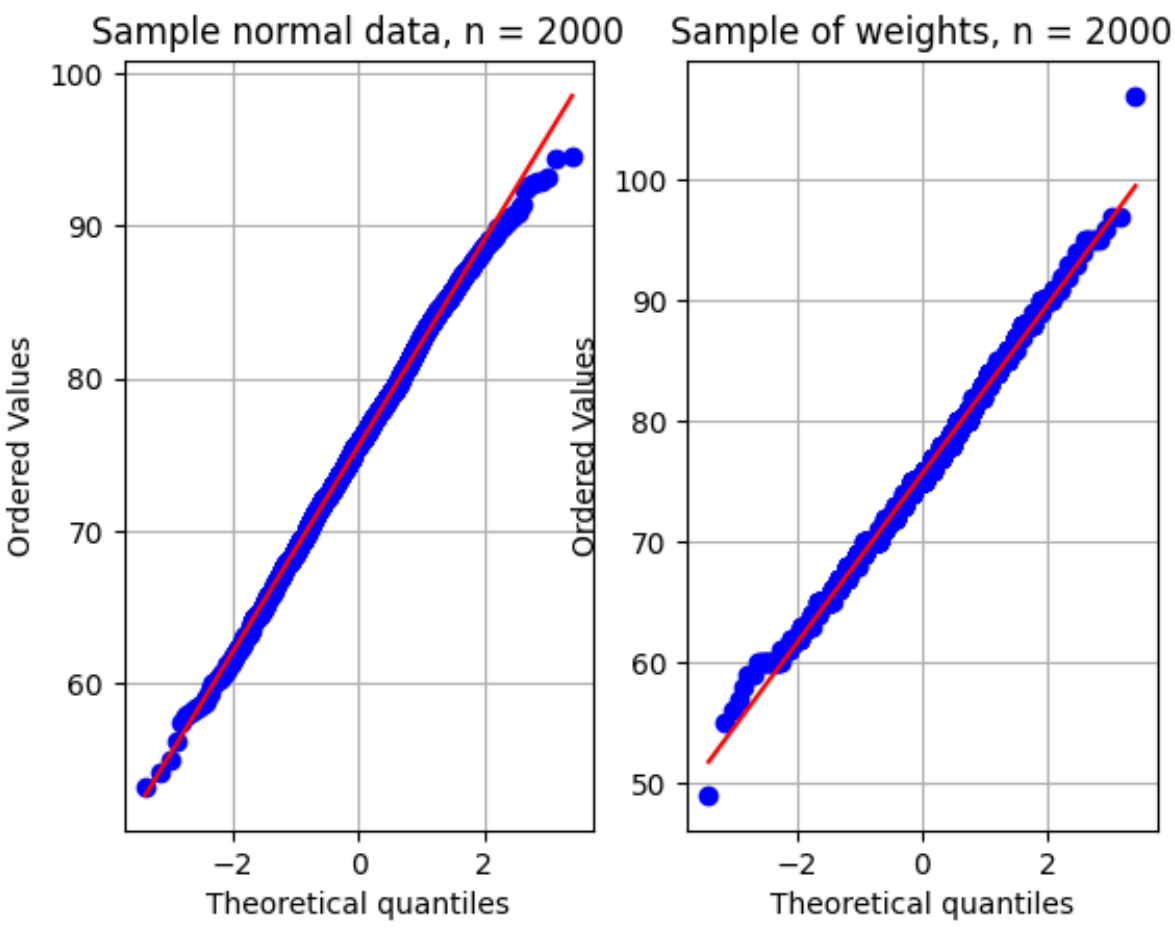
The p-value for generated normal sample is around 0.87, which is a high value and shows that this data is most likely to come from a normal distribution.

The p-value for weight samples is around 0.55 which again is not perfect, but shows that the weight sample when $n = 100$ seems to come from a normal distribution.

Part E

In [19]:
```
compare(500)
compare(2000)
```





These are the results for n = 500 and 2000:

n	**Normal sampling** p-value	**Weight sampling** p-value
100	0.87	0.55
500	0.35	0.04
2000	0.76	5.86×10^{-8}

As you can see as we use bigger n, our normal sampling gets closer and closer to normal distribution. But the weight samples get futher from normal distribtuin.

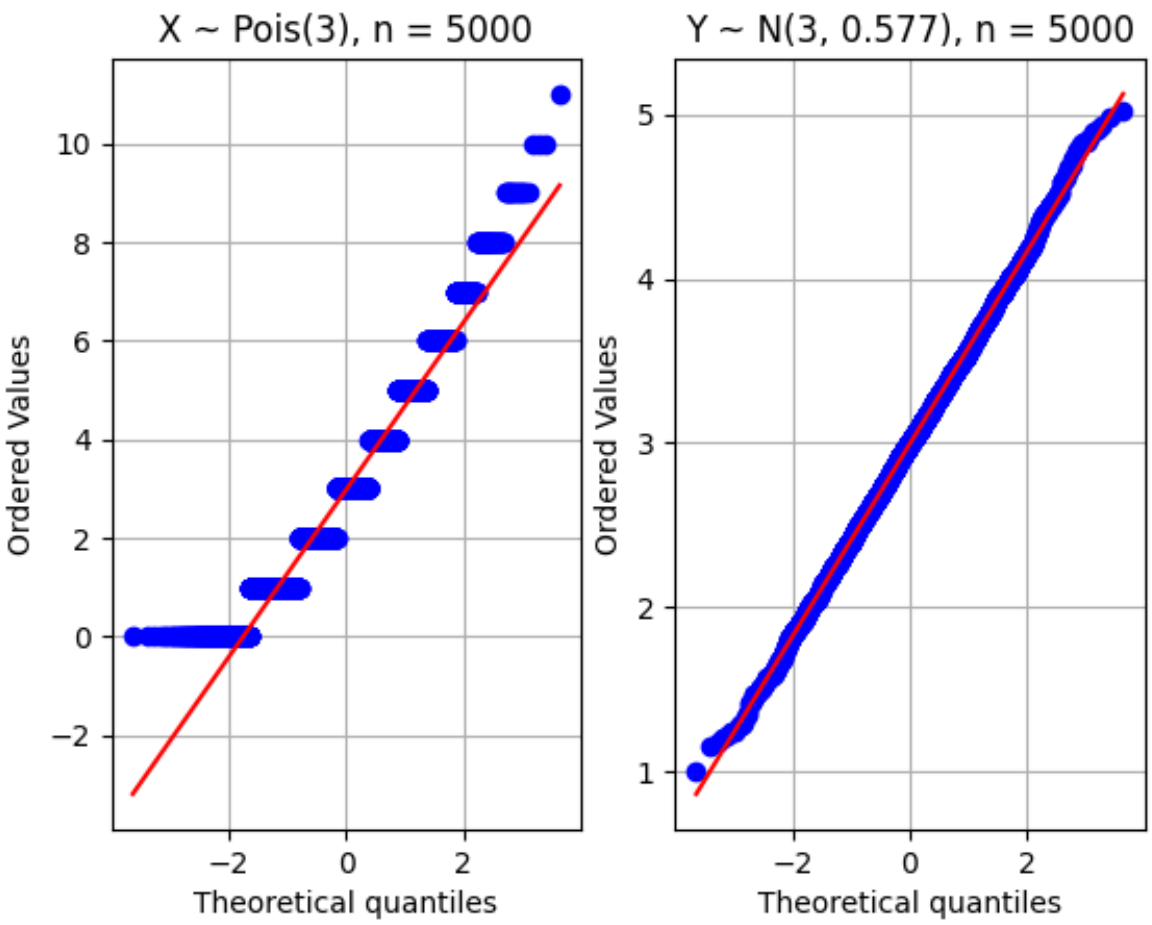
Porblem 4

Part A

This is the histogram for $\lambda = 3$ and $n = 5000$:

In [20]:
```
from scipy.stats import poisson

def poisson_comp(lam, n):
    pois_data = poisson.rvs(lam, size=n)
    norm_data = np.random.normal(lam, lam ** -0.5, n)

    fig, (ax1, ax2) = plt.subplots(1, 2)
```

```
    stats.probplot(pois_data, dist="norm", plot=ax1)
    stats.probplot(norm_data, dist="norm", plot=ax2)

    ax1.set_title('X ~ Pois({}), n = {}'.format(lam, n))
    ax2.set_title('Y ~ N({}, {:.3g}), n = {}'.format(lam, lam ** -0.5, n))
    ax1.grid(True)
    ax2.grid(True)
    plt.show()

    statistic, p_value = stats.shapiro(norm_data)
    statistic, p_value = stats.shapiro(pois_data)

poisson_comp(3, 5000)
```



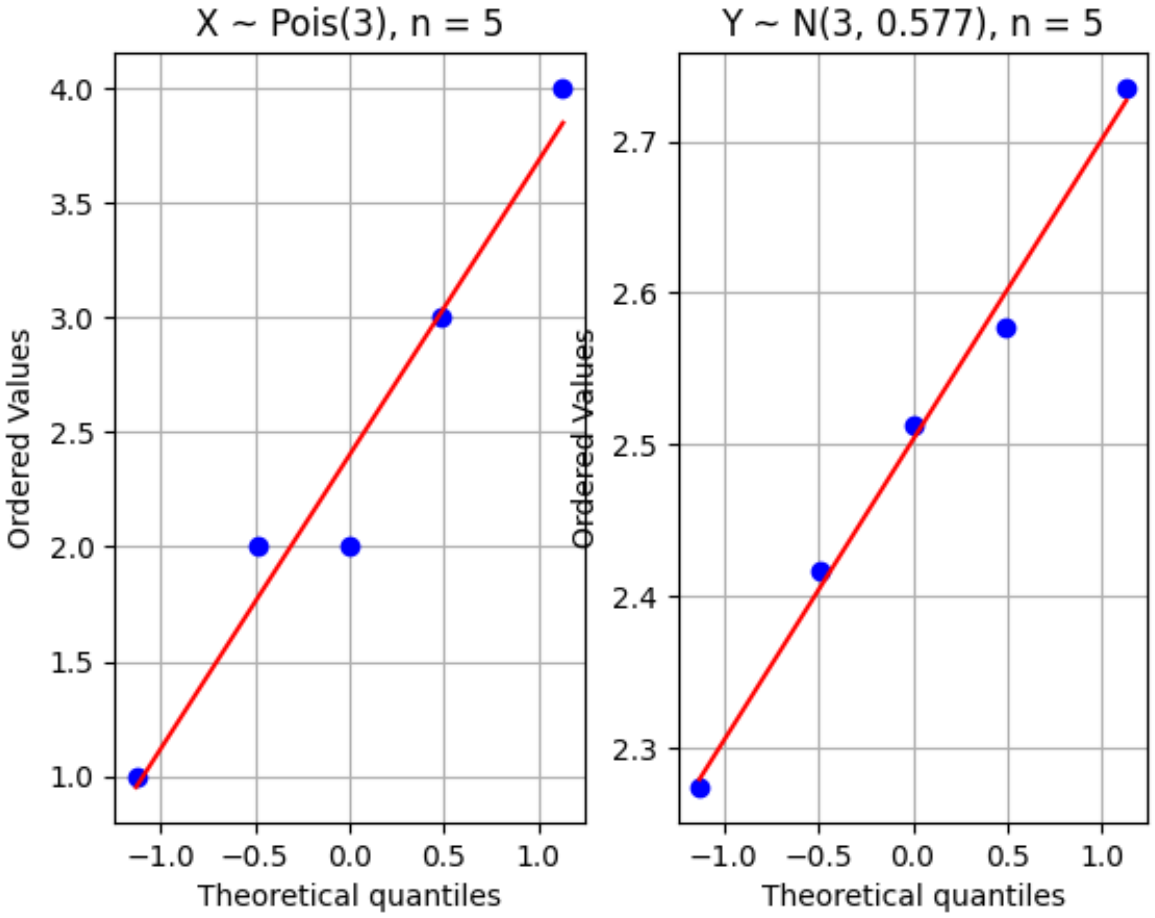
Part B

These are the results for $n = 5,\ 50,\ 500$:

n	**Normal sampling** p-value	**Poisson sampling** p-value
5	0.32	0.23
50	0.15	0.01
500	0.72	5.55×10^{-11}
5000	0.79	1.73×10^{-38}

As it can be observed, the p-value increases as we increase n for normal sampling, but the more n gets, the further poisson sampling gets from normal destribution.

In [21]:
```
poisson_comp(3, 5)
poisson_comp(3, 50)
poisson_comp(3, 500)
```


X ~ Pois(3), n = 5
Y ~ N(3, 0.577), n = 5
Ordered Values
Theoretical quantiles
X ~ Pois(3), n = 50
Y ~ N(3, 0.577), n = 50
X ~ Pois(3), n = 500
Y ~ N(3, 0.577), n = 500

In [ ]: