1. **Optimization and Machine Learning**

   In data science and computer science, we optimize a lot of stuff. For example, in linear regression we optimize for the intercept and coefficients of our model by updating the weights w so that the straight line formed by this linear regression can separate the data in a more proper way. In logistic regression, we optimize the error function to maximize the likelihood of the probability of all the history events happening. In neural networks we optimize the weights in our network (more on that in later classes!), etc.

   In one sentence, "optimization" simply refers to minimizing/maximizing a function. For example, what value of x minimizes the function $f(x) = (x - 2)^2 + 5$? What is the minimum value? Answers $x = 2, and\ f(x) = 5$:

   Based on the first month classes, you can start to think of machine learning as a three-step process:

   - **Choose your model**: controls the space of possible functions that map X to y (e.g., a linear model can learn linear functions)
   - **Choose your loss function (or Error)**: tells us how to compare these various functions (e.g., $y = 3x_2 + 2x_1 + 5$ is a better model than $y = -x_2 + 10x_1 + 1$) ?
   - **Choose your optimization algorithm (e.g. pocket algorithm for classification, gradient descent for logistic regression)**: finds the minimum of the loss function (e.g., what is the optimum value $w_0, and\ w_1$ of in $y = w_o + w_1x_1$)

2. **Loss Functions, we use error in the slides**

   Loss functions (also often called "objective functions", “error functions”, or "cost functions", although some debate that these are slightly different things) are what we use to map the performance of a model to a real number and it's the thing we want to optimize! For example, here's the mean squared error (MSE), which is a common loss function has been discussed in the class:

   $$f(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y_i} - \hat{y_i})^2$$

   Where y_i is the actual response and $\hat{y_i}$ is the predicted response

   Consider a simple linear regression model $\hat{y_i} = w_0 + w_1x_i$, then the loss function is:

   $$f(w) = \frac{1}{n}\sum_{i=1}^{n}((w_0 + wx_i) - y_i)^2$$

   With given data samples $\{(x_1, y_1), (x_2, y_2) \dots (x_n, y_n)$ The optimization problem here is to find the values of $w_0, and\ w_1$ that minimizes the MSE.

Task1 (30pts): Based on the first month class content and above information, answer the below questions:

a. **Explain the difference between a model, loss function, and optimization algorithm in the context of machine learning (15 pts).**
b. **Please explain the difference between linear classification, linear regression, and logistic regression (6 pts).**
c. **Please explain the difference between supervised learning, unsupervised learning, and reinforcement learning. Give necessary examples to explain your answer (9pts).**

3. Code implementation

Import below libraries

```
import numpy as np
import pandas as pd
from scipy.optimize import minimize
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression, LogisticRegression
from sklearn.metrics import mean_squared_error, mean_absolute_error
from utils import plotting as libs
```

a. **Optimizing Linear Regression**

We'll use a dataset of Pokemon "attack" and "defense" stats to do this

```
# read data samples from the csv file, and get the first 10 data samples
df = (pd.read_csv("data/pokemon.csv", usecols=['name', 'defense', 'attack'], index_col=0)
      .reset_index()
    )
```

Task2: Explore the pandas dataframe by yourself, and print out how many data samples the Pokemon dataset has? What is the range of “defense”and “attack”?. Screenshot your output (5pts)

I'm going to start with just 10 data samples from the pokemon dataset to optimize the linear regression.

```
x = df_10['defense']
y = df_10['attack']
#plot the 10 data samples in the Scatter plotb by calling the plot_pokemon function in the plotting library
fig = libs.plot_pokemon(x, y)
fig.write_image('figs/x_y.png')
```

Your output will look like:

	name	attack	defense
0	Bulbasaur	49	49
1	Ivysaur	62	63
2	Venusaur	100	123
3	Charmander	52	43
4	Charmeleon	64	58
5	Charizard	104	78
6	Squirtle	48	65
7	Wartortle	63	80
8	Blastoise	103	120
9	Caterpie	30	35

Plot the data samples of defense and attack,

```
#plot the 10 data samples in the Scatter plotb by calling the plot_pokemon function in the plotting library
fig = libs.plot_pokemon(x, y)
fig.write_image('figs/x_y.png')
```

Recall simple linear regression: $\widehat{y_i} = w_0 + w_1 x_i$ (where w_0 is the intercept and w_1 is the slope coefficient). If we assume ($w_0, w_1 = (10, 0.5)$ then we would have:

Task 3: Please plot the fitted linear regression line with weight $w_0, w_1 = (10, 0.5)$ by using the plot_pokemon. Screenshot your output (10pts)

```
y_hat = 10 + 0.5 * x
#plot linear regression line with weights w = (w0, w1) = (10, 0.5)
#your code ???
```

From the fitted linear regression line, we can see that the fit is not very good... We need to optimize it! A **loss function** can help quantify the fit of our model and we want to find the parameters of our model that minimize the loss function. We'll use mean-squared-error (MSE) as our loss function:

$$f(w) = \frac{1}{n}\sum_{i=1}^{n}((w_0 + w_1 x_i) - y_i)^2$$

Where n is the number of data samples we have (10 in our case). We'll use the sklearn function `mean_squared_error()` which I imported at the top of the notebook to calculate MSE for us:

```
#the error/loss of the current fitted linear regression line
mse = libs.mean_squared_error(y, y_hat)
print(mse)
```

And the MSE output is : 680.75

So this is the MSE across all training examples. For now, let's assume the intercept is 0 ($w_0 = 0$) and **just focus on optimizing the slope** (w_1). One thing we could do is try many different values for the slope and find the one that minimizes the MSE:

```
# try many different values for the slope and find the one that minimizes the MSE
slopes = np.arange(0.4, 1.65, 0.05)
mse = pd.DataFrame({"slope": slopes,
                    "MSE": [mean_squared_error(y, m * x) for m in slopes]})
print(mse)
```

The output is:

```
    slope        MSE
0    0.40  1770.0760
1    0.45  1478.6515
2    0.50  1216.7500
3    0.55   984.3715
4    0.60   781.5160
5    0.65   608.1835
6    0.70   464.3740
7    0.75   350.0875
8    0.80   265.3240
9    0.85   210.0835
10   0.90   184.3660
11   0.95   188.1715
12   1.00   221.5000
13   1.05   284.3515
14   1.10   376.7260
15   1.15   498.6235
16   1.20   650.0440
17   1.25   830.9875
18   1.30  1041.4540
19   1.35  1281.4435
20   1.40  1550.9560
21   1.45  1849.9915
22   1.50  2178.5500
23   1.55  2536.6315
24   1.60  2924.2360
```

Task4: What is the optimal linear regression line function based on the MSE and slope values given by the above figure? Please plot the optimal linear regression line by using the plot_pokemon. Screenshot your output (10pts)

Grid search is the method that is usually used to try the number of values of a parameter. Using the below code, you can generate a dynamic figure which displays the linear line change with the change of slopes value.

```
fig = libs.plot_grid_search(x, y, slopes, mean_squared_error)
fig.show()
fig.write_html("mse_gridsearch.html")
```

Task5: Explain what the mse.gridsearch figure shows, such as how mse and the line will change when slope changes? (10pts)

b. Gradient Descent With One Parameter

Gradient descent is an optimization algorithm that can help us optimize our loss function more efficiently than the "manual" approach we tried above. As the name suggests, we are going to leverage the gradient of our loss function to help us optimize our model parameters w. The gradient is just a vector of (partial) derivatives of the loss function with respect to the model parameters. Sounds complicated but it's not at all.

In plain English, the gradient will tell us two things:

- Which direction to move our parameter in to decrease loss (i.e., should we increase or decrease its value?)
- How far to move it (i.e., should we adjust it by 0.1 or 2 or 50 etc.?)

Let's forget about the intercept now and just work with this simple linear regression model: $\widehat{y_i} = wx_i$. For this model, the loss function has the form:

$$f(w) = \frac{1}{n}\sum_{i=1}^{n}((wx_i) - y_i)^2$$

The gradient of this function with respect to the parameter w is:

$$\frac{d}{dw}f(w) = \frac{1}{n}\sum_{i=1}^{n} 2x_i(w_i x_i - y_i)$$

Let's code that up and calculate the gradient of our loss function at a slope of $w = 0.5$, what is the output of the gradient?

```
def gradient(x, y, w):
    return 2 * (x * (w * x - y)).mean()

print(gradient(x, y, w=0.5))
```

So this is the gradient across the 10 training samples and tells us how to adjust to reduce the MSE loss over the 10 training samples! Recall from calculus that **the gradient actually points in the direction of steepest ascent**. We want to move in the direction of steepest descent (the negative of the gradient) to reduce our loss. For example, the above gradient is negative, but we obviously need to **increase** the value of our slope (w) to reduce our loss as you can see here:

```
fig = libs.plot_gradient_m(x, y, 0.5, mse["slope"], mse["MSE"], gradient)
fig.show()
```

The amount we adjust our slope each iteration is controlled by a "**learning rate**", denoted α (note the minus in the equation below which accounts for flipping the sign of the gradient as discussed above):

$$w_{new} = w - \alpha \times gradient$$

$$w_{new} = w - \alpha \frac{d}{dw} f(w)$$

α is a hyperparameter that can be optimized, typical values range from 0.001 to 0.9. With the math out of the way, we're now ready to use gradient descent to optimize our slope. Gradient descent is an iterative algorithm, meaning that we keep repeating it until we meet some stopping criteria. Typically we stop gradient descent when:

- the step size is smaller than some pre-defined threshold; or,
- a certain number of steps is completed.

So the pseudo code for gradient descent boils down to this:

1. begin with with some arbitrary w

while stopping criteria not met:

2. calculate mean gradient across all examples
3. update w based on gradient and learning rate
4. Repeat

Let's go ahead and implement that now:

```
# """Gradient descent for optimizing slope in simple linear regression with no intercept."""
def gradient_descent(x, y, w, alpha, ϵ=2e-4, max_iterations=5000, print_progress=10):
    print(f"Iteration 0. w = {w:.2f}.")
    iterations = 1  # init iterations
    delta_w = 2 * ϵ      # init delta_w.

    while abs(delta_w) > ϵ and iterations <= max_iterations:
        g = gradient(x, y, w)  # calculate current gradient
        delta_w = alpha * g    # change in w
        #your code!!!!              # adjust w based on gradient * learning rate
        if iterations % print_progress == 0:  # periodically print progress
            print(f"Iteration {iterations}. w = {w:.2f}.")
        iterations += 1        # increase iteration

    print("Terminated!")
    print(f"Iteration {iterations - 1}. w = {w:.2f}.")
```

Task6: In the gradient_descent function, I left one line of code, which will be used to update the weight w based the gradient and learning rate, please complete code for this line to finish the gradient_descent function, and the run the gradient_descent by (10pts):

```
gradient_descent(x, y, w=0.5, alpha=0.00001)
```

Screenshot your implementation of the gradient_descent function and also the output of the above gradient_descent function call

Let's take a look at the journey our slope parameter of our simple linear model went on during its optimization by:

```
fig = libs.plot_gradient_descent(x, y, w=0.5, alpha=0.00001)
fig.write_image('figs/gd1.png')
```

Task7 (25pts): investigating how the learning rate will change the optimization process:
Run the gradient_descent function of w = 0.5 but with learning rate different, try alpha = 0.00005, 0.00015, and 0.00018, then plot all the gradient descent figures by using the plot_gradient_descent function (5pts).
Answer the below two questions :

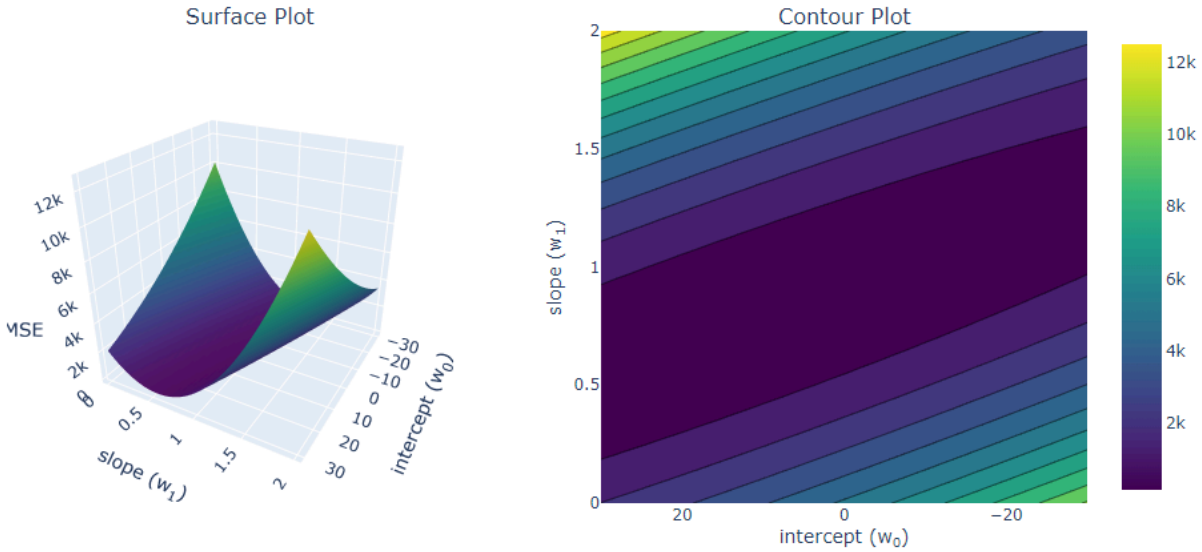
- **Explain how the gradient descent algorithm works (10pts)?**
- **Explain how the learning rate will affect the learning processes (10pts)?**

```
gradient_descent(x, y, w=0.5, alpha=0.00001)
fig = libs.plot_gradient_descent(x, y, w=0.5, alpha=0.00001)
fig.write_image('figs/gd1.png')
```

c. Gradient Descent With Two Parameters

Cool, we just implemented gradient descent from scratch! In this section, we'll try optimizing for two parameters, the intercept and slope of a simple linear regression model simultaneously.
Most of the models you'll be working with will have more than just one parameter to update - neural networks typically have hundreds, thousands, and even millions of parameters! So, let's extend the above workflow to two parameters, the intercept (w_0) and the slope (w_1). Just to help you get a visual of what's going on, let's take our "manual grid search approach" from earlier and make a plot of it but this time with two parameters:

```
slopes = np.arange(0, 2.05, 0.05)
intercepts = np.arange(-30, 31, 2)
fig = libs.plot_grid_search_2d(x, y, slopes, intercepts)
fig.show()
fig.write_html("mse_gridsearch_2d.html")
```



Above is the surface of MSE for different values of `intercept` (w_0) and `slope` (w_1). The approach for implementing gradient descent is exactly as before, but we're operating on two parameters now and the gradient of the intercept is a little different to the slope:

$$f(w) = \frac{1}{n}\sum_{i=1}^{n}((w_0 + w_1 x) - y_i))^2$$

$$\frac{\partial}{\partial w_0} f(w) = \frac{1}{n}\sum_{i=1}^{n} 2((w_0 + w_1 x) - y_i)$$

$$\frac{\partial}{\partial w_1} f(w) = \frac{1}{n}\sum_{i=1}^{n} 2x_i((w_0 + w_1 x) - y_i)$$

Let's define a function that returns these two gradients (partial derivatives) of the slope and intercept.

```
def gradient(x, y, w):
    grad_w0 = (1/len(x)) * 2 * sum(w[0] + w[1] * x - y)
    grad_w1 = (1/len(x)) * 2 * sum(x * (w[0] + w[1] * x - y))
    return np.array([grad_w0, grad_w1])

print(gradient(x, y, w=[10, 0.5]))
```

You can already see that the gradient of our slope is orders of magnitude larger than our intercept... we'll look more at this issue shortly. For now let's re-write our gradient descent function from earlier. It's almost exactly the same as before, but now we are updating the slope **and** the intercept each iteration:

```python
def gradient_descent(x, y, w, alpha, ϵ=2e-4, max_iterations=5000, print_progress=10):
    """Gradient descent for optimizing simple linear regression."""

    print(f"Iteration 0. Intercept {w[0]:.2f}. Slope {w[1]:.2f}.")
    iterations = 1  # init iterations
    delta_w = np.array([ ϵ, ϵ])      # init. delta_w     # init. delta_w

    while abs(delta_w.sum()) > ϵ and iterations <= max_iterations:
        g = gradient(x, y, w)  # calculate current gradient
        delta_w = alpha * g         # change in w
        #   your code !!!!       # adjust w based on gradient * learning rate
        if iterations % print_progress == 0:  # periodically print progress
            print(f"Iteration {iterations}. Intercept {w[0]:.2f}. Slope {w[1]:.2f}.")
        iterations += 1        # increase iteration

    print("Terminated!")
    print(f"Iteration {iterations - 1}. Intercept {w[0]:.2f}. Slope {w[1]:.2f}.")
```

Similar to Task6:, complete the function by giving one line of code to update the implementation of the gradient_descent for two parameters (5pts).

Then run the gradient_descent function:

```python
gradient_descent(x, y, w=[10, 0.5], alpha=0.00001)
```

Hmm... our algorithm worked but our intercept never changed. Let's take a look at what happened:

```python
fig = libs.plot_gradient_descent_2d(x, y, w=[10, 0.5], m_range=np.arange(0, 2.05, 0.05), b_range=np.arange(-30, 31, 2), alpha=0.00001)
fig.write_image('figs/gd2.png')
```

Only the slope changed in value (we see a vertical line in the plot above, with no change in the intercept parameter). That's because the slope gradient is **wayyyyy** bigger than the intercept gradient.

d. Optimizing Logistic Regression

In this final section, I'm going to demonstrate optimizing a Logistic Regression model to drive home some of the points we learned. I'm going to sample 70 "legendary" pokemon (which are typically super-powered) and "non-legendary" pokemon from our dataset:

```python
df = pd.read_csv("data/pokemon.csv", index_col=0, usecols=['name', 'defense', 'legendary']).reset_index()
leg_ind = df["legendary"] == 1
df = pd.concat(
    (df[~leg_ind].sample(sum(leg_ind), random_state=123), df[leg_ind]),
    ignore_index=True,
).sort_values(by='defense')

print(df.head(10))
```

	name	defense	legendary
23	Sunkern	30	0
2	Gastly	30	0
127	Cosmog	31	1
25	Mankey	35	0
5	Remoraid	35	0
6	Kirlia	35	0
60	Tyrogue	35	0
67	Poochyena	35	0
58	Buizel	35	0
11	Noibat	35	0

```
x = StandardScaler().fit_transform(df[['defense']]).flatten()  # we saw before the standardizing is a good idea for optimization
y = df['legendary'].to_numpy()
fig = libs.plot_logistic(x, y)
fig.write_image('figs/lg.png')
```

Task8: Investigate the StandardScaler() function by yourself, explain what it is, when to use it, and other scaling methods can be used to normalize the training data samples (20pts)

We'll be using the "trick of ones" to help us implement these computations efficiently. For example, consider the following simple linear regression model with an intercept and a slope:

$$\hat{y} = \boldsymbol{w}^T \boldsymbol{x} = w_0 \times 1 + w_1 \times x$$

Let's represent that in matrix form:

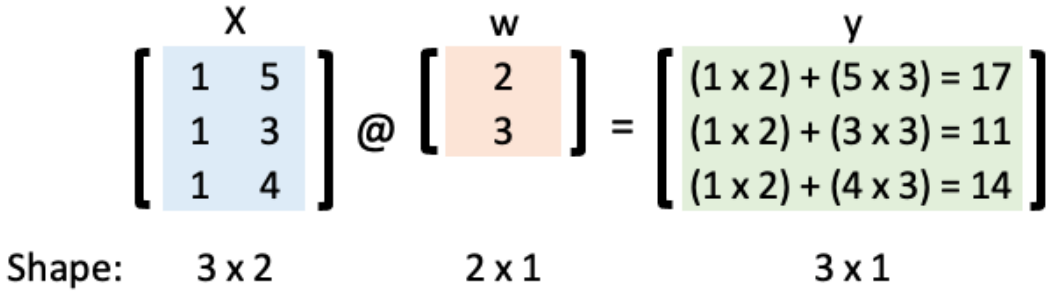
$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \end{bmatrix}$$

Now we can calculate y using matrix multiplication and the "matmul" Python operator:

```
w = np.array([2, 3])
X = np.array([[1, 5], [1, 3], [1, 4]])
X @ w
```

The output will be:

```
array([17, 11, 14])
```



We're going to create a logistic regression model to classify a Pokemon as "legendary" or not. Recall that in logistic regression we map our linear model to a probability:

$$z = \boldsymbol{w}^T \boldsymbol{x}$$

$$P(y=1) = \frac{1}{(1+\exp(-z))}$$

For classification purposes, we typically then assign this probability to a discrete class (0 or 1) based on a threshold (0.5 by default):

$$y = \begin{cases} 0, & P(y=1) \leq 0.5 \\ 1, & P(y=1) > 0.5 \end{cases}$$

Let's code that up:

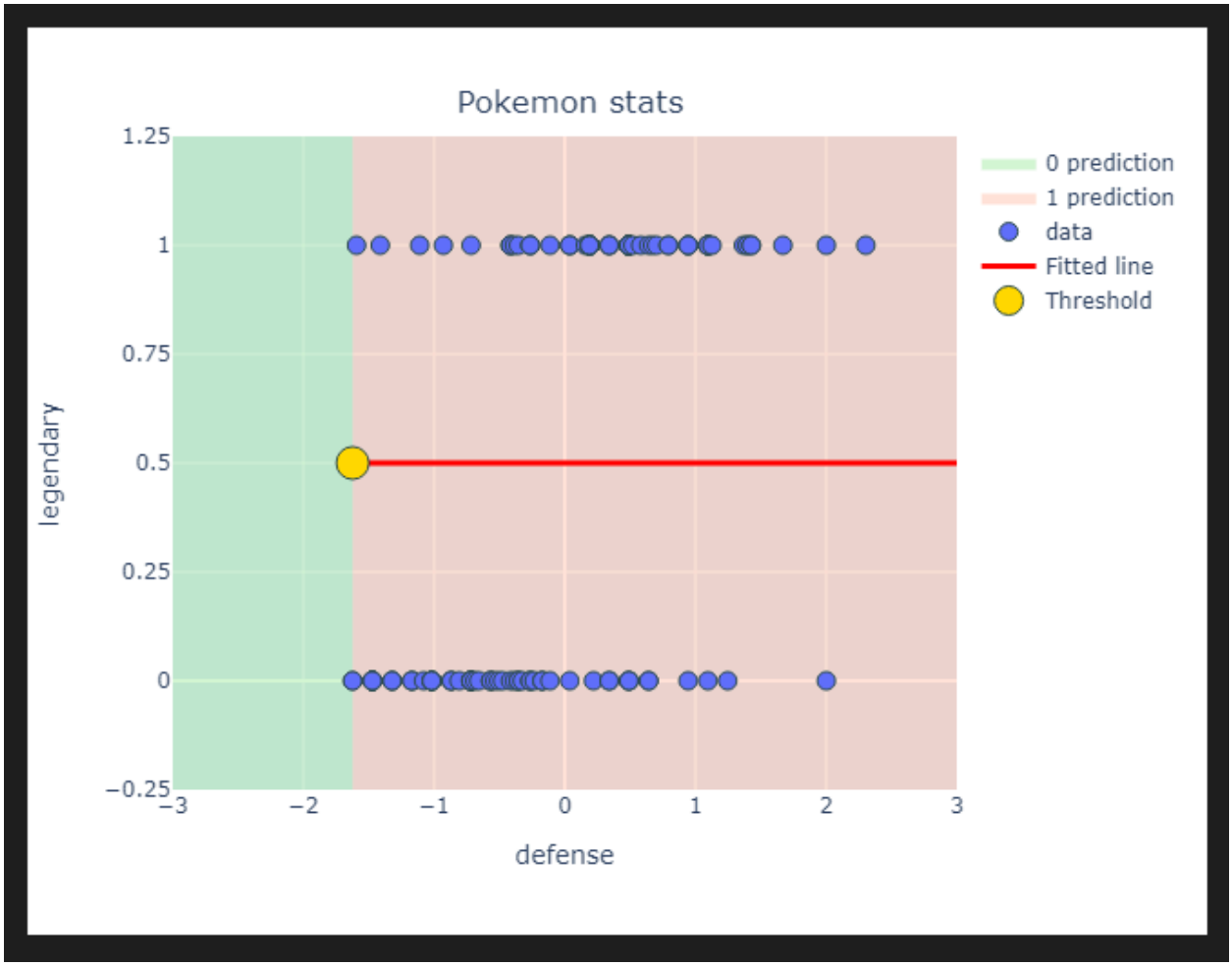
```python
def sigmoid(x, w, output="soft", threshold=0.5):
    p = 1 / (1 + np.exp(-x @ w))
    if output == "soft":
        return p
    elif output == "hard":
        return np.where(p > threshold, 1, 0)
```

Lets first try w = [0, 0]

```python
ones = np.ones((len(x), 1))
X = np.hstack((ones, x[:, None]))  # add column of ones for the intercept term
w = [0, 0]

y_soft = sigmoid(X, w)
y_hard = sigmoid(X, w, "hard")
fig = libs.plot_logistic(x, y, y_soft, threshold=0.5)
fig.write_image('figs/lg1.png')
```

The plot_logistic will look like:



Task9: try w = [-1, 3], plot the logistic figure as above, what's you find (15pts)?

Now, Let's calculate the accuracy of the above model:

```
def accuracy(y, y_hat):
    return (y_hat == y).sum() / len(y)

print(accuracy(y, y_hard))
```

Task10: explain what's the accuracy and how it's been calculated (10pts)?

Just like in the linear regression example earlier, we want to optimize the values of our weights! We need a loss function! We use the log loss (cross-entropy loss) to optimize logistic regression. Here's the loss function and its gradient:

$$f(w) = -\frac{1}{n}\sum_{i=1}^{n} y_i \log\left(\frac{1}{1+\exp(-w^T x_i)}\right) + (1-y_i)\log\left(1-\frac{1}{1+\exp(-w^T x_i)}\right)$$

$$\frac{\partial f(w)}{\partial w} = \frac{1}{n}\sum_{i=1}^{n} x_i\left(\frac{1}{1+\exp(-w^T x_i)} - y_i\right)$$

```python
def logistic_loss(w, X, y):
    return -(y * np.log(sigmoid(X, w)) + (1 - y) * np.log(1 - sigmoid(X, w))).mean()


def logistic_loss_grad(w, X, y):
    return (X.T @ (sigmoid(X, w) - y)) / len(X)

w_opt = minimize(logistic_loss, np.array([-1, 1]), jac=logistic_loss_grad, args=(X, y)).x
print(w_opt)
```

Let's check our solution against the `sklearn` implementation:

```python
lr = LogisticRegression(penalty=None).fit(x.reshape(-1, 1), y)
print(f"w0: {lr.intercept_[0]:.2f}")
print(f"w1: {lr.coef_[0][0]:.2f}")
```

This is what the optimized model looks like:

```python
y_soft = sigmoid(X, w_opt)
fig = libs.plot_logistic(x, y, y_soft, threshold=0.5)
fig.write_image('figs/lg3.png')
```

I mean, that's so cool! We replicated the sklearn behavior from scratch!!!! By the way, I've been doing things in 2D here because it's easy to visualize, but let's double check that we can work in more dimensions by using `attack`, `defense` and `speed` to classify a Pokemon as `legendary` or not:

Task11: Perform the logistic regression on the pokemon dataset by three dimensions(`attack`, `defense` and `speed`) to classify a Pokemon as `legendary` or not with 140 data samples(70 for legendary, 70 for not legendary). Show your code implementation and indicate the optimal weights (50pts).