Problem 1

a. Regression
b. Classification

Problem 2

a. 0.034
b. 0.027
c. 0.028

Problem 3

d is in the direction of gradient **descent**, however **a** is in the direction of the gradient.

Problem 4

a. Class B
b. Class R

M1-L1 Problem 1 (12 points)

You are given 14 temperature measurements from 14 thermocouples in a factory. A model has produced 14 temperature predictions, one for each thermocouple. You must compute the the error vector and MSE between the predicted and measured temperatures via a few methods.

Run the next cell to load the data; then proceed through the notebook.

- `y_data` is y, a 14x1 array of temperature measurements (in deg C)
- `y_pred` is $\hat{y}$, a 14x1 array of temperature predictions

In [6]:
```
import numpy as np
np.set_printoptions(precision=4)

y_data = np.array([[20,21,30,30,21,25,38,37,30,22,22,38,20,35]],dtype=np.double).T
y_pred = np.array([[21,21,31,30,20,28,36,32,31,20,21,39,21,34]],dtype=np.double).T

print("[y_data, y_pred] = \n")
print(np.c_[y_data, y_pred])
```

```
[y_data, y_pred] =

[[20. 21.]
 [21. 21.]
 [30. 31.]
 [30. 30.]
 [21. 20.]
 [25. 28.]
 [38. 36.]
 [37. 32.]
 [30. 31.]
 [22. 20.]
 [22. 21.]
 [38. 39.]
 [20. 21.]
 [35. 34.]]
```

Error vector

First, compute the error vector $y_{err} = y - \hat{y}$. Call the result `y_err`. It should be 14x1.

You may do this with a loop, or -- better yet -- by simply subtracting the two arrays.

In [2]:
```
# YOUR CODE GOES HERE
# Compute y_err
y_err = y_data - y_pred
```

```
print("Size of y_err:", np.shape(y_err))
```

```
Size of y_err: (14, 1)
```

Mean squared error (MSE)

Now compute the MSE,

$$ MSE = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2 = \frac{1}{N} \sum_{i=1}^N y_{err}^2 $$

MSE with Loop

First, compute this quantity by using a for loop to loop through y_err, performing the necessary operations to compute MSE.
Call the result `MSE_loop`.
Your result should be $\approx$ 3.5714.

In [3]:
```
# YOUR CODE GOES HERE
# Compute MSE_loop
MSE_loop = 0.00
for i in range(0,len(y_err)):
  MSE_loop = MSE_loop + y_err[i]**2
MSE_loop = MSE_loop/len(y_err)

print("MSE (loop) = ", MSE_loop)
```

```
MSE (loop) =  [3.5714]
```

MSE by matrix multiplication

Another way to compute the MSE is by recognizing that the sum $ \sum_{i=1}^N y_{err}^2 $ equals the matrix product $ y_{err}' \cdot y_{err} $.
Therefore: $$ MSE = \frac{1}{N} y_{err}' \cdot y_{err} $$

Compute the MSE this way. Call it `MSE_mm`, and make sure the result is the same. This is a much more efficient way of computing the MSE in Python.

Note that you can compute the transpose of a 2D array `A` with `A.T`, and you can multiply matrices `A` and `B` with `A @ B`.

In [4]:
```
# YOUR CODE GOES HERE
# Compute MSE_mm
MSE_mm = y_err.T@y_err/len(y_err)

print("MSE (matrix multiplication) = ", MSE_mm)
```

```
MSE (matrix multiplication) =  [[3.5714]]
```

MSE by numpy mean

Now you will compute the MSE once more, but using numpy operations. Use `np.mean()` to take an average. Compute the square of `y_err` with either `np.square()` or `y_err * y_err`.

Call your `MSE_np`, and make sure the result is the same. This is also much more efficient than a Python for loop.

In [5]:
```
# YOUR CODE GOES HERE
# Compute MSE_np
MSE_np = np.mean(np.square(y_err))

print("MSE (Numpy) = ", MSE_np)
```

```
MSE (Numpy) =  3.5714285714285716
```

In [ ]:

M1-L2 Problem 1 (12 points)

In this problem, we are given a function $L(w_1, w_2)$ with a known functional form. You will perform gradient descent to find a global minimum. The goal is to find what initial guesses and learning rates (step sizes) lead the algorithm to find the global minimum.

The function $L(w_1, w_2)$ is defined as: $$ L(w_1, w_2) = \cos(4 w_1 + w_2 / 4 - 1) + w_2^2 + 2 w_1^2 $$ A Python function for `L(w_1, w_2)` is given.

Gradients

First, we must define a gradient of L. That is $\nabla L = \left[\frac{\partial L}{\partial w_1} , \frac{\partial L}{\partial w_2}\right]$. First, compute these derivatives by hand. Then, in the cell below, complete the functions for the derivatives of `L` with respect to `w1` and `w2`.

In [4]:
```
import numpy as np
import matplotlib.pyplot as plt

def L(w1, w2):
    return np.cos(4*w1 + w2/4 - 1) + w2*w2 + 2*w1*w1

def dLdw1(w1, w2):
    # YOUR CODE GOES HERE
    return -np.sin(4*w1 + w2/4 - 1)*4 + 4*w1

def dLdw2(w1, w2):
    # YOUR CODE GOES HERE
    return -np.sin(4*w1 + w2/4 - 1)/4 + 2*w2
```

Gradient Descent

The function `plot_gd` performs gradient descent by calling your derivative functions. Take a look at how this works. Then, run the interactive gradient descent cell that follows and answer the questions below.

In [5]:
```
def plot_gd(w1, w2, log_stepsize, log_steps):
    stepsize = 10**log_stepsize
    steps = int(10**log_steps)

    # Gradient Descent
    w1s = np.zeros(steps+1)
    w2s = np.zeros(steps+1)

    for i in range(steps):
        w1s[i], w2s[i] = w1, w2
        w1 = w1 - stepsize * dLdw1(w1s[i],w2s[i])
```

```
        w2 = w2 - stepsize * dLdw2(w1s[i],w2s[i])
    w1s[steps], w2s[steps] = w1, w2

    # Plotting
    vals = np.linspace(-1,1,50)
    x, y = np.meshgrid(vals,vals)
    z = L(x,y)

    plt.figure(figsize=(7,5.8),dpi=120)
    plt.contour(x,y,z,colors="black", levels=np.linspace(-.5,3,6))
    plt.pcolormesh(x,y,z,shading="nearest",cmap="Blues")
    plt.colorbar()

    plt.plot(w1s,w2s,"g-",marker=".",markerfacecolor="black",markeredgecolor="None"
    plt.scatter(w1s[0],w2s[0],zorder=100, color="blue",marker="o",label=f"$w_0$ = [
    plt.scatter(w1,w2,zorder=100,color="red",marker="x",label=f"$w^*$ = [{w1:.2f},
    plt.legend(loc="upper left")

    plt.axis("equal")
    plt.box(False)
    plt.xlabel("$w_1$")
    plt.ylabel("$w_2$")
    plt.xlim(-1,1)
    plt.ylim(-1,1)
    plt.title(f"Step size = {stepsize:.0e}; {steps} steps")
    plt.show()
```

In [6]:

```
%matplotlib inline
from ipywidgets import interact, interactive, fixed, interact_manual, Layout, Float

slider1 = FloatSlider(
    value=0,
    min=-1,
    max=1,
    step=.1,
    description='w1 guess',
    disabled=False,
    continuous_update=True,
    orientation='horizontal',
    readout=False,
    layout = Layout(width='550px')
)

slider2 = FloatSlider(
    value=0,
    min=-1,
    max=1,
    step=.1,
    description='w2 guess',
    disabled=False,
    continuous_update=True,
    orientation='horizontal',
    readout=False,
    layout = Layout(width='550px')
)
```

```
slider3 = FloatSlider(
    value=-1.5,
    min=-3,
    max=0,
    step=.5,
    description='step size',
    disabled=False,
    continuous_update=True,
    orientation='horizontal',
    readout=False,
    layout = Layout(width='550px')
)

slider4 = FloatSlider(
    value=2,
    min=0,
    max=3,
    step=.25,
    description='steps',
    disabled=False,
    continuous_update=True,
    orientation='horizontal',
    readout=False,
    layout = Layout(width='550px')
)


interactive_plot = interactive(
    plot_gd,
    w1 = slider1,
    w2 = slider2,
    log_stepsize = slider3,
    log_steps = slider4,
    )
output = interactive_plot.children[-1]
output.layout.height = '620px'

interactive_plot
```

Out[6]:
```
interactive(children=(FloatSlider(value=0.0, description='w1 guess', layout=Layout
(width='550px'), max=1.0, mi…
```

Questions

Play around with the sliders above to get an intuition for which initial conditions/learning rates lead us to find the global minimum at [-0.42, -0.05]. Then answer the following questions:

1. Set w_0 to [0.2, 0.8] and step size to 1e-01. After 100 steps of gradient descent, what w^* do we reach?

2. Keep parameters from the previous question, but change the initial guess to [0.3, 0.8]. Now what is the optimum we find?

3. Set w_0 to [-1.0, -1.0] and number of iterations to 1000 and step size to 1e-03. What w^* do we reach, and why is it not exactly the global minimum?

4. In general, what happens if we set learning rate too large?

Answers

1. w_* = [-0.42, -0.05]
2. w_* = [0.80, 0.10]
3. w_* = [-0.42, -0.18]. We dont reach the global minimum as the initial guess is considerably far away from the minimum point, in addition to the step size being very small. Hence, the model is not able to converge to the global minimum in the specified number of steps.
4. If we set the learning rate too large, the gradient descent generally fails to converge. This is because the model's parameters are modified too aggresively/drastically and this causes the model to diverge or oscillate.

In [ ]:

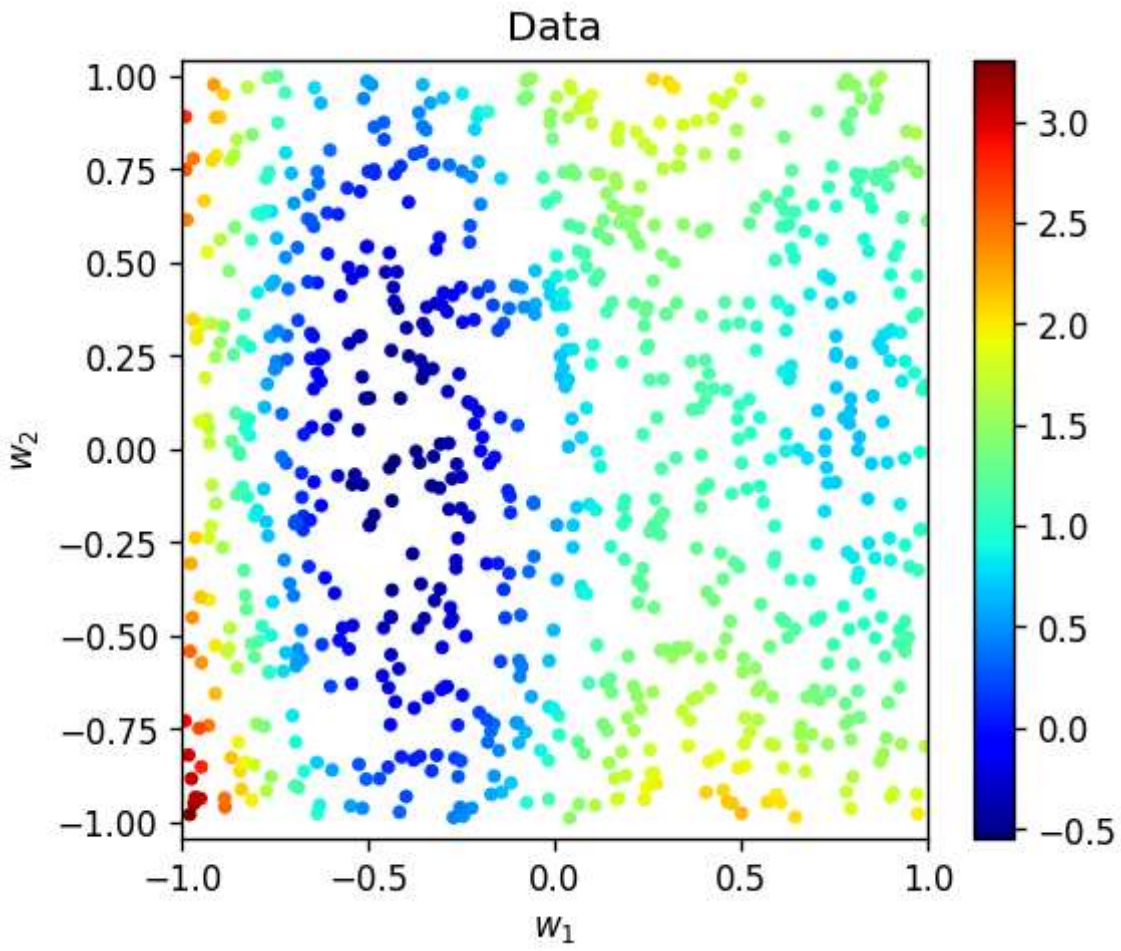
M1-L2 Problem 2 (12 points)

In this problem, you will implement a K-NN regressor from scratch. Start by running the following cell to load the dataset.

Dataset:

- `w1_data` : w_1 values
- `w2_data` : w_2 values
- `L_data` : L values

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(42)
N = 876
w1_data = np.random.uniform(-1,1,N)
w2_data = np.random.uniform(-1,1,N)
L_data = np.cos(4*w1_data + w2_data/4 - 1) + w2_data**2 + 2*w1_data**2

plt.figure(figsize=(5,4.2),dpi=120)
plt.scatter(w1_data,w2_data,s=10,c=L_data,cmap="jet")
plt.colorbar()
plt.axis("equal")
plt.xlabel("$w_1$")
plt.ylabel("$w_2$")
plt.xlim(-1,1)
plt.ylim(-1,1)
plt.title("Data")
plt.show()
```



K - Nearest Neighbors Regressor

Distance function

Now we will implement an unweighted K-NN regressor. First, finish the function `distance(w1, w2)` which computes the euclidean distance between a point `[w1, w2]` and each pair from `w1_data, w2_data`. The function should return an array of distances with the same length as `w1_data` or `w2_data`. Instead of using a for loop, you can do this by subtracting each individual scalar from the corresponding data array. For example, `w1 - w1_data` is an array that contains the difference between `w1` and each element in `w1_data`. Complete the function to compute the array $\sqrt{(w_1 - w_{1,data (i)})^2 + (w_2 - w_{2,data (i)})^2}$.

In [2]:
```
def distance(w1, w2):
    # YOUR CODE GOES HERE
    dist = np.sqrt(np.square(w1-w1_data)+np.square(w2-w2_data))
    return dist

# Check that the function outputs the correct array size
assert(distance(0, 0).shape == w1_data.shape)
```

Sorting a distance array

You can get the k-smallest elements of an array by using the `np.argpartition()` function. `np.argpartition(A, k)[:k]` returns an array of `k` indices corresponding to the k-smallest values in `A`. If we apply this to an array of distances from a point w to each data point, we can get the indices of the k-nearest neighbors of w. Complete the function below to do this.

In [3]:
```
def get_knn_indices(w1, w2, k):
    d = distance(w1, w2)
    # YOUR CODE GOES HERE
    indices = np.argpartition(d,k)[:k]
    return indices

# Check the function on the point w=(0,0) with k=5
indices = get_knn_indices(0,0,5)
print("5 points nearest (0,0):", indices, "\n(Should be 255 733 538 815 501)")
```

```
5 points nearest (0,0): [815 501 733 255 538]
(Should be 255 733 538 815 501)
```

Unweighted regression

After acquiring the indices of the nearest points, you can determine the output values at these points by indexing into `L_data`, as in: `L_data[indices]`. Then, the function `np.mean()` can be used to compute the average value of these points. Complete the function below to do this. Return from this function a single value, the average of the k points closest to w.

In [4]:
```
def knn_regress(w1, w2, k):
    indices = get_knn_indices(w1, w2, k)
    # YOUR CODE GOES HERE
    avg = np.mean(L_data[indices])
    return avg

# Check the function on the point w=(0,0) with k=2
val = knn_regress(0,0,2)
print("Mean of 2 points nearest (0,0):", val, "\n(Should be about 0.72)")
```

```
Mean of 2 points nearest (0,0): 0.7190087852048137
(Should be about 0.72)
```

Plotting the K-NN function

Now we will evaluate the function on a meshgrid of points. `np.meshgrid` is used frequently for 2D visualization, so step through the next few cells to see how it works.

First, we choose arrays of values for `w1` and `w2` that we want to be the x- and y-coordinates of grid points:

In [5]:
```
w1_vals = np.linspace(-1,1,50)
w2_vals = np.linspace(-1,1,50)
print("w1 grid values:",w1_vals)
print("w2 grid values:",w2_vals)
```

```
w1 grid values: [-1.         -0.95918367 -0.91836735 -0.87755102 -0.83673469 -0.7959
1837
 -0.75510204 -0.71428571 -0.67346939 -0.63265306 -0.59183673 -0.55102041
 -0.51020408 -0.46938776 -0.42857143 -0.3877551  -0.34693878 -0.30612245
 -0.26530612 -0.2244898  -0.18367347 -0.14285714 -0.10204082 -0.06122449
 -0.02040816  0.02040816  0.06122449  0.10204082  0.14285714  0.18367347
  0.2244898   0.26530612  0.30612245  0.34693878  0.3877551   0.42857143
  0.46938776  0.51020408  0.55102041  0.59183673  0.63265306  0.67346939
  0.71428571  0.75510204  0.79591837  0.83673469  0.87755102  0.91836735
  0.95918367  1.        ]
w2 grid values: [-1.         -0.95918367 -0.91836735 -0.87755102 -0.83673469 -0.7959
1837
 -0.75510204 -0.71428571 -0.67346939 -0.63265306 -0.59183673 -0.55102041
 -0.51020408 -0.46938776 -0.42857143 -0.3877551  -0.34693878 -0.30612245
 -0.26530612 -0.2244898  -0.18367347 -0.14285714 -0.10204082 -0.06122449
 -0.02040816  0.02040816  0.06122449  0.10204082  0.14285714  0.18367347
  0.2244898   0.26530612  0.30612245  0.34693878  0.3877551   0.42857143
  0.46938776  0.51020408  0.55102041  0.59183673  0.63265306  0.67346939
  0.71428571  0.75510204  0.79591837  0.83673469  0.87755102  0.91836735
  0.95918367  1.        ]
```

Next, we get a 'cartesian product' of these arrays -- we get every combination of them; these will be our grid points. For this we use `np.meshgrid()`.

Note that we flatten these arrays to get 1-D arrays of the grid points' coordinates:

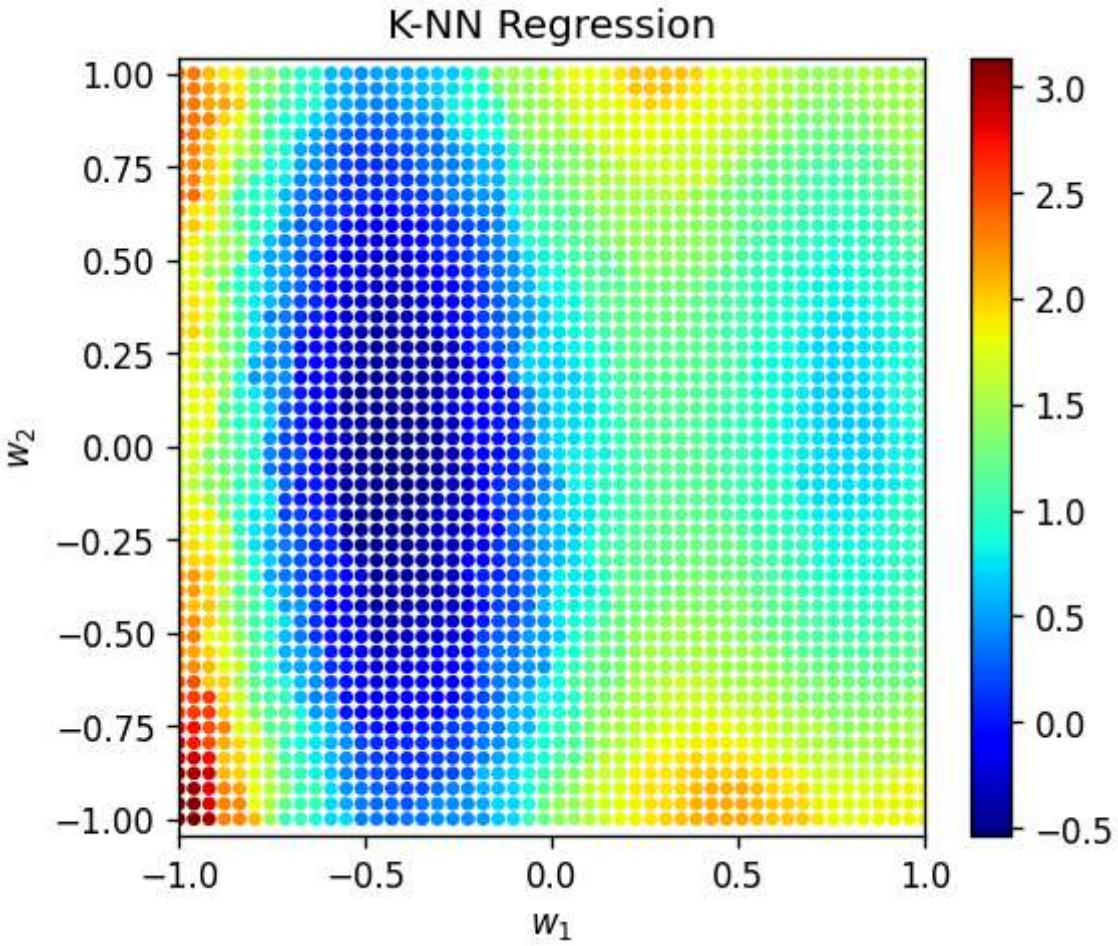
In [6]:
```
w1s, w2s = np.meshgrid(w1_vals, w2_vals)
print("Size of w1 grid point array:", w1s.shape)
print("Size of w2 grid point array:", w2s.shape)

w1_grid, w2_grid = w1s.flatten(), w2s.flatten()
print("Flattened size of w1 grid point array:", w1_grid.shape)
print("Flattened size of w2 grid point array:", w2_grid.shape)
```

```
Size of w1 grid point array: (50, 50)
Size of w2 grid point array: (50, 50)
Flattened size of w1 grid point array: (2500,)
Flattened size of w2 grid point array: (2500,)
```

Now, we can loop through these arrays to call our K-NN regression function on the whole meshgrid, and plot it. Here we set k = 4, but this will be changed later.

In [7]:
```
k = 4
L_grid = np.zeros_like(w1_grid)
for i in range(len(L_grid)):
    L_grid[i] = knn_regress(w1_grid[i], w2_grid[i],k)
```

In [8]:
```
plt.figure(figsize=(5,4.2),dpi=120)
plt.scatter(w1_grid,w2_grid,s=10,c=L_grid,cmap="jet")
plt.colorbar()
plt.axis("equal")
plt.xlabel("$w_1$")
plt.ylabel("$w_2$")
plt.xlim(-1,1)
plt.ylim(-1,1)
plt.title("K-NN Regression")
plt.show()
```



Question

Go back a couple cells and experiment with changing the `k` value. Is the regression function "smoother" with lower or higher `k`? Why do you think that is?

As we increase the k value, the function becomes smoother. This is because at higher k values, the averaging is done over a larger number of neighbors. This means that for a small variation in the data, the prediction value dooes changes smoothly as compared to the case of a lower value of k.

Problem 1 (36 points)

Problem Description

In this problem you will implement gradient descent on the following function: $f(x) = x^2 + 3x +6\textrm{sin}(x)$. You will define your own gradient function $\texttt{fgrad}(x)$, and then using the provided learning rate $\eta = 0.15$ and initial guess $x_0 = 8$, you will print the value of x and $f(x)$ for the first 10 iterations.

Fill out the notebook as instructed, making the requested plots and printing necessary values.

Summary of deliverables:

Functions:

- `fgrad(x)`

Results:

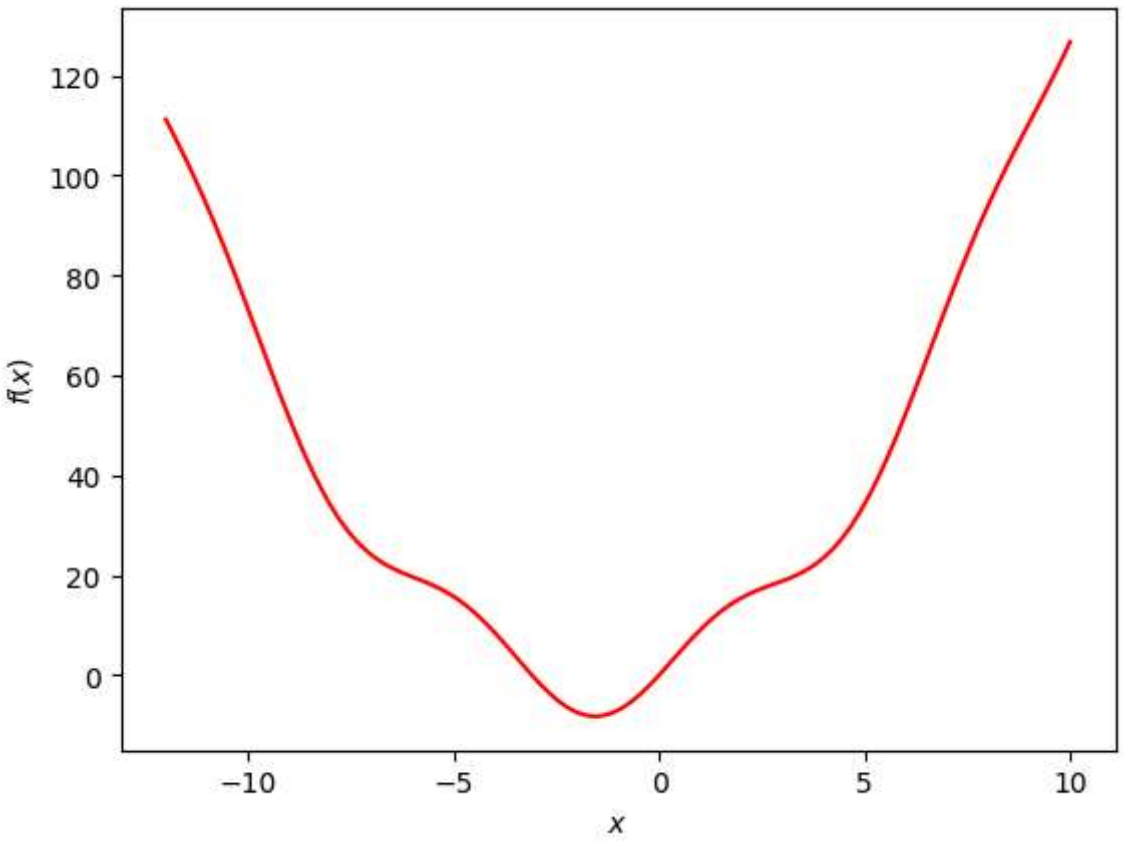
- Printed values of x and $f(x)$ for the first 10 iterations of gradient descent

Discussion:

- Do your printed values appear to be converging towards the minimum of the function?

Imports and provided functions:

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt

def f(x):
    return x**2 + 3*x + 6*np.sin(x)

def plotfx():
    # Sample function
    xs = np.linspace(-12,10,100)
    ys = f(xs)
    # Plot function
    plt.plot(xs,ys,'r-')
    plt.xlabel('$x$')
    plt.ylabel('$f(x)$')
    plt.show()

# Visualize the function
plotfx()
```



First define the function $\texttt{fgrad}(x)$

In [2]:
```
# Your fgrad(x) function goes here
def fgrad(x):
    return 2*x + 3 + 6*np.cos(x)
```

Fill in the following code with the gradient descent update rule

For reference, your 10th iteration should have $x = -1.554$ and $f(x) = -8.246$

In [3]:
```
iter = 10
eta = 0.15
x = 8

for i in range(iter):
    # YOUR GRADIENT DESCENT CODE GOES HERE
    x = x - eta*fgrad(x)

    print('Iteration %d, x = %.3f, f(x) = %.3f' %(i+1, x, f(x)))
```

```
Iteration 1, x = 5.281, f(x) = 38.675
Iteration 2, x = 2.762, f(x) = 18.138
Iteration 3, x = 2.319, f(x) = 16.734
Iteration 4, x = 1.786, f(x) = 14.410
Iteration 5, x = 0.993, f(x) = 8.988
Iteration 6, x = -0.247, f(x) = -2.147
Iteration 7, x = -1.496, f(x) = -8.233
Iteration 8, x = -1.565, f(x) = -8.246
Iteration 9, x = -1.551, f(x) = -8.246
Iteration 10, x = -1.554, f(x) = -8.246
```

Briefly discuss whether your printed values of x and $f(x)$ appear to have converged to the minimum of the function.

Feel free to refer to the provided plot of $f(x)$ above

Your response goes here

We can see that as the value of x changes initially, there is a significant dip in the value of $f(x)$. However, as we approach the 10th iteration, the value of x starts to approach or converge to -1.554. Along with this, it can also be seen that the value of $f(x)$ converges to around -8.246. Therefore, it looks like gradient descent has found a local minima.

In [ ]:

Problem 2 (36 points)

Problem Description

Here, you will perform *weighted* KNN regression.

After you write your own code for weighted KNN regression, you will also try out sklearn's built-in KNN regressor.

Fill out the notebook as instructed, making the requested plots and printing necessary values.

Summary of deliverables:

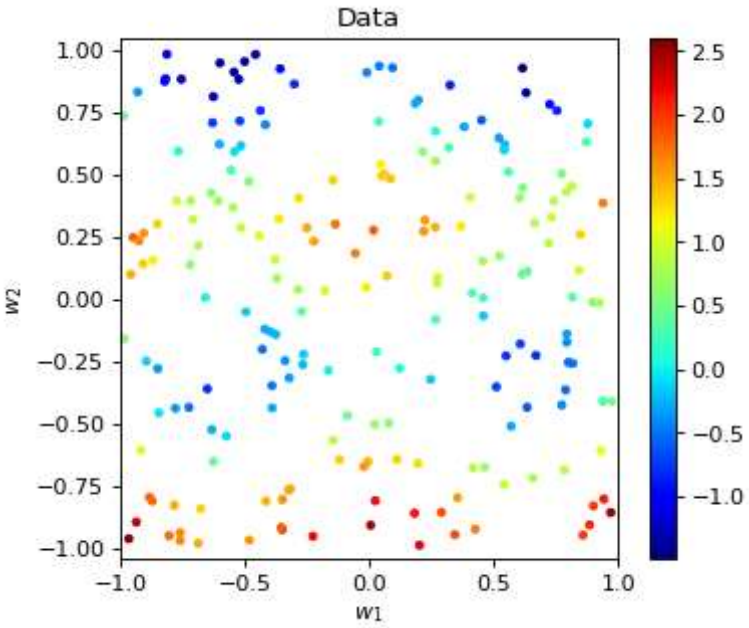
Functions:

- `weighted_knn(w1, w2, k)`

Plots:

- 3 plots of by-hand KNN results
- 3 plots of sklearn.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsRegressor

# Data generation -- don't change
np.random.seed(42)
N = 200
w1_data = np.random.uniform(-1,1,N)
w2_data = np.random.uniform(-1,1,N)
L_data = np.cos(4*w1_data) + np.sin(5*w2_data) + 2*w1_data**2 - w2_data/2
# (end of data generation)

plt.figure(figsize=(5,4.2),dpi=80)
plt.scatter(w1_data,w2_data,s=10,c=L_data,cmap="jet")
plt.colorbar()
plt.axis("equal")
plt.xlabel("$w_1$")
plt.ylabel("$w_2$")
plt.xlim(-1,1)
plt.ylim(-1,1)
plt.title("Data")
plt.show()
```



Weighted KNN function

Here, define a function, `weighted_knn(w1, w2, k)`, which takes in a point at [`w1`, `w2`] and a `k` value, and returns the weighted KNN prediction.

- As in the lecture activity, data is in the variables `w1_data`, `w2_data`, and `L_data`.
- You can create as many helper functions as you want
- The key difference between unweighted and weighted KNN is summarized below:

Unweighted KNN

1. Find the k data points closest to the target point w
2. Get the output values at each of these points
3. Average these values together: this is the prediction at w

Weighted KNN

1. Find the k data points closest to the target point w
2. Compute the proximity of each of these points as $\text{prox}_i = 1/(\text{distance}(w, w_i) + 1e-9)$
3. For each w_i, multiply prox_i by the output value at w_i, and divide by the sum of all k proximities
4. Add all k of these results together: this is the prediction at w

In [2]:
```
# defining distance function
def distance(w1, w2):
    return np.sqrt((w1 - w1_data)**2 + (w2 - w2_data)**2)
```

```
def weighted_knn(w1, w2, k):
    # YOUR CODE GOES HERE
    d = distance(w1, w2)
    indices = np.argpartition(d, k)[:k]

    prox = 1/(d[indices] + np.exp(-9))
    weights = prox/np.sum(prox)
    knn_weighted = np.sum(np.dot(L_data[indices], weights))/k
    return knn_weighted
```

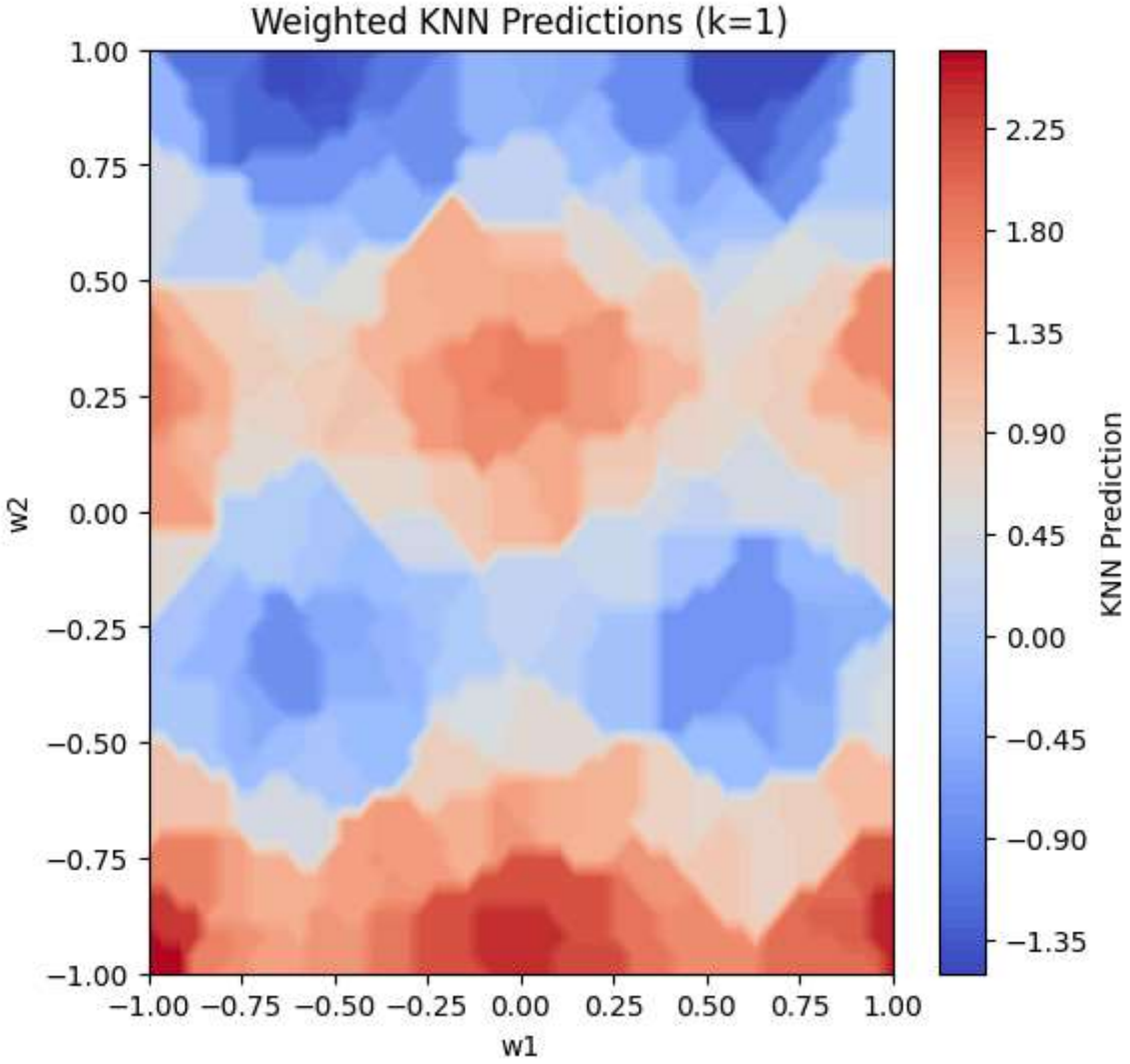
Plotting

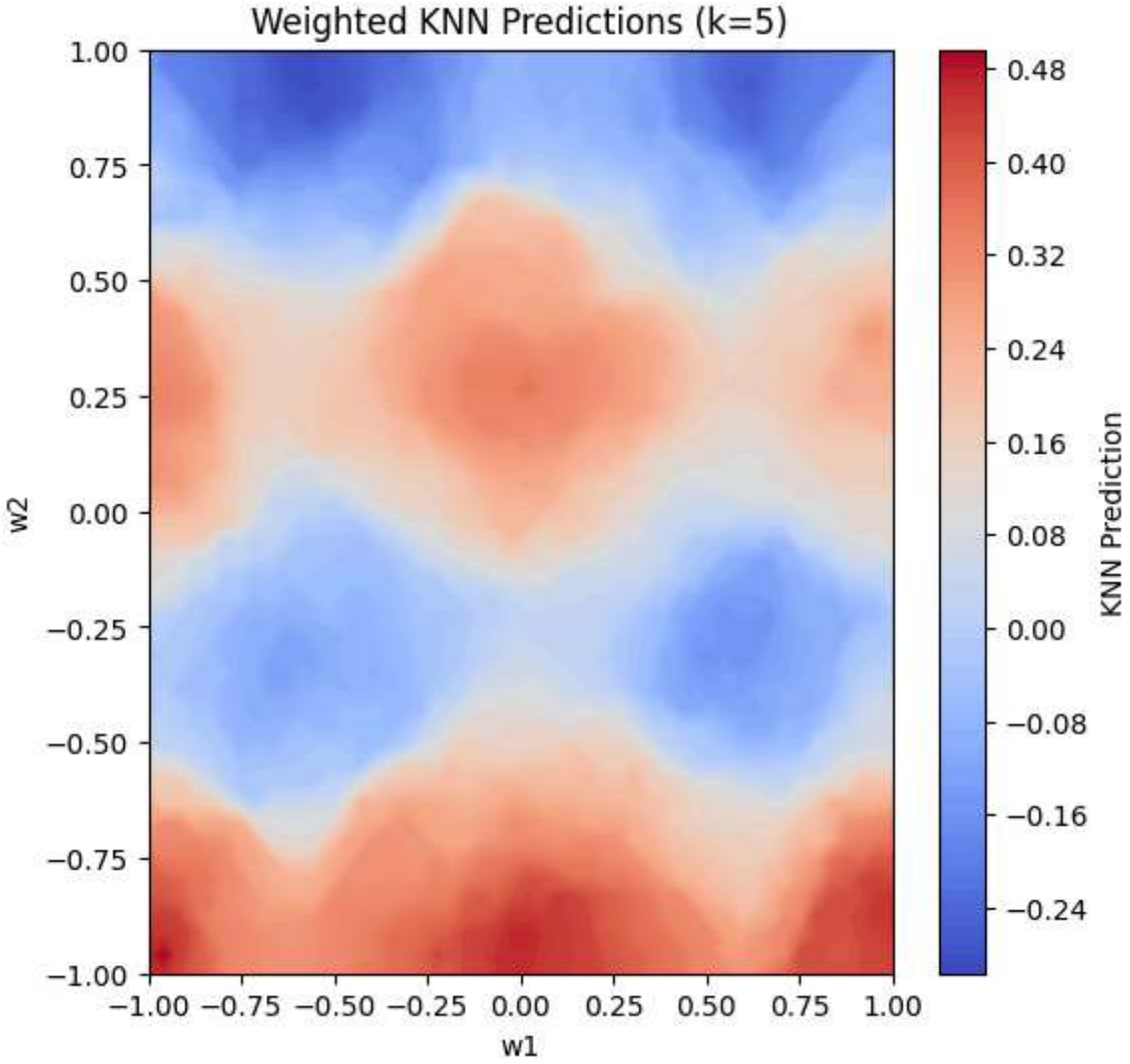
Now create 3 plots showing KNN regressor predictions for k values [1, 5, 25].

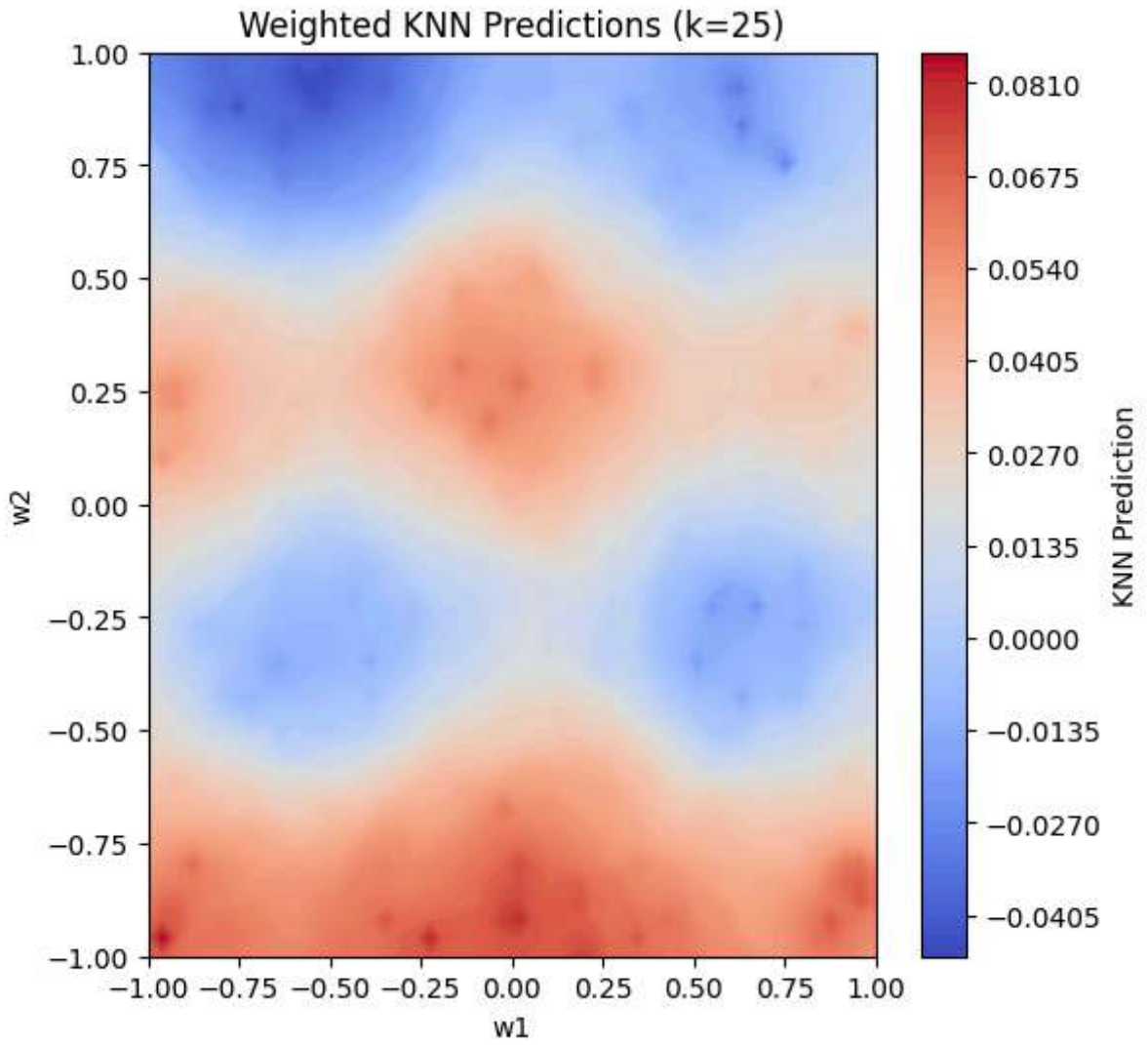
You should plot a 50x50 grid of points on a grid for `w1` and `w2` values between -1 and 1. Consult the lecture activity for how to do this.

We recommend creating a function, e.g. `plot(k)`, so that you need to rewrite less code.

In [4]:
```
# YOUR CODE GOES HERE
# Visualize results for k = 1, 5, and 25
def plot(k):
    vals = np.linspace(-1,1,50)
    x, y = np.meshgrid(vals, vals)

    data = predictions(x, y, k)

    plt.figure(figsize=(6, 6))
    plt.contourf(x, y, data, cmap='coolwarm', levels=100)
    plt.colorbar(label='KNN Prediction')
    plt.title(f'Weighted KNN Predictions (k={k})')
    plt.xlabel('w1')
    plt.ylabel('w2')
    plt.show()


def predictions(x, y, k):
    L = np.zeros((50, 50))
    for i in range(50):
        for j in range(50):
            L[i, j] = weighted_knn(x[i, j], y[i, j], k)
    return L

for k in [1, 5, 25]:
    plot(k)
```


Weighted KNN Predictions (k=1)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
w2
−1.00 −0.75 −0.50 −0.25 0.00 0.25 0.50 0.75 1.00
w1
2.25
1.80
1.35
0.90
0.45
0.00
−0.45
−0.90
−1.35
KNN Prediction


Weighted KNN Predictions (k=5)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
w2
−1.00 −0.75 −0.50 −0.25 0.00 0.25 0.50 0.75 1.00
w1
0.48
0.40
0.32
0.24
0.16
0.08
0.00
−0.08
−0.16
−0.24
KNN Prediction



Using SciKit-Learn

We can also use sklearn's `KNeighborsRegressor()`, which is a very efficient implementation of KNN regression.

The code to do this has been done for one case below. First, make note of how this is done.

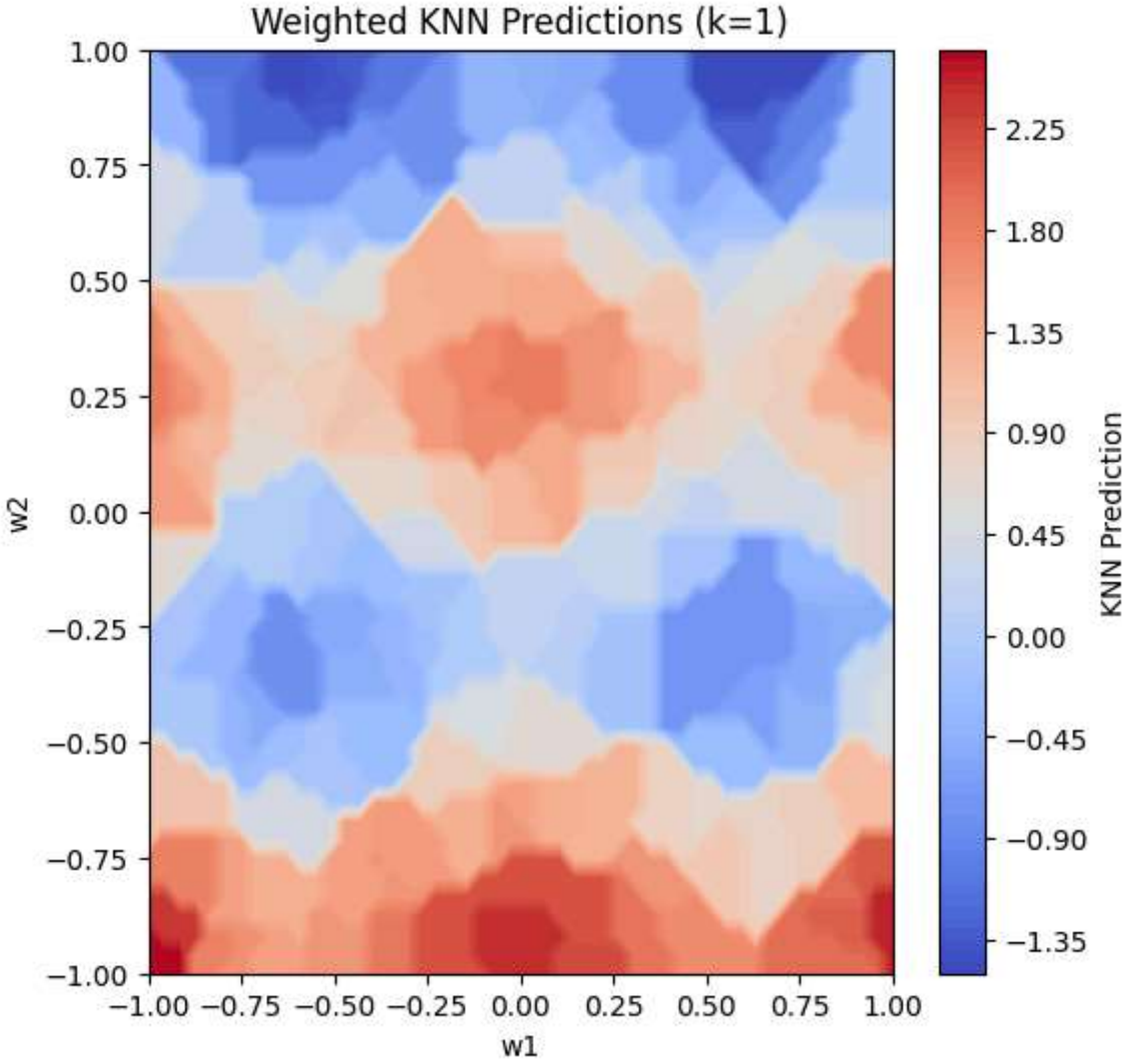
In [5]:
```
model = KNeighborsRegressor(n_neighbors = 1, weights="distance")
X = np.vstack([w1_data,w2_data]).T
model.fit(X, L_data)

# Get a prediction at a point (0, 0):
print(model.predict(np.array([[0,0]])))
```

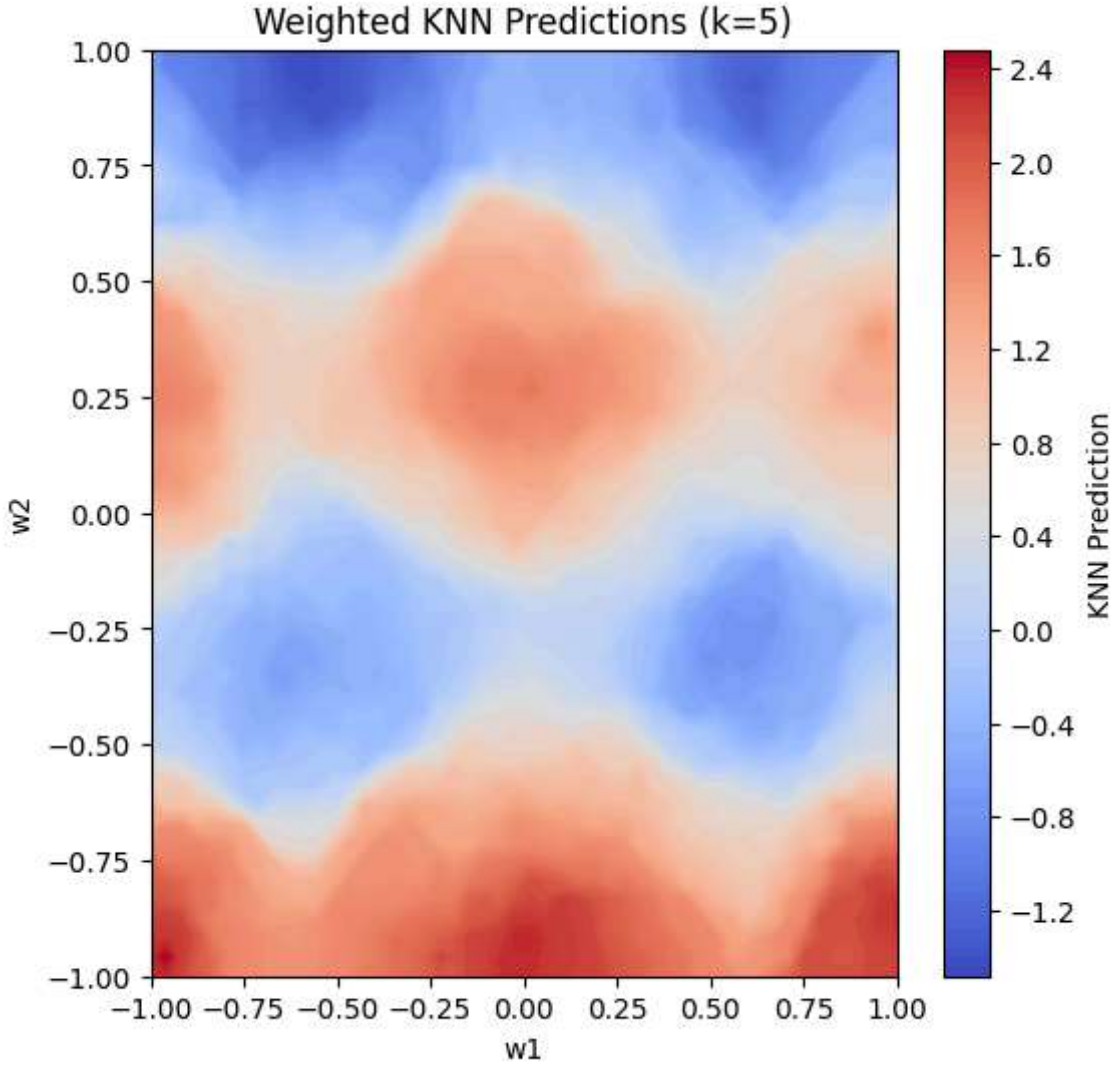
```
[1.19743607]
```

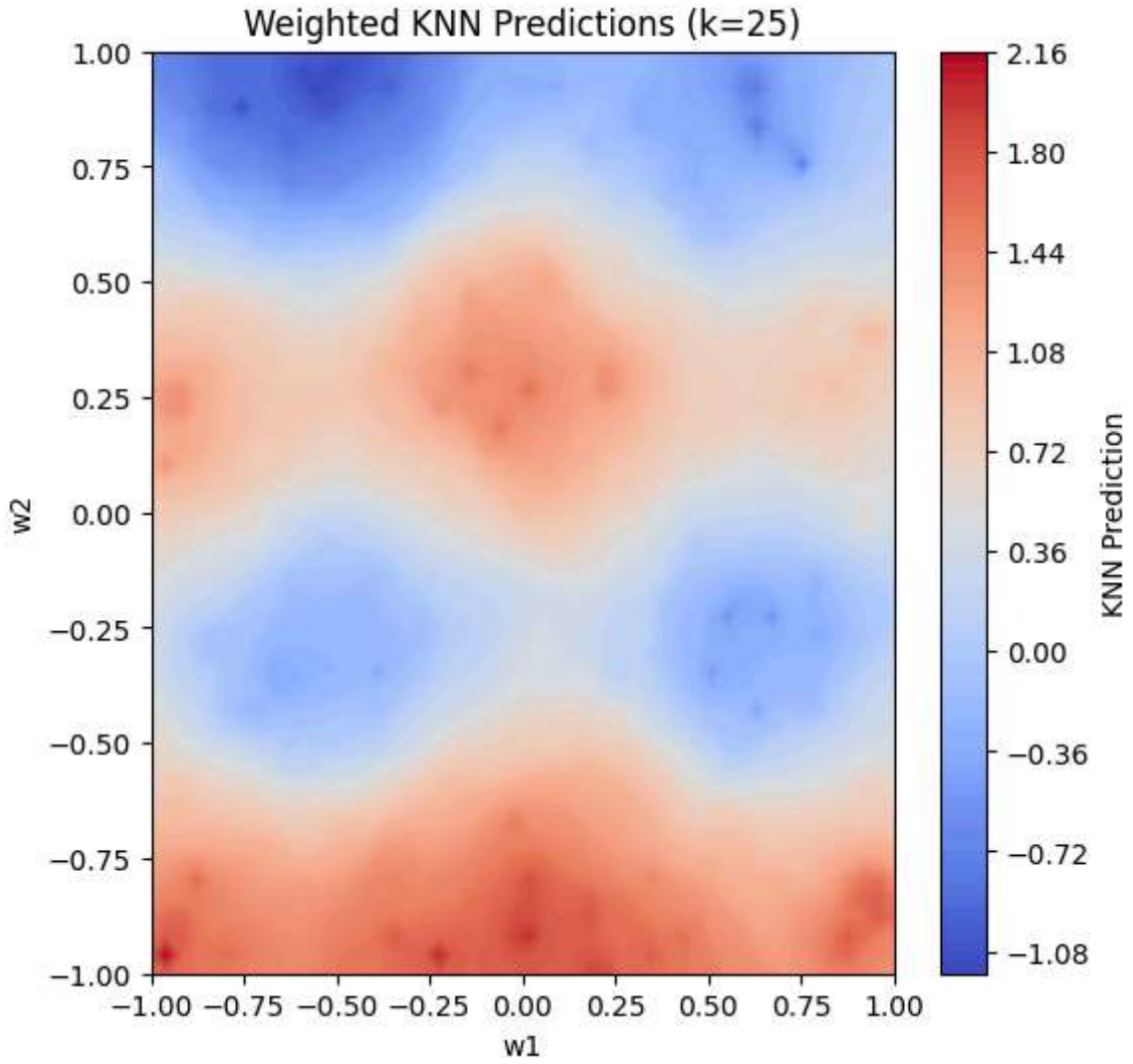
Now create 3 plots for the same values of k as before, using this KNN implementation instead. You can make sure these are visually the same as your from-scratch KNN regressor.

In [6]:

```python
# YOUR CODE GOES HERE
# Visualize sklearn results for k = 1, 5, and 25
def plot(k):
    vals = np.linspace(-1,1,50)
    x, y = np.meshgrid(vals, vals)
    X = np.vstack([w1_data, w2_data]).T # making the input grid

    model = KNeighborsRegressor(n_neighbors = k, weights="distance") # defining the
    model.fit(X, L_data) # fitting the model onto the data

    data = predictions(model, x, y) # calling function to make predictions at the g

    plt.figure(figsize=(6, 6))
    plt.contourf(x, y, data, cmap='coolwarm', levels=100)
    plt.colorbar(label='KNN Prediction')
    plt.title(f'Weighted KNN Predictions (k={k})')
    plt.xlabel('w1')
    plt.ylabel('w2')
    plt.show()

# function to calculate weighted_knn for the grid input
def predictions(model, x, y):
    L = np.zeros((50, 50))
    for i in range(50):
        for j in range(50):
            L[i, j] = model.predict(np.array([[x[i, j], y[i, j]]]))[0]
    return L

for k in [1, 5, 25]:
    plot(k)
```


Weighted KNN Predictions (k=1)
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
w2
−1.00 −0.75 −0.50 −0.25 0.00 0.25 0.50 0.75 1.00
w1
2.25
1.80
1.35
0.90
0.45
0.00
−0.45
−0.90
−1.35
KNN Prediction


Weighted KNN Predictions (k=5)
w1
w2
KNN Prediction
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
2.4
2.0
1.6
1.2
0.8
0.4
0.0
−0.4
−0.8
−1.2


Weighted KNN Predictions (k=25)
w1
w2
KNN Prediction
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
2.16
1.80
1.44
1.08
0.72
0.36
0.00
−0.36
−0.72
−1.08

In [ ]:

In [39]:

```
# I wrote my own code for weighted knn regression as the sklearn library
# was not very easy to work with for custom distances and weights

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# function to compute distance between two points
def arc_length(x):
    r = 1
    dist = r*(abs(np.arctan2(x[1], x[0]) - np.arctan2(y_data, x_data)))
    return dist

# function to perform regression for a point
def regress(x, k):
    d = arc_length(x)
    indices = np.argpartition(d, k)[:k]

    prox = np.square(d[indices])
    weights = prox/np.sum(prox)  # calculating custom weights

    knn = np.sum(np.dot(v_data[indices], weights))/k  # regressing over two closest
    return knn

# predicting values at unknown points
def predictions(points, k):
    l, b = np.shape(points)
    L = np.zeros(l)
    for i in range(l):
            L[i] = regress(points[i], k)  # predicting the values at each query poi
    return L

# importing data from file
path = ('hw1_bonus.xlsx')
known = pd.read_excel(path, "Known")
unknown = pd.read_excel(path, "Unknown")
x_data = known['x'].values
y_data = known['y'].values
v_data = known['v'].values
x = unknown['x'].values
y = unknown['y'].values
points = np.array([x, y]).T

k = 2
data = predictions(points, k)  # getting output from knn regression

# plotting
plt.figure(figsize = (7, 5.8), dpi = 300)
t = np.linspace(0,np.pi*2,100)
plt.scatter(x_data, y_data, c = v_data, cmap = 'RdBu', s = 50, edgecolors = 'black'
plt.scatter(points[:,0], points[:,1], c = data, cmap = 'RdBu', s = 10, edgecolors =
plt.plot(np.cos(t), np.sin(t), linewidth = 0.5, zorder = 0)
plt.colorbar(label='KNN Prediction')
plt.title(f'Weighted KNN Predictions (k={k})')
```

```
plt.xlabel('x')
plt.ylabel('y')
plt.xticks(np.arange(-1, 1.5, 0.5))
plt.yticks(np.arange(-1, 1.5, 0.5))
plt.xlim([-1.25, 1.25])
plt.ylim([-1.25, 1.25])
plt.grid(zorder = 0)
plt.legend(["Known", "Unknown"])
plt.show()
```