Task 1 k-NearestNeighbours

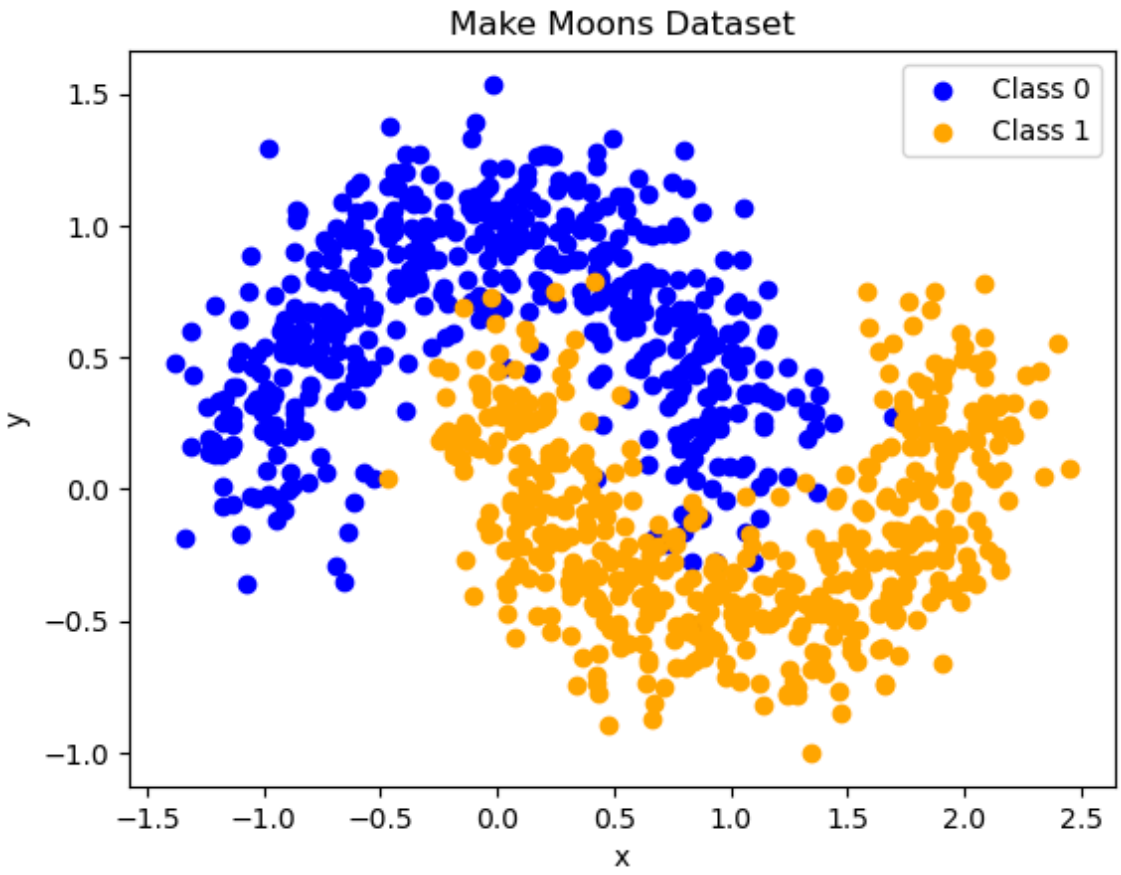
In [1]:
```
import numpy as np
from sklearn.datasets import make_moons
from sklearn.neighbors import KNeighborsClassifier
import matplotlib.pyplot as plt
```

Create dataset

In [2]:
```
N = 1000
N_train = int(N*0.9) # Use 90% for training
N_test = N - N_train # Rest for testing
x, y = make_moons(n_samples=N, noise=0.2,random_state=0)
# Split into train and test set
xtrain, ytrain = x[:N_train,...], y[:N_train,...]
xtest, ytest = x[N_train:,...], y[N_train:,...]
```

1. Visualize the dataset. You can use matplotlib or any other plotting library of your choice. (1P)

In [3]:
```
# Scatter plot for class 0
plt.scatter(x[y==0, 0], x[y==0, 1], c='blue', label='Class 0')
# Scatter plot for class 1
plt.scatter(x[y==1, 0], x[y==1, 1], c='orange', label='Class 1')

plt.title("Make Moons Dataset")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.show()
```



2. Implement a method kneighbours to return the indices and the distance of the k nearest neighbours from the training set for a given query point. Use the euclidean distance as distance metric. (2P)
3. Add a predict functionality to your KNN class which returns the predicted label for a given query point. (1P)

In [4]:
```
class KNN:
    def __init__(self, k):
        self.k = k
    def fit(self, x, y):
        # Fit routine
        self.x = x
        self.y = y
    def kneighbours(self, q):
        # Return nearest neighbour indices and distances
        # Pairwise squared distances that are summed up along each x's dimer
        distances = np.sqrt(np.sum((self.x - q) ** 2, axis=1))

        # Sort x according to their distances to q and take the first k elem
        indices = np.argsort(distances)[0:self.k]

        return indices, distances[indices]

    def predict(self, q):
        # Prediction function - Majority class vote
        indices, _ = self.kneighbours(q)

        # Count the occurance of each class in the k-neighbours and then tak
        pred = np.argmax(np.bincount(self.y[indices]))
        return pred
```

4. Fit your KNN model for k = 5 to the data. Repeat this step using the KNeighborsClassifier provided by sklearn and make sure both return the same predictions. (1P)

In [5]:
```
k = 5

# Selects a random element from the xtrain array
random_row = np.random.choice(xtrain.shape[0])
q = np.array(xtrain[random_row])

# Reshape q into an 2 dimensional array with 1 row of length len(q)
q = q.reshape(1, -1)

# My own implementation
knn = KNN(k)
knn.fit(xtrain, ytrain)
prediction = knn.predict(q)
print("My prediction:", prediction)

# Sklearn implementation
sk_knn = KNeighborsClassifier(k)
sk_knn.fit(xtrain, ytrain)

sk_prediction = sk_knn.predict(q)[0]
print("Sklearn prediction:", sk_prediction)
```

```
My prediction: 0
Sklearn prediction: 0
```

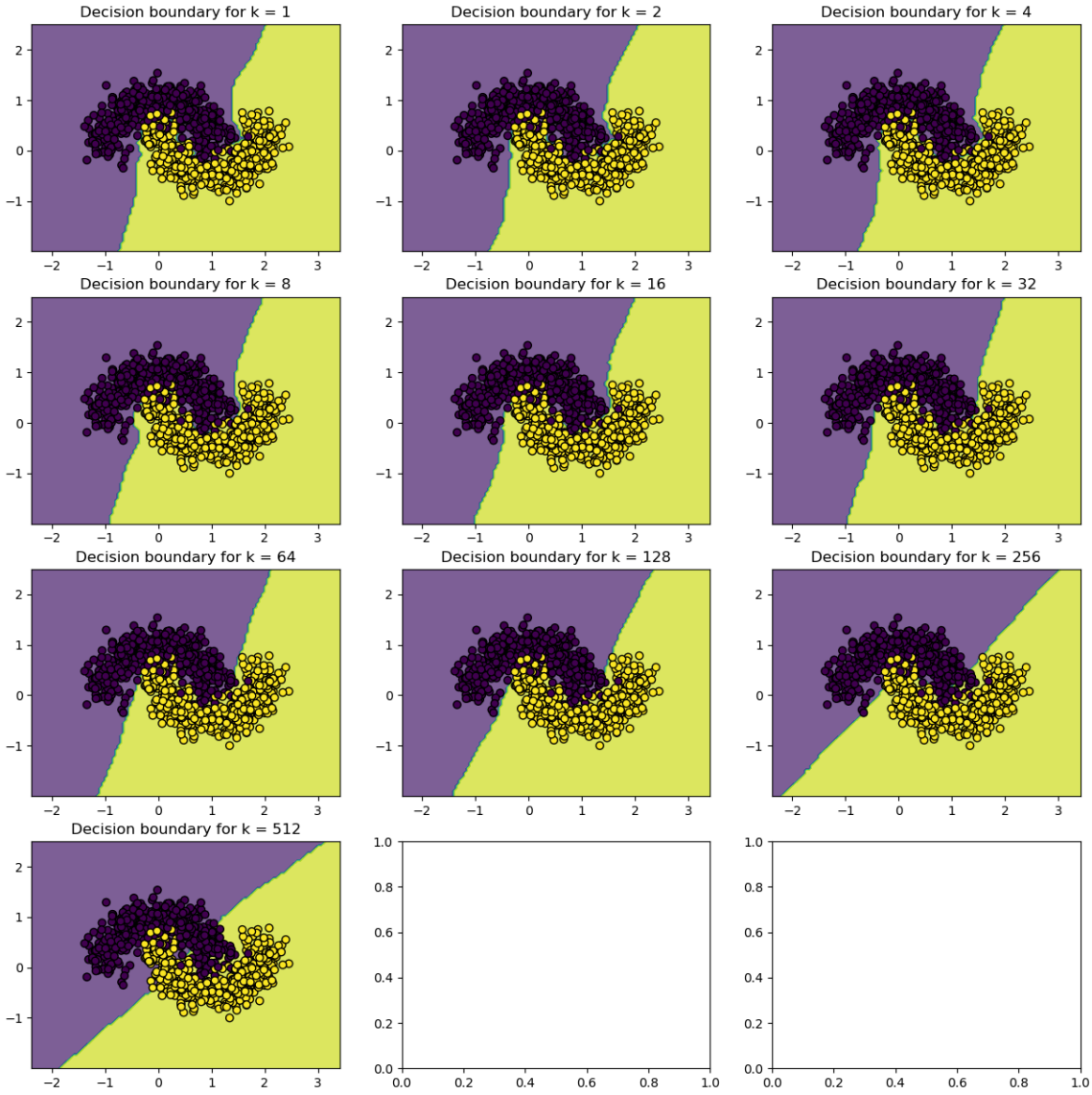
5. Run your KNN model with different values of k = 2i for i = 0, ..., 9. (1P)

In [6]:
```
k = 2

for exp in range(10):
  iter_k = k ** exp

  knn = KNN(iter_k)
  knn.fit(xtrain, ytrain)
  prediction = knn.predict(q)
  print(f'k: {iter_k}, prediction: {prediction}')
```

```
k: 1, prediction: 0
k: 2, prediction: 0
k: 4, prediction: 0
k: 8, prediction: 0
k: 16, prediction: 0
k: 32, prediction: 0
k: 64, prediction: 0
k: 128, prediction: 0
k: 256, prediction: 0
k: 512, prediction: 0
```

6. Plot the decision boundary for each k. (1P) Hint: Evaluate the classifier on a grid within a box. Use around 100 points in each direction and generate the grid via np.meshgrid. Visualize the area with a contour plot (contourf using matplotlib).

In [7]:
```
k = 2

# Create a grid of points
x_min, x_max = xtrain[:, 0].min() - 1, xtrain[:, 0].max() + 1 # feature 1
y_min, y_max = xtrain[:, 1].min() - 1, xtrain[:, 1].max() + 1 # feature 2
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.05),
                     np.arange(y_min, y_max, 0.05))

fig, axs = plt.subplots(4, 3, figsize=(15, 15))

for exp in range(10):
    iter_k = k ** exp
    knn = KNN(iter_k)
    knn.fit(xtrain, ytrain)

    z = np.empty_like(xx)

    # Make the knn prediction for every point in the grid
    for i in range(xx.shape[0]):
        for j in range(xx.shape[1]):
            q = np.array([xx[i, j], yy[i, j]]).reshape(1, -1)

            z[i, j] = knn.predict(q)

    # Calculate the row and column indices
    x_axis = exp // 3
    y_axis = exp % 3

    # Plot the decision boundary along with the data points for every k
    axs[x_axis][y_axis].contourf(xx, yy, z, alpha=0.7)
    axs[x_axis][y_axis].scatter(xtrain[:, 0], xtrain[:, 1], c=ytrain, edgecc
    axs[x_axis][y_axis].set_title(f'Decision boundary for k = {iter_k}')
plt.show()
```



7. How does the decision boundary change with k? What would happen if k is equal to the number of train samples? (1P)

The higher the k, the smoother the decision boundary. This is because more points are taken into account and thus noise values have less impact. If k = len(xtrain), then all data points will be taken into account and thus the class, to which more data points belong to will always be chosen. -> There won't be a decision boundary anymore

8. Report class probabilities p(c) on the train set. Further plot p(xn) = k NV ⋆ by estimating V ⋆ as the area of the smallest circle needed to include k nearest neighbours for query point xn. Use k = 2, 4, 8, 16, 32. Repeat the same plots for p(xn|c). (2P)

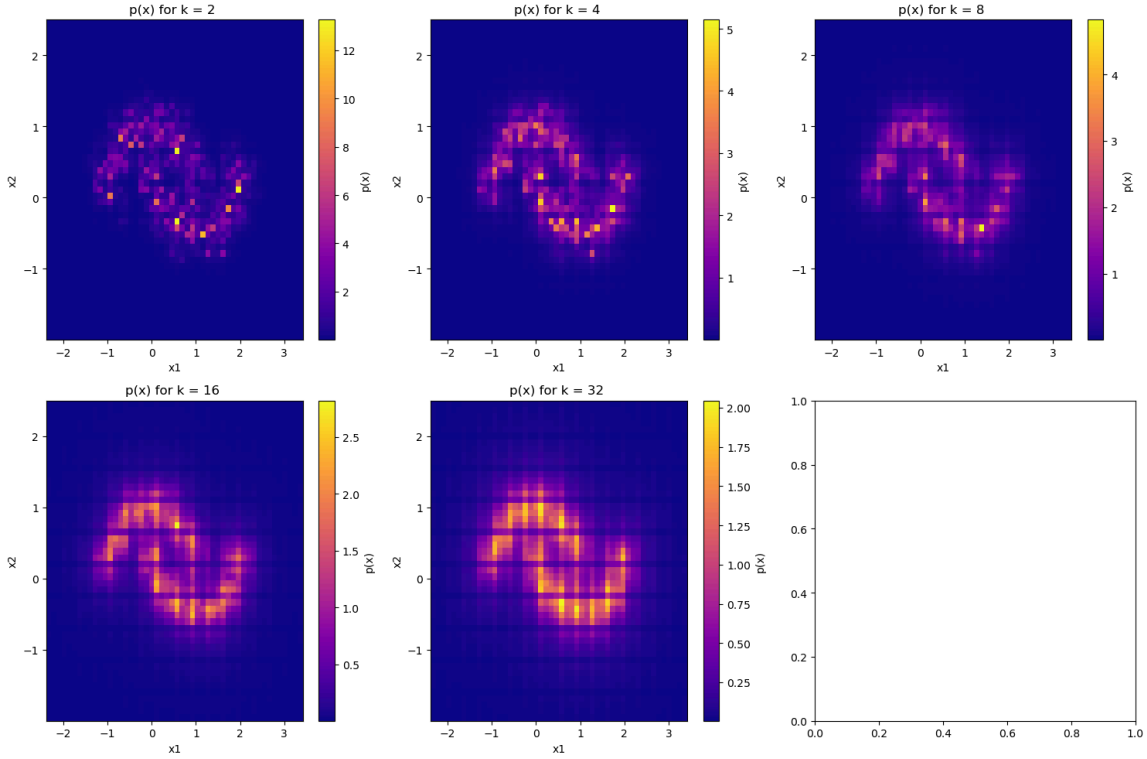
In [8]:
```
p0 = round(sum(ytrain == 0) / len(ytrain), 4)
print("p(0) =", p0)

p1 = round(sum(ytrain == 1) / len(ytrain), 4)
print("p(1) =", p1)
```

```
p(0) = 0.5033
p(1) = 0.4967
```

In [9]:
```
def euclid(a, b): np.sqrt(sum((a-b) ** 2))
```

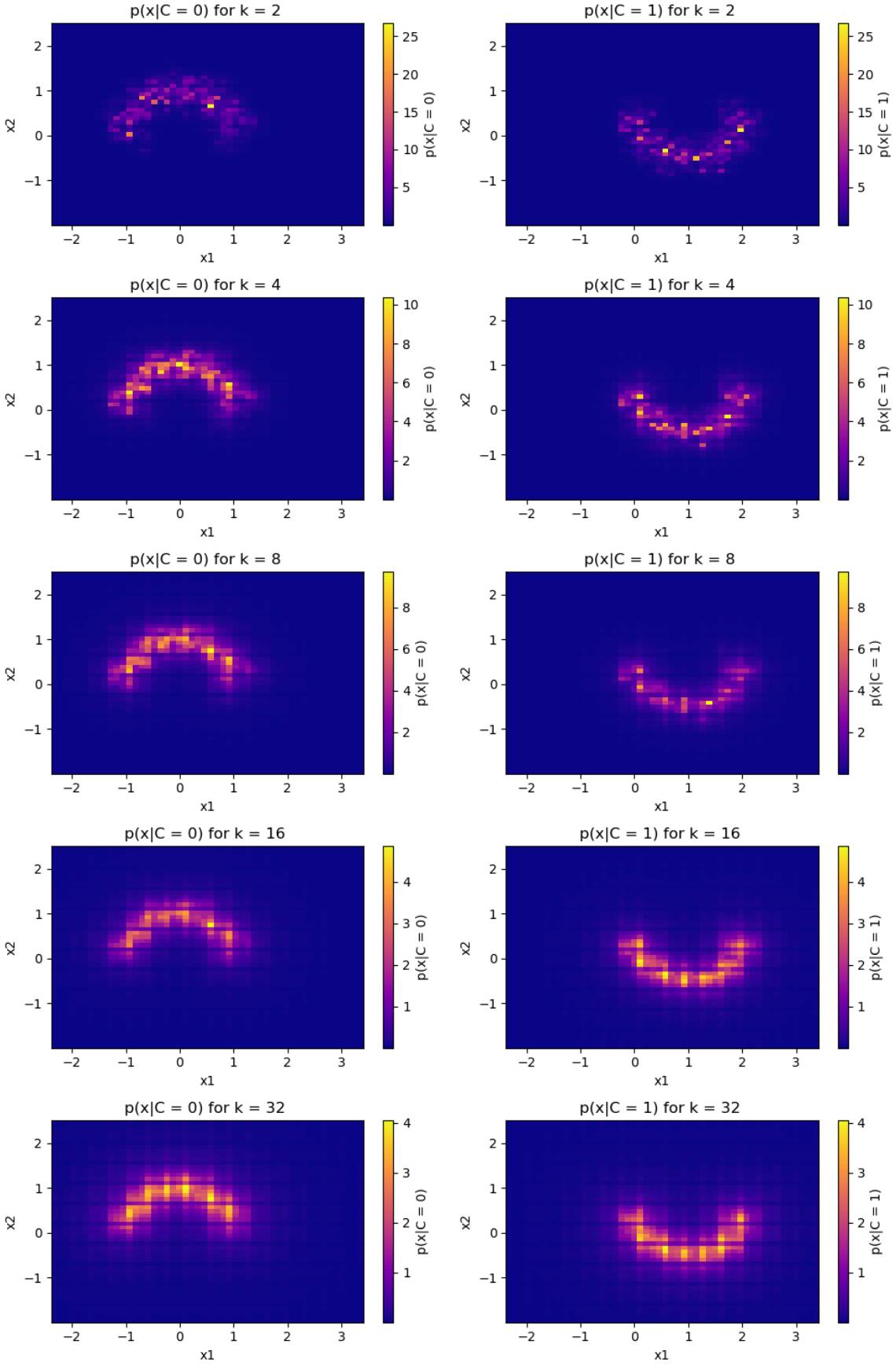
In [10]:
```
def px(x, k):
  knn = KNN(k)
  knn.fit(xtrain, ytrain)

  _, distances = knn.kneighbours(x)

  radius = distances[-1]
  n = xtrain.shape[0]
  v = np.pi * (radius ** 2)
  return k / (n * v)
```

In [11]:
```
fig, axs = plt.subplots(2, 3, figsize=(15, 10))

# Create a grid of points
x_min, x_max = xtrain[:, 0].min() - 1, xtrain[:, 0].max() + 1 # feature 1
y_min, y_max = xtrain[:, 1].min() - 1, xtrain[:, 1].max() + 1 # feature 2
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.05),
                     np.arange(y_min, y_max, 0.05))

# Define the values of k
k_values = [2, 4, 8, 16, 32]

for k in k_values:
    # Initialize an empty list to store the values of p(x)
    px_values = np.zeros(xx.shape)

    for i in range(xx.shape[0]):
        for j in range(xx.shape[1]):
            q = np.array([xx[i, j], yy[i, j]]).reshape(1, -1)
            px_values[i, j] = px(q, k)

    x_axis = int((np.log2(k) - 1) // 3)
    y_axis = int((np.log2(k) - 1) % 3)

    # Plot the data in a histogram
    h = axs[x_axis][y_axis].hist2d(xx.ravel(), yy.ravel(), weights=px_values
    axs[x_axis][y_axis].set_title(f"p(x) for k = {k} ")
    axs[x_axis][y_axis].set_xlabel("x1")
    axs[x_axis][y_axis].set_ylabel("x2")
    fig.colorbar(h[3], ax=axs[x_axis][y_axis], label='p(x)')

plt.tight_layout() # Avoid overlapping of subplots
plt.show()
```



In [12]:
```python
def pxc(x, k, c):
  knn = KNN(k)
  # Calculate dataset, where all points are of class c
  xtrain_c = xtrain[ytrain == c]
  ytrain_c = ytrain[ytrain == c]

  knn.fit(xtrain_c, ytrain_c)

  _, distances = knn.kneighbours(x)
  radius = distances[-1]
  n_c = xtrain_c.shape[0]
  v = np.pi * (radius ** 2)
  pxc = k / (n_c * v)
  #print(pxc)
  return pxc
```

In [13]:
```
fig, axs = plt.subplots(5, 2, figsize=(10, 15))

# Create a grid of points
x_min, x_max = xtrain[:, 0].min() - 1, xtrain[:, 0].max() + 1 # feature 1
y_min, y_max = xtrain[:, 1].min() - 1, xtrain[:, 1].max() + 1 # feature 2
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.05),
                     np.arange(y_min, y_max, 0.05))

# Define the values of k
k_values = [2, 4, 8, 16, 32]


for k in k_values:
    # Initialize an empty list to store the values of p(x)
    px0_values = np.zeros(xx.shape)
    px1_values = np.zeros(xx.shape)

    for i in range(xx.shape[0]):
        for j in range(xx.shape[1]):
            q = np.array([xx[i, j], yy[i, j]]).reshape(1, -1)
            px0_values[i][j] = pxc(q, k, 0) # Values for class 0
            px1_values[i][j] = pxc(q, k, 1) # Values for class 1

    x_axis = int((np.log2(k) - 1))
    y_axis = int((np.log2(k) - 1) % 3)

    # Plot the data in a histogram
    h = axs[x_axis][0].hist2d(xx.ravel(), yy.ravel(), weights=px0_values.rav
    axs[x_axis][0].set_title(f"p(x|C = 0) for k = {k} ")
    axs[x_axis][0].set_xlabel("x1")
    axs[x_axis][0].set_ylabel("x2")

    h = axs[x_axis][1].hist2d(xx.ravel(), yy.ravel(), weights=px1_values.rav
    axs[x_axis][1].set_title(f"p(x|C = 1) for k = {k} ")
    axs[x_axis][1].set_xlabel("x1")
    axs[x_axis][1].set_ylabel("x2")
    fig.colorbar(h[3], ax=axs[x_axis][0], label='p(x|C = 0)')
    fig.colorbar(h[3], ax=axs[x_axis][1], label='p(x|C = 1)')

plt.tight_layout() # Avoid overlapping of subplots
plt.show()
```


p(x|C = 0) for k = 2
p(x|C = 1) for k = 2
p(x|C = 0) for k = 4
p(x|C = 1) for k = 4
p(x|C = 0) for k = 8
p(x|C = 1) for k = 8
p(x|C = 0) for k = 16
p(x|C = 1) for k = 16
p(x|C = 0) for k = 32
p(x|C = 1) for k = 32
x1
x2
p(x|C = 0)
p(x|C = 1)

In [ ]:

Task 2 Linear Regression

In [5]:
```
import numpy as np
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
```

1. Complete the fit and predict routine in task2.py. (2P)

In [6]:
```
def make_data(noise=0.2, outlier=1):
    prng = np.random.RandomState(0)
    n = 500
    x0 = np.array([0, 0])[None, :] + noise * prng.randn(n, 2)
    y0 = np.ones(n)
    x1 = np.array([1, 1])[None, :] + noise * prng.randn(n, 2)
    y1 = -1 * np.ones(n)
    x = np.concatenate([x0, x1])
    y = np.concatenate([y0, y1]).astype(np.int32)
    xtrain, xtest, ytrain, ytest = train_test_split(
        x, y, test_size=0.1, shuffle=True, random_state=0
    )
    xplot, yplot = xtrain, ytrain
    outlier = outlier * np.array([1, 1.75])[None, :]
    youtlier = np.array([-1])
    xtrain = np.concatenate([xtrain, outlier])
    ytrain = np.concatenate([ytrain, youtlier])
    return xtrain, xtest, ytrain, ytest, xplot, yplot


class LinearLeastSquares:
    def fit(self, x, y):

        # Add a column of ones for the bias term
        x = np.c_[np.ones(x.shape[0]), x]

        # @ is matrix multiplication in numpy
        self.params = np.linalg.inv(np.transpose(x) @ x) @ np.transpose(x) @
    def predict(self, x):
        # Predict routine
        # Add a column of ones for the bias term
        x = np.c_[np.ones(x.shape[0]), x]

        prediction = x @ self.params
        return prediction
```

The file task2.py contains a synthetic dataset for which one can control the number of outliers n during generation. Generate datasets for n = 2i , i = 0, ..., 4 and solve the following tasks:

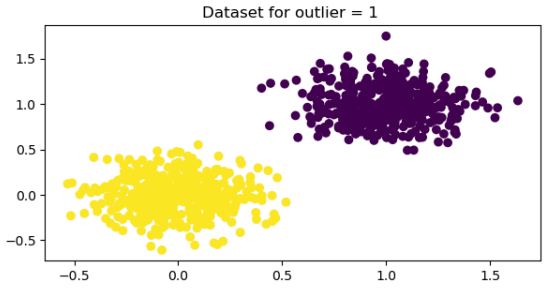
2. Visualize the dataset (0.5P)
3. Fit your LLS model to the data, report accuracy on the test set and visualize the decision boundary of the classifier. (1P)
4. How is the fit affected by the outlier? Give a short explanation. (0.5P)
   - outlier = 1: The classifier still classifies perfectly

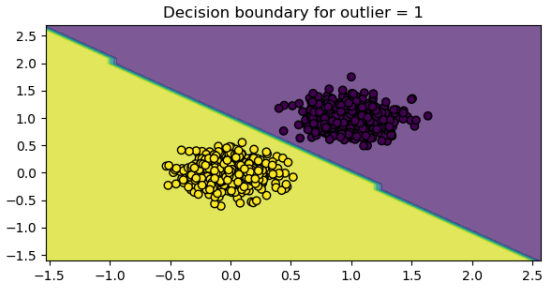
- outlier = 2: The classifier still classifies perfectly
- outlier = 4: The classifier still classifies perfectly
- outlier = 8: The classifier is vissibly affected by the far away outlier. The classification of the data points suffers, because the LLS tries to minimize the sum of the squared differences, which shifts the decision boundary in direction of the outlier.

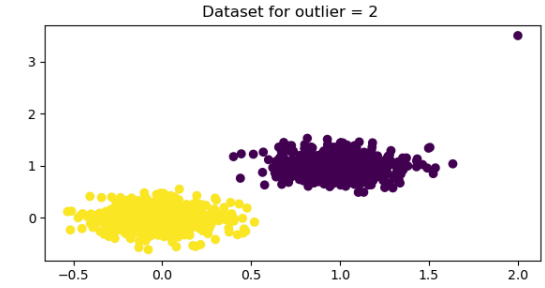
4.1 4.2. 4.3. outlier = 4: The classifier still classifies perfectly d. outlier = 8:

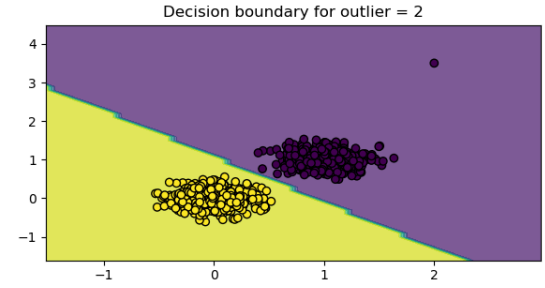
In [7]:
```
fig, axs = plt.subplots(4, 2, figsize=(15, 15))
lls = LinearLeastSquares()

for exp in range(4):
    xtrain, xtest, ytrain, ytest, xplot, yplot = make_data(outlier=2 ** exp)

    # 2. Visualize the dataset
    axs[exp][0].scatter(xtrain[:, 0], xtrain[:, 1], c=ytrain)
    axs[exp][0].set_title(f"Dataset for outlier = {2 ** exp} ")

    # Train the model
    lls.fit(xtrain, ytrain)

    # 3. Calculate the accuracy of the model on the test set
    ypred = lls.predict(xtest)
    ypred = np.where(ypred >= 0, 1, -1)  # Apply threshold
    accuracy = np.mean(ypred == ytest)
    print(f"Accuracy: {accuracy * 100}%")


    lls.predict(np.array([[10, 10]]))


    # Create a grid of points
    x_min, x_max = xtrain[:, 0].min() - 1, xtrain[:, 0].max() + 1
    y_min, y_max = xtrain[:, 1].min() - 1, xtrain[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                         np.arange(y_min, y_max, 0.1))

    z = np.empty_like(xx)

    # Make the lls prediction for every point in the grid
    for i in range(xx.shape[0]):
        for j in range(xx.shape[1]):
            q = np.array([xx[i, j], yy[i, j]]).reshape(1, -1)
            z[i, j] = lls.predict(q)[0]
            # Set a threshold at 0, so that we have a classification with 2
            z[i, j] = np.where(z[i, j] >= 0, 1, -1)

    # Plot the decision boundary along with the data points for every k
    axs[exp][1].contourf(xx, yy, z, alpha=0.7)
    axs[exp][1].scatter(xtrain[:, 0], xtrain[:, 1], c=ytrain, edgecolor='k')
    axs[exp][1].set_title(f'Decision boundary for outlier = {2 ** exp}')
plt.show()
```

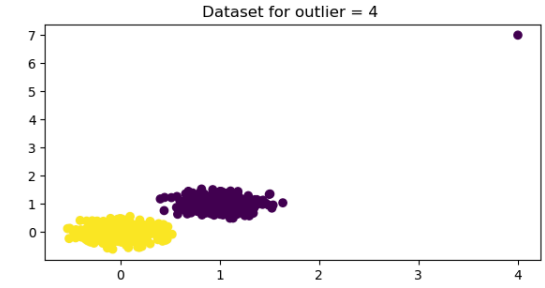
```
Accuracy: 100.0%
Accuracy: 100.0%
Accuracy: 100.0%
Accuracy: 98.0%
```

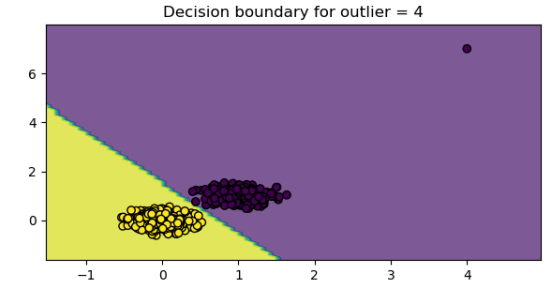

Dataset for outlier = 1

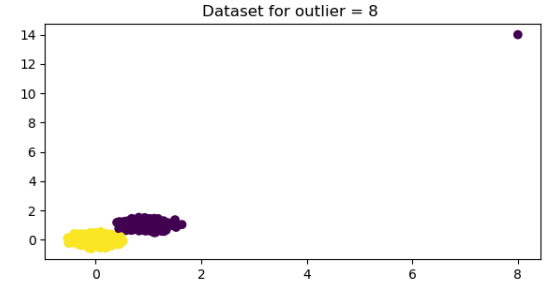

Decision boundary for outlier = 1

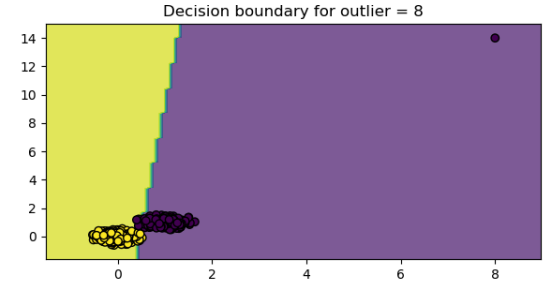

Dataset for outlier = 2


Decision boundary for outlier = 2


Dataset for outlier = 4


Decision boundary for outlier = 4


Dataset for outlier = 8


Decision boundary for outlier = 8

In [ ]:

In [ ]:

Task 3 Softmax Regression & Optimization

In [5]:
```
import numpy as np
from tqdm import tqdm
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_digits
import matplotlib.pyplot as plt
```

1. Derive a formula for the gradient ∂L ∂θ(c) j . To do so, just derive the solution for a single example Li and apply the chain rule of calculus ∂L ∂θ = ∂L ∂p ∂p ∂g ∂g ∂θ with g = θ0 + PD j=1 θjxj . For this task it is enough if you report derivatives of each chain rule component alone to get full points. (3P)

Report each chain rule component derivative for a single L

1. ∂L/∂p

$$\frac{\partial L_i}{\partial p} = \sum_{c=1}^{C} \frac{1}{p(c|x)} * \mathbb{1}[y_i = c]$$

2. ∂p/∂g

$$\frac{\partial p(g_i)}{\partial g_j} = \begin{array}{ll} p(g_i) * (1 - p(g_i)) & if\ i = jp \\ -p(g_i) * p(g_j) & if\ i \neq j \end{array}$$

3. ∂g/∂θ

$$\frac{\partial g}{\partial \theta} = \begin{pmatrix} 1 \\ x_1 \\ x_2 \\ \dots \\ x_D \end{pmatrix}$$

load dataset

In [6]:
```
data = load_digits()
x, y = (data.images / 16.0).reshape(-1, 8 * 8), data.target
xtrain, xtest, ytrain, ytest = train_test_split(x, y, test_size=0.25, shuff
```

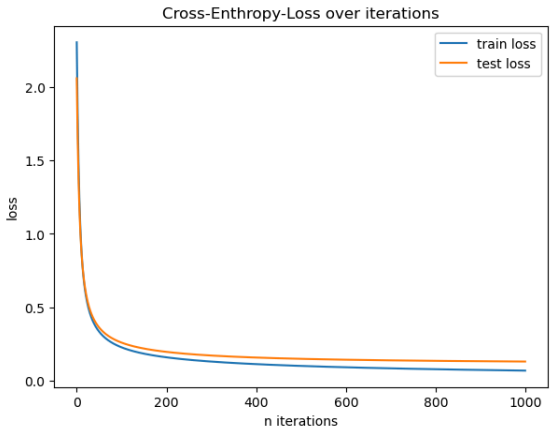
In [7]:
```
def softmax(logits):
    # Subtract the maximum element from each element to improve numerical st
    logits -= np.max(logits)
    # Calculate the softmax. The transpose is necessary, so that each row wi
    sm = (np.exp(logits).T / np.sum(np.exp(logits), axis=1)).T
    return sm
```

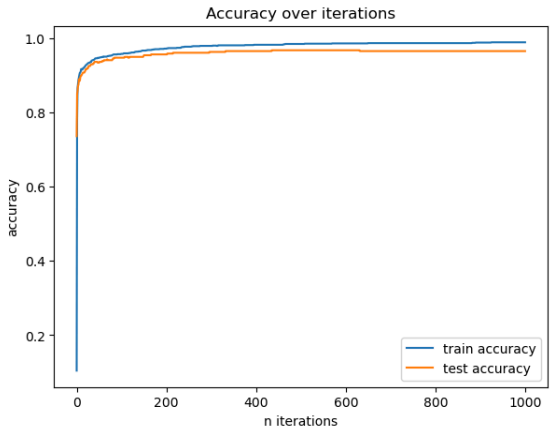
In [8]:
```
# Helper function to one hot encode class labels
def one_hot_encode(y, num_classes):
    return np.eye(num_classes)[y]
```

In [9]:
```
def cross_entropy_loss(y_true, y_pred):
    return -np.sum(y_true * np.log(y_pred)) / y_true.shape[0]
```

In [10]:
```
# Initialize parameters
weights = np.ones((1,10,x.shape[1])) #np.random.normal(0,1,size=(1,10,x.shap
bias = np.zeros((1,10))

lr = 0.001
num_iterations = 1000
n = x.shape[0] # Sample number

weights = np.reshape(weights, (64, 10)) # Reshape weights, so that they matc
ytrain_one_hot = one_hot_encode(ytrain, weights.shape[1]) # One hot encode y
ytest_one_hot = one_hot_encode(ytest, weights.shape[1]) # One hot encode ytr

train_losses = np.zeros(num_iterations)
train_accuracies = np.zeros(num_iterations)
test_losses = np.zeros(num_iterations)
test_accuracies = np.zeros(num_iterations)

for i in range(num_iterations):
    ### Optimization ###
    # Prediction
    score_train = xtrain @ weights + bias # Calculate dot product between da
    y_train_pred = softmax(score_train) # Predicted class probabilities

    # Calculate the gradients
    gradients = xtrain.T @ (y_train_pred - ytrain_one_hot)
    bias_gradient = np.mean(y_train_pred - ytrain_one_hot, axis=0)

    # Optimization with weight and bias updates
    weights = weights - lr * gradients
    bias = bias - lr * bias_gradient

    ### Visualization ###
    # Compute the cross-entropy loss
    train_losses[i] = cross_entropy_loss(ytrain_one_hot, y_train_pred)
    train_accuracies[i] = np.mean(np.argmax(y_train_pred, axis=1) == ytrain)
    #accuracies[i] = np.mean( y_pred == one_hot_encode(ytrain, y_pred.shape[

    # Compute loss and accuracy for test split
    score_test = xtest @ weights + bias
    y_test_pred = softmax(score_test)
    test_losses[i] = cross_entropy_loss(ytest_one_hot, y_test_pred)
    test_accuracies[i] = np.mean(np.argmax(y_test_pred, axis=1) == ytest)

# Plot accuracies and losses over n iterations
fig, axs = plt.subplots(1, 2, figsize=(15, 5))
axs[0].plot(range(num_iterations), train_losses, label="train loss")
axs[0].plot(range(num_iterations), test_losses, label="test loss")
axs[0].set_title("Cross-Enthropy-Loss over iterations")
axs[0].set_xlabel("n iterations")
axs[0].set_ylabel("loss")
axs[0].legend()

axs[1].plot(range(num_iterations), train_accuracies, label="train accuracy")
axs[1].plot(range(num_iterations), test_accuracies, label="test accuracy")
axs[1].set_title("Accuracy over iterations")
axs[1].set_xlabel("n iterations")
axs[1].set_ylabel("accuracy")
axs[1].legend()

plt.show()
```

```
print("Final test accuracy:", round(test_accuracies[-1], 4))
```






```
Final test accuracy: 0.9644
```