Principal Component Analysis

The goal of this question is to build a conceptual understanding of dimensionality reduction using PCA and implement it on a toy dataset. You'll only have to use numpy and matplotlib for this question.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
```

In [2]:
```
# (a) Load data (features)
def load_data():

    data = np.load("features.npy")
    data = (data - np.mean(data))/np.std(data)

    return data
```

In [3]:
```
# (b) Perform eigen decomposition and return eigen pairs in desecending orde
def eigendecomp(X):

    # Calculate covariance
    covariance = (X.T.dot(X))/X.shape[0]

    # Eigen Decomposition
    eig_vals, eig_vecs = np.linalg.eig(covariance)

    # Sort the eigen pairs in desecending order of eigen values
    index = np.argsort(eig_vals)
    sorted_eig_vals = eig_vals[index]
    sorted_eig_vals = sorted_eig_vals[::-1]
    sorted_eig_vecs = eig_vecs[:,index]
    sorted_eig_vecs = sorted_eig_vecs[:,::-1]

    return (sorted_eig_vals, sorted_eig_vecs)
```

In [4]:
```python
# (c) Evaluate using variance_explained as the metric
def eval(X):

    # Reduce the dimensionality to k
    ve_list = []
    print("variance explained when reducing the dimensionality to k")
    for k in range(X.shape[0]):
        print(f"k = {k+1}")
        ve = X[0:k+1].sum()/X.sum()
        print(f"variance explained = {ve}")
        ve_list.append(ve)
    print("\n")

    # Compute the variance explained for each eigenvalue
    print("variance explained for each eigenvalue")
    for i in range(len(ve_list)):
        print(f"The {i+1}th eigenvalue: {X[i]}")
        if i == 0:
            print(f"variance explained = {ve_list[i]}")
        else:
            print(f"variance explained = {ve_list[i]-ve_list[i-1]}")
```

In [5]:
```python
# (d) Visualize after projecting to 2-D space
def viz(X, sorted_eig_vecs):
    x_pca = np.dot(X, sorted_eig_vecs[:,0:2])
    y  = np.load("labels.npy", allow_pickle=True)
    labels = np.unique(y)
    plt.scatter(x_pca[(y == 0),0], x_pca[(y == 0), 1], c='red', label='Class
    plt.scatter(x_pca[(y == 1),0], x_pca[(y == 1), 1], c='blue', label='Clas
    plt.scatter(x_pca[(y == 2),0], x_pca[(y == 2), 1], c='green', label='Cla
    plt.legend()
    plt.title("2-D dimensionality reduction with PCA")
    plt.xlabel("First Principal Component")
    plt.ylabel("Second Principal Component")
```

In [6]:
```python
features = load_data()
print("Normalized Features shape: ", features.shape)
```

```
Normalized Features shape:  (150, 8)
```

In [7]:
```python
sorted_eig_vals, sorted_eig_vecs = eigendecomp(features)
```

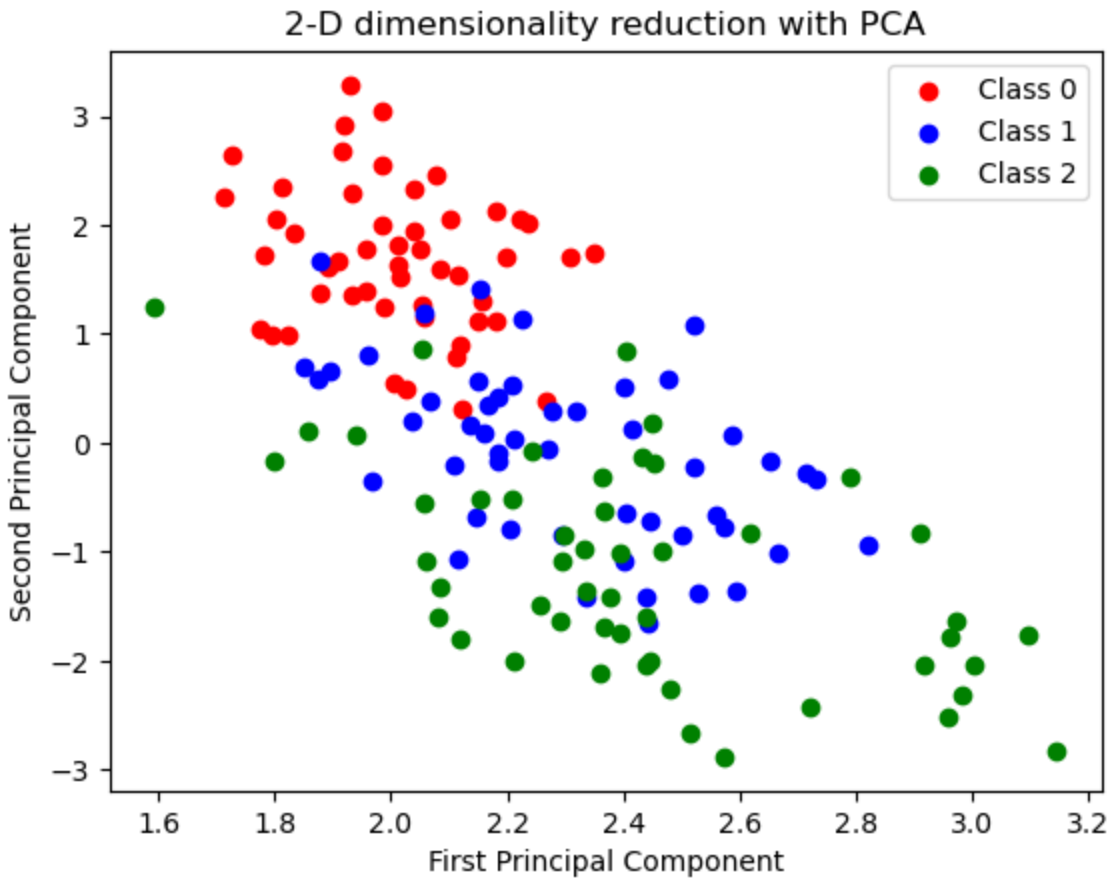
In [8]:
```python
print("Sorted EigenValues: ", sorted_eig_vals)
```

```
Sorted EigenValues:  [5.13151080e+00 2.09783761e+00 6.99123742e-01 5.842435
12e-02
 1.31034968e-02 1.72078267e-15 1.95426183e-16 1.38091055e-16]
```

In [9]:
```python
eval(sorted_eig_vals)
```

```
variance explained when reducing the dimensionality to k
k = 1
variance explained = 0.6414388505934037
k = 2
variance explained = 0.9036685512903241
k = 3
variance explained = 0.991059018995762
k = 4
variance explained = 0.9983620628941539
k = 5
variance explained = 0.9999999999999998
k = 6
variance explained = 1.0
k = 7
variance explained = 1.0
k = 8
variance explained = 1.0


variance explained for each eigenvalue
The 1th eigenvalue: 5.131510804747234
variance explained = 0.6414388505934037
The 2th eigenvalue: 2.0978376055753647
variance explained = 0.2622297006969204
The 3th eigenvalue: 0.6991237416435045
variance explained = 0.0873904677054379
The 4th eigenvalue: 0.05842435118713472
variance explained = 0.00730304389839187
The 5th eigenvalue: 0.013103496846766986
variance explained = 0.0016379371058459213
The 6th eigenvalue: 1.7207826731306866e-15
variance explained = 2.220446049250313e-16
The 7th eigenvalue: 1.954261828767496e-16
variance explained = 0.0
The 8th eigenvalue: 1.380910546060434e-16
variance explained = 0.0
```

In [10]:
```
viz(features, sorted_eig_vecs)
```

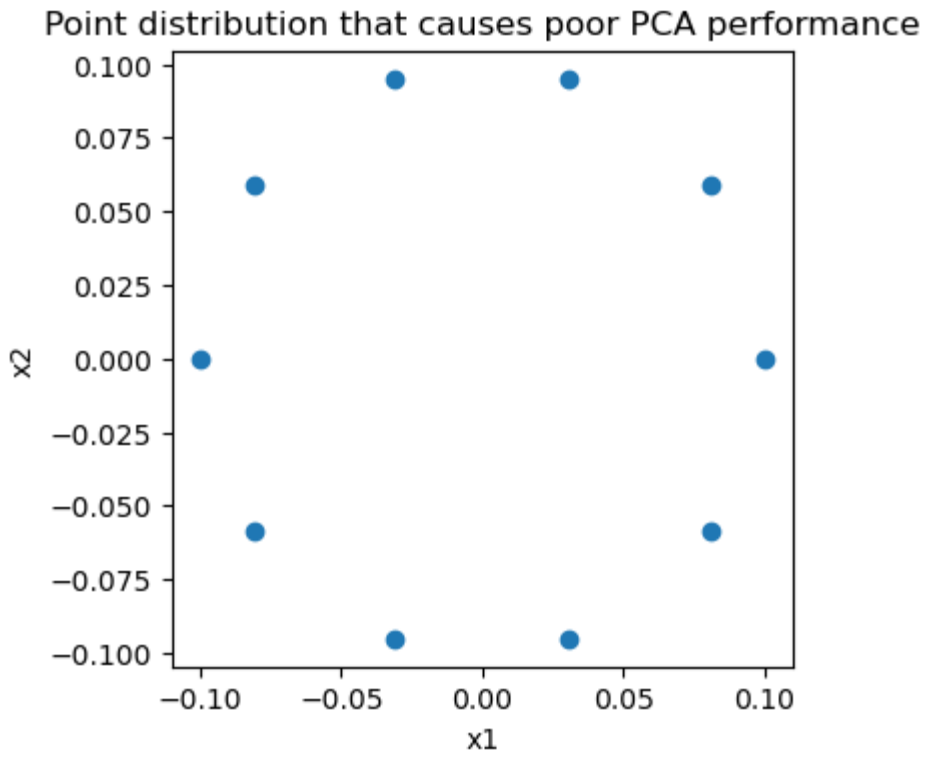


(e): Assume you have a dataset with the original dimensionality as 2 and you have to reduce it to 1. Provide a sample scatter plot of the original data (less than 10 datapoints) where PCA might produce misleading results. You can plot it by hand and then take a picture. In the next cell, switch to Markdown mode and use the command: ![title]()

In [11]:
```python
# Creating sample data
import numpy as np
import matplotlib.pyplot as plt

def circle_points(r, n):
    for r, n in zip(r, n):
        t = np.linspace(0, 2*np.pi, n, endpoint=False)
        x = r * np.cos(t)
        y = r * np.sin(t)
        circles = (np.c_[x, y])
    return circles
```

In [12]:
```python
r = [0.1]
n = [10]
circle = circle_points(r, n)
```

In [13]:
```python
plt.figure(figsize = (4,4))
plt.scatter(circle[:,0], circle[:,1])
plt.xlabel("x1")
plt.ylabel("x2")
plt.title("Point distribution that causes poor PCA performance")
plt.show()
```



Data that has a distribution like above, that is without a specific direction, is not well suited for PCA since it doesn't really have a principal component.

This problem was adapted from Professor Farimani's paper. If you are interested in learning more, you can read it here.

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

## Scikit learn tools
from sklearn import model_selection
from sklearn.model_selection import train_test_split
from sklearn.cluster import KMeans
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score
```

In [2]:
```
# (a)
# data preprocessing
acids = np.array(['TRP', 'ALA', 'TYR', 'PRO', 'HIS', 'THR', 'GLY', 'SER', 'C
         'MET', 'LEU', 'VAL', 'ASN', 'GLN', 'ARG', 'LYS'])
df = pd.read_csv("data.csv")
# Added a representative header name for the columns that didn't have a heac
current = ['current-' + c for c in acids]
# res_time = ['res_time-' + c for c in acids]

df.columns.values[::2] = current
# df.columns.values[1::2] = res_time
```

In [3]:
```
df.head()
```

Out[3]:

	current-TRP	TRP	current-ALA	ALA	current-TYR	TYR	current-PRO	PRO	current-HIS	HIS	...	curren V
0	1.46	2600	270.6	75.8	-1.23	1800	17.7	170.0	117.6	1090	...	7.46
1	21.60	2500	288.4	52.1	11.30	1700	21.3	91.0	105.8	1040	...	10.80
2	12.60	3200	284.8	72.5	14.60	1800	21.1	74.0	113.0	1070	...	12.78
3	6.31	2600	284.5	70.7	6.59	1800	23.8	130.0	117.2	1040	...	17.93
4	-3.39	3300	283.5	41.7	4.60	1900	15.7	72.0	111.1	1030	...	14.07

5 rows × 40 columns

In [4]:
```
# This rearranges the dataframe into 3 columns: label, Current, Residence ti

df2 = pd.melt(df, value_vars = acids, value_name = 'res_time', var_name = 'l
df2['current'] = pd.melt(df, value_vars = current)['value']
cols = ['label', 'current', 'res_time']
df2 = df2[cols]
data = df2.to_numpy()
X = data[:,1:]
y = data[:,0]

le = LabelEncoder()
le.fit(y)
y = le.transform(y)

print("Shape of x is: ", X.shape)
print("Shape of y is: ", y.shape)
print(f"Range of y is ({y.min()}, {y.max()})")
```

```
Shape of x is:  (2000, 2)
Shape of y is:  (2000,)
Range of y is (0, 19)
```

In [5]:
```
df2.head()
```

Out[5]:

	label	current	res_time
0	TRP	1.46	2600.0
1	TRP	21.60	2500.0
2	TRP	12.60	3200.0
3	TRP	6.31	2600.0
4	TRP	-3.39	3300.0

In [6]:
```
# Split the data set into 70% train and 30% test.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30)
```

In [7]:
```
# (b)
# k-means with 20 clusters
kmeans = KMeans(n_clusters=20).fit(X_train)
```

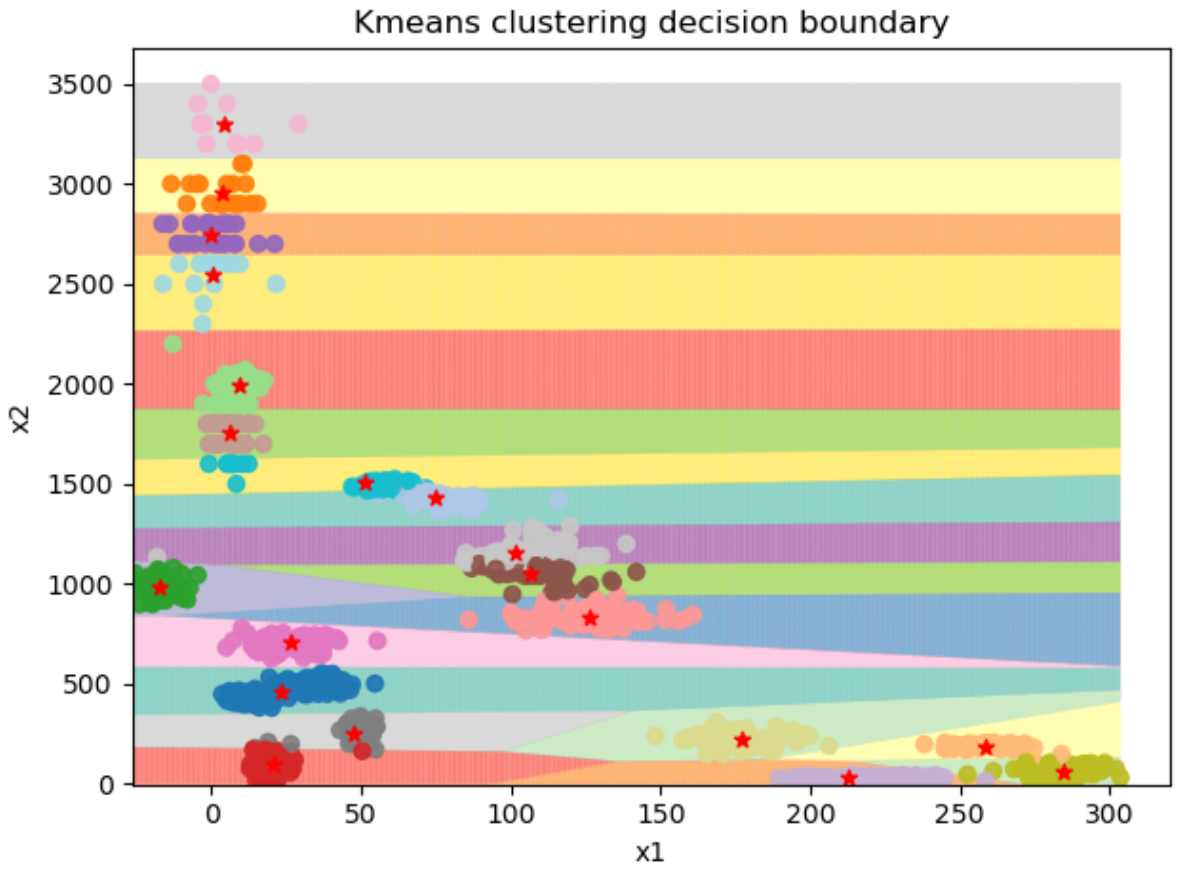
In [8]:
```
kmeans.cluster_centers_
```

Out[8]:
```
array([[ 2.32689201e+01,  4.62042014e+02],
       [ 7.49209867e+01,  1.43182133e+03],
       [ 3.71189474e+00,  2.95789474e+03],
       [ 2.58707848e+02,  1.89111392e+02],
       [-1.75601125e+01,  9.86037778e+02],
       [ 9.06783708e+00,  1.99312360e+03],
       [ 2.04933433e+01,  1.00874478e+02],
       [ 1.26219650e+02,  8.32972500e+02],
       [-5.82884615e-02,  2.75000000e+03],
       [ 2.12772803e+02,  2.83437273e+01],
       [ 1.06704018e+02,  1.05092500e+03],
       [ 6.12152500e+00,  1.76000000e+03],
       [ 2.64125147e+01,  7.06265147e+02],
       [ 4.45836364e+00,  3.30000000e+03],
       [ 4.76701585e+01,  2.50208659e+02],
       [ 1.01642829e+02,  1.15245526e+03],
       [ 2.84362319e+02,  6.41971014e+01],
       [ 1.76991176e+02,  2.22500000e+02],
       [ 5.11267213e+01,  1.50787541e+03],
       [ 3.78187500e-01,  2.54375000e+03]])
```

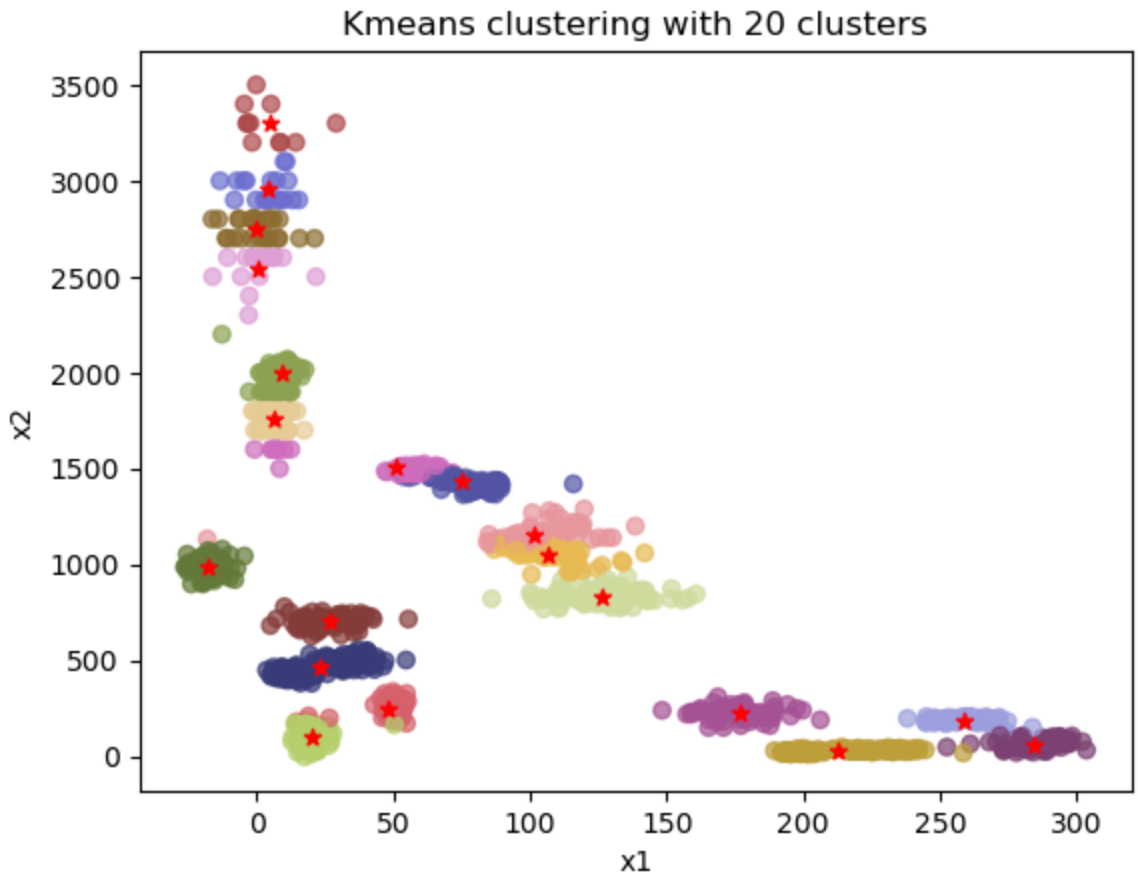
In [9]:
```
# Generate the meshgrid and plot the data with decision boundary.
x1, x2 = np.meshgrid(np.arange(X_train[:,0].min(), X_train[:,0].max()),
                     np.arange(X_train[:,1].min(), X_train[:,1].max()))
x1_flat = x1.flatten()
x2_flat = x2.flatten()
x1_flat_reshaped = x1_flat.reshape(x1_flat.shape[0],1)
x2_flat_reshaped = x2_flat.reshape(x1_flat.shape[0],1)
X_mesh = np.hstack((x1_flat_reshaped,x2_flat_reshaped))
```

In [10]:
```
mesh_labels = kmeans.predict(X_mesh)
mesh_labels = mesh_labels.reshape(x1.shape)
```

In [11]:
```
plt.scatter(X_train[:,0], X_train[:,1], c=kmeans.labels_, alpha=0.9, cmap='t
plt.pcolormesh(x1, x2, mesh_labels, cmap="Set3", alpha=0.03, shading="gourau
plt.scatter(kmeans.cluster_centers_[:,0], kmeans.cluster_centers_[:,1], c='r
plt.title("Kmeans clustering decision boundary")
plt.xlabel("x1")
plt.ylabel("x2")
plt.tight_layout()
```



In [12]:
```
# plot the data colored by the cluster labels obtained through kmeans.
plt.scatter(X_train[:,0], X_train[:,1], c=kmeans.labels_, alpha=0.7, cmap='t
plt.scatter(kmeans.cluster_centers_[:,0], kmeans.cluster_centers_[:,1], c='r
plt.title("Kmeans clustering with 20 clusters")
plt.xlabel("x1")
plt.ylabel("x2")
plt.show()
```

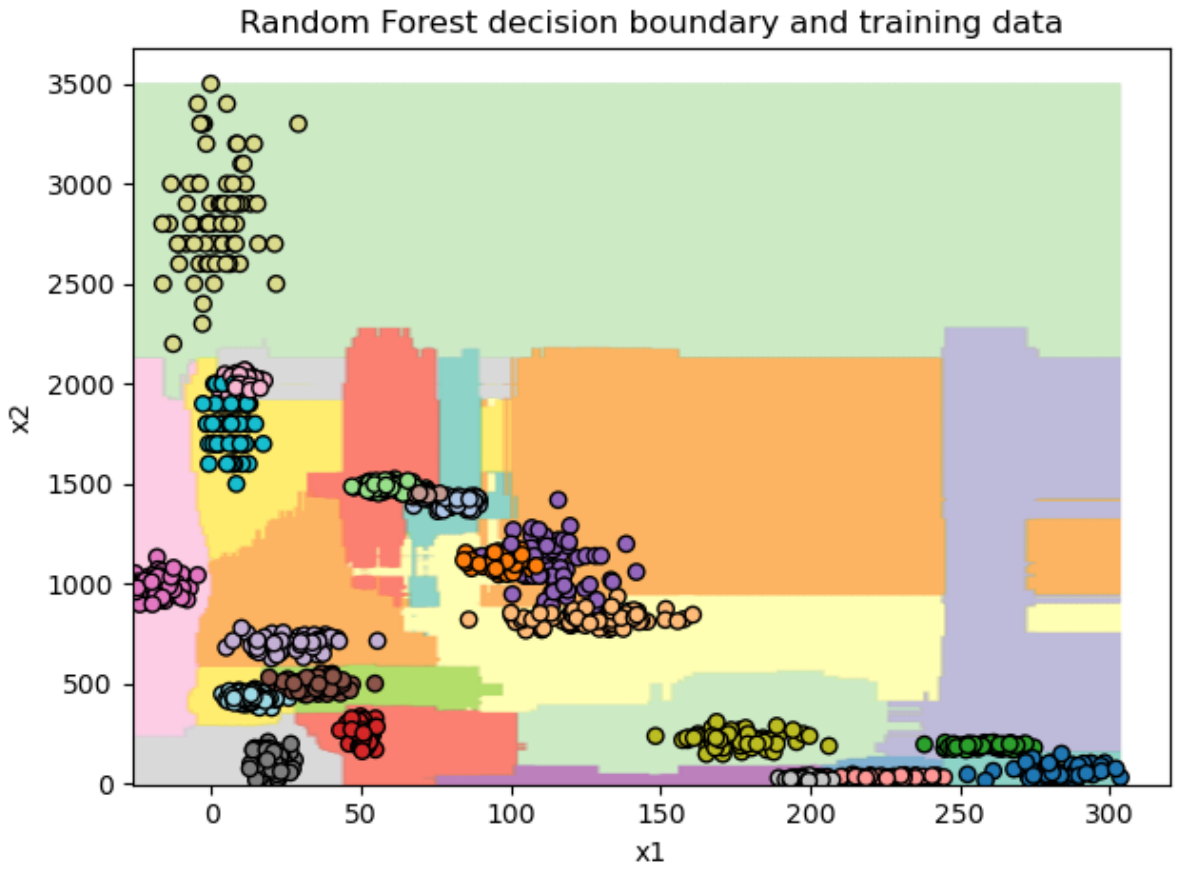


In [13]:
```
# (c)
# random forest with default estimators
rf = RandomForestClassifier()
rf.fit(X_train, y_train)
```

Out[13]:
```
▾ RandomForestClassifier

RandomForestClassifier()
```

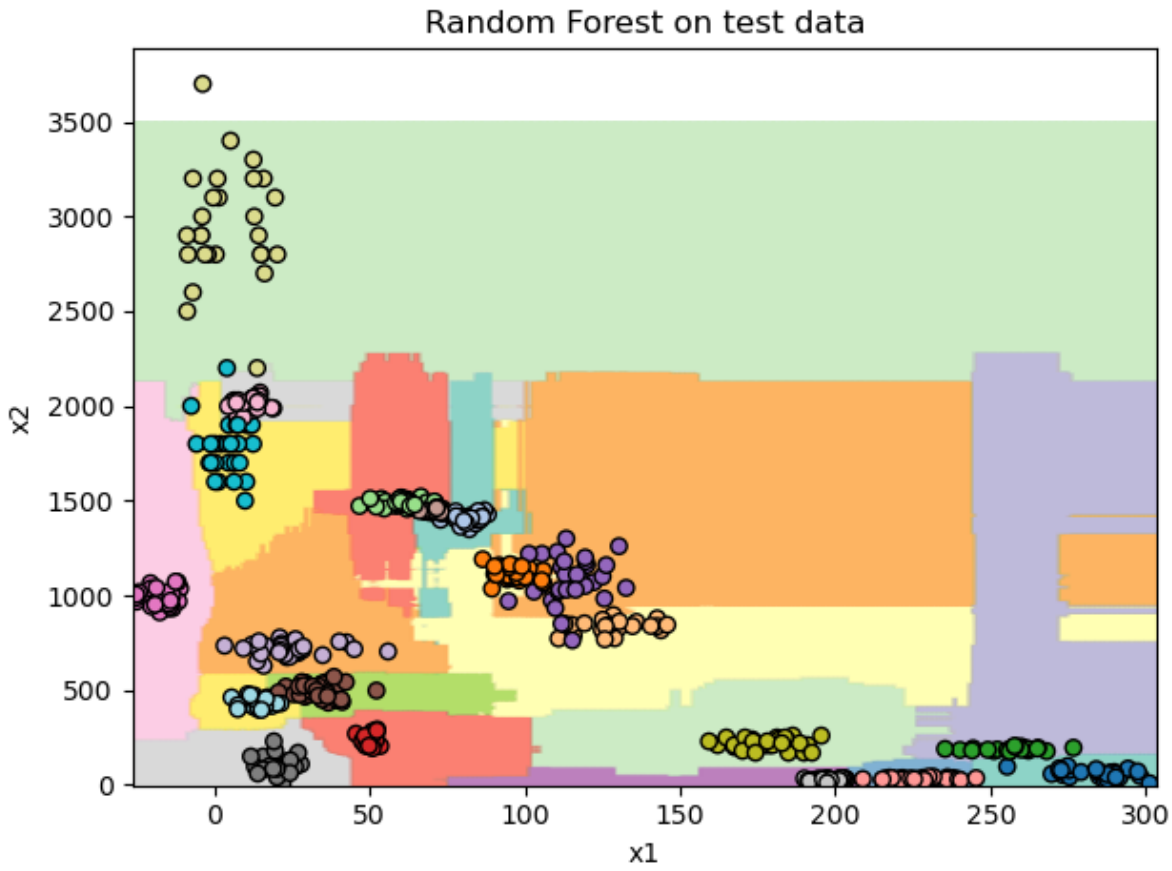
In [14]:
```
rf_mesh_labels = rf.predict(X_mesh)
rf_mesh_labels = rf_mesh_labels.reshape(x1.shape)
```

In [15]:
```
# Plot the random forest decision boundary and training data
plt.scatter(X_train[:,0], X_train[:,1], c=y_train, cmap='tab20', edgecolors=
plt.pcolormesh(x1, x2, rf_mesh_labels, cmap="Set3", alpha=0.1, shading="gour
plt.title("Random Forest decision boundary and training data")
plt.xlabel("x1")
plt.ylabel("x2")
plt.tight_layout()
```

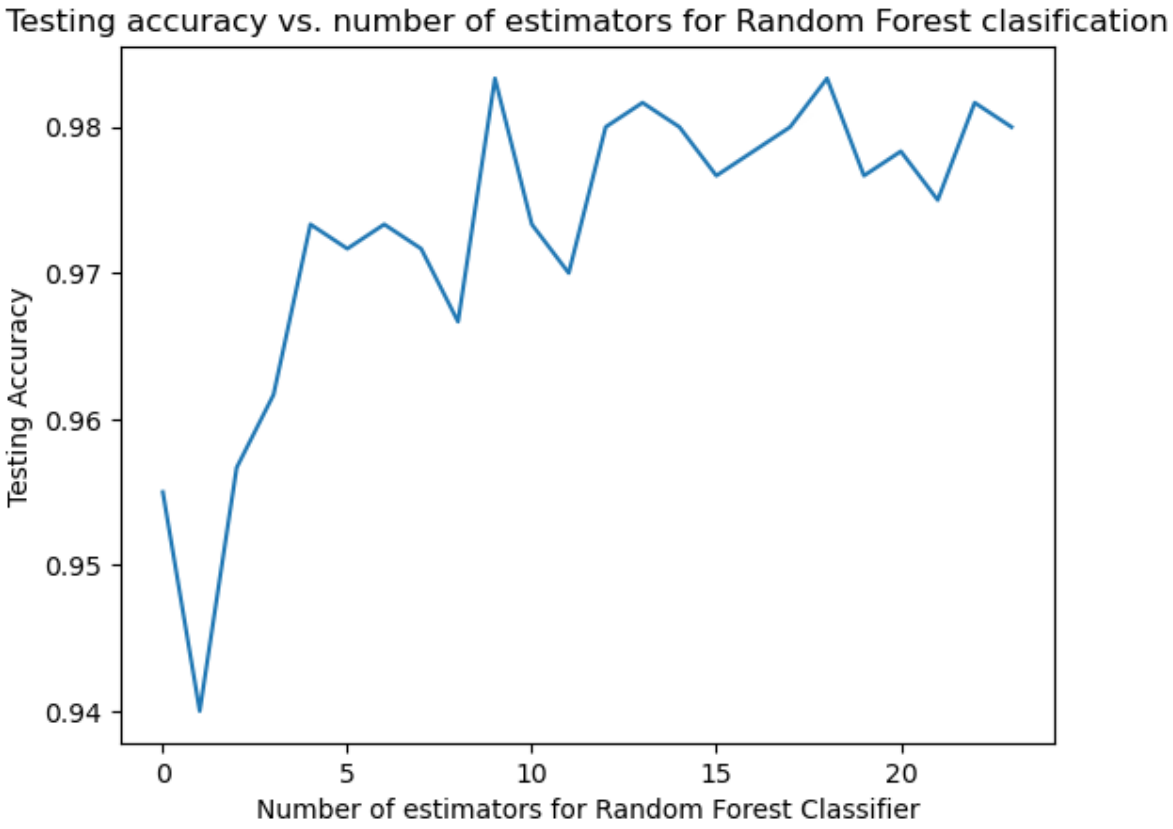


In [16]:
```
# Plot random forest prediction on the test data and print accuracy.

plt.scatter(X_test[:,0], X_test[:,1], c=y_test, cmap='tab20', edgecolors='bl
plt.pcolormesh(x1, x2, rf_mesh_labels, cmap="Set3", alpha=0.1, shading="gour
plt.title("Random Forest on test data")
plt.xlabel("x1")
plt.ylabel("x2")
plt.tight_layout()
print( f"Accuracy on test data: {accuracy_score(rf.predict(X_test), y_test)*
```

```
Accuracy on test data: 98.16666666666667 %
```



In [17]:
```
# Get accuracy of random forest for estimators 1 to 25, and plot the accurac
accuracy =[]
for k in range(1, 25):
    rf = RandomForestClassifier(n_estimators=k)
    rf.fit(X_train, y_train)
    y_pred = rf.predict(X_test)
    accuracy.append(accuracy_score(y_test, y_pred))

plt.plot(accuracy)
plt.xlabel('Number of estimators for Random Forest Classifier')
plt.ylabel('Testing Accuracy')
plt.title("Testing accuracy vs. number of estimators for Random Forest clasi
plt.show()
```



In [18]:
```
# (d)
# Analysis
```

Looking at the results of the algorithms, they actually don't seem too different. This is interesting considering k-means is unsupervised. The most noticeable difference to me is in the decision boundaries, and this is where k-means may be underfitting, because the boundaries seem to go straight in the vertical axis, except in the left side of the plot where there are more clusters close together and the decision boundaries get squeezed. It also seems to extend some of the clusters to points that are very far away and clearly should belong to a different category. This si definitely an effect of being unsupervised, as the algorithm has no way to know what label each data point has. In comparison, random forest shows decision boundaries that group data with the same true label together much nicer, although there does seem to be some overfitting with some boundaries seeming to have high variance.

Running the code multiple times with varying random initial values, these initial values seem to have a noticeable effect on both the algorithms, especially k-means. Since k-means is unsupervised it does seem to depend slightly on the initial centroid coordinates. The random forest accuracy score varied a little when running a few times, but it was fairly small (1-2 percent difference).

The plot of accuracy vs number of estimators for random forest gives useful information about the optimal number of estimators, since it looks like the accuracy converges at a certain point and adding more estimators seems to have less and less effect on it. Looking at my plot, 12-15 estimators may be a good number to lower the computing load.

Note for question3

- Please follow the template to complete q3
- You may create new cells to report your results and observations

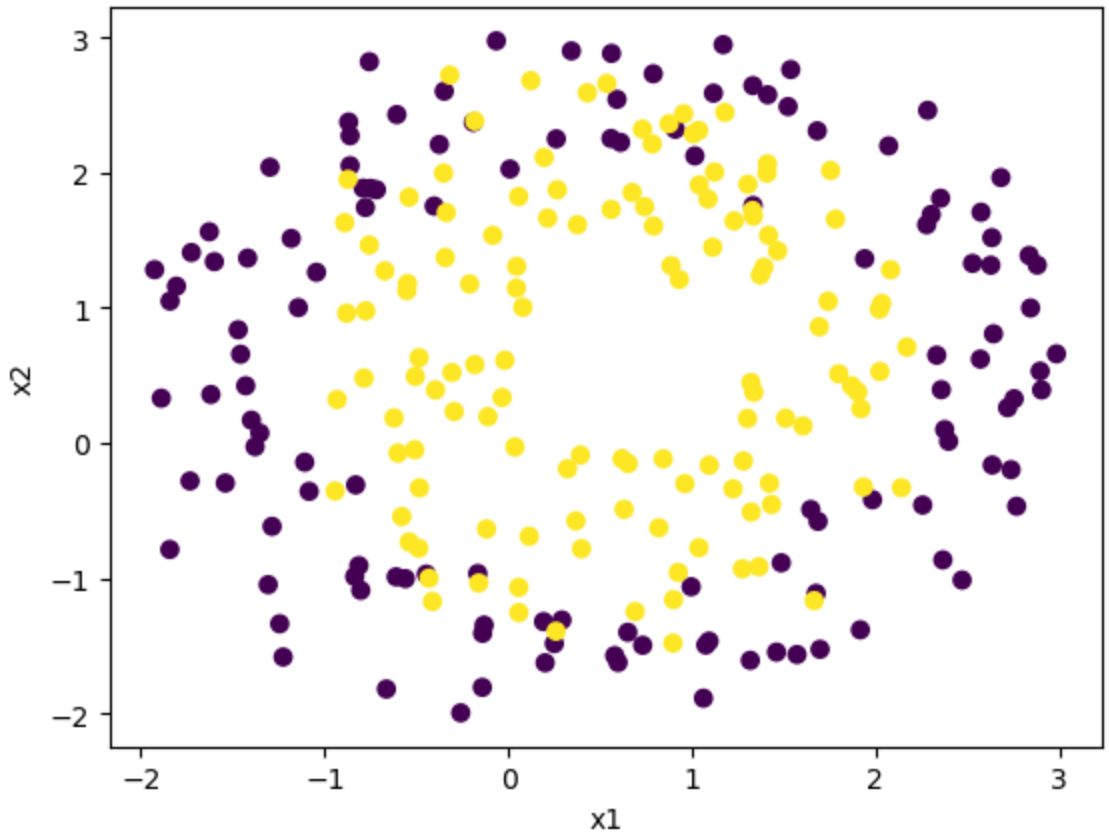
In [1]:
```
# Import libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

P1. Load data and plot

TODO

- load q3_data.csv
- plot the points of different labels with different color

In [2]:
```
# Load dataset
df = pd.read_csv('q3_data.csv', header=None)
x = df.iloc[:,:2].to_numpy()
y = df.iloc[:,-1].to_numpy()

# Plot points
plt.scatter(x[:,0], x[:,1], c=y)
plt.xlabel("x1")
plt.ylabel("x2")
plt.show()
```



P2. Feature mapping

TODO

- implement function **map_feature()** to transform data from original space to the 28D space specified in the write-up

In [3]:
```
# Transform points to 28D space
def map_feature(X):
    aug = []
    for data in X:
        feature=[1]
        for i in range(1,7):
            for j in range(0,i+1):
                feature.append((data[0]**j) * (data[1]**(i-j)))
        aug.append(feature)
    return np.transpose(aug)
```

P3. Regularized Logistic Regression

TODO

- implement function **logistic_regpression_regularized()** as required in the write-up
- draw the decision boundary

Hints

- recycling code from HW2 is allowed
- you may use functions defined this section for part 4 below
- although optional for the report, plotting the convergence curve will be helpful

In [4]:
```
X = map_feature(x)
```

In [5]:
```
X.shape
```

Out[5]:
```
(28, 252)
```

In [6]:
```
# Define your functions here

# Pass in the required arguments
# Implement the sigmoid function
def sigmoid(weights, X):
    e = np.dot(weights, X)
    sig = 1 / (1 + np.exp(-e))
    return sig

# Pass in the required arguments
# The function should return the gradients
def calculate_gradients(pred, X, Y, weights, reg):
    grads = np.array(np.mean((pred - Y) * X[0,:]))
    for i in range(1, X.shape[0]):
        grads = np.append(grads,np.mean((pred - Y) * X[i,:]) + reg * weights
    return grads

# Update the weights using gradients calculated using above function and lea
# The function should return the updated weights to be used in the next step
def update_weights(prev_weights, current_grads, learning_rate):

    for i in range(len(prev_weights)):
        prev_weights[i] = prev_weights[i] - learning_rate*current_grads[i]
    return prev_weights

# Use the implemented functions in the main function
# 'main' function should return weights after all the iterations
# Dont forget to divide by the number of datapoints wherever necessary!
# Initialize the intial weights randomly

def main(X, Y, learning_rate = 0.00005, num_steps = 50000, reg = 1):
    weights = np.zeros(28)


    for i in range(num_steps):
        sig = sigmoid(weights, X)
        grads = calculate_gradients(sig, X, Y, weights, reg)
        weights = update_weights(weights, grads, learning_rate)
    sig = sigmoid(weights, X)
    grads = calculate_gradients(sig, X, Y, weights, reg)
    weights = update_weights(weights, grads, learning_rate)
    return weights

# Pass in the required arguments (final weights and input)
# The function should return the predictions obtained using sigmoid function
def predict(weights, X):
    Y_hat = sigmoid(weights, X)
    return Y_hat # np.where(Y_hat > 0.5, 1, 0)


# Plot decision boundary
```

In [7]:
```
weights = main(X,y)
```

In [8]:
```
print("Final weights are:\n",weights)
```

```
Final weights are:
 [ 0.21006022  0.06950377  0.10119934  0.06790672  0.00730979  0.09657131
  0.06996385  0.06685955  0.06128291  0.18470807 -0.04490942  0.04799657
  0.02092143  0.05679367  0.08581837  0.08843819  0.06268453  0.03550517
  0.05999456  0.05394752  0.24501007 -0.03291173 -0.02275976 -0.10666548
  0.03102841 -0.11417006  0.00078319 -0.15749374]
```

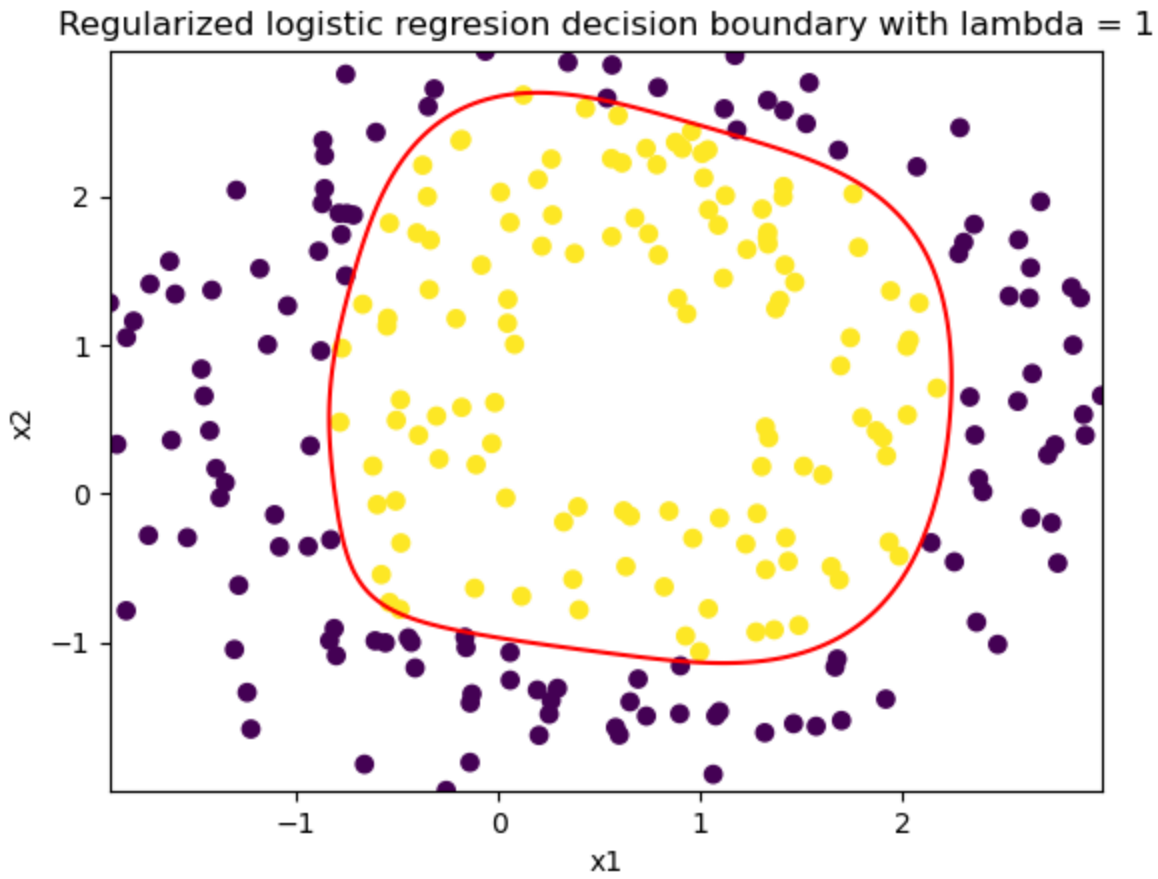
In [9]:
```
final_prediction = predict(weights, X)
```

In [10]:
```
np.where(final_prediction > 0.5, 1, 0)
```

Out[10]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
       0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0,
       1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1,
       1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1,
       1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1,
       0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 0, 1, 1, 1, 1])
```

In [11]:
```
xx, yy = np.meshgrid(np.linspace(x[:,0].min(), x[:,0].max(), num=200),
                     np.linspace(x[:,1].min(), x[:,1].max(), num=200))
Z = predict(weights, map_feature(np.c_[xx.ravel(), yy.ravel()])).reshape(xx.
```

In [12]:
```
plt.scatter(x[:,0], x[:,1], c=np.where(final_prediction > 0.5, 1, 0))
plt.contour(xx, yy, Z, levels = [0.5], colors='red')
plt.xlabel("x1")
plt.ylabel("x2")
plt.title("Regularized logistic regresion decision boundary with lambda = 1"
plt.show()
```



P4. Tune the strength of regularization

TODO

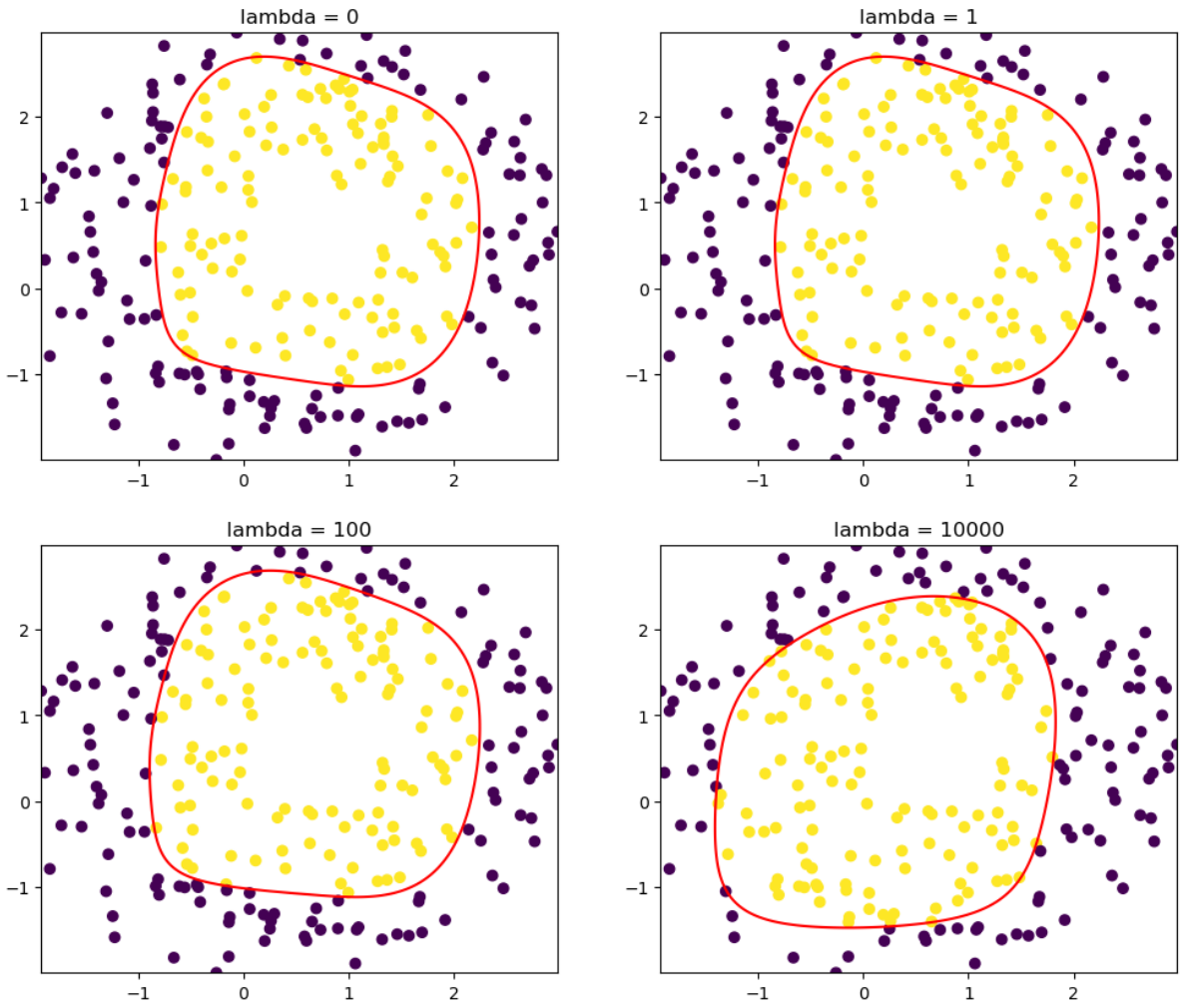
- tweak the hyper-parameter λ to be $[0, 1, 100, 10000]$
- draw the decision boundaries

In [13]:
```
# lambda = 0
final_weights_0 = main(X,y, reg = 0)
pred_0 = predict(final_weights_0, X)

# lambda = 1
final_weights_1 = main(X,y, reg = 1)
pred_1 = predict(final_weights_1, X)

# lambda = 100
final_weights_100 = main(X,y, reg = 100)
pred_100 = predict(final_weights_100, X)

# lambda = 10000
final_weights_10000 = main(X,y, reg = 10000)
pred_10000 = predict(final_weights_10000, X)
```

In [14]:
```
xx, yy = np.meshgrid(np.linspace(x[:,0].min(), x[:,0].max(), num=200),
                     np.linspace(x[:,1].min(), x[:,1].max(), num=200))
Z0 = predict(final_weights_0, map_feature(np.c_[xx.ravel(), yy.ravel()])).re
Z1 = predict(final_weights_1, map_feature(np.c_[xx.ravel(), yy.ravel()])).re
Z100 = predict(final_weights_100, map_feature(np.c_[xx.ravel(), yy.ravel()])
Z10000 = predict(final_weights_10000, map_feature(np.c_[xx.ravel(), yy.ravel
```

In [15]:
```
fig, axs = plt.subplots(2, 2, figsize=(12,10))
fig.suptitle("Regularized logistic regresion decision boundaries with differ
axs[0, 0].scatter(x[:,0], x[:,1], c=np.where(pred_0 > 0.5, 1, 0))
axs[0, 0].contour(xx, yy, Z0, 1, levels=[0.5], colors='red')
axs[0, 0].set_title("lambda = 0")
axs[0, 1].scatter(x[:,0], x[:,1], c=np.where(pred_1 > 0.5, 1, 0))
axs[0, 1].contour(xx, yy, Z1, 1, levels=[0.5], colors='red')
axs[0, 1].set_title("lambda = 1")
axs[1, 0].scatter(x[:,0], x[:,1], c=np.where(pred_100 > 0.5, 1, 0))
axs[1, 0].contour(xx, yy, Z100, 1, levels=[0.5], colors='red')
axs[1, 0].set_title("lambda = 100")
axs[1, 1].scatter(x[:,0], x[:,1], c=np.where(pred_10000 > 0.5, 1, 0))
axs[1, 1].contour(xx, yy, Z10000, levels=[0.5], colors='red')
axs[1, 1].set_title("lambda = 10000")
```

Out[15]:
```
Text(0.5, 1.0, 'lambda = 10000')
```



Answer for part (d) here:

Increasing the lambda hyper-parameter seems to let the decision boundary grow, meaning that it gets larger and encompasses more of the data. This happens because, by increasing lambda we are increasing the regularization, setting a tighter constraint for the gradients during training. This regularization reduces overfitting which is why the resulting decision boundary is larger and with a lower variance.