Homework 3

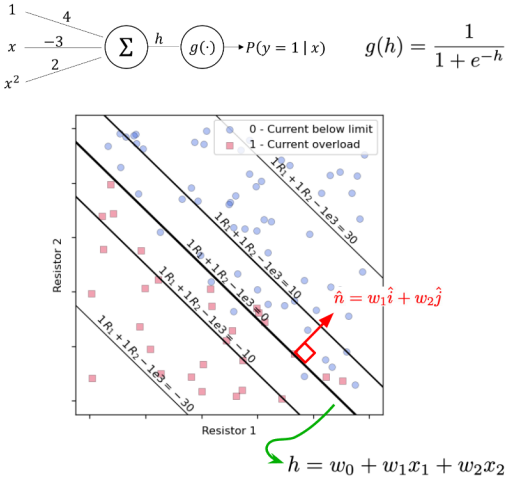
Instructions
This homework contains **6** concepts and **9** programming questions. In MS word or a similar text editor, write down the problem number and your answer for each problem. Combine all answers for concept questions in a single PDF file. Export/print the Jupyter notebook as a PDF file including the code you implemented and the outputs of the program. Make sure all plots and outputs are visible in the PDF.

Combine all answers into a single PDF named andrewID_hw3.pdf and submit it to Gradescope before the due date. Refer to the syllabus for late homework policy. Please assign each question a page by using the "Assign Questions and Pages" feature in Gradescope. Submission to anywhere else than Gradescope will not be graded.

Problem 1 (3 Points)
The sigmoid function is useful because

1. It restricts the output between -1 and 1 (T/F)
2. It has a probabilistic interpretation (T/F)
3. It is easily differentiable (T/F)

Problem 2 (2 Points)



Consider the level sets that correspond to different decision boundaries in the figure.

1) What are the bounds on the values of h?

2) What are the bounds on the values of *g(h)* where *g()* is the sigmoid function?

Problem 3 (1 Points)

More L2 regularization always leads to better fitting models. (T/F)

Problem 4 (1 Points)

Consider the following 4 class problem. A given test point ***x*** is evaluated by six binary classifiers with the following results:

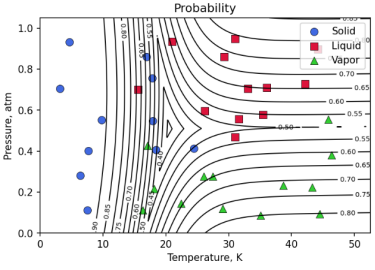
A vs. B → class A
A vs. C → class C
A vs. D → class D
B vs. C → class C
B vs. D → class D
C vs. D → class D

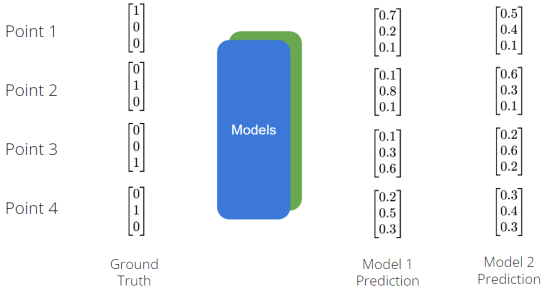
What is the predicted class for the test point?

Problem 5 (1 Points)

For what number of classes is the number of classifiers required for one-versus-one and one-versus-rest classifiers equal?

Problem 6 (2 Points)





Consider the phase problem from the slides. There are three classes: solid, liquid, and vapor. We have four test points with ground truth labels shown below. We train two models that output the predictions below. By inspection, which model is best?

AIML CONCEPT HW-3

Srecharan Selvam - sselvam

Problem-1 :-

① FALSE – Restricts b/w 0 and 1

② TRUE

③ TRUE

Problem-2 :-

① h can take values from $-\infty$ to $+\infty$

∴ Bounds on the fig is $-\infty$ to ∞

② $g(h) = 1/(1+e^{-h})$ y nature of sigmoid function.

$g(h)$ is bound between 0 & 1

↳ $0 < g < 1$

Problem 3:-

→FALSE

Large weights are Penalised by L2 Regularisation to prevent overfitting, but too much of it might result in underfitting and a too-simplistic model. Regularisation must be balanced for the best model fit.

Problem 4:-

→To determine the predicted class, we need to see the class which had won the most number of times.

→Here, we get

A → 1 Time
B → 0 Time
C → 2 Times
D → 3 Times

∴ Hence, The test point is CLASS-D

Problem-5

The number of classifiers for OVO = $\frac{n(n-1)}{2}$

The number of classifiers for OVR = n classes

$$\frac{n(n-1)}{2} = n$$

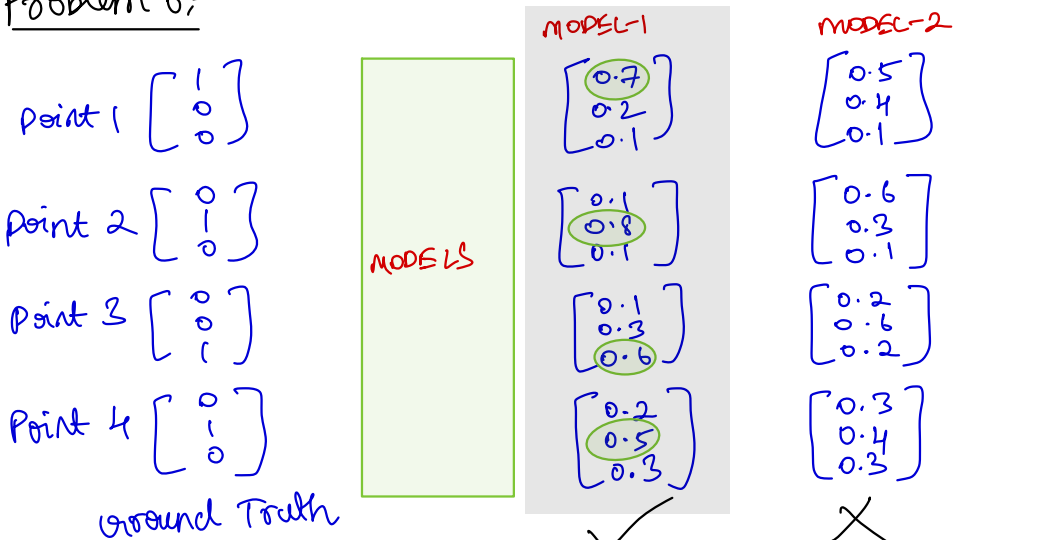
$$n(n-1) = 2n$$

$$(n-1) = 2$$

$$n = 2+1$$

$$n = 3$$

Problem 6:-



By Inspection, model 1's prediction ✓ seems to be better than model 2's prediction ✗

MODEL-1

Problem 1 (5 points)

In [26]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Sigmoid function

Define a function, `sigmoid(h)`, which computes and returns the sigmoid `g(h)` given an input `h`. Recall the mathematical formulation of sigmoid:

$$g(h) = \frac{1}{1+e^{-h}}$$

In [27]:
```
def sigmoid(h):
    # YOUR CODE GOES HERE
    return 1/(1+np.exp(-h))
```

Transformation function

In logistic regression, we transform the input before applying the sigmoid function. This transformation can take many forms, but here let's define a function `transform_quadratic(x,w)` that takes in an input `x`, and a weight vector `w`, and returns the sum $w_0 \cdot 1 + w_1 \cdot x + w_2 \cdot x^2$.

In [28]:
```
def transform_quadratic(x, w):
    # YOUR CODE GOES HERE
    return w[0]*1+w[1]*x+w[2]*x*x
```

Example

Now, we will use both `sigmoid()` and `transform_quadratic()` in a logistic regression context.

Suppose a logistic regression model states that:

$$P(y = 1 \mid x) = g(\mathbf{w}'x),$$

for $g(h)$ the sigmoid function and $\mathbf{w} = [4, -3, 2]$.

Use the functions you wrote to compute $P(y = 1 \mid x = 1.2)$ and $P(y = 1 \mid x = 7)$. Print these probabilities.

In [30]:
```
w = [4,-3,2]
for x in [1.2, 7.]:
```

```
    P = sigmoid(transform_quadratic(x,w))
    print(f"x = {x:3} --> P(y=1) = {P}")
```

```
x = 1.2 --> P(y=1) = 0.9637362836253517
x = 7.0 --> P(y=1) = 1.0
```

Problem 2 (5 points)

In [11]:
```
import numpy as np
import matplotlib.pyplot as plt


def plot_data(data, c, title="", xlabel="$x_1$",ylabel="$x_2$",classes=["",""],alpha=1
    N = len(c)
    colors = ['royalblue','crimson']
    symbols = ['o','s']

    plt.figure(figsize=(5,5),dpi=120)

    for i in range(2):
        x = data[:,0][c==i]
        y = data[:,1][c==i]

        plt.scatter(x,y,color=colors[i],marker=symbols[i],edgecolor="black",linewidths

    plt.legend(loc="upper right")
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    ax = plt.gca()
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    plt.xlim([-0.05,1.05])
    plt.ylim([-0.05,1.05])
    plt.title(title)

def plot_contour(predict, mapXY = None):
    res = 500
    vals = np.linspace(-0.05,1.05,res)
    x,y = np.meshgrid(vals,vals)
    XY = np.concatenate((x.reshape(-1,1),y.reshape(-1,1)),axis=1)
    if mapXY is not None:
        XY = mapXY(XY)
    contour = predict(XY).reshape(res, res)
    plt.contour(x, y, contour)
```

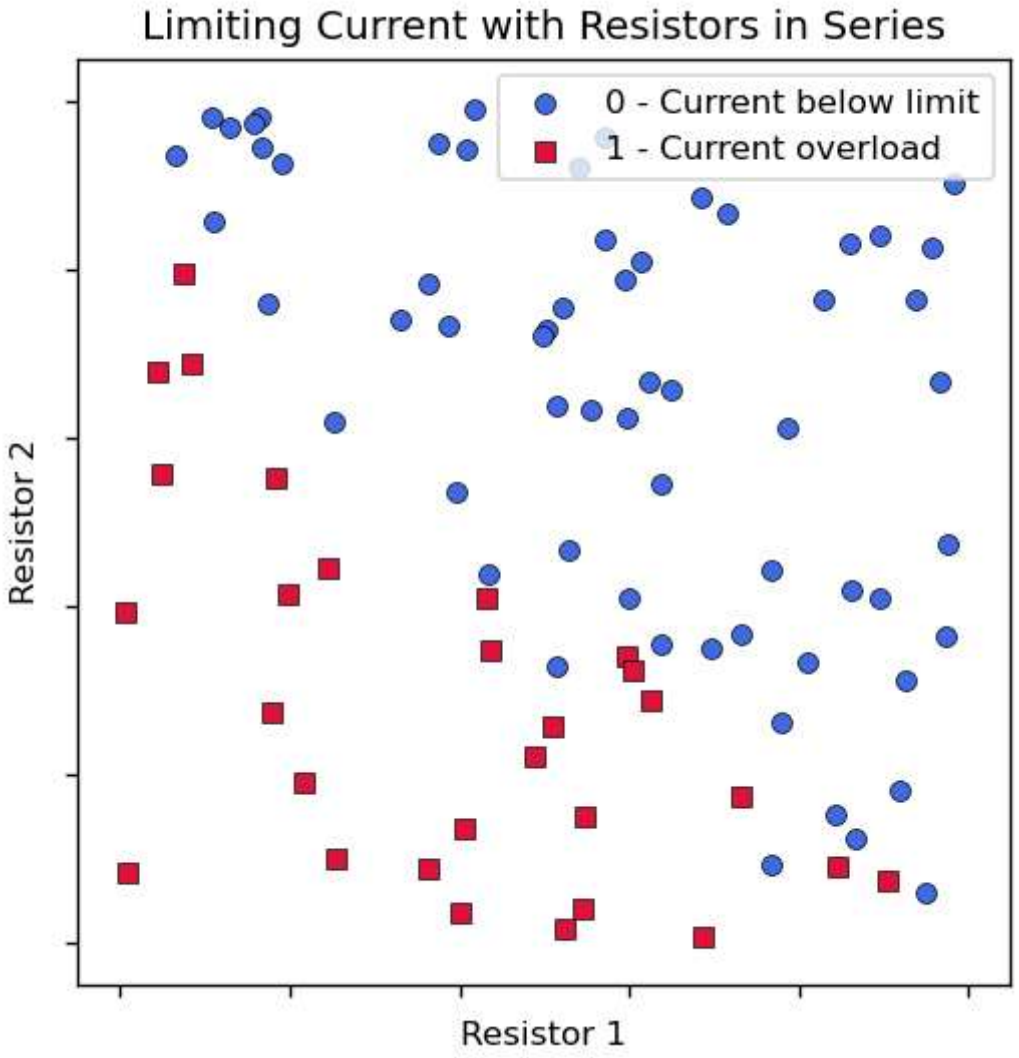
Generate Dataset

(Don't edit this code.)

In [12]:
```
def get_line_dataset():
    np.random.seed(4)
    x = np.random.rand(90)
    y = np.random.rand(90)

    h = 1/.9*x + 1/0.9*y - 1

    d = 0.1
    x1, y1 = x[h<-d], y[h<-d]
    x2, y2 = x[np.abs(h)<d], y[np.abs(h)<d]
    x3, y3 = x[h>d], y[h>d]
```

```
    c1 = np.ones_like(x1)
    c2 = (np.random.rand(len(x2)) > 0.5).astype(int)
    c3 = np.zeros_like(x3)
    xs = np.concatenate([x1,x2,x3],0)
    ys = np.concatenate([y1,y2,y3],0)
    c = np.concatenate([c1,c2,c3],0)
    return np.vstack([xs,ys]).T,c
```

In [13]:
```
data, classes = get_line_dataset()
format = dict(title="Limiting Current with Resistors in Series", xlabel="Resistor 1",

plot_data(data, classes, **format)
```



Define helper functions

First, fill in code to complete the following functions. You may use code you wrote in the previous question.

- `sigmoid(h)` to compute the sigmoid of an input `h`
- (Given) `transform(data, w)` to add a column of ones to `data` and then multiply by the 3-element vector `w`

- (Given) `loss(data,y,w)` to compute the logistic regression loss function:

$$L(x,y,w) = \sum_{i=1}^{n} -y^{(i)} \cdot \ln(g(w'x^{(i)})) - (1-y^{(i)}) \cdot \ln(1-g(w'x^{(i)}))$$

- `gradloss(data,y,w)` to compute the gradient of the loss function with respect to w: $$ \frac{\partial L}{wj} = \sum{i=1}^n (g(w'x^{(i)}) - y^{(i)}) x_j^{(i)}

In [32]:
```
def sigmoid(h):
    # YOUR CODE GOES HERE
    return 1/(1+np.exp(-h))

def transform(data, w):
    xs = data[:,0]
    ys = data[:,1]
    ones = np.ones_like(xs)
    h = w[0]*ones + w[1]*xs + w[2]*ys
    return h

def loss(data, y, w):
    wt_x = transform(data,w)
    J1 = -np.log(sigmoid(wt_x)) * y
    J2 = -np.log(sigmoid(wt_x)) * (1-y)
    L = np.sum(J1 + J2)
    return L

def gradloss(data, y, w):
    # YOUR CODE GOES HERE
    xs = data[:, 0]                 # x coordinates
    ys = data[:, 1]                 # y coordinates
    ones = np.ones_like(data[:,0])  # Column vector of 1's
    wt_x = transform(data,w)        # Weighted sum of inputs
    gy = sigmoid(wt_x) - y

    w0 = np.sum(gy * ones)
    w1 = np.sum(gy * xs)
    w2 = np.sum(gy * ys)

    return np.array([w0, w1,w2])
```

Gradient Descent

Now you'll write a gradient descent loop. Given a number of iterations and a step size, continually update `w` to minimize the loss function. Use the `gradloss` function you wrote to compute a gradient, then move `w` by `stepsize` in the direction opposite the gradient. Return the optimized `w`.

In [35]:
```
def grad_desc(data, y, w0=np.array([0,0,0]), iterations=100, stepsize=0.1):
    # YOUR CODE GOES HERE
    w=w0
    for _ in range(iterations):
        grad = gradloss(data, y, w)
        w=w-(stepsize * grad)
    return w
```

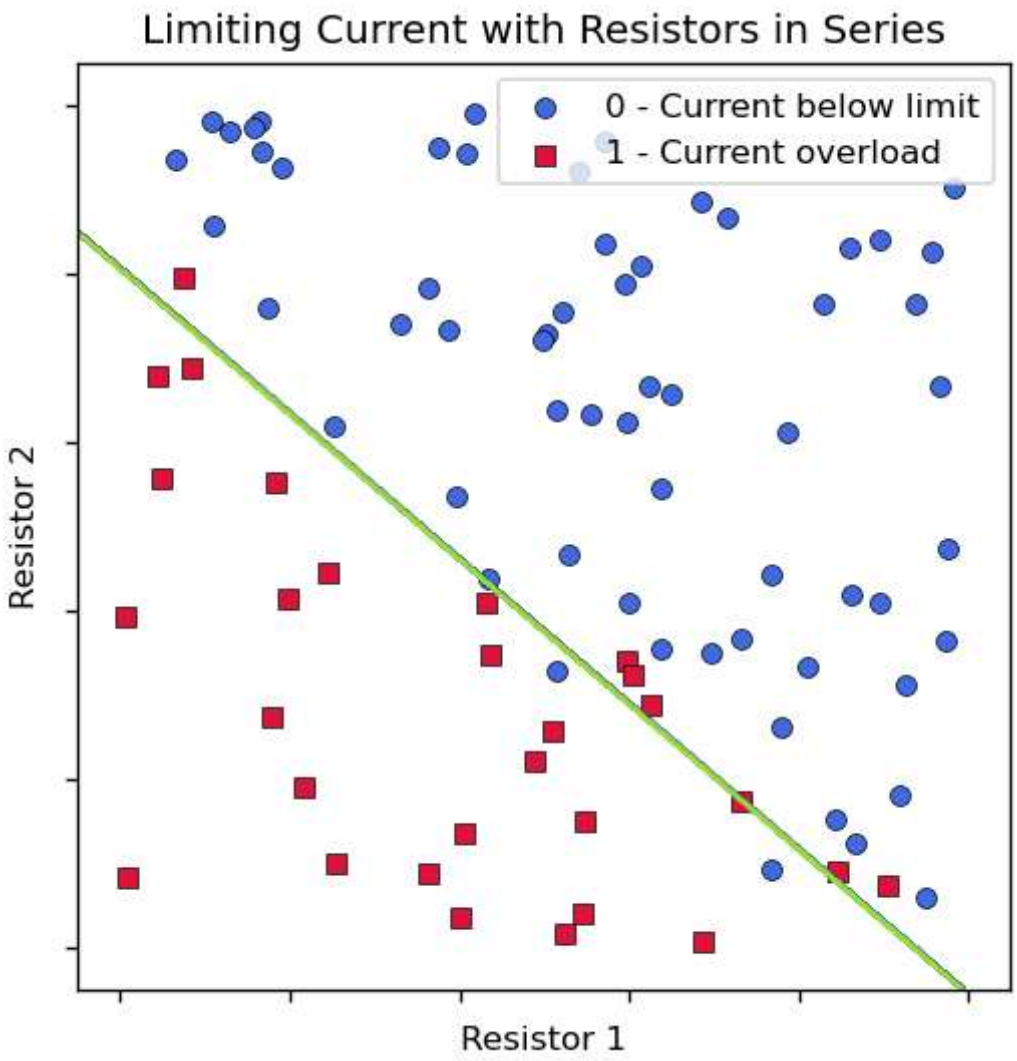
Test your classifier

Run these cells to find the optimal `w`, compute the accuracy on the training data, and plot a decision boundary.

In [36]:
```
w = grad_desc(data,classes)
preds = np.round(sigmoid(transform(data, w))).astype(int)
accuracy = np.sum(preds == classes) / len(classes) * 100
print("          w = ", w)
print("True Classes: ", classes.astype(int))
print(" Predictions: ", preds)
print("    Accuracy: ", accuracy, r"%")
```

```
          w =  [ 7.99449326 -8.54560847 -9.92653181]
True Classes:  [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 1 1 0 1 1 1 1 1 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 0 0 0]
 Predictions:  [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 1 0 0 1 0 0 0 1 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 0 0]
    Accuracy:  91.11111111111111 %
```

In [37]:
```
predict = lambda data: np.round(sigmoid(transform(data, w)))
plot_data(data, classes, **format)
plot_contour(predict)
plt.show()
```


Limiting Current with Resistors in Series
0 - Current below limit
1 - Current overload
Resistor 2
Resistor 1

Problem 3 (5 points)

In [20]:
```
import numpy as np
import matplotlib.pyplot as plt


def plot_data(data, c, title="", xlabel="$x_1$",ylabel="$x_2$",classes=["",""],alpha=1
    N = len(c)
    colors = ['royalblue','crimson']
    symbols = ['o','s']

    plt.figure(figsize=(5,5),dpi=120)

    for i in range(2):
        x = data[:,0][c==i]
        y = data[:,1][c==i]

        plt.scatter(x,y,color=colors[i],marker=symbols[i],edgecolor="black",linewidths

    plt.legend(loc="upper right")
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    ax = plt.gca()
    ax.set_xticklabels([])
    ax.set_yticklabels([])
    plt.xlim([-0.05,1.05])
    plt.ylim([-0.05,1.05])
    plt.title(title)

def plot_contour(predict, mapXY = None):
    res = 500
    vals = np.linspace(-0.05,1.05,res)
    x,y = np.meshgrid(vals,vals)
    XY = np.concatenate((x.reshape(-1,1),y.reshape(-1,1)),axis=1)
    if mapXY is not None:
        XY = mapXY(XY)
    contour = predict(XY).reshape(res, res)
    plt.contour(x, y, contour)
```

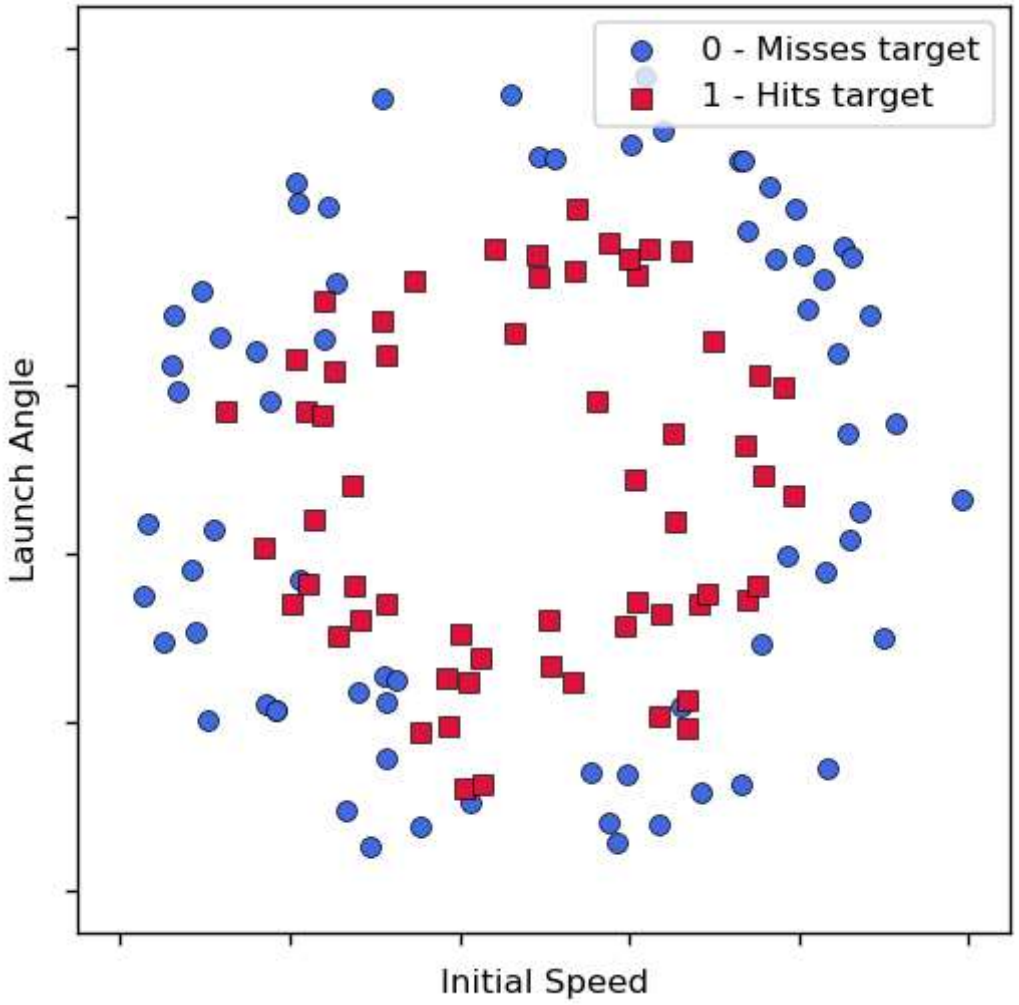
Generate Dataset

(Don't edit this code.)

In [21]:
```
def sample_ring(N,x,y,ro,ri):
    theta = np.random.rand(N)*2*np.pi
    r = np.random.rand(N)
    r = np.sqrt(r*(ro**2-ri**2)+ri**2)
    xs = x + r * np.cos(theta)
    ys = y + r * np.sin(theta)
    return xs, ys

def get_ring_dataset():
    np.random.seed(0)
    c0 = sample_ring(70,0.5,0.5,0.5,0.3)
```

```
    c1 = sample_ring(60,0.45,0.47,0.36,0.15)
    xs = np.concatenate([c0[0],c1[0]],0)
    ys = np.concatenate([c0[1],c1[1]],0)
    c = np.concatenate([np.zeros(70),np.ones(60)],0)
    return np.vstack([xs,ys]).T, c
```

In [22]:
```
data, classes = get_ring_dataset()
format = dict(xlabel="Initial Speed",ylabel="Launch Angle", classes=["0 - Misses targe

plot_data(data, classes, **format)
```



Feature Expansion

Define a function to expand 2 features into more features For the features x_1 and x_2, expand into:

- 1
- x_1
- x_2
- x_1^2
- x_2^2
- $\sin(x_1)$

- $\cos(x_1)$
- $\sin(x_2)$
- $\cos(x_2)$
- $\sin^2(x_1)$
- $\cos^2(x_1)$
- $\sin^2(x_2)$
- $\cos^2(x_2)$
- $\exp(x_1)$
- $\exp(x_2)$

In [23]:
```
def feature_expand(x):
    x1 = x[:,0].reshape(-1, 1)
    x2 = x[:,1].reshape(-1, 1)

    # YOUR CODE GOES HERE:
    columns = [np.ones_like(x1), x1, x2, x1*x1, x2*x2, np.sin(x1), np.cos(x1), np.sin(

    X = np.concatenate(columns, axis=1)
    return X

features = feature_expand(data)
print("Dataset size:", np.shape(data))
print("Expanded dataset size:", np.shape(features))
```

```
Dataset size: (130, 2)
Expanded dataset size: (130, 15)
```

Logistic Regression

Use SciKit-Learn's Logistic Regression model to learn the decision boundary for this data, using regularization. (The `C` argument controls regularization strength.)

Train this model on your expanded feature set.

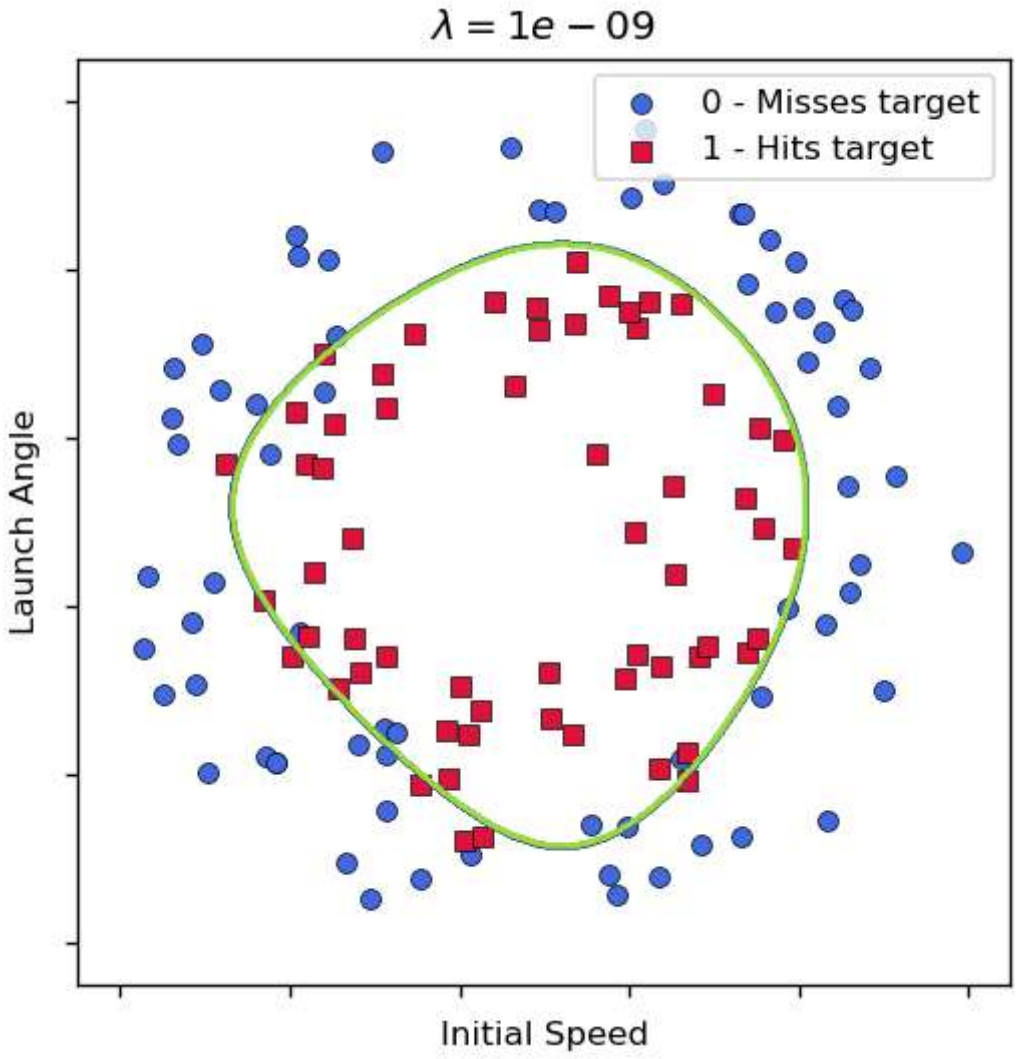
Details about how to use this are here: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

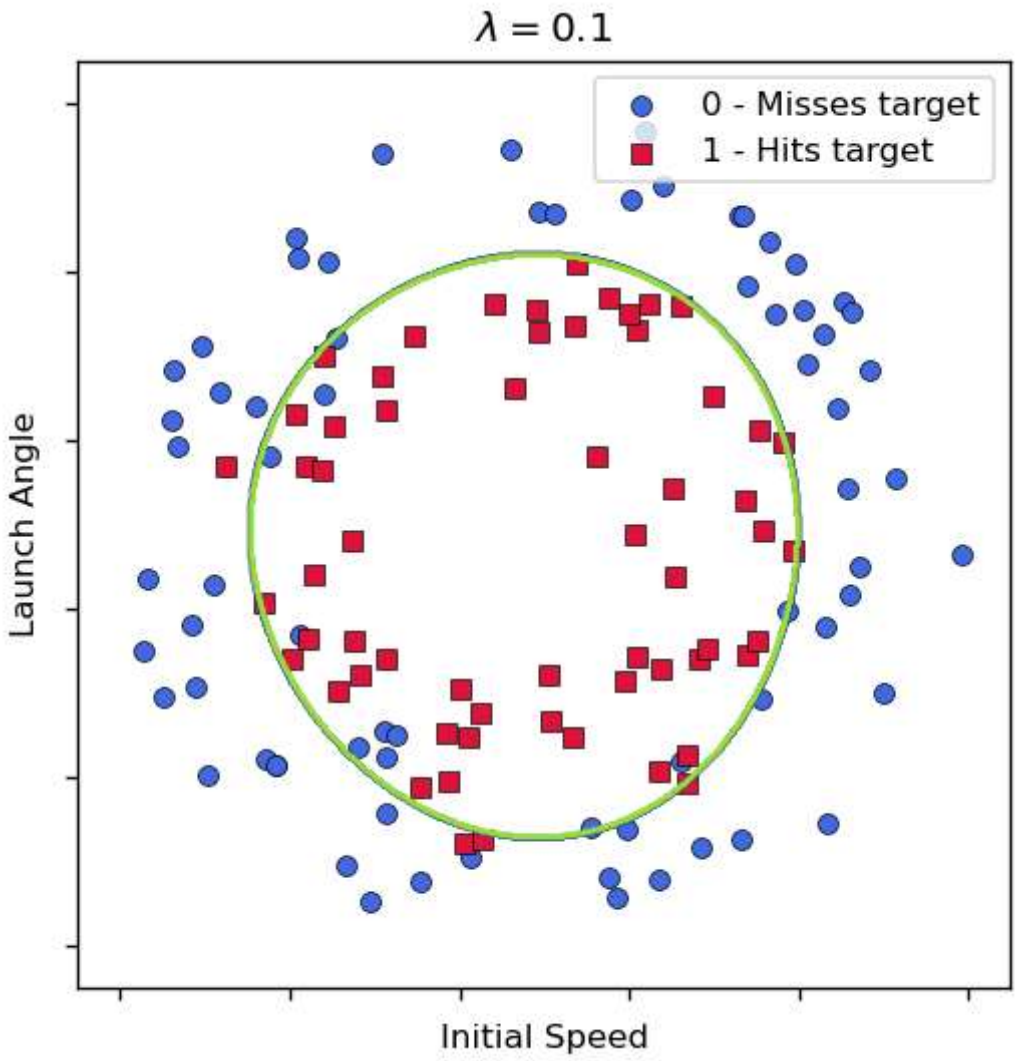
Notes:

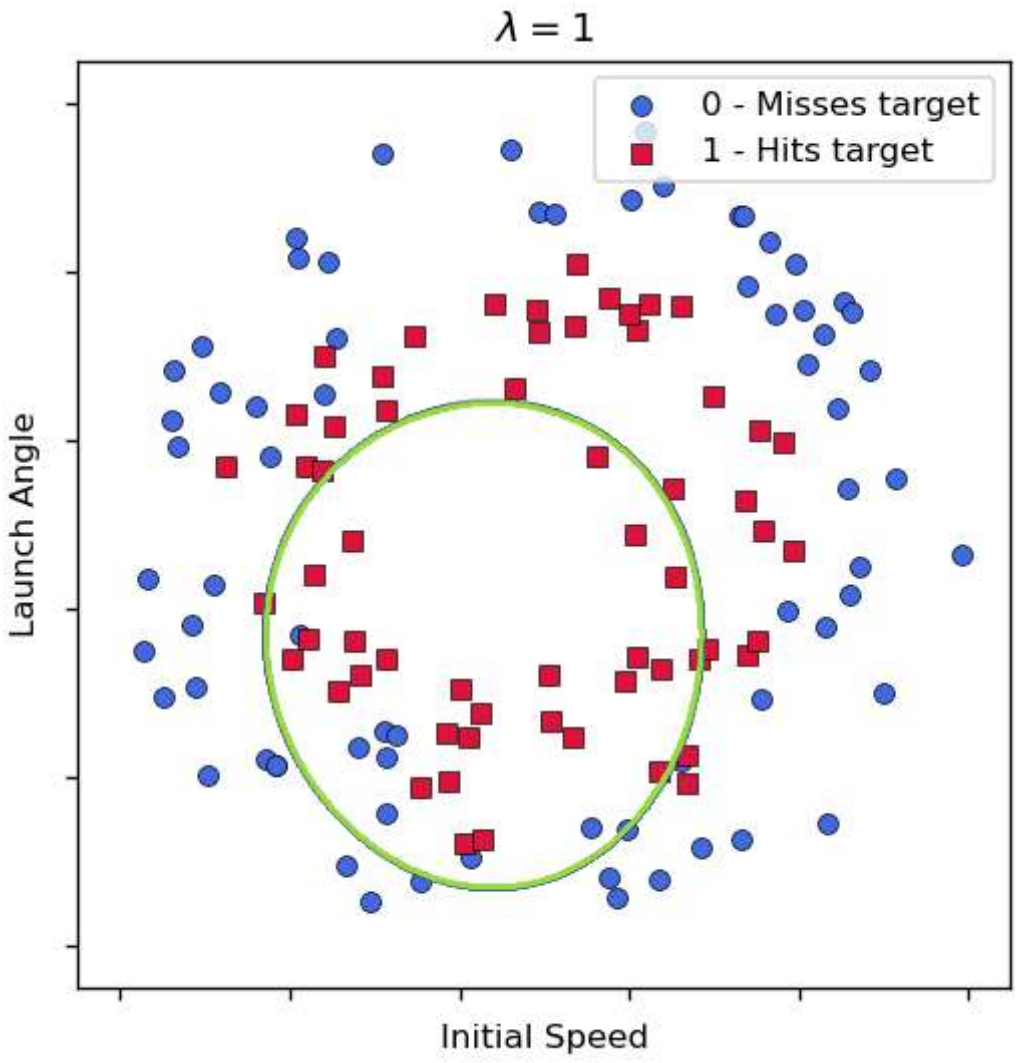
- λ is related to sklearn's regularization strength C by: $\lambda = 1/C$
- You may want to increase the maximum number of iterations

In [30]:
```
from sklearn.linear_model import LogisticRegression

def get_logistic_regressor(features, classes, L = 1):
    # YOUR CODE GOES HERE
    lam = 1/L
    reg = LogisticRegression(C=lam, max_iter=10000)   # - Instantiate model with regul
    reg.fit(features, classes)                        # - Fit model to expanded data
    return reg
```

In [31]:
```
for L in [1e-9, 1e-1, 1]:
    model = get_logistic_regressor(features, classes, L)
    plot_data(data, classes, **format, title=f"$\lambda={L}$")
    plot_contour(model.predict, feature_expand)
    plt.show()
```




$\lambda = 0.1$
0 - Misses target
1 - Hits target
Launch Angle
Initial Speed



As λ increases, note what happens to the decision boundary. Why does this occur?

In [32]:
```
print("The decision boundary smoothes out and may become less overfitting as lambda ir
```

```
The decision boundary smoothes out and may become less overfitting as lambda increase
s. This is because higher regularisation results from greater lambda values, which pe
nalises overly complicated models that could be trying to match the noise in the trai
ning data. On the other hand, when lambda is very small, the model is more independen
t to fit the training data closely, which might result in overfitting or noise captur
e.
```

Problem 4 (5 points)

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from scipy.stats import mode
from sklearn.linear_model import LogisticRegression
```

One-vs-One Multinomial Classification

Load Dataset

(Don't edit this)

- (x,y) values are stored in rows of `xy`
- class values are in `c`

In [3]:
```
x = np.array([7.4881350392732475,16.351893663724194,22.427633760716436,29.048831829968
y = np.array([0.11120957227224215,0.1116933996874757,0.14437480785146242,0.1181820299
xy = np.vstack([x,y]).T
c = np.array([0,2,2,2,2,2,0,2,2,2,2,2,0,0,2,0,1,2,0,0,1,1,1,2,0,1,0,1,1,1,0,0,1,1,1,1
```

Binomial classification function

You are given a function that performs binomial classification by using sklearn's `LogisticRegression` tool: `classify = get_binomial_classifier(xy, c, A, B)`

To use it, input:

- `xy`, an array in which each row contains (x,y) coordinates of data points
- `c`, an array that specifies the class each point in `xy` belongs to
- `A`, the class of the first group (0, 1, or 2 in this problem)
- `B`, the class of the second group (0, 1, or 2 in this problem), but different from `A`

The function outputs a classifier function (`classify()` in this case), used to classify any new `xy` into group A or B, such as by using `classify(xy)`.

In [4]:
```
def get_binomial_classifier(xy, c, A, B):
    assert A != B
    xyA, xyB = xy[c==A], xy[c==B]
    cA, cB = c[c==A], c[c==B]
    model = LogisticRegression()
    xy_new = np.concatenate([xyA, xyB], 0)
    c_new = np.concatenate([cA, cB], 0)
    model.fit(xy_new,c_new)
```

```
    def classify(xy):
        pred = model.predict(xy)
        return pred

    return classify
```

Coding a 1v1 classifier

Now you will create a one-vs-one classifier to do multinomial classification. This will generate binomial classifiers for each pair of classes in the dataset. Then to predict the class of a new point, classify it using each of the binomial classifiers, and select the majority winner as the class prediction.

Complete the two functions we have started:

- `generate_all_classifiers(xy, c)` which returns a list of binary classifier functions for all possible pairs of classes (among 0, 1, and 2 in this problem)
- `classify_majority(classifiers, xy)` which loops through a list of classifiers and gets their predictions for each point in `xy`. Then using a majority voting scheme at each point, return the overall class predictions for each point.

In [38]:
```
def generate_all_classifiers(xy, c):
    # YOUR CODE GOES HERE
    # Use get_binomial_classifier() to get binomial classifiers for each pair of class
    # and return a list of these classifiers
    classifiers=[]
    cl=np.unique(c)
    for i in range(len(cl)):
        for j in range(i+1,len(cl)):
            A=cl[i]
            B=cl[j]
            clr=get_binomial_classifier(xy,c,A,B)
            classifiers.append(clr)
    return classifiers


def classify_majority(classifiers, xy):
    n=len(np.unique(c))
    pred_1=np.zeros((len(xy), n))
    for i in classifiers:
        pred_2=i(xy)
        for j in range(len(xy)):
            pred_1[j][pred_2[j]]=pred_1[j][pred_2[j]] + 1
    majority_pred=np.argmax(pred_1, axis=1)

    return majority_pred
```

Trying out our multinomial classifier:

In [39]:
```
classifiers = generate_all_classifiers(xy, c)
preds = classify_majority(classifiers, xy)
accuracy = np.sum(preds == c) / len(c) * 100
print("True Classes:", c)
print(" Predictions:", preds)
print("    Accuracy:", accuracy, r"%")
```

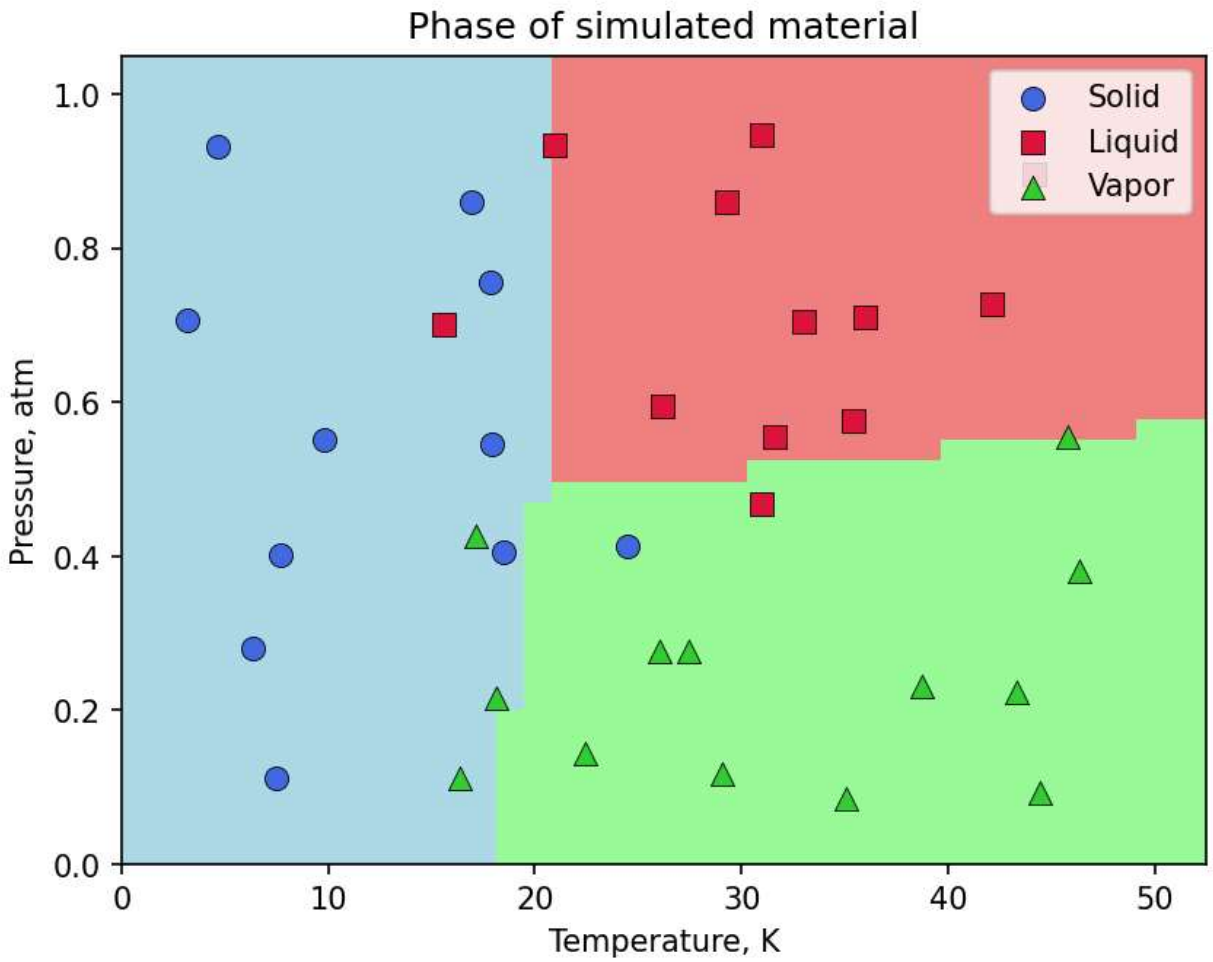
```
True Classes: [0 2 2 2 2 2 0 2 2 2 2 2 0 0 2 0 1 2 0 0 1 1 1 2 0 1 0 1 1 1 0 0 1 1 1
1]
 Predictions: [0 0 2 2 2 2 0 0 2 2 2 2 0 0 0 2 2 2 0 0 1 1 1 1 0 0 0 1 1 1 0 0 1 1 1
1]
    Accuracy: 80.55555555555556 %
```

Plotting a Decision Boundary

Here, we have made some plotting functions -- run these cells to visualize the decision boundaries.

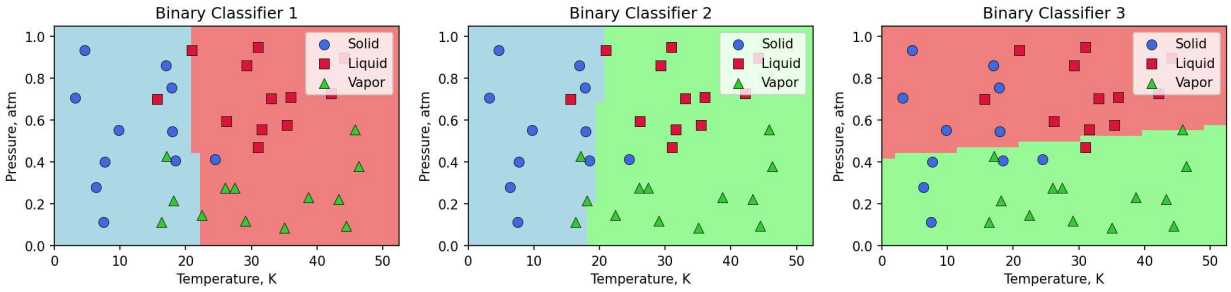
In [42]:
```
def plot_data(x, y, c, title="Phase of simulated material", newfig=True):
    xlim = [0,52.5]
    ylim = [0,1.05]
    markers = [dict(marker="o", color="royalblue"), dict(marker="s", color="crimson"),
    labels = ["Solid", "Liquid", "Vapor"]

    if newfig:
        plt.figure(dpi=150)

    for i in range(1+max(c)):
        plt.scatter(x[c==i], y[c==i], s=60, **(markers[i]), edgecolor="black", linewic

    plt.title(title)
    plt.legend(loc="upper right")
    plt.xlim(xlim)
    plt.ylim(ylim)
    plt.xlabel("Temperature, K")
    plt.ylabel("Pressure, atm")
    plt.box(True)

def plot_colors(classifiers, res=40):
    xlim = [0,52.5]
    ylim = [0,1.05]
    xvals = np.linspace(*xlim,res)
    yvals = np.linspace(*ylim,res)
    x,y = np.meshgrid(xvals,yvals)
    XY = np.concatenate((x.reshape(-1,1),y.reshape(-1,1)),axis=1)
    if type(classifiers) == list:
        color = classify_majority(classifiers,XY).reshape(res,res)
    else:
        color = classifiers(XY).reshape(res,res)
    cmap = ListedColormap(["lightblue","lightcoral","palegreen"])
    plt.pcolor(x, y, color, shading="nearest", zorder=-1, cmap=cmap,vmin=0,vmax=2)
    return
```

In [43]:
```
plot_data(x,y,c)
plot_colors(classifiers)
plt.show()
```



We can also look at the results of each binary classifier:

In [44]:
```
plt.figure(figsize=(16,3),dpi=150)
for i in range(3):
    plt.subplot(1,3,i+1)
    plot_data(x, y, c, title=f"Binary Classifier {i+1}", newfig=False)
    plot_colors(classifiers[i])
plt.show()
```



Problem 5 (5 points)

In [4]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.linear_model import LogisticRegression
```

One-vs-All (One-vs-Rest) Multinomial Classification

Load Dataset

(Don't edit this)

- (x,y) values are stored in rows of `xy`
- class values are in `c`

In [5]:
```
x = np.array([7.4881350392732475,16.351893663724194,22.427633760716436,29.048831829968
y = np.array([0.11120957227224215,0.1116933996874757,0.14437480785146242,0.1181820299
xy = np.vstack([x,y]).T
c = np.array([0,2,2,2,2,2,0,2,2,2,2,2,0,0,2,0,1,2,0,0,1,1,1,2,0,1,0,1,1,1,0,0,1,1,1,1
```

Binomial classification function

You are given a function that performs binomial classification by using sklearn's `LogisticRegression` tool: `prob = get_ovr_prob_function(xy, c, A)`

To use it, input:

- `xy`, an array in which each row contains (x,y) coordinates of data points
- `c`, an array that specifies the class each point in `xy` belongs to
- `A`, the class of the group (0, 1, or 2 in this problem) -- classifies into A or "rest"

The function outputs a probability function (`prob()` in this case), used to determine the probability that each `xy` is class A or [not A], such as by using `prob(xy)`.

In [6]:
```
def get_ovr_prob_function(xy, c, A):
    c_new = (c == A).astype(int)

    model = LogisticRegression()
    model.fit(xy,c_new)

    def prob(xy):
        pred = model.predict_proba(xy)[:,1]
        return pred.flatten()
```

```
    return prob
```

Coding an OvR classifier

Now you will create a one-vs-rest classifier to do multinomial classification. Binomial predictions will be made for each class vs. the rest of the classes. The class whose binomial prediction gives the highest probability is the selected class.

Complete the two functions we have started:

- `generate_ovr_prob_functions(xy, c)` which returns a list of binary classifier probability functions for all possible classes (0, 1, and 2 in this problem)
- `classify_ovr(probs, xy)` which loops through a list of ovr classifier probabilities and gets the probability of belonging to each class, for each point in `xy`. Then taking the highest probability for each, return the overall class predictions for each point.

In [44]:
```
def generate_ovr_prob_functions(xy, c):
    # YOUR CODE GOES HERE
    cls=np.unique(c)  # if c=[0,1,0,2,1,2], then classes will be [0,1,2]
    prob=[]
    for i in cls:
        prob_fn=get_ovr_prob_function(xy, c, i)
        prob.append(prob_fn)
    return prob

def classify_ovr(probs, xy):
    # YOUR CODE GOES HERE
    every_prob=[]
    for i in probs:
        every_prob.append(i(xy))

    every_prob=np.array(every_prob)
    predictions=np.argmax(every_prob, axis=0)
    return predictions
```

Trying out our multinomial classifier:

In [45]:
```
probs = generate_ovr_prob_functions(xy, c)
preds = classify_ovr(probs, xy)
accuracy = np.sum(preds == c) / len(c) * 100
print("True Classes:", c)
print(" Predictions:", preds)
print("    Accuracy:", accuracy, r"%")
```

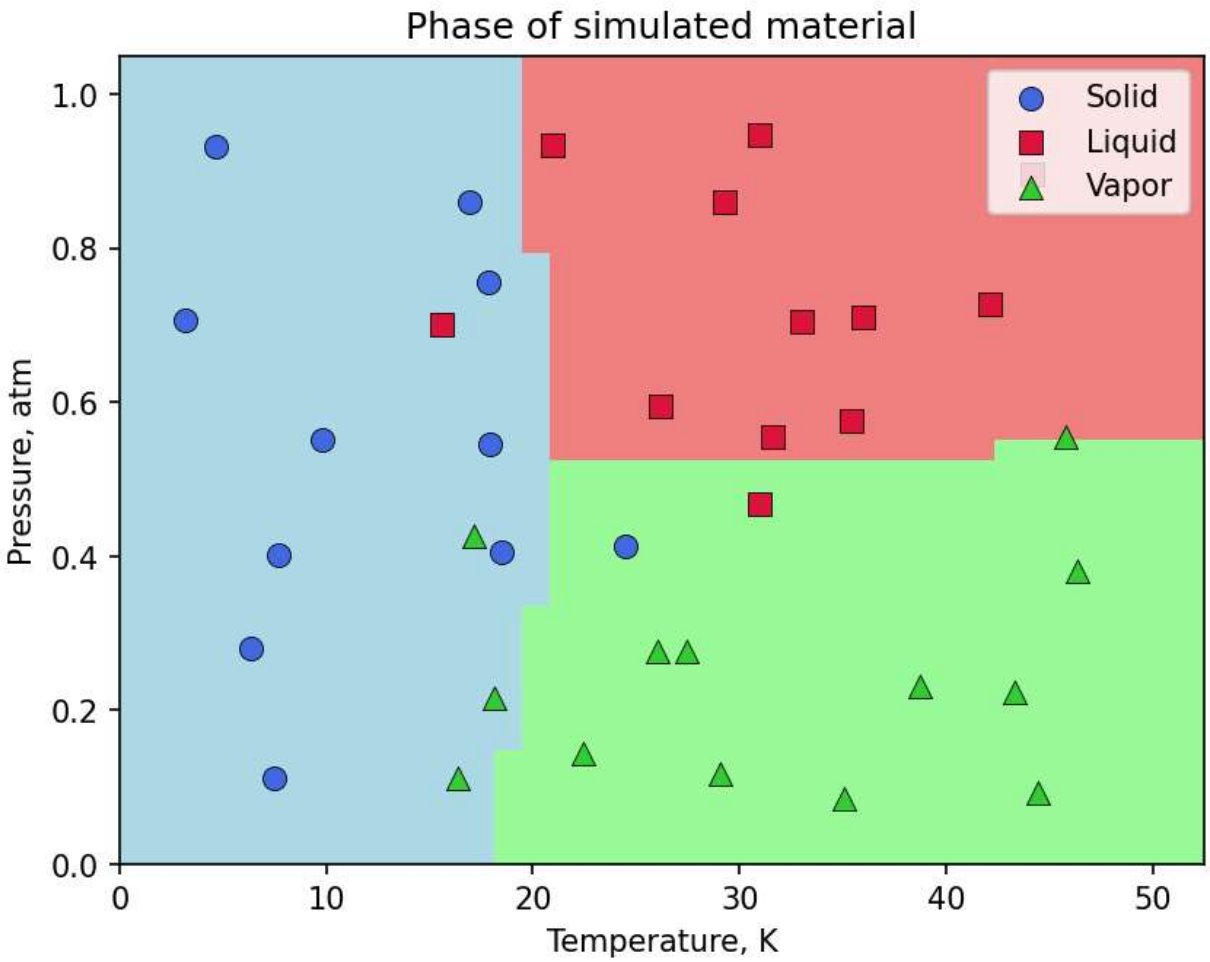
```
True Classes: [0 2 2 2 2 2 0 2 2 2 2 2 0 0 2 0 1 2 0 0 1 1 1 2 0 1 0 1 1 1 0 0 1 1 1
1]
 Predictions: [0 0 2 2 2 2 0 0 2 2 2 2 0 0 0 2 2 2 0 0 1 1 1 1 0 0 0 1 1 1 0 0 1 1 1
1]
    Accuracy: 80.55555555555556 %
```

Plotting multnomial classifier results

Here, we have made some plotting functions -- run these cells to visualize the decision boundaries.

In [46]:
```
def plot_data(x, y, c, title="Phase of simulated material", newfig=True):
    xlim = [0,52.5]
    ylim = [0,1.05]
    markers = [dict(marker="o", color="royalblue"), dict(marker="s", color="crimson"),
    labels = ["Solid", "Liquid", "Vapor"]

    if newfig:
        plt.figure(dpi=150)

    for i in range(1+max(c)):
        plt.scatter(x[c==i], y[c==i], s=60, **(markers[i]), edgecolor="black", linewid

    plt.title(title)
    plt.legend(loc="upper right")
    plt.xlim(xlim)
    plt.ylim(ylim)
    plt.xlabel("Temperature, K")
    plt.ylabel("Pressure, atm")
    plt.box(True)

def plot_ovr_colors(probs, res=40):
    xlim = [0,52.5]
    ylim = [0,1.05]
    xvals = np.linspace(*xlim,res)
    yvals = np.linspace(*ylim,res)
    x,y = np.meshgrid(xvals,yvals)
    XY = np.concatenate((x.reshape(-1,1),y.reshape(-1,1)),axis=1)

    color = classify_ovr(probs,XY).reshape(res,res)

    cmap = ListedColormap(["lightblue","lightcoral","palegreen"])
    plt.pcolor(x, y, color, shading="nearest", zorder=-1, cmap=cmap,vmin=0,vmax=2)
    return
```

In [47]:
```
plot_data(x,y,c)
plot_ovr_colors(probs)
plt.show()
```


Phase of simulated material
Solid
Liquid
Vapor
Pressure, atm
Temperature, K
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40
50

Problem 6 (5 points)

In [11]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.linear_model import LogisticRegression
```

Multinomial Classification in SciKit-Learn

Load Dataset

(Don't edit this)

- (x,y) values are stored in rows of `xy`
- class values are in `c`

In [12]:
```
x = np.array([7.4881350392732475,16.351893663724194,22.427633760716436,29.048831829968
y = np.array([0.11120957227224215,0.1116933996874757,0.14437480785146242,0.1181820299
xy = np.vstack([x,y]).T
c = np.array([0,2,2,2,2,2,0,2,2,2,2,2,0,0,2,0,1,2,0,0,1,1,1,2,0,1,0,1,1,1,0,0,1,1,1,1
```

Logistic Regression

SciKit-Learn's Logistic Regression model will perform multinomial classification automatically.

Create an sklearn `LogisticRegression()` class and train this model on the dataset

Details about how to use this are here: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

In [15]:
```
from sklearn.linear_model import LogisticRegression

def get_logistic_regressor(features, classes):
    # YOUR CODE GOES HERE
    reg = LogisticRegression(multi_class='multinomial')  # - Instantiate model with re
    reg.fit(features, classes)                           # - Fit model to data
    return reg
```

In [16]:
```
model = get_logistic_regressor(xy, c)
preds = model.predict(xy)
accuracy = np.sum(preds == c) / len(c) * 100
print("True Classes:", c)
print(" Predictions:", preds)
print("    Accuracy:", accuracy, r"%")
```

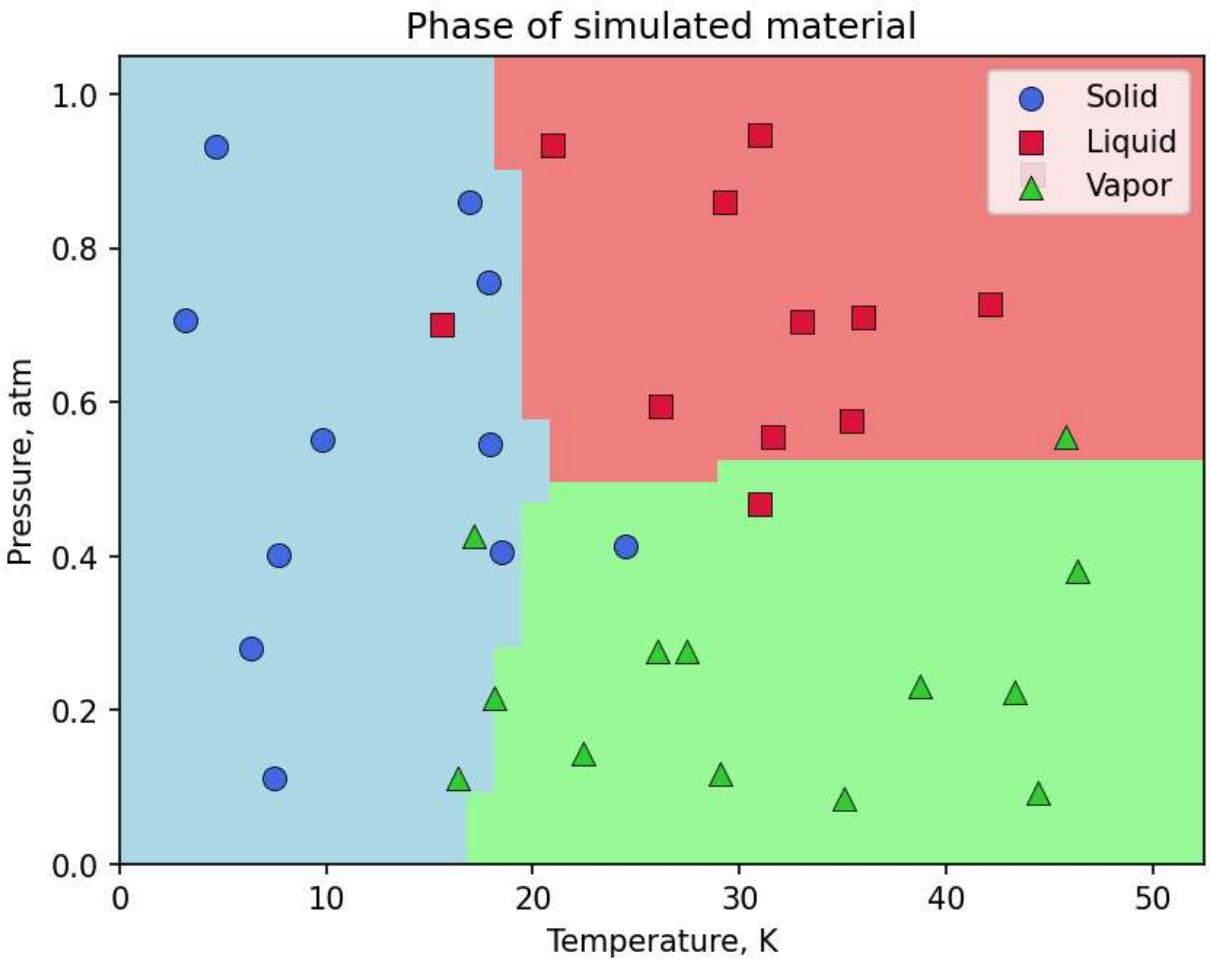
```
True Classes: [0 2 2 2 2 2 0 2 2 2 2 2 0 0 2 0 1 2 0 0 1 1 1 2 0 1 0 1 1 1 0 0 1 1 1
1]
 Predictions: [0 0 2 2 2 2 0 0 2 2 2 2 0 0 0 2 2 2 0 0 1 1 1 1 0 0 0 1 1 1 0 0 1 1 1
1]
    Accuracy: 80.55555555555556 %
```

Plotting Multinomial Classifier Results

Here, we have made some plotting functions -- run these cells to visualize the decision boundaries.

In [17]:
```
def plot_data(x, y, c, title="Phase of simulated material", newfig=True):
    xlim = [0,52.5]
    ylim = [0,1.05]
    markers = [dict(marker="o", color="royalblue"), dict(marker="s", color="crimson"),
    labels = ["Solid", "Liquid", "Vapor"]

    if newfig:
        plt.figure(dpi=150)

    for i in range(1+max(c)):
        plt.scatter(x[c==i], y[c==i], s=60, **(markers[i]), edgecolor="black", linewid

    plt.title(title)
    plt.legend(loc="upper right")
    plt.xlim(xlim)
    plt.ylim(ylim)
    plt.xlabel("Temperature, K")
    plt.ylabel("Pressure, atm")
    plt.box(True)

def plot_sklearn_colors(model, res=40):
    xlim = [0,52.5]
    ylim = [0,1.05]
    xvals = np.linspace(*xlim,res)
    yvals = np.linspace(*ylim,res)
    x,y = np.meshgrid(xvals,yvals)
    XY = np.concatenate((x.reshape(-1,1),y.reshape(-1,1)),axis=1)

    color = model.predict(XY).reshape(res,res)

    cmap = ListedColormap(["lightblue","lightcoral","palegreen"])
    plt.pcolor(x, y, color, shading="nearest", zorder=-1, cmap=cmap,vmin=0,vmax=2)
    return
```

In [18]:
```
plot_data(x,y,c)
plot_sklearn_colors(model)
plt.show()
```


Phase of simulated material
Solid
Liquid
Vapor
Pressure, atm
Temperature, K
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40
50

Problem 7 (20 points)

Problem Description

A projectile is launched with input x- and y-velocity components. A dataset is provided, which contains launch velocity components as input and whether a target was hit (0/1) as an output. This data has a nonlinear decision boundary.

You will use gradient descent to train a logistic regression model on the dataset to predict whether any given launch velocity will hit the target.

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

You are welcome to use any of the code provided in the previous problems.

Summary of deliverables:

Functions (described in later section)

- `sigmoid(h)`
- `map_features(data)`
- `loss(data,y,w)`
- `grad_loss(data,y,w)`
- `grad_desc(data, y, w0, iterations, stepsize)`

Results:

- Print final `w` after training on the training data
- Plot of loss throughout training
- Print model percent classification accuracy on the training data
- Print model percent classification accuracy on the testing data
- Plot that shows the training data as data points, along with a decision boundary

Imports and Utility Functions:

In [106…

```
import numpy as np
import matplotlib.pyplot as plt

def plot_data(data, c, title="", xlabel="$x_1$",ylabel="$x_2$",classes=["",""],alpha=1
    N = len(c)
    colors = ['royalblue','crimson']
    symbols = ['o','s']

    plt.figure(figsize=(5,5),dpi=120)

    for i in range(2):
        x = data[:,0][c==i]
        y = data[:,1][c==i]
```

```
        plt.scatter(x,y,color=colors[i],marker=symbols[i],edgecolor="black",linewidths

    plt.legend(loc="upper right")
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    ax = plt.gca()
    plt.xlim([-0.05,1.05])
    plt.ylim([-0.05,1.05])
    plt.title(title)

def plot_contour(w):
    res = 500
    vals = np.linspace(-0.05,1.05,res)
    x,y = np.meshgrid(vals,vals)
    XY = np.concatenate((x.reshape(-1,1),y.reshape(-1,1)),axis=1)
    prob = sigmoid(map_features(XY) @ w.reshape(-1,1))
    pred = np.round(prob.reshape(res, res))
    plt.contour(x, y, pred)
```

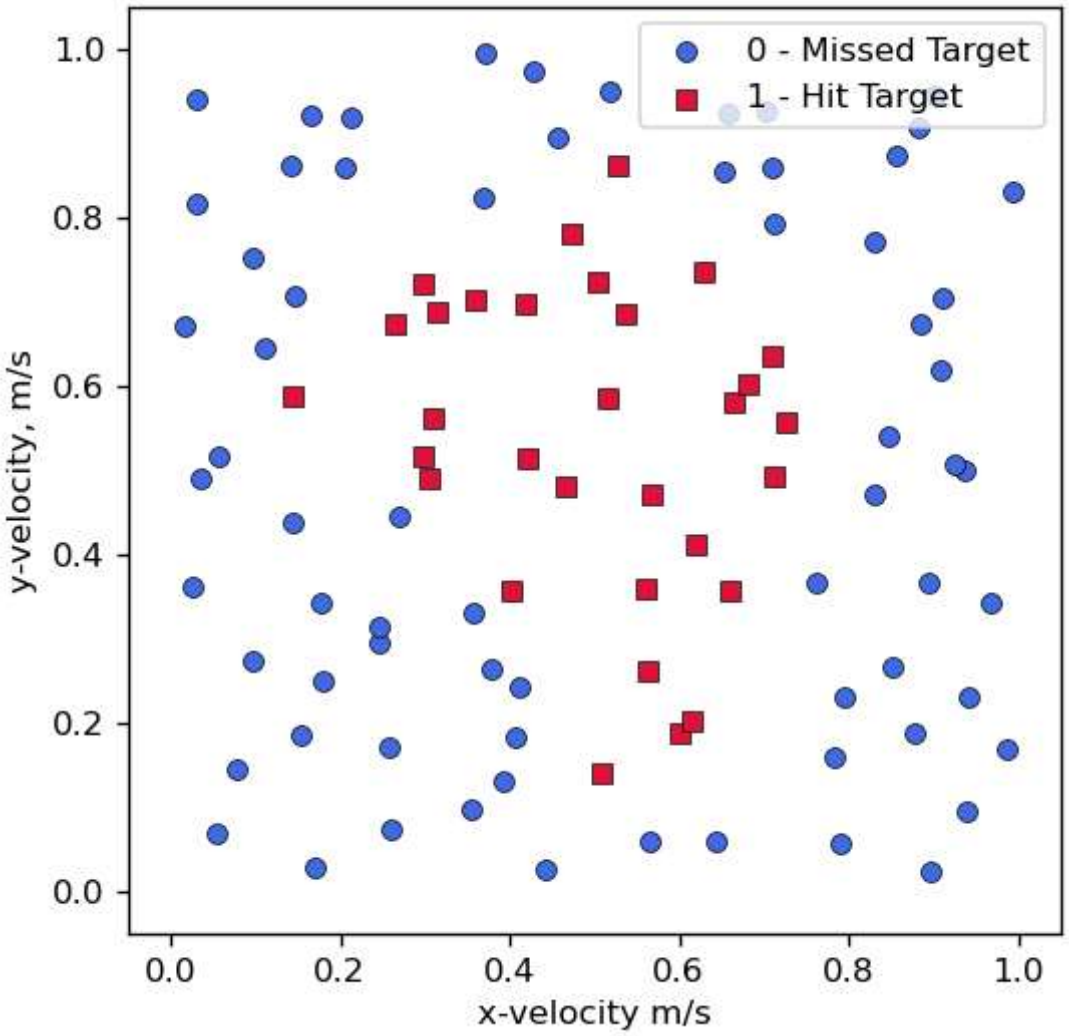
Load Data

This cell loads the dataset into the following variables:

- `train_data` : Nx2 array of input features, used for training
- `train_gt` : Array of ground-truth classes for each point in `train_data`
- `test_data` : Nx2 array of input features, used for testing
- `test_gt` : Array of ground-truth classes for each point in `test_data`

In [107…

```
train = np.load("w3-hw1-data-train.npy")
test = np.load("w3-hw1-data-test.npy")
train_data, train_gt = train[:,:2], train[:,2]
test_data, test_gt = test[:,:2], test[:,2]
format = dict(xlabel="x-velocity m/s", ylabel="y-velocity, m/s", classes=["0 - Missed
plot_data(train_data, train_gt, **format)
```



Helper Functions

Here, implement the following functions:

`sigmoid(h)`:

- Input: `h`, single value or array of values
- Returns: The sigmoid of h (or each value in h)

`map_features(data)`:

- Input: `data`, Nx2 array with rows (x_i, y_i)
- Returns: Nx45 array, each row with $(1, x_i,\ y_i,\ x_i^2,\ x_i y_i,\ y_i^2,\ x_i^3,\ x_i^2 y_i,\ \ldots)$ with all terms through 8th-order

`loss(data, y, w)`:

- Input: `data`, Nx2 array of un-transformed input features
- Input: `y`, Ground truth class for each input
- Input: `w`, Array with 45 weights

- Returns: Loss: $L(x, y, w) = \sum_{i=1}^{n} -y^{(i)} \cdot \ln(g(w'x^{(i)})) - (1 - y^{(i)}) \cdot \ln(1 - g(w'x^{(i)}))$

`grad_loss(data, y, w)`:

- Input: `data`, Nx2 array of un-transformed input features
- Input: `y`, Ground truth class for each input
- Input: `w`, Array with 45 weights
- Returns: Gradient of loss with respect to weights: $\frac{\partial L}{w_j} = \sum_{i=1}^{n}(g(w'x^{(i)}) - y^{(i)})x_j^{(i)}$

In [108…

```
# YOUR CODE GOES HERE
def sigmoid(h):
    return 1/(1+np.exp(-h))

def map_features(data):     # Data is Nx2 array
    deg=8
    n=np.ones((data.shape[0],1))
    for i in range(1,deg+1):
        for j in range(i+1):
            t1=data[:,0]**(i-j)
            t2=data[:,1]**j
            t=(t1*t2).reshape(data.shape[0],1)
            n=np.hstack((n,t))
    return n

def loss(data,y,w):
    mapp=map_features(data)
    pred=sigmoid(np.dot(mapp, w))               # Dot product between mapp and w
    a=y*np.log(pred)
    b=(1 - y)*np.log(1-pred)
    loss=(-a)-b
    L=np.sum(loss)/len(y)
    return L

def grad_loss(data,y,w):
    mapp=map_features(data)
    pred=sigmoid(np.dot(mapp, w))              # Dot product between mapp and w
    g=np.dot(mapp.T, (pred - y))
    grad=g/len(y)
    return grad
```

Gradient Descent

Now, write a gradient descent function with the following specifications:

`grad_desc(data, y, w0, iterations, stepsize)`:

- Input: `data`, Nx2 array of un-transformed input features
- Input: `y`, array of size N with ground-truth class for each input
- Input: `w0`, array of weights to use as an initial guess (size)
- Input `iterations`, number of iterations of gradient descent to perform
- Input: `stepsize`, size of each gradient descent step
- Return: Final `w` array after last iteration

- Return: Array containing loss values at each iteration

In [109…
```
# YOUR CODE GOES HERE
def grad_desc(data,y,w0,iterations,stepsize):
    w=w0
    loss_val=[]
    for _ in range(iterations):
        grad=grad_loss(data,y,w)
        w=w-stepsize*grad
        loss_val.append(loss(data,y,w))
    return w,loss_val
```

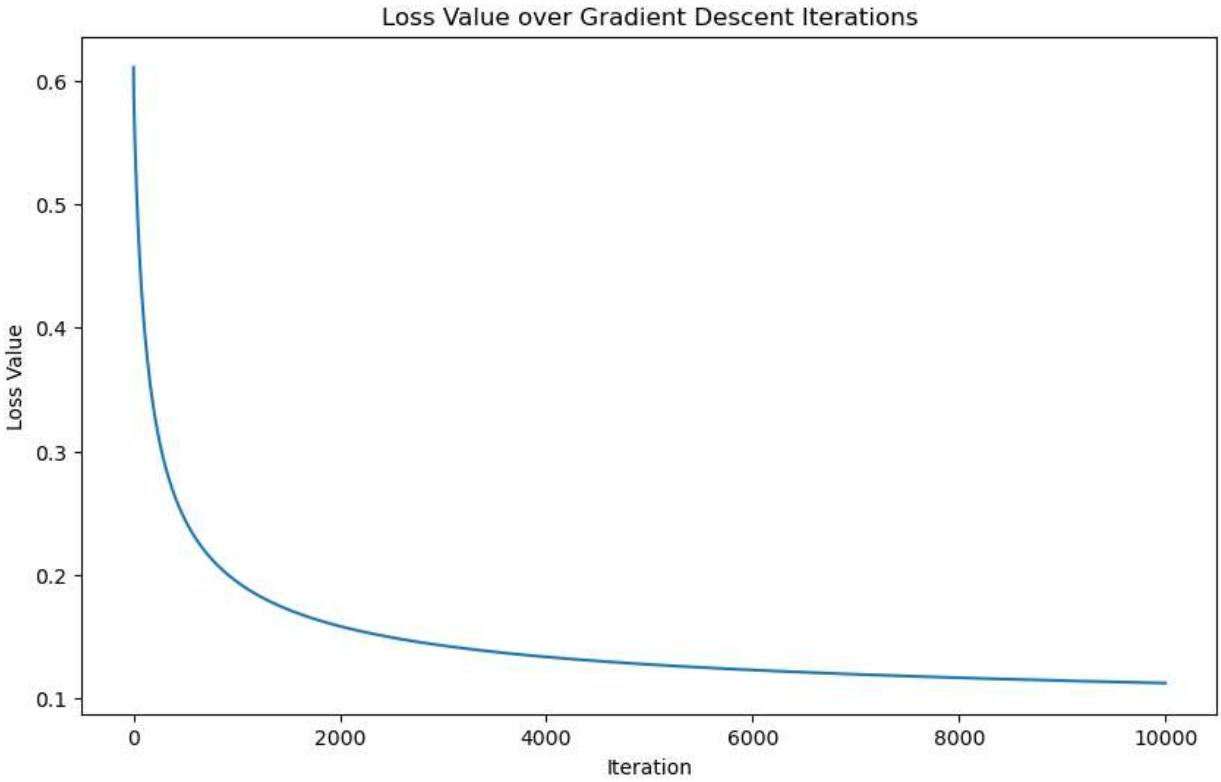
Training

Run your gradient descent function and plot the loss as it converges. You may have to tune the step size and iteration count.

Also print the final vector `w`.

In [155…
```
# YOUR CODE GOES HERE (training)
w0=np.zeros((45, 1))
iterations=10000
stepsize=1

w,losses=grad_desc(train_data,train_gt.reshape(-1, 1),w0,iterations,stepsize)
```

In [156…
```
# YOUR CODE GOES HERE (loss plot, print w)
# Plotting loss values
plt.figure(figsize=(10,6))
plt.plot(losses)
plt.xlabel('Iteration')
plt.ylabel('Loss Value')
plt.title('Loss Value over Gradient Descent Iterations')
plt.show()

print("The final w vector values are =\n", w)
```


Loss Value over Gradient Descent Iterations
Loss Value
0.6
0.5
0.4
0.3
0.2
0.1
0
2000
4000
6000
8000
10000
Iteration

```
The final w vector values are =
 [[-12.21871464]
 [ 17.20342511]
 [ 11.27052278]
 [  7.02448207]
 [  9.69991315]
 [  6.04638802]
 [ -2.34793777]
 [  1.32583895]
 [  3.80357027]
 [ -0.03426935]
 [ -7.6652794 ]
 [ -3.15598135]
 [ -1.0800548 ]
 [  0.7425595 ]
 [ -4.17503571]
 [ -9.85519735]
 [ -5.02572657]
 [ -3.23014815]
 [ -1.77583154]
 [ -0.96387174]
 [ -6.29424337]
 [-10.16644293]
 [ -5.45870365]
 [ -3.84939362]
 [ -2.83821177]
 [ -1.98700554]
 [ -1.8042255 ]
 [ -6.98434394]
 [ -9.51076143]
 [ -5.1811598 ]
 [ -3.73842947]
 [ -3.0106039 ]
 [ -2.4592646 ]
 [ -1.96380234]
 [ -2.09719495]
 [ -6.83473434]
 [ -8.44658482]
 [ -4.60559766]
 [ -3.32434116]
 [ -2.76875471]
 [ -2.43561618]
 [ -2.11436961]
 [ -1.80727891]
 [ -2.07787314]
 [ -6.26325476]]
```

Accuracy

Compute the accuracy of the model, as a percent, for both the training data and testing data

In [157…

```
# YOUR CODE GOES HERE
def accuracy(data,y,w):
    mapp=map_features(data)
    pred=sigmoid(np.dot(mapp,w))
    pred_class=np.round(pred)
    Y=y.reshape(-1, 1)
    c=np.sum(pred_class==Y)
```

```
    x=(c/len(y))*100
    return x

train_accuracy = accuracy(train_data, train_gt, w)
test_accuracy = accuracy(test_data, test_gt, w)
print(f"Accuracy for the training data is = {train_accuracy:.2f}%")
print(f"Accuracy for the testing data is = {test_accuracy:.2f}%")
```

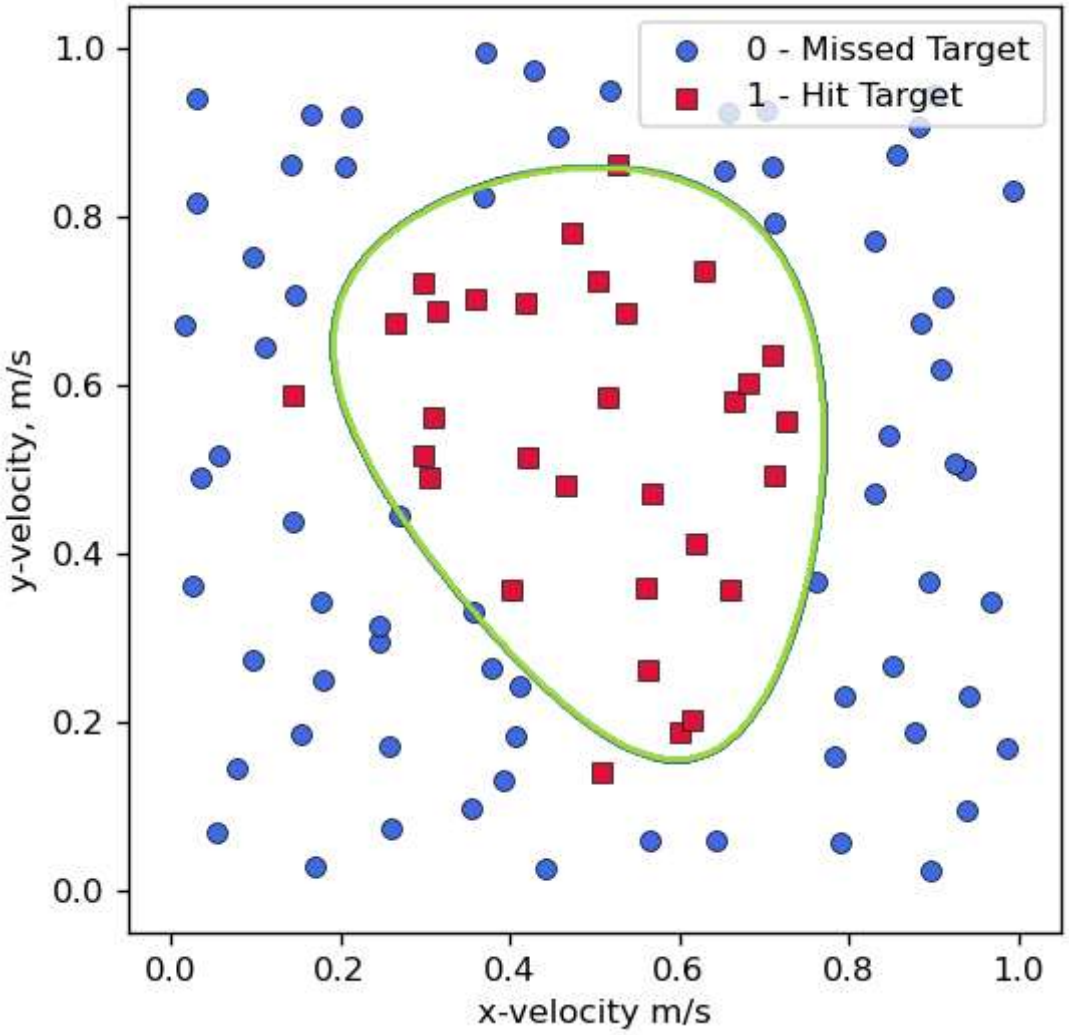
```
Accuracy for the training data is = 96.00%
Accuracy for the testing data is = 96.00%
```

Visualize Results

Use the provided plotting utilities to plot the decision boundary with the data.

In [159…

```
# You may have to modify this code, i.e. if you named 'w' differently)
plot_data(train_data, train_gt, **format)
plot_contour(w)
```



In [ ]:

In [ ]:

Problem 8 (20 Points)

Problem Description

Several molecular dynamics simulations have been carried out for a material, and the phase (solid/liquid/vapor) at different temperature/pressure combinations has been recorded.

You will use gradient descent to train a One-vs-Rest logistic regression model on data with 3 classes. Fill out the notebook as instructed, making the requested plots and printing necessary values.

You are welcome to use any of the code provided in the previous problems.

Summary of deliverables:

- 3 binomial classification `w` vectors, corresponding to each class
- Function `classify(xy)` that evaluates all 3 models at a given array of points, returning the class prediction as the model with the highest probability
- Print model percent classification accuracy on the training data
- Print model percent classification accuracy on the testing data
- Plot that shows the training data as data points, along with the class of a grid of points in the background, as in the lecture activity.

Imports and Utility Functions:

In [9]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

def plot_data(x, y, c,title="Phase of simulated material"):
    xlim = [0,52.5]
    ylim = [0,1.05]
    markers = [dict(marker="o", color="royalblue"), dict(marker="s", color="crimson"),
    labels = ["Solid", "Liquid", "Vapor"]

    plt.figure(dpi=150)

    for i in range(1+max(c)):
        plt.scatter(x[c==i], y[c==i], s=60, **(markers[i]), edgecolor="black", linewid

    plt.title(title)
    plt.legend(loc="upper right")
    plt.xlim(xlim)
    plt.ylim(ylim)
    plt.xlabel("Temperature, K")
    plt.ylabel("Pressure, atm")
    plt.box(True)

def plot_colors(classify, res=40):
```

```
    xlim = [0,52.5]
    ylim = [0,1.05]
    xvals = np.linspace(*xlim,res)
    yvals = np.linspace(*ylim,res)
    x,y = np.meshgrid(xvals,yvals)
    XY = np.concatenate((x.reshape(-1,1),y.reshape(-1,1)),axis=1)

    color = classify(XY).reshape(res,res)

    cmap = ListedColormap(["lightblue","lightcoral","palegreen"])
    plt.pcolor(x, y, color, shading="nearest", zorder=-1, cmap=cmap,vmin=0,vmax=2)
    return
```

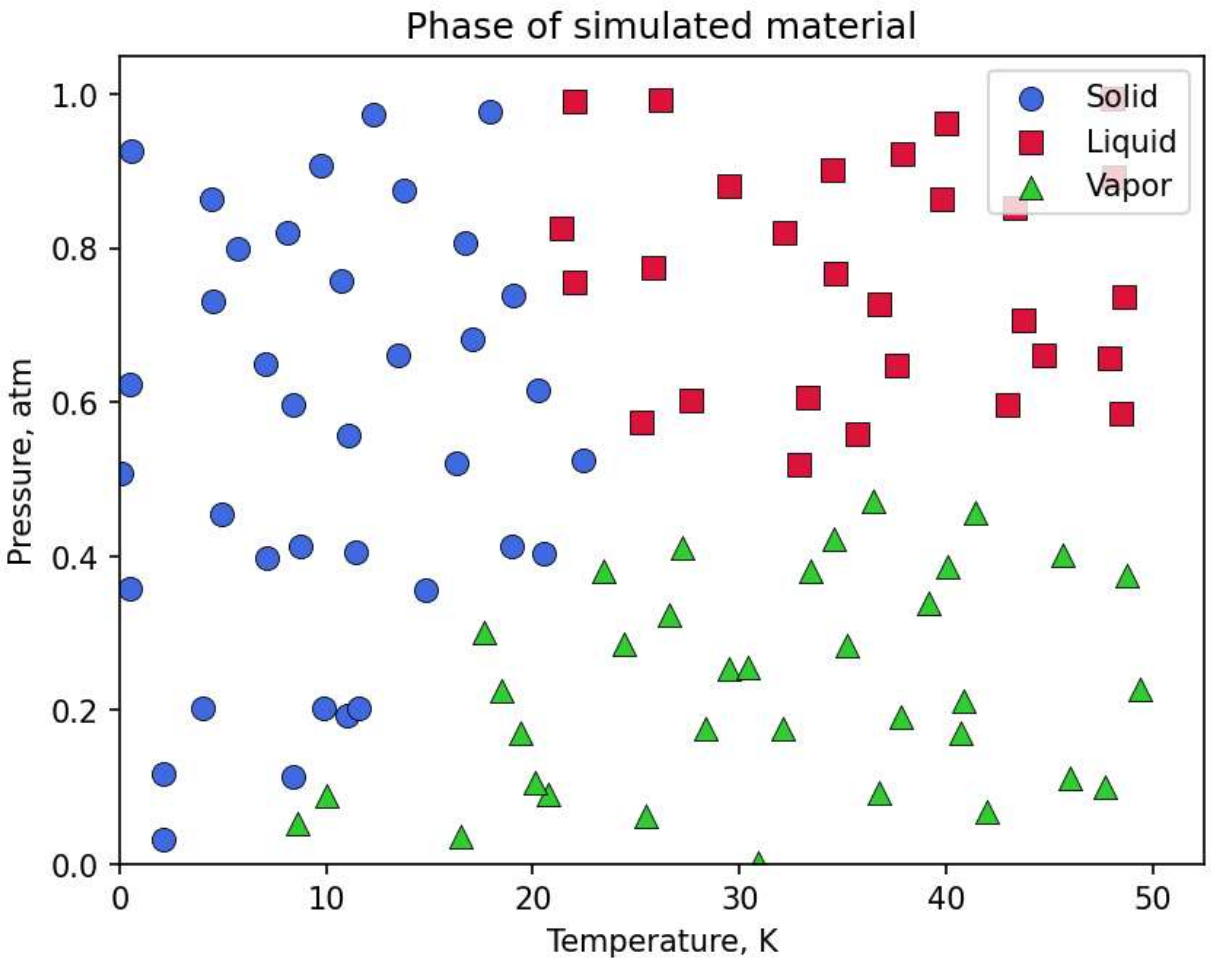
Load Data

This cell loads the dataset into the following variables:

- `train_data` : Nx2 array of input features, used for training
- `train_gt` : Array of ground-truth classes for each point in `train_data`
- `test_data` : Nx2 array of input features, used for testing
- `test_gt` : Array of ground-truth classes for each point in `test_data`

In the class arrays, 0 = solid, 1 = liquid, 2 = vapor.

In [10]:
```
train = np.load("w3-hw2-data-train.npy")
test = np.load("w3-hw2-data-test.npy")
train_data, train_gt = train[:,:2], train[:,2].astype(int)
test_data, test_gt = test[:,:2], test[:,2].astype(int)
plot_data(train_data[:,0], train_data[:,1],train_gt)
```



Gradient Descent

Here, write all of the necessary code to perform gradient descent and train 3 logistic regression models for a 1-vs-rest scenario. Use linear decision boundaries (features should only be 1, temperature, pressure)

Feel free to reuse code from the first problem or lecture activities.

We have provided the following function to help with the one-vs-all method:

`convert_to_binary_dataset(classes, A)`:

- Input: data, Nx2 array of temperature-pressure data
- Input: classes, array (size N) of class values for each point in `data`
- Input: A, the class (0, 1, or 2 here) to use as '1' in the binary dataset
- Returns: `classes_binary`, copy of classes where class A is 1, and all other classes are 0.

In [14]:
```
import numpy as np
def convert_to_binary_dataset(classes, A):
    classes_binary=(classes==A).astype(int)
    return classes_binary
```

In [32]:
```
# YOUR CODE GOES HERE (gradient descent and related functions)

def sigmoid(h):
    return 1/(1+np.exp(-h))

def transform(data, w):
    xs=data[:,0]
    ys=data[:,1]
    ones=np.ones_like(xs)
    t=w[1]*xs+w[2]*ys+w[0]*ones
    return t

def loss(data,y,w):
    wt=transform(data,w)
    stability=1e-10
    a=-np.log(sigmoid(wt))*y
    b=-np.log(1-sigmoid(wt)+stability)*(1 - y)
    loss=np.sum(a+b)/len(y)
    return loss

def gradloss(data, y, w):
    wt=transform(data,w)
    wty=sigmoid(wt)-y
    xs=data[:,0]
    ys=data[:,1]
    ones=np.ones_like(xs)
    w0=np.sum(wty*ones)/len(y)
    w1=np.sum(wty*xs)/len(y)
    w2=np.sum(wty*ys)/len(y)
    grad=np.array([w0,w1,w2])
    return grad

def grad_desc(data,y,w0=np.array([0,0,0]),iterations=1000,stepsize=0.01):
    w=w0
    loss_val=np.zeros(iterations)
    for i in range(iterations):
        grad=gradloss(data,y,w)
        w=w-stepsize*grad
        loss_val[i]=loss(data,y,w)
    return w
```

Training

Train your 3 models and print the `w` vector corresponding to each class

In [67]:
```
# YOUR CODE GOES HERE (training)
def convert_to_binary_dataset(y,A):
    return np.where(y==A,1,0)

def train(x_t,y_t):
    w_c=[]
    for A in range(3):
        y_bin=convert_to_binary_dataset(y_t,A)
        w_A=grad_desc(x_t,y_bin)
        w_c.append(w_A)
    return w_c
```

```
w_c = train(train_data,train_gt)

# YOUR CODE GOES HERE (print "w"s)
w_c_str = '\n'.join([f"The corresponding w vector for class {i}: {w}" for i, w in enum
print(w_c_str)
```

```
The corresponding w vector for class 0: [ 1.07205218 -0.11179952  0.69629142]
The corresponding w vector for class 1: [-1.13562268  0.02606585  0.16219895]
The corresponding w vector for class 2: [-0.66770194  0.03106601 -1.15509146]
```

Classification function

Write a function `classify(xy)` that will evaluate each model and select the appropriate class.

In [70]:
```
def classify(xy):
    # Add bias column to the input data
    xy=np.hstack([np.ones((xy.shape[0],1)),xy])
    prob=[sigmoid(np.dot(xy, w_A)) for w_A in w_c]
    return np.argmax(prob,axis=0)
```

Accuracy

Compute and print the accuracy on the training and testing sets as a percent

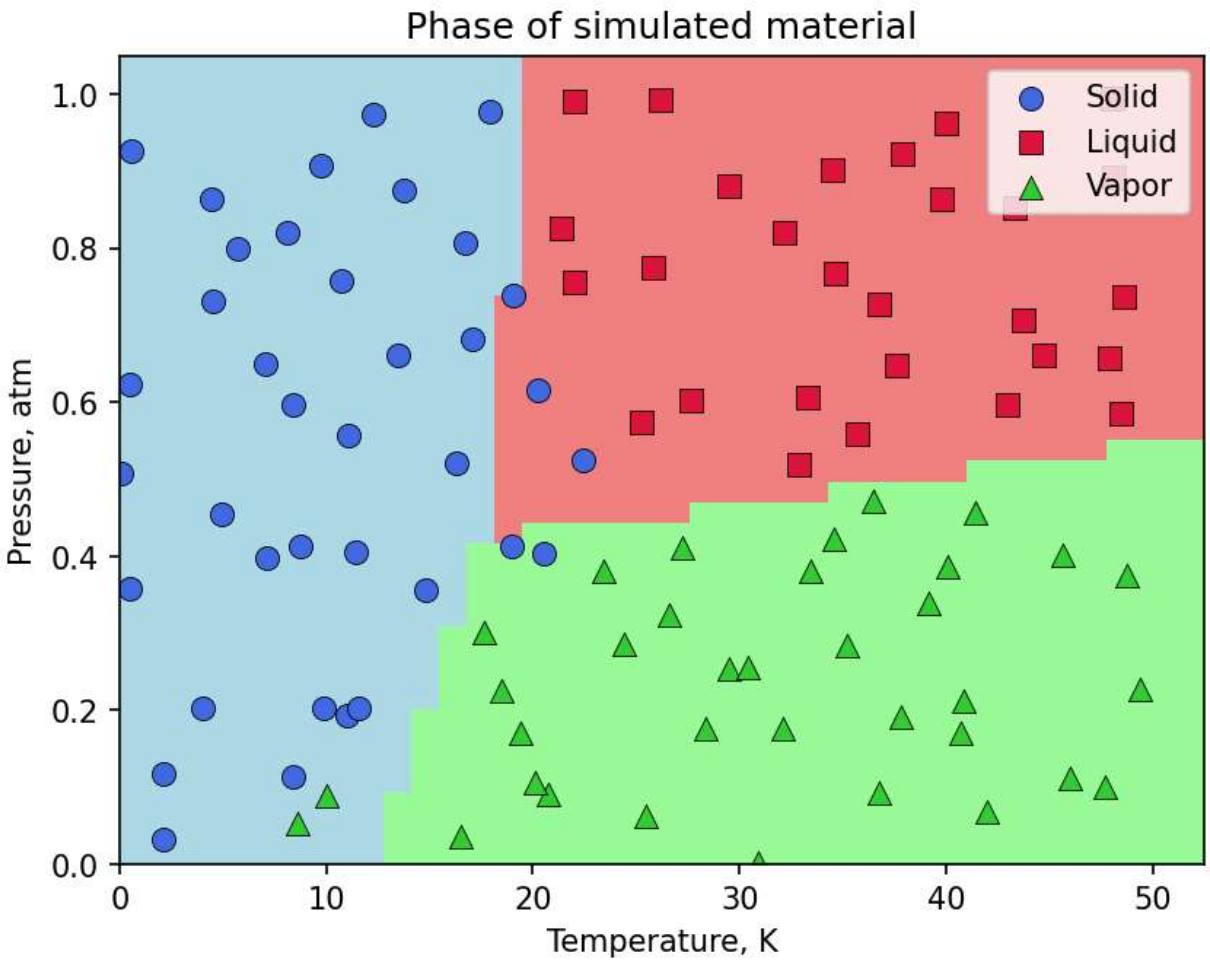
In [75]:
```
# YOUR CODE GOES HERE (accuracy)
def accuracy(data,gt):
    preds=classify(data)
    return np.mean(preds==gt)*100

print(f'The Training Accuracy is = {accuracy(train_data, train_gt):.2f}%')
print(f'The Testing Accuracy is = {accuracy(test_data, test_gt):.2f}%')
```

```
The Training Accuracy is = 93.00%
The Testing Accuracy is = 88.00%
```

Plot results

Run this cell to visualize the data along with the results of `classify()`

In [76]:
```
plot_data(train_data[:,0], train_data[:,1], train_gt)
plot_colors(classify)
```


Phase of simulated material
Solid
Liquid
Vapor
Pressure, atm
Temperature, K
0.0
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40
50

Problem 9 (20 Points)

Problem description

So far, we have worked with ~2 dimensional problems with 2-3 classes. Most often in ML, there are many more explanatory variables and classes than this. In this problem, you'll be training logistic regression models on a database of grayscale images of hand-drawn digits, using SciKit-Learn. Now there are 400 (20x20) input features and 10 classes (digits 0-9).

As usual, you can use any code from previous problems.

Summary of deliverables

- OvR model accuracy on training data
- OvR model accuracy on testing data
- Multinomial model accuracy on training data
- Multinomial model accuracy on testing data

Imports and Utility Functions:

In [11]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression

def visualize(xdata, index, title=""):
    image = xdata[index,:].reshape(20,20).T
    plt.figure()
    plt.imshow(image,cmap = "binary")
    plt.axis("off")
    plt.title(title)
    plt.show()
```

Load data

The following cell loads in training and testing data into the following variables:

- `x_train` : 4000x400 array of input features, used for training
- `y_train` : Array of ground-truth classes for each point in `x_train`
- `x_test` : 1000x400 array of input features, used for testing
- `y_test` : Array of ground-truth classes for each point in `x_test`

You can visualize a digit with the `visualize(x_data, index)` function.

In [12]:
```
x_train = np.load("w3-hw3-train_x.npy")
y_train = np.load("w3-hw3-train_y.npy")
```

```
x_test = np.load("w3-hw3-test_x.npy")
y_test = np.load("w3-hw3-test_y.npy")

visualize(x_train,1234)
```

Logistic Regression Models

Use sklearn's `LogisticRegression` to fit a multinomial logistic regression model on the training data. You may need to increase the `max_iter` argument for the model to converge.

Train 2 models: one using the One-vs-Rest method, and another that minimizes multinomial loss. You can do these by setting the `multi_class` argument to "ovr" and "multinomial", respectively.

More information: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

In [35]:
```
# YOUR CODE GOES HERE (sklearn models)
# Train OvR model
ovr=LogisticRegression(multi_class="ovr",max_iter=1000)
ovr.fit(x_train,y_train)

# Train Multinomial model
multinom=LogisticRegression(multi_class="multinomial", solver="lbfgs", max_iter=1000)
multinom.fit(x_train,y_train)
```

Out[35]:
```
LogisticRegression
LogisticRegression(max_iter=10000, multi_class='multinomial')
```

Accuracy

Compute and print the accuracy of each model on the training and testing sets as a percent.

In [36]:
```
# YOUR CODE GOES HERE (print the 4 requested accuracy values)
def accuracy(model,x,y):
    pred=model.predict(x)
    c=np.sum(pred==y)
    p=(c/len(y))*100
    return p

ovr_train=accuracy(ovr,x_train,y_train)                    # OvR model - train data
print(f"Accuracy of the OvR model in training data is = {ovr_train:.2f}%")
ovr_test=accuracy(ovr,x_test,y_test)                       # OvR model - test data
print(f"Accuracy of the OvR model in testing data is = {ovr_test:.2f}%")

multinom_train=accuracy(multinom,x_train,y_train)          # Multinomial - train data
print(f"Accuracy of the Multinomial model in training data is = {multinom_train:.2f}%"
multinom_test=accuracy(multinom,x_test,y_test)             # Multinomial - test data
print(f"Accuracy of the Multinomial model in testing data is = {multinom_test:.2f}%")
```

```
Accuracy of the OvR model in training data is = 94.73%
Accuracy of the OvR model in testing data is = 90.70%
Accuracy of the Multinomial model in training data is = 96.47%
Accuracy of the Multinomial model in testing data is = 91.40%
```