Problem 1

1. False
2. True
3. True

Problem 2

1. The values of h can range from -∞ to ∞
2. g(h) ranges from 0 to 1 (open set)

Problem 3

False. Too much regularization can cause the model complexity to be compromised too much, resulting in poor classification.

Problem 4

Class D. This is because class D was predicted the most number of times, making it the most likely class.

Problem 5

3 classes

Problem 6

Model 1. This is because model 1 offers the lower cross entropy amongst the two classes.

M3-L1 Problem 1 (6 points)

In [1]:
```
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 [2]:
```
def sigmoid(h):
    # YOUR CODE GOES HERE
    sig = 1 / (1 + np.exp(-h))
    return sig
```

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 [5]:
```
def transform_quadratic(x, w):
    # YOUR CODE GOES HERE
    sum = np.dot(x, w)
    return sum
```

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 [6]:
```
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) = [0.99183743 0.02659699 0.9168273 ]
x = 7.0 --> P(y=1) = [1.00000000e+00 7.58256042e-10 9.99999168e-01]
```

M3-L1 Problem 2 (6 points)

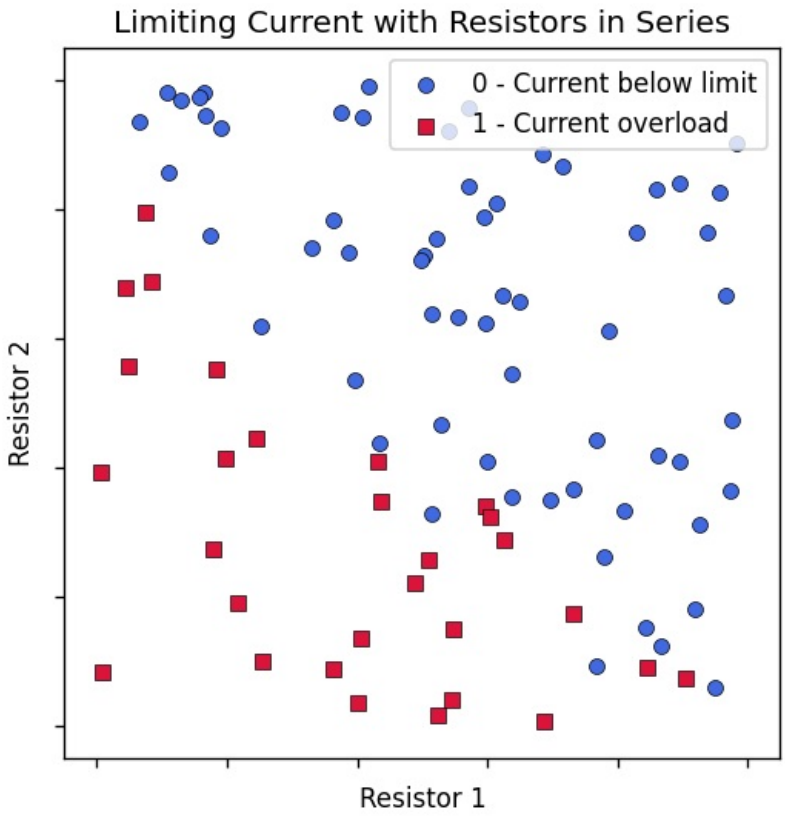
In [40]:
```
import numpy as np
import matplotlib.pyplot as plt


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=0.4,label=classes[i],alp

    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 [41]:
```
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 [42]:
```
data, classes = get_line_dataset()
format = dict(title="Limiting Current with Resistors in Series", xlabel="Resistor 1", ylabel="Resistor 2", clas

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}{w_j} = \sum_{i=1}^n (g(w'x^{(i)}) - y^{(i)}) x_j^{(i)}

In [44]:
```
def sigmoid(h):
    # YOUR CODE GOES HERE
    sig = 1/(1 + np.exp(-h))
    return sig

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(1 - sigmoid(wt_x)) * (1-y)
    L = np.sum(J1 + J2)
    return L

def gradloss(data, y, w):
    # YOUR CODE GOES HERE
    grad = 0
    xs = data[:, 0]
    ys = data[:, 1]
    ones = np.ones_like(xs)
    wt_x = transform(data,w)
    grad_xs = np.dot(sigmoid(wt_x) - y, xs)
    grad_ys = np.dot(sigmoid(wt_x) - y, ys)
    grad_ones = np.dot(sigmoid(wt_x) - y, ones)
    return np.array([grad_ones, grad_xs, grad_ys])
```

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 [45]:
```
def grad_desc(data, y, w0=np.array([0,0,0]), iterations=100, stepsize=0.1):
    # YOUR CODE GOES HERE
    w = w0
    for i in range(iterations):
        grad = gradloss(data, y, w)
        w = w - stepsize * grad
    return w
```

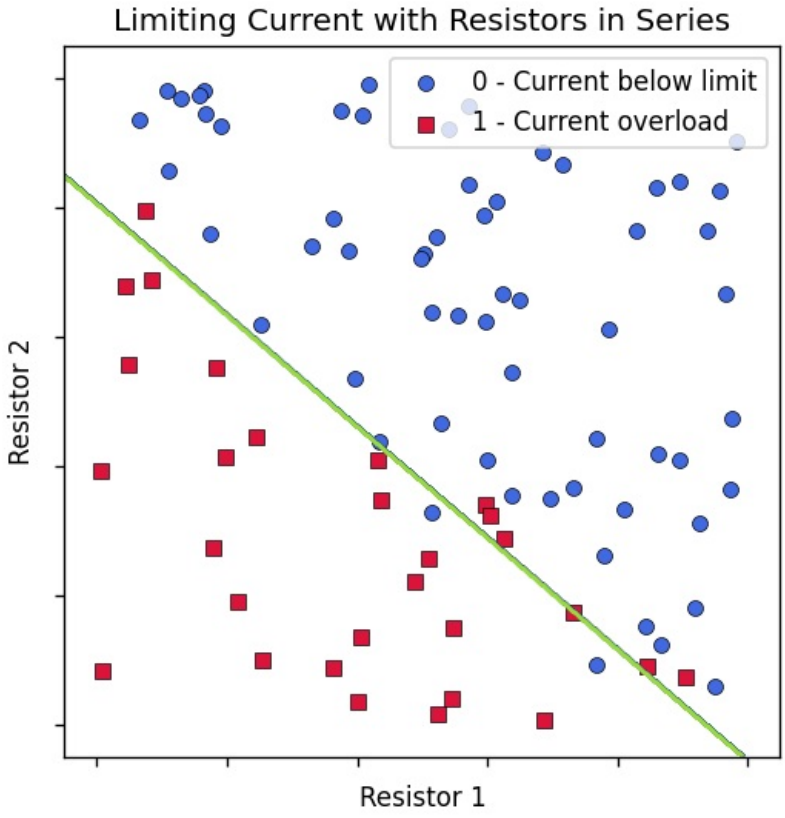
Test your classifier

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

In [46]:
```
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]
 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]
    Accuracy:  91.11111111111111 %
```

In [47]:
```
predict = lambda data: np.round(sigmoid(transform(data, w)))
plot_data(data, classes, **format)
plot_contour(predict)
plt.show()
```



M3-L1 Problem 3 (6 points)

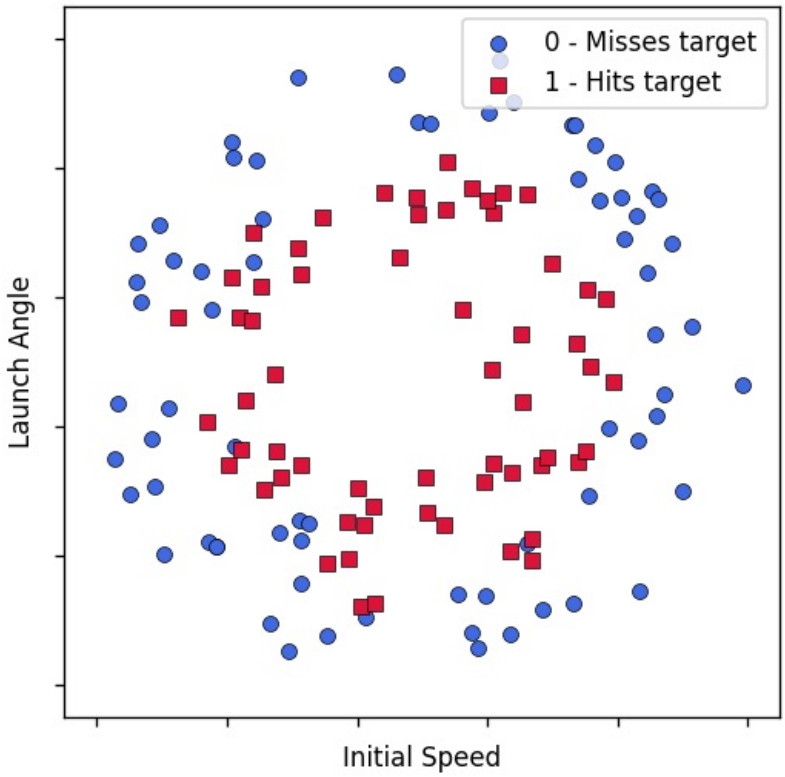
In [24]:
```
import numpy as np
import matplotlib.pyplot as plt


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=0.4,label=classes[i],alpl

    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 [25]:
```
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 [26]:
```
data, classes = get_ring_dataset()
format = dict(xlabel="Initial Speed",ylabel="Launch Angle", classes=["0 - Misses target", "1 - Hits target"])

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 [34]:
```
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, np.square(x1), np.square(x2), np.sin(x1), np.cos(x1), np.sin(x2), np.c
    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

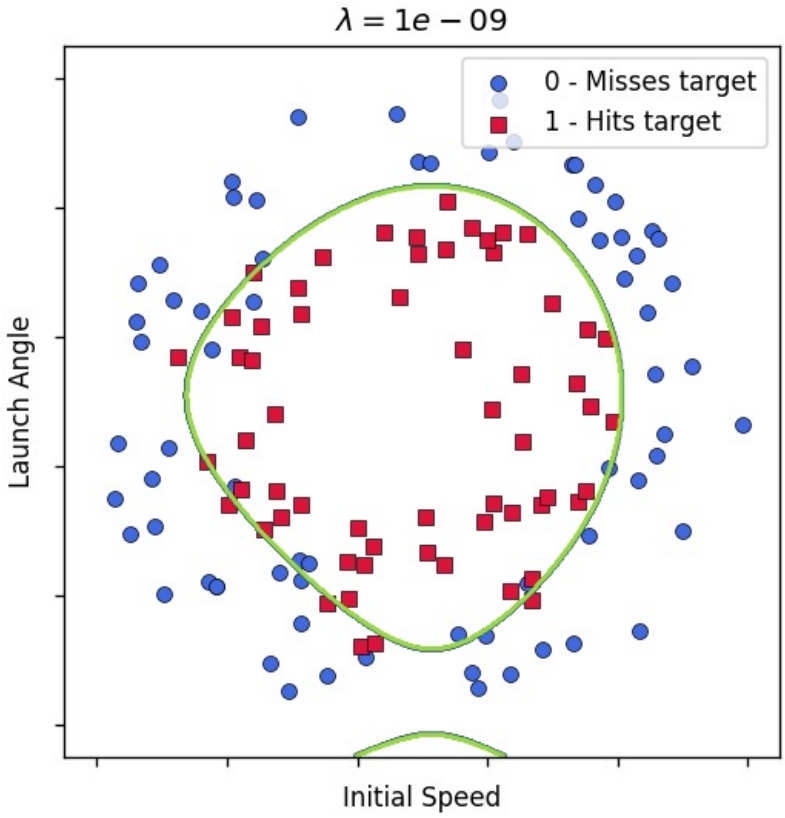
Notes:

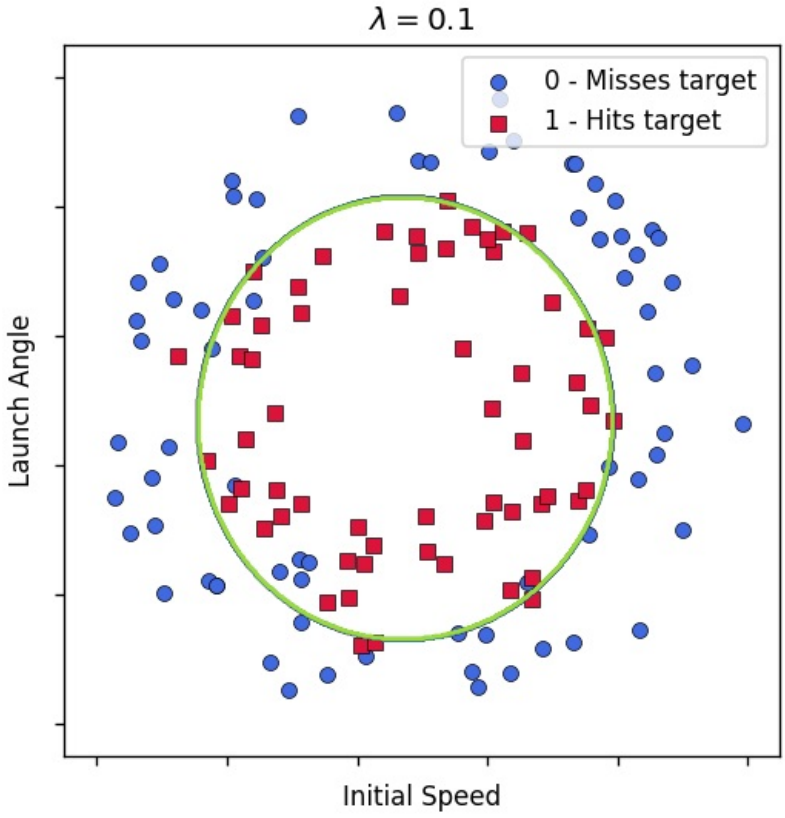
- λ is related to sklearn's regularization strength C by: $\lambda = 1/C$
- You may want to increase the maximum number of iterations

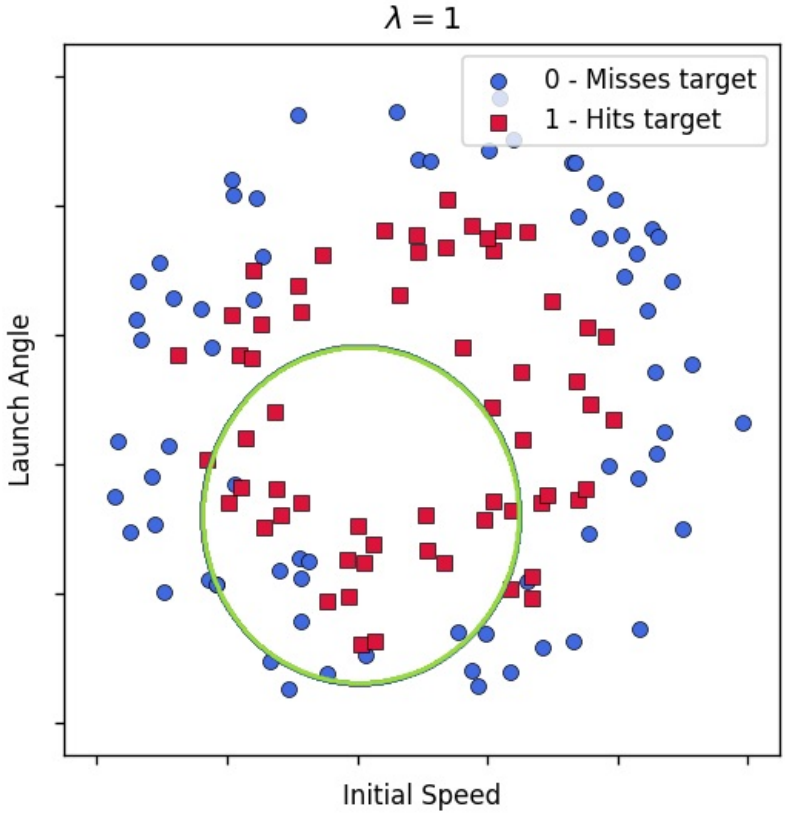
In [37]:
```
from sklearn.linear_model import LogisticRegression

def get_logistic_regressor(features, classes, L = 1):
    # YOUR CODE GOES HERE
    # - Instantiate model with regularization
    model = LogisticRegression(C = 1/L, max_iter = 10000, tol = 1e-6)
    # - Fit model to expanded data
    model.fit(features, classes)

    return model
```

In [38]:
```
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()
```

```
<>:3: SyntaxWarning: invalid escape sequence '\l'
<>:3: SyntaxWarning: invalid escape sequence '\l'
C:\Users\barat\AppData\Local\Temp\ipykernel_7948\1698848802.py:3: SyntaxWarning: invalid escape sequence '\l'
  plot_data(data, classes, **format, title=f"$\lambda={L}$")
```




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



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

As lambda increases up to a certain value, the decision boundary does a better job at seperating the classes. At a very low lambda value, the decision boundary is very complex and wavy. However, overfitting to outliers and noise is reduced due to regularization. However, as lambda increases too much, we see that the model complexity is compromised too much. So much so, that the model is boundary to separate the classes properly.

M3-L2 Problem 1 (6 points)

In [1]:
```
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 [2]:
```
x = np.array([7.4881350392732475,16.351893663724194,22.427633760716436,29.04883182996897,35.03654799338904,44.4
y = np.array([0.11120957227224215,0.1116933996874757,0.14437480785146242,0.11818202991034835,0.0859507900573786
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 [39]:
```
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 [95]:
```
def generate_all_classifiers(xy, c):
    # YOUR CODE GOES HERE
    # Use get_binomial_classifier() to get binomial classifiers for each pair of classes,
```

```
    # and return a list of these classifiers
    classify_01 = get_binomial_classifier(xy, c, 0, 1)
    classify_12 = get_binomial_classifier(xy, c, 1, 2)
    classify_20 = get_binomial_classifier(xy, c, 2, 0)

    return [classify_01, classify_12, classify_20]

def classify_majority(classifiers, xy):
    # YOUR CODE GOES HERE
    preds = np.zeros([len(xy), 3])
    preds[:, 0] = classifiers[0](xy)
    preds[:, 1] = classifiers[1](xy)
    preds[:, 2] = classifiers[2](xy)
    final_pred = np.zeros([len(xy), 1])
    count = np.zeros([3, 1])

    for i in range(len(xy)):
        count[0] = np.count_nonzero(preds[i, :] == 0)
        count[1] = np.count_nonzero(preds[i, :] == 1)
        count[2] = np.count_nonzero(preds[i, :] == 2)
        pred = np.argmax(count)
        final_pred[i] = pred

    return final_pred.reshape([1, len(xy)])
```

Trying out our multinomial classifier:

In [96]:
```
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"%")
```

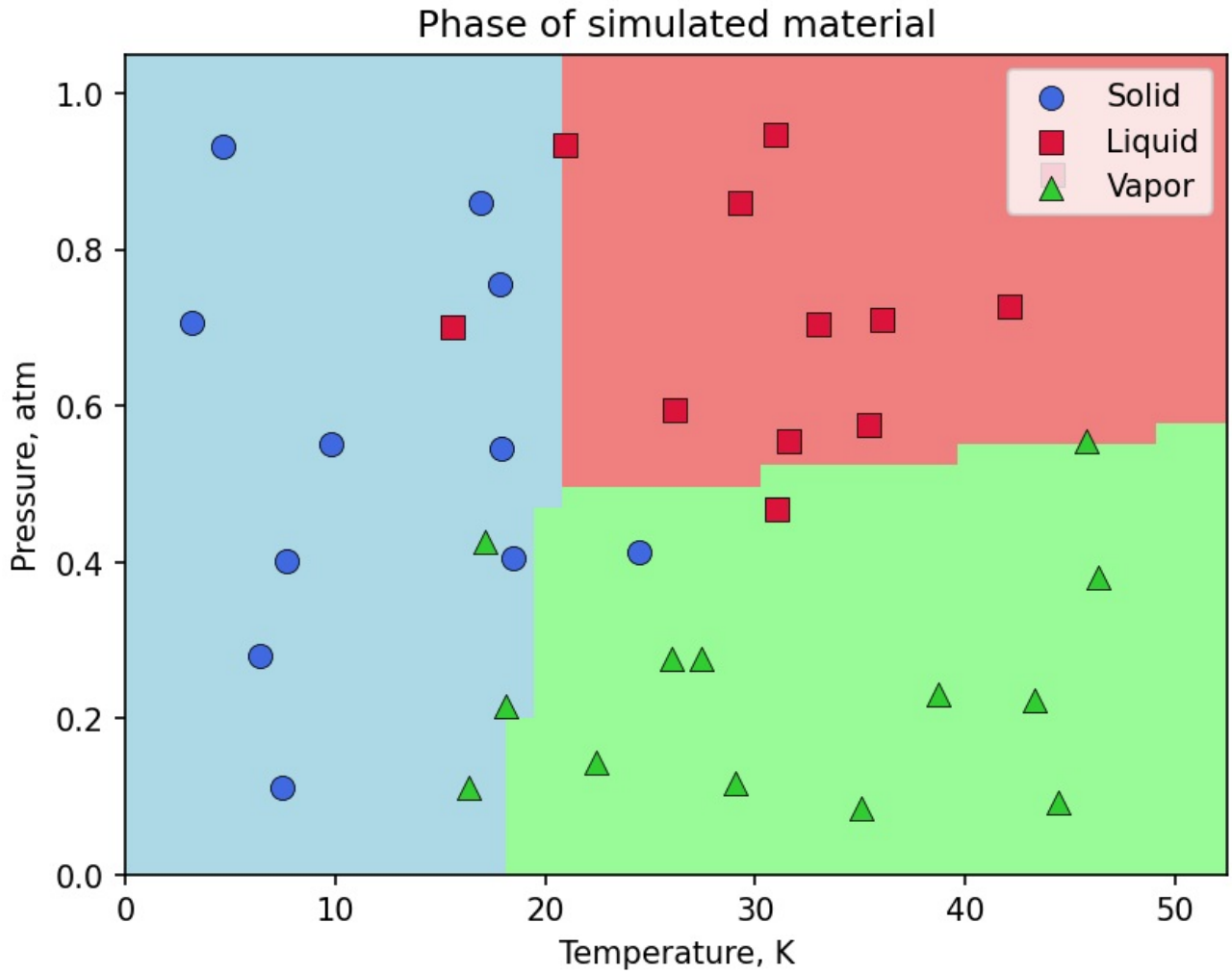
```
True Classes: [0 2 2 2 2 2 0 2 2 2 2 2 0 0 2 0 1 2 0 0 1 1 1 2 0 1 0 1 1 1 0 0 1 1 1 1]
 Predictions: [[0. 0. 2. 2. 2. 2. 0. 0. 2. 2. 2. 2. 0. 0. 0. 2. 2. 2. 0. 0. 1. 1. 1. 1.
  0. 0. 0. 1. 1. 1. 0. 0. 1. 1. 1. 1.]]
    Accuracy: 80.55555555555556 %
```

Plotting a Decision Boundary

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

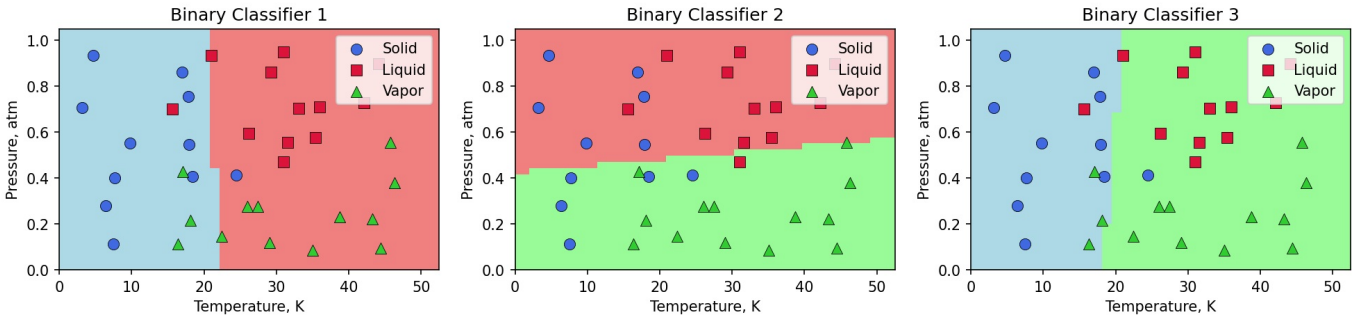
In [97]:
```
def plot_data(x, y, c, title="Phase of simulated material", newfig=True):
    xlim = [0,52.5]
    ylim = [0,1.05]
    markers = [dict(marker="o", color="royalblue"), dict(marker="s", color="crimson"), dict(marker="^", color="
    labels = ["Solid", "Liquid", "Vapor"]

    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", linewidths=0.4,label=labels[i])

    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 [98]:
```
plot_data(x,y,c)
plot_colors(classifiers)
plt.show()
```



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

In [99]:
```
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()
```



M3-L2 Problem 2 (6 points)

In [1]:
```
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 [2]:
```
x = np.array([7.4881350392732475,16.351893663724194,22.427633760716436,29.04883182996897,35.03654799338904,44.4
y = np.array([0.11120957227224215,0.1116933996874757,0.14437480785146242,0.11818202991034835,0.0859507900573786
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 [4]:
```
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 [29]:
```
def generate_ovr_prob_functions(xy, c):
    # YOUR CODE GOES HERE
    classify_0 = get_ovr_prob_function(xy, c, 0)
    classify_1 = get_ovr_prob_function(xy, c, 1)
    classify_2 = get_ovr_prob_function(xy, c, 2)
    return [classify_0, classify_1, classify_2]

def classify_ovr(probs, xy):
    # YOUR CODE GOES HERE
    preds = np.zeros([len(xy), 1])
```

```
    probabilities = np.zeros([len(xy), 3])

    probabilities[:, 0] = probs[0](xy)
    probabilities[:, 1] = probs[1](xy)
    probabilities[:, 2] = probs[2](xy)

    for i in range(len(xy)):
        preds[i] = np.argmax(probabilities[i, :])

    return preds.reshape(1, len(xy))
```

Trying out our multinomial classifier:

In [30]:
```
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"%")
```

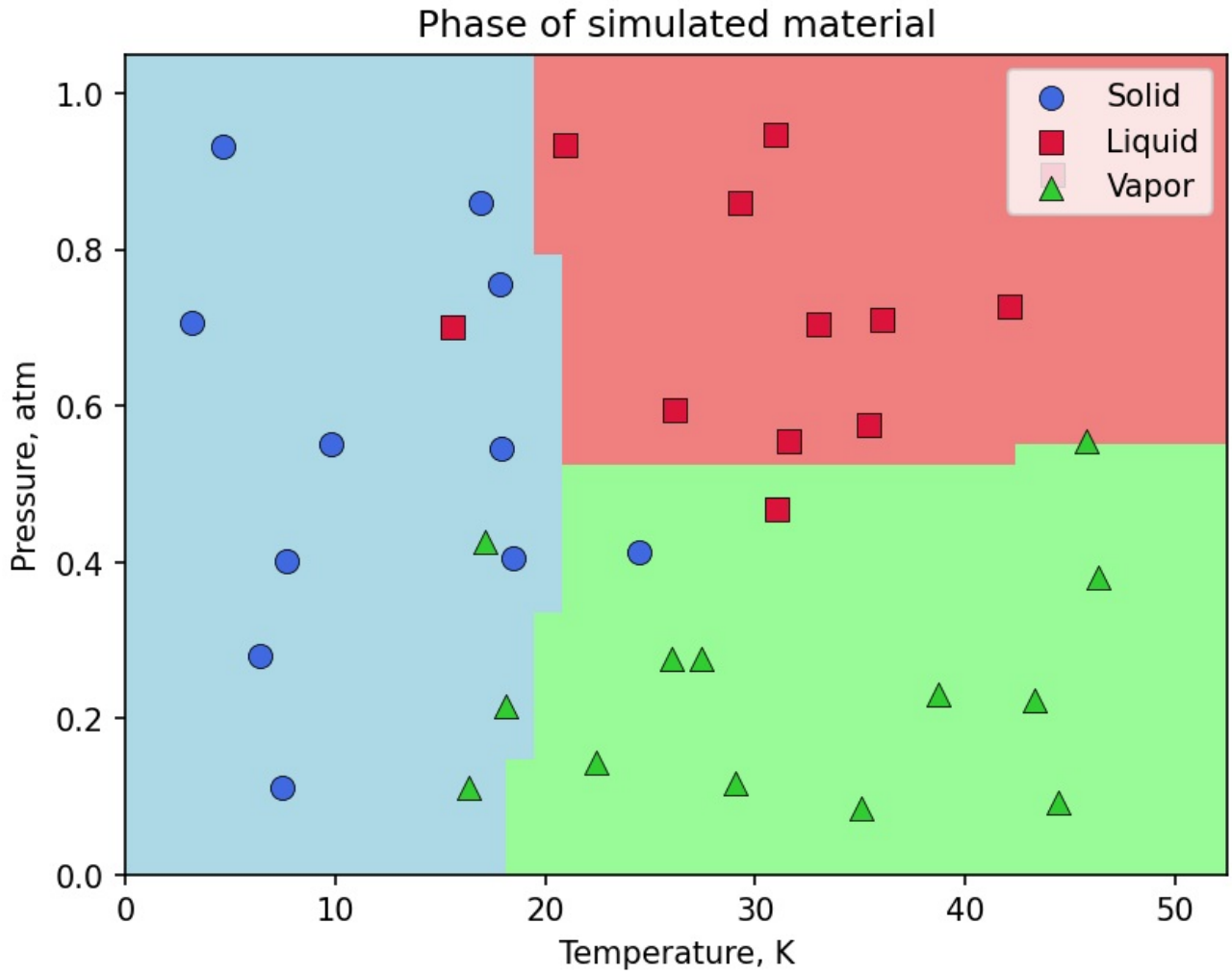
```
True Classes: [0 2 2 2 2 2 0 2 2 2 2 2 0 0 2 0 1 2 0 0 1 1 1 2 0 1 0 1 1 1 0 0 1 1 1 1]
 Predictions: [[0. 0. 2. 2. 2. 2. 0. 0. 2. 2. 2. 2. 0. 0. 0. 2. 2. 2. 0. 0. 1. 1. 1. 1.
  0. 0. 0. 1. 1. 1. 0. 0. 1. 1. 1. 1.]]
    Accuracy: 80.55555555555556 %
```

Plotting multnomial classifier results

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

In [31]:
```
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"), dict(marker="^", color="
    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", linewidths=0.4,label=labels[i])

    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 [32]:
```
plot_data(x,y,c)
plot_ovr_colors(probs)
plt.show()
```



M3-L2 Problem 3 (6 points)

In [1]:
```
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 [2]:
```
x = np.array([7.4881350392732475,16.351893663724194,22.427633760716436,29.04883182996897,35.03654799338904,44.4
y = np.array([0.11120957227224215,0.1116933996874757,0.14437480785146242,0.11818202991034835,0.0859507900573786
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 [4]:
```
from sklearn.linear_model import LogisticRegression

def get_logistic_regressor(features, classes):
    # YOUR CODE GOES HERE
    # - Instantiate model with regularization
    # - Fit model to data
    model = LogisticRegression()
    model.fit(features, classes)

    return model
```

In [5]:
```
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 %
```

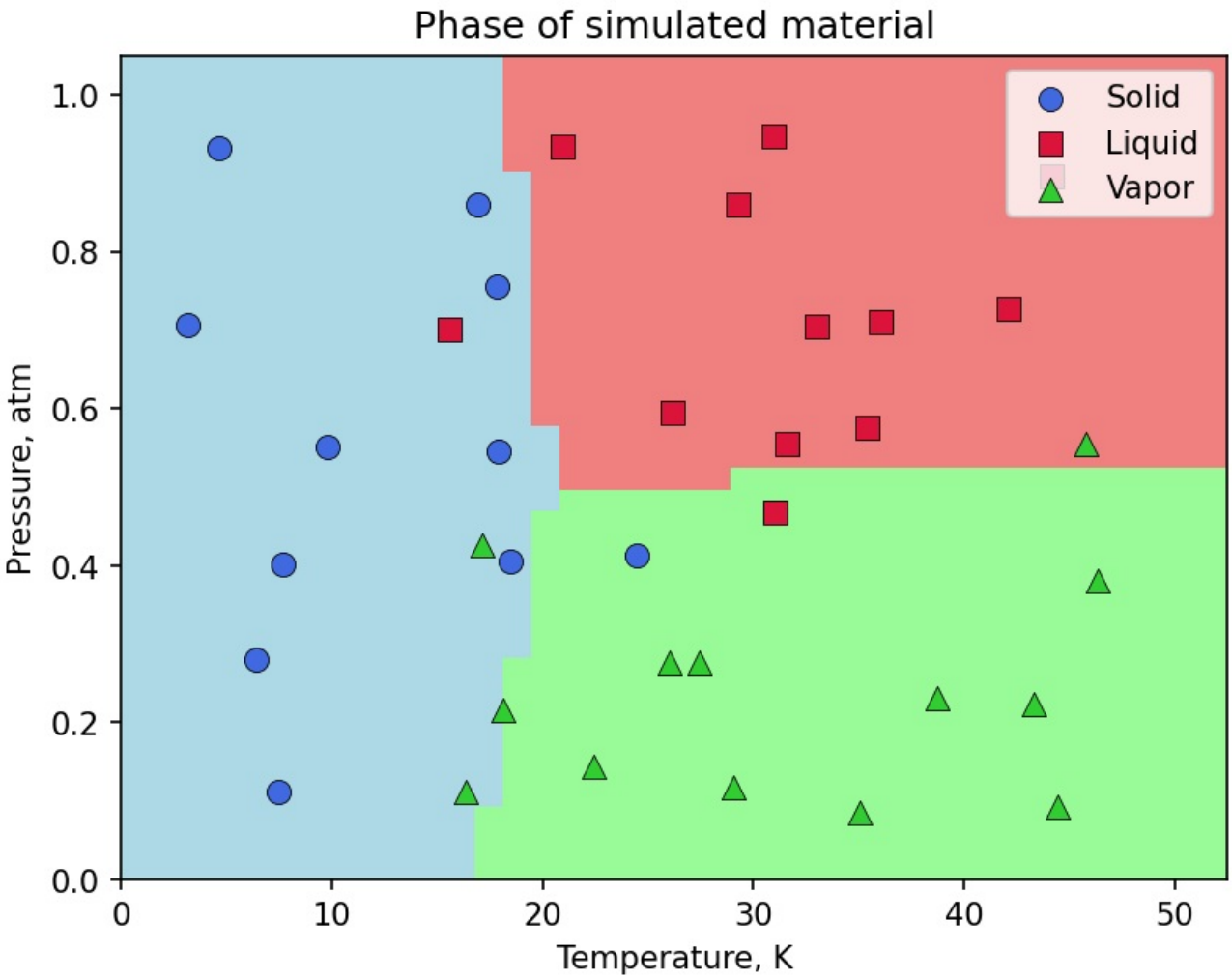
Plotting Multinomial Classifier Results

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

In [6]:
```
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"), dict(marker="^", color="
    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", linewidths=0.4,label=labels[i])

    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 [7]:
```
plot_data(x,y,c)
plot_sklearn_colors(model)
plt.show()
```



Problem 1 (24 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 [140…

```
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=0.4,label=classes[i],alpl

    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)
```

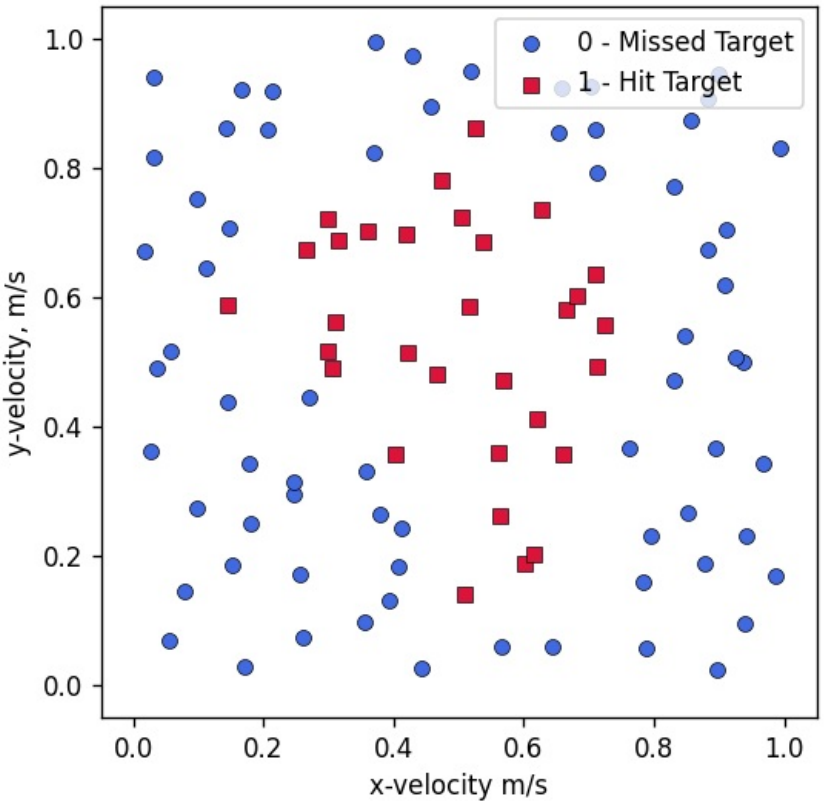
Load Data

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 [141…
```
train = np.load("data/w3-hw1-data-train.npy")
test = np.load("data/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 Target","1 - Hit Target"]
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 $(1, x_i,\ y_i,\ x_i^2,\ x_iy_i,\ y_i^2,\ x_i^3,\ x_i^2y_i,\ \dots)$ 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: $\frac{\partial L}{w_j} = \sum_{i=1}^{n}(g(w'x^{(i)}) - y^{(i)})x_j^{(i)}$

deg_x= 0 | 1 | 2 | 3 | 4 deg_y= 0 | 0,1 | 0,1,2 | 0,1,2,3 |

In [142…
```
# YOUR CODE GOES HERE
def sigmoid(h):
```

```
    sig = 1/(1 + np.exp(-h))

    return sig


def map_features(data):
    features = np.zeros([len(data[:, 0]), 45])
    xs = data[:, 0]
    ys = data[:, 1]
    features[:, 0] = np.zeros([len(data[:, 0]), 1]).flatten()
    count = 1

    for i in range(9):
        for j in range(i + 1):
            for k in range(i + 1):
                if(j + k == i and j + k != 0):
                    features[:, count] = np.multiply(np.power(xs, j), np.power(ys, k))
                    count += 1

    return features


def transform(data, w):
    features = map_features(data)
    h = features @ w

    return h


def loss(data, y, w):
    features = map_features(data)
    J = 0
    J1 = 0
    J2 = 0

    for i in range(len(y)):
        J1 += y[i] * (- np.log(sigmoid(features[i, :] @ w)))
        J2 += (1 - y[i]) * (- np.log(1 - sigmoid(features[i, :] @ w)))
        J += J1 + J2

    return J


def grad(data, y, w):
    x_w = transform(data, w)
    features = map_features(data)
    gradient = np.zeros([45, 1])
    g = 0

    for i in range(45):
        for j in range(len(y)):
            g += (sigmoid(x_w[j]) - y[j]) * features[j, i]
        gradient[i] = g
    return gradient
```

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 [143…

```
# YOUR CODE GOES HERE
def grad_desc(data, y, w0, iterations, stepsize):
    w = w0
    loss_hist = np.zeros([iterations + 1, 1])
    loss_hist[0] = loss(data, y, w)

    for i in range(iterations):
        gradient = grad(data, y, w)
        w = w - stepsize * gradient
        loss_hist[i + 1] = loss(data, y, w)
```

```
    return w, loss_hist
```

Training

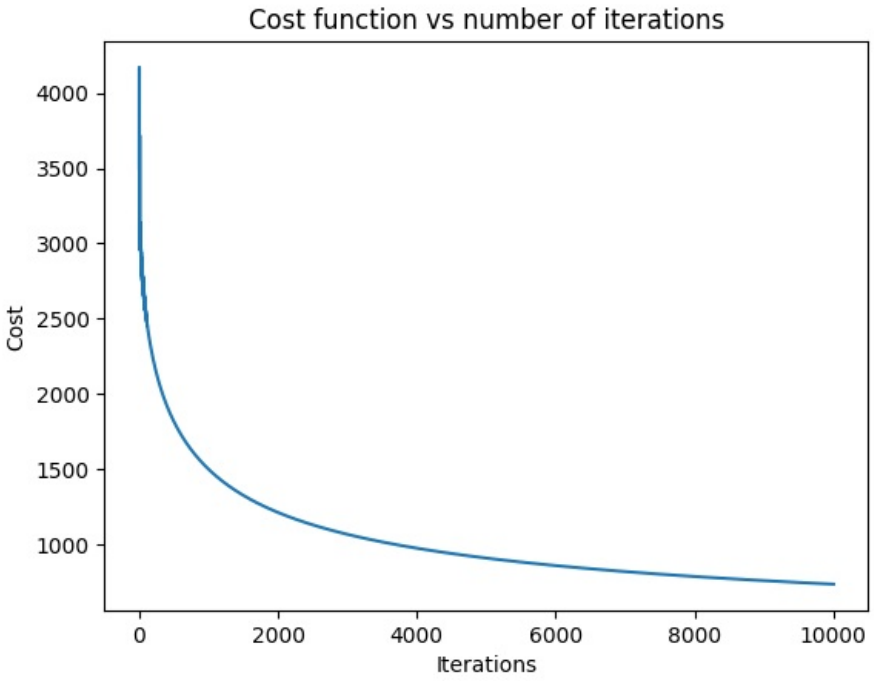
Run your gradient descent function and plot the loss as it converges. You may have to tune the step size and iteration count.

Also print the final vector `w`.

In [149…
```
# YOUR CODE GOES HERE (training)
w0 = np.zeros([45, 1])
iterations = 10000
stepsize = 1e-2
loss_hist = np.zeros([iterations, 1])

w, loss_hist = grad_desc(train_data, train_gt, w0, iterations, stepsize)
```

In [150…
```
# YOUR CODE GOES HERE (loss plot, print w)
iterator = np.linspace(0, iterations, len(loss_hist))

plt.plot(iterator, loss_hist)
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.title('Cost function vs number of iterations')
plt.show

print('Final w: ', w.T)
```

```
Final w:  [[  0.          -5.99101782 -12.10112491  -7.21592278   8.90526712
    7.76572235   9.44068888  19.4648382   23.87290864  17.34992449
   16.06520508  22.00081849  24.14294505  22.41102602  12.86332893
   10.1937581   13.83031864  15.0613945   13.59649098   9.50544653
   -0.75521828  -3.67397091  -1.25944028  -0.40252455  -1.42930127
   -4.12714289  -8.75136851 -18.42722934 -20.99304412 -19.18638039
  -18.47140694 -19.14234352 -20.95270643 -23.82018724 -28.17548695
  -36.69961034 -38.672274   -37.14614854 -36.46015544 -36.86353798
  -38.11063737 -40.01708438 -42.6211497  -46.40804292 -53.63043726]]
```



Accuracy

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

In [155…
```
# YOUR CODE GOES HERE
# predictions for train data
x_w_train = transform(train_data, w)
preds_train = sigmoid(x_w_train)
predictions_train = np.where(preds_train >= 0.5, 1, 0)
print('Ground truth for training data: ', train_gt.T)
print('Predicted values for training data: ', predictions_train.T)
accuracy_train = np.sum(predictions_train.T == train_gt) / len(train_gt) * 100
print('Accuracy for training data: ', accuracy_train)

# predictions for test data
```

```
x_w_test = transform(test_data, w)
preds_test = sigmoid(x_w_test)
predictions_test = np.where(preds_test >= 0.5, 1, 0)
print('\n\nGround truth for test data: ', test_gt.T)
print('\nPredicted values for test data: ', predictions_test.T)
accuracy_test = np.sum(predictions_test.T == test_gt) / len(test_gt) * 100
print('Accuracy for test data: ', accuracy_test)
```

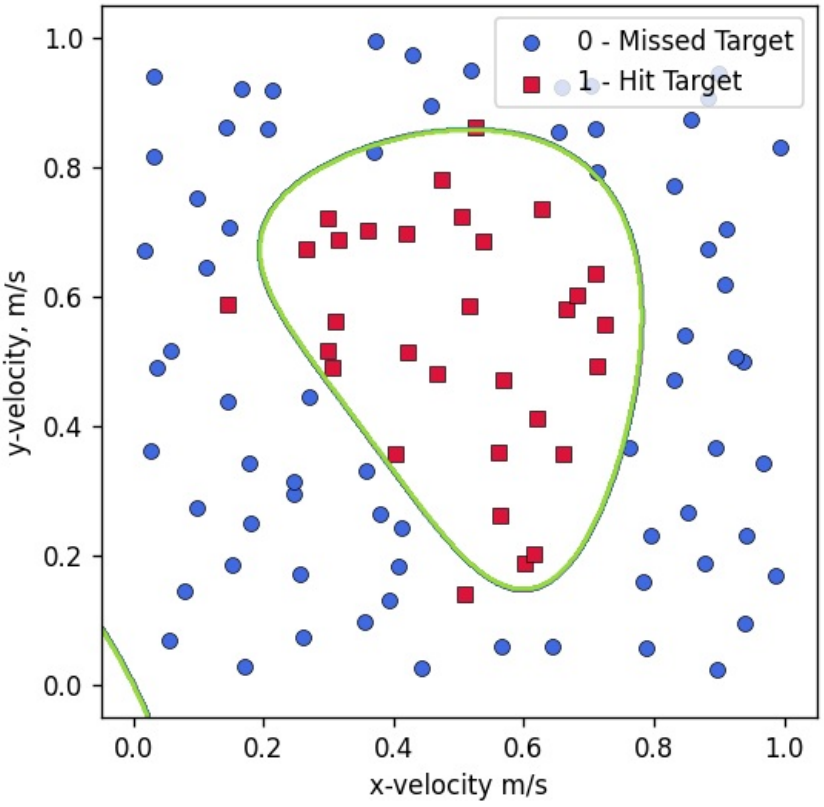
```
Ground truth for training data:  [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 0. 0. 0. 0. 0. 0. 0.
 0. 1. 1. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 1.
 0. 0. 0. 1. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0. 1. 1. 1. 1. 1. 1. 0. 0. 0. 0.
 1. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0.]
Predicted values for training data:  [[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 0 0 0 0 0 0 0 1 1
  1 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 0 0 0 0
  1 1 1 1 1 1 0 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]]
Accuracy for training data:  96.0


Ground truth for test data:  [0. 0. 0. 0. 0. 0. 1. 0. 1. 0. 0. 1. 1. 0. 0. 0. 0. 1. 1. 0. 0. 0. 1. 0.
 0.]

Predicted values for test data:  [[0 0 0 0 0 0 1 0 1 0 0 0 1 0 0 0 0 1 1 0 0 0 1 0 0]]
Accuracy for test data:  96.0
```

Visualize Results

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

In [156…

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



Problem 2 (24 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 [58]:
```
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"), dict(marker="^", color="
    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", linewidths=0.4,label=labels[i])

    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

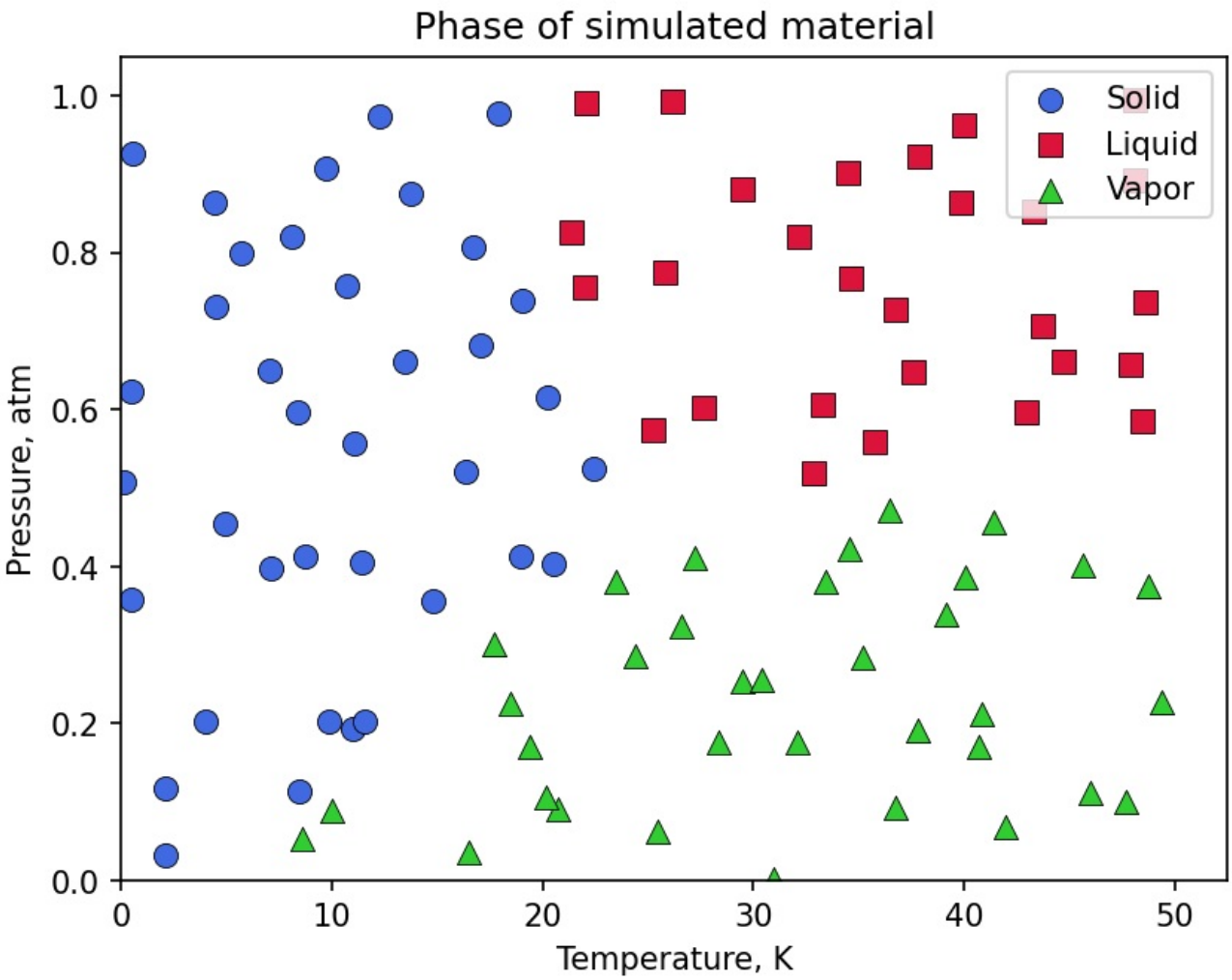
This cell loads the dataset into the following variables:

- `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 [59]:
```
train = np.load("data/w3-hw2-data-train.npy")
```

```
test = np.load("data/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 [60]:
```
def convert_to_binary_dataset(classes, A):
    classes_binary = (classes == A).astype(int)
    return classes_binary
```

In [103…
```
# YOUR CODE GOES HERE (gradient descent and related functions)
# YOUR CODE GOES HERE
def sigmoid(h):
    sig = 1/(1 + np.exp(-h))

    return sig


def map_features(data):
    features = np.zeros([len(data[:, 0]), 3])

    temp = data[:, 0].reshape(-1, 1)
    pressure = data[:, 1].reshape(-1, 1)
    columns = [np.ones_like(temp), temp, pressure]
```

```
    features = np.concatenate(columns, axis = 1)

    return features


def transform(data, w):
    features = map_features(data)
    h = features @ w

    return h


def loss(data, y, w):
    features = map_features(data)
    J = 0
    J1 = 0
    J2 = 0

    for i in range(len(y)):
        J1 += y[i] * (- np.log(sigmoid(features[i, :] @ w)))
        J2 += (1 - y[i]) * (- np.log(1 - sigmoid(features[i, :] @ w)))
        J += J1 + J2

    return J


def grad(data, y, w):
    x_w = transform(data, w)
    features = map_features(data)
    gradient = np.zeros([3, 1])
    g = 0

    for i in range(3):
        for j in range(len(y)):
            g += (sigmoid(x_w[j]) - y[j]) * features[j, i]
        gradient[i] = g
    return gradient


def grad_desc(data, y, w0, iterations, stepsize):
    w = w0
    loss_hist = np.zeros([iterations + 1, 1])
    loss_hist[0] = loss(data, y, w)

    for i in range(iterations):
        gradient = grad(data, y, w)
        w = w - stepsize * gradient
        loss_hist[i + 1] = loss(data, y, w)

    return w, loss_hist
```

Training

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

In [104…

```
# YOUR CODE GOES HERE (training)
class_0 = convert_to_binary_dataset(train_gt, 0)
w0 = 2 * np.ones([3, 1])
iterations0 = 10000
stepsize0 = 1e-5
loss_hist0 = np.zeros([iterations0, 1])

class_1 = convert_to_binary_dataset(train_gt, 1)
w1 = np.array([-3, 0, 3]).reshape([3, 1])
iterations1 = 10000
stepsize1 = 1e-5
loss_hist1 = np.zeros([iterations1, 1])

class_2 = convert_to_binary_dataset(train_gt, 2)
w2 = -np.ones([3, 1])
iterations2 = 10000
stepsize2 = 1e-5
loss_hist2 = np.zeros([iterations2, 1])


w0, loss_hist0 = grad_desc(train_data, class_0, w0, iterations0, stepsize0)
w1, loss_hist1 = grad_desc(train_data, class_1, w1, iterations1, stepsize1)
w2, loss_hist2 = grad_desc(train_data, class_2, w2, iterations2, stepsize2)

# YOUR CODE GOES HERE (print "w"s)
print('w for class A: ', w0.T)
```

```
print('w for class B: ', w1.T)
print('w for class C: ', w2.T)
```

```
C:\Users\barat\AppData\Local\Temp\ipykernel_6588\3607079642.py:35: RuntimeWarning: divide by zero encountered in
log
  J2 += (1 - y[i]) * (- np.log(1 - sigmoid(features[i, :] @ w)))
C:\Users\barat\AppData\Local\Temp\ipykernel_6588\3607079642.py:35: RuntimeWarning: invalid value encountered in
multiply
  J2 += (1 - y[i]) * (- np.log(1 - sigmoid(features[i, :] @ w)))
w for class A:  [[ 2.42063926 -0.15970045  0.25995618]]
w for class B:  [[-3.58596716  0.04751326  3.00972978]]
w for class C:  [[-1.25611237  0.04447028 -0.94434379]]
```

Classification function

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

In [105…
```
def classify(xy):
    probs = np.zeros([len(xy), 3])
    probs[:, 0] = sigmoid(transform(xy, w0)).flatten()
    probs[:, 1] = sigmoid(transform(xy, w1)).flatten()
    probs[:, 2] = sigmoid(transform(xy, w2)).flatten()
    classes = np.zeros([len(xy), 1])

    for i in range(len(xy)):
        classes[i] = np.argmax(probs[i, :])

    return classes
```

Accuracy

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

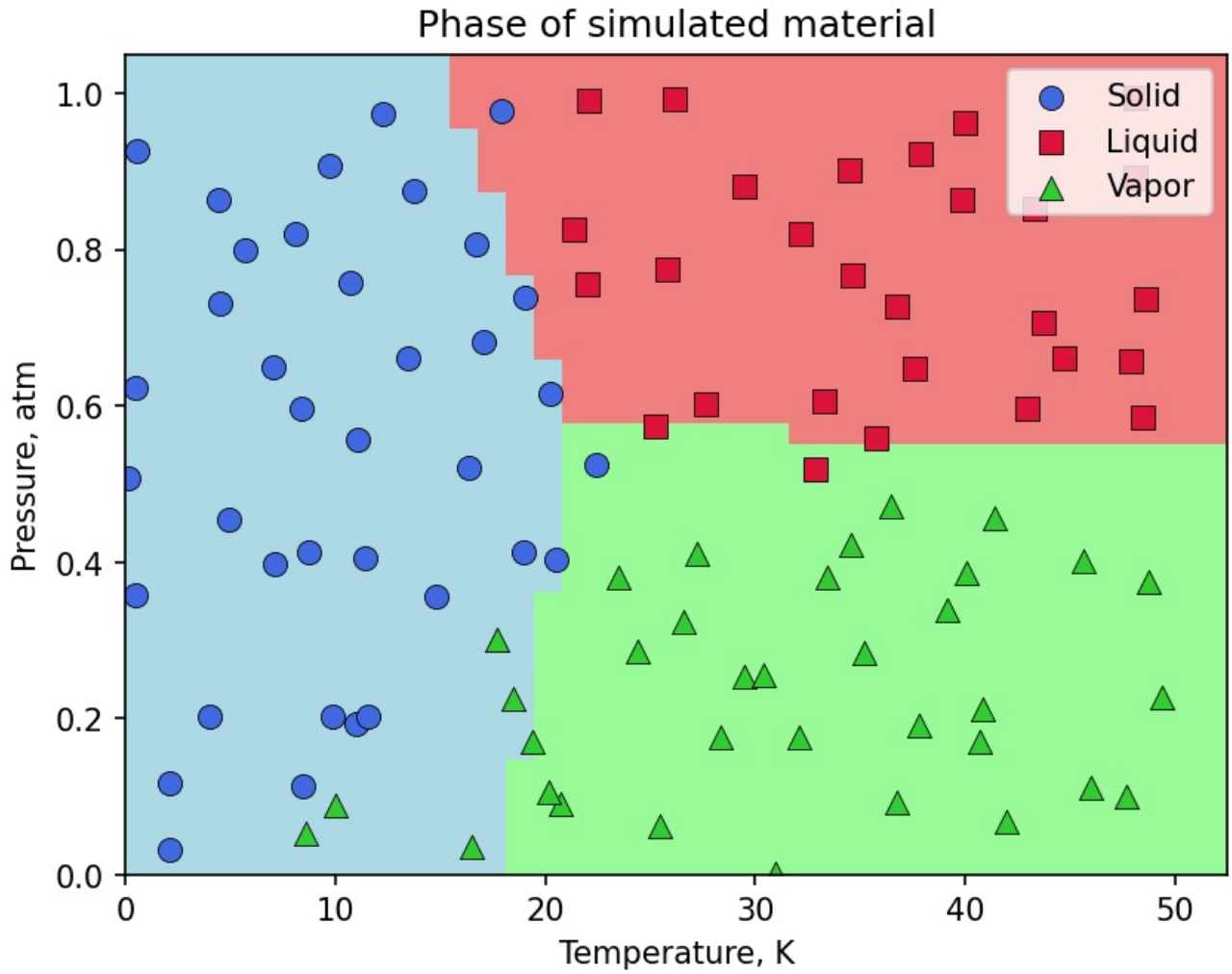
In [106…
```
# YOUR CODE GOES HERE (accuracy)
# YOUR CODE GOES HERE
# predictions for train data
preds_train = classify(train_data)
# print('Predicted classes for training data: ', preds_train.T)
# print('Ground truth for test data: ', train_gt)
accuracy_train = np.sum(preds_train.T == train_gt) / len(train_gt) * 100
print('\nAccuracy for training data: ', accuracy_train)

# predictions for test data
preds_test = classify(test_data)
# print('\nPredicted classes for testing data: ', preds_test.T)
# print('Ground truth for test data: ', test_gt.T)
accuracy_test = np.sum(preds_test.T == test_gt) / len(test_gt) * 100
print('\nAccuracy for test data: ', accuracy_test)
```

```
Accuracy for training data:  90.0

Accuracy for test data:  84.0
```

Plot results

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

In [107…
```
plot_data(train_data[:,0], train_data[:,1], train_gt)
plot_colors(classify)
```



Problem 3 (24 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 [1]:
```
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 [2]:
```
x_train = np.load("data/w3-hw3-train_x.npy")
y_train = np.load("data/w3-hw3-train_y.npy")
x_test = np.load("data/w3-hw3-test_x.npy")
y_test = np.load("data/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 [3]:
```
# YOUR CODE GOES HERE (sklearn models)
model_ovr = LogisticRegression(max_iter = 5000, multi_class = 'ovr')
model_multi = LogisticRegression(max_iter = 5000, multi_class = 'multinomial')

model_ovr.fit(x_train, y_train)
model_multi.fit(x_train, y_train)
```

```
c:\Users\barat\AppData\Local\Programs\Python\Python312\Lib\site-packages\sklearn\linear_model\_logistic.py:1256:
FutureWarning: 'multi_class' was deprecated in version 1.5 and will be removed in 1.7. Use OneVsRestClassifier(L
ogisticRegression(..)) instead. Leave it to its default value to avoid this warning.
  warnings.warn(
c:\Users\barat\AppData\Local\Programs\Python\Python312\Lib\site-packages\sklearn\linear_model\_logistic.py:1247:
FutureWarning: 'multi_class' was deprecated in version 1.5 and will be removed in 1.7. From then on, it will alw
ays use 'multinomial'. Leave it to its default value to avoid this warning.
  warnings.warn(
```

Out[3]:

LogisticRegression

```
LogisticRegression(max_iter=5000, multi_class='multinomial')
```

Accuracy

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

In [9]:
```
# YOUR CODE GOES HERE (print the 4 requested accuracy values)
preds_train_ovr = model_ovr.predict(x_train)
preds_train_multi = model_multi.predict(x_train)

preds_test_ovr = model_ovr.predict(x_test)
preds_test_multi = model_multi.predict(x_test)

# calculating accuracy values
accuracy_train_ovr = np.sum(preds_train_ovr.T == y_train) / len(y_train) * 100
accuracy_train_multi = np.sum(preds_train_multi.T == y_train) / len(y_train) * 100

accuracy_test_ovr = np.sum(preds_test_ovr.T == y_test) / len(y_test) * 100
accuracy_test_multi = np.sum(preds_test_ovr.T == y_test) / len(y_test) * 100

print("Accuracies for training data: \nOne-vs-Rest method: ", accuracy_train_ovr, '\nMultinomial Classification
print("\n\nAccuracies for test data: \nOne-vs-Rest method: ", accuracy_test_ovr, '\nMultinomial Classification:
```

```
Accuracies for training data:
One-vs-Rest method:  94.675
Multinomial Classification:  96.45


Accuracies for test data:
One-vs-Rest method:  90.8
Multinomial Classification:  90.8
```