Problem 3

Use this notebook to write your code for problem 3 by filling in the sections marked `# TODO` and running all cells.

In [61]:
```
import numpy as np
import matplotlib.pyplot as plt
import itertools
from prettytable import PrettyTable

from perceptron_helper import (
    predict,
    plot_data,
    boundary,
    plot_perceptron,
)

%matplotlib inline
```

Implementation of Perceptron

First, we will implement the perceptron algorithm. Fill in the `update_perceptron()` function so that it finds a single misclassified point and updates the weights and bias accordingly. If no point exists, the weights and bias should not change.

Hint: You can use the `predict()` helper method, which labels a point 1 or -1 depending on the weights and bias.

In [62]:
```
def update_perceptron(X, Y, w, b):
    """
    This method updates a perceptron model. Takes in the previous weights
    and returns weights after an update, which could be nothing.

Inputs:
        X: A (N, D) shaped numpy array containing N points of D dimensions.
        Y: A (N, ) shaped numpy array containing the N labels (one for each point
in X).
        w: A (D, ) shaped numpy array containing the initial weight vector of D di
mensions.
        b: A float containing the bias term.

    Output:
        next_w: A (D, ) shaped numpy array containing the next weight vector
                after updating on a single misclassified point, if one exists.
        next_b: The next float bias term after updating on a single
                misclassified point, if one exists.
    """
    next_w, next_b = np.copy(w), np.copy(b)

    #===============================================
    # TODO: Implement update rule for perceptron.
    #================================================

    for i in range(len(X)):
        # if misclassified
        if predict(X[i], w, b) != Y[i]:
            for j in range(len(X[i])):
                # adjust weights for every dimension
                next_w[j] = w[j] + Y[i]*X[i][j]
                next_b = b + Y[i]

    return next_w, next_b
```

Next you will fill in the `run_perceptron()` method. The method performs single updates on a misclassified point until convergence, or max_iter updates are made. The function will return the final weights and bias. You should use the `update_perceptron()` method you implemented above.

In [63]:
```
def run_perceptron(X, Y, w, b, max_iter):
    """
    This method runs the perceptron learning algorithm. Takes in initial weights
    and runs max_iter update iterations. Returns final weights and bias.

    Inputs:
        X: A (N, D) shaped numpy array containing N points of D dimensions.
        Y: A (N, ) shaped numpy array containing the N labels (one for each point
in X).
        w: A (D, ) shaped numpy array containing the initial weight vector of D di
mensions.
        b: A float containing the initial bias term.
        max_iter: An int for the maximum number of updates evaluated.

    Output:
        w: A (D, ) shaped numpy array containing the final weight vector.
        b: The final float bias term.
    """

    #============================================
    # TODO: Implement perceptron update loop.
    #=============================================

    for _ in range(max_iter):
        w, b = update_perceptron(X, Y, w, b)

    return w, b
```

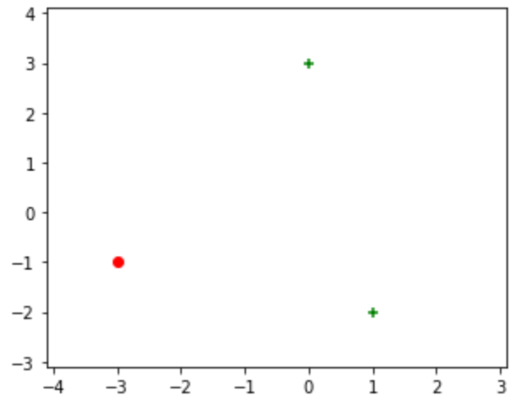
Problem 3A

Visualizing a Toy Dataset

We will begin by training our perceptron on a toy dataset of 3 points. The green points are labelled +1 and the red points are labelled -1. We use the helper function `plot_data()` to do so.

In [64]:
```
X = np.array([[ -3, -1], [0, 3], [1, -2]])
Y = np.array([ -1, 1, 1])
```

In [65]:
```
fig = plt.figure(figsize=(5,4))
ax = fig.gca(); ax.set_xlim(-4.1, 3.1); ax.set_ylim(-3.1, 4.1)
plot_data(X, Y, ax)
```



Running the Perceptron

Next, we will run the perceptron learning algorithm on this dataset. Update the code to show the weights and bias at each timestep and the misclassified point used in each update. You may change the `update_perceptron()` method to do this, but be sure to update the starter code as well to reflect those changes.

Run the below code, and fill in the corresponding table in the set.

In [66]:
```
# Initialize weights and bias.
weights = np.array([0.0, 1.0])
bias = 0.0

weights, bias = run_perceptron(X, Y, weights, bias, 16)

print()
print ("final w = %s, final b = %.1f" % (weights, bias))
```

```
final w = [2. 0.], final b = 3.0
```

Visualizating the Perceptron

Getting all that information in table form isn't very informative. Let us visualize what the decision boundaries are at each timestep instead.

The helper functions `boundary()` and `plot_perceptron()` plot a decision boundary given a perceptron weights and bias. Note that the equation for the decision boundary is given by:

$$w_1 x_1 + w_2 x_2 + b = 0.$$

Using some algebra, we can obtain x_2 from x_1 to plot the boundary as a line.

$$x_2 = \frac{-w_1 x_2 - b}{w_2}.$$

Below is a redefinition of the `run_perceptron()` method to visualize the points and decision boundaries at each timestep instead of printing. Fill in the method using your previous `run_perceptron()` method, and the above helper methods.

Hint: The axs element is a list of Axes, which are used as subplots for each timestep. You can do the following:

```
ax = axs[i]
```

to get the plot correponding to $t = i$. You can then use ax.set_title() to title each subplot. You will want to use the `plot_data()` and `plot_perceptron()` helper methods.

In [78]:
```
def run_perceptron(X, Y, w, b, axs, max_iter):
    """
    This method runs the perceptron learning algorithm. Takes in initial weights
    and runs max_iter update iterations. Returns final weights and bias.

    Inputs:
        X: A (N, D) shaped numpy array containing N points of D dimensions.
        Y: A (N, ) shaped numpy array containing the N labels (one for each point
in X).
        w: A (D, ) shaped numpy array containing the initial weight vector of D di
mensions.
        b: A float containing the initial bias term.
        axs: A list of Axes that contain suplots for each timestep.
        max_iter: An int for the maximum number of updates evaluated.

    Output:
        The final weight and bias vectors.
    """

    #============================================
    # TODO: Implement perceptron update loop.
    #=============================================
    time = 0
    t = PrettyTable()

    t.field_names = ["t", "b", "w1", "w2"]
    t.add_row([time, b, w[0], w[1]])

    for i in range(max_iter):
        time += 1
        w, b = update_perceptron(X, Y, w, b)
        ax = axs[i]
        title = "Iteration " + str(i)
        ax.set_title(title)
        plot_perceptron(w, b, ax)
        plot_data(X, Y, ax)
        t.add_row([time, b, w[0], w[1]])

    print(t)
    return w, b
```

Run the below code to get a visualization of the perceptron algorithm. The red region are areas the perceptron thinks are negative examples.

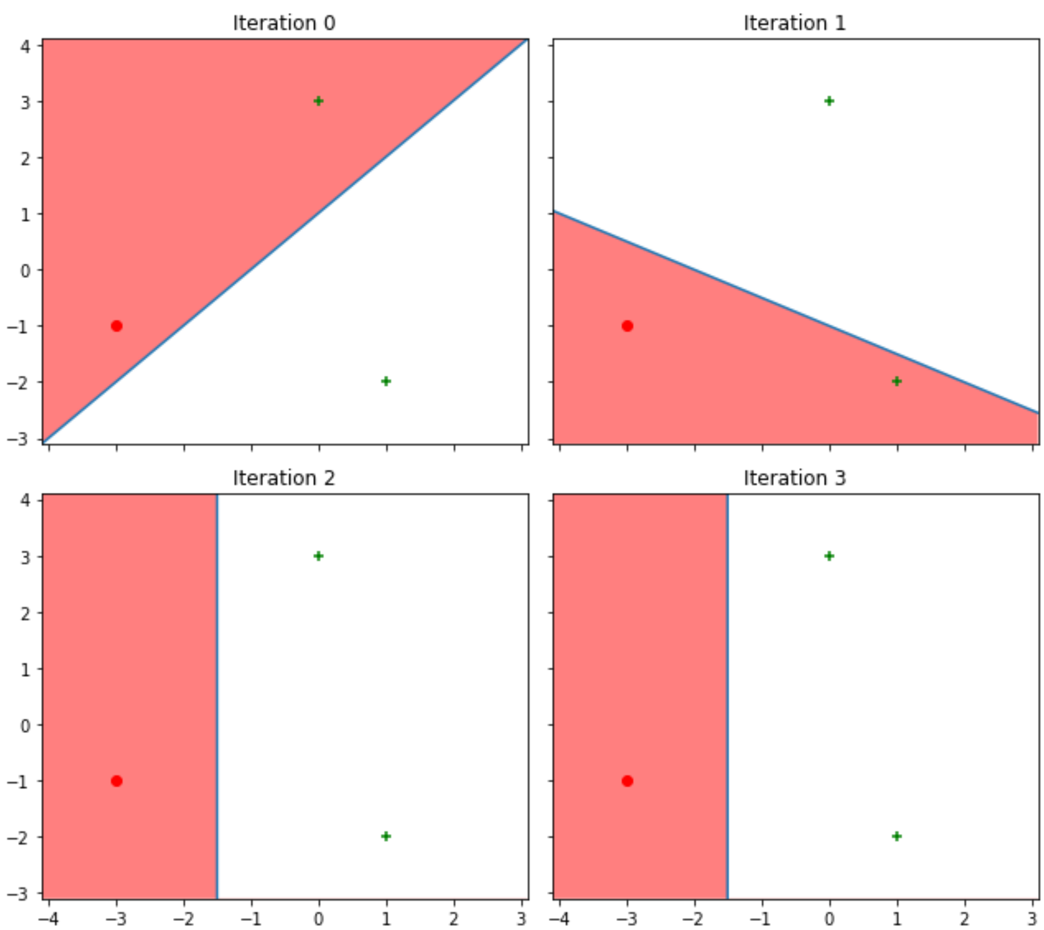
In [79]:
```python
# Initialize weights and bias.
weights = np.array([0.0, 1.0])
bias = 0.0

# Note: there are 4 subplots and max_iter=4 below. Plot BEFORE each timestep updat
e.
f, ax_arr = plt.subplots(2, 2, sharex=True, sharey=True, figsize=(9,8))
axs = list(itertools.chain.from_iterable(ax_arr))
for ax in axs:
    ax.set_xlim(-4.1, 3.1); ax.set_ylim(-3.1, 4.1)

run_perceptron(X, Y, weights, bias, axs, 4)

f.tight_layout()
```

```
+---+-----+-----+------+
| t |  b  |  w1 |  w2  |
+---+-----+-----+------+
| 0 | 0.0 | 0.0 | 1.0  |
| 1 | 1.0 | 1.0 | -1.0 |
| 2 | 2.0 | 1.0 | 2.0  |
| 3 | 3.0 | 2.0 | 0.0  |
| 4 | 3.0 | 2.0 | 0.0  |
+---+-----+-----+------+
```



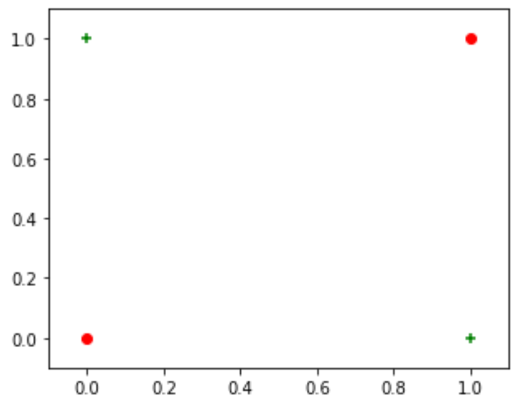
Problem 3C

Visualize a Non-linearly Separable Dataset.

We will now work on a dataset that cannot be linearly separated, namely one that is generated by the XOR function.

In [33]:
```
X = np.array([[0, 1], [1, 0], [0, 0], [1, 1]])
Y = np.array([1, 1, -1, -1])
```

In [34]:
```
fig = plt.figure(figsize=(5,4))
ax = fig.gca(); ax.set_xlim(-0.1, 1.1); ax.set_ylim(-0.1, 1.1)
plot_data(X, Y, ax)
```



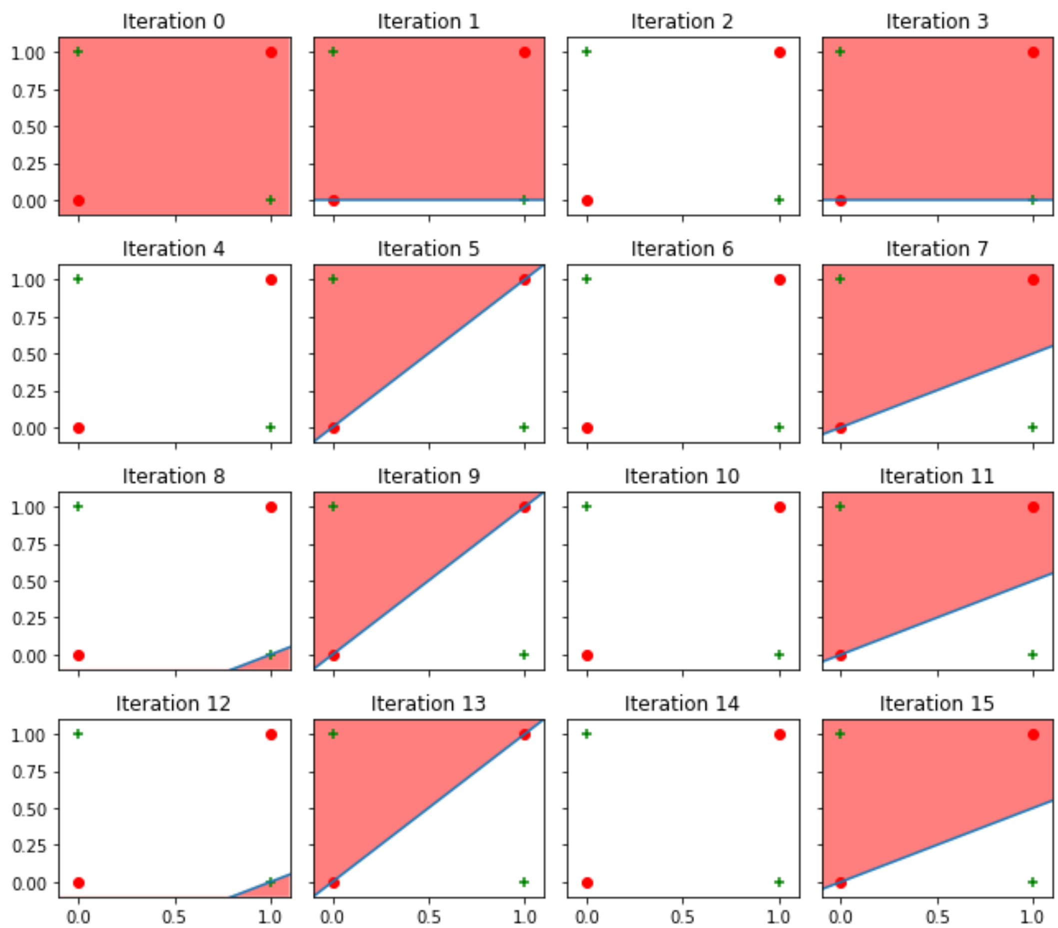
We will now run the perceptron algorithm on this dataset. We will limit the total timesteps this time, but you should see a pattern in the updates. Run the below code.

In [35]:
```
# Initialize weights and bias.
weights = np.array([0.0, 1.0])
bias = 0.0

# Note: there are 16 subplots and max_iter=16. Plot BEFORE each timestep update.
f, ax_arr = plt.subplots(4, 4, sharex=True, sharey=True, figsize=(9,8))
axs = list(itertools.chain.from_iterable(ax_arr))
for ax in axs:
    ax.set_xlim(-0.1, 1.1); ax.set_ylim(-0.1, 1.1)

run_perceptron(X, Y, weights, bias, axs, 16)

f.tight_layout()
```



In [ ]: