Sample solutions

Logistic regression

Beyond regression, another important data analysis task is *classification*, in which you are given a set of labeled data points and you wish to lea the labels. The canonical example of a classification algorithm is *logistic regression*, the topic of this notebook.

> Although it's called "regression" it is really a model for classification.

Here, you'll consider *binary classification*. Each data point belongs to one of $c = 2$ possible classes. By convention, we will denote these *class l* and "1." However, the ideas can be generalized to the multiclass case, i.e., $c > 2$, with labels $\{0, 1, \ldots, c-1\}$.

You'll also want to review from earlier notebooks the concept of gradient ascent/descent (or "steepest ascent/descent"), when optimizing a scala vector variable.

Part 0: Introduction

This part of the notebook introduces you to the classification problem through a "geometric interpretation."

Setup

In [1]:
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
from IPython.display import display, Math

%matplotlib inline

import matplotlib as mpl
mpl.rc("savefig", dpi=100) # Adjust for higher-resolution figures
```

A note about slicing columns from a Numpy matrix. If you want to extract a column `i` from a Numpy matrix `A` *and* keep it as a column vector, use the slicing notation, `A[:, i:i+1]`. Not doing so can lead to subtle bugs. To see why, compare the following slices.

In [2]:
```
A = np.array ([[1, 2, 3],
               [4, 5, 6],
               [7, 8, 9]
              ], dtype=float)

print ("A[:, :] ==\n", A)
print ("\na0 := A[:, 0] ==\n", A[:, 0])
print ("\na1 := A[:, 2:3] == \n", A[:, 2:3])

print ("\nAdd columns 0 and 2?")
a0 = A[:, 0]
a1 = A[:, 2:3]
print (a0 + a1)
```

```
A[:, :] ==
 [[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.]]

a0 := A[:, 0] ==
 [1. 4. 7.]

a1 := A[:, 2:3] ==
 [[3.]
 [6.]
 [9.]]

Add columns 0 and 2?
[[ 4.  7. 10.]
 [ 7. 10. 13.]
 [10. 13. 16.]]
```

Aside: Broadcasting in Numpy. What is happening in the operation, `a0 + a1`, shown above? When the shapes of two objects do not match, N figure out if there is a natural way to make them compatible. See this supplemental notebook (./mo_numpy_mo_problems.ipynb) for information "broadcasting rule," along with other Numpy tips.

Example data: Rock lobsters!

As a concrete example of a classification task, consider the results of the following experiment (http://www.stat.ufl.edu/~winner/data/lobster_surv

Some marine biologists started with a bunch of lobsters of varying sizes (size being a proxy for the stage of a lobster's development). They then exposed these lobsters to a variety of predators. Finally, the outcome that they measured is whether the lobsters survived or not.

The data is a set of points, one point per lobster, where there is a single predictor (the lobster's size) and the response is whether the lobsters su "1") or died (label "0").

> For the original paper, see this link (http://downeastinstitute.org/assets/files/Published%20papers/Wilkinson%20et%20al%202015-1.pdf) what we can only guess is what marine biologists do in their labs, see this image (http://i.imgur.com/dQDKgys.jpg) (or this possibly not-s for-work alternative (http://web.archive.org/web/20120628012654/http://www.traemcneely.com/wp-content/uploads/2012/04/wpid-Lobste Fights-e1335308484734.jpeg)).

Start by downloading this data.

In [3]:
```
import requests
import os
import hashlib
import io

def on_vocareum():
    return os.path.exists('.voc')

def download(file, local_dir="", url_base=None, checksum=None):
    local_file = "{}{}".format(local_dir, file)
    if not os.path.exists(local_file):
        if url_base is None:
            url_base = "https://cse6040.gatech.edu/datasets/"
        url = "{}{}".format(url_base, file)
        print("Downloading: {} ...".format(url))
        r = requests.get(url)
        with open(local_file, 'wb') as f:
            f.write(r.content)

    if checksum is not None:
        with io.open(local_file, 'rb') as f:
            body = f.read()
            body_checksum = hashlib.md5(body).hexdigest()
            assert body_checksum == checksum, \
                "Downloaded file '{}' has incorrect checksum: '{}' instead of '{}'".format(local_file,
                                                                                           body_checksu
                                                                                           checksum)
    print("'{}' is ready!".format(file))


if on_vocareum():
    URL_BASE = "https://cse6040.gatech.edu/datasets/rock-lobster/"
    DATA_PATH = "../resource/asnlib/publicdata/"
else:
    URL_BASE = "https://github.com/cse6040/labs-fa17/raw/master/datasets/rock-lobster/"
    DATA_PATH = ""

datasets = {'lobster_survive.dat.txt': '12fc1c22ed9b4d7bf04bf7e0fec996b7',
            'logreg_points_train.csv': '25bbca6105bae047ac4d62ee8b76c841',
            'log_likelihood_soln.npz': '5a9e17d56937855727afa6db1cd83306',
            'grad_log_likelihood_soln.npz': 'a67c00bfa95929e12d423105d8412026',
            'hess_log_likelihood_soln.npz': 'b46443fbf0577423b084122503125887'}


for filename, checksum in datasets.items():
    download(filename, local_dir=DATA_PATH, url_base=URL_BASE, checksum=checksum)

print("\n(All data appears to be ready.)")
```

```
'grad_log_likelihood_soln.npz' is ready!
'hess_log_likelihood_soln.npz' is ready!
'log_likelihood_soln.npz' is ready!
'lobster_survive.dat.txt' is ready!
'logreg_points_train.csv' is ready!

(All data appears to be ready.)
```

Here is a plot of the raw data, which was taken from this source (http://www.stat.ufl.edu/~winner/data/lobster_survive.dat).

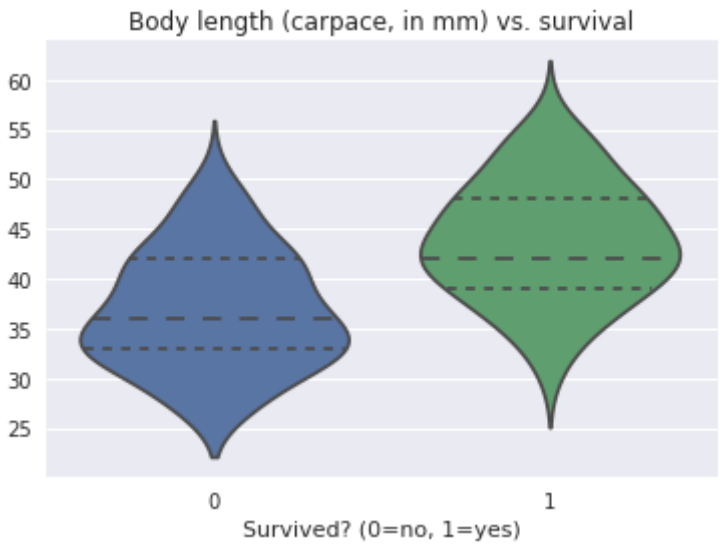
In [4]:
```
df_lobsters = pd.read_table('{}lobster_survive.dat.txt'.format(DATA_PATH),
                            sep=r'\s+', names=['CarapaceLen', 'Survived'])
display(df_lobsters.head())
print("...")
display(df_lobsters.tail())
```

	CarapaceLen	Survived
0	27	0
1	27	0
2	27	0
3	27	0
4	27	0

...

	CarapaceLen	Survived
154	54	1
155	54	1
156	54	1
157	54	1
158	57	1

In [5]:
```
ax = sns.violinplot(x="Survived", y="CarapaceLen",
                    data=df_lobsters, inner="quart")
ax.set(xlabel="Survived? (0=no, 1=yes)",
       ylabel="",
       title="Body length (carpace, in mm) vs. survival");
```



Although the classes are distinct in the aggregate, where the median carapace (outer shell) length is around 36 mm for the lobsters that died an those that survived, they are not cleanly separable.

Notation

To develop some intuition and a classification algorithm, let's formulate the general problem and apply it to synthetic data sets.

Let the data consist of m observations of d continuously-valued predictors. In addition, for each data observation we observe a binary label whc either 0 or 1.

Just like our convention in the linear regression case, represent each observation, or data point, by an *augumented* vector, $\hat{x}_i^T$,

$$\hat{x}_i^T \equiv \begin{pmatrix} x_{i,0} & x_{i,1} & \cdots & x_{i,d-1} & 1 \end{pmatrix}.$$

That is, the point is the d coordinates augmented by an initial dummy coordinate whose value is 1. This convention is similar to what we did in li regression.

We can also stack these points as rows of a matrix, X, again, just as we did in regression:

$$X \equiv \begin{pmatrix} \hat{x}_0^T \\ \hat{x}_1^T \\ \vdots \\ \hat{x}_{m-1}^T \end{pmatrix} = \begin{pmatrix} x_{0,1} & x_{0,2} & \cdots & x_{0,d} & 1 \\ x_{1,1} & x_{1,2} & \cdots & x_{1,d} & 1 \\ & & \vdots & & 1 \\ x_{m-1,1} & x_{m-1,2} & \cdots & x_{m-1,d} & 1 \end{pmatrix}.$$

We will take the labels to be a binary vector, $y^T \equiv (y_0, y_1, \ldots, y_{m-1})^T$.

Example: A synthetic training set. We've pre-generated a synethetic data set consisting of labeled data points. Let's download and inspect it, and then visually.

In [6]:
```
df = pd.read_csv('{}logreg_points_train.csv'.format(DATA_PATH))

display(df.head())
```

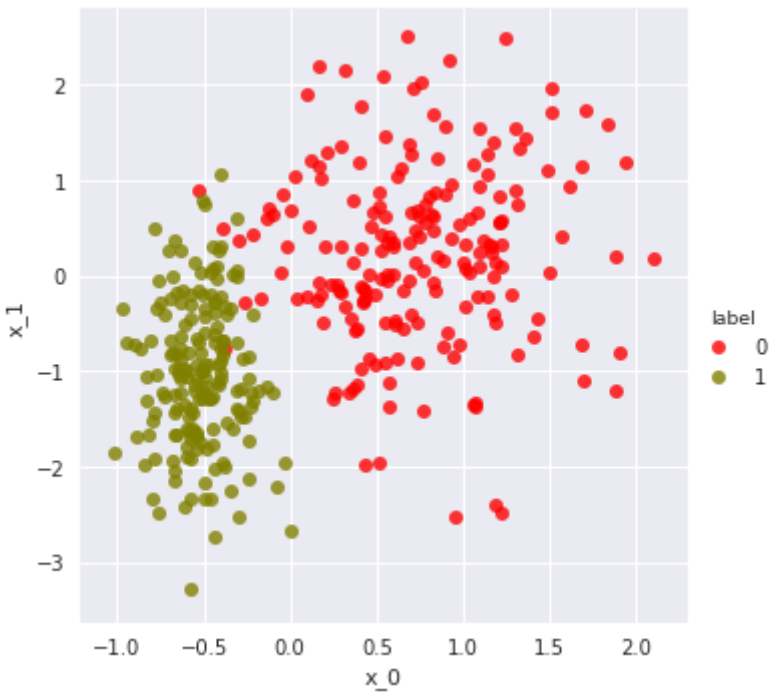
```
print("...")
display(df.tail())
```

	x_0	x_1	label
0	-0.234443	-1.075960	1
1	0.730359	-0.918093	0
2	1.432270	-0.439449	0
3	0.026733	1.050300	0
4	1.879650	0.207743	0

...

	x_0	x_1	label
370	1.314300	0.746001	0
371	-0.759737	-0.042944	1
372	0.683560	-0.047791	0
373	0.774747	0.743837	0
374	0.899119	1.576390	0

In [7]:
```
def make_scatter_plot(df, x="x_0", y="x_1", hue="label",
                      palette={0: "red", 1: "olive"},
                      size=5):
    sns.lmplot(x=x, y=y, hue=hue, data=df, palette=palette,
               fit_reg=False)

mpl.rc("savefig", dpi=120) # Adjust for higher-resolution figures
make_scatter_plot(df)
```



Next, let's extract the coordinates as a Numpy matrix of `points` and the labels as a Numpy column vector `labels`. Mathematically, the `points` r corresponds to X and the `labels` vector corresponds to y.

In [8]:
```
points = np.insert(df.as_matrix (['x_0', 'x_1']), 2, 1.0, axis=1)
labels = df.as_matrix(['label'])

print ("First and last 5 points:\n", '='*23, '\n', points[:5], '\n...\n', points[-5:], '\n')
print ("First and last 5 labels:\n", '='*23, '\n', labels[:5], '\n...\n', labels[-5:], '\n')
```

```
First and last 5 points:
 ======================
 [[-0.234443  -1.07596    1.       ]
 [ 0.730359  -0.918093   1.       ]
 [ 1.43227   -0.439449   1.       ]
 [ 0.0267327  1.0503     1.       ]
 [ 1.87965    0.207743   1.       ]]
...
 [[ 1.3143     0.746001   1.       ]
 [-0.759737  -0.0429435  1.       ]
 [ 0.68356   -0.0477909  1.       ]
 [ 0.774747   0.743837   1.       ]
 [ 0.899119   1.57639    1.       ]]

First and last 5 labels:
 ======================
 [[1]
 [0]
 [0]
```

```
 [0]
 [0]]
...
 [[0]
 [1]
 [0]
 [0]
 [0]]
```

Linear discriminants and the heaviside function

Suppose you think that the *boundary* between the two clusters may be represented by a line. For the synthetic data example above, I hope you' such a model is not a terrible one.

A linear boundary is also known as a *linear discriminant*. Any point x on this line may be described by $\theta^T x$, where θ is a vector of coefficients:

$$\theta \equiv \begin{pmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_d \end{pmatrix}.$$

For example, suppose our observations have two predictors each ($d = 2$). Let the corresponding data point be $x^T \equiv (x_0, x_1, x_2 = 1.0)$. The means that

$$\begin{aligned} \theta^T x = 0 \quad &= \quad \theta_0 x_0 + \theta_1 x_1 + \theta_2 \\ \implies \qquad x_1 \quad &= \quad -\frac{\theta_2}{\theta_1} - \frac{\theta_0}{\theta_1} x_0. \end{aligned}$$

So that describes points *on* the line. However, given *any* point x in the d-dimensional space that is *not* on the line, $\theta^T x$ still produces a value: th positive on one side of the line ($\theta^T x > 0$) or negative on the other ($\theta^T x < 0$).

In other words, you can use the linear discriminant function, $\theta^T x$, to *generate* a label for each point x: just reinterpret its sign!

If you want "0" and "1" labels, the *heaviside function*, $H(y)$, will convert a positive y to the label "1" and all other values to "0."

$$H(y) \equiv \begin{cases} 1 & \text{if } y > 0 \\ 0 & \text{if } y \leq 0 \end{cases}.$$

Exercise 0 (2 points). Given the a $m \times (d+1)$ matrix of augmented points (i.e., the X matrix) and a column vector θ of length $d+1$, implem to compute the value of the linear discriminant at each point. That is, the function should return a (column) vector y where the $y_i = \theta^T x_i$.

In [9]:
```
def lin_discr (X, theta):
    ### BEGIN SOLUTION
    return X.dot(theta)
    ### END SOLUTION
```

In [10]:
```
# Test cell: `lin_discr__check`
import random
theta_test = [random.random() for _ in range (3)]
x0_test = [random.random() for _ in range (2)]
x1_test = [(-theta_test[2] - theta_test[0]*x0) / theta_test[1] for x0 in x0_test]
X_test = np.array ([[x0*2 for x0 in x0_test] + [x0*0.5 for x0 in x0_test],
                    x1_test + x1_test,
                    [1.0, 1.0, 1.0, 1.0],]).T
print(X_test, "\n")
LD_test = lin_discr(X_test, np.array([theta_test]).T)
print (LD_test)
assert (LD_test[:2] > 0).all ()
assert (LD_test[2:] < 0).all ()
print("\n(Passed.)")
```

```
[[ 1.05035323 -0.52024135  1.        ]
 [ 1.49196858 -0.59315241  1.        ]
 [ 0.26258831 -0.52024135  1.        ]
 [ 0.37299215 -0.59315241  1.        ]]

[[ 0.13320893]
 [ 0.1892159 ]
 [-0.06660446]
 [-0.09460795]]

(Passed.)
```

Exercise 1 (2 points). Implement the *heaviside function*, $H(y)$. Your function should allow for an arbitrary *matrix* of input values and should app heaviside function to each element. In the returned matrix, the elements should have a **floating-point type**.

Example, the code snippet

```
A = np.array([[-0.5, 0.2, 0.0],
              [4.2, 3.14, -2.7]])
```

```
print(heaviside(A))
```

should display

```
[[ 0.  1.  0.]
 [ 1.  1.  0.]]
```

There are several possible approaches that lead to one-line solutions. One uses only logical and arithmetic operators, which you will rec implemented as elementwise operations for Numpy arrays. Another uses Numpy's `sign()` (http://docs.scipy.org/doc/numpy/reference/generated/numpy.sign.html) function.

In [11]:
```
def heaviside(Y):
    ### BEGIN SOLUTION
    return 1.0*(Y > 0.0)

    # Alternative solution:
    #return (np.sign(Y) > 0) * 1.0
    ### END SOLUTION
```

In [12]:
```
# Test cell: `heaviside__check`

Y_test = np.array([[-2.3, 1.2, 7.],
                   [0.0, -np.inf, np.inf]])
H_Y_test = heaviside(Y_test)

print("Y:\n", Y_test)
print("\nH(Y):\n", H_Y_test)

assert (H_Y_test.astype(int) == np.array([[0, 1, 1], [0, 0, 1]])).all ()

print ("\n(Passed.)")
```

```
Y:
 [[-2.3  1.2  7. ]
 [ 0.  -inf  inf]]

H(Y):
 [[0. 1. 1.]
 [0. 0. 1.]]

(Passed.)
```

For the next exercise, we'll need the following functions.

In [13]:
```
def heaviside_int(Y):
    """Evaluates the heaviside function, but returns integer values."""
    return heaviside(Y).astype(dtype=int)

def gen_lin_discr_labels(points, theta, fun=heaviside_int):
    """
    Given a set of points and the coefficients of a linear
    discriminant, this function returns a set of labels for
    the points with respect to this discriminant.
    """
    score = lin_discr(points, theta)
    labels = fun(score)
    return labels

def plot_lin_discr(theta, df, x="x_0", y="x_1", hue="label",
                   palette={0: "red", 1: "olive"}, size=5,
                   linewidth=2):
    lm = sns.lmplot(x=x, y=y, hue=hue, data=df, palette=palette,
                    size=size, fit_reg=False)

    x_min, x_max = df[x].min(), df[x].max()
    y_min, y_max = df[y].min(), df[y].max()

    x1_min = (-theta[2][0] - theta[0][0]*x_min) / theta[1][0]
    x1_max = (-theta[2][0] - theta[0][0]*x_max) / theta[1][0]
    plt.plot([x_min, x_max], [x1_min, x1_max], linewidth=linewidth)

    def expand_interval(x_limits, percent=10.0):
        x_min, x_max = x_limits[0], x_limits[1]
        if x_min < 0:
            x_min *= 1.0 + 1e-2*percent
        else:
            x_min *= 1.0 - 1e-2*percent
        if x_max > 0:
            x_max *= 1.0 + 1e-2*percent
        else:
            x_max *= 1.0 + 1e-2*percent
        return (x_min, x_max)
    x_view = expand_interval((x_min, x_max))
    y_view = expand_interval((y_min, y_max))
```

```
    lm.axes[0,0].set_xlim(x_view[0], x_view[1])
    lm.axes[0,0].set_ylim(y_view[0], y_view[1])

def mark_matches(a, b, exact=False):
    """
    Given two Numpy arrays of {0, 1} labels, returns a new boolean
    array indicating at which locations the input arrays have the
    same label (i.e., the corresponding entry is True).

    This function can consider "inexact" matches. That is, if `exact`
    is False, then the function will assume the {0, 1} labels may be
    regarded as the same up to a swapping of the labels. This feature
    allows

      a == [0, 0, 1, 1, 0, 1, 1]
      b == [1, 1, 0, 0, 1, 0, 0]

    to be regarded as equal. (That is, use `exact=False` when you
    only care about "relative" labeling.)
    """
    assert a.shape == b.shape
    a_int = a.astype(dtype=int)
    b_int = b.astype(dtype=int)
    all_axes = tuple(range(len(a.shape)))
    assert ((a_int == 0) | (a_int == 1)).all()
    assert ((b_int == 0) | (b_int == 1)).all()

    exact_matches = (a_int == b_int)
    if exact:
        return exact_matches

    assert exact == False
    num_exact_matches = np.sum(exact_matches)
    if (2*num_exact_matches) >= np.prod(a.shape):
        return exact_matches
    return exact_matches == False # Invert

def count_matches(a, b, exact=False):
    """
    Given two sets of {0, 1} labels, returns the number of mismatches.

    This function can consider "inexact" matches. That is, if `exact`
    is False, then the function will assume the {0, 1} labels may be
    regarded as similar up to a swapping of the labels. This feature
    allows

      a == [0, 0, 1, 1, 0, 1, 1]
      b == [1, 1, 0, 0, 1, 0, 0]

    to be regarded as equal. (That is, use `exact=False` when you
    only care about "relative" labeling.)
    """
    matches = mark_matches(a, b, exact=exact)
    return int(matches.sum())
```

Exercise 2 (2 points). For the synthetic data you loaded above, try by hand to find a value for θ such that $H(\theta^T x)$ "best" separates the two clus this θ in a variable named `my_theta`, which should be a Numpy *column vector*. That is, define `my_theta` here using a line like:

```
my_theta = np_col_vec([3., 0., -1.])
```

where `np_col_vec` is defined below and the list of values are your best guesses at discriminating coefficients. The test code will check that your makes no more than ten misclassifications.

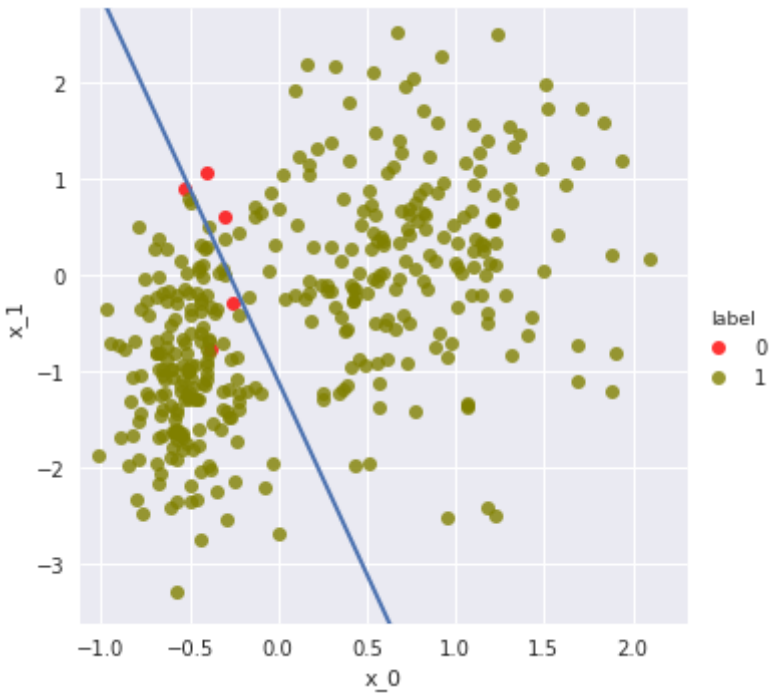
> Hint: We found a set of coefficients that commits just 5 errors for the 375 input points.

In [14]:
```
def np_col_vec (list_values):
    """Returns a Numpy column vector for the given list of scalar values."""
    return np.array ([list_values]).T

# Redefine `my_theta` as instructed above to reduce the number of mismatches:
my_theta = np_col_vec([-1., 3., 0.]) # 123 mismatches
### BEGIN SOLUTION
my_theta = np_col_vec([-6.5, -1., -1.35]) # 5 mismatches
my_theta = np_col_vec([-2., -0.5, -0.55]) # 5 mismatches
### END SOLUTION
```

In [15]:
```
# Here are the labels generated by your discriminant:
my_labels = gen_lin_discr_labels(points, my_theta)

# Here is a visual check:
num_mismatches = len(labels) - count_matches(labels, my_labels)
print ("Detected", num_mismatches, "out of", len(labels), "mismatches.")

df_matches = df.copy ()
```

```
df_matches['label'] = mark_matches (my_labels, labels).astype (dtype=int)

mpl.rc("savefig", dpi=100) # Adjust for higher-resolution figures
plot_lin_discr (my_theta, df_matches)

assert num_mismatches <= 10
```

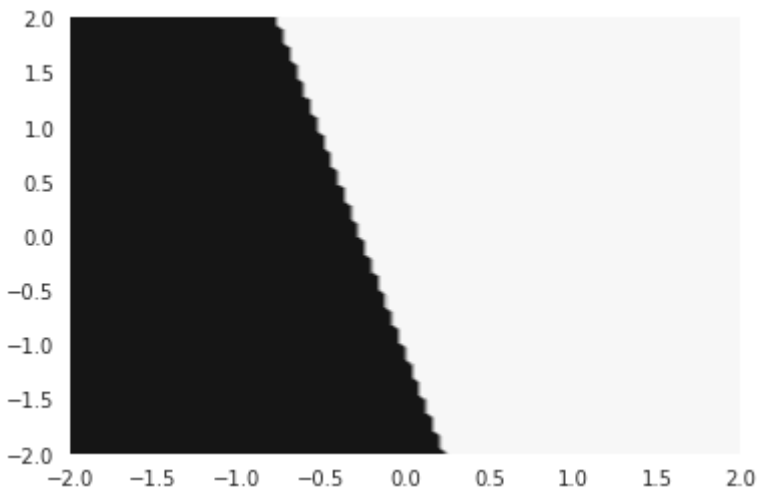
```
Detected 5 out of 375 mismatches.
```



How the heaviside divides the space. The heaviside function, $H(\theta^T x)$, enforces a sharp boundary between classes around the $\theta^T x = 0$ lin following code produces a contour plot (https://matplotlib.org/api/_as_gen/matplotlib.axes.Axes.contourf.html) to show this effect: there will be a line between 0 and 1 values, with one set of values shown as a solid dark area and the remaining as a solid light-colored area.

In [16]:
```
x0 = np.linspace(-2., +2., 100)
x1 = np.linspace(-2., +2., 100)
x0_grid, x1_grid = np.meshgrid(x0, x1)
h_grid = heaviside(my_theta[2] + my_theta[0]*x0_grid + my_theta[1]*x1_grid)
plt.contourf(x0, x1, h_grid)
```

Out[16]:
```
<matplotlib.contour.QuadContourSet at 0x7f0ec25ad978>
```



Part 1: The logistic (or sigmoid) function as an alternative discriminant

As the lobsters example suggests, real data are not likely to be cleanly separable, especially when the number of features we have at our dispos small.

Since the labels are 0 or 1, you could look for a way to interpret labels as *probabilities* rather than as hard (0 or 1) labels. One such function is th *function*, also referred to as the *logit* or *sigmoid* (https://en.wikipedia.org/wiki/Sigmoid_function) function.

$$G(y) \equiv \frac{1}{1+e^{-y}}$$

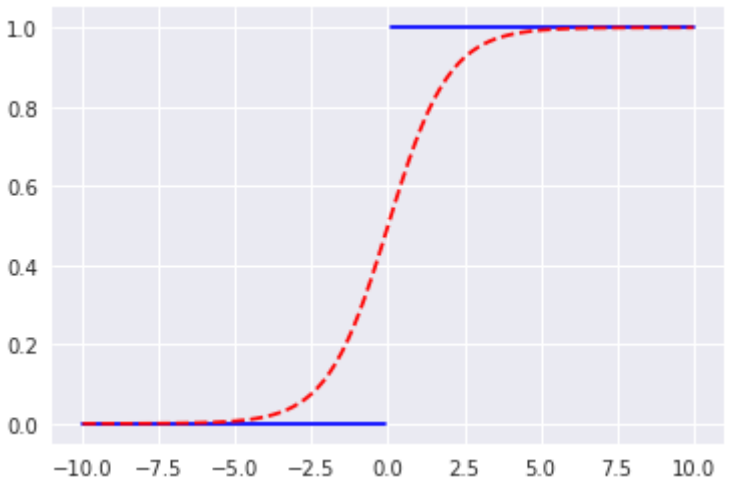
The logistic function takes any value in the range $(-\infty, +\infty)$ and produces a value in the range $(0, 1)$. Thus, given a value y, we can interpre conditional probability that the label is 1 given y, i.e., $G(y) \equiv \Pr[\text{label is } 1 \mid y]$.

Exercise 3 (2 points). Implement the logistic function. Inspect the resulting plot of $G(y)$ in 1-D and then the contour plot of $G(\theta^T x)$. Your functi accept a Numpy matrix of values, `Y`, and apply the sigmoid elementwise.

In [17]:
```
def logistic(Y):
    ### BEGIN SOLUTION
    return 1.0 / (1.0 + np.exp (-Y))
    ### END SOLUTION

# Plot your function for a 1-D input.
y_values = np.linspace(-10, 10, 100)
```

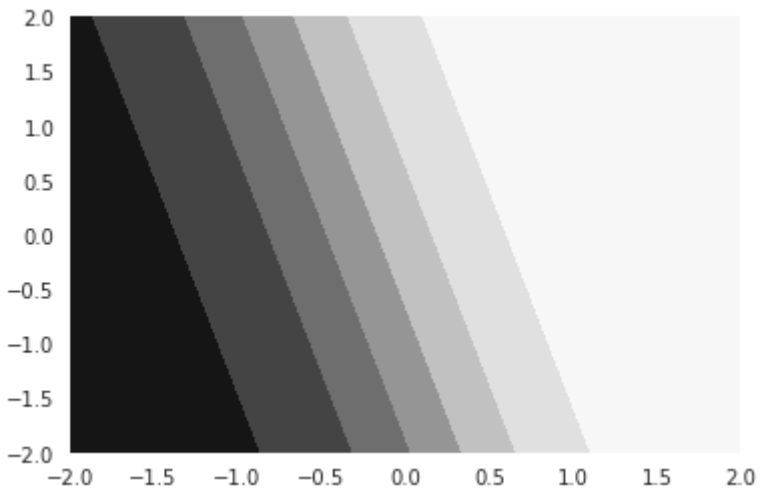
```
mpl.rc("savefig", dpi=120) # Adjust for higher-resolution figures
sns.set_style("darkgrid")
y_pos = y_values[y_values > 0]
y_rem = y_values[y_values <= 0]
plt.plot(y_rem, heaviside (y_rem), 'b')
plt.plot(y_pos, heaviside (y_pos), 'b')
plt.plot(y_values, logistic (y_values), 'r--')
#sns.regplot (y_values, heaviside (y_values), fit_reg=False)
#sns.regplot (y_values, logistic (y_values), fit_reg=False)
```

Out[17]: [<matplotlib.lines.Line2D at 0x7f0ec257c2b0>]



In [18]:
```
# Test cell: `logistic__check`

assert logistic(np.log(3)) == 0.75
assert logistic(-np.log(3)) == 0.25

g_grid = logistic(my_theta[2] + my_theta[0]*x0_grid + my_theta[1]*x1_grid)
plt.contourf (x0, x1, g_grid)
assert ((np.round(g_grid) - h_grid).astype(int) == 0).all()

print ("\n(Passed.)")
```

```
(Passed.)
```



Exercise 4 (*optional*; ungraded). Consider a set of 1-D points generated by a *mixture of Gaussians*. That is, suppose that there are two Gaussia over the 1-dimensional variable, $x \in (-\infty, +\infty)$, that have the *same* variance (σ^2) but *different* means (μ_0 and μ_1). Show that the conditiona observing a point labeled "1" given x may be written as,

$$\Pr[l = 1 \mid x] \propto \frac{1}{1 + e^{-(\theta_0 x + \theta_1)}},$$

for a suitable definition of θ_0 and θ_1.

Hints. Since the points come from Gaussian distributions,

$$\Pr[x \mid l] \equiv \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu_l)^2}{2\sigma^2}\right).$$

To rewrite $\Pr[l \mid x]$ in terms of $\Pr[x \mid l]$, recall *Bayes's rule* (also: Bayes's theorem (https://en.wikipedia.org/wiki/Bayes%27_theorem)):

$$\Pr[l = 1 \mid x] = \frac{\Pr[x \mid l = 1] \Pr[l = 1]}{\Pr[x]},$$

where the denominator can be expanded as

$$\Pr[x] = \Pr[x \mid l = 0] \Pr[l = 0] + \Pr[x \mid l = 1] \Pr[l = 1].$$

You may assume the prior probabilities of observing a 0 or 1 are given by $\Pr[l = 0] \equiv p_0$ and $\Pr[l = 1] \equiv p_1$.

> The point of this derivation is to show you that the definition of the logistic function does not just arise out of thin air. It also hints that you expect a final algorithm for logistic regression based on using $G(y)$ as the discriminant will work well when the classes are best explain mixture of Gaussians.

Generalizing to d-dimensions. The preceding exercise can be generalized to d-dimensions. Let θ and x be $(d+1)$-dimensional points. Ther

$$\Pr\left[l = 1 \,|\, x\right] \propto \frac{1}{1 + \exp(-\theta^T x)}.$$

Exercise 5 (*optional*; ungraded). Verify the following properties of the logistic function, $G(y)$.

$$\begin{aligned}
G(y) &= \tfrac{e^y}{e^y+1} && \text{(P1)} \\
G(-y) &= 1 - G(y) && \text{(P2)} \\
\frac{dG}{dy} &= G(y)G(-y) && \text{(P3)} \\
\frac{d}{dy}\left[\ln G(y)\right] &= G(-y) && \text{(P4)} \\
\frac{d}{dy}\ln[1 - G(y)] &= -G(y) && \text{(P5)}
\end{aligned}$$

Answers. In all of the derivations below, we use the fact that $G(y) > 0$.

(P1). Multiply the numerator and denominator by e^y.

(P2). Start with the right-hand side, $1 - G(y)$, apply some algebra, and then apply (P1).

$$1 - G(y) = \frac{e^y + 1}{e^y + 1} - \frac{e^y}{e^y + 1} = \frac{1}{e^y + 1} \cdot \frac{e^{-y}}{e^{-y}} = \frac{e^{-y}}{e^{-y} + 1} = G(-y).$$

(P3). By direct calculation and application of (P1):

$$\frac{dG}{dy} = \frac{d}{dy}\left(1 + e^{-y}\right)^{-1} = -\left(1 + e^{-y}\right)^{-2} \cdot (-e^{-y}) = \underbrace{\frac{1}{1 + e^{-y}}}_{=G(y)} \cdot \underbrace{\frac{e^{-y}}{1 + e^{-y}}}_{=G(-y)} = G(y) \cdot G(-y).$$

(P4). By the chain rule and application of (P3):

$$\frac{d}{dy}\ln G(y) = \left(\frac{d}{dG}\ln G\right)\frac{dG}{dy} = \frac{1}{G(y)} \cdot G(y)G(-y) = G(-y).$$

(P5). By combining (P2), variable substitution and the chain rule, and (P4),

$$\frac{d}{dy}\ln[1 - G(y)] = \frac{d}{dy}\ln G(-y) = \underbrace{\left[\frac{d}{dz}\ln G(z)\right] \cdot \frac{dz}{dy}}_{\text{Let } z \equiv -y} = G(-z) \cdot (-1) = -G(y).$$

Part 2: Determining the discriminant via maximum likelihood estimation

Previously, you determined θ for our synthetic dataset by hand. Can you compute a good θ automatically? One of the standard techniques in sta perform a *maximum likelihood estimation* (MLE) of a model's parameters, θ. Indeed, you may have seen or used MLE to derive the normal equa regression in a more "statistically principled" way.

"Likelihood" as an objective function. MLE derives from the following idea. Consider the joint probability of observing all of the labels, given tl the parameters, θ:

$$\Pr[y \,|\, X, \theta].$$

Suppose these observations are independent and identically distributed (i.i.d.). Then the joint probability can be factored as the product of indivic probabilities,

$$\begin{aligned}
\Pr[y \,|\, X, \theta] = \Pr[y_0, \ldots, y_{m-1} \,|\, \hat{x}_0, \ldots, \hat{x}_{m-1}, \theta] &= \Pr[y_0 \,|\, \hat{x}_0, \theta] \cdots \Pr[y_{m-1} \,|\, \hat{x}_{m-1}, \theta] \\
&= \prod_{i=0}^{m-1} \Pr[y_i \,|\, \hat{x}_i, \theta].
\end{aligned}$$

The *maximum likelihood principle* says that you should choose θ to maximize the chances (or "likelihood") of seeing these particular observatior $\Pr[y \,|\, X, \theta]$ is now an objective function to maximize.

For both mathematical and numerical reasons, we will use the *logarithm* of the likelihood, or *log-likelihood*, as the objective function instead. Let'

$$\begin{aligned}
\mathcal{L}(\theta; y, X) &\equiv \log\left\{\prod_{i=0}^{m-1} \Pr[y_i \,|\, \hat{x}_i, \theta]\right\} \\
&= \sum_{i=0}^{m-1} \log \Pr[y_i \,|\, \hat{x}_i, \theta].
\end{aligned}$$

> We are using the symbol $\log$, which could be taken in any convenient base, such as the natural logarithm ($\ln y$) or the information theor base-two logarithm ($\log_2 y$).

The MLE fitting procedure then consists of two steps:

- For the problem at hand, decide on a model of $\Pr[y_i \,|\, \hat{x}_i, \theta]$.
- Run any optimization procedure to find the θ that maximizes $\mathcal{L}(\theta; y, X)$.

Part 3: MLE for logistic regression

Let's say you have decided that the logistic function, $G(\hat{x}_i^T\theta) = G(\theta^T\hat{x}_i)$, is a good model of the probability of producing a label y_i given the c $\hat{x}_i^T$. Under the i.i.d. assumption, you can interpret the label y_i as the result of flipping a coin, or a Bernoulli trial (https://en.wikipedia.org/wiki/Ber where the probability of success ($y_i = 1$) is defined as $g_i = g_i(\theta) \equiv G(\hat{x}_i^T\theta)$. Thus,

$$\Pr[y_i \mid \hat{x}_i, \theta] \equiv g_i^{y_i} \cdot (1 - g_i)^{1-y_i}.$$

The log-likelihood in turn becomes,

$$\begin{aligned}
\mathcal{L}(\theta; y, X) &= \sum_{i=0}^{m-1} y_i \ln g_i + (1 - y_i) \ln(1 - g_i) \\
&= \sum_{i=0}^{m-1} y_i \ln \frac{g_i}{1 - g_i} + \ln(1 - g_i) \\
&= \sum_{i=0}^{m-1} y_i \theta^T \hat{x}_i + \ln(1 - g_i).
\end{aligned}$$

You can write the log-likelihood more compactly in the language of linear algebra.

Convention 1. Let $u \equiv (1, \ldots, 1)^T$ be a column vector of all ones, with its length inferred from context. Let $A = \begin{pmatrix} a_0 & a_1 & \cdots & a_{n-1} \end{pmatrix}$ be where $\{a_i\}$ denote its n columns. Then, the sum of the columns is

$$\sum_{i=0}^{n-1} a_i = (a_0\ a_1\ \cdots\ a_{n-1}) \cdot \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} = Au.$$

Convention 2. Let $A = (a_{ij})$ be any matrix and let $f(y)$ be any function that we have defined by default to accept a scalar argument y and pr result. For instance, $f(y) = \ln y$ or $f(y) = G(y)$. Then, assume that $B = f(A)$ applies $f(\cdot)$ elementwise to A, returning a matrix B whose $b_{ij} = f(a_{ij})$.

With these notational conventions, convince yourself that these are two different ways to write the log-likelihood for logistic regression.

$$\begin{aligned}
\text{(V1)} \quad \mathcal{L}(\theta; y, X) &= y^T \ln G(X\theta) + (u - y)^T \ln[u - G(X\theta)] \\
\text{(V2)} \quad \mathcal{L}(\theta; y, X) &= y^T X\theta + u^T \ln G(-X\theta)
\end{aligned}$$

Exercise 6 (2 points). Implement the log-likelihood function in Python by defining a function with the following signature:

```
def log_likelihood (theta, y, X):
    ...
```

> To compute the elementwise logarithm of a matrix or vector, use Numpy's log (https://docs.scipy.org/doc/numpy/reference/generated/numpy.log.html) function.

In [19]:
```
def log_likelihood(theta, y, X):
### BEGIN SOLUTION
    u = np.ones((len (X), 1)) # column of all ones
    z = X.dot(theta)
    return y.T.dot(z) + u.T.dot(np.log(logistic(-z)))

def log_likelihood_alt(theta, y, X):
    z = X.dot(theta)
    g = logistic(z)
    return y.T.dot(np.log(g)) + (1.0-y).T.dot(np.log(1.0-g))
### END SOLUTION
```

In [20]:
```
# Test cell: `log_likelihood__check`

if False:
    d_soln = 10
    m_soln = 1000
    theta_soln = np.random.random ((d_soln+1, 1)) * 2.0 - 1.0
    y_soln = np.random.randint (low=0, high=2, size=(m_soln, 1))
    X_soln = np.random.random ((m_soln, d_soln+1)) * 2.0 - 1.0
    X_soln[:, 0] = 1.0
    L_soln = log_likelihood (theta_soln, y_soln, X_soln)
    np.savez_compressed('log_likelihood_soln',
                        d_soln, m_soln, theta_soln, y_soln, X_soln, L_soln)

npzfile_soln = np.load('{}log_likelihood_soln.npz'.format(DATA_PATH))
d_soln = npzfile_soln['arr_0']
m_soln = npzfile_soln['arr_1']
theta_soln = npzfile_soln['arr_2']
y_soln = npzfile_soln['arr_3']
X_soln = npzfile_soln['arr_4']
L_soln = npzfile_soln['arr_5']
```

```
L_you = log_likelihood(theta_soln, y_soln, X_soln)
your_err = np.max(np.abs(L_you/L_soln - 1.0))
display(Math(r'\left\|\dfrac{\mathcal{L}_{\tiny \mbox{yours}} - \mathcal{L}_{\tiny \mbox{solution}}}{\m
{\tiny \mbox{solution}}}\right\|_\infty \approx %g' % your_err))
assert your_err <= 1e-12

print ("\n(Passed.)")
```

$$\left\| \frac{\mathcal{L}_{\text{yours}} - \mathcal{L}_{\text{solution}}}{\mathcal{L}_{\text{solution}}} \right\|_\infty \approx 0$$

```
(Passed.)
```

Part 4: Computing the MLE solution via gradient ascent: theory

To optimize the log-likelihood with respect to the parameters, θ, you want to "set the derivative to zero" and solve for θ.

For example, recall that in the case of linear regression via least squares minimization, carrying out this process produced an *analytic* solution fo parameters, which was to solve the normal equations.

Unfortunately, for logistic regression---or for most log-likelihoods you are likely to ever write down---you *cannot* usually derive an analytic solutio you will need to resort to numerical optimization procedures.

Gradient ascent, in 1-D. A simple numerical algorithm to maximize a function is *gradient ascent* (or *steepest ascent*). If instead you are minimiz function, then the equivalent procedure is gradient (or steepest) *descent*. Here is the basic idea in 1-D.

Suppose we wish to find the maximum of a scalar function $f(x)$ in one dimension. At the maximum, $\dfrac{df(x)}{dx} = 0$.

Suppose instead that $\dfrac{df}{dx} \neq 0$ and consider the value of f at a nearby point, $x + s$, as given approximately by a truncated Taylor series:

$$f(x+s) = f(x) + s\frac{df(x)}{dx} + \mathcal{O}(s^2).$$

To make progress toward maximizing $f(x)$, you'd like to choose s so that $f(x+s) > f(x)$. One way is to choose $s = \alpha \cdot \text{sign}\left(\dfrac{df}{dx}\right)$, whe $0 < \alpha \ll 1$ is "small:"

$$f\left(x + \alpha \cdot \text{sign}\left(\frac{df}{dx}\right)\right) \approx f(x) + \alpha\left|\frac{df}{dx}\right| + \mathcal{O}(\alpha^2).$$

If α is small enough, then you can neglect the $\mathcal{O}(\alpha^2)$ term and $f(x+s)$ will be larger than $f(x)$, thus making progress toward finding a maxi

This scheme is the basic idea: starting from some initial guess x, refine the guess by taking a small step s *in the direction* of the derivative, i.e.,

Gradient ascent in higher dimensions. Now suppose x is a vector rather than a scalar. Then the value of f at a nearby point $f(x+s)$, wher becomes

$$f(x+s) = f(x) + s^T \nabla_x f(x) + \mathcal{O}(\|s\|^2),$$

where $\nabla_x f(x)$ is the gradient of f with respect to x. As in the 1-D case, you want a step s such that $f(x+s) > f(x)$. To make as much pro possible, let's choose s to be parallel to $\nabla_x\, f(x)$, that is, proportional to the gradient:

$$s \equiv \alpha \frac{\nabla_x\, f(x)}{\|\nabla_x\, f(x)\|}.$$

Again, α is a fudge (or "gentle nudge?") factor. You need to choose it to be small enough that the high-order terms of the Taylor approximation b negligible, yet large enough that you can make reasonable progress.

The gradient ascent procedure applied to MLE. Applying gradient ascent to the problem of maximizing the log-likelihood leads to the followin

- Start with some initial guess, $\theta(0)$.
- At each iteration $t \geq 0$ of the procedure, let $\theta(t)$ be the current guess.
- Compute the direction of steepest ascent by evaluating the gradient, $\Delta_t \equiv \nabla_{\theta(t)} \left\{\mathcal{L}(\theta(t); y, X)\right\}$.
- Define the step to be $s_t \equiv \alpha \dfrac{\Delta_t}{\|\Delta_t\|}$, where α is a suitably chosen fudge factor.
- Take a step in the direction of the gradient, $\theta(t+1) \leftarrow \theta(t) + s_t$.
- Stop when the parameters don't change much *or* after some maximum number of steps.

This procedure should remind you of one you saw in a prior notebook (the least mean square algorithm for online regression!). As was true at th tricky bit is how to choose α.

> There is at least one difference between this procedure and the online regression procedure you learned earlier. Here, we are optimizin the *full* dataset rather than processing data points one at a time. (That is, the step iteration variable t used above is not used in exactly t same way as the step iteration in LMS.)
>
> Another question is, how do we know this procedure will converge to the global maximum, rather than, say, a local maximum? For that y need a deeper analysis of a specific $\mathcal{L}(\theta; y, X)$, to show, for instance, that it is convex in θ.

Implementing logistic regression using MLE by gradient ascent

Let's apply the gradient ascent procedure to the logistic regression problem, in order to determine a good θ.

Exercise 7 (*optional*; ungraded). Show the following.

$$\nabla_\theta \left\{\mathcal{L}(\theta; y, X)\right\} = X^T \left[y - G(X \cdot \theta)\right].$$

Answer. From (V2),

$$\mathcal{L}(\theta; y, X) = y^T X\theta + u^T \ln G(-X\theta).$$

Thus,

$$\nabla_\theta \{\mathcal{L}(\theta; y, X)\} = \nabla_\theta (y^T X\theta) + \nabla_\theta \left(u^T \ln G(-X\theta)\right).$$

Let's consider each term in turn.

For the first term, apply the gradient identities to obtain

$$\nabla_\theta \left(y^T X\theta\right) = \nabla_\theta \left(\theta^T X^T y\right) = X^T y.$$

For the second term, recall the scalar interpretation of $u^T \ln G(-X\theta)$.

$$u^T \ln G(-X\theta) = \sum_{j=0}^{m-1} \ln G\left(-\hat{x}_j^T \theta\right).$$

The i-th component of the gradient is

$$\frac{\partial}{\partial \theta_i} \sum_{j=0}^{m-1} \ln G\left(-\hat{x}_j^T \theta\right) = \sum_{j=0}^{m-1} \frac{\partial}{\partial \theta_i} \ln G\left(-\hat{x}_j^T \theta\right).$$

Let's evaluate the summand:

$$\frac{\partial}{\partial \theta_i} \ln G\left(-\hat{x}_j^T \theta\right) = \underbrace{\left[\frac{d}{dz} \ln G(z)\right] \cdot \left[\frac{\partial z}{\partial \theta_i}\right]}_{\text{Let } z \equiv -\hat{x}_j^T \theta} = G(-z) \cdot \frac{\partial}{\partial \theta_i}\left(-\hat{x}_j^T \theta\right) = -G\left(\hat{x}_j^T \theta\right) \cdot x_{ji}.$$

Thus, the i-th component of the gradient becomes

$$\left[\nabla_\theta \left(u^T \ln G(-X\theta)\right)\right]_i = -\sum_{j=0}^{m-1} G\left(\hat{x}_j^T \theta\right) \cdot x_{ji}.$$

In other words, the full gradient vector is

$$\nabla_\theta \left(u^T \ln G(-X\theta)\right) = -X^T G(X\theta).$$

Putting the two components together,

$$\nabla_\theta \{\mathcal{L}(\theta; y, X)\} = X^T y - X^T G(X\theta) = X^T \left[y - G(X\theta)\right].$$

Exercise 8 (2 points). Implement a function to compute the gradient of the log-likelihood. Your function should have the signature,

```
def grad_log_likelihood (theta, y, X):
    ...
```

In [21]:
```
def grad_log_likelihood(theta, y, X):
    """Returns the gradient of the log-likelihood."""
    ### BEGIN SOLUTION
    return X.T.dot(y - logistic(X.dot(theta)))
    ### END SOLUTION
```

In [22]:
```
# Test cell: `grad_log_likelihood_code__check`

if False:
    d_grad_soln = 6
    m_grad_soln = 399
    theta_grad_soln = np.random.random((d_grad_soln+1, 1)) * 2.0 - 1.0
    y_grad_soln = np.random.randint(low=0, high=2, size=(m_grad_soln, 1))
    X_grad_soln = np.random.random((m_grad_soln, d_grad_soln+1)) * 2.0 - 1.0
    X_grad_soln[:, 0] = 1.0
    L_grad_soln = grad_log_likelihood(theta_grad_soln, y_grad_soln, X_grad_soln)
    np.savez_compressed('grad_log_likelihood_soln',
                        d_grad_soln, m_grad_soln, theta_grad_soln, y_grad_soln, X_grad_soln, L_grad_sol

npzfile_grad_soln = np.load ('{}grad_log_likelihood_soln.npz'.format(DATA_PATH))
d_grad_soln = npzfile_grad_soln['arr_0']
m_grad_soln = npzfile_grad_soln['arr_1']
theta_grad_soln = npzfile_grad_soln['arr_2']
y_grad_soln = npzfile_grad_soln['arr_3']
X_grad_soln = npzfile_grad_soln['arr_4']
```

```
L_grad_soln = npzfile_grad_soln['arr_5']

L_grad_you = grad_log_likelihood (theta_grad_soln, y_grad_soln, X_grad_soln)
your_grad_err = np.max (np.abs (L_grad_you/L_grad_soln - 1.0))
display (Math (r'\left\|\dfrac{\nabla\, \mathcal{L}_{\tiny \mbox{yours}} - \nabla\,\mathcal{L}_{\tiny \
n}}}{\nabla\, \mathcal{L}_{\tiny \mbox{solution}}}\right\|_\infty \approx %g' % your_grad_err))
assert your_grad_err <= 1e-12

print ("\n(Passed.)")
```

$$\left\| \frac{\nabla \mathcal{L}_{\text{yours}} - \nabla \mathcal{L}_{\text{solution}}}{\nabla \mathcal{L}_{\text{solution}}} \right\|_\infty \approx 2.22045e-16$$

```
(Passed.)
```

Exercise 9 (4 points). Implement the gradient ascent procedure to determine θ, and try it out on the sample data.

Recall the procedure (repeated from above):

- Start with some initial guess, $\theta(0)$.
- At each iteration $t \geq 0$ of the procedure, let $\theta(t)$ be the current guess.
- Compute the direction of steepest ascent by evaluating the gradient, $\Delta_t \equiv \nabla_{\theta(t)} \{\mathcal{L}(\theta(t); y, X)\}$.
- Define the step to be $s_t \equiv \alpha \dfrac{\Delta_t}{\|\Delta_t\|}$, where α is a suitably chosen fudge factor.
- Take a step in the direction of the gradient, $\theta(t+1) \leftarrow \theta(t) + s_t$.
- Stop when the parameters don't change much *or* after some maximum number of steps.

In the code skeleton below, we've set up a loop to run a fixed number, `MAX_STEP`, of gradient ascent steps. Also, when normalizing the step Δ_t, norm.

> In your solution, we'd like you to store *all* guesses in the matrix `thetas`, so that you can later see how the $\theta(t)$ values evolve. To extract particular column `t`, use the notation, `theta[:, t:t+1]`. This notation is necessary to preserve the "shape" of the column as a column

In [23]:
```
ALPHA = 0.1
MAX_STEP = 250

# Get the data coordinate matrix, X, and labels vector, y
X = points
y = labels.astype(dtype=float)

# Store *all* guesses, for subsequent analysis
thetas = np.zeros((3, MAX_STEP+1))

for t in range(MAX_STEP):
    # Fill in the code to compute thetas[:, t+1:t+2]
    ### BEGIN SOLUTION
    theta_t = thetas[:, t:t+1]
    delta_t = grad_log_likelihood(theta_t, y, X)
    delta_t = delta_t / np.linalg.norm(delta_t, ord=2)
    thetas[:, t+1:t+2] = theta_t + ALPHA*delta_t
    ### END SOLUTION

theta_ga = thetas[:, MAX_STEP:]
print("Your (hand) solution:", my_theta.T.flatten())
print("Computed solution:", theta_ga.T.flatten())

print("\n=== Comparisons ===")
display(Math (r'\dfrac{\theta_0}{\theta_2}:'))
print("Your manual (hand-picked) solution is", my_theta[0]/my_theta[2], \
      ", vs. MLE (via gradient ascent), which is", theta_ga[0]/theta_ga[2])
display(Math (r'\dfrac{\theta_1}{\theta_2}:'))
print("Your manual (hand-picked) solution is", my_theta[1]/my_theta[2], \
      ", vs. MLE (via gradient ascent), which is", theta_ga[1]/theta_ga[2])

print("\n=== The MLE solution, visualized ===")
ga_labels = gen_lin_discr_labels(points, theta_ga)
df_ga = df.copy()
df_ga['label'] = mark_matches(ga_labels, labels).astype (dtype=int)
plot_lin_discr(theta_ga, df_ga)
```

```
Your (hand) solution: [-2.   -0.5  -0.55]
Computed solution: [-15.57666992  -3.03431905  -3.79328353]

=== Comparisons ===
```

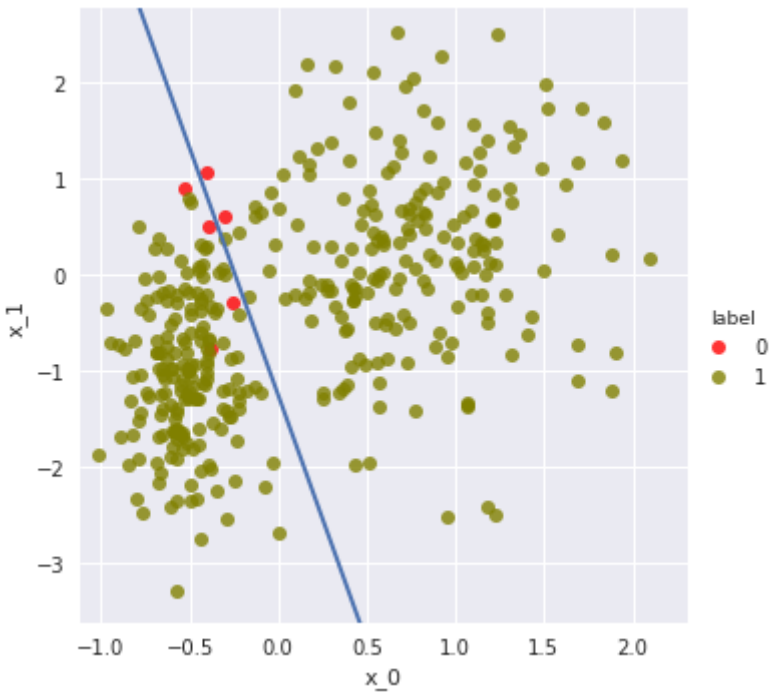
$\dfrac{\theta_0}{\theta_2}$:

```
Your manual (hand-picked) solution is [3.63636364] , vs. MLE (via gradient ascent), which is [4.1063816
```

$\dfrac{\theta_1}{\theta_2}$:

```
Your manual (hand-picked) solution is [0.90909091] , vs. MLE (via gradient ascent), which is [0.7999188
```

```
=== The MLE solution, visualized ===
```



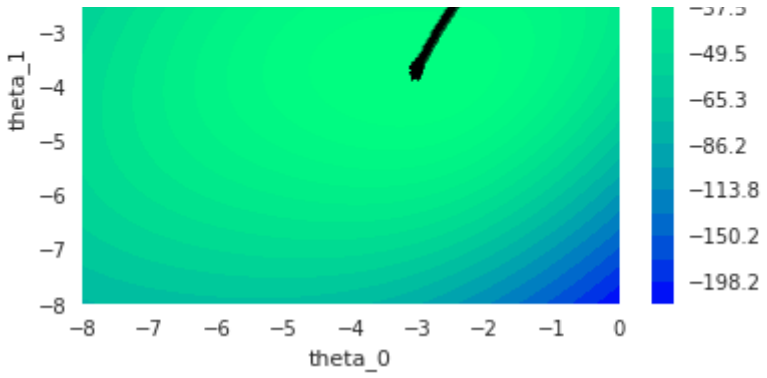
In [24]:
```
print ("\n=== Mismatch counts ===")

my_labels = gen_lin_discr_labels (points, my_theta)
my_mismatches = len (labels) - count_matches (labels, my_labels)
print ("Your manual (hand-picked) solution has", num_mismatches, "mismatches.")

ga_labels = gen_lin_discr_labels (points, theta_ga)
ga_mismatches = len (labels) - count_matches (labels, ga_labels)
print ("The MLE method produces", ga_mismatches, "mismatches.")

assert ga_mismatches <= 8
print ("\n(Passed.)")
```

```
=== Mismatch counts ===
Your manual (hand-picked) solution has 5 mismatches.
The MLE method produces 6 mismatches.

(Passed.)
```

The gradient ascent trajectory. Let's take a look at how gradient ascent progresses. (You might try changing the α parameter and see how it a results.)

In [25]:
```
n_ll_grid = 100
x1 = np.linspace(-8., 0., n_ll_grid)

x2 = np.linspace(-8., 0., n_ll_grid)
x1_grid, x2_grid = np.meshgrid(x1, x2)

ll_grid = np.zeros((n_ll_grid, n_ll_grid))
for i1 in range(n_ll_grid):
    for i2 in range(n_ll_grid):
        theta_i1_i2 = np.array([[thetas[0, MAX_STEP]],
                                [x1_grid[i1][i2]],
                                [x2_grid[i1][i2]]])
        ll_grid[i1][i2] = log_likelihood(theta_i1_i2, y, X)

# Determine a color scale
def v(x):
    return -np.log(np.abs(x))
    return x

def v_inv(v):
    return -np.exp(np.abs(v))
    return v

v_min, v_max = v(ll_grid.min()), v(ll_grid.max())
v_range = v_max - v_min
v_breaks = v_inv(np.linspace(v_min, v_max, 20))

p = plt.contourf(x1, x2, ll_grid, v_breaks, cmap=plt.cm.get_cmap("winter"))
plt.xlabel('theta_0')
plt.ylabel('theta_1')
plt.title('log-likelihood')
plt.colorbar()
plt.plot(thetas[1, :], thetas[2, :], 'k*-')
```

Out[25]:
```
[<matplotlib.lines.Line2D at 0x7f0ec2484dd8>]
```



Part 5 (optional): Numerical optimization via Newton's method

The fudge factor, α, in gradient ascent should give you pause. Can you choose the step size or direction in a better or more principled way?

One idea is *Newton's method* (http://www.math.uiuc.edu/documenta/vol-ismp/13_deuflhard-peter.pdf), summarized below.

> This part of the notebook has additional exercises, but they are all worth 0 points. (So if you submit something that is incomplete or fails test cells, you won't lose any points.)

The basic idea, in 1-D. Suppose you start at a point x and, assuming you are not yet at the optimum, you have decided to take a step of size s at $f(x+s)$.

How do you choose s? In gradient ascent, you do so by following the gradient, which points in an "upward" direction.

In Newton's method, you will pick s in a different way: choose s to maximize $f(x+s)$.

That should strike you as circular; the whole problem from the beginning was to maximize $f(x)$. The trick, in this case, is not to maximize $f(x$ - rather, let's replace it with some approximation, $q(s) \approx f(x+s)$, and maximize $q(s)$ instead.

A simple choice for $q(s)$ is a *quadratic* function in s. This choice is motivated by two factors: (a) since it's quadratic, it should have some sort of (and hopefully an actual maximum), and (b) it is a higher-order approximation than a linear one, and so hopefully more accurate than a linear on

$$f(x+s) \quad \approx \quad f(x) + s\frac{df}{dx} + \frac{1}{2}s^2\frac{d^2f}{dx^2} \quad \equiv \quad q(s).$$

To maximize $q(s)$, take its derivative and then solve for the s_* such that $q(s_*) = 0$:

$$\begin{aligned} \left.\frac{dq}{ds}\right|_{s=s_*} &= \frac{df}{dx} + s_*\frac{d^2f}{dx^2} = 0 \\ \implies s_* &= -\frac{df}{dx}\left(\frac{d^2f}{dx^2}\right)^{-1}. \end{aligned}$$

That is, the optimal step s_* is the negative of the first derivative of f divided by its second derivative.

Generalizing to higher dimensions. To see how this procedure works in higher dimensions, you will need not only the gradient of $f(x)$, but al which is the moral equivalent of a second derivative.

Definition: **the Hessian.** Let $f(v)$ be a function that takes a *vector* v of length n as input and returns a scalar. The *Hessian* of $f(v)$ is an $n \times n$ $H_v(f)$, whose entries are all n^2 possible second-order partial derivatives with respect to the components of v. That is, let h_{ij} be the (i,j) elem Then we define

$$h_{ij} \equiv \frac{\partial^2}{\partial v_i \partial v_j} f(v).$$

Armed with a Hessian, the Newton step is defined as follows, by direct analogy to the 1-D case. First, the Taylor series approximation of $f(x+$ multidimensional variables is, as it happens,

$$f(x+s) \quad \approx \quad f(x) + s^T \nabla_x f + \frac{1}{2} s^T H_x(f)\, s \quad \equiv \quad q(s).$$

As in the 1-D case, we want to find an extreme point of $q(s)$. Taking its "derivative" (gradient), $\nabla_s q$, and setting it to 0 yields,

$$\begin{aligned} \nabla_s\, q(s) &= \nabla_x\, f(x) + H_x(f)\, s = 0 \\ \implies H_x(f) \cdot s &= -\nabla_x\, f(x). \end{aligned}$$

In other words, to choose the next step s, Newton's method suggests that you must *solve* a system of linear equations, where the matrix is the H and the right-hand side is the negative gradient of f.

Summary: Newton's method. Summarizing the main ideas from above, Newton's method to maximize the scalar objective function $f(x)$ wher consists of the following steps:

- Start with some initial guess $x(0)$

- Start with some initial guess $x(0)$.
- At step t, compute the *search direction* $s(t)$ by solving $H_{x(t)}(f) \cdot s(t) = -\nabla_x f(x(t))$.
- Compute a new (and hopefully improved) guess by the update, $x(t+1) \leftarrow x(t) + s(t)$.

Implementing logistic regression via a Newton-based MLE

To perform MLE for the logistic regression model using Newton's method, you need both the gradient of the log-likelihood as well as the Hessian know how to compute the gradient from the preceding exercises; so what about the Hessian?

Notationally, that calculation will be a little bit easier to write down and program with the following definition.

Definition: **Elementwise product**. Let $A \equiv (a_{ij})$ and $B \equiv (b_{ij})$ be $m \times n$ matrices. Denote the *elementwise product* of A and B by $A \odot B$. $C = A \odot B$, then element $c_{ij} = a_{ij} \cdot b_{ij}$.

If A is $m \times n$ but B is instead just $m \times 1$, then we will "auto-extend" B. Put differently, if B has the same number of rows as A but only 1 col will take $C = A \odot B$ to have elements $c_{ij} = a_{ij} \cdot b_i$.

In Python, you can use `np.multiply()` (http://docs.scipy.org/doc/numpy/reference/generated/numpy.multiply.html) for elementwise multiplicatio arrays.

In [26]:
```
A = np.array([[1, 2, 3],
              [4, 5, 6]])
B = np.array([[-1, 2, -3],
              [4, -5, 6]])

print(np.multiply(A, B)) # elementwise product
print()
print(np.multiply(A, B[:, 0:1])) # "auto-extend" version
```

```
[[ -1   4  -9]
 [ 16 -25  36]]

[[-1 -2 -3]
 [16 20 24]]
```

Exercise 10 (optional; ungraded). Show that the Hessian of the log-likelihood for logistic regression is

$$H_\theta\left(\mathcal{L}(\theta; l, X)\right) \quad = \quad -\left(X \odot G(X\theta)\right)^T \left(X \odot G(-X\theta)\right).$$

Exercise 11 (0 points). Implement a function to compute the Hessian of the log-likelihood. The signature of your function should be,

```
def hess_log_likelihood (theta, y, X):
      ...
```

In [27]:
```
### BEGIN SOLUTION
def hess_log_likelihood(theta, y, X):
    """Returns the Hessian of the log-likelihood."""
    z = X.dot(theta)
    A = np.multiply(X, logistic(z))
    B = np.multiply(X, logistic(-z))
    return -A.T.dot(B)
### END SOLUTION
```

In [28]:
```
# Test cell: `hess_log_likelihood__check`

if False:
    d_hess_soln = 20
    m_hess_soln = 501
    theta_hess_soln = np.random.random ((d_hess_soln+1, 1)) * 2.0 - 1.0
    y_hess_soln = np.random.randint (low=0, high=2, size=(m_hess_soln, 1))
    X_hess_soln = np.random.random ((m_hess_soln, d_hess_soln+1)) * 2.0 - 1.0
    X_hess_soln[:, 0] = 1.0
    L_hess_soln = hess_log_likelihood (theta_hess_soln, y_hess_soln, X_hess_soln)
    np.savez_compressed ('hess_log_likelihood_soln',
                         d_hess_soln, m_hess_soln, theta_hess_soln, y_hess_soln, X_hess_soln, L_hess_so

npzfile_hess_soln = np.load ('{}hess_log_likelihood_soln.npz'.format(DATA_PATH))
d_hess_soln = npzfile_hess_soln['arr_0']
m_hess_soln = npzfile_hess_soln['arr_1']
theta_hess_soln = npzfile_hess_soln['arr_2']
y_hess_soln = npzfile_hess_soln['arr_3']
X_hess_soln = npzfile_hess_soln['arr_4']
L_hess_soln = npzfile_hess_soln['arr_5']

L_hess_you = hess_log_likelihood(theta_hess_soln, y_hess_soln, X_hess_soln)
your_hess_err = np.max(np.abs(L_hess_you/L_hess_soln - 1.0))
display(Math(r'\left\|\dfrac{H_{\tiny \mbox{yours}} - H_{\tiny \mbox{solution}}}{H_{\tiny \mbox{solutio
_\infty \approx %g' % your_hess_err))
assert your_hess_err <= 1e-12
```

```
print ("\n(Passed.)")
```

$$\left\| \frac{H_{\text{yours}} - H_{\text{solution}}}{H_{\text{solution}}} \right\|_{\infty} \approx 9.74776e-14$$

```
(Passed.)
```

Exercise 12 (0 points). Finish the implementation of a Newton-based MLE procedure for the logistic regression problem.

In [29]:
```
MAX_STEP = 10

# Get the data coordinate matrix, X, and labels vector, l
X = points
y = labels.astype(dtype=float)

# Store *all* guesses, for subsequent analysis
thetas_newt = np.zeros((3, MAX_STEP+1))

for t in range(MAX_STEP):
    ### BEGIN SOLUTION
    theta_t = thetas_newt[:, t:t+1]
    g_t = grad_log_likelihood(theta_t, y, X)
    H_t = hess_log_likelihood(theta_t, y, X)
    s_t = np.linalg.solve(H_t, -g_t)
    thetas_newt[:, t+1:t+2] = theta_t + s_t
    ### END SOLUTION

theta_newt = thetas_newt[:, MAX_STEP:]
print ("Your (hand) solution:", my_theta.T.flatten())
print ("Computed solution:", theta_newt.T.flatten())

print ("\n=== Comparisons ===")
display (Math (r'\dfrac{\theta_0}{\theta_2}:'))
print ("Your manual (hand-picked) solution is", my_theta[0]/my_theta[2], \
      ", vs. MLE (via Newton's method), which is", theta_newt[0]/theta_newt[2])
display (Math (r'\dfrac{\theta_1}{\theta_2}:'))
print ("Your manual (hand-picked) solution is", my_theta[1]/my_theta[2], \
      ", vs. MLE (via Newton's method), which is", theta_newt[1]/theta_newt[2])

print ("\n=== The MLE solution, visualized ===")
newt_labels = gen_lin_discr_labels(points, theta_newt)
df_newt = df.copy()
df_newt['label'] = mark_matches(newt_labels, labels).astype (dtype=int)
plot_lin_discr(theta_newt, df_newt)
```

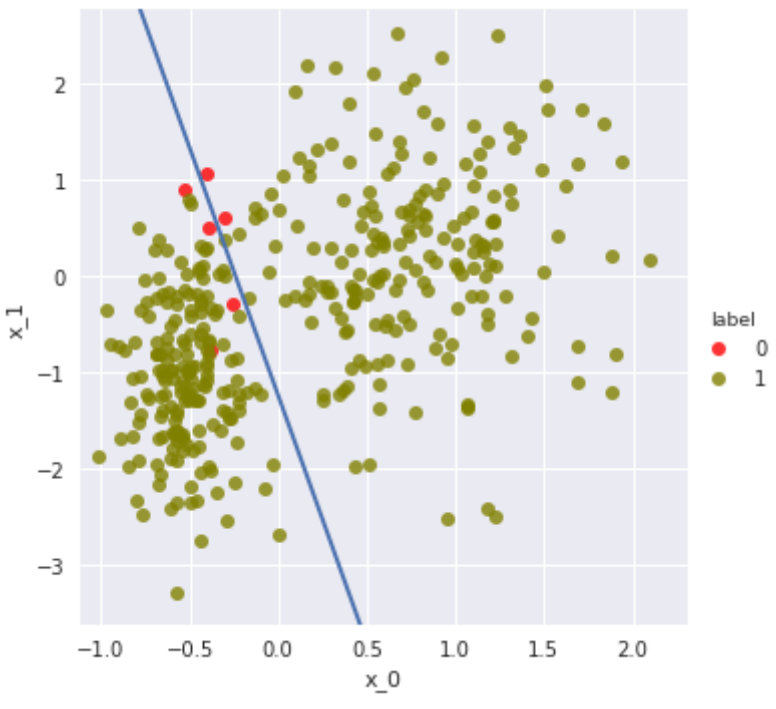
```
Your (hand) solution: [-2.   -0.5  -0.55]
Computed solution: [-15.63082207  -3.04255951  -3.76500606]

=== Comparisons ===
```

$$\frac{\theta_0}{\theta_2}:$$

```
Your manual (hand-picked) solution is [3.63636364] , vs. MLE (via Newton's method), which is [4.1516060
```

$$\frac{\theta_1}{\theta_2}:$$

```
Your manual (hand-picked) solution is [0.90909091] , vs. MLE (via Newton's method), which is [0.8081154

=== The MLE solution, visualized ===
```



In [30]:
```
# Test cell: `logreg_mle_newt__check`

print ("\n=== Mismatch counts ===")

my_labels = gen_lin_discr_labels(points, my_theta)
```

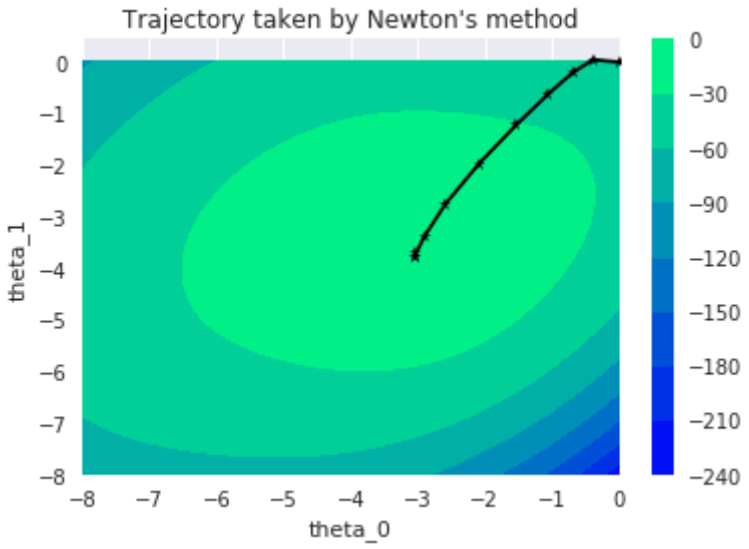
```
my_labels = gen_lin_discr_labels (points, my_theta)
my_mismatches = len (labels) - count_matches (labels, my_labels)
print ("Your manual (hand-picked) solution has", num_mismatches, "mismatches.")

newt_labels = gen_lin_discr_labels (points, theta_newt)
newt_mismatches = len (labels) - count_matches (labels, newt_labels)
print ("The MLE+Newton method produces", newt_mismatches, "mismatches.")

assert newt_mismatches <= ga_mismatches
print ("\n(Passed.)")
```

```
=== Mismatch counts ===
Your manual (hand-picked) solution has 5 mismatches.
The MLE+Newton method produces 6 mismatches.

(Passed.)
```

The following cell creates a contour plot of the log-likelihood, as done previously in this notebook. Add code to display the trajectory taken by Ne method.

In [31]:
```
p = plt.contourf(x1, x2, ll_grid, cmap=plt.cm.get_cmap("winter"))
plt.xlabel('theta_0')
plt.ylabel('theta_1')
plt.title('Trajectory taken by Newton\'s method')
plt.colorbar()
plt.plot(thetas_newt[1, :], thetas_newt[2, :], 'k*-')
```

Out[31]:
```
[<matplotlib.lines.Line2D at 0x7f0ec2609b38>]
```



How many steps does this optimization procedure take compared to gradient ascent? What is the tradeoff?

‹ Previous

Next Up: Topic 14: Clustering via k-means
13 min ›

Legal

Terms of Service & Honor Code
Privacy Policy
Accessibility Policy
Trademark Policy
Sitemap

Connect

Blog
Contact Us
Help Center
Media Kit
Donate