TUTORIAL 3a

# Logistic regression

*Course:* Math 535 (http://www.math.wisc.edu/~roch/mmids/) - Mathematical Methods in Data Science (MMiDS)
*Author:* Sebastien Roch (http://www.math.wisc.edu/~roch/), Department of Mathematics, University of Wisconsin-Madison
*Updated:* Oct 18, 2020


## Recap and further background

In this notebook, we illustrate the use of gradient descent on binary classification by logistic regression.

**Classification.** Quoting Wikipedia (https://en.wikipedia.org/wiki/Statistical_classification), recall that classification is the following machine learning task:

> In machine learning and statistics, classification is the problem of identifying to which of a set of categories (sub-populations) a new observation belongs, on the basis of a training set of data containing observations (or instances) whose category membership is known. Examples are assigning a given email to the "spam" or "non-spam" class, and assigning a diagnosis to a given patient based on observed characteristics of the patient (sex, blood pressure, presence or absence of certain symptoms, etc.). Classification is an example of pattern recognition. In the terminology of machine learning, classification is considered an instance of supervised learning, i.e., learning where a training set of correctly identified observations is available.

The input data is of the form $\{(\boldsymbol{\alpha}_i, b_i) : i = 1, \ldots, n\}$ where $\boldsymbol{\alpha}_i \in \mathbb{R}^d$ are the features and $b_i \in \{0, 1\}$ is the label. As before we use a matrix representation: $A \in \mathbb{R}^{n \times d}$ has rows $\boldsymbol{\alpha}_j^T$, $j = 1, \ldots, n$ and $\mathbf{b} = (b_1, \ldots, b_n)^T \in \{0, 1\}^n$.

**Logistic model.** We summarize the logistic regression approach. Our goal is to find a function of the features that approximates the probability of the label $1$. For this purpose, we model the log-odds (https://en.wikipedia.org/wiki/Logit) (or logit function) of the probability of label $1$ as a linear function of the features

$$\log \frac{p(\boldsymbol{\alpha}; \mathbf{x})}{1 - p(\boldsymbol{\alpha}; \mathbf{x})} = \boldsymbol{\alpha}^T \mathbf{x}$$

where $\mathbf{x} \in \mathbb{R}^d$. Inverting this expression gives

$$p(\boldsymbol{\alpha}; \mathbf{x}) = \sigma(\boldsymbol{\alpha}^T \mathbf{x})$$

where the sigmoid (https://en.wikipedia.org/wiki/Logistic_function) function is

$$\sigma(t) = \frac{1}{1 + e^{-t}}$$

for $t \in \mathbb{R}$.

We seek to maximize the likelihood of the data assuming the labels are independent given the features, which is given by

$$\mathcal{L}(\mathbf{x}; A, \mathbf{b}) = \prod_{i=1}^{n} p(\boldsymbol{\alpha}_i; \mathbf{x})^{b_i} (1 - p(\boldsymbol{\alpha}_i; \mathbf{x}))^{1-b_i}$$

Taking a logarithm, multiplying by $-1/n$ and substituting the sigmoid function, we want to minimize the cross-entropy loss (https://en.wikipedia.org/wiki/Cross_entropy#Cross-entropy_loss_function_and_logistic_regression)

$$\ell(\mathbf{x}; A, \mathbf{b}) = -\frac{1}{n} \sum_{i=1}^{n} b_i \log(\sigma(\boldsymbol{\alpha}^T \mathbf{x})) - \frac{1}{n} \sum_{i=1}^{n} (1 - b_i) \log(1 - \sigma(\boldsymbol{\alpha}^T \mathbf{x})).$$

That is, we solve

$$\min_{\mathbf{x} \in \mathbb{R}^d} \ell(\mathbf{x}; A, \mathbf{b}).$$

**Gradient descent.** To use gradient descent, we need the gradient of $\ell$. We use the *Chain Rule* and first compute the derivative of $\sigma$ which is

$$\sigma'(t) = \frac{e^{-t}}{(1 + e^{-t})^2} = \frac{1}{1 + e^{-t}} \left( 1 - \frac{1}{1 + e^{-t}} \right) = \sigma(t)(1 - \sigma(t)).$$

The latter expression is known as the logistic differential equation (https://en.wikipedia.org/wiki/Logistic_function#Logistic_differential_equation). It arises in a variety of applications, including the modeling of population dynamics (https://towardsdatascience.com/covid-19-infection-in-italy-mathematical-models-and-predictions-7784b4d7dd8d). Here it will be a convenient way to compute the gradient. Indeed observe that by the *Chain Rule*

$$\nabla_{\mathbf{x}} \, \sigma(\boldsymbol{\alpha}^T \mathbf{x}) = \sigma(\boldsymbol{\alpha}^T \mathbf{x})(1 - \sigma(\boldsymbol{\alpha}^T \mathbf{x})) \, \boldsymbol{\alpha}$$

where we use a subscript $\mathbf{x}$ to make it clear that the gradient is with respect to $\mathbf{x}$.

By another application of the *Chain Rule*

$$\begin{aligned}\nabla_{\mathbf{x}}\,\ell(\mathbf{x};A,\mathbf{b}) &= -\frac{1}{n}\sum_{i=1}^{n}\frac{b_i}{\sigma(\boldsymbol{\alpha}_i^T\mathbf{x})}\nabla_{\mathbf{x}}\,\sigma(\boldsymbol{\alpha}_i^T\mathbf{x}) + \frac{1}{n}\sum_{i=1}^{n}\frac{1-b_i}{1-\sigma(\boldsymbol{\alpha}_i^T\mathbf{x})}\nabla_{\mathbf{x}}\,\sigma(\boldsymbol{\alpha}_i^T\mathbf{x})\\ &= -\frac{1}{n}\sum_{i=1}^{n}\left(\frac{b_i}{\sigma(\boldsymbol{\alpha}_i^T\mathbf{x})} - \frac{1-b_i}{1-\sigma(\boldsymbol{\alpha}_i^T\mathbf{x})}\right)\sigma(\boldsymbol{\alpha}_i^T\mathbf{x})(1-\sigma(\boldsymbol{\alpha}_i^T\mathbf{x}))\,\boldsymbol{\alpha}_i\\ &= -\frac{1}{n}\sum_{i=1}^{n}(b_i - \sigma(\boldsymbol{\alpha}_i^T\mathbf{x}))\,\boldsymbol{\alpha}_i.\end{aligned}$$

To compute the Hessian, we note that

$$\nabla_{\mathbf{x}}(\sigma(\boldsymbol{\alpha}^T\mathbf{x})\,\alpha_j) = \sigma(\boldsymbol{\alpha}^T\mathbf{x})(1-\sigma(\boldsymbol{\alpha}^T\mathbf{x}))\,\boldsymbol{\alpha}\,\alpha_j$$

so that

$$\nabla_{\mathbf{x}}(\sigma(\boldsymbol{\alpha}^T\mathbf{x})\,\boldsymbol{\alpha}) = \sigma(\boldsymbol{\alpha}^T\mathbf{x})(1-\sigma(\boldsymbol{\alpha}^T\mathbf{x}))\,\boldsymbol{\alpha}\boldsymbol{\alpha}^T.$$

Thus

$$\nabla_{\mathbf{x}}^2\,\ell(\mathbf{x};A,\mathbf{b}) = \frac{1}{n}\sum_{i=1}^{n}\sigma(\boldsymbol{\alpha}_i^T\mathbf{x})(1-\sigma(\boldsymbol{\alpha}_i^T\mathbf{x}))\,\boldsymbol{\alpha}_i\boldsymbol{\alpha}_i^T$$

where $\nabla_{\mathbf{x}}^2$ indicates the Hessian with respect to the $\mathbf{x}$ variables.

---

**Lemma (Convexity of logistic regression):** The function $\ell(\mathbf{x};A,\mathbf{b})$ is convex as a function of $\mathbf{x}\in\mathbb{R}^d$.

---

*Proof:* Indeed, the Hessian is positive semidefinite: for any $\mathbf{z}\in\mathbb{R}^d$

$$\begin{aligned}\mathbf{z}^T\,\nabla_{\mathbf{x}}^2\,\ell(\mathbf{x};A,\mathbf{b})\,\mathbf{z} &= \frac{1}{n}\sum_{i=1}^{n}\sigma(\boldsymbol{\alpha}_i^T\mathbf{x})(1-\sigma(\boldsymbol{\alpha}_i^T\mathbf{x}))\,\mathbf{z}^T\boldsymbol{\alpha}_i\boldsymbol{\alpha}_i^T\mathbf{z}\\ &= \frac{1}{n}\sum_{i=1}^{n}\sigma(\boldsymbol{\alpha}_i^T\mathbf{x})(1-\sigma(\boldsymbol{\alpha}_i^T\mathbf{x}))\,(\mathbf{z}^T\boldsymbol{\alpha}_i)^2\\ &\geq 0\end{aligned}$$

since $\sigma(t)\in[0,1]$ for all $t$. □

Convexity is one reason for working with the cross-entropy loss (rather than the mean squared error (https://math.stackexchange.com/questions/1582452/logistic-regression-prove-that-the-cost-function-is-convex) for instance).

**Update formula.** For step size $\beta$, one step of gradient descent is therefore

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \beta \frac{1}{n} \sum_{i=1}^{n} (b_i - \sigma(\boldsymbol{\alpha}_i^T \mathbf{x}^k)) \, \boldsymbol{\alpha}_i.$$

In stochastic gradient descent (https://en.wikipedia.org/wiki/Stochastic_gradient_descent) (SGD), a variant of gradient descent, we pick a sample $I$ uniformly at random in $\{1, \ldots, n\}$ and update as follows

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \beta \, (b_I - \sigma(\boldsymbol{\alpha}_I^T \mathbf{x}^k)) \, \boldsymbol{\alpha}_I.$$

For the mini-batch version of SGD, we pick a random sub-sample $\mathcal{B} \subseteq \{1, \ldots, n\}$ of size $B$

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \beta \frac{1}{B} \sum_{i \in \mathcal{B}} (b_i - \sigma(\boldsymbol{\alpha}_i^T \mathbf{x}^k)) \, \boldsymbol{\alpha}_i.$$

The key observation about the two stochastic updates above is that, in expectation, they perform a step of gradient descent. That turns out to be enough and it has computational advantages. Many variants exist and we will encounter some of them in the next notebook.

# 1 Lebron James 2017 NBA Playoffs dataset

We start with a simple dataset from UC Berkeley's DS100 (http://www.ds100.org) course. The file `lebron.csv` is available here (https://github.com/DS-100/textbook/tree/master/content/ch/17). Quoting the course's textbook [DS100 (https://www.textbook.ds100.org/intro.html), Section 17.1]:

> In basketball, players score by shooting a ball through a hoop. One such player, LeBron James, is widely considered one of the best basketball players ever for his incredible ability to score. LeBron plays in the National Basketball Association (NBA), the United States's premier basketball league. We've collected a dataset of all of LeBron's attempts in the 2017 NBA Playoff Games using the NBA statistics website (https://stats.nba.com/ (https://stats.nba.com/)).

We first load the data and look at its summary.

In [1]:
```
# Julia version: 1.5.1
using CSV, DataFrames, Plots, LinearAlgebra, Statistics
```

In [2]:
```
df = CSV.read("lebron.csv")
first(df,5)
```

Out[2]: 5 rows × 7 columns (omitted printing of 1 columns)

| | game_date | minute | opponent | action_type | shot_type | shot_distance |
|---|---|---|---|---|---|---|
| | Int64 | Int64 | String | String | String | Int64 |
| 1 | 20170415 | 10 | IND | Driving Layup Shot | 2PT Field Goal | 0 |
| 2 | 20170415 | 11 | IND | Driving Layup Shot | 2PT Field Goal | 0 |
| 3 | 20170415 | 14 | IND | Layup Shot | 2PT Field Goal | 0 |
| 4 | 20170415 | 15 | IND | Driving Layup Shot | 2PT Field Goal | 0 |
| 5 | 20170415 | 18 | IND | Alley Oop Dunk Shot | 2PT Field Goal | 0 |

In [3]:
```
nrow(df)
```

Out[3]: 384

In [4]:
```
describe(df)
```

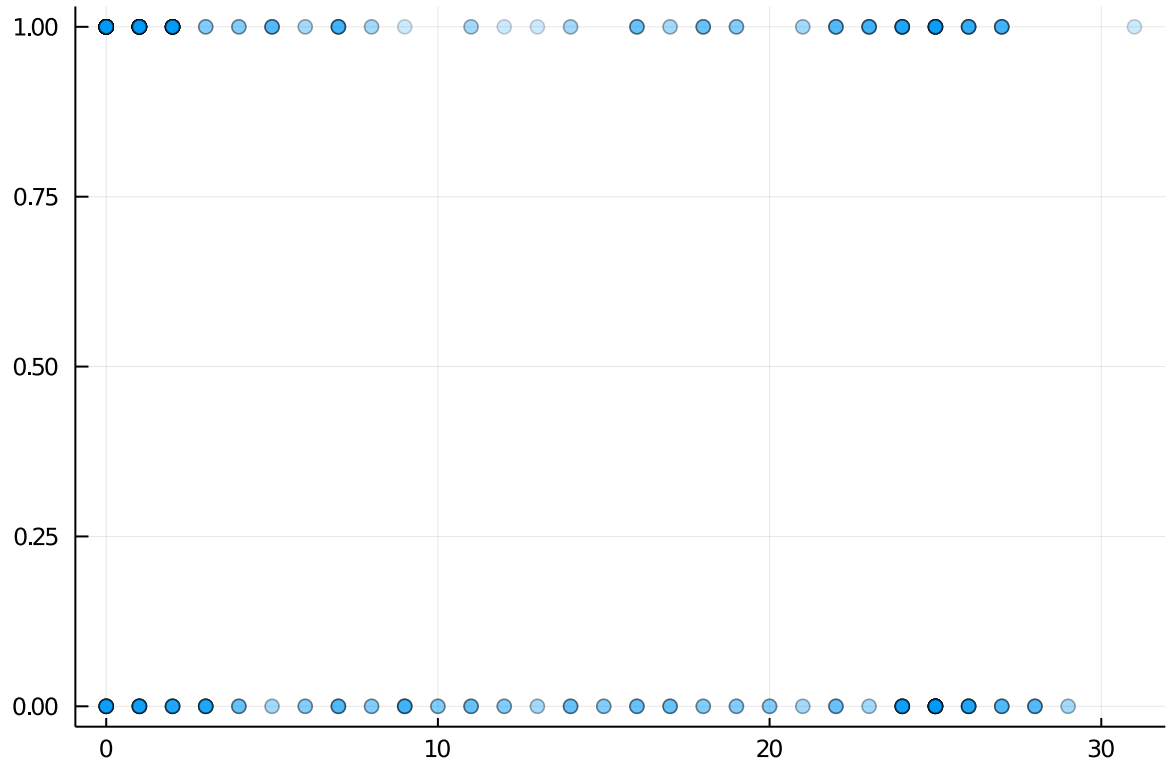
Out[4]: 7 rows × 8 columns (omitted printing of 3 columns)

| | variable | mean | min | median | max |
|---|---|---|---|---|---|
| | Symbol | Union… | Any | Union… | Any |
| 1 | game_date | 2.01705e7 | 20170415 | 2.01705e7 | 20170612 |
| 2 | minute | 24.4062 | 1 | 25.0 | 48 |
| 3 | opponent | | BOS | | TOR |
| 4 | action_type | | Alley Oop Dunk Shot | | Turnaround Jump Shot |
| 5 | shot_type | | 2PT Field Goal | | 3PT Field Goal |
| 6 | shot_distance | 10.6953 | 0 | 6.5 | 31 |
| 7 | shot_made | 0.565104 | 0 | 1.0 | 1 |

The two columns we will be interested in are `shot_distance` (LeBron's distance from the basket when the shot was attempted (ft)) and `shot_made` (0 if the shot missed, 1 if the shot went in). As the summary table above indicates, the average distance was `10.6953` and the frequency of shots made was `0.565104`. We extract those two columns and display them on a scatter plot.

In [5]:
```
feature = df[:,:shot_distance]
label = df[:,:shot_made]
scatter(feature, label; legend=false, alpha=0.2)
```

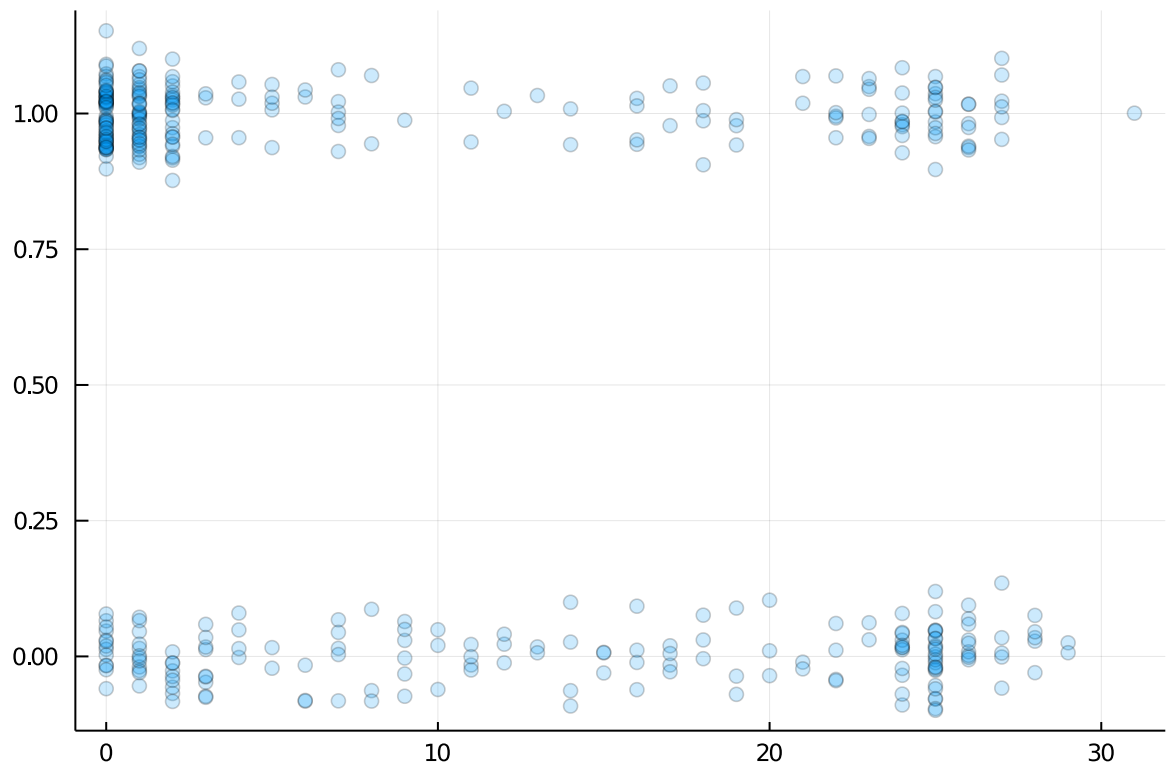
Out[5]:




As you can see, this kind of data is hard to vizualize because of the superposition of points with the same $x$ and $y$-values. One trick is to jiggle the $y$'s a little bit by adding Gaussian noise. We do this next and plot again.

In [6]:
```
label_jitter = label .+ 0.05*randn(length(label))
scatter(feature, label_jitter; legend=false, alpha=0.2)
```

Out[6]:




To run gradient descent (GD), we first implement a function computing an update. It takes as input a function `grad_fn` computing the gradient itself.

In [7]:
```
function desc_update(grad_fn, A, b, curr_x, β)
    gradient = grad_fn(curr_x, A, b)
    return curr_x .- β*gradient
end
```

Out[7]:
```
desc_update (generic function with 1 method)
```

We are ready to implement GD. Our function takes as input a function `loss_fn` computing the objective, a function `grad_fn` computing the gradient and the dataset `A` and `b`. Optional parameters are an initial guess `init_x`, the step size, and the number of iterations. (To get $\beta$, try `\beta<tab>` where `<tab>` is the tab key.)

In [8]:
```
function mmids_gd(loss_fn, grad_fn, A, b, init_x; β=1e-3, niters=1e5)

    # current x and loss
    curr_x, curr_l = init_x, loss_fn(init_x, A, b)

    # until the maximum number of iterations
    for iter = 1:niters
        curr_x = desc_update(grad_fn, A, b, curr_x, β) # gradient step
    end

    return curr_x
end
```

Out[8]:
```
mmids_gd (generic function with 1 method)
```

We apply GD to logistic regression. We first construct the data matrices $A$ and $\mathbf{b}$. To allow an affine function of the features, we add a column of $1$'s as we have done before.

In [9]:
```
A = [ones(length(label)) feature]
b = label;
```

To implement `loss_fn` and `grad_fn`, we define the sigmoid first.

In [10]:
```
sigmoid = z -> 1/(1+exp(-z))
pred_fn = (x, A) -> sigmoid.(A*x)
loss_fn = (x, A, b) -> mean(-b.*log.(pred_fn(x, A))
    .-(1 .- b).*log.(1 .- pred_fn(x, A)))
```

Out[10]:
```
#6 (generic function with 1 method)
```

In [11]:
```
grad_fn = (x, A, b) -> -A'*(b.-pred_fn(x, A))/length(b)
```

Out[11]:
```
#8 (generic function with 1 method)
```

We run GD starting from $(0, 0)$ with a step size of $0.01$.

In [12]:
```
init_x = zeros(size(A,2))
@time best_x = mmids_gd(loss_fn, grad_fn, A, b, init_x;
    β=0.01, niters=10000)
```

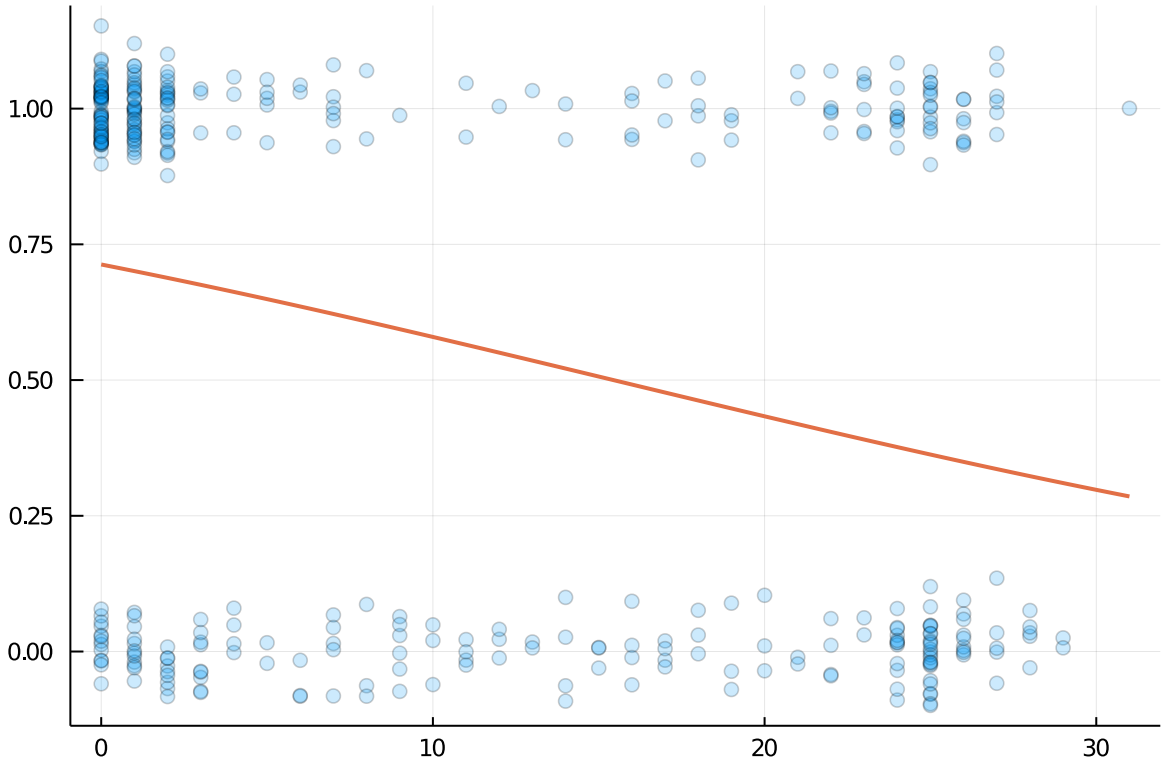
```
  0.730639 seconds (2.22 M allocations: 262.630 MiB, 9.85% gc time)
```

Out[12]:
```
2-element Array{Float64,1}:
  0.9095669233878183
 -0.058907173767627455
```

Finally we plot the results.

In [13]:
```
grid = LinRange(minimum(feature), maximum(feature), 100)
feature_grid = [ones(length(grid)) grid]
predict_grid = sigmoid.(feature_grid*best_x)
scatter(feature, label_jitter, legend=false, alpha=0.2)
plot!(grid,predict_grid;lw=2)
```

Out[13]:




## 2 South African Heart Disease dataset

We analyze a dataset from [ESL (https://web.stanford.edu/~hastie/ElemStatLearn/)], which can be downloaded here (https://web.stanford.edu/~hastie/ElemStatLearn/data.html). Quoting [ESL (https://web.stanford.edu/~hastie/ElemStatLearn/), Section 4.4.2]

> The data [...] are a subset of the Coronary Risk-Factor Study (CORIS) baseline survey, carried out in three rural areas of the Western Cape, South Africa (Rousseauw et al., 1983). The aim of the study was to establish the intensity of ischemic heart disease risk factors in that high-incidence region. The data represent white males between 15 and 64, and the response variable is the presence or absence of myocardial infarction (MI) at the time of the survey (the overall prevalence of MI was 5.1% in this region). There are 160 cases in our data set, and a sample of 302 controls. These data are described in more detail in Hastie and Tibshirani (1987).

We load the data, which we slightly reformatted and look at a summary.

In [14]:
```
df = CSV.read("SAHeart.csv")
first(df,5)
```

Out[14]: 5 rows × 9 columns

| | sbp | tobacco | ldl | adiposity | typea | obesity | alcohol | age | chd |
|---|---|---|---|---|---|---|---|---|---|
| | Float64 | Float64 | Float64 | Float64 | Float64 | Float64 | Float64 | Float64 | Float64 |
| 1 | 160.0 | 12.0 | 5.73 | 23.11 | 49.0 | 25.3 | 97.2 | 52.0 | 1.0 |
| 2 | 144.0 | 0.01 | 4.41 | 28.61 | 55.0 | 28.87 | 2.06 | 63.0 | 1.0 |
| 3 | 118.0 | 0.08 | 3.48 | 32.28 | 52.0 | 29.14 | 3.81 | 46.0 | 0.0 |
| 4 | 170.0 | 7.5 | 6.41 | 38.03 | 51.0 | 31.99 | 24.26 | 58.0 | 1.0 |
| 5 | 134.0 | 13.6 | 3.5 | 27.78 | 60.0 | 25.99 | 57.34 | 49.0 | 1.0 |

In [15]:
```
nrow(df)
```

Out[15]: 462

In [16]:
```
describe(df)
```

Out[16]: 9 rows × 8 columns

| | variable | mean | min | median | max | nunique | nmissing | eltype |
|---|---|---|---|---|---|---|---|---|
| | Symbol | Float64 | Float64 | Float64 | Float64 | Nothing | Nothing | DataType |
| 1 | sbp | 138.327 | 101.0 | 134.0 | 218.0 | | | Float64 |
| 2 | tobacco | 3.63565 | 0.0 | 2.0 | 31.2 | | | Float64 |
| 3 | ldl | 4.74032 | 0.98 | 4.34 | 15.33 | | | Float64 |
| 4 | adiposity | 25.4067 | 6.74 | 26.115 | 42.49 | | | Float64 |
| 5 | typea | 53.1039 | 13.0 | 53.0 | 78.0 | | | Float64 |
| 6 | obesity | 26.0441 | 14.7 | 25.805 | 46.58 | | | Float64 |
| 7 | alcohol | 17.0444 | 0.0 | 7.51 | 147.19 | | | Float64 |
| 8 | age | 42.816 | 15.0 | 45.0 | 64.0 | | | Float64 |
| 9 | chd | 0.34632 | 0.0 | 0.0 | 1.0 | | | Float64 |

Our goal to predict `chd`, which stands for coronary heart disease, based on the other variables (which are briefly described here (https://web.stanford.edu/~hastie/ElemStatLearn/datasets/SAheart.info.txt)). We use logistic regression again.

We first construct the data matrices. We only use three of the predictors, as the convergence is quite slow. Try it for yourself!

In [17]:
```
feature = Matrix(df[:,[2,3,8]]);
```

In [18]:
```
label = Vector(df[:,end]);
```

In [19]:
```
A = [ones(length(label)) feature]
b = label;
```

We re-use the functions `loss_fn` and `grad_fn`, which were written for general logistic regression problems.

In [20]:
```
init_x = zeros(size(A,2))
@time best_x = mmids_gd(loss_fn, grad_fn, A, b, init_x)
```

```
  1.626671 seconds (2.40 M allocations: 2.555 GiB, 27.64% gc time)
```

Out[20]:
```
4-element Array{Float64,1}:
 -2.874431352998339
  0.08270555787374635
  0.12693709028694988
  0.03062386463774865
```

To get a sense of how accurate the result is, we compare our predictions to the true labels. By prediction, let us say that we mean that we predict label $1$ whenever $\sigma(\mathbf{a}^T \mathbf{x}) > 1/2$. We try this on the training set. (A better approach would be to split the data into training and testing sets, but we will not do this here.)

In [21]:
```
function logis_acc(x, A, b)
    return sum((pred_fn(x, A) .> 0.5) .== b)/length(b)
end
```

Out[21]:
```
logis_acc (generic function with 1 method)
```

In [22]:
```
logis_acc(best_x, A, b)
```

Out[22]:
```
0.7164502164502164
```

We also try mini-batch stochastic gradient descent (SGD). The only modification needed to the code is to pick a random mini-batch.

In [23]:
```
function mmids_sgd(loss_fn, grad_fn, A, b, init_x;
        β=1e-3, niters=1e5, batch=40)

    # current x and loss
    curr_x, curr_l = init_x, loss_fn(init_x, A, b)

    # until the maximum number of iterations
    nsamples = length(b) # number of samples
    for iter = 1:niters
        I = rand(1:nsamples, batch)
        curr_x = desc_update(grad_fn, A[I,:], b[I], curr_x, β) # gradien
t step
    end

    return curr_x
end
```

Out[23]:
```
mmids_sgd (generic function with 1 method)
```

In [24]:
```
init_x = zeros(size(A,2))
@time best_x = mmids_sgd(loss_fn, grad_fn, A, b, init_x)
```

```
  0.496897 seconds (2.31 M allocations: 525.985 MiB, 20.96% gc time)
```

Out[24]:
```
4-element Array{Float64,1}:
 -2.878980925842902
  0.07410072901414849
  0.12082225047396995
  0.03573090324445373
```

In [25]:
```
logis_acc(best_x, A, b)
```

Out[25]:
```
0.7207792207792207
```