[1]:
```
# make figures better:
%matplotlib inline
import matplotlib
font = {'weight':'normal','size':22}
matplotlib.rc('font', **font)
matplotlib.rc('figure', figsize=(9.0, 6.0))
matplotlib.rc('xtick.major', pad=10) # xticks too close to border!

from IPython.display import set_matplotlib_formats
set_matplotlib_formats('png', 'pdf')

import warnings
warnings.filterwarnings('ignore')
nicered = "#E6072A"
niceblu = "#424FA4"
nicegrn = "#6DC048"
```

DS1 Lecture ~~12~~ 13

James Bagrow, james.bagrow@uvm.edu, http://bagrow.com

Logistic regression

Regressing on categorical data

Linear regression works well for continuous data, with a goal of determining if there is a significant relationship between endogenous and exogenous variables, and to predict the endogenous value given exogenous variabels.

But what if we have a dependent/endogenous/response variable that is **categorical**? (See below for the easier case of a categorical exogenous (x) variable.)

[2]:
```
df = pd.read_csv("patientData__age_ageGrp_CHD.csv")
df = df.drop(["ID","AGEGRP"],axis=1)

print(df.head())
```

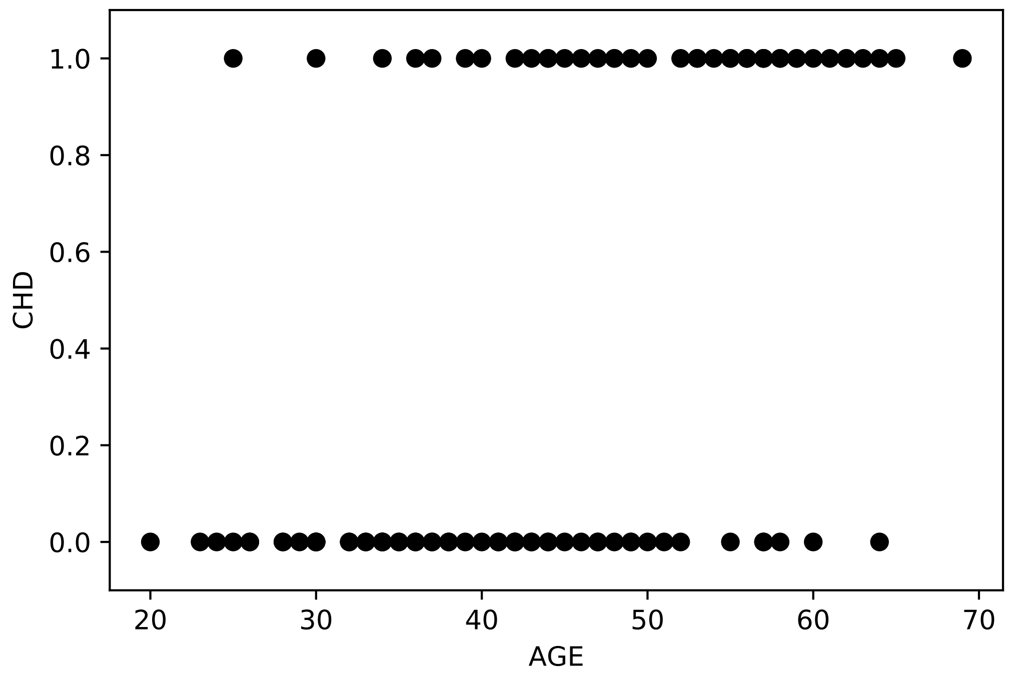
```
   AGE  CHD
0   20    0
1   23    0
2   24    0
3   25    0
4   25    1
```

These data take the age of a male medical subject versus whether or not he had significant **coronary heart disease**. (Data taken from *Applied Logistic Regression*, 3rd ed.)

- CHD = 1 if disease present, CHD = 0 if not.
- This is known as an **indicator** variable.

Let's plot these data:

[3]:
```
plt.plot(df["AGE"],df["CHD"],'o',c='k')
plt.ylim(-0.1,1.1)
plt.xlabel("AGE")
plt.ylabel("CHD");
```



There seems to be some **ordering** of these points, there are more 0's for lower `AGE`, more 1's for higher `AGE`. But having CHD depends on genetics, lifestyle, etc., so `AGE` does not completely determine `CHD`. We see this with the *overlap* between the two rows of points.

We need to model this statistically, so we turn to regression:

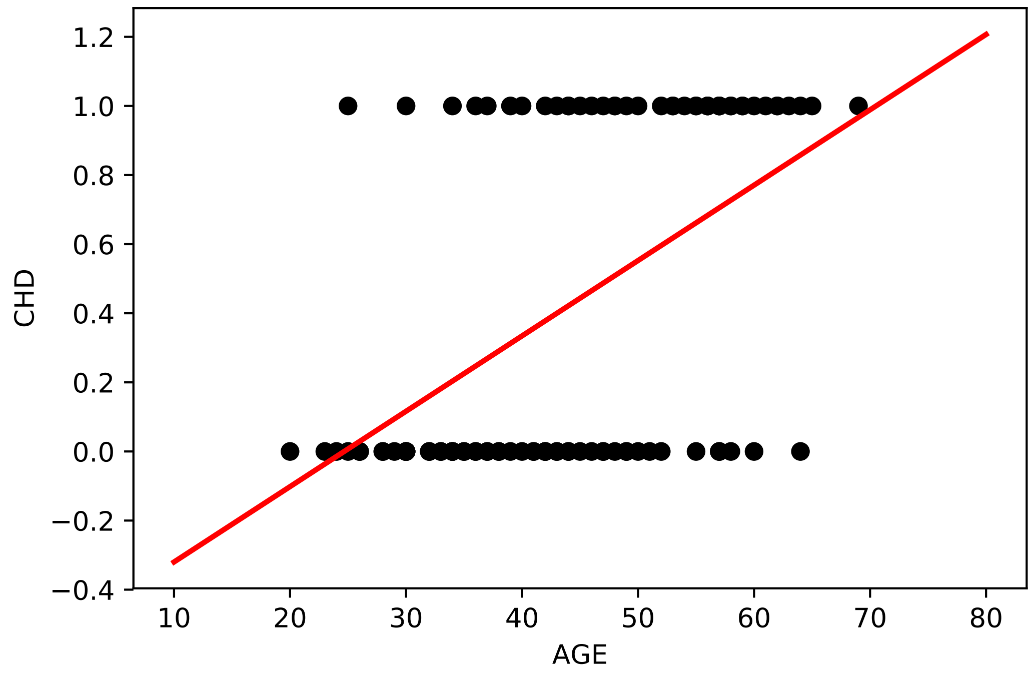
- Simple 1D regression for now:

[4]:
```
x = df["AGE"]
y = df["CHD"]
beta1, beta0, r, p, se = scipy.stats.linregress(x,y)

print(" Coeffs =", beta0,beta1)
print("    R^2 =", r**2)
print("p-value =", p)
```

```
 Coeffs = -0.5379603516067275 0.02181073347468967
    R^2 = 0.26398907354555046
p-value = 4.574870770876871e-08
```

Not bad, we see a very significant trend. Let's plot the line as well:

[5]:
```
plt.plot(df["AGE"],df["CHD"],'o',c='k')

xline = np.linspace(10,80,100)
yline = beta0 + beta1*xline

plt.plot(xline,yline, 'r-', lw=2)
plt.xlabel("AGE")
plt.ylabel("CHD");
```



Now we see problems. The line doesn't really look like the datapoints. And it predicts y-values outside of the plausible range of the CHD variable (y(20) = -0.1)?

- Let's dig into these data a little more.

Q: If we repeated the study multiple times and averaged `CHD` for all the subjects of a given age, say, `AGE=25`, what have we calculated?

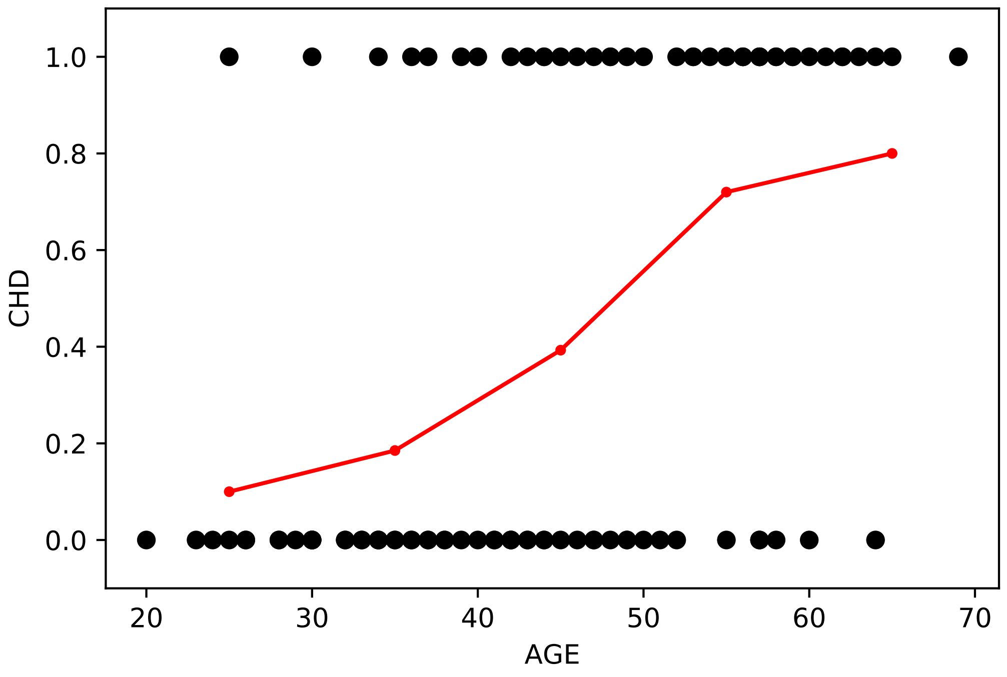
A: The fraction of 25 year olds with CHD (b/c having heart disease means `CHD=1`. This estimates the **probability** of having CHD.

We don't have the resources to repeat the study, so instead of averaging for every year, let's average for every decade of ages (20 year olds, 30 year olds, etc.)

- And then plot the curve of this probability.

[6]:
```
BS = scipy.stats.binned_statistic
bin_edges = [20,30,40,50,60,70]

y_bins,bin_edges, misc = BS(df["AGE"],df["CHD"],
                            statistic="mean",
                            bins=bin_edges)

# but this function doesn't return the bin CENTERS:
x_bins = (bin_edges[:-1]+bin_edges[1:])/2

plt.plot(x_bins,y_bins,'r.-')
```

```python
plt.plot(df["AGE"],df["CHD"],'o',c='k')
plt.ylim(-0.1,1.1)
plt.xlabel("AGE")
plt.ylabel("CHD")


plt.show()
```



We can think of the red curve as the estimated probability $P(CHD = 1 \mid AGE)$.

Modeling this probability

Unlike the linear model, our y-variable (this probability) is **bounded** to lie between 0 and 1 (inclusive). What we should really do is build a form of regression that accounts for this, which linear regression does not.

We need to think of a curve $y(x) = Pr(...)$ that looks a bit like the red curve above. It should go to zero as $x \to -\infty$ and go to 1 as $x \to \infty$...

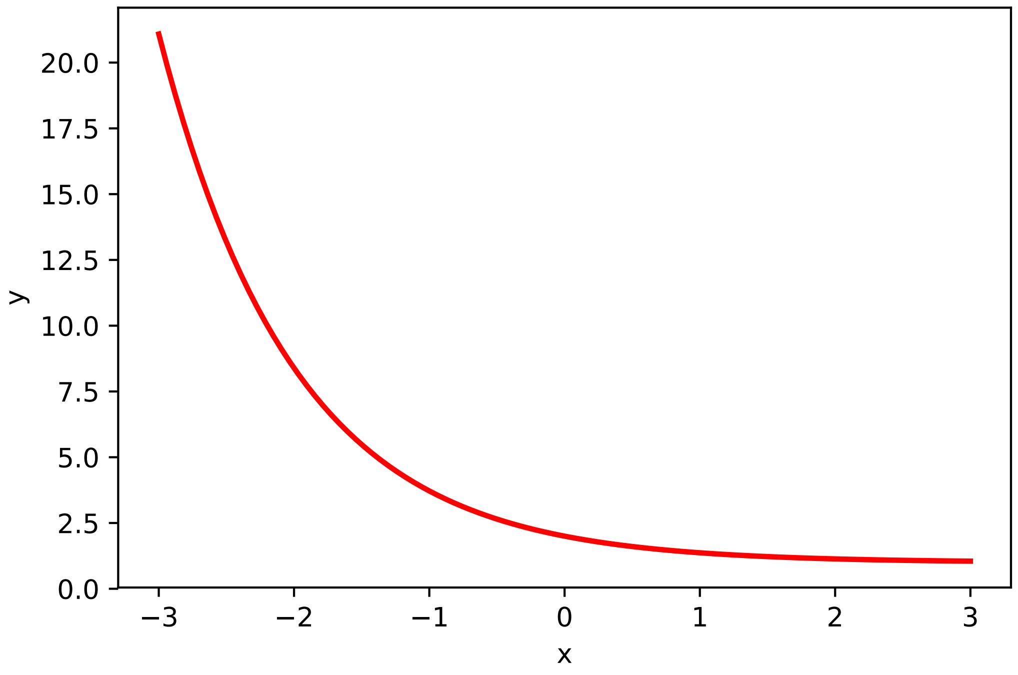
Hmm....

- Well, it can't be a line, or a trig function...
- An exponential curve $y(x) = \exp(-x)\ldots$ that goes to zero as $x \to \infty$, maybe we can flip it around... but it explodes as $x \to -\infty$, so that won't work...

Hmm...

- What about something like $y(x) = 1/(1-x)\ldots$ well that explodes at $x = 1\ldots$
- If we take $\exp(-x)$ and lift it by 1 we get...

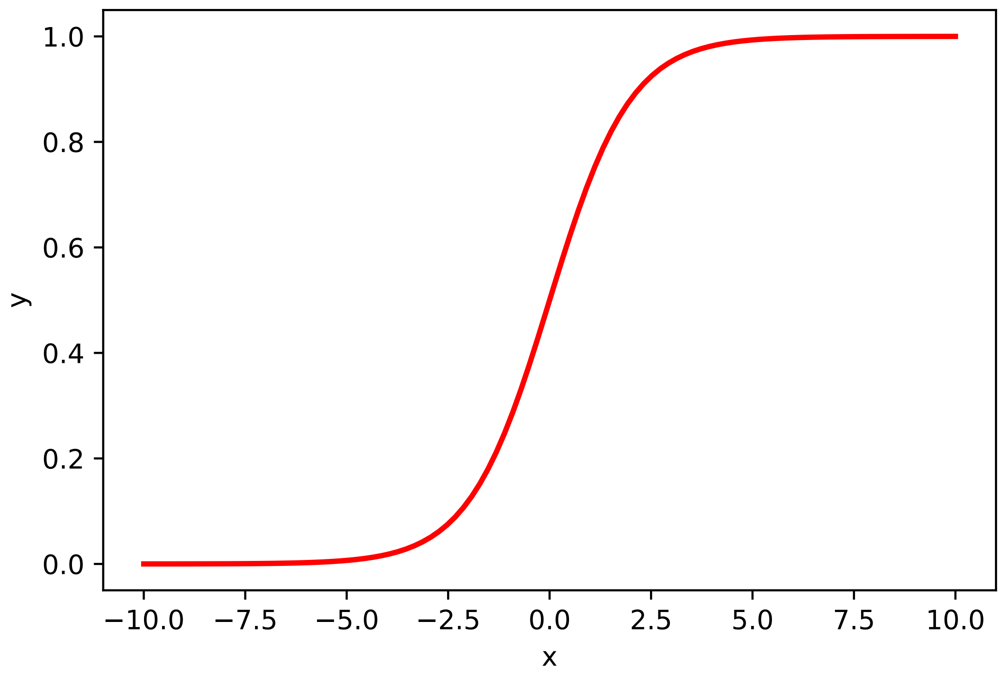
[7]:
```
x = np.linspace(-3,3,100)
y = 1.0 + np.exp(-x)

plt.plot(x,y,'r-',lw=2)
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```



OK, that goes to one on the right side, but it explodes on the left…

- What if we flip it over: $1/\left(1+\exp(-x)\right)$. That will make the left side go to zero…

[8]:
```
x = np.linspace(-10,10,100)
y = 1.0 / (1.0+np.exp(-x))

plt.plot(x,y,'r-',lw=2)
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```



AH HA!! We've got it!

A function of the form:

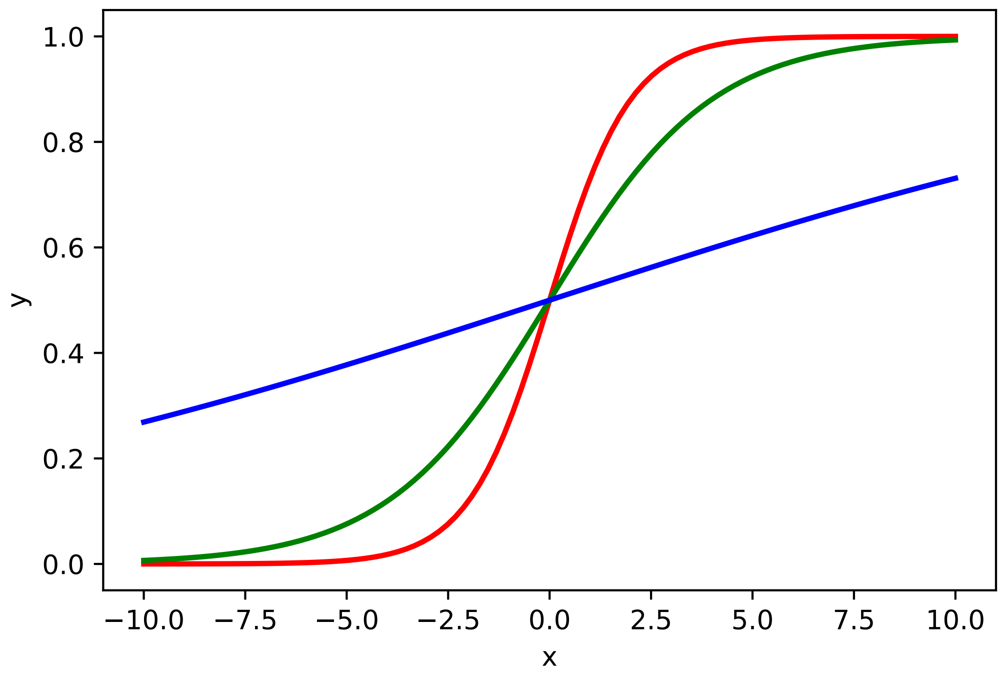
$$y(x) = \frac{1}{1+e^{-x}}$$

is called a sigmoid function.

We can also make changes to the x variable to control aspects of the curve.

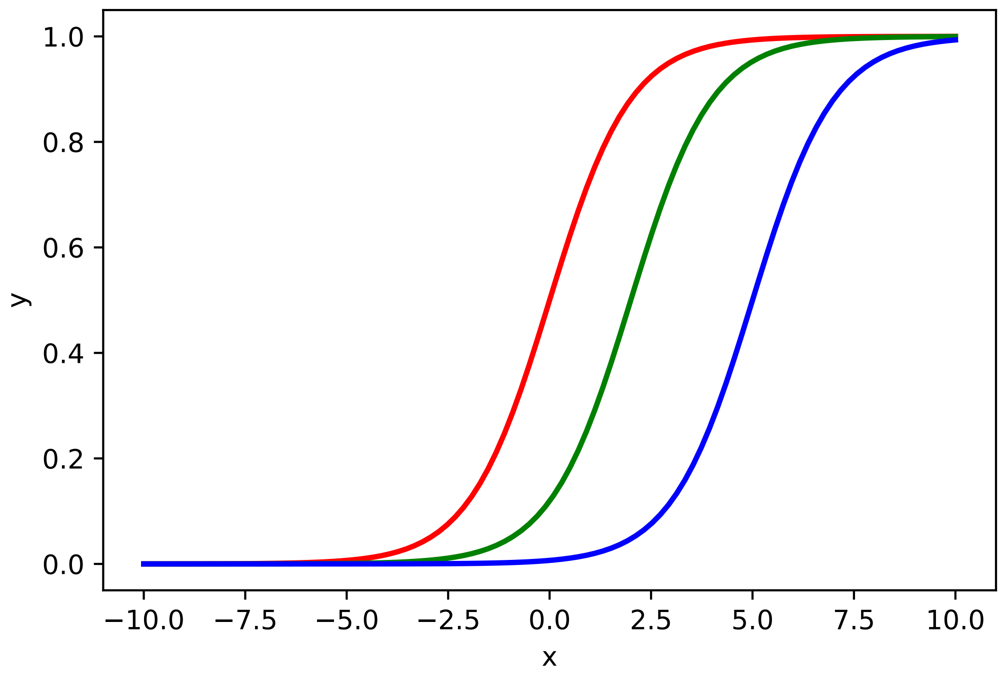
By dividing x by a number we can control how **steep** the crossover is:

[9]:
```
y1 = 1.0 / (1.0+np.exp(-x))
y2 = 1.0 / (1.0+np.exp(-x/2))
y3 = 1.0 / (1.0+np.exp(-x/10))

plt.plot(x,y1,'r-',lw=2)
plt.plot(x,y2,'g-',lw=2)
plt.plot(x,y3,'b-',lw=2)
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```



And by **shifting** *x* by a constant we can tune where the crossover occurs:

[10]:
```
y1 = 1.0 / ( 1.0 + np.exp(-(x-0)) )
y2 = 1.0 / ( 1.0 + np.exp(-(x-2)) )
y3 = 1.0 / ( 1.0 + np.exp(-(x-5)) )

plt.plot(x,y1,'r-',lw=2)
plt.plot(x,y2,'g-',lw=2)
plt.plot(x,y3,'b-',lw=2)
plt.xlabel("x")
plt.ylabel("y")
plt.show()
```



And this curve happens to look (quite?) a bit like the average $P(\text{CHD})$ from our data.

Odds

I'm sure you've seen people talk about odds before, especially with sports

- "Chicago has 4-to-1 odds against Denver on Sunday"

This means that the Chicago team is four times more likely to beat Denver. If we say p is the probability that Chicago wins, then $1 - p$ is the probability that Denver wins (ignoring a tie or whatever...) and the **odds** are:

$$\frac{p}{1-p}$$

This quantity is known as the **odds**.

A funny thing happens when we assume this probability is a function of x, $p(x)$, and that function is a sigmoid:

$$p(x) = \frac{1}{1 + e^{-f(x)}}$$

where for now the term being exponentiated is some unspecified function $f(x)$.

Plugging this into the odds:

$$\frac{p}{1-p} = e^{f(x)}$$

- Wow, that's very simple.

This is one of the motivations for choosing a sigmoid function over other possibilities such as arctan, it has a very **simple** odds.

We can go one step more by taking the **log** of the odds:

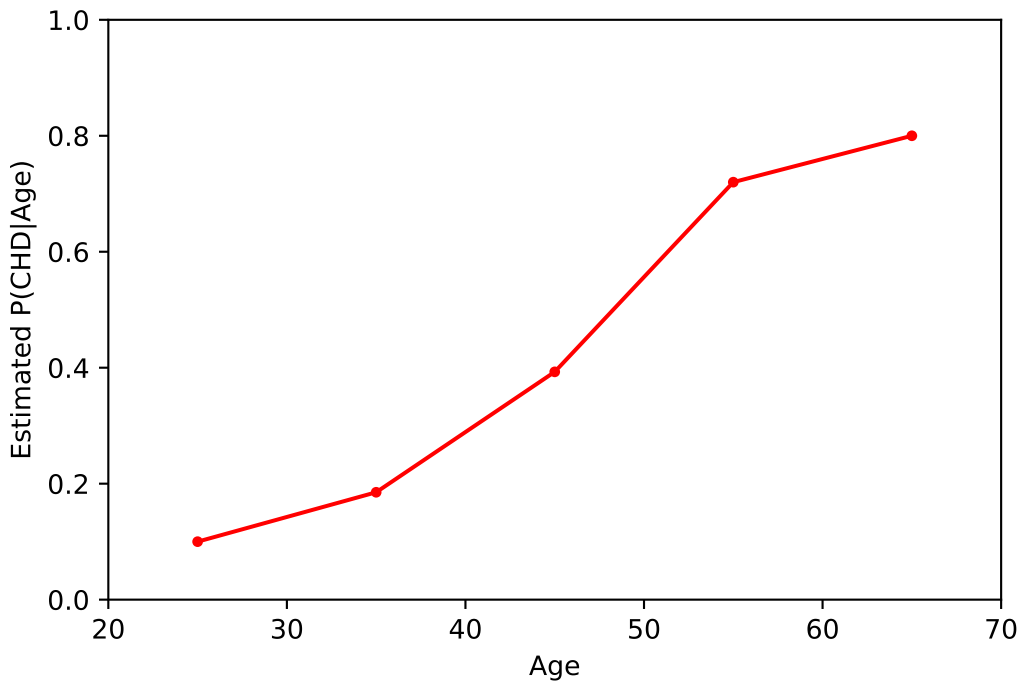
$$\ln\left(\frac{p}{1-p}\right) = f(x)$$

So we recover our function $f(x)$.

- This quantity, the logarithm of the odds is known as the **logit**, or log odds. It is the inverse sigmoid function.

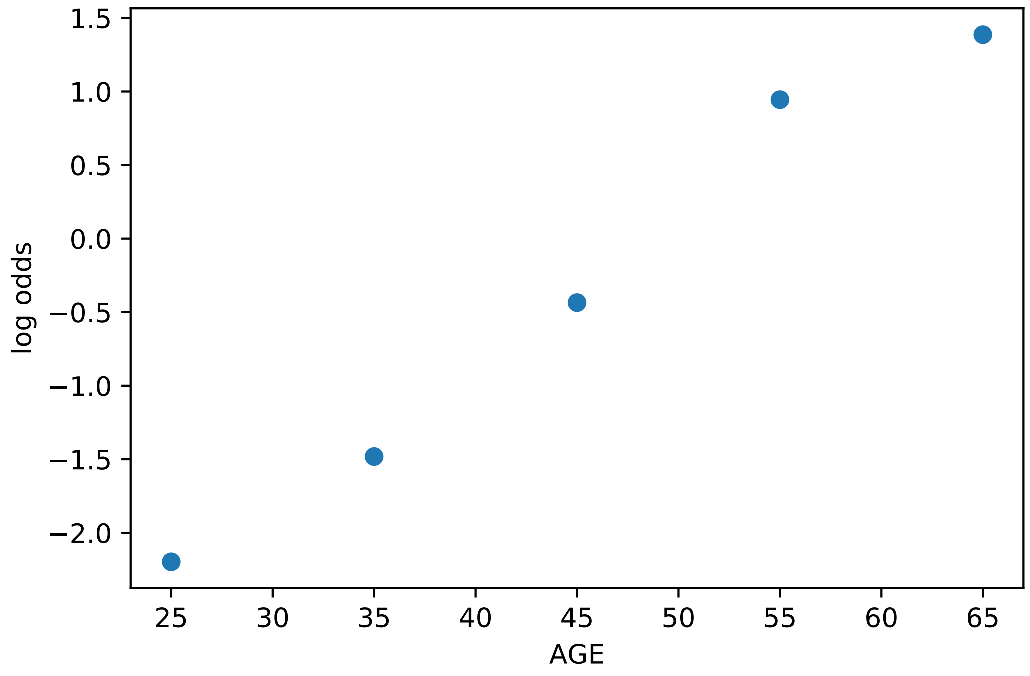
Let's look at that patient data again.

```
[11]: plt.plot(x_bins,y_bins,'r.-')
plt.xlim(20,70); plt.ylim(0,1)
plt.xlabel("Age")
plt.ylabel("Estimated P(CHD|Age)");
```



Let's compute the odds empirically...

```
[12]: x = x_bins
p = y_bins

logit = np.log(1.0*p/(1.0-p))

plt.plot(x,logit,'o')
plt.xlabel("AGE")
plt.ylabel("log odds");
```

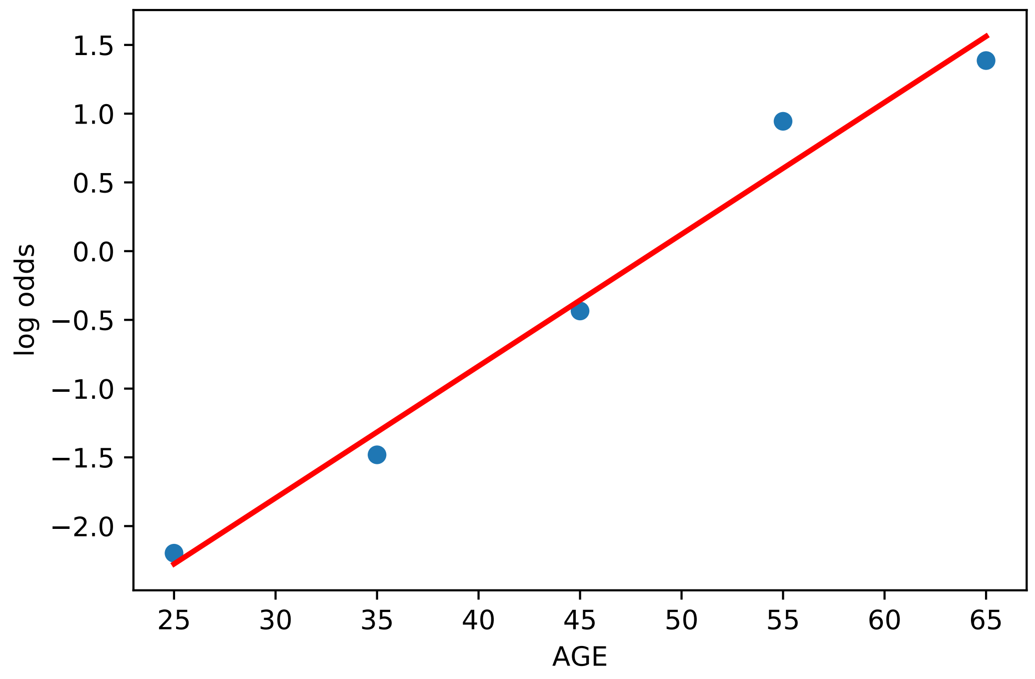


There's still a bit of a curve to the data, but it looks much more like a straight line.

Let's do a regression:

[13]:
```
beta1, beta0, r, p, se = scipy.stats.linregress(x,logit)

print(" Coeffs =", beta0,beta1)
print("    R^2 =", r**2)
print("p-value =", p)

plt.plot(x,logit,'o')

xline = np.linspace(x.min(),x.max(),100)
yline = beta0 + beta1*xline
plt.plot(xline,yline, 'r-', lw=2)

plt.xlabel("AGE")
plt.ylabel("log odds")
plt.show()
```

```
 Coeffs = -4.673575055916286 0.09593104026677286
    R^2 = 0.9800420992383078
p-value = 0.0012038757829788479
```



R^2 has increased dramatically from 0.26 (although it's an unfair comparison)...

This is the core idea behind **logistic regression**: Assume the *logit* statistically obeys a linear model:

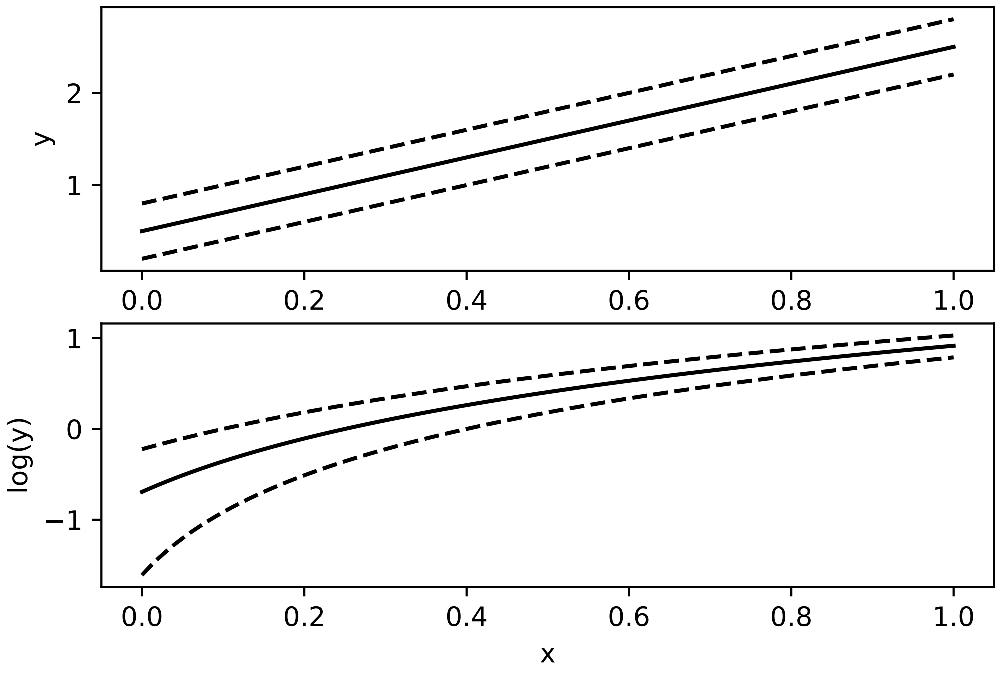
$$\text{logit}(p) = \mathbf{X}\beta$$

where $\mathbf{X}$ is now our design matrix; as with multiple linear regression, we are not just limited to one x variable.

There is a problem however. We have applied a **nonlinear transformation** to the endogenous variable by passing it through the logit. This means we shouldn't just use ordinary least squares to estimate the β.

- Why? Because iid residuals in "linear space" are no longer iid in "log-space":

[14]:
```
x = np.linspace(0,1,100)

y = 2*x + 0.5

yp = y + 0.3
ym = y - 0.3

plt.subplot(211)
plt.plot(x,y, 'k-')
plt.plot(x,yp,'k--', x,ym, 'k--')
plt.xlabel("x"); plt.ylabel("y");

plt.subplot(212)
plt.plot(x,np.log(y), 'k-')
plt.plot(x,np.log(yp),'k--', x,np.log(ym), 'k--')
plt.xlabel("x"); plt.ylabel("log(y)")
plt.show();
```



Assuming iid errors is part of the proof that least-squares regression is unbiased.

Instead of regressing on the transformed variables (which also requires computing p somehow) we can use **maximum likelihood estimation** since we can interpret the endogenous variable probabilistically.

- Sometimes we can write down a simple closed-form equation for the location of the maximum likelihood (in this case as a function of β), but we cannot do this with logistic regression. So instead we use an optimization method to numerically estimate and search for optimum.
- This is a pain to code up ourselves because we can't write down a closed-form expression for the estimators. Instead we need to use a nonlinear optimization technique. Fortunately, `statsmodels` has **got our backs**!

Doing logistic regression

First we should add a column of 1's for the constant term β_0. Statsmodels still uses this even though we aren't going to use OLS.

[15]:
```
df["constant"] = 1.0
print(df.head())
```

```
   AGE  CHD  constant
0   20    0       1.0
1   23    0       1.0
2   24    0       1.0
3   25    0       1.0
4   25    1       1.0
```

Now we fit the model using Logit:

[16]:
```
import statsmodels.api as sm

logit = sm.Logit(df['CHD'], df[['constant','AGE']])
#           ^^^^^

# fit the model
result = logit.fit()
```

```
Optimization terminated successfully.
         Current function value: 0.536765
         Iterations 6
```

Now when fitting, statsmodels gives us details about how the optimization is going, because sometimes the algorithm won't converge to a final answer.

This time it worked great, let's take a peek:

[17]:
```
print(result.summary2())
```

```
                         Results: Logit
=================================================================
Model:              Logit            Pseudo R-squared: 0.214
Dependent Variable: CHD              AIC:              111.3531
Date:               2019-10-31 10:12 BIC:              116.5634
No. Observations:   100              Log-Likelihood:   -53.677
Df Model:           1                LL-Null:          -68.331
Df Residuals:       98               LLR p-value:      6.1680e-08
Converged:          1.0000           Scale:            1.0000
No. Iterations:     6.0000
------------------------------------------------------------------
               Coef.   Std.Err.    z     P>|z|    [0.025   0.975]
------------------------------------------------------------------
constant      -5.3095    1.1337 -4.6835  0.0000  -7.5314  -3.0875
AGE            0.1109    0.0241  4.6102  0.0000   0.0638   0.1581
=================================================================
```

We see that AGE is a very significant predictor for CHD. . . .

- There are z-tests instead of t-tests
- There's no talk about the residuals (the third panel is missing)
- What is "Pseudo R-squared"?
- How do we interpret the numbers under coef?

What if we had forgotten AGE in our model:

[18]:
```
logit_dumb = sm.Logit(df['CHD'], df[['constant']])

# fit the model
result_dumb = logit_dumb.fit()
print(result_dumb.summary2())
```

```
Optimization terminated successfully.
         Current function value: 0.683315
         Iterations 4
                        Results: Logit
===============================================================
Model:              Logit            Pseudo R-squared: 0.000
Dependent Variable: CHD              AIC:              138.6630
Date:               2019-10-31 10:12 BIC:              141.2682
No. Observations:   100              Log-Likelihood:   -68.331
Df Model:           0                LL-Null:          -68.331
Df Residuals:       99               LLR p-value:      nan
Converged:          1.0000           Scale:            1.0000
No. Iterations:     4.0000
----------------------------------------------------------------
              Coef.   Std.Err.     z     P>|z|    [0.025  0.975]
----------------------------------------------------------------
constant     -0.2819    0.2020  -1.3954  0.1629  -0.6777  0.1140
===============================================================
```

Now we just have a constant model, and the pseudo R2 is zero…

The difference between our estimate and the data, called the residual, should not be thought of the same way as with linear regression, because the p has different variance.

Instead, to evaluate goodness-of-fit, `statsmodels` reports what's called McFadden's Pseudo-R-squared. It is the ratio of the likelihoods L of the model you fit vs. the constant model:

$$R^2 = 1 - \frac{\ln L(\text{fitted model})}{\ln L(\text{constant model})}$$

In the second case out fitted model is a constant model so we have $R^2 = 0$.

- There is debate (of course!) but I've seen R^2 between 0.2 and 0.4 to be "good" and $R^2 > 0.4$ to be "very good".
- R^2's for logistic regression almost never get to be near 1.0…

Interpreting logistic regression coefficients

In the fitted logistic model above we had an `AGE` coefficient of 0.1109. What does this mean?

For **linear regression** the coefficient is the **slope**. A one-unit change in x_i gives a β_i-unit change in y, holding all other x_j's ($j \neq i$) fixed. If there's no change, no significant relationship, then we should find $\beta_i \approx 0$.

For **logistic regression** the coefficients are still slopes, but now they are slopes for how the **logit changes**. This is not so intuitive. Instead we can convert them back to see how the **odds** change by undoing the log:

[19]:
```
print(np.exp(result_dumb.params))
```

```
constant    0.754386
dtype: float64
```

This means the odds for having CHD (regardless of age!) is approximate 3 to 4, according to our "dumb" constant model. Solving the odds for p, the probability is now:

[20]:
```
print(1.0 / (1.0+np.exp(-result_dumb.params)))
print()
print(df.mean()) # mean of each variable
```

```
constant    0.43
dtype: float64

AGE         44.38
CHD          0.43
constant     1.00
dtype: float64
```

So we see that the constant-only model is giving us the overall probability of having CHD in the data (43%).

What do we get with the model incorporating age?

[21]:
```
print(result.summary2())
```

```
                         Results: Logit
=================================================================
Model:              Logit            Pseudo R-squared: 0.214
Dependent Variable: CHD              AIC:              111.3531
Date:               2019-10-31 10:12 BIC:              116.5634
No. Observations:   100              Log-Likelihood:   -53.677
Df Model:           1                LL-Null:          -68.331
Df Residuals:       98               LLR p-value:      6.1680e-08
Converged:          1.0000           Scale:            1.0000
No. Iterations:     6.0000
------------------------------------------------------------------
              Coef.   Std.Err.    z     P>|z|    [0.025   0.975]
------------------------------------------------------------------
constant     -5.3095    1.1337  -4.6835  0.0000  -7.5314  -3.0875
AGE           0.1109    0.0241   4.6102  0.0000   0.0638   0.1581
=================================================================
```

[22]:
```
print(np.exp(result.params))
```

```
constant    0.004945
AGE         1.117307
dtype: float64
```

1.117 means that, for a one-unit increase in `AGE` we see, on average, an 11.7% increase in the probability of having heart disease!

We can also apply this calculation to the confidence intervals:

[23]:
```
# odds and 95% CI
params = result.params
conf = result.conf_int()
conf['OR'] = params
conf.columns = ['2.5%', '97.5%', 'OR']
print(np.exp(conf))
```

```
              2.5%     97.5%        OR
constant  0.000536  0.045614  0.004945
AGE       1.065842  1.171257  1.117307
```

- An insignificant coefficient in logistic regression has a value near 1.0.

So that's the idea behind logistic regression. By using a function that incorporates the bounded nature of the data we can more naturally find statistical models.

The coefficients give us information about how the endogenous variable (P(CHD) in this case) depends on the exogenous variable(s) (age). We can also use this model for **classification**.

A more advanced example

Let's try logistic regression on the following data:

[24]:
```
# read the data in
df = pd.read_csv("https://stats.idre.ucla.edu/stat/data/binary.csv")

# rename the 'rank' column because there is also a DataFrame
# method called 'rank'
df.columns = ["admit", "gre", "gpa", "prestige"]

# take a look at the dataset:
print(df.head())
print()
```

```
   admit  gre   gpa  prestige
0      0  380  3.61         3
1      1  660  3.67         3
2      1  800  4.00         1
3      1  640  3.19         4
4      0  520  2.93         4
```

These data describe 400 college students trying to get into grad school. These give the GRE and GPA of the undergrad, and the "prestige" of the undergrad's university (1 is the most prestigous).

(These data are provided by a UCLA online stats tutorial; the Python code was adopted from this very nice blog post.)

Let's summarize all the statistics about the columns:

[25]:
```
print(df.describe())
```

```
            admit         gre         gpa   prestige
count  400.000000  400.000000  400.000000  400.00000
mean     0.317500  587.700000    3.389900    2.48500
std      0.466087  115.516536    0.380567    0.94446
min      0.000000  220.000000    2.260000    1.00000
25%      0.000000  520.000000    3.130000    2.00000
50%      0.000000  580.000000    3.395000    2.00000
75%      1.000000  660.000000    3.670000    3.00000
max      1.000000  800.000000    4.000000    4.00000
```

And here is a frequency table comparing the counts of students being admitted to school versus the "prestige" of the school:

[26]:
```
# table comparing prestige vs admitted (1) or not admitted (0):
print(pd.crosstab(df['admit'], df['prestige']))
```

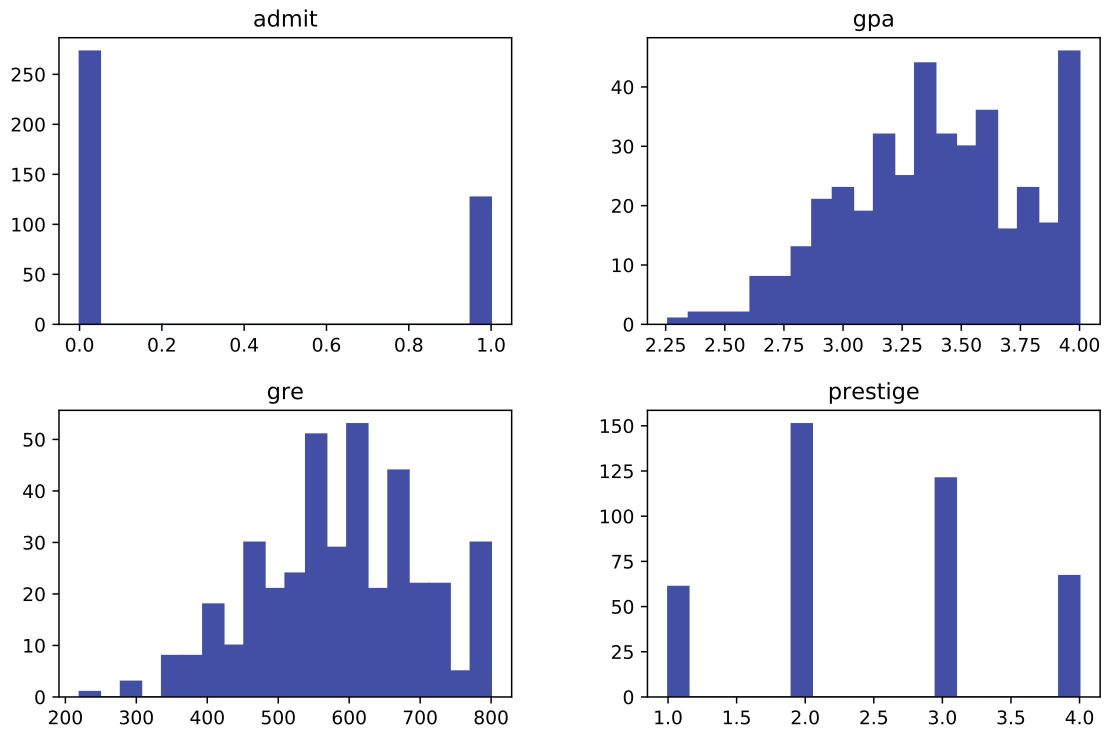
```
prestige   1   2   3   4
admit
0         28  97  93  55
1         33  54  28  12
```

As the prestige drops, the chances of being admitted (`admit = 1`) goes down. Makes sense.

Pandas can quickly summarize each column of data with a histogram:

- (We are being sloppy by not labeling our axes!)

[27]:
```
df.hist(bins=20,fc=niceblu,ec=niceblu, grid=False, figsize=(10,6.5));
```



- Although we would be better served *counting* (with `Counter()`) the `admit` and `prestige` data since they are integer-valued, this is fine for now as a quick-and-dirty visual summary.

Now we can handle GRE and GPA no problem. But look at `prestige`. It is categorical. We know that prestige of 4 is more prestigious than prestige of 1, but is it really **four times** greater? How is prestige measured? What we really want to do is consider only the notion of the different categories, and not make direct comparisons of the magnitude of `prestige`. How can we incorporate this into a regression?

- If `prestige` only had two values we could map those to 0 and 1 using an *indicator variable* and include them in a single equation.

Categorical exogenous variables

We can use such an indicator (or dummy) variable without problem in logistic regression.

And we can also extend it to categorical variables that are **not** binary, they may take any number of distinct values, by using **multiple** binary or dummy variables. This is called **binarization**.

Let's do it for the "prestige" variable. Pandas has a nice builtin for this:

[28]:
```
dummy_ranks = pd.get_dummies(df['prestige'], prefix='prestige')
# `prefix` is the string that begins the names of the columns
# of dummy variable

print(dummy_ranks.head())
```

```
   prestige_1  prestige_2  prestige_3  prestige_4
0           0           0           1           0
1           0           0           1           0
2           1           0           0           0
3           0           0           0           1
4           0           0           0           1
```

There is a problem though, the sum of these four columns in the data is always 1, because each student went to *some* undergrad school.

This means that there is a **linear dependence** between the columns, which can mess up the regression (multicollinearity). This is known as the "dummy variable trap".

- The solution is to **drop** one of the variables, which we don't actually need (we know the dropped variable is 1 if we see all the remaining variables are zero).

Let's drop `prestige_1` and replace the original `prestige` column with the remaining dummy variables in our dataframe:

[29]:
```
cols_to_keep = ['admit', 'gre', 'gpa']

print(df[cols_to_keep].head())
print()

print(dummy_ranks.ix[:, 'prestige_2':].head()) # we can slice dummy vars!

data = df[cols_to_keep].join(dummy_ranks.ix[:, 'prestige_2':])
print()
print(data.head())
```

```
   admit  gre   gpa
0      0  380  3.61
1      1  660  3.67
2      1  800  4.00
3      1  640  3.19
4      0  520  2.93

   prestige_2  prestige_3  prestige_4
0           0           1           0
1           0           1           0
2           0           0           0
3           0           0           1
4           0           0           1

   admit  gre   gpa  prestige_2  prestige_3  prestige_4
0      0  380  3.61           0           1           0
```

```
1       1  660  3.67           0           1           0
2       1  800  4.00           0           0           0
3       1  640  3.19           0           0           1
4       0  520  2.93           0           0           1
```

Add a constant column for the regression (statsmodels uses this, but not all stats packages need a constant column to add a constant term to the model):

[30]:
```
data["constant"] = 1.0
print(data.head())
```

```
   admit  gre   gpa  prestige_2  prestige_3  prestige_4  constant
0      0  380  3.61           0           1           0       1.0
1      1  660  3.67           0           1           0       1.0
2      1  800  4.00           0           0           0       1.0
3      1  640  3.19           0           0           1       1.0
4      0  520  2.93           0           0           1       1.0
```

And now we can regress:

[31]:
```
train_cols = data.columns[1:]

logit = sm.Logit(data['admit'], data[train_cols])

# fit the model
result = logit.fit()
```

```
Optimization terminated successfully.
         Current function value: 0.573147
         Iterations 6
```

[32]:
```
print(result.summary2())
```

```
                         Results: Logit
=================================================================
Model:              Logit            Pseudo R-squared: 0.083
Dependent Variable: admit            AIC:              470.5175
Date:               2019-10-31 10:12 BIC:              494.4663
No. Observations:   400              Log-Likelihood:   -229.26
Df Model:           5                LL-Null:          -249.99
Df Residuals:       394              LLR p-value:      7.5782e-08
Converged:          1.0000           Scale:            1.0000
No. Iterations:     6.0000
------------------------------------------------------------------
               Coef.  Std.Err.    z     P>|z|    [0.025   0.975]
------------------------------------------------------------------
gre            0.0023   0.0011  2.0699  0.0385   0.0001   0.0044
gpa            0.8040   0.3318  2.4231  0.0154   0.1537   1.4544
prestige_2    -0.6754   0.3165 -2.1342  0.0328  -1.2958  -0.0551
prestige_3    -1.3402   0.3453 -3.8812  0.0001  -2.0170  -0.6634
prestige_4    -1.5515   0.4178 -3.7131  0.0002  -2.3704  -0.7325
constant      -3.9900   1.1400 -3.5001  0.0005  -6.2242  -1.7557
=================================================================
```

And let's examine the odds:

[33]:
```
print(np.exp(result.params))
```

```
gre           1.002267
gpa           2.234545
prestige_2    0.508931
prestige_3    0.261792
prestige_4    0.211938
constant      0.018500
dtype: float64
```

We see that GPA has a bigger impact on the odds of being accepted than GRE, although this is because the GRE has a far larger range; a one-unit change in GPA is much bigger than a one-unit change in GRE (refer back to the `df.hist()` histograms to see this). GRE still has a significant effect on the odds, as we see from its p-value.

How do we intepret the `prestige_*` variables?

The omitted rank acts as a baseline, so the 0.5089 for `prestige_2` means we expect a 50% drop in the probability of being accepted if the student was from a rank 2 school compared with a rank 1 school, while holding GRE and GPA fixed. The other coefficients are likewise the change in the odds for those ranks relative to rank 1.

- These kinds of conclusions are the **power** of logistic regression, and they are one reason why logistic regression is not **just** classification!

Look at confidence intervals for the coefficients:

[34]:
```
# odds and 95% CI
params = result.params
conf = result.conf_int()
conf['odds'] = params
conf.columns = ['2.5%', '97.5%', 'odds']
print(np.exp(conf))
```

```
                2.5%     97.5%      odds
gre         1.000120  1.004418  1.002267
gpa         1.166122  4.281877  2.234545
prestige_2  0.273692  0.946358  0.508931
prestige_3  0.133055  0.515089  0.261792
prestige_4  0.093443  0.480692  0.211938
constant    0.001981  0.172783  0.018500
```

Finally, let's check out a **constant-only** model for comparison:

[35]:
```
logit = sm.Logit(data['admit'], data["constant"])

# fit the model
result = logit.fit()

print(result.summary2())
print()
print("Odds =", float(np.exp(result.params)))
print()
print("p =", float(1.0/(1+ np.exp(-result.params))))

print()
print("mean(admit) =", data["admit"].mean())
```

```
Optimization terminated successfully.
         Current function value: 0.624971
         Iterations 4
                        Results: Logit
===============================================================
Model:              Logit            Pseudo R-squared: 0.000
Dependent Variable: admit            AIC:              501.9765
Date:               2019-10-31 10:12 BIC:              505.9680
No. Observations:   400              Log-Likelihood:   -249.99
Df Model:           0                LL-Null:          -249.99
Df Residuals:       399              LLR p-value:      nan
Converged:          1.0000           Scale:            1.0000
No. Iterations:     4.0000
----------------------------------------------------------------
             Coef.  Std.Err.     z     P>|z|    [0.025   0.975]
----------------------------------------------------------------
constant    -0.7653    0.1074  -7.1249  0.0000  -0.9758  -0.5548
===============================================================
```

```
Odds = 0.46520146520146527
```

```
p = 0.3175
```

```
mean(admit) = 0.3175
```

Once again, we recover the baseline probability for admission (0.31) with a constant model

Summary

- Logistic regression uses *some* of the concepts from linear regression to deal with categorical (binary in this case) endogenous data.
- Using binarization, we can extend this to categorical exogenous data as well, even when it's not two-category (binary) data.
- There are even ways to generalize this further, to non-binary categorical endogenous (y) data. This is known as **multinomial logistic regression**.