DATA SOCIETY®

The premiere data science training for professionals

Outline: Logistic Regression

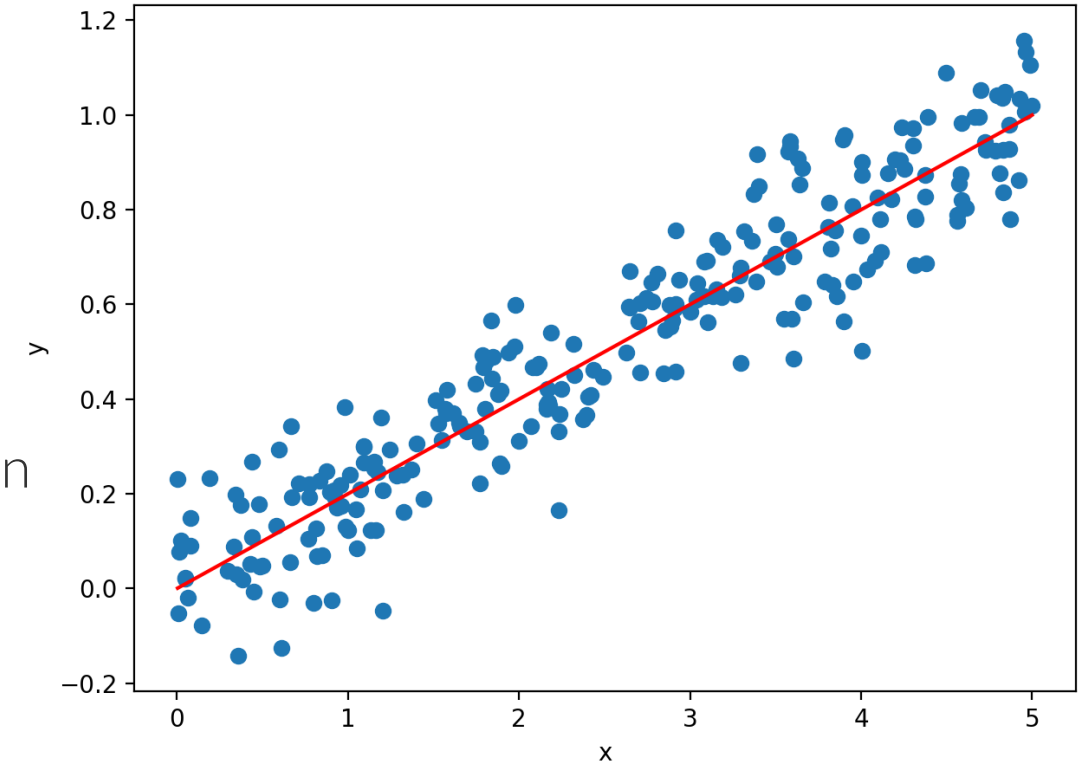
1. The Classification Problem
2. Odds and the Logistic Regression Formulation
3. Python implementation
4. Assessing model quality
5. Multinomial Logistic Regression

Outline: Logistic Regression

1. The Classification Problem
2. Odds and the Logistic Regression Formulation
3. Python implementation
4. Assessing model quality
5. Multinomial Logistic Regression

The Classification Problem

- We have seen linear regression
  - Predicts a continuous variable
  - As a function of other continuous or categorical variables
- Examples:
  - Model spending by fans based on number of years attending games, experience ranking, income
  - Model the height of a plan as a function of the temperature and rainfall
  - Model a patient's blood pressure as a function of their age and weight

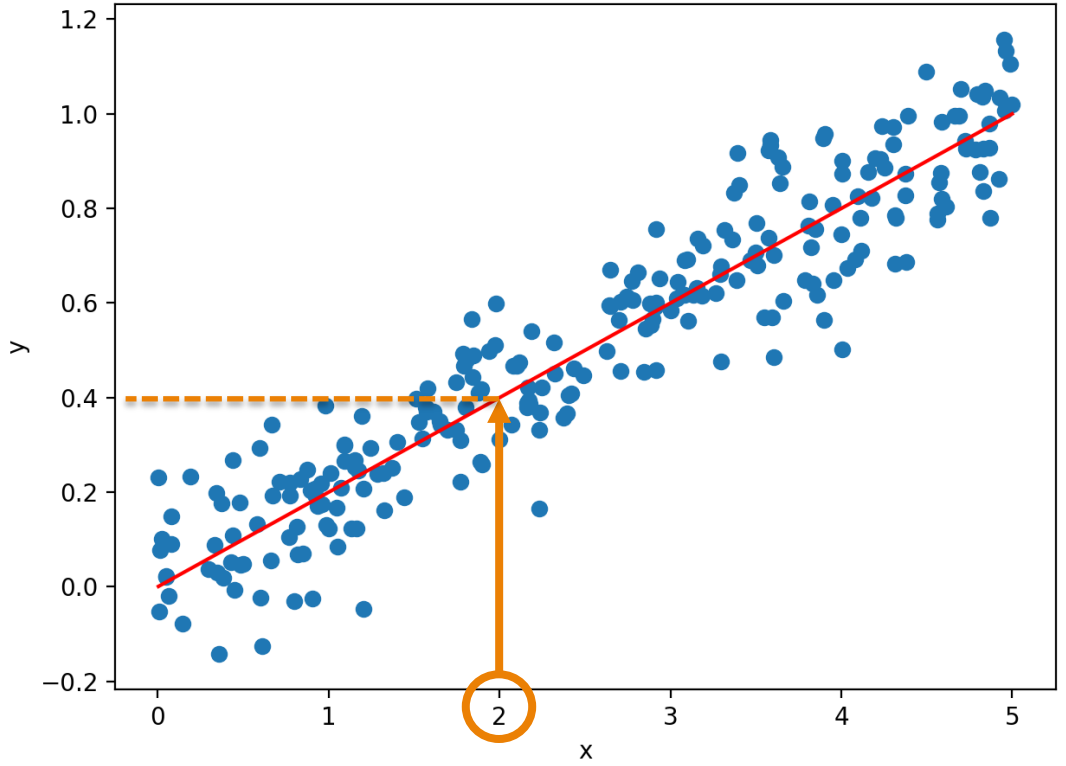


The Classification Problem

- What if the outcome we care about is not a continuous value?
- Example:
  - Which candidate will someone vote for? (A or B)
  - Will someone pass or fail a test? (Pass or Fail)
  - Will someone default on their credit card payment? (Default or No Default)
  - Which of several categories should a book be assigned? (Ctg1, Ctg2, Ctg3, …)
- This type of problem is called a *classification problem*
- Linear regression and Classification form the two most common types of supervised machine learning problems.
  - Incidentally, the name "logistic regression" is somewhat of a misnomer, since we are solving a classification problem, not a regression!

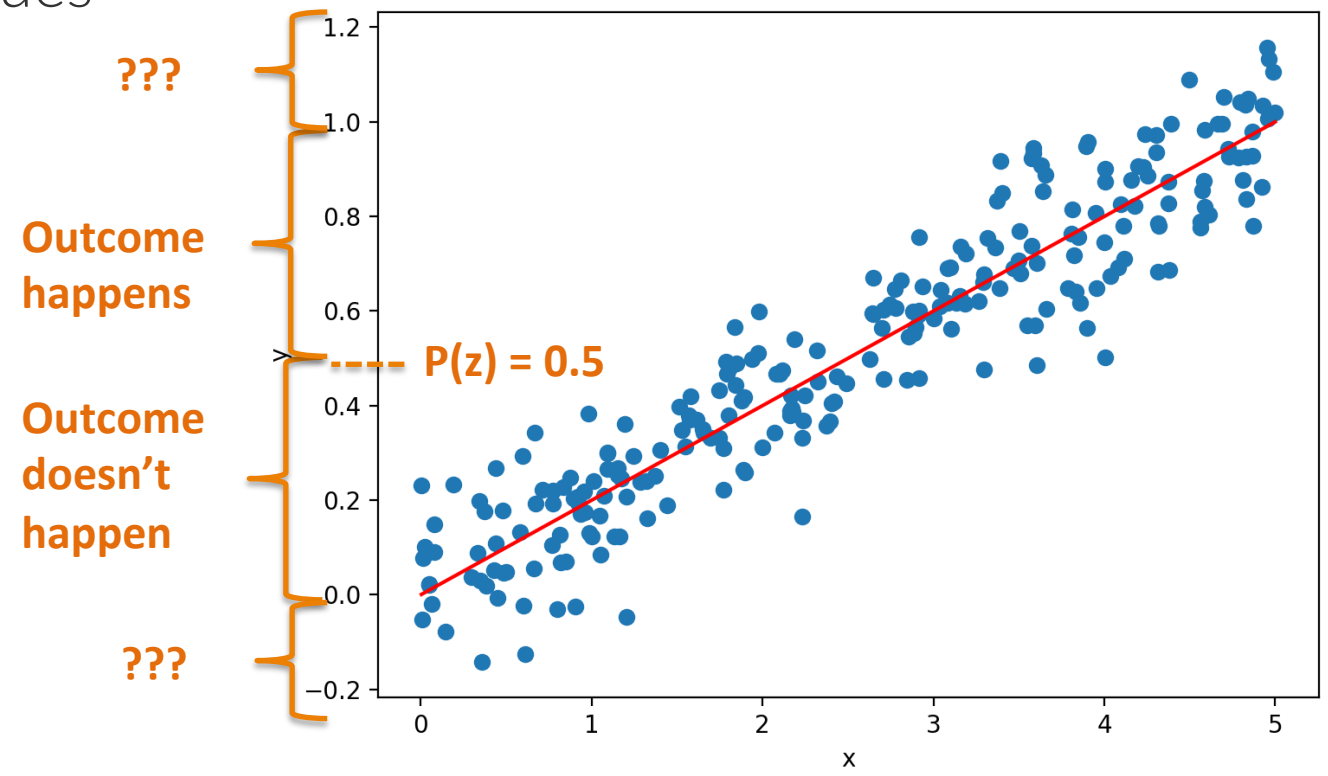
The Classification Problem

- It is difficult to think of a way to use linear regression to solve a classification problem
- One thought would be to treat the continuous y variable as a probability
- If probability < 0.5, assume outcome won't happen. If probability ≥ 0.5, assume will happen
- Example:
  - x = 1 → y = 0.4
  - Probability of desired outcome = 0.4
  - Since probability < 0.5, assume desired outcome won't happen when x =2.



The Classification Problem

- So with this plan, any points above 0.5 are assigned to one outcome, and points below 0.5 are assigned to the other.
- But with a line, we can obtain values larger than 1 and less than 0, which does not make sense as a probability value
- We need a function that automatically cuts off at 0 and 1
  - The logistic function does this.



Outline: Logistic Regression

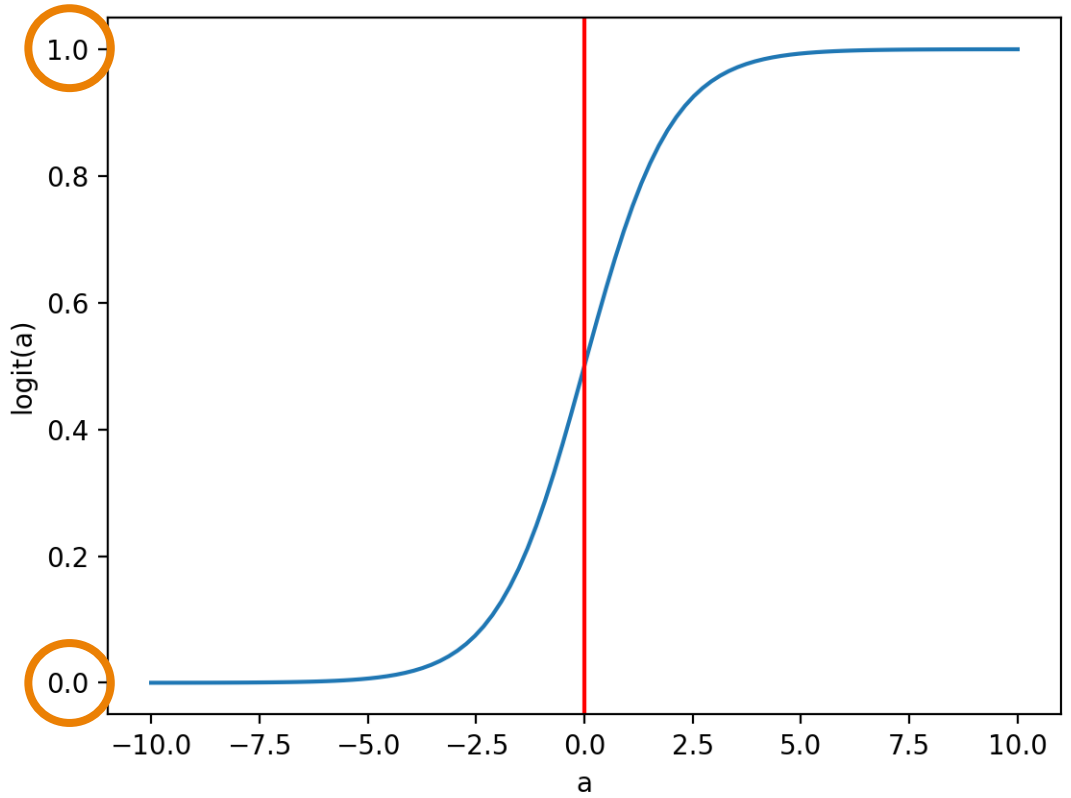
1. The Classification Problem
2. Odds and the Logistic Regression Formulation
3. Python implementation
4. Assessing model quality
5. Multinomial Logistic Regression

Odds and the Logistic Regression Formulation

- We desire a function that automatically cuts off at 0 and 1.
- The logistic function has an S-shape which approaches 0 and 1 but never crosses them:
- P(y) = logistic(a)

$$logistic(a) = \frac{1}{1 + e^{-a}}$$

- e is Euler's number: 2.718281….
  - (aka the base of the natural log function and appearing in a wide range of mathematical functions)



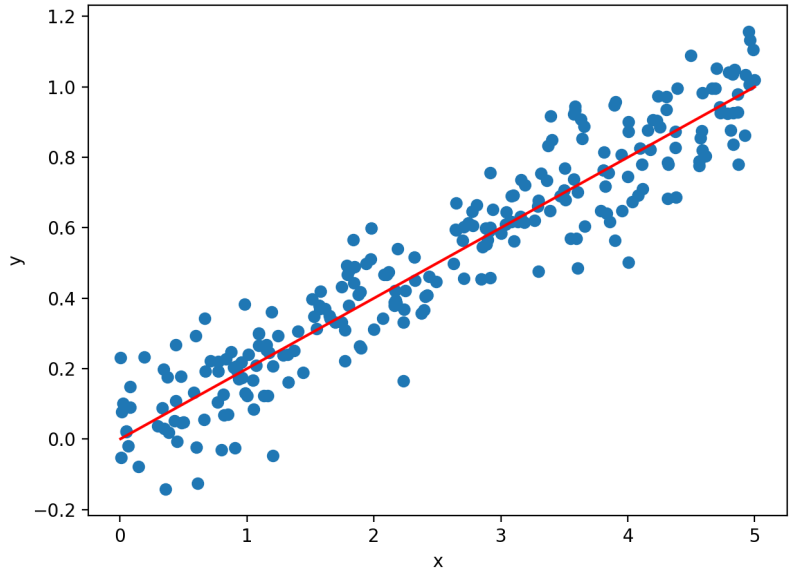
Odds and the Logistic Regression Formulation

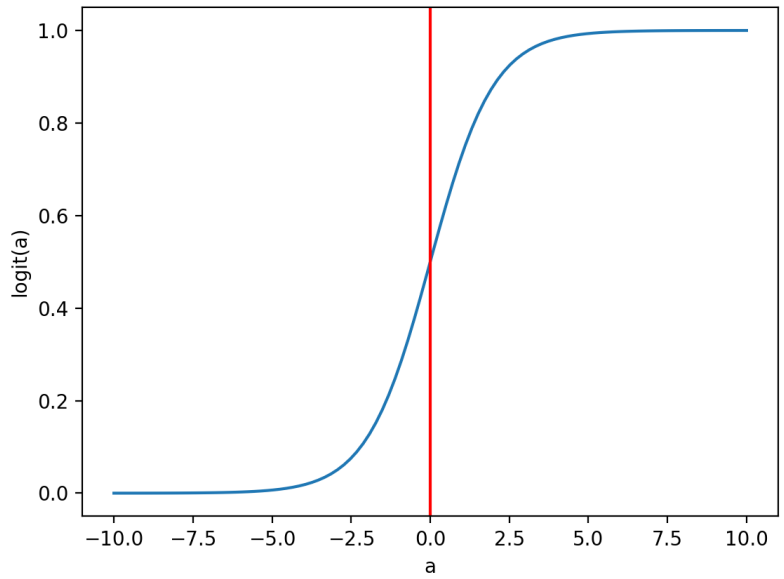
- For a, we use a linear formula, like our linear regression model:

$$logistic(a) = \frac{1}{1 + e^{-a}} \qquad a = \beta_0 + \beta_1 x_1 + \beta_2 x_2$$

Note: these betas are *not* the OLS linear regression coefficients

- The interpretation is: we transform the line into the S-curve via the logistic function





Exercise time!

Odds and the Logistic Regression Formulation

- So we have:

$$P(y) = logistic(a) = \frac{1}{1 + e^{-a}}$$

- Where

$$a = \beta_0 + \beta_1 x_1 + \beta_2 x_2$$

Note: these betas are *not* the OLS linear regression coefficients

- We need a way to estimate these betas

Odds and the Logistic Regression Formulation

- We have:

$$P(y) = logistic(a) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2)}}$$

- If we solve the logistic equation for the linear term, we get the following

$$\beta_0 + \beta_1 x_1 + \beta_2 x_2 = log \frac{P(y)}{1 - P(y)}$$

Odds and the Logistic Regression Formulation

- The right hand side has an intuitive interpretation:

$$\beta_0 + \beta_1 x_1 + \beta_2 x_2 = log \boxed{\frac{P(y)}{1 - P(y)}}$$

This is the *odds* of event y

- Odds = probability of event occurring, divided by probability of event not occurring
  - Example: probability of winning divided by probability of losing
  - "2 to 1 odds" means twice as likely to win as to lose

Exercise time!

Odds and the Logistic Regression Formulation

$$\beta_0 + \beta_1 x_1 + \beta_2 x_2 = log \frac{P(y)}{1 - P(y)}$$

Right-hand-side is called the "Log odds" or "logit"

- We need to find the "best fit" values of the beta's
- In linear regression we used the OLS method to find the intercept and coefficients
- For logistic regression, we use a more general method called *Maximum Likelihood Estimation* (MLE)
  - "Find the beta's that are most likely, given our data"

Odds and the Logistic Regression Formulation

$$\beta_0 + \beta_1 x_1 + \beta_2 x_2 = log \frac{P(y)}{1 - P(y)}$$

Right-hand-side is called the "Log odds" or "logit"

- What is the interpretation of the beta's?
- "An increase of 1 unit in x_1 leads to a β_1 increase in the *log odds* of y"

  OR

- "An increase of 1 unit in x_1 leads to an e^{β_1} increase in the *odds* of y"

Outline: Logistic Regression

1. The Classification Problem
2. Odds and the Logistic Regression Formulation
3. Python implementation
4. Assessing model quality
5. Multinomial Logistic Regression

Example: Credit Card Default

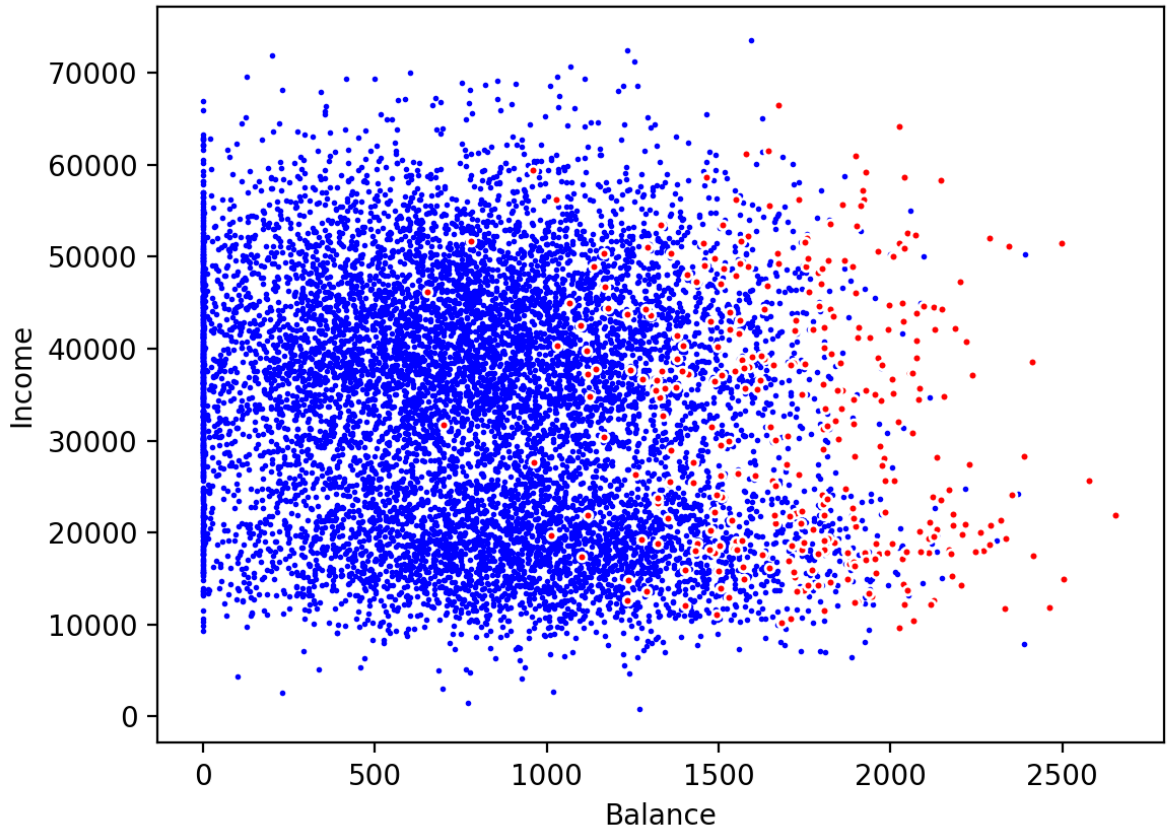
- We will try to predict whether or not someone will default on their credit card debt.
- We have their current balance, their income, and whether or not they are a student.

credit_data - DataFrame

Index	default	student	balance	income
0	0	0	729.526	44361.6
1	0	1	817.18	12106.1
2	0	0	1073.55	31767.1
3	0	0	529.251	35704.5
4	0	0	785.656	38463.5
5	0	1	919.589	7491.56
6	0	0	825.513	24905.2
7	0	1	808.668	17600.5
8	0	0	1161.06	37468.5

Example: Credit Card Default

- Make a plot showing the balance amount and the income.
- Each dot is colored by whether or not they defaulted on their next payment
  - Red = defaulted (`default` = 1)
  - Blue = didn't default (`default` = 0)



Exercise time!

Logistic Regression Estimation

Script

```
from sklearn.linear_model import LogisticRegression
credit_data = pd.read_csv("default.csv")

# Make a LR model using the default settings
model = LogisticRegression()
lr_fit = model.fit(X = credit_data[['balance']],
                   y = credit_data['default'])

beta0 = lr_fit.intercept_[0] # y intercept
beta1 = lr_fit.coef_[0][0] # coefficient of balance
```

Logistic Regression Estimation

Script

```
# Calculate model predictions
y_pred = lr_fit.predict(credit_data['balance'].values.reshape(-1,1))

# Correct classification rate (accuracy):
y_data = credit_data['default']
N = len(credit_data)
sum(y_data == y_pred) / N
```

Odds and the Logistic Regression Formulation

$$\beta_0 + \beta_1 x_1 = log \frac{P(y)}{1 - P(y)}$$

Right-hand-side is called the "Log odds" or "logit"

- We get β_0 = -9.465. So if x_1 (balance) is 0, the probability of default is

  $P(\text{default} \mid \text{balance} = 0) = e^{-9.465} = 7.751\text{E-}05$ (nearly 0)

- And $\beta_1 = 0.00478$. So an increase in balance of $1 leads to an increase of $e^{.00478} = 1.004$ in the odds of default

Exercise time!

Outline: Logistic Regression

1. The Classification Problem
2. Odds and the Logistic Regression Formulation
3. Python implementation
4. Assessing model quality
5. Multinomial Logistic Regression

Assessing Model Quality

- We want to measure *classification quality / error*
- Quality can be measured in terms of *correct classification rate*:
  - Predicted default, and the person did in fact default **"True positive"**
  - Predicted wouldn't default, and the person in fact didn't default **"True negative"**
- Error is measured in terms of *misclassification rate*:
  - Predicted default, but they didn't default **"False positive"**
  - Predicted they wouldn't default, but they did default **"False negative"**

Assessing Model Quality

- The four classification rates can be summarized in what is called a "confusion matrix:"

	Prediction = 1	Prediction = 0
Actual = 1	True Positive	False Negative
Actual = 0	False Positive	True Negative

Accuracy = True positive + True negative

- The diagonal elements give you the rate of correct classification
- The off-diagonal elements give you the misclassification rate

Exercise time!

Assessing Model Quality

- Four our model of default vs balance, we have the following rates:

	Prediction = 1	Prediction = 0
Actual = 1	0.87%	2.46%
Actual = 0	0.28%	96.39%

Accuracy = 0.87% + 96.39% = 97.26%

- We have a very high true negative rate
  - We predict no default, and in fact they did not default.
  - What concern might you have seeing this?

Assessing Model Quality

- Because the vast majority of individuals did not default, what if we simply had a model that predicted "no" for everyone?

Script

```
# What if we just guessed "no default" for everyone?

baserate = sum(y_data == 0) / N
```

- The baseline "no default" rate is 96.67%, so we have only improved our predictive power slightly over this dummy model
  - Takeaway: be careful to not be misled by high accuracy in a model where the base rate is very large (or very small)

Exercise time!

Model Validation

- In a real world setting, we should really be dividing our data set into a test and training set.

Model Validation

- We would like for our model to not just do well on the data that we currently have, but also on any future data we might receive
- For example, we have 10,000 credit card customers in our data set for the current month
- Next month, we will want to predict their default rate, and we hope our model will give us an accurate measure.
- To simulate having unseen data, we artificially remove some data and treat is as the "test." The remaining data ("training") we use to estimate the model. Then we see how well the model performs on the test data.
- Test/ Train also helps us avoid *overfitting*

Model Validation

Script

```
from sklearn.model_selection import train_test_split
X = credit_data[['balance', 'student']]
y_data = credit_data['default']
X_train, X_test, y_train, y_test = train_test_split(X, y_data,
test_size=0.2, random_state=0)

lr_fit2_train = LogisticRegression().fit(X_train, y_train)
lr_fit2_train.intercept_
lr_fit2_train.coef_
```

Model Validation

Script

```
# How does it do on training data?
y_pred2_train = lr_fit2_train.predict(X_train)
sum(y_pred2_train == y_train) / len(y_train)
```

97.22% accuracy on training data

```
# How does it do on test data?
y_pred2_test = lr_fit2_train.predict(X_test)
sum(y_pred2_test == y_test) / len(y_test)
```

97.01% accuracy on test data

Model Validation

Script

```
# How does it do on test data?
y_pred2_test = lr_fit2_train.predict(X_test)
sum(y_pred2_test==y_test)/len(y_test)
```

97.01% accuracy on test data

Adjusting the threshold

- To review, our current setup is:
  1. For each data point, calculate $\beta_0 + \beta_1 x_1 + \beta_2 x_2$
  2. Plug this value into the logistic function:

$$logistic(a) = \frac{1}{1 + e^{-a}}$$

  3. Interpret this value is as a probability (i.e. probability of default)
  4. If this probability > 0.5, then classify as default, otherwise classify as no default.

- What would happen if we used a larger value than 0.5?
- A smaller value?

Adjusting the threshold

- If we used a smaller value, such as 0.3, more defaults would be predicted. This would **increase true positive** and decrease false negatives
  - But would increase the **false positive rate** and decrease true negative rate

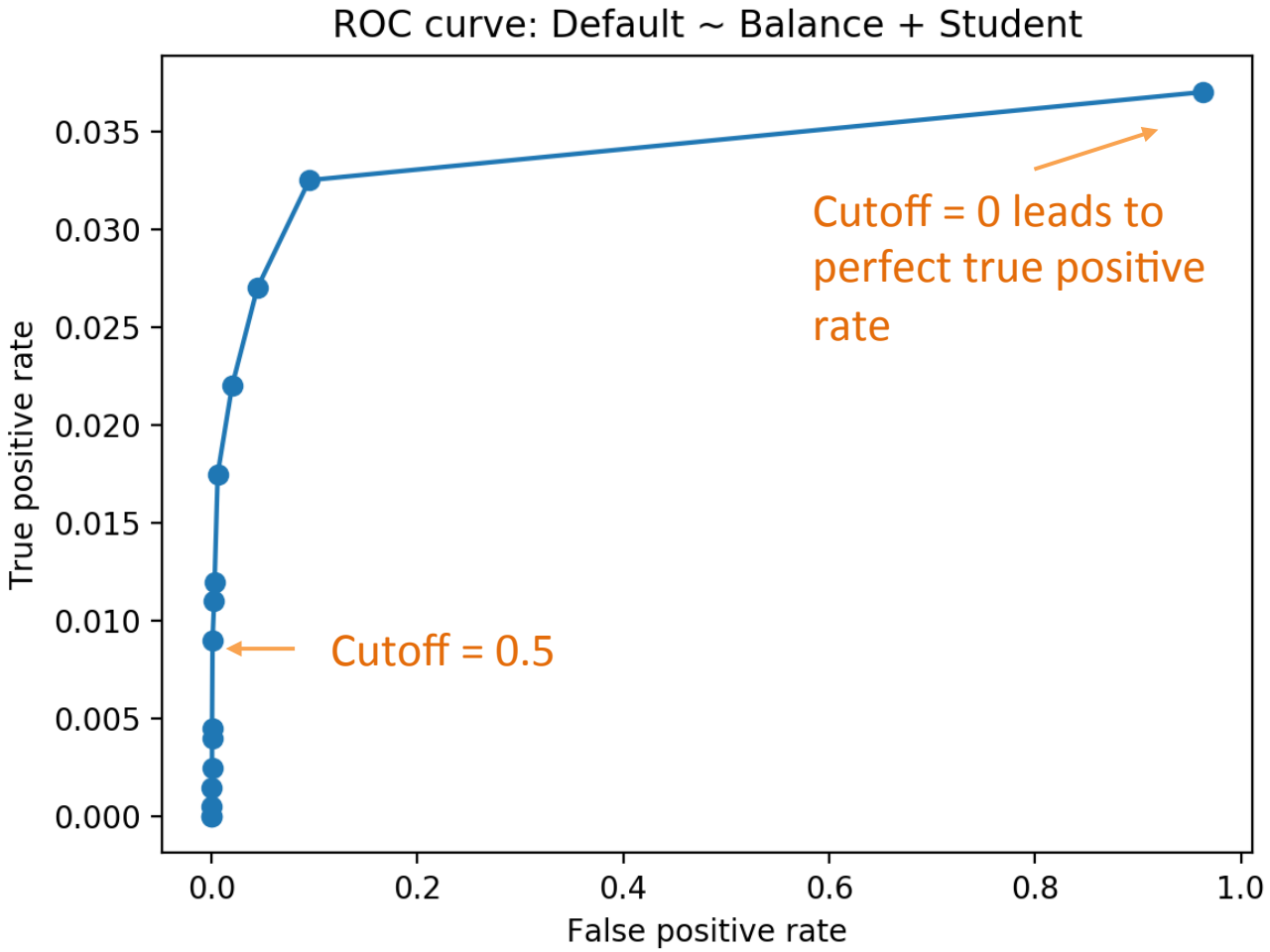
- If we used a larger value, such as 0.7, fewer defaults would be predicted. This would decrease **false positives** and increase the true negative rate
  - But would decrease **true positive** and increase false negative

Exercise time!

Adjusting the threshold

- We can plot this relationship using what's called an ROC curve
- Plot the True Positive Rate vs the False Positive Rate, using different cutoff values to determine each point
- See the lecture script for how to calculate the ROC

```
In [561]: y_test.mean()
Out[561]: 0.037
```



Outline: Logistic Regression

1. The Classification Problem
2. Odds and the Logistic Regression Formulation
3. Python implementation
4. Assessing model quality
5. Multinomial Logistic Regression

Classification with Multiple Outcomes

- Up to this point, we have handled outcomes that are "binary"
  - Will someone default? Yes or No
  - Will someone pass a test? Yes or No
- The binary logistic regression can be expanded to model scenarios where there are multiple discrete outcomes.
- These can be "ordered" or "unordered"
  - Ordered example: A survey that asks "How satisfied are you with the product (1-5)?"
  - Unordered example: Which type of secondary school will someone apply to (

Exercise time!

Classification with Multiple Outcomes

- Behind the scenes, a multinomial logistic regression create a separate binary logistic regression for each of the possible classes
- For example: with 3 classes we'll get 3 sets of coefficients. For each x, it finds the most likely class, among the three sets
- Fortunately, `sklearn` does this all under the hood! So our code will look just like it did for binary logistic regression

Script

```
prog_data = pd.read_csv("hsbdemo.csv")
prog_data['female'] = pd.get_dummies(prog_data['gender'])['female']



# Dummy coded ses = socio-economic-status
prog_data = pd.concat([prog_data, pd.get_dummies(prog_data['ses'])],
                                                    axis = 1)
```

Classification with Multiple Outcomes

Script

```
X = prog_data[['write', 'math', 'high', 'low', 'middle', 'female']]
y = np.ravel(prog_data['prog'])

multinom = LogisticRegression()
multinom_fit = multinom.fit(X, y)
multinom_fit.intercept_
multinom_fit.coef_
y_pred = multinom_fit.predict(X)

sum(y_pred == y) / len(prog_data)
```

This model only has 62% accuracy