# The analytical basetable

import pandas as pd

basetable = pd.DataFrame("import\_basetable.csv")

population\_size = len(basetable)

targets = sum(basetable["Target"])



# Structure of the base table

Consider the predictive modeling problem where you want to predict whether a candidate donor will make a donation in the next year. To build the model, you use historical data and calculate the target in 2017. The target is 1 if a donation is made in 2017 and 0 otherwise. Below, the first lines of the base table are given. It contains the number of donations made in 2016, the number of donations made in 2017, the age of the donor and the target. Which columns can be used as candidate predictors?

| **Donations 2016** | **Donations 2017** | **Age** | **Donation in 2017 (target)** |
| --- | --- | --- | --- |
| 5 | 2 | 68 | 1 |
| 3 | 0 | 65 | 0 |
| 2 | 0 | 23 | 0 |
| 8 | 6 | 56 | 1 |

**Answer the question**

**50 XP**

**Possible Answers**

Donations 2016, Donations 2017 and Age

Donations 2016 and Age

Donations 2017 and Age

Donations 2016 and Donations 2017

Correct! Only information before 2017 can be used.

# Exploring the base table

Before diving into model building, it is important to understand the data you are working with. In this exercise, you will learn how to obtain the population size, number of targets and target incidence from a given basetable.

**Instructions**

**100 XP**

* The basetable is loaded in a pandas object basetable. Assign the number of rows to the variable population\_size and print it.
* Assign the number of targets equal to one to the variable targets\_count and print it.
* Print the target incidence, this is the ratio of targets\_count and population\_size.

[**Take Hint (-30 XP)**](javascript:void(0))

# Assign the number of rows in the basetable to the variable 'population\_size'.

population\_size = len(basetable)

# Print the population size.

print(population\_size)

# Assign the number of targets to the variable 'targets\_count'.

targets\_count = sum(basetable.target)

# Print the number of targets.

print(targets\_count)

# Print the target incidence.

print(targets\_count / population\_size)

<script.py> output:

100000

4990

0.0499

+100 XP

Great job! As in many real world predictive modeling cases, the incidence is rather low. This makes model building challenging!

# Exploring the predictive variables

It is always useful to get a better understanding of the population. Therefore, one can have a closer look at the predictive variables. Recall that you can select a column in a pandas dataframe by indexing as follows:

basetable["variable"]

To count the number of occurrences of a certain value in a column, you can use the sum method:

sum(basetable["variable"]==value)

In this exercise you will find out whether there are more males than females in the population.

**Instructions**

**100 XP**

**Instructions**

**100 XP**

* Count and print the number of females.
* Count and print the number of males.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Did you call sum()?

# Count and print the number of females.

print(sum(basetable.gender == 'F'))

# Count and print the number of males.

print(sum(basetable.gender == 'M'))

<script.py> output:

50624

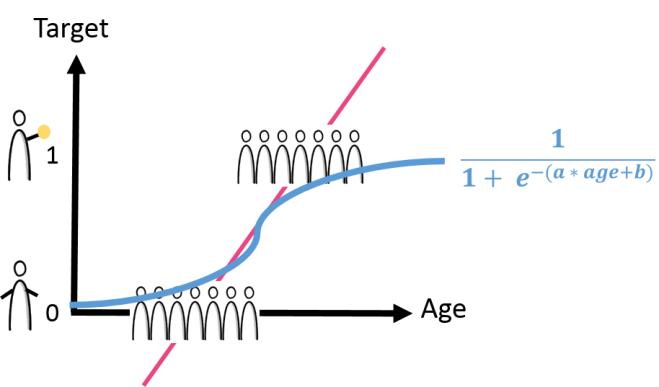
49376

Well done! From these results, you can see that there are more females in the basetable.

# Logistic regression in Python

Logistic regression: the logit function

* Output of a\*age + b*a*∗*age*+*b* is a real number
* We want to predict a 0 or a 1
* Logit function transforms a\*age + b*a*∗*age*+*b* to a probability



from sklearn import linear\_model

logreg = linear\_model.LogisticRegression()

X = basetable[["age"]]

y = basetable[["target"]]

logreg.fit(X,y)

print(logreg.coef\_)

[[ 0.02449202]]

print(logreg.intercept\_)

[-4.3299131]

Multivariate logistic regression

Univariate: ax + b*ax*+*b*

Multivariate: a\_1x\_1 + a\_2x\_2 + ... + a\_nx\_n + b*a*​1​​*x*​1​​+*a*​2​​*x*​2​​+...+*a*​*n*​​*x*​*n*​​+*b*

X = basetable[["age","max\_gift","income\_low"]]

y = basetable[["target"]]

logreg.fit(X,y)

print(logreg.coef\_)

[[ 0.0243308 0.03906065 -0.76793773]]

print(logreg.intercept\_)

[-8.80643545]

**Interpretation of coefficients**

Assume you built a logistic regression model to predict which donors are most likely to donate for a project, using age and time\_since\_last\_gift (number of months since the last gift) as predictors. The output of the logistic regression model is as follows:

y = 0.3 + 4.5\*age - 2.3\*time\_since\_last\_gift

Which of the following statements holds, according to the model?

**Answer the question**

**50 XP**

**Possible Answers**

Older donors that recently donated are most likely to donate.

Younger donors that recently donated are most likely to donate.

Older donors that didn't donate in a long time are most likely to donate.

Younger donors that didn't donate in a long time are most likely to donate.

Take Hint (-15xp)

**Incorrect Submission**

Incorrect. The coefficient associated with time since last gift is negative, which means that time since last gift is negatively correlated with the target: the longer since the last donation, the less likely he will donate.

Awesome, thanks for your feedback!



 +50 XP

Correct! By looking at the coefficients you can see that age is positively correlated with the target, and time since last gift negatively.

# Building a logistic regression model

You can build a logistic regression model using the module linear\_model from sklearn. First, you create a logistic regression model using the LogisticRegression() method:

logreg = linear\_model.LogisticRegression()

Next, you need to feed data to the logistic regression model, so that it can be fit. X contains the predictive variables, whereas y has the target.

X = basetable[["predictor\_1","predictor\_2","predictor\_3"]]`

y = basetable[["target"]]

logreg.fit(X,y)

In this exercise you will build your first predictive model using three predictors.

**Instructions**

**100 XP**

* Import the methodlinear\_model from sklearn.
* The basetable is loaded as basetable. Note that the column "gender" has been transformed to gender\_F so that it can be used as a predictor. Construct a dataframe X that contains the predictors age, gender\_F and time\_since\_last\_gift.
* Construct a dataframe y that contains the target.
* Create a logistic regression model.
* Fit the logistic regression model on the given basetable.

# Import linear\_model from sklearn.

from sklearn import linear\_model

# Create a dataframe X that only contains the candidate predictors age, gender\_F and time\_since\_last\_gift.

X = basetable[['age', 'gender\_F', 'time\_since\_last\_gift']]

# Create a dataframe y that contains the target.

y = basetable[['target']]

# Create a logistic regression model logreg and fit it to the data.

logreg = linear\_model.LogisticRegression()

logreg.fit(X,y)

LogisticRegression(C=1.0, class\_weight=None, dual=False, fit\_intercept=True,

intercept\_scaling=1, max\_iter=100, multi\_class='ovr', n\_jobs=1,

penalty='l2', random\_state=None, solver='liblinear', tol=0.0001,

verbose=0, warm\_start=False)

 +100 XP

Nice! The logreg now holds a logistic regression model that is fit to your data.

**Showing the coefficients and intercept**

Once the logistic regression model is ready, it can be interesting to have a look at the coefficients to check whether the model makes sense.

Given a fitted logistic regression model logreg, you can retrieve the coefficients using the attribute coef\_. The order in which the coefficients appear, is the same as the order in which the variables were fed to the model. The intercept can be retrieved using the attribute intercept\_.

The logistic regression model that you built in the previous exercises has been added and fitted for you in logreg.

**Instructions**

**100 XP**

* Assign the coefficients of the logistic regression model to the list coef.
* Assign the intercept of the logistic regression model to the variable intercept.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Did you define the variable intercept without errors?

# Construct a logistic regression model that predicts the target using age, gender\_F and time\_since\_last gift

predictors = ["age","gender\_F","time\_since\_last\_gift"]

X = basetable[predictors]

y = basetable[["target"]]

logreg = linear\_model.LogisticRegression()

logreg.fit(X, y)

# Assign the coefficients to a list coef

coef = logreg.coef\_

for p,c in zip(predictors,list(coef[0])):

print(p + '\t' + str(c))

# Assign the intercept to the variable intercept

intercept = logreg.intercept\_

print(intercept)

# Construct a logistic regression model that predicts the target using age, gender\_F and time\_since\_last gift

predictors = ["age","gender\_F","time\_since\_last\_gift"]

X = basetable[predictors]

y = basetable[["target"]]

logreg = linear\_model.LogisticRegression()

logreg.fit(X, y)

# Assign the coefficients to a list coef

coef = logreg.coef\_

for p,c in zip(predictors,list(coef[0])):

print(p + '\t' + str(c))

# Assign the intercept to the variable intercept

intercept = logreg.intercept\_

print(intercept)

 +100 XP

Great! The coefficient of gender\_F is positive, meaning that women are more likely to donate.

# The logistic regression function

Calculated Manually:

0.545 \* gender\_F

+ 0.021 \* age

-0.001 \* time\_since\_last\_gift

-3.39

* Female (gender\_F=1)
* age 72
* 120 days since last gift

0.545 \* 1

+ 0.021 \* 72

-0.001 \* 120

-3.39

= -1.45



Calculated by scikit-learn in Python:

Making predictions in Python

* Female (gender\_F=1)
* Age 72
* 120 days since last gift

logreg.predict\_proba([1, 72, 120])

array([[ 0.8204144, 0.1795856]])

Making predictions in Python

new\_data = current\_data[["gender\_F","age","time\_since\_last\_gift"]]

predictions = logreg.predict\_proba(new\_data)

# Making predictions

Once your model is ready, you can use it to make predictions for a campaign. It is important to always use the latest information to make predictions.

In this exercise you will, given a fitted logistic regression model, learn how to make predictions for a new, updated basetable.

The logistic regression model that you built in the previous exercises has been added and fitted for you in logreg.

**Instructions**

**100 XP**

* The latest data is in current\_data. Create a data frame new\_data that selects the relevant columns from current\_data.
* Assign to predictions the predictions for the observations in new\_data.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Did you call logreg.predict\_proba()?

# Fit a logistic regression model

from sklearn import linear\_model

X = basetable[["age","gender\_F","time\_since\_last\_gift"]]

y = basetable[["target"]]

logreg = linear\_model.LogisticRegression()

logreg.fit(X, y)

# Create a dataframe new\_data from current\_data that has only the relevant predictors

new\_data = current\_data[["age","gender\_F","time\_since\_last\_gift"]]

# Make a prediction for each observation in new\_data and assign it to predictions

predictions = logreg.predict\_proba(new\_data)

print(predictions[0:5])

<script.py> output:

[[0.94254824 0.05745176]

[0.97375722 0.02624278]

[0.96793614 0.03206386]

[0.90081414 0.09918586]

[0.96936349 0.03063651]]

 +100 XP

Well done! The predictions consist of two values. The second value is the probability that the observation is a target.

# Donor that is most likely to donate

The predictions that result from the predictive model reflect how likely it is that someone is a target. For instance, assume that you constructed a model to predict whether a donor will donate more than 50 Euro for a certain campaign. If the prediction for a certain donor is 0.82, it means that there is an 82% chance that he will donate more than 50 Euro.

In this exercise you will find the donor that is most likely to donate more than 50 Euro.

Recall that you can sort a pandas dataframe df according to a certain column c using

df\_sorted = df.sort(["c"])

and that you can select the first and last row of a pandas dataframe using

first\_row = df.head(1)

last\_row = df.tail(1)

**Instructions**

**100 XP**

* The predictions are in a pandas dataframe predictions that has two columns: the donor ID and the probability to be target. Sort these predictions such that the donors with lowest probability to donate are first.
* Select and print the row in this sorted dataframe that has the donor that is most likely to donate more than 50 Euro according to the model.

[**Take Hint (-30 XP)**](javascript:void(0))

**Incorrect Submission**

Did you call predictions\_sorted.tail()?

# Sort the predictions

predictions\_sorted = predictions.sort(['probability'])

# Print the row of predictions\_sorted that has the donor that is most likely to donate

print(predictions\_sorted.tail(1))

<script.py> output:

donor\_ID probability

651 3413 0.119776

 +100 XP

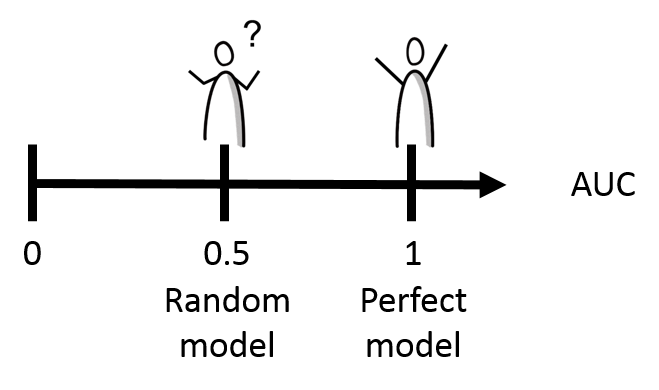
Great job. The donor that is most likely to donate still has a rather low probability to donate, this is due to the fact that the overal target incidence is low.

Variable selection: motivation

Drawbacks of models with many variables:

* Over-fitting
* Hard to maintain or implement
* Hard to interpret, multi-collinearity

Model evaluation: AUC



import numpy as np

from sklearn.metrics import roc\_auc\_score

roc\_auc\_score(true\_target, prob\_target)

# Which model is best?

Imagine you built 4 models:

A: A model with 10 variables that has an AUC of 0.76 B: A model with 10 variables that has an AUC of 0.73 C: A model with 15 variables that has an AUC of 0.76 D: A model with 15 variables that has an AUC of 0.73

Which model is best, assuming all variables are equally easy to calculate and maintain.

**Answer the question**

**50 XP**

**Possible Answers**

Model A

Model B

Model C

Model D

 +50 XP

Correct! This is the least complex model with best performance.

# Calculating AUC

The AUC value assesses how well a model can order observations from low probability to be target to high probability to be target. In Python, the roc\_auc\_score function can be used to calculate the AUC of the model. It takes the true values of the target and the predictions as arguments.

You will make predictions again, before calculating its roc\_auc\_score.

**Instructions**

**70 XP**

**Instructions**

**70 XP**

* The model logreg from the last chapter has been created and fitted for you, the dataframe X contains the predictor columns of the basetable. Make predictions for the objects in the basetable.
* Select the second column of predictions, as it contains the predictions for the target.
* The true values of the target are loaded in y. Use the roc\_auc\_score function to calculate the AUC of the model.

[**Show Answer (-70 XP)**](javascript:void(0))

**Incorrect Submission**

Check your call of roc\_auc\_score(). Did you correctly specify the argument y\_score? Expected predictions\_target, but got predictions.

# Make predictions

predictions = logreg.predict\_proba(X)

predictions\_target = predictions[:,1]

# Calculate the AUC value

auc = roc\_auc\_score(y, predictions\_target)

print(round(auc,2))

<script.py> output:

0.69

 +70 XP

Nice! An AUC of 0.69 is a typical result for this type of cases. Let's check if we can improve using different sets of variables.

# Using different sets of variables

Adding more variables and therefore more complexity to your logistic regression model does not automatically result in more accurate models. In this exercise you can verify whether adding 3 variables to a model leads to a more accurate model.

variables\_1 and variables\_2 are available in your environment: you can print them to the console to explore what they look like.

**Instructions**

**100 XP**

* Fit the logreg model using variables\_2 which contains 3 additional variables compared to variables\_1.
* Make predictions for this model.
* Calculate the AUC of this model.

[**Take Hint (-30 XP)**](javascript:void(0))

# Create appropriate dataframes

X\_1 = basetable[variables\_1]

X\_2 = basetable[variables\_2]

y = basetable[["target"]]

# Create the logistic regression model

logreg = linear\_model.LogisticRegression()

# Make predictions using the first set of variables and assign the AUC to auc\_1

logreg.fit(X\_1, y)

predictions\_1 = logreg.predict\_proba(X\_1)[:,1]

auc\_1 = roc\_auc\_score(y, predictions\_1)

# Make predictions using the second set of variables and assign the AUC to auc\_2

logreg.fit(X\_2, y)

predictions\_2 = logreg.predict\_proba(X\_2)[:,1]

auc\_2 = roc\_auc\_score(y, predictions\_2)

# Print auc\_1 and auc\_2

print(round(auc\_1,2))

print(round(auc\_2,2))

<script.py> output:

0.69

0.69

 +100 XP

Well done. You can see that the model with 5 variables has the same AUC as the model using only 2 variables. Adding more variables doesn't always increase the AUC.

# Implementation of the AUC function

from sklearn import linear\_model

from sklearn.metrics import roc\_auc\_score

def auc(variables, target, basetable):

X = basetable[variables]

y = basetable[target]

logreg = linear\_model.LogisticRegression()

logreg.fit(X, y)

predictions = logreg.predict\_proba(X)[:,1]

auc = roc\_auc\_score(y, predictions)

return(auc)

auc = auc(["age","gender\_F"],["target"],basetable)

print(round(auc,2))

0.54

## Calculating the next best variable

def next\_best(current\_variables,candidate\_variables, target, basetable):

best\_auc = -1

best\_variable = None

for v in candidate\_variables:

auc\_v = auc(current\_variables + [v], target, basetable)

if auc\_v >= best\_auc:

best\_auc = auc\_v

best\_variable = v

return best\_variable

current\_variables = ["age","gender\_F"]

candidate\_variables = ["min\_gift","max\_gift","mean\_gift"]

next\_variable = next\_best(current\_variables, candidate\_variables, basetable)

print(next\_variable)

min\_gift

## The forward stepwise variable selection procedure

candidate\_variables = ["mean\_gift","min\_gift","max\_gift",

"age","gender\_F","country\_USA","income\_low"]

current\_variables = []

target = ["target"]

max\_number\_variables = 5

number\_iterations = min(max\_number\_variables, len(candidate\_variables))

for i in range(0,number\_iterations):

next\_var = next\_best(current\_variables,candidate\_variables,target,basetable)

current\_variables = current\_variables + [next\_variable]

candidate\_variables.remove(next\_variable)

print(current\_variables)

['max\_gift', 'mean\_gift', 'min\_gift', 'age', 'gender\_F']

# Selecting the next best variable

The forward stepwise variable selection method starts with an empty variable set and proceeds in steps, where in each step the next best variable is added. To implement this procedure, two handy functions have been implemented for you.

The auc function calculates for a given variable set variables the AUC of the model that uses this variable set as predictors. The next\_best function calculates which variable should be added in the next step to the variable list.

In this exercise, you will experiment with these functions to better understand their purpose. You will calculate the AUC of a given variable set, calculate which variable should be added next, and verify that this indeed results in an optimal AUC.

**Instructions**

**100 XP**

**Instructions**

**100 XP**

* The auc function has been implemented for you. Calculate the AUC of a model that uses "max\_gift", "mean\_gift" and "min\_gift" as predictors. You should pass these variables in a list as the first argument to the auc function.
* The next\_best function has been implemented for you. Calculate which variable should be added next, given that "max\_gift", "mean\_gift" and "min\_gift" are currently in the model, and "age" and "gender\_F" are the candidate next predictors. The first argument of the next\_best function is a list with the current variables, while the second argument is a list with the candidate predictors.
* Calculate the AUC of a model that uses "max\_gift", "mean\_gift", "min\_gift" and "age" as predictors.
* Calculate the AUC of a model that uses "max\_gift", "mean\_gift", "min\_gift" and "gender\_F" as predictors.

# Calculate the AUC of a model that uses "max\_gift", "mean\_gift" and "min\_gift" as predictors

auc\_current = auc(['max\_gift', 'mean\_gift', 'min\_gift'], ["target"], basetable)

print(round(auc\_current,4))

# Calculate which variable among "age" and "gender\_F" should be added to the variables "max\_gift", "mean\_gift" and "min\_gift"

next\_variable = next\_best(['max\_gift', 'mean\_gift', 'min\_gift'], ['age', 'gender\_F'], ["target"], basetable)

print(next\_variable)

# Calculate the AUC of a model that uses "max\_gift", "mean\_gift", "min\_gift" and "age" as predictors

auc\_current\_age = auc(['max\_gift', 'mean\_gift', 'min\_gift', 'age'], ["target"], basetable)

print(round(auc\_current\_age,4))

# Calculate the AUC of a model that uses "max\_gift", "mean\_gift", "min\_gift" and "gender\_F" as predictors

auc\_current\_gender\_F = auc(['max\_gift', 'mean\_gift', 'min\_gift', 'gender\_F'], ["target"], basetable)

print(round(auc\_current\_gender\_F,4))

<script.py> output:

0.715

age

0.7174

0.7162

 +100 XP

Nice! The model that has age as next variable has a better AUC than the model that has gender\_F as next variable. Therefore, age is selected as the next best variable.

# Calculate the AUC of a model that uses "max\_gift", "mean\_gift", "min\_gift" and "gender\_F" as predictors

auc\_current\_gender\_F = auc(['max\_gift', 'mean\_gift', 'min\_gift', 'age', 'gender\_F'], ["target"], basetable)

print(round(auc\_current\_gender\_F,4))

0.7186

**Finding the order of variables**

The forward stepwise variable selection procedure starts with an empty set of variables, and adds predictors one by one. In each step, the predictor that has the highest AUC in combination with the current variables is selected.

In this exercise you will learn to implement the forward stepwise variable selection procedure. To this end, you can use the next\_best function that has been implemented for you. It can be used as follows:

next\_best(current\_variables,candidate\_variables,target,basetable)

where current\_variables is the list of variables that is already in the model and candidate\_variables the list of variables that can be added next.

**Instructions**

**100 XP**

* Use the function next\_best to calculate the next best variable and assign it to next\_variable.
* Update the current\_variables list.
* Update the candidate\_variables list.

# Find the candidate variables

candidate\_variables = list(basetable.columns.values)

candidate\_variables.remove("target")

# Initialize the current variables

current\_variables = []

# The forward stepwise variable selection procedure

number\_iterations = 5

for i in range(0, number\_iterations):

next\_variable = next\_best(current\_variables, candidate\_variables, ["target"], basetable)

current\_variables += [next\_variable]

candidate\_variables.remove(next\_variable)

print("Variable added in step " + str(i+1) + " is " + next\_variable + ".")

print(current\_variables)

<script.py> output:

Variable added in step 1 is max\_gift.

Variable added in step 2 is number\_gift.

Variable added in step 3 is time\_since\_last\_gift.

Variable added in step 4 is mean\_gift.

Variable added in step 5 is income\_high.

['max\_gift', 'number\_gift', 'time\_since\_last\_gift', 'mean\_gift', 'income\_high']

# Correlated variables

The first 10 variables that are added to the model are the following:

['max\_gift', 'number\_gift', 'time\_since\_last\_gift', 'mean\_gift', 'income\_high', 'age', 'country\_USA', 'gender\_F', 'income\_low', 'country\_UK']

As you can see, min\_gift is not added. Does this mean that it is a bad variable? You can test the performance of the variable by using it in a model as a single variable and calculating the AUC. How does the AUC of min\_gift compare to the AUC of income\_high? To this end, you can use the function auc():

auc(variables, target, basetable)

It can happen that a good variable is not added because it is highly correlated with a variable that is already in the model. You can test this calculating the correlation between these variables:

import numpy

numpy.corrcoef(basetable["variable\_1"],basetable["variable\_2"])[0,1]

**Instructions**

**100 XP**

* Calculate the AUC of the model using the variable min\_gift only.
* Calculate the AUC of the model using the variable income\_high only.
* Calculate the correlation between the variable min\_gift and mean\_gift.

import numpy as np

# Calculate the AUC of the model using min\_gift only

auc\_min\_gift = auc(['min\_gift'], ["target"], basetable)

print(round(auc\_min\_gift,2))

# Calculate the AUC of the model using income\_high only

auc\_income\_high = auc(['income\_high'], ["target"], basetable)

print(round(auc\_income\_high,2))

# Calculate the correlation between min\_gift and mean\_gift

correlation = np.corrcoef(basetable["min\_gift"], basetable["mean\_gift"])[0,1]

print(round(correlation,2))

<script.py> output:

0.57

0.52

0.76

 +100 XP

Well done! You can observe that min\_gift has more predictive power than income\_high, but that it is highly correlated with mean\_gift and therefore not included in the selected variables.

**Exercise**

**Exercise**

**Partitioning**

In order to properly evaluate a model, one can partition the data in a train and test set. The train set contains the data the model is built on, and the test data is used to evaluate the model. This division is done randomly, but when the target incidence is low, it could be necessary to stratify, that is, to make sure that the train and test data contain an equal percentage of targets.

In this exercise you will partition the data with stratification and verify that the train and test data have equal target incidence. The train\_test\_split method has already been imported, and the X and y dataframes are available in your workspace.

**Instructions**

**100 XP**

* Stratify these dataframes using the train\_test\_split method. Make sure that train and test set are the same size, and have equal target incidence.
* Calculate the target incidence of the train set. This is the number of targets in the train set divided by the number of observations in the train set.
* Calculate the target incidence of the test set.

[**Take Hint (-30 XP)**](javascript:void(0))

**Hint**

* The test\_size argument indicates which percentage of the data should be in the test set. It should be chosen such that the test set contains half of the data. To make sure train and test set have equal target incidence, the stratify argument should hold the target dataframe y.
* To obtain the number of targets in train, you should sum the observations in train["target"].

# Load the partitioning module

from sklearn.cross\_validation import train\_test\_split as tts

# Create dataframes with variables and target

X = basetable.drop("target", 1)

y = basetable["target"]

# Carry out 50-50 partititioning with stratification

X\_train, X\_test, y\_train, y\_test = tts(X, y, test\_size = 0.5, stratify = y)

# Create the final train and test basetables

train = pd.concat([X\_train, y\_train], axis=1)

test = pd.concat([X\_test, y\_test], axis=1)

# Check whether train and test have same percentage targets

print(round(sum(train["target"])/len(train), 2))

print(round(sum(test["target"])/len(test), 2))

<script.py> output:

0.05

0.05

In [1]:

+70 XP

Splendid! The stratify option makes sure the target incidence is the same in both train and test.

**Exercise**

**Exercise**

**Evaluating a model on test and train**

The function auc\_train\_test calculates the AUC of model that is built on a train set and evaluated on a test set:

auc\_train, auc\_test = auc\_train\_test(variables, target, train, test)

with variables a list of the names of the variables that is used in the model.

In this exercise, you will apply this function, and check whether the train and test AUC are similar.

**Instructions**

**100 XP**

* The basetable is loaded. Partition the basetable such that the train set contains 70% of the data, and make sure that train and test set have equal target incidence.
* Calculate the train and test AUC of the model using "age" and "gender\_F" as predictors using the auc\_train\_test function.

[**Take Hint (-30 XP)**](javascript:void(0))

# Load the partitioning module

from sklearn.cross\_validation import train\_test\_split as tts

# Create dataframes with variables and target

X = basetable.drop('target', 1)

y = basetable["target"]

# Carry out 70-30 partititioning with stratification

X\_train, X\_test, y\_train, y\_test = tts(X, y, test\_size = 0.3, stratify = y)

# Create the final train and test basetables

train = pd.concat([X\_train, y\_train], axis=1)

test = pd.concat([X\_test, y\_test], axis=1)

# Apply the auc\_train\_test function

auc\_train, auc\_test = auc\_train\_test(["age", "gender\_F"], ["target"], train, test)

print(round(auc\_train,2))

print(round(auc\_test,2))

In [1]: auc\_train\_test

Out[1]: <function \_\_main\_\_.auc\_train\_test>

<script.py> output:

0.55

0.52

In [2]:

+100 XP

Great work! It could happen that the test AUC is slightly lower than the train AUC. This is a perfectly normal phenomenon called over-fitting.