Reference

Data Camp course.

Crucial base table concepts

Timeline violations

You want to construct a model that predicts which donors are most likely to donate more than 50 Euro in April 2018. To build the predictive model, we reconstruct the timeline one year back in time, so the target period of the basetable is April 2017. Below are the donations of one donor in 2016 and 2017.

Donation ID	Donor ID	Donation date	Donation Amount
1	1	2016-04-01	EUR 40
2	1	2016-09-21	EUR 50
3	1	2016-12-14	EUR 20
4	1	2017-01-31	EUR 50
5	1	2017-04-22	EUR 100
6	1	2017-05-16	EUR 50

Which donations can be used to construct predictive variables in this basetable?

Answer: All donations except donations 5 and 6. Donations 5 and 6 were made during and after the target period and cannot be used as predictive variable.

Available data

- Construct a model that predicts whether someone will donate in a certain year, e.g.2017. This means that the target is based on donations made in 2017, and that the predictive variables are based on donations made before 2017.
- Construct a new pandas dataframe that excludes donations made in 2017 or later.

In [ ]:
```
import pandas as pd
from datetime import datetime

gifts = pd.read_csv("gifts.csv")
print(gifts.head())
gifts["date"] = pd.to_datetime(gifts["date"])


start_target = datetime(year=2017, month=1, day=1)
print(start_target)

gifts_before_2017 = gifts[gifts["date"] < start_target]

print(len(gifts_before_2017))
```

Timeline violation

The example below violates the timeline as it uses information from the target period (2017) to construct the predictive variables.

In [ ]:
```
import pandas as pd

basetable = pd.read_csv("basetable.csv")
print(basetable.head())

X = basetable[["amount_2017"]]
y = basetable[["target"]]

logreg = linear_model.LogisticRegression()
logreg.fit(X, y)

predictions = logreg.predict_proba(X)[:,1]

auc = roc_auc_score(y, predictions)
print(round(auc, 2))
```

Select the relevant population

- Want to construct a predictive model that selects donors that are most likely to donate to a campaign in June 2018. You want to give these donors a call on June 1st 2018. For this specific campaign, you want to work with young donors only, i.e. age 18 to 25 (inclusive).
- To build the model, construct a basetable with a timeline that is exactly one year earlier than the true timeline.

What age should the donors in the population of the basetable have?

Answer: Age 18 to 25 on June 1st 2017

A timeline compliant population

- Construct a basetable for a predictive model that predicts whether donors will donate in 2018.
- The timeline indicates that the population should contain all donors that donated at least once since January 1st 2013, but made no donations after January 1st 2017.
- Construct a set with the donor ids of all donors in the population.

In [ ]:
```
gifts_include = gifts[gifts["date"].dt.year >= 2013]

gifts_exclude = gifts[gifts["date"].dt.year >= 2017]

donors_include = set(gifts_include["id"])

donors_exclude = set(gifts_exclude["id"])

population = donors_include.difference(donors_exclude)
print(len(population))
```

Removing duplicate objects

- Construct a predictive model in order to select donors that are most likely to respond on a letter.
- The population of the basetable should contain donors that have an adress available, and that have privacy settings that allow to send them a letter.

In [ ]:
```
donors_population = donors[(donors["address"] == 1) & (donors["letter_allowed"] == 1)]

population_list = list(donors_population["donor_id"])

population = set(population_list)
print(len(population))
```

Calculate an event target

- Organising a charity event and want to predict which donors are most likely to attend this event.
- You organized a similar event in the past, so you can use that information to construct a predictive model. Given is a list **population** with unique donor **ids** for this basetable and a list **attend_event** with donors in the population that attended this past event.
- Construct a basetable with two columns: the donor_id and the target, which is 1 if the donor attended the event and 0 otherwise.

In [ ]:
```
# Basetable with one column: donor_id
basetable = pd.DataFrame(population, columns=["donor_id"])

basetable["target"] = pd.Series([1 if donor_id in attend_event else 0 for donor_id in basetable["donor_id"]])

print(round(basetable["target"].sum() / len(basetable), 2))
```

Calculate an aggregated target

- Construct a predictive model that predicts which donors are most likely to donate more than 50 euro in a certain month.
- Given is a basetable that already has one row for each donor in the population, the column donor_id represents the donor. The timeline indicates that the target should be 1 if the donor has donated more than 50 euro in January 2017 and 0 else.
- The pandas dataframe gifts_201701 contains all donations in January 2017.
- Add the target column to the basetable.

In [ ]:
```
gifts_summed = gifts_201701.groupby("id")["amount"].sum().reset_index()

targets = list(gifts_summed["id"][gifts_summed["amount"] > 50])

basetable["target"] = pd.Series([1 if donor_id in targets else 0 for donor_id in basetable["donor_id"]])

print(round(basetable["target"].sum() / len(basetable), 2))
```

```
   Unnamed: 0  id       date  amount

140 140 22 2017-01-19 128.0
255 255 40 2017-01-08 123.0
386 386 56 2017-01-28 119.0
395 395 56 2017-01-09 82.0
524 524 77 2017-01-16 145.0

0.04
```

The target incidence is 0.01, meaning that 1% of the donors in the population donates more than 50 Euros in January 2017.

Creating variables

Adding age

- Given is an early stage basetable that contains the donor ID and birth date of candidate donors.
- Add the age of these donors to the basetable. Keep in mind the following timeline of the basetable:

In [ ]:
```
reference_date = datetime.date(2017, 5, 1)

basetable["age"] = pd.Series([calculate_age(date_of_birth, reference_date)
                              for date_of_birth in basetable["date_of_birth"]])

print(round(basetable["age"].mean()))
```

Adding the donor segment

- Add the segment of a donor to the basetable. A selected group of donors that has made many donations in the past is assigned a segment: bronze, silver or gold. Given is an early stage `basetable` and a pandas dataframe `segments` that contains the segments for a selected group of the donors in the basetable.
- Ways for adding a column or multiple coulumns: (1) join (or merge). (2) using dataframe `[column_name]` . (3) pd.concat()

In [ ]:
```
# Add the donor segment to the basetable
basetable = pd.merge(basetable, segments, on=["donor_id"], how="left")

# Count the number of donors in each segment
basetable.groupby("segment").size()

# Count the number of donors with no segment assigned
print(basetable["segment"].isna().sum())
```

4745

You can observe that there are a high number of missing values. You will learn how to deal with them in the next chapter.

Adding living place

- Add the living place of the donors to the basetable. The living places of the donors are given in `living_places` where the living place for each donor is given together with the start and end valid date of that living place.

- May 1st 2017 is the reference date.
- living_places_reference_date should contain rows from living_places where the reference date is between "startdate" and "enddate".

In [ ]:
```
reference_date = datetime.date(2017, 1, 1)

# Select living place reference date
living_places_reference_date = living_places[(living_places["start_date"] <= reference_date) &
                                             (living_places["end_date"] > reference_date)]

# Add living place to the basetable
basetable = pd.merge(basetable, living_places_reference_date[["donor_ID", "living_place"]], on="donor_ID")
```

This exercise shows that even a static variable like living place should be added compliant with the timeline.

Maximum value last year

- Add the maximum amount that a donor donated in 2017, but before May 1st, 2017 to the basetable.
- For each donor in the population, add the maximum amount that this donor donated in 2017 to 'basetable'.

In [ ]:
```
start_date = datetime.date(2017, 1, 1)
end_date = datetime.date(2017, 5, 1)

gifts_2017 = gifts[(gifts["date"] >= start_date) & (gifts["date"] < end_date)]

gifts_2017_bydonor = gifts_2017.groupby(["id"])["amount"].max().reset_index()
gifts_2017_bydonor.columns = ["donor_ID", "max_amount"]

basetable = pd.merge(basetable, gifts_2017_bydonor)
```

Recency of donations

Add the recency, the time since the last donation. Given are two dataframes basetable and gifts, that contain the early stage basetable and the gifts made by donors over time. Add for each donor in the population the recency in days.

In [ ]:
```
# Reference date to calculate the recency
reference_date = datetime.date(2017, 5, 1)

gifts_before_reference = gifts[(gifts["date"] < reference_date)]

last_gift = gifts_before_reference.groupby(["id"])["date"].max().reset_index()
last_gift["recency"] = reference_date - last_gift["date"]

basetable = pd.merge(basetable, last_gift[["id", "recency"]], how="left")
```

```
 id   recency
0          3  396 days
1          5  291 days
2          6  414 days
3          9  536 days
4         13  271 days
5         18  528 days
6         22   66 days
7         23   42 days
The above is part of basetable.
```

Ratio of last month's and last year's average

An interesting variable to add to the basetable is the average gift a donor donated last month compared to the average gift a donor donated last year.

In [ ]:
```
# Average gift last month and year for each donor
average_gift_last_month = gifts_last_month.groupby("id")["amount"].mean().reset_index()
average_gift_last_month.columns = ["donor_ID", "mean_gift_last_month"]

# Average gift last year for each donor
average_gift_last_year = gifts_last_year.groupby("id")["amount"].mean().reset_index()
average_gift_last_year.columns = ["donor_ID", "mean_gift_last_year"]

# Add average gift last month and year to basetable
basetable = pd.merge(basetable, average_gift_last_month, on="donor_ID", how="left")
basetable = pd.merge(basetable, average_gift_last_year, on="donor_ID", how="left")

# Calculate ratio of last month's and last year's average
basetable["ratio_month_year"] = basetable["mean_gift_last_month"] / basetable["mean_gift_last_year"]
print(basetable.head())
```

```
   donor_ID  mean_gift_last_month  mean_gift_last_year  ratio_month_year
0         3            101.666667            87.538462          1.161394
1         5            104.000000            96.166667          1.081456
2         6            156.000000           120.666667          1.292818
3         9            107.000000           104.875000          1.020262
4        13             98.000000            98.000000          1.000000
```

Note that the newly created variable might have missing values if no donations are made last year. You will learn how to deal with this in the next chapter.

Absolute difference between two years

Add the absolute difference in donations made in the last year (2017) and the number of donations made in the year before that (2016).

In [ ]:
```
# Number of gifts in 2016 and 2017 for each donor
gifts_2016_bydonor = gifts_2016.groupby("id").size().reset_index()
gifts_2016_bydonor.columns = ["donor_ID", "donations_2016"]
gifts_2017_bydonor = gifts_2017.groupby("id").size().reset_index()
gifts_2017_bydonor.columns = ["donor_ID", "donations_2017"]

# Add number of gifts in 2016 and 2017 to the basetable
basetable = pd.merge(basetable, gifts_2016_bydonor, on="donor_ID", how="left")
basetable = pd.merge(basetable, gifts_2017_bydonor, on="donor_ID", how="left")

# Calculate the number of gifts in 2017 minus number of gifts in 2016
basetable.fillna(0)
basetable["gifts_2017_min_2016"] = basetable["donations_2017"] - basetable["donations_2016"]
print(basetable.head(50))
```

Performance of evolution variables

- Given is a basetable that has 3 regular predictive variables, namely "gender_F", "age", "donations_2017", and an evolution variable "donations_2017_min_2016" that contains the number of donations made in 2017 minus the number of donations made in 2016.
- Check the added value of using evolution variables. You will construct two predictive models, one using the regular predictive variables given for you in variables_regular and one replacing "donations_2017" by "donations_2017_min_2016", these variables are given for you in variables_evolution. T

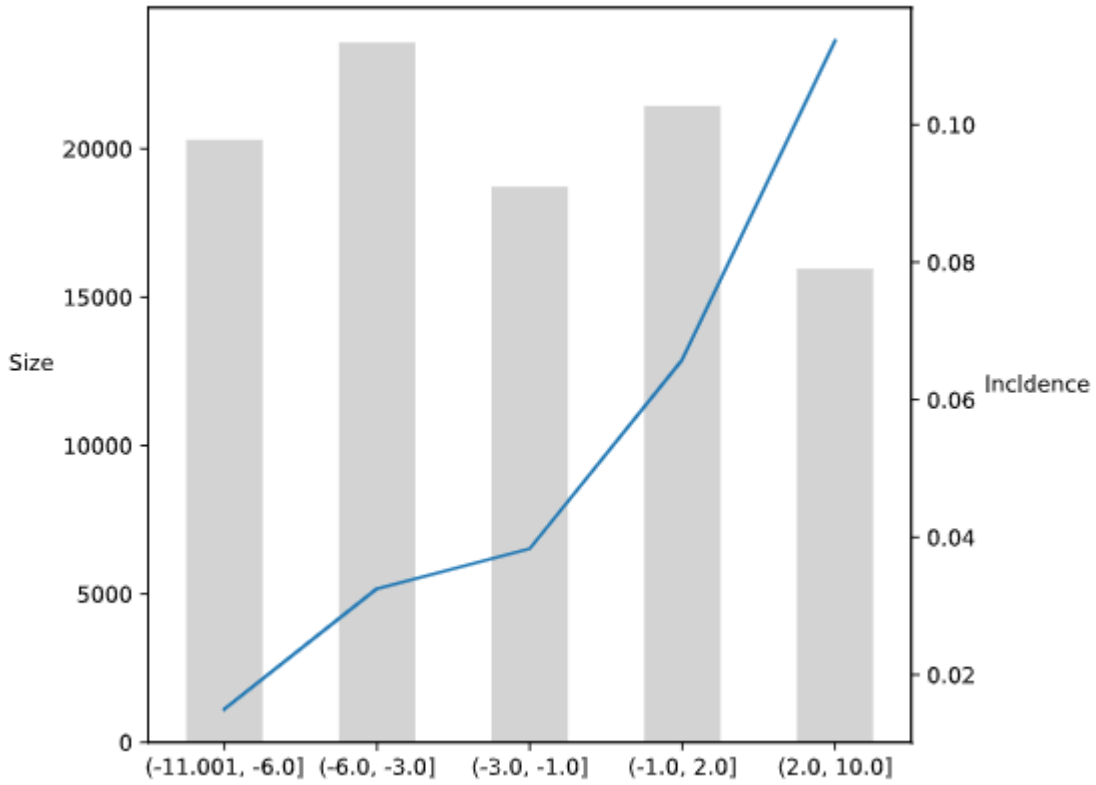
In [ ]:
```
# Select the evolution variables and fit the model
X_evolution = basetable[variables_evolution]
logreg.fit(X_evolution, y)

# Make predictions and calculate the AUC
predictions_evolution = logreg.predict_proba(X_evolution)[:,1]
auc_evolution = roc_auc_score(y, predictions_evolution)

print(round(auc_regular, 2))
print(round(auc_evolution, 2))
```

```
0.6
0.7
```

Meaning of evolution

- Investigate the link between the variable "donations2017min_2016" that you added to the basetable in the previous exercises and the target, using a predictor insight graph.
- The methods to create the predictor insight graph are pre-programmed.
- To plot the predictor insight graph of a continuous variable variable in a basetable, follow these steps:
  - Discretize the variable in n_bins bins:
  - `basetable["variable_disc"]` = pd.qcut(`basetable["variable"]` , n_bins)
  - Construct the predictor insight graph table:
  - pig_table = create_pig_table(basetable, "target","variable_disc")
  - Plot the predictor insight graph based on this table:
  - plot_pig(pig_table,"variable_disc")

In [ ]:
```
# Discretize the variable in 5 bins and add to the basetable
basetable["donations_2017_min_2016_disc"] = pd.qcut(basetable["donations_2017_min_2016"], 5)

# Construct the predictor insight graph table
pig_table = create_pig_table(basetable, "target", "donations_2017_min_2016_disc")

plot_pig(pig_table, "donations_2017_min_2016_disc")
```



Apparently, people that donated more in 2017 are more likely to donate again. **So this evolution variable created is important**. This also suggests that **it is possible to extract much more useful variables from the original**.

Data preparation

Once you derived variables from the raw data, it is time to clean the data and prepare it for modeling.

Creating a dummy from a two-category variable

- Given is a basetable with one predictive variable "gender". Make sure that "gender" can be used as a predictive variable in a logistic regression model by creating dummy variables for it.
- **Need first transfer to categorical type first and then create dummy. Sometimes the variable might be string type but not categorical.**

In [ ]:
```
dummies_gender = pd.get_dummies(basetable["gender"], drop_first=True)

basetable = pd.concat([basetable, dummies_gender], axis=1)

# Delete the original variable from the basetable
del basetable["gender"]
print(basetable.head())
```

```
   donor_id  M
0     32768  0
1     65543  1
2     65546  0
3     32778  1
4     32780  0
```

As gender only has two values and avoiding multicollinearity means that one dummy variable is dropped, only one column is added to the basetable. This is because and F and M are not independent variables.

However, if categorical variable is target, then we don't need to drop, right?

Creating dummies from a many-categories variable

In [ ]:
```
dummies_country = pd.get_dummies(basetable["country"], drop_first=True)

basetable = pd.concat([basetable, dummies_country], axis=1)

# Delete the original variable from the basetable
del basetable["country"]
print(basetable.head())
```

```
   donor_id  India  UK  USA
0     32768      1   0    0
1     65543      0   0    1
```

```
2     65546      1   0    0
3     32778      0   1    0
4     32780      1   0    0
```

Creating a missing value dummy

- Given a basetable that has a predictive variable "total_donations" that has the total number of donations a donor ever made. * This variable can have missing values, indicating that this donor never made a donation before. This is important information on its own, so it is appropriate to create a variable "no_donations" that indicates whether "total_donations" is missing.
- **Not just categorical values can be dummy variables. Missing value can also be**.

In [ ]:
```
# Create dummy indicating missing values
basetable["no_donations"] = pd.Series([1 if b else 0 for b in basetable["total_donations"].isna()])

# Calculate percentage missing values
number_na = sum(basetable["no_donations"] == 1)

print(round(number_na / len(basetable), 2))
```

0.09

There are about 9% donors in the basetable that never donated before.

Replace missing values with the median value

median value usually cannot be affected by outliers. That might be the reason we often heard about median housing price in a region.

In [ ]:
```
median_age = basetable["age"].median()
print(median_age)

# Replace missing values by the median
basetable["age"] = basetable["age"].fillna(median_age)

# Calculate the median of age after replacement
median_age = basetable["age"].median()
print(median_age)
```

```
58.0
58.0
```

The median does not change after replace missing values by the median. This is an indication that replacing missing values by the median will not affect the predictive model a lot.

Replace missing values with a fixed value

In [ ]:
```
replacement = 0

# Replace missing values by the appropriate value
basetable["total_donations"] = basetable["total_donations"].fillna(replacement)
```

Donors that did not make any donation yet, have a total donation amount of 0.

Influence of outliers on predictive models

- There is a variable "maximum donation last year" to the basetable. There is one very rich donor that donated 100 000 Euro last year, while all other donations are between 10 and 150 Euro.

Should this donor be considered as an outlier?

Answer: Yes. This value will drastically influence the predictive model, and as a result the predictive model will not be representative of the majority of the donors.

Handle outliers with winsorization

Given is a basetable with two variables: "sum_donations" and "donor_id". "sum_donations can contain outliers when donors have donated exceptional amounts. Therefore, you want to winsorize this variable such that **the 5% highest amounts are replaced by the upper 5% percentile value**.

In [ ]:
```
from scipy.stats.mstats import winsorize

print(basetable["sum_donations"].min())
print(basetable["sum_donations"].max())

lower_limit = 0.0

# Winsorize the variable sum_donations
basetable["sum_donations_winsorized"] = winsorize(basetable["sum_donations"], limits=[lower_limit, 0.05])

# Check maximum sum of donations after winsorization
print(basetable["sum_donations_winsorized"].max())
```

```
 50
 999
 953
After winsorization, the maximum sum of donations drops from 999 to 953.
```

Handle outliers with standard deviation

In [ ]:
```
print(basetable["age"].max())

# Calculate mean and standard deviation of age
mean_age = basetable["age"].mean()
std_age = basetable["age"].std()

# Calculate the lower and upper limits
lower_limit = mean_age - std_age*3
upper_limit = mean_age + std_age*3

# Add a variable age_no_outliers to the basetable with outliers replaced
basetable["age_mod"] = (pd.Series([min(max(a, lower_limit), upper_limit)
                                   for a in basetable["age"]]))
print(basetable["age_mod"].max())
```

```
140
93.87378815168613
```

The unrealistic value 140 is now replaced by 94. **Aware of the trick of min(max()), though it is not so readable**.

Square root and logarithmic transformations

- This helps to flatten out differences between values.
- An alternative for the logarithmic function is the square root: this function will also flatten out differences between values but the effect is slightly weaker.

In [ ]:
```
basetable["donations_log"] = np.log(basetable["donations"])

basetable["donations_sqrt"] = np.sqrt(basetable["donations"])

print(basetable)
```

```
   donor_id  donations  donations_log  donations_sqrt
0         1        100       4.605170       10.000000
1         2       1100       7.003065       33.166248
```

```
2          3       10000        9.210340       100.000000
3          4       11000        9.305651       104.880885
```

Observe that the differences in the column 'donations_sqrt' are smaller than in the column 'donations_log'. It depends on the situation which one to use, one option could be to add both to the predictive model and let the variable selection algorithm decide which one to use.

Adding interactions to the basetable

In [ ]:
```
print(auc(["age"], basetable))

print(auc(["country_Spain"], basetable))

print(auc(["age", "country_Spain"], basetable))

basetable["spain_age"] = basetable["country_Spain"] * basetable["age"]
basetable["france_age"] = basetable["country_France"] * basetable["age"]

print(auc(["age", "country_Spain", "spain_age", "france_age"], basetable))
```

```
0.5119930184229574
0.5013328819047175
0.5126954569158086
0.5294313945252606
```

In this exercise you can observe that in some cases, adding interactions can improve the predictive power of your model. **Interactions, mainly nonlinearities, can be taken into account by many other models that handle nonlinearity more efficiently. For example, SVM, neural network, decision tree models**.

Advanced base table concepts

Sometimes target or variables change heavily with the seasons.

Detecting seasonality

In [ ]:
```
mean_per_month = gifts.groupby("month")["amount"].mean().reset_index()
print(mean_per_month)

number_per_month = gifts.groupby("month").size().reset_index()
print(number_per_month)

median_per_month = gifts.groupby("month")["amount"].median().reset_index()
print(median_per_month)
```

```
    month     amount
0       1  60.382880
1       2  57.154470
2       3  56.380417
3       4  56.532410
4       5  56.633451
5       6  56.557590
6       7  56.064738
7       8  56.607829
8       9  56.556385
9      10  57.192264
10     11  59.161910
11     12  63.167676
    month      0
0       1  13213
1       2   9743
2       3   8446
3       4   8516
4       5   8239
5       6   8300
6       7   8295
7       8   8532
8       9   8442
9      10   9695
10     11  12019
11     12  16681
    month  amount
```

0	1	61.0
1	2	57.0
2	3	56.0
3	4	57.0
4	5	57.0
5	6	57.0
6	7	56.0
7	8	57.0
8	9	57.0
9	10	57.0
10	11	59.0
11	12	63.0

As you can see, people tend to donate more during the holidays.

The effect of seasonality

HERE Assume you want to predict whether a candidate donor will donate next month. As predictive variable, you want to include the maximum gift of each donor in the previous month. As you learned in the video, the mean amount of the gifts in July and September are similar, but the gifts are fairly higher in December. In this exercise, you will see how this can influence the performance of your model.

The logistic regression model is created and fitted for you in logreg on the data in July.

Instructions 1/3 The test data for July is in test_july. Make predictions for the test data in July and calculate the AUC.

In [ ]:
```
# AUC of model in July:
predictions = logreg.predict_proba(test_july[["age", "max_amount"]])[:,1]
auc = roc_auc_score(test_july["target"], predictions)
print(auc)
```

0.5540804074722869

The test data for September is in test_september. Make predictions for the test data in September and calculate the AUC.

In [ ]:
```
# AUC of model in September:
predictions = logreg.predict_proba(test_september[["age", "max_amount"]])[:,1]
auc = roc_auc_score(test_september["target"], predictions)
print(auc)
```

0.5416164683739391

The test data for December is in test_december. Make predictions for the test data in December and calculate the AUC.

In [ ]:
```
# AUC of model in December:
predictions = logreg.predict_proba(test_december[["age", "max_amount"]])[:,1]
auc = roc_auc_score(test_december["target"], predictions)
print(auc)
```

0.5294167112159428

A model constructed using the data in July, will have about the same test AUC in July as in September, but the AUC will decrease if the model is used in December.

Target values

Assume you want to predict which donors will donate more than 100 Euro next month. In order to obtain a basetable that is large enough, you use two snapshots: one snapshot with target period April 2018 and one with target period May 2018. Given is a donor that made a donation of 130 Euro in April 2018 and 90 Euro in May 2018.

Which statement is correct?

Answer: The donor appears twice in the basetable, once with target value 0 and once with target value 1.

The donor did not donate more than 100 Euro in May, so for the second snapshot the target value will

Calculating snapshot targets

Assume you want to predict who will make large donations, i.e. 500 Euro or more, in January 2019. As January is a very specific month, it is better to only use January snapshots. In order to obtain enough data, you want to include two snapshots, namely January 2018 and January 2017. In this exercise you will learn how to calculate the target values for these different snapshots.

gifts_january_2017 and gifts_january_2018 contain gifts made in January 2017 and January 2018 respectively. Calculate the sum of donations in the target periods of both snapshots for each donor.

In [ ]:
```
# Sum of donations for each donor
gifts_summed_january_2017 = gifts_january_2017.groupby("donor_id")["amount"].sum().reset_index()
gifts_summed_january_2018 = gifts_january_2018.groupby("donor_id")["amount"].sum().reset_index()
```

In [ ]:
```
# Sum of donations for each donor
gifts_summed_january_2017 = gifts_january_2017.groupby("donor_id")["amount"].sum().reset_index()
gifts_summed_january_2018 = gifts_january_2018.groupby("donor_id")["amount"].sum().reset_index()

# List with targets in January 2017
targets_january_2017 = list(gifts_summed_january_2017["donor_id"][gifts_summed_january_2017["amount"] >= 500])
targets_january_2018 = list(gifts_summed_january_2018["donor_id"][gifts_summed_january_2018["amount"] >= 500])
```

In [ ]:
```
# Sum of donations for each donor
gifts_summed_january_2017 = gifts_january_2017.groupby("donor_id")["amount"].sum().reset_index()
gifts_summed_january_2018 = gifts_january_2018.groupby("donor_id")["amount"].sum().reset_index()

# List with targets in January 2017
targets_january_2017 = list(gifts_summed_january_2017["donor_id"][gifts_summed_january_2017["amount"] >= 500])
targets_january_2018 = list(gifts_summed_january_2018["donor_id"][gifts_summed_january_2018["amount"] >= 500])

# Add targets to the basetables
basetable_january_2017["target"] = pd.Series([1 if donor_id in targets_january_2017 else 0 for donor_id in baseta
basetable_january_2018["target"] = pd.Series([1 if donor_id in targets_january_2018 else 0 for donor_id in baseta
```

In [ ]:
```
# Sum of donations for each donor
gifts_summed_january_2017 = gifts_january_2017.groupby("donor_id")["amount"].sum().reset_index()
gifts_summed_january_2018 = gifts_january_2018.groupby("donor_id")["amount"].sum().reset_index()

# List with targets in January 2017
targets_january_2017 = list(gifts_summed_january_2017["donor_id"][gifts_summed_january_2017["amount"] >= 500])
targets_january_2018 = list(gifts_summed_january_2018["donor_id"][gifts_summed_january_2018["amount"] >= 500])

# Add targets to the basetables
basetable_january_2017["target"] = pd.Series([1 if donor_id in targets_january_2017 else 0 for donor_id in baseta
basetable_january_2018["target"] = pd.Series([1 if donor_id in targets_january_2018 else 0 for donor_id in baseta

# Target incidences
print(round(sum(basetable_january_2017["target"]) / len(basetable_january_2017), 2))
print(round(sum(basetable_january_2018["target"]) / len(basetable_january_2018), 2))
```

```
0.01
0.01
```

Observe that the target incidences are similar as they should be.

Calculating aggregated variables

In the previous exercise you constructed two basetables corresponding with two snapshots, and added the target values to these basetables. In this exercise, you will learn how to add aggregated variables. You will add the sum of donations last month (December 2016 and December 2017 for the respective snapshots) to both basetables.

gifts_december_2016 contains all donations made in December 2016. For each donor, calculate the maximum donation made.
gifts_december_2017 contains all donations made in December 2017. For each donor, calculate the maximum donation made.

In [ ]:
```
# Maximum of donations for each donor in december 2016
gifts_max_december_2016 = gifts_december_2016.groupby("donor_id")["amount"].max().reset_index()
gifts_max_december_2016.columns = ["donor_id", "max_amount"]

# Maximum of donations for each donor in december 2017
gifts_max_december_2017 = gifts_december_2017.groupby("donor_id")["amount"].max().reset_index()
gifts_max_december_2017.columns = ["donor_id", "max_amount"]
```

Add the variable "max_amount" to the basetable with target period January 2017 given in basetable_january_2017. Add the variable "max_amount" to the basetable with target period January 2018 given in basetable_january_2018.

In [ ]:
```
# Maximum of donations for each donor in december 2016
gifts_max_december_2016 = gifts_december_2016.groupby("donor_id")["amount"].max().reset_index()
gifts_max_december_2016.columns = ["donor_id", "max_amount"]

# Maximum of donations for each donor in december 2017
gifts_max_december_2017 = gifts_december_2017.groupby("donor_id")["amount"].max().reset_index()
gifts_max_december_2017.columns = ["donor_id", "max_amount"]

# Add max_amount to the basetables
basetable_january_2017 = pd.merge(basetable_january_2017, gifts_max_december_2016, on="donor_id", how="left")
basetable_january_2018 = pd.merge(basetable_january_2018, gifts_max_december_2017, on="donor_id", how="left")

# Show the basetables
print(basetable_january_2017.head())
print(basetable_january_2018.head())
```

```
   donor_id  target  max_amount
0     69050       0         NaN
1     80836       0         NaN
2     80069       0         NaN
3     79955       0         NaN
4     15184       0         NaN
   donor_id  target  max_amount
0      1992       0         NaN
1       241       0         NaN
2     56731       0         NaN
```

```
3     50955       0         NaN
4     65353       0         NaN
```

You added one variable to the basetables. Similarly, you can add various other variables like minimum donation, mean donation, sum of donations and so on.

Stacking basetables

In the previous exercise you constructed two basetables basetable_january_2017 and basetable_january_2018 that correspond to two snapshots with target periods January 2017 and January 2018, respectively. In order to have enough data, another snapshot is added, resulting in basetable_january_2016. In this exercise, you will learn to construct the final basetable by stacking the basetables, and verify that there are enough observations in it.

Instructions 100 XP Construct a basetable by adding basetable_january_2017 to basetable_january_2016. Add basetable_january_2018 to basetable. Print the number of observations in the final basetable.

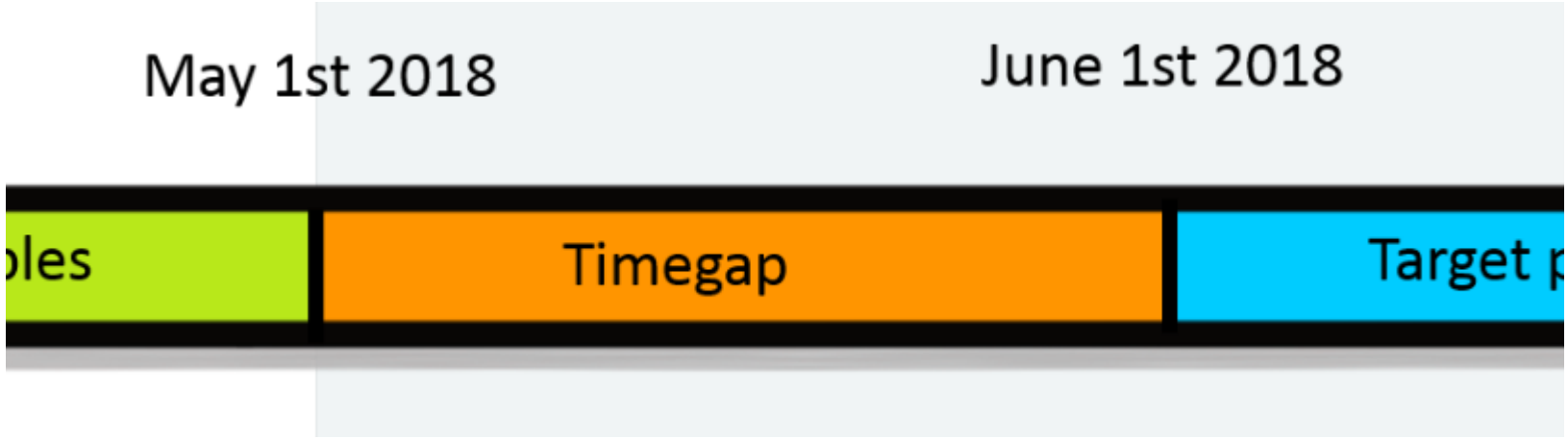
In [ ]:
```
# Add basetable_january_2017 to basetable_january_2016
basetable = basetable_january_2016.append(basetable_january_2017)

# Add basetable_january_2018 to basetable
basetable = basetable.append(basetable_january_2018)

# Number of observations in the final basetable
print(len(basetable))
```

32572

Using 3 snapshots, you obtain a basetable with about 20 000 observations.

Events during the timegap

Assume you want to construct a predictive model to predict who will make a donation of at least 10 Euro in June 2019. The timeline reconstructed in history is as follows, the timegap is at May 2018. A donor made a donation on May 2nd 2018.

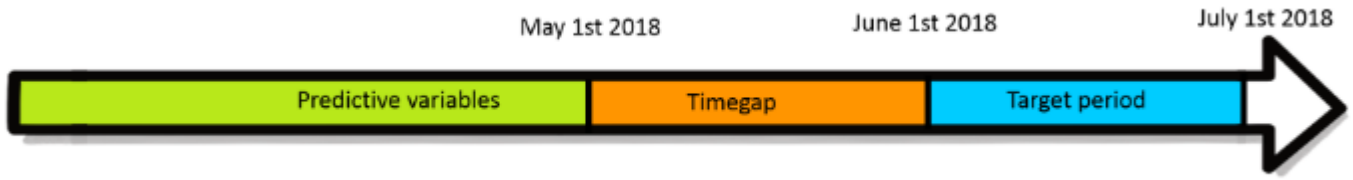


To which goal can this donation be used?
Hint: The donation is made during the timegap!

Answer: This donation cannot be used with this timeline.

Calculating aggregated variables with timegap

Assume you want to construct a predictive model to predict which donors will make a donation of at least 10 Euro during the next month. The timeline is given below, there is a timegap of one month. In this exercise, you will learn how to add a variable "mean_donation_last_month" to the basetable.



Fill out the start (April 1st 2018) and end date (May 1st 2018) of the period on which the predictive variable "mean_donation_last_month" is calculated.

In [ ]:
```
# Start and end date of last month
start_date = datetime.date(2018, 4, 1)
end_date = datetime.date(2018, 5, 1)
```

All donations ever made are in gifts. Assign gifts made during that period to gifts_last_month.

In [ ]:
```
# Start and end date of last month
start_date = datetime.date(2018, 4, 1)
end_date = datetime.date(2018, 5, 1)

# Gifts made last month
gifts_last_month = gifts[(gifts["date"] >= start_date) & (gifts["date"] < end_date)]
```

Calculate the mean gift made last month for each donor in gifts_last_month.

In [ ]:
```
# Start and end date of last month
start_date = datetime.date(2018, 4, 1)
end_date = datetime.date(2018, 5, 1)

# Gifts made last month
gifts_last_month = gifts[(gifts["date"] >= start_date) & (gifts["date"] < end_date)]

# Mean gift made last month
gifts_last_month_mean = gifts_last_month.groupby("donor_id")["amount"].mean().reset_index()
gifts_last_month_mean.columns = ["donor_id", "mean_donation_last_month"]
```

Add the variable "mean_donation_last_month" to the basetable that already contains "donor_id" and "target".

In [ ]:
```
# Start and end date of last month
start_date = datetime.date(2018, 4, 1)
end_date = datetime.date(2018, 5, 1)

# Gifts made last month
gifts_last_month = gifts[(gifts["date"] >= start_date) & (gifts["date"] < end_date)]

# Mean gift made last month
gifts_last_month_mean = gifts_last_month.groupby("donor_id")["amount"].mean().reset_index()
gifts_last_month_mean.columns = ["donor_id", "mean_donation_last_month"]

# Add mean_donation_last_month to the basetable
basetable = pd.merge(basetable, gifts_last_month_mean, on="donor_id", how="left")
print(basetable.head(10))
```

```
   donor_id  target  mean_donation_last_month
0     48149       0                       NaN
1     72234       1                       NaN
2       392       1                       NaN
3     39614       1                       NaN
4     66454       1                       NaN
5     74778       0                     673.0
6     79767       1                       NaN
7     34717       1                       NaN
8     21008       0                       NaN
9     25354       0                       NaN
```

As you can see, many of the donors did not make a donation last month. Next step would be to fill out the missing values with 0 for these donors.

Adding age with timegap

The basetable of previous exercise is given. The timeline is given below, there is a timegap of one month. In this exercise, you will learn how to add the age of the donors to the basetable, compliant with the timeline.

Instructions 100 XP Fill out the reference date on which the age should be calculated, this is the start date of the timegap. Add a column "age" to the basetable that is the age of the donor on the reference date. The function calculate_age has been implemented for you. It takes date_of_birth and reference_date as arguments. Print the mean age of all donors.

In [ ]:
```
# Reference date
reference_date = datetime.date(2018,5,1)

# Add age to the basetable
basetable["age"] = pd.Series([calculate_age(date_of_birth, reference_date)
                              for date_of_birth in basetable["date_of_birth"]])

# Calculate mean age
print(round(basetable["age"].mean()))
```

63.0

When calculating the age, the reference date should indeed be the first day of the timegap.