# ASSIGNMENT 1 – STOCKHOLM BANK OF ECONOMICS

Swedish banks are required by regulations to ensure that any mortgage provided for a house or an apartment is financially feasible. Customers must have sufficient financial resources to support the interest rate and amortizations while ensuring that they have enough money left to meet their daily needs. This becomes even more crucial when dealing with non-fixed interest rates, where interest rates may increase by several percentage points over time.

However, there are situations where statistical modeling is not possible or desirable. In cases where the data does not have labels, either labels need to be created, or some other alternative to classical supervised learning must be attempted. In situations where regulations require specific definitions or calculations, a statistical approach may not only be unnecessary but also detrimental.

In this assignment, we will examine a mortgage granting process for a Swedish bank. Although this example has been slightly simplified to fit within the course's scope, it still differs only marginally from what is currently being used in practice by various financial institutions.

Even if historical default labels were available, they may present a distorted picture, where the recent close-to-no default rates say little about the portfolio during a major recession. Furthermore, regulators demand clear and explainable rules for any individual decision. This makes the problem more suitable for a classical rule-based system than a black-box model with little mechanistic understanding.

The purpose of this assignment is to implement a rule-based system in practice while getting used to writing and tinkering with code in Python. It will consist of a few functions, which can be independently used and tested.

As we are in the domain of software engineering, the requirements may appear a bit on the fuzzy side. The people on the business side of the bank aren't as tech savvy as you, and they're doing the best they can.

*Legal Asterix; All data used in this assignment should be considered fictitious and any resemblances of personal data to real people are purely coincidental.*

## PRELUDE

At the end of this assignment, you are required to submit one single Python file (.py). If you have developed your code in a notebook file (.ipynb), you need to convert it into a .py file before submission. You can ask the instructors or Google for help with the conversion process. Using a notebook to develop your code is convenient, but make sure to convert it into a .py file before the deadline.

You can work in teams of two, but each individual student must submit a copy of the assignment on Canvas.

Grading will be based on the quality of your code and how well it solves the problem. Your code is expected to be well-structured and readable. Use meaningful names for variables and functions and add comments to make your code easier to work with.

You can restructure or rename any section of the assignment as long as it makes sense, and the output remains the same.

The customer and loan application data are available in a MySQL database. You can complete the first four steps of the assignment without downloading the customer data. However, in the later steps, you will need the data to classify all the customers. Since the loan applications do not have labels, it is challenging to use them for testing purposes.

You can connect to MySQL using the credentials provided. You can find the code for loading data from SQL in the appendix.

## STEP 1 – AMORTIZATION RATE:

Swedish regulation only mandates amortization under certain circumstances.

*Households taking new mortgages with a loan-to-value ratio over 70 percent must amortise two percent of their loan every year, while households with loan-to-value ratios between 50 and 70 percent must amortize one percent. Households that have mortgages that are more than 4.5 times larger than their total income before tax need to amortise an additional one percent a year.*

- The Swedish FSA (Finansinspektionen)

Create a function that accepts the following arguments:

property_valuation, loan_size, gross_yearly_income

Where property valuation is the market value of the real estate in question (usually the selling price).

The function should return a number indicating the amortization rate, e.g., 0, 0.01, 0.02, or 0.03.

## STEP 2 – CHILDREN.

Children can be assumed to cost 3700 SEK per month to support. Child support is provided by the government every month. The 2023 years numbers can be found in the table below.

| No. Of Children | Monthly Child Support |
|---|---|
| 1 | 1250 |
| 2 | 2650 |
| 3 | 4480 |
| 4 | 6740 |
| 5 | 9240 |
| 6 | 11740 |

For any additional child after, 1250 extra is given.

Write a function, that given a number of children, returns the total anticipated cost per year.

## Step 3 – Calculate Taxes

To simplify the process of calculating income tax for customers in the Stockholm region, a function is required that takes the gross yearly income and municipality as input parameters. The calculation of income tax involves two taxes: municipality tax [kommunalskatt] and state income tax [statlig inkomstskatt]. The rates for municipality tax vary slightly across different municipalities within the Stockholm region and can be found in the TaxRates table of the Kalp database as well as in the table below. State income tax is an additional 20 percent tax on income earned above 554,900 SEK per year. The output of the function should be the amount of income tax to be paid, rounded to the nearest integer.

| Municipality | Income Tax |
|---|---|
| **Botkyrka** | 32.23 |
| **Danderyd** | 30.43 |
| **Ekerö** | 31.03 |
| **Haninge** | 31.76 |
| **Huddinge** | 31.55 |
| **Järfälla** | 31.07 |
| **Lidingö** | 29.92 |
| **Nacka** | 30.06 |
| **Norrtälje** | 31.8 |
| **Nykvarn** | 32.05 |
| **Nynäshamn** | 31.93 |
| **Salem** | 31.5 |
| **Sigtuna** | 32.08 |
| **Sollentuna** | 30.2 |
| **Solna** | 29.45 |
| **Stockholm** | 29.82 |
| **Sundbyberg** | 31.33 |
| **Södertälje** | 32.23 |
| **Tyresö** | 31.58 |
| **Täby** | 29.63 |
| **Upplands Väsby** | 31.5 |
| **Upplands-Bro** | 31.48 |
| **Vallentuna** | 30.98 |
| **Vaxholm** | 31.38 |
| **Värmdö** | 31.06 |
| **Österåker** | 28.98 |

## Step 4 – Calculating disposable income

To calculate the total disposable income of a customer after all expenses, we need to write a function. This function will use the previously built functions to calculate the total amount paid in taxes, amortization rate, and the cost for children. The cost of living is assumed to be 9700 per month for an individual. Additionally, customers should be able to survive an interest rate of 6.5% on the loan granted; hence, the cost of the potential loan must also be factored into the disposable income calculation.

The purpose of the function is to return the disposable income rounded to the nearest integer. To avoid inconvenience, the function should take a single input parameter in the form of a dictionary, instead of having several parameters. This dictionary will contain all the necessary values associated with the customer.

For a customer who wants to finance a home, the cost of living in an apartment is assumed to be 4,000 per month, while the cost of living in a house is assumed to be 4,500 per month. The "housing_type" variable on the customer dictionary holds either the value "apartment" or "house" to indicate the home type they are attempting to finance.

Example input of a customer created as a dictionary

```
customer = {

    "num_children": 2, # number of children

    "downpayment": 1800000, # Amount customer has in cash

    "municipality": "Sundbyberg",

    "gross_yearly_income": 384000, #Total salary before tax

    "housing_type": "apartment", # This can be "house" or "apartment"

    "requested_loan": 4400000, # How much the customer wants to borrow

    "property_valuation": 6200000 # total apartment value

}
```

## Step 5 – Plugging in the customers

We should consider testing our current progress on our customer base. The appendix includes some code examples on how to execute a SQL query and save the results in a Pandas data frame. Since the Customer table is small, we can run the entire table without worrying about memory constraints on our local machine.

At this first stage, let us download the customer information and create a loop (lambda expressions or similar is also an option) that calculates the disposable income for each customer. At this stage, it's enough to print the output.

## Step 6 – Costs of servicing existing loans

It is important to consider that customers may have pre-existing loans such as car loans, credit cards, or similar. The cost of servicing these loans should be factored into the calculation, which can be simplified as the loan amount multiplied by the interest rate. Since customers may have multiple loans, and each loan may have different amounts and interest rates, the existing loans are stored in a separate table in the database called "CustomerLoan". As there is a "one to many" relationship between customers and their loans, an aggregation needs to be performed in either SQL or Python to calculate the total cost of all the loans for each customer.

Modify the existing code to include the total cost of existing loans for each customer when calculating their disposable income.

## Step 7 – So how big can the mortgage be?

What is the maximum amount of mortgage that we can offer to a customer? Answering this question is not as easy as it may seem, given that the loan's size affects the disposable income, which in turn affects the loan's size. This makes solving it using classical calculus a non-trivial task. Instead of trying to solve it algebraically, we will take a brute-force iterative approach.

In this step, we will develop a new function that takes the same customer input as the disposable income function. This function will use a loop to iterate over possible loan values, while utilizing the disposable income function from the previous step. Customers aren't allowed to have negative disposable income, so that's one hard limit we'd have to base our decisions on. Finally, the loan can't exceed 85% of the property value. The bank cannot grant negative loans either, for obvious reasons. Hence, if the customer has negative disposable income even without a loan, the maximum loan that the bank can offer should be 0.

We aim to calculate the maximum mortgage that the bank can grant for each customer. A margin of +/- 1000 SEK is sufficient, considering that mortgages are typically large. Round any output to the nearest integer.

## STEP 8 – STORING THE OUTPUT

To create a rule-based system that can integrate with the bank's infrastructure, we need to write the data back so as to be readable by other programs within the bank. Although we could write the results back to the database, a simpler option would be to just create a JSON file to the current directory.

In the appendix, some additional code stubs can be found showing how to generate a JSON file from a Pandas data frame. It should contain a list of dictionaries where each dictionary represents a customer, and the dictionary should have the following keys:

- customer_id  # This is the ID of the customer as seen in the database
- answer_amortization #This is the output from our amortization function e.g. 0.2
- answer_total_child_cost # The integer containing the output from step 2
- answer_taxes # An integer from step 3, containing the total taxes for said customer
- answer_existing_loans_cost # An integer of the total loan cost for this customer
- answer_disposable_income # Integer with customers' disposable income at the requested loan
- answer_max_loan # Total amount the bank would be willing to lend to this customer

Name the file assignment1_<your-registration.number.json> e.g., assignment1_25124.json, and save the file in the folder you are currently running (see appendix for example).

## STEP 9 – SUBMITTING THE WORK

We would like to have the Python code as well as the JSON output. Save your code file as a .py and name the file assignment1_<enrollment-number>.py e.g., assignment1_25123.py before uploading it together with the JSON file to Canvas.

If you work in teams of two, please add a comment somewhere in the beginning with your code, your enrollment number, and your partner's enrollment number. Please do not add names, to not influence the grading process by subconscious biases.

# Appendix

## Installing drivers for MySql

Anaconda comes preinstalled with many packages allowing you to handle relational databases in general. As they come in many different flavors, individual drivers for each database need to be installed separately. In this course, we will be using MySql.

The recommended drivers for PyMySql can be installed either using conda or pip depending on preference.

*conda install pymysql*

or

pip install pymysql

## Connecting to MySql from Python

The generic relational database driver in Anaconda is called SqlAlchemy, which will do all the heavy lifting for us. The following code sets up a connection to MySql, runs a SQL query, and downloads the dataset into a data frame

```
host = '
username = '
password = '
schema =

# format replaces {} in the string with the respective inputs to the function
connection_string = "mysql+pymysql://{}:{}@{}/{}".format(username, password, host,
schema)
# this creates a connection from the string we just created
connection = create_engine(connection_string)

# TODO HERE YOU SHOULD PUT YOUR OWN SQL QUERY
query = """
SELECT * FROM ...
"""
# use the connection to run SQL query and store it into a dataframe
df = pd.read_sql_query(con=connection.connect(), sql=text(query))

df.drop(columns=['firstname', "lastname"], inplace=True)
```

## DEALING WITH JSON

JSON is a computer and human-readable format for storing data. It is in many ways more convenient than CSV, which in practice is quite ill-defined. What happens if a value contains a comma? There are of course ways around this, but it has over the years caused a lot of confusion.

### USING PANDAS

Dumping a Pandas data frame into JSON in the current working directory is done using this one-liner.

```
# Write df to a JSON file

df.to_json("assignment1_25123.json", orient="records", indent=2)
```

Where “assignment1_25123.json” is the name of the file you wish to create

Orient=”records” keeps the format centered around rows as opposed to columns

Indent is how many spaces of indentation every new line should get. This makes it more pleasant to read but has no impact on the information stored.

The content of the file will then be a list of dictionaries. Every dictionary (denoted by { }) represents one customer.

An example of the file should then look something like this:


```
[
  {
    "customer_id":"0008cce4-5e52-4eac-a3af-04535574",
    "answer_disposable_income":-265834,
    "answer_total_child_cost":154920,
    "answer_amortization":0.01,
    "answer_max_loan":5166459,
    "answer_taxes":478341,
    "answer_existing_loans_cost": 472
  },
  {
    "customer_id":"003c0ebc-edbd-4289-a2c0-b6775d00",
    "answer_disposable_income":-437543,
    "answer_total_child_cost":57000,
    "answer_amortization":0.03,
…
```