group1-banker-doc

October 7, 2020

1 Model development

1.1 Policy

We assume that $A = \{a_0, a_1\}$, we further define a_0 as "deny credit" and a_1 as "offer credit". We also know from the project text that the utility function is assumed to be linear. This implies that among $E(utility|a_i)$ and $E(utility|a_j)$, we will always choose action a_i over action a_j if $E(utility|a_i) > E(utility|a_j)$ adapted from (Dimitrakakis, 2020, p. 48). We will "blindly" choose the action that maximizes expected utility.

1.2 Utility function

There are two possible actions that we could perform: $\{a_0, a_1\}$. We also have the following rewards, which are defined as possible outcomes for the bank (Dimitrakakis, 2020, p. 47). In our case these rewards are $\{-m, 0, m((1+r)^n - 1)\}$. Further there are two possible actions for the bank when it comes to deciding for each new customer if they will be granted credit.

Further, we will use the definition of expected utility

$$E[U|a_t = a] = \sum_r U(r) Pr(r|a_t = a)$$

adapted from (Dimitrakakis, 2020, p. 48). This will be used for each action to calculate its expected utility. We can further define the rewards as r_0 = "the debtor defaults" and r_1 = "the debtor does not default". If we use the assumption that the utility function is linear, we can say that $U(r)$ is proportional to r.

1.2.1 Grant credit

If we decide to grant credit ($a_{t} = 1 $) we have the following expected utility considering the rewards above:

$$E[U|a_t = 1] = U(r = r_0) Pr(r = r_0|a_t = 1) + U(r = r_1) Pr(r = r_1|a_t = 1)$$

which becomes

$$E[U|a_t = 1] = -m \cdot Pr(r = r_0|a_t = 1) + m((1+r)^n - 1) \cdot Pr(r = r_1|a_t = 1)$$

In our code, we have defined $Pr(r = r_1|a_t = 1)$ as the variable "p_c" and $Pr(r = r_0|a_t = 1)$ as 1-"p_c".

1.2.2 Do not grant credit

If we decide not to grant credit ($a_{t} = 0$).

$$E[U|a_t = 0] = 0 \cdot Pr(r = r_0|a_t = 0) + 0 \cdot Pr(r = r_1|a_t = 0) = 0$$

1.3 Expected utility

If we do not give out the loan, the expected utility is 0, as there is nothing to gain or loose. If we give a loan there are two possible outcomes; either they are able to pay it back with interest, or we loose the investment m. We can therefor write the expected utility if we give a loan as:

$$E(U(x)|a = a_{\text{loan}}) = m[(1 + r)^n - 1]p(x_1) - mp(x_2),$$

where $p(x_1)$ is the probability of paying back the loan with interest and $p(x_2)$ is the probability of loosing the interest.

[1]:
```
def expected_utility(self, x, action):
    """Calculate expected utility using the decision maker model.

    Args:
        x: A new observation.
        action: Whether or not to grant the loan.

    Returns:
        The expected utility of the decision maker.
    """
    if action == 0:
        return 0

    r = self.rate
    p_c = self.predict_proba(x)

    # duration in months
    n = x['duration']
    # amount
    m = x['amount']

    e_x = p_c * m * ((1 + r) ** n - 1) + (1 - p_c) * (-m)
    return e_x
```

1.4 Fitting a model

We chose to use a logistic regression model. It predicts the probability of a binary categorical variable beeing 1. A fresh random state is also given to the model for reproducable results.

[2]:
```
def fit(self, X, y):
    """Fits a logistic regression model.
```

```
    Args:
        X: The covariates of the data set.
        y: The response variable from the data set.

    Notes:
        Using logistic regression, adapted from
        https://scikit-learn.org/stable/modules/generated/sklearn.
        linear_model.LogisticRegression.html
    """
    self.data = [X, y]

    log_reg_object = LogisticRegression(random_state=1, max_iter=2000)
    self.model = log_reg_object.fit(X, y)

def predict_proba(self, x):
    """Predicts the probability for [0,1] given a new observation given the
    model.

    Args:
        x: A new, independent observation.
    Returns:
        The prediction for class 1 given as the second element in the
        probability array returned from the model.
    """
    x = self._reshape(x)
    return self.model.predict_proba(x)[0][1]

def _reshape(self, x):
    """Reshapes Pandas Seris to a row vector.

    Args:
        x: Pandas Series.

    Returns:
        A ndarray as a row vector.
    """
    return x.values.reshape((1, len(x)))
```

When reading the data we one hot encode all the catagorical variables which means that they loose the information in the order. This could be fixed by instead giving them an integer value, but then we assume a linear relationship between the order of the categories.

1.5 Best action

The best action is the action that gives the highest utility. In the event of the utilities beeing equal, we chose to not give a loan. Because of the linear utility of the investor it does not matter what we do in this situation, but we figured it is better to not accept unnecessary variability.

[3]:
```
def get_best_action(self, x):
    """Gets the best action defined as the action that maximizes utility.

    Args:
        x: A new observation.
    Returns:
        Best action based on maximizing utility.
    """
    expected_utility_give_loan = self.expected_utility(x, 1)
    expected_utility_no_loan = self.expected_utility(x, 0)

    if expected_utility_give_loan > expected_utility_no_loan:
        return 1
    else:
        return 0
```

2 Testing the model against random model

[4]:
```
%matplotlib inline

import random_banker
import group1_banker
from sklearn.model_selection import train_test_split
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

np.random.seed(24092020)

features = ['checking account balance', 'duration', 'credit history',
            'purpose', 'amount', 'savings', 'employment', 'installment',
            'marital status', 'other debtors', 'residence time',
            'property', 'age', 'other installments', 'housing', 'credits',
            'job', 'persons', 'phone', 'foreign', 'repaid']

data_raw = pd.read_csv("german.data",
                       delim_whitespace=True,
                       names=features)
data_raw.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 21 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
```

```
 0   checking account balance  1000 non-null   object
 1   duration                  1000 non-null   int64
 2   credit history            1000 non-null   object
 3   purpose                   1000 non-null   object
 4   amount                    1000 non-null   int64
 5   savings                   1000 non-null   object
 6   employment                1000 non-null   object
 7   installment               1000 non-null   int64
 8   marital status            1000 non-null   object
 9   other debtors             1000 non-null   object
 10  residence time            1000 non-null   int64
 11  property                  1000 non-null   object
 12  age                       1000 non-null   int64
 13  other installments        1000 non-null   object
 14  housing                   1000 non-null   object
 15  credits                   1000 non-null   int64
 16  job                       1000 non-null   object
 17  persons                   1000 non-null   int64
 18  phone                     1000 non-null   object
 19  foreign                   1000 non-null   object
 20  repaid                    1000 non-null   int64
dtypes: int64(8), object(13)
memory usage: 164.2+ KB
```

2.1 Transforming the data

[5]:
```
numeric_variables = ['duration', 'age', 'residence time', 'installment',
             'amount', 'persons', 'credits']
data = data_raw[numeric_variables]

# Mapping the response to 0 and 1
data["repaid"] = data_raw["repaid"].map({1:1, 2:0})
```

```
<ipython-input-5-ad017a9a2f47>:6: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  data["repaid"] = data_raw["repaid"].map({1:1, 2:0})
```

[6]:
```
# Create dummy variables for all the catagorical variables
not_dummy_names = numeric_variables + ["repaid"]
dummy_names = [x not in not_dummy_names for x in features]
dummies = pd.get_dummies(data_raw.iloc[:,dummy_names], drop_first=True)
data = data.join(dummies)
```

2.2 Testing decision makers

[7]:
```
def utility_from_obs(predicted_decision, true_decision, amount, duration,␣
↪interest_rate):
    """Calculates utility from a single observation.

    Args:
        predicted_decision: the model's best action
        true_decision: if the observation repaid or not
        amount: the lending amount
        duration: the number of periods
        interest_rate: the interest rate of the loan

    Returns:
        The utility from the single observation given our action.
    """
    if predicted_decision == 1:
        if true_decision == 1:
            return amount*((1 + interest_rate)**duration - 1)
        else:
            return -amount
    else:
        return 0
```

[8]:
```
def utility_from_test_set(X, y, decision_maker, interest_rate):
    """Calculates total utility from a given test set.

    Args:
        X: the covariates of the test set
        y: the response variable of the test set
        decision_maker: the decision maker to use in order to calculate utility
        interest_rate: the interest rate to use when calculating utility

    Returns:
        The sum of utility from the test set and the sum of utility divided by
        total amount.
    """

    num_obs = len(X)
    obs_utility = np.zeros(num_obs)
    obs_amount = np.zeros_like(obs_utility)

    for new_obs in range(num_obs):
        predicted_decision = decision_maker.get_best_action(X.iloc[new_obs])
        true_decision = y.iloc[new_obs]

        amount = X['amount'].iloc[new_obs]
```

```
        duration = X['duration'].iloc[new_obs]

        obs_utility[new_obs] = utility_from_obs(
            predicted_decision, true_decision, amount, duration, interest_rate)
        obs_amount[new_obs] = amount

    return np.sum(obs_utility), np.sum(obs_utility)/np.sum(obs_amount)
```

[9]:
```
def compare_decision_makers(num_of_tests, response, interest_rate):
    """Tests the random banker against our group1 banker.

    Args:
        num_of_tests: the number of tests to run
        response: the name of the response variable
        interest_rate: the interest rate to use when calculating utility
    """
    bank_utility_random = np.zeros(num_of_tests)
    bank_investment_random = np.zeros_like(bank_utility_random)
    bank_utility_group1 = np.zeros(num_of_tests)
    bank_investment_group1 = np.zeros_like(bank_utility_group1)

    results = {}

    # decision makers
    r_banker = random_banker.RandomBanker()
    r_banker.set_interest_rate(interest_rate)
    n_banker = group1_banker.Group1Banker()
    n_banker.set_interest_rate(interest_rate)

    # get data
    X = data
    covariates = X.columns[X.columns != response]

    for i in range(num_of_tests):
        X_train, X_test, y_train, y_test = train_test_split(
            X[covariates], X[response], test_size=0.2)

        # fit models
        r_banker.fit(X_train, y_train)
        n_banker.fit(X_train, y_train)

        bank_utility_random[i], bank_investment_random[i] =␣
↪utility_from_test_set(
            X_test, y_test, r_banker, interest_rate)
        bank_utility_group1[i], bank_investment_group1[i] =␣
↪utility_from_test_set(
            X_test, y_test, n_banker, interest_rate)
```

```
    results["utility_random_banker"] = bank_utility_random
    results["investment_random_banker"] = bank_investment_random
    results["utility_group1_banker"] = bank_utility_group1
    results["investment_group1_banker"] = bank_investment_group1

    return results
```

[10]:
```
%%time
response = 'repaid'
results = pd.DataFrame(compare_decision_makers(100, response, 0.05))
```

```
CPU times: user 42.8 s, sys: 322 ms, total: 43.2 s
Wall time: 22.7 s
```

[11]:
```
results.describe()
```

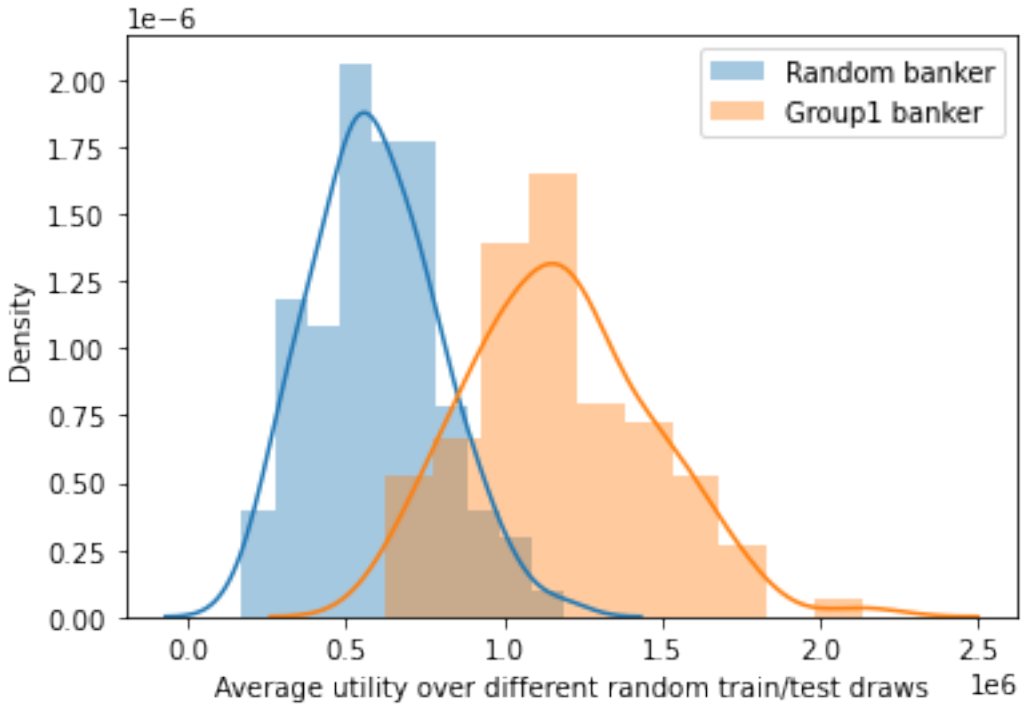
[11]:

	utility_random_banker	investment_random_banker	utility_group1_banker	\
count	1.000000e+02	100.000000	1.000000e+02	
mean	5.922753e+05	0.913618	1.180550e+06	
std	1.995408e+05	0.296520	2.872515e+05	
min	1.715205e+05	0.270745	6.235157e+05	
25%	4.595768e+05	0.692373	9.772184e+05	
50%	5.880903e+05	0.913637	1.181998e+06	
75%	7.179540e+05	1.119127	1.364043e+06	
max	1.191101e+06	1.889368	2.137712e+06	

	investment_group1_banker
count	100.000000
mean	1.825611
std	0.423820
min	0.971803
25%	1.476474
50%	1.817679
75%	2.091771
max	3.053918

[12]:
```
sns.distplot(results["utility_random_banker"], label="Random banker")
sns.distplot(results["utility_group1_banker"], label="Group1 banker")
plt.legend()
plt.xlabel("Average utility over different random train/test draws")
plt.ylabel("Density")
```

[12]:
```
Text(0, 0.5, 'Density')
```



2.3 Results

Based on 100 random train/test splits, our model using logistic regression is considerably better than the random model.

3 Questions

3.0.1 Is it possible to ensure that the policy maximises revenue?

We cannot ensure with 100 % certainty that the policy maximises revenue. This is due to the fact that the model is not perfect and we are maximising the expected utility. There is an uncertainty in the estimated probability of a new individual being credit-worthy. If there were no uncertainty in this and we knew in advance which of the new individuals were going to pay back, we could ensure that a policy would maximise revenue by only lending to the credit-worthy individuals.

3.0.2 How can we take into account the uncertainty due to limited/biased data?

In an ideal world we would have an independent replication of our study and check whether or not new, independent data would develop the same type of policy that we obtained with the current data set. Another option is simulation of data in order to validate our policy based on constructed data (Dimitrakakis, 2020, pp. 35-36). When it comes to the current policy, there are several ways of taking this uncertainty into account when considering our objective of maximising expected utility.

3.0.3 What are the consequences if the model is wrong?

There will always be a possibility that the model performs a lot worse in practice. For example our data set could be a bad representation of the population because of some change in the population properties after it was collected. Therefor it is important not to put too much trust into new models. Simple interpretable models have a clear advantage in this regard. If the model is simpel enough you might spot errors before the model is tested in practice, and if you find a model to be wrong, it will be easier to understand why.

- potential loss of income due to hard-to-interpret-model, customers may go elsewhere and the competition may gain
- better to give credit correclty to most customers while letting some edge cases be rejected?

3.0.4 How can we take into account the risk of the model being wrong?

It is possible to consider the two types of error the model can generate:

1. classify new individuals as credit-worthy when they are in fact not (a_{10})
2. classify new individuals as not being credit-worthy when they in fact are (a_{01})

We can indicate the class of actual credit-worthy individuals as 'positive' (a_1) and the class of individuals not being credit-worthy as 'negatives' (a_0). Then our error (1) can be called a false positive and the other type of error (2) as a false negative. This corresponds to type 1 and type 2 erros, where the probability of type 1 errors is equivalent to the probability of false positive and type 2 error is equivalent to the probability of false negative. These probabilities can be estimated by looking at the fraction $\frac{a_{10}}{\#a_1}$. Similarily for the false negatives $\frac{a_{01}}{\#a_0}$ (Azzalini & Scarpa, 2012, p. 139). For our example, if a new individual is classified as credit-worthy and this is a false positive, it implies the loss $-m$ (the lost investment). While if a new individual is classified as not being credit-worthy and this is a false negative, it implies the loss of $-m[(1+r)^n - 1]$ (the lost return on investment).

- type 1 vs type 2 errors
- conservative models
- we already use a sort of cost matrix by deciding the credit dependent on the expected utility

		True	
		No default	Default
Predicted	No default	$-m((1+r)^n - 1)$	$-m$
	Default	0	0

- we could take into account the risk of the model being wrong by trying to control the type 1 (false positive), we want to minimize false positives because this causes us to lose -m. We could enfore a higher threshold than 0 for the expected utility and then we could check this by demanding that the probability for type 1 errors (false positive) should be within an accepted range, e.g. maximum 5 %. We could attempt to find this threshold for type 1 erros by looking at the training data and check the probability for type 1 errors on a holdout set of the training data.

3.0.5 Does the existence of the database raise any privacy concerns?

The database is anonymized because there are no directly identifying attributes about the individuals (Dimitrakakis, 2020, p. 74). However, there is very specific information about the individuals in the database, such as age, personal status, sex and information about the employement situation of the individuals. There is also information about the housing and property situation of the different individuals.

There seems to be a high probability of inferring personal information about the individuals in the database by using for example record linkage.

3.0.6 How would secret database and public credit decisions affect privacy?

It would obviously be better to have the database secret instead of the database also being public. It could however be possible to infer a lot of information from the secret database by knowing the public credit decision.

Because the bank is publishing the actual credit decisions, differential privacy is not possible. If, for example, an adversary with information about all but one attribute for a specific row, the adversary could infer the information of the missing attribute.

3.0.7 How can we protect the data of the people in the training set?

We could use differential privacy to obscure the information in the public credit decisions that have been made by the bank.

- if the database is secret, but the credit decisions are public (we assume that the identities of the individuals also are public): we would have a randomised response mechanism for the credit decision. This randomised response mechanism will ensure that the public responses are correct with a probability ≤ 1 (Dimitrakakis, 2020, p. 77). This would make it harder to infer information about the individuals from the public information. Even if the database is not public, there is a risk of dishonest employees and/or digital attacks on the bank.
- if the database is public and the credit decisions are public: for categorical attributes, we could use a randomised response mechanism. For the numerical attributes, we can apply a Laplace mechanism in order to make the information differential private. In this scenario as weell, the bank is still vulnerable to dishonest employees and digital attacks as the original (true) data is still available inside the bank.

3.0.8 How can we protect the data of the people that apply for loans?

We assume that the data for a new individual could potentially be leaked and considered non-secret. In that case we could apply the same techniques as for people in the training set. We would then apply a random mechanism to the input data for the new applicant.

3.0.9 Implement a private decision making mechanism for people that apply for new loans

3.0.10 Estimate the amount of loss in utility as we change the privacy guarantee

4 References

Azzalini, A. & Scarpa, B. (2012). Data analysis and data mining: An introduction. Oxford: Oxford University Press.

Dimitrakakis, C. (2020). *Machine learning in science and society.* Unpublished. Department of Informatics, University of Oslo.

[ ]: