name-banker-doc

September 24, 2020

1 Model development

1.1 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]:
```python
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
```

1.2 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]:
```python
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.

1.3 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]:
```python
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]:
```python
import random_banker
import name_banker
from sklearn.model_selection import train_test_split
import numpy as np
import pandas as pd

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})
```

[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 name 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_name = np.zeros(num_of_tests)
    bank_investment_name = np.zeros_like(bank_utility_name)

    # decision makers
    r_banker = random_banker.RandomBanker()
    r_banker.set_interest_rate(interest_rate)
    n_banker = name_banker.NameBanker()
    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_name[i], bank_investment_name[i] = utility_from_test_set(
            X_test, y_test, n_banker, interest_rate)

    print(
        f"Avg. utility [random]\t= {np.sum(bank_utility_random)/num_of_tests}")
```

```
    print(
        f"Avg. ROI [random]     \t= {np.sum(bank_investment_random)/
↪num_of_tests}")
    print(
        f"Avg. utility [name]   \t= {np.sum(bank_utility_name)/num_of_tests}")
    print(
        f"Avg. ROI [name]       \t= {np.sum(bank_investment_name)/num_of_tests}")
```

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

```
Avg. utility [random]   = 592275.3023303407
Avg. ROI [random]       = 0.9136178799312753
Avg. utility [name]     = 1179658.5252952026
Avg. ROI [name]         = 1.8242265461380003
CPU times: user 3min 13s, sys: 3min 58s, total: 7min 12s
Wall time: 1min 8s
```

2.3 Results

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