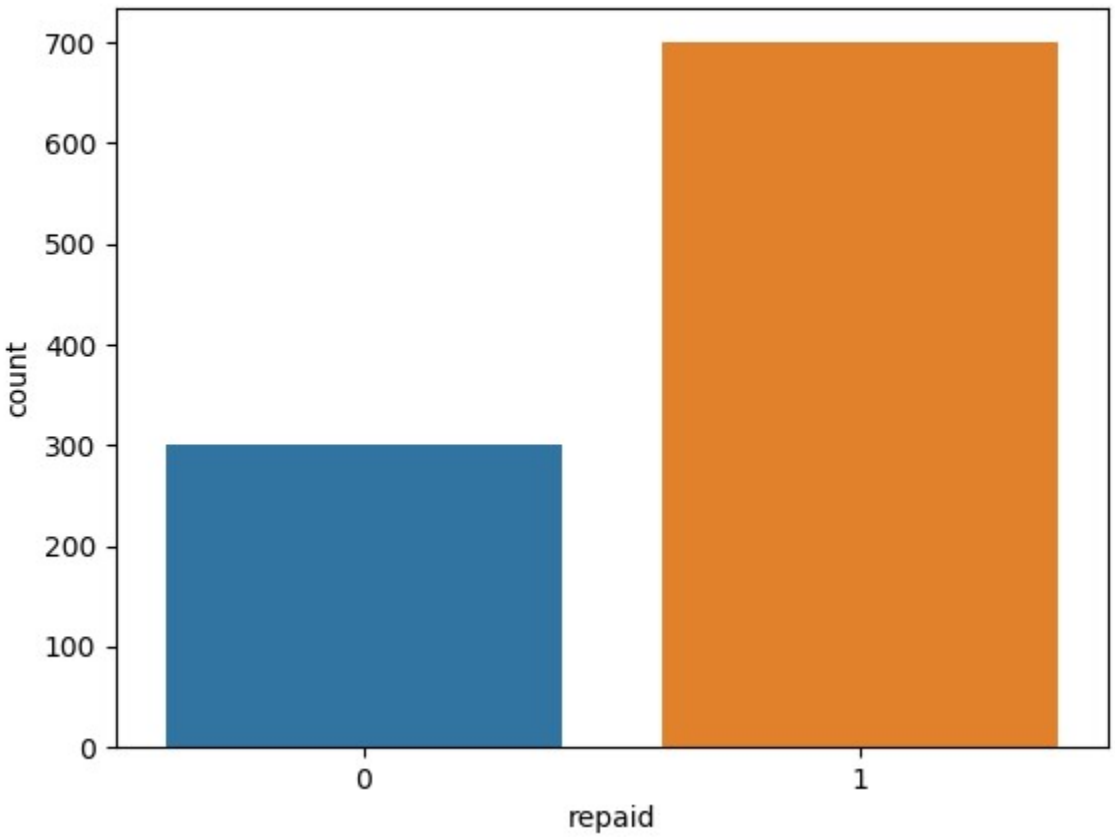
IN-STK5000 Project 1, report 1

First: Exploring and preprocessing the data. We start by looking at the german credit data file. When we make a countplot of the outcome variable (which is whether a given loan was repaid, where 0 is no and 1 is yes), we see that the data is imbalanced.

```
df.loc[df['repaid'] == 2, 'repaid'] = 0
sns.countplot(x = df[target], data = df)
```



The percentage of the outcome being 0, or not repaid, is 30%, while the percentage for the outcome being 1, or repaid, is 70%. For our model it is important to balance out our data, because the probability of a new data point belonging to the majority class (here 'repaid') is much higher than it belonging to the minority class (Yen & Lee, 2008). Therefore, a machine learning algorithm is more likely to classify new observations to the majority class. Meaning, In this case, if a model would predict only ones, it would be right 70% of the time. In order to balance the data we use the imbalanced-learn package in Python.

```
def underSample(X, y):
    from imblearn.under_sampling import RandomUnderSampler
    rus = RandomUnderSampler(random_state=42)
    return rus.fit_resample(X, y)

def overSample(X, y):
    from imblearn.over_sampling import SMOTE
    os = SMOTE(random_state=0)
    columns = X.columns
    os_data_X,os_data_y=os.fit_sample(X, y)
    os_data_X = pd.DataFrame(data=os_data_X,columns=columns )
    os_data_y= pd.DataFrame(data=os_data_y,columns=['y'])
```

```
    return os_data_X, os_data_y
```

Whether we had to under- or oversample our data is based on trial-and-error. The undersampling proved to work better for our data. After this the zeros and ones are 50/50 divided.

Using one-hot encoding the categorical attributes are converted in order to be able to use them in the machine learning algorithm. Normalising is done in the model itself, and therefore not in the preprocessing. The data is split into 80% training data and 20% test data.

Second: Designing a policy for giving or denying credit to individuals The choice for giving or denying credit to individuals is based on their probability for being credit-worthy. Given this probability, and taking into account the length of the loan, we can calculate the expected utility of giving a loan, using the formula E(U) = ((m(1-r)n-1)*p)-m(1-p)

```
def expected_utility(self, X):
    p = self.predict_proba(self.parse_X(X))
    gain = self.calculate_gain(X)
    expected_utilitiy = (gain.values*p.flatten()
                         -X['amount'].values*(1-p.flatten()))
    return expected_utilitiy
```

The probability is calculated using a neural network.

fit(): As a first layer for the model we use batch normalization. This centers and normalizes the input values. The main model is a simple fully connected artificial neural network with layer sizes 16 and 8, and elu activations. We use L2 regularization. The final layer consists of a single neuron with a sigmoid activation. The network is trained using binary cross-entropy loss.

The model is trained using a batch size of 32 and 10 epochs of training with the Adam optimizer.

```
def build_network(self, X, y):
    model = Sequential()
    model.add(BatchNormalization())
    for layer_size in self.layer_sizes:
        model.add(Dense(layer_size,
                        activation='elu',
                        kernel_regularizer=regularizers.l2(0.001)))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy',
                  optimizer=self.optimizer,
                  metrics=['accuracy'])
    return model

def fit(self, X, y):
    y = self.parse_y(y.values.reshape(-1,1))
    X = self.parse_X(X)
    os_data_X, os_data_y = underSample(X,y)
    self.model = self.build_network(os_data_X, os_data_y)
    self.model.fit(os_data_X,
                   os_data_y,
                   epochs = 10,
                   batch_size=32,
                   validation_split=0.05)
```

predict_proba(): This function merely outputs the result of running the trained network forward.

```
def predict_proba(self, X):
    return self.model.predict(X)
```

TODO: Add confusion matrix of the neural network

Now we have a model that is well trained and working. So we will be able to combine it with our policy for giving credit. We will retrieve the result of the function expected_utility(X). If the result is greater than 0, that is to say if we can make money

with this loan, the action will take the value 1. If the value returned is 0, the loan must not be granted. Since we made a change of value at the beginning, we change the 0 to 2.

```
def get_best_action(self, X):
    actions = (self.expected_utility(X) > 0).astype(int).flatten()
    actions[np.where(actions == 0)] = 2
    return actions
```

A 100-way cross validation of NeuralBanker gave the result (38.45, 1229.55), a big improvement compared to RandomBankers (-4960.33, 7593.33)