Wine Quality Classification

About dataset

The two datasets are related to red and white variants of the Portuguese "Vinho Verde" wine. The reference [Cortez et al., 2009]. Due to privacy and logistic issues, only physicochemical (inputs) and sensory (the output) variables are available (e.g. there is no data about grape types, wine brand, wine selling price, etc.).

Input variables (based on physicochemical tests):

1 - fixed acidity

2 - volatile acidity

3 - citric acid

4 - residual sugar

5 - chlorides

6 - free sulfur dioxide

7 - total sulfur dioxide

8 - density

9 - pH

10 - sulphates

11 - alcohol

Output variable (based on sensory data):

12 - quality (score between 0 and 10)

Imports

In [362]:
```
import pandas as pd
import numpy as np
from sklearn import tree
from sklearn.utils import shuffle
from sklearn.preprocessing import LabelEncoder, StandardScaler, MinMaxScaler
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Dense
import seaborn as sns
from tensorflow import keras
```

Loading data and shuffling the dataset

In [363]:
```
df = pd.read_csv("winequalityN.csv")
df = shuffle(df)
```

Out[363]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulf dioxi
count	6487.000000	6489.000000	6494.000000	6495.000000	6495.000000	6497.000000	6497.0000
mean	7.216579	0.339691	0.318722	5.444326	0.056042	30.525319	115.7445
std	1.296750	0.164649	0.145265	4.758125	0.035036	17.749400	56.5218
min	3.800000	0.080000	0.000000	0.600000	0.009000	1.000000	6.0000
25%	6.400000	0.230000	0.250000	1.800000	0.038000	17.000000	77.0000
50%	7.000000	0.290000	0.310000	3.000000	0.047000	29.000000	118.0000
75%	7.700000	0.400000	0.390000	8.100000	0.065000	41.000000	156.0000
max	15.900000	1.580000	1.660000	65.800000	0.611000	289.000000	440.0000

In [364]:

Out[364]:

	type	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphate
5339	red	11.9	0.40	0.65	2.15	0.068	7.0	27.0	0.99880	3.06	0.6
3217	white	5.8	0.33	0.23	5.00	0.053	29.0	106.0	0.99458	3.13	0.5
3992	white	6.7	0.19	0.32	3.70	0.041	26.0	76.0	0.99173	2.90	0.5

Dataset info

As we can see we have unfortunately inconsistent data in some rows there are data gaps

In [365]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 6497 entries, 5339 to 63
Data columns (total 13 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   type                  6497 non-null   object 
 1   fixed acidity         6487 non-null   float64
 2   volatile acidity      6489 non-null   float64
 3   citric acid           6494 non-null   float64
 4   residual sugar        6495 non-null   float64
 5   chlorides             6495 non-null   float64
 6   free sulfur dioxide   6497 non-null   float64
 7   total sulfur dioxide  6497 non-null   float64
 8   density               6497 non-null   float64
 9   pH                    6488 non-null   float64
 10  sulphates             6493 non-null   float64
 11  alcohol               6497 non-null   float64
 12  quality               6497 non-null   int64  
dtypes: float64(11), int64(1), object(1)
memory usage: 710.6+ KB
```

In [366]:
```
df.isnull().sum()
```

Out[366]:
```
type                     0
fixed acidity           10
volatile acidity         8
citric acid              3
residual sugar           2
chlorides                2
free sulfur dioxide      0
total sulfur dioxide     0
density                  0
pH                       9
sulphates                4
alcohol                  0
quality                  0
dtype: int64
```

The first database preprocess

At the beginning, we delete the rows in which we have missing data

In [367]:
```
classic_df = df.dropna(axis=0)
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 6463 entries, 5339 to 63
Data columns (total 13 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   type                  6463 non-null   object 
 1   fixed acidity         6463 non-null   float64
 2   volatile acidity      6463 non-null   float64
 3   citric acid           6463 non-null   float64
 4   residual sugar        6463 non-null   float64
 5   chlorides             6463 non-null   float64
 6   free sulfur dioxide   6463 non-null   float64
 7   total sulfur dioxide  6463 non-null   float64
 8   density               6463 non-null   float64
 9   pH                    6463 non-null   float64
 10  sulphates             6463 non-null   float64
 11  alcohol               6463 non-null   float64
 12  quality               6463 non-null   int64  
dtypes: float64(11), int64(1), object(1)
memory usage: 706.9+ KB
```

In [368]:

Out[368]:
```
6    2820
5    2128
7    1074
4     214
8     192
3      30
9       5
Name: quality, dtype: int64
```

Then we divide our data into X and y where X is input and y is output (quality)

In [369]:
```
X = classic_df.drop(columns="quality")
y = classic_df['quality']
```

Out[369]:

	type	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphate
5339	red	11.9	0.40	0.65	2.15	0.068	7.0	27.0	0.99880	3.06	0.6
3217	white	5.8	0.33	0.23	5.00	0.053	29.0	106.0	0.99458	3.13	0.5
3992	white	6.7	0.19	0.32	3.70	0.041	26.0	76.0	0.99173	2.90	0.5

In [370]:

Out[370]:
```
5339    6
3217    5
3992    7
2215    6
87      6
1868    7
1376    6
2188    6
20      8
192     6
2702    5
3882    5
1513    6
6069    6
6459    5
Name: quality, dtype: int64
```

We replace the quality rating with versions with only two solutions - good and average

In [371]:
```
bins = [0, 5.5, 10]
labels = ["average", "good"]
y = pd.cut(y, bins=bins, labels=labels)
```

Out[371]:
```
5339       good
3217    average
3992       good
2215       good
87         good
1868       good
1376       good
2188       good
20         good
192        good
2702    average
3882    average
1513       good
6069       good
6459    average
Name: quality, dtype: category
Categories (2, object): ['average' < 'good']
```

We replace average and good with 0 and 1 and wine types (red and white) also with 0 and 1 as well

In [372]:
```
le = LabelEncoder()
y = le.fit_transform(y)
```

Out[372]:
```
array([1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0])
```

In [373]:
```python
X['type'] = le.fit_transform(X['type'])
```

Out[373]:

	type	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates
5339	0	11.9	0.40	0.65	2.15	0.068	7.0	27.0	0.99880	3.06	0.68
3217	1	5.8	0.33	0.23	5.00	0.053	29.0	106.0	0.99458	3.13	0.52
3992	1	6.7	0.19	0.32	3.70	0.041	26.0	76.0	0.99173	2.90	0.57
2215	1	8.5	0.28	0.34	13.80	0.041	32.0	161.0	0.99810	3.13	0.40
87	1	6.8	0.25	0.31	13.30	0.050	69.0	202.0	0.99720	3.22	0.48

Data standardization

In [374]:
```python
sc = StandardScaler()
X = sc.fit_transform(X)
```

Out[374]: (6463, 12)

Data split into test and training set 30 - 70%

In [375]:

Decision trees

We use two decision trees - one without constraints and the other with a maximum depth of three levels

In [376]:
```python
clf = tree.DecisionTreeClassifier()
```

In [377]:
```python
clf = clf.fit(x_train, y_train)
```

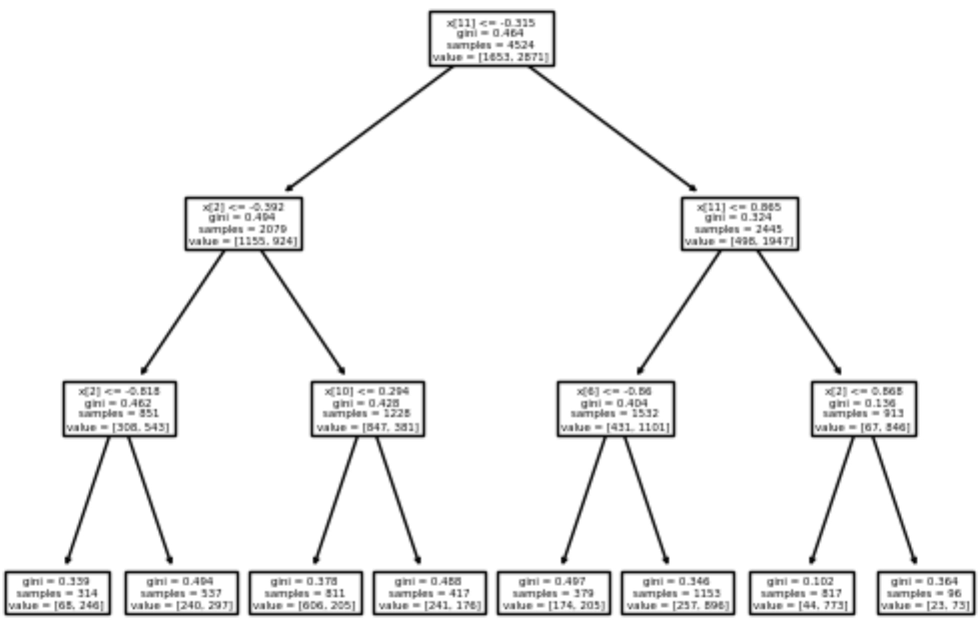
In [378]:
```
tree.plot_tree(clf)
```

Out[378]:
```
[Text(0.5338475113646644, 0.9772727272727273, 'x[11] <= -0.315\ngini = 0.464\
nsamples = 4524\nvalue = [1653, 2871]'),
 Text(0.25366312533924373, 0.9318181818181818, 'x[2] <= -0.392\ngini = 0.494\
nsamples = 2079\nvalue = [1155, 924]'),
 Text(0.06310569137868645, 0.8863636363636364, 'x[2] <= -0.818\ngini = 0.462\
nsamples = 851\nvalue = [308, 543]'),
 Text(0.03690971593993125, 0.8409090909090909, 'x[4] <= 0.884\ngini = 0.339\n
samples = 314\nvalue = [68, 246]'),
 Text(0.029491586755925458, 0.7954545454545454, 'x[6] <= -0.958\ngini = 0.41
7\nsamples = 182\nvalue = [54, 128]'),
 Text(0.025511127193776007, 0.75, 'x[4] <= -0.493\ngini = 0.484\nsamples = 1
7\nvalue = [10, 7]'),
 Text(0.02406368735299439, 0.7045454545454546, 'x[5] <= 2.707\ngini = 0.165\n
samples = 11\nvalue = [10, 1]'),
 Text(0.022616247512212775, 0.6590909090909091, 'gini = 0.0\nsamples = 10\nva
lue = [10, 0]'),
 Text(0.025511127193776007, 0.6590909090909091, 'gini = 0.0\nsamples = 1\nval
ue = [0, 1]'),
 Text(0.026958567034557627, 0.7045454545454546, 'gini = 0.0\nsamples = 6\nval
ue = [0, 6]'),
```

In [379]:

Out[379]:

```
[Text(0.5, 0.875, 'x[11] <= -0.315\ngini = 0.464\nsamples = 4524\nvalue = [16
53, 2871]'),
 Text(0.25, 0.625, 'x[2] <= -0.392\ngini = 0.494\nsamples = 2079\nvalue = [11
55, 924]'),
 Text(0.125, 0.375, 'x[2] <= -0.818\ngini = 0.462\nsamples = 851\nvalue = [30
8, 543]'),
 Text(0.0625, 0.125, 'gini = 0.339\nsamples = 314\nvalue = [68, 246]'),
 Text(0.1875, 0.125, 'gini = 0.494\nsamples = 537\nvalue = [240, 297]'),
 Text(0.375, 0.375, 'x[10] <= 0.294\ngini = 0.428\nsamples = 1228\nvalue = [8
47, 381]'),
 Text(0.3125, 0.125, 'gini = 0.378\nsamples = 811\nvalue = [606, 205]'),
 Text(0.4375, 0.125, 'gini = 0.488\nsamples = 417\nvalue = [241, 176]'),
 Text(0.75, 0.625, 'x[11] <= 0.865\ngini = 0.324\nsamples = 2445\nvalue = [49
8, 1947]'),
 Text(0.625, 0.375, 'x[6] <= -0.86\ngini = 0.404\nsamples = 1532\nvalue = [43
1, 1101]'),
 Text(0.5625, 0.125, 'gini = 0.497\nsamples = 379\nvalue = [174, 205]'),
 Text(0.6875, 0.125, 'gini = 0.346\nsamples = 1153\nvalue = [257, 896]'),
 Text(0.875, 0.375, 'x[2] <= 0.868\ngini = 0.136\nsamples = 913\nvalue = [67,
846]'),
 Text(0.8125, 0.125, 'gini = 0.102\nsamples = 817\nvalue = [44, 773]'),
 Text(0.9375, 0.125, 'gini = 0.364\nsamples = 96\nvalue = [23, 73]')]
```



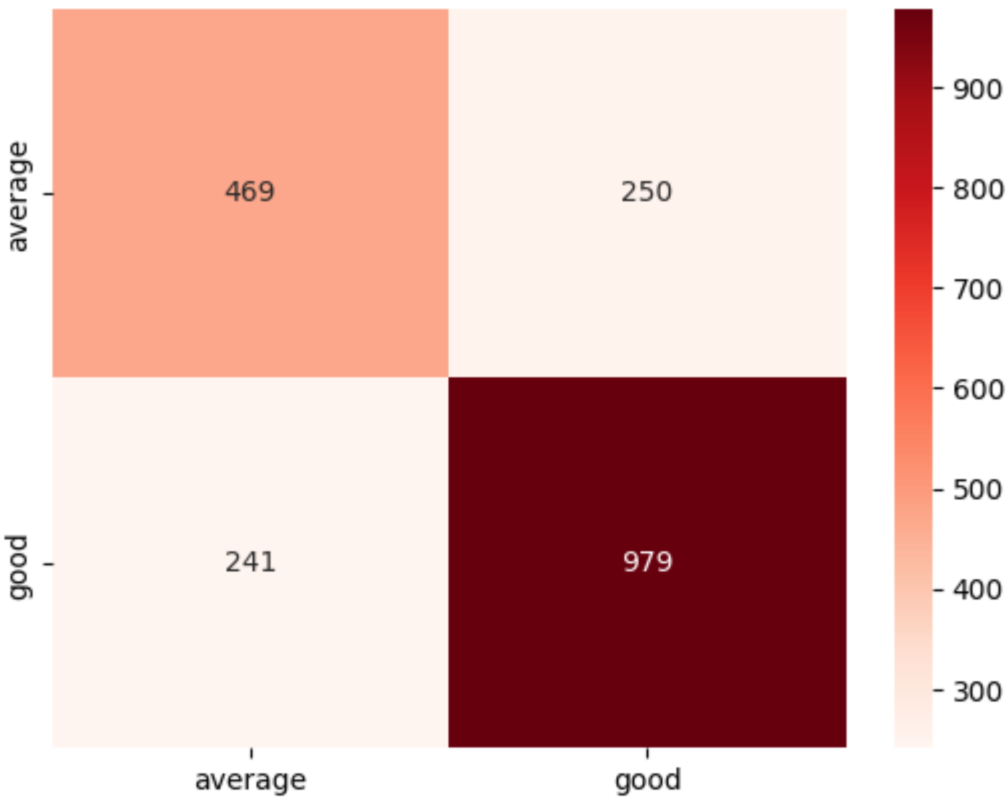
In [380]:
```
prediction = clf.predict(x_test)
prediction_smaller = clf_smaller.predict(x_test)

print("Accuracy on test data set with bigger tree: ", accuracy_score(predictio
```

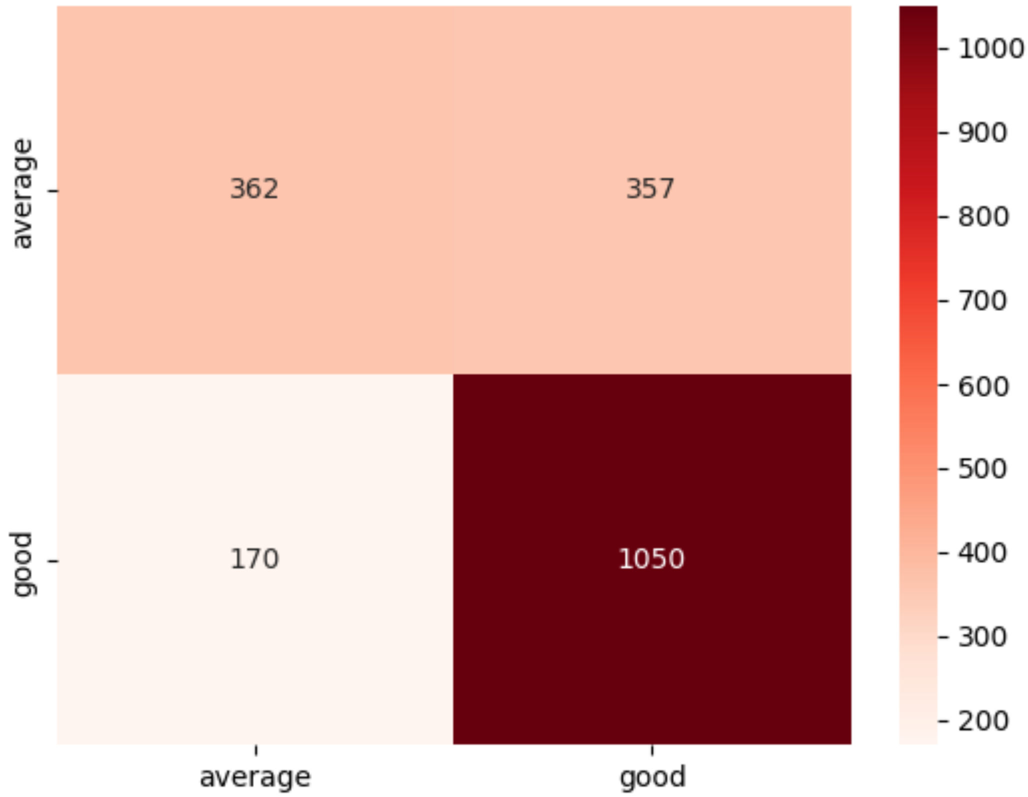
```
Accuracy on test data set with bigger tree:  0.7467766890149562
Accuracy on test data set with smaller tree:  0.7282104177411036
```

As we can see, a deeper decision tree performs better, but it takes more time

In [381]:



In [382]:
```
ax = sns.heatmap(confusion_matrix(y_test, prediction_smaller), annot=True, fmt
```



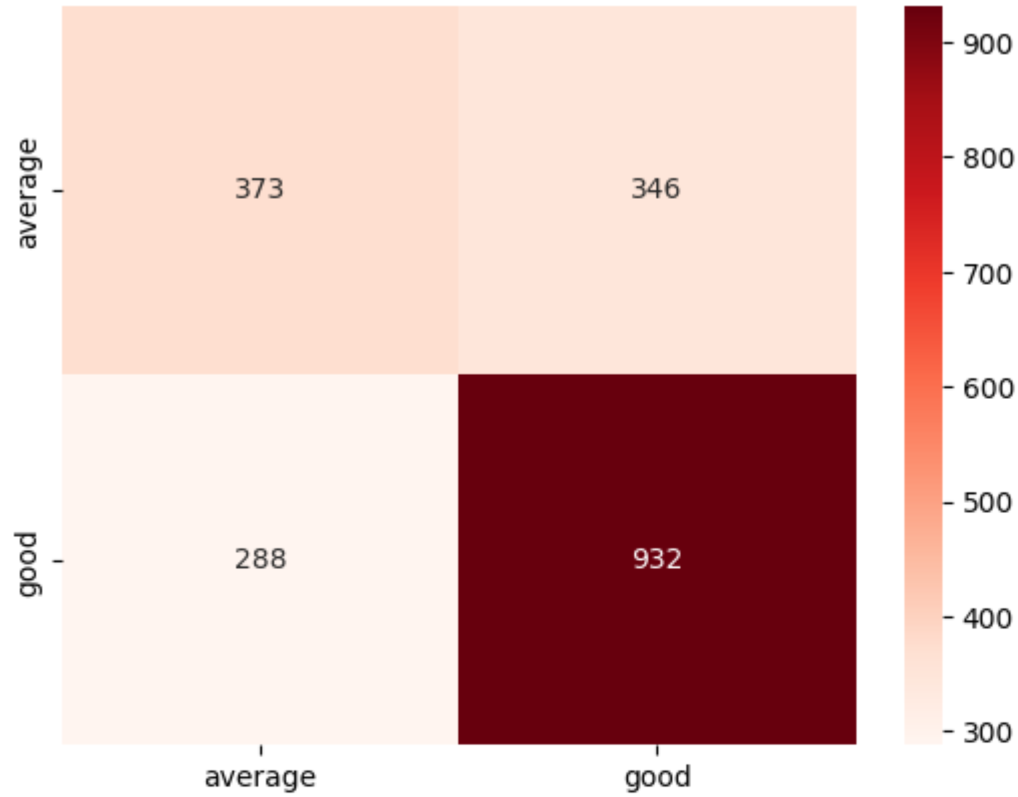
Naive-Bayes

In [383]:
```
model = GaussianNB()
model.fit(x_train, y_train)
```

```
Accuracy on test data set:  0.6730273336771532
```

As we can see, it has worse accuracy than the decision trees

In [384]:
```python
prediction = model.predict(x_test)
```



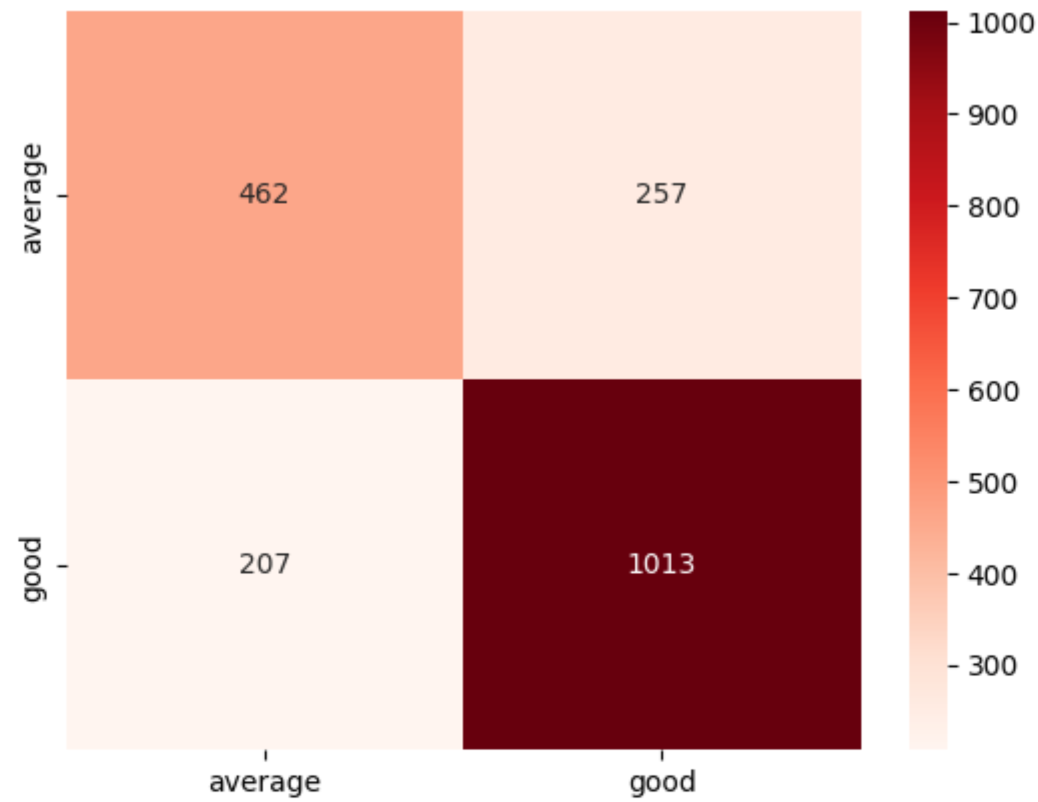
K-nearest neighbors

First try with three neighbors

In [385]:
```python
knn = KNeighborsClassifier(n_neighbors=3, metric='euclidean')
knn.fit(x_train, y_train)
```

```
Accuracy on test data set:  0.7607013924703455
```

In [386]:
```
prediction = knn.predict(x_test)
```

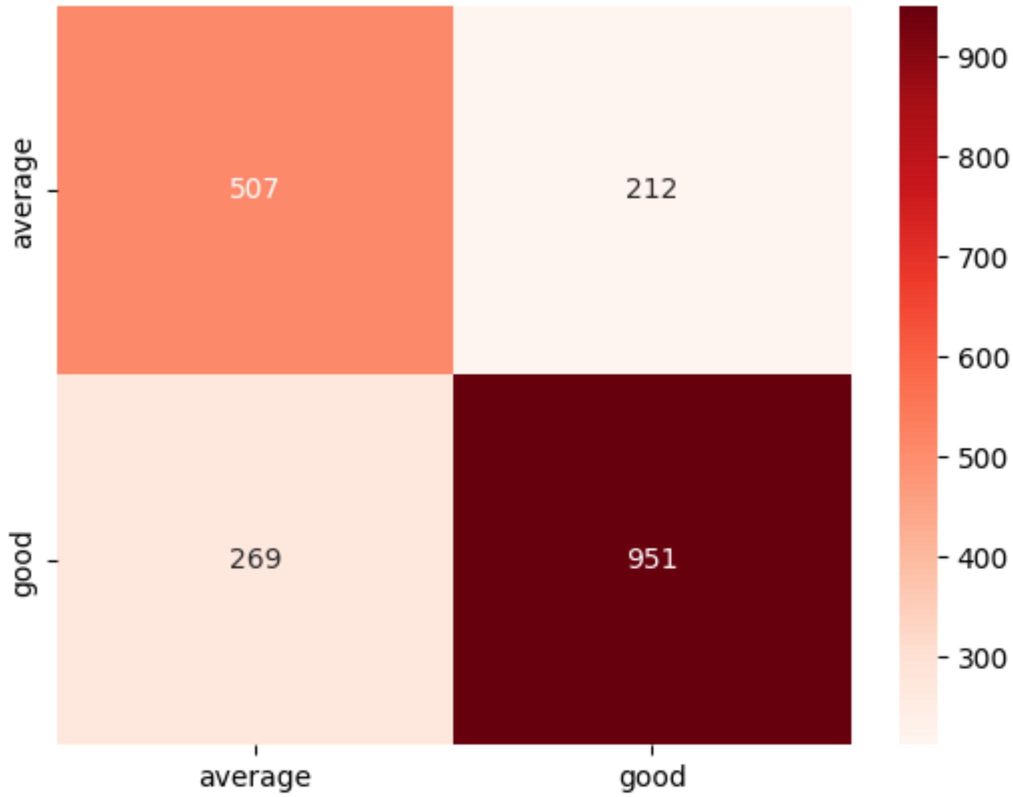


Six neighbors

In [387]:
```
knn = KNeighborsClassifier(n_neighbors=6, metric='euclidean')
knn.fit(x_train, y_train)
```

```
Accuracy on test data set:  0.7519339865910263
```

In [388]:
```
prediction = knn.predict(x_test)
```

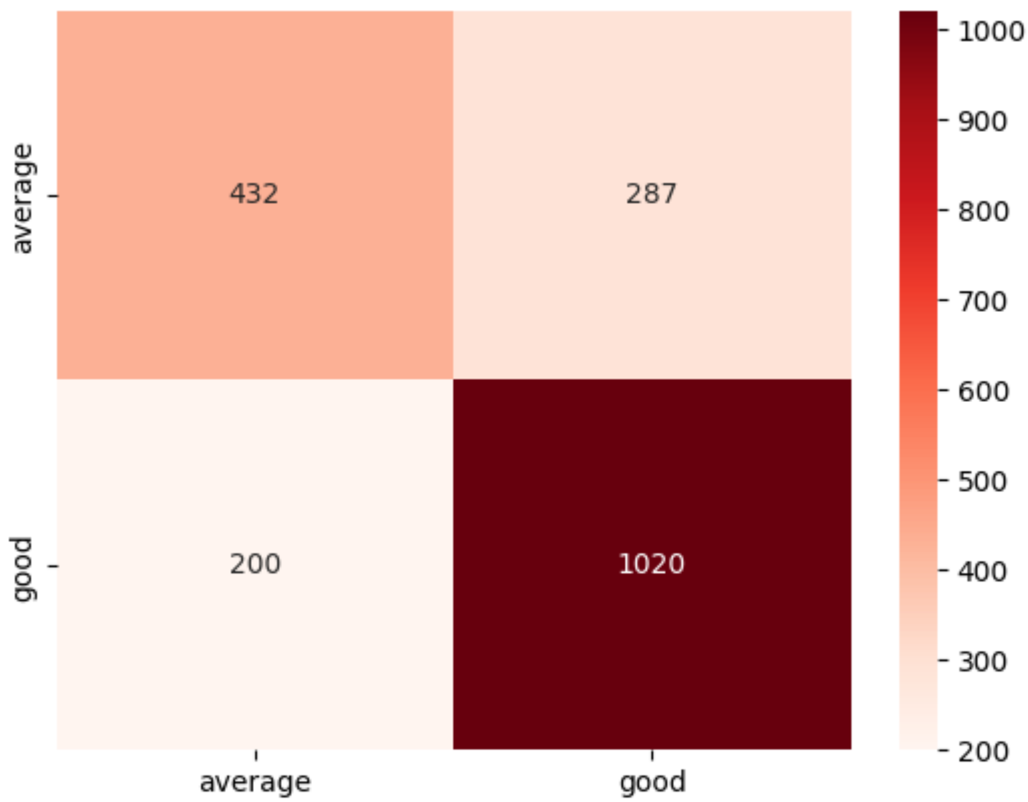


Nine neighbors

In [389]:
```
knn = KNeighborsClassifier(n_neighbors=9, metric='euclidean')
knn.fit(x_train, y_train)
```

```
Accuracy on test data set:  0.7488396080453842
```

In [390]:
```python
prediction = knn.predict(x_test)
```

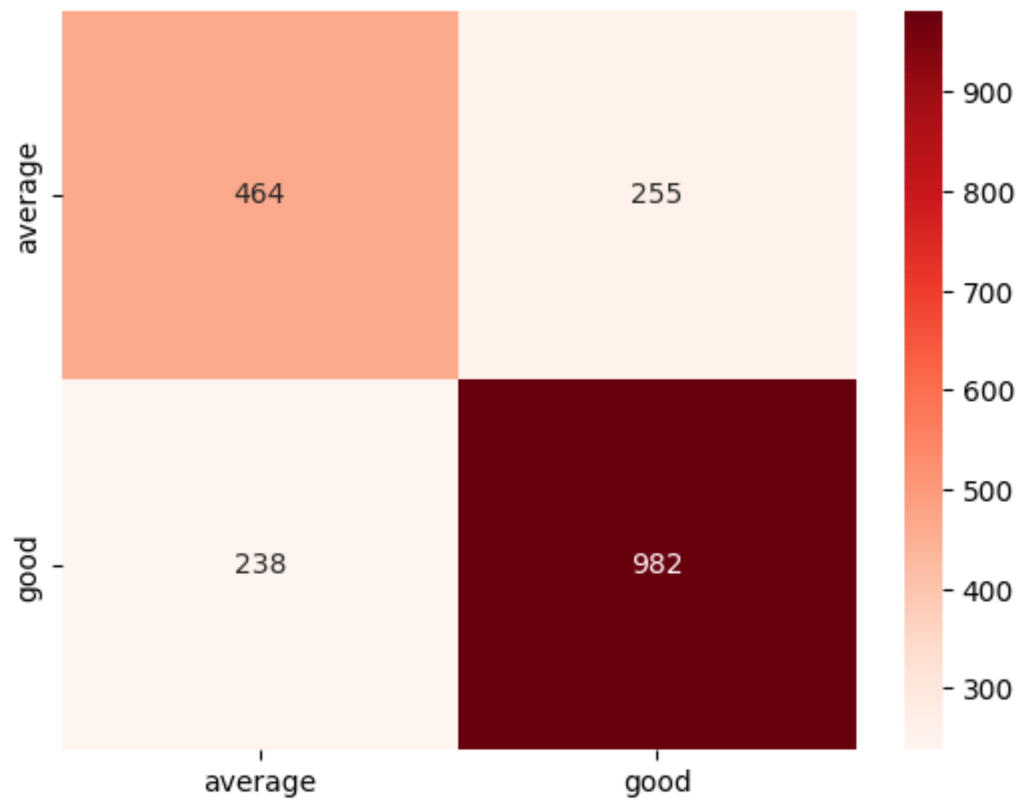


Twelve neighbors

In [391]:
```python
knn = KNeighborsClassifier(n_neighbors=12, metric='euclidean')
knn.fit(x_train, y_train)
```

```
Accuracy on test data set:  0.7457452294997421
```

In [392]:
```
prediction = knn.predict(x_test)
```



As we can see, there is no big difference in the results

Neural networks

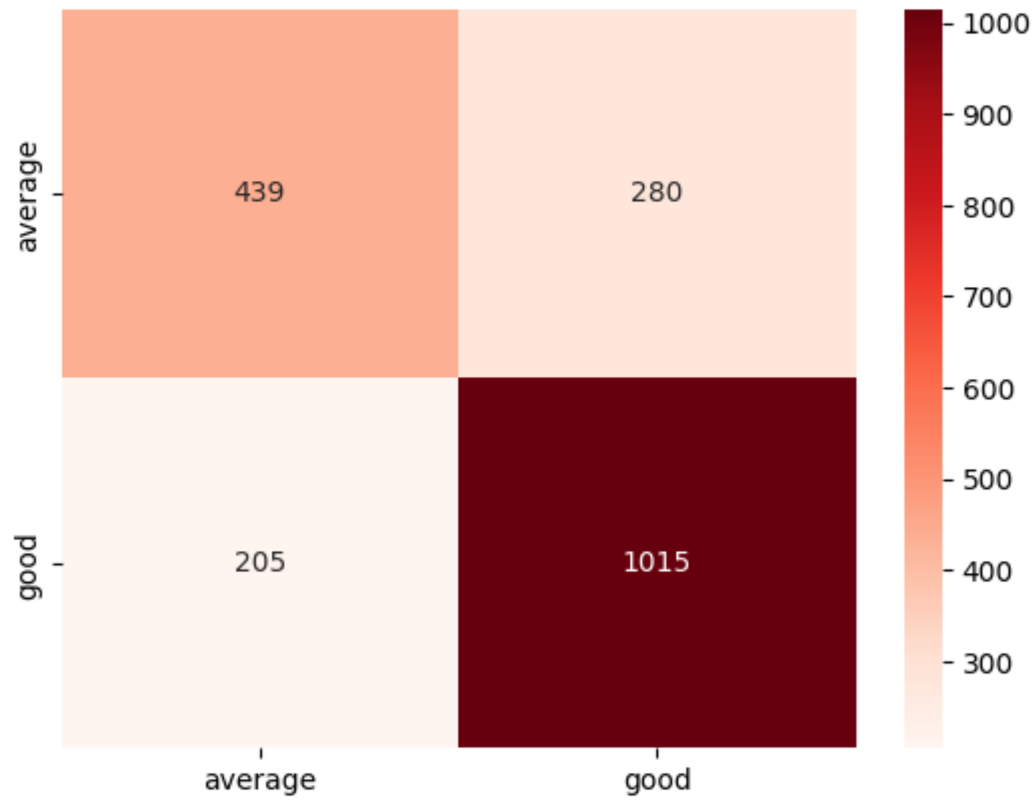
First try with MLPClassifier from sklearn with structure 6, 3, relu activation function, solver adam

In [393]:
```
clf = MLPClassifier(solver='adam', alpha=1e-5, hidden_layer_sizes=(6, 3), rand
clf = clf.fit(x_train, y_train)
prediction = clf.predict(x_test)
```

```
Accuracy on test data set:  0.7498710675605983

c:\Users\Piotr Damrych\AppData\Local\Programs\Python\Python311\Lib\site-packa
ges\sklearn\neural_network\_multilayer_perceptron.py:686: ConvergenceWarning:
Stochastic Optimizer: Maximum iterations (200) reached and the optimization h
asn't converged yet.
  warnings.warn(
```

In [394]:



Preparation of the dataset for neural networks with keras

In [395]:

```
y_train = keras.utils.to_categorical(y_train, num_classes=2)
y_test_to_validation = keras.utils.to_categorical(y_test, num_classes=2)
```

Out[395]:

```
array([[0., 1.],
       [0., 1.],
       [0., 1.],
       [0., 1.],
       [1., 0.]], dtype=float32)
```

First try with 3, 6, 2 structure, relu and sigmoid activation functions, solver adam

In [396]:

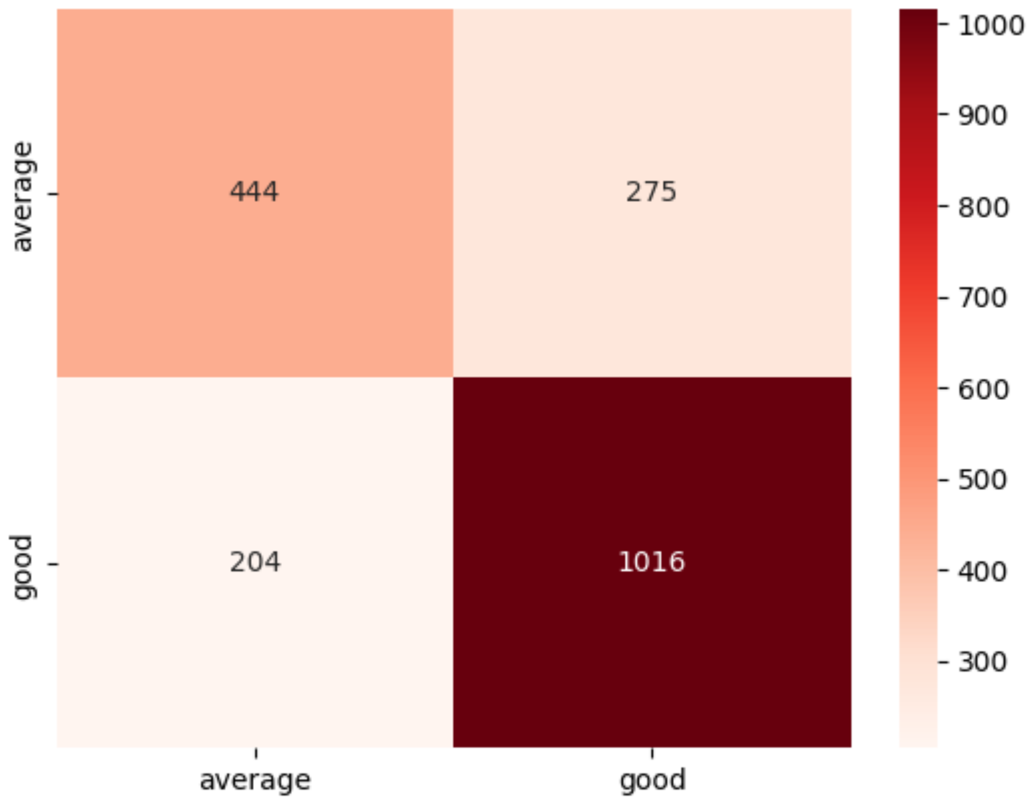
```
model = Sequential()
model.add(Dense(3, activation='relu', input_dim=(x_train.shape[1])))
model.add(Dense(6, activation='relu'))
model.add(Dense(2, activation='softmax'))

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy
```

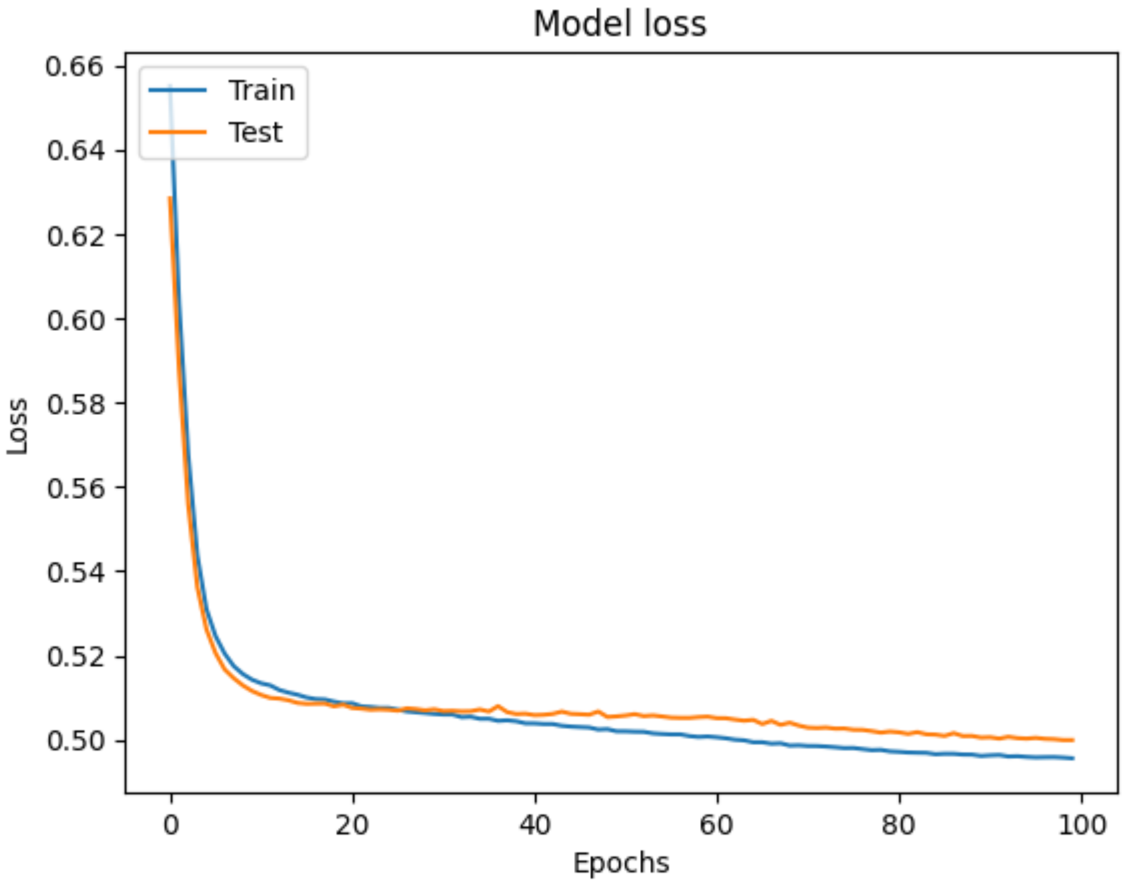
In [397]:
```
pred = model.predict(x_test)
pred = np.argmax(pred, axis=-1)
```

```
61/61 [==============================] - 0s 1ms/step
Accuracy:  0.7529654461062403
```

In [398]:



In [399]:
```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend(['Train', 'Test'], loc='upper left')
```



Second try with 3, 6, 2 structure, elu and sigmoid activation functions, solver adam

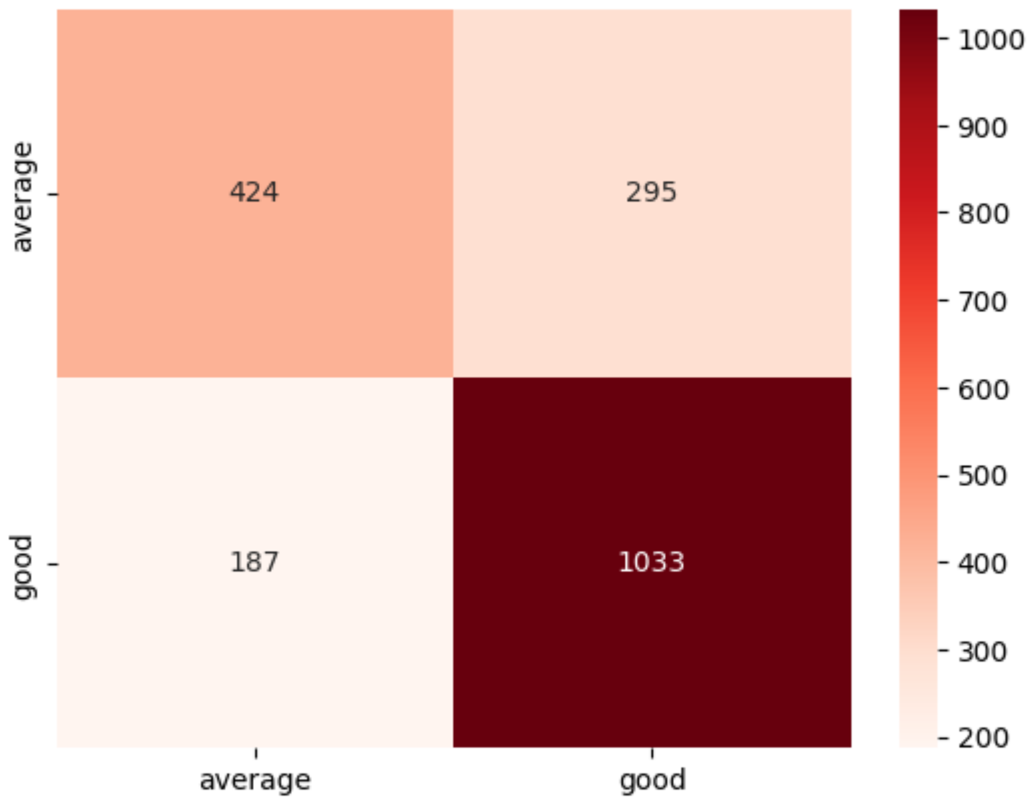
In [400]:
```
model = Sequential()
model.add(Dense(3, activation='elu', input_dim=(x_train.shape[1])))
model.add(Dense(6, activation='elu'))
model.add(Dense(2, activation='sigmoid'))

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy
```

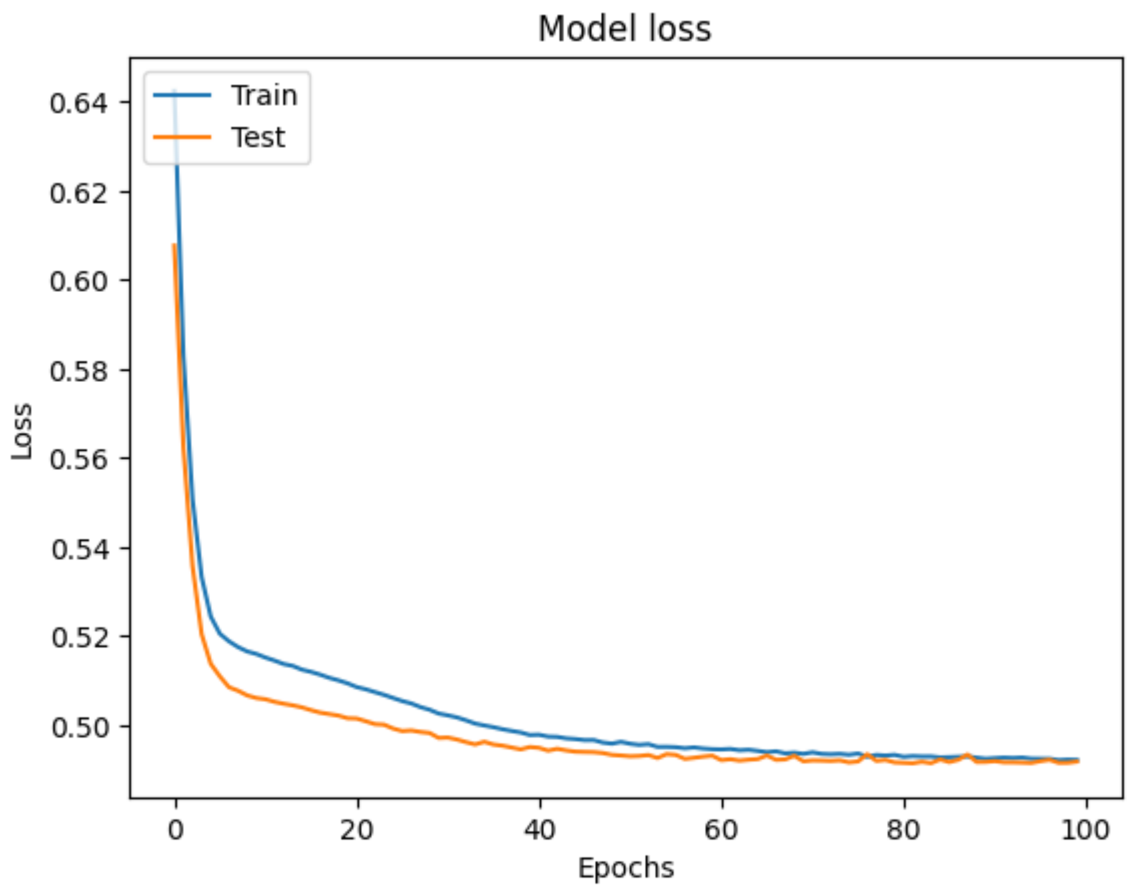
In [401]:
```
pred = model.predict(x_test)
pred = np.argmax(pred, axis=-1)
```

```
61/61 [==============================] - 0s 1ms/step
Accuracy:  0.7514182568334193
```

In [402]:


average
good
424
295
187
1033
average
good
1000
900
800
700
600
500
400
300
200

In [403]:
```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend(['Train', 'Test'], loc='upper left')
```



Third try with 3, 4, 2 structure, tanh and sigmoid activation functions, solver adam

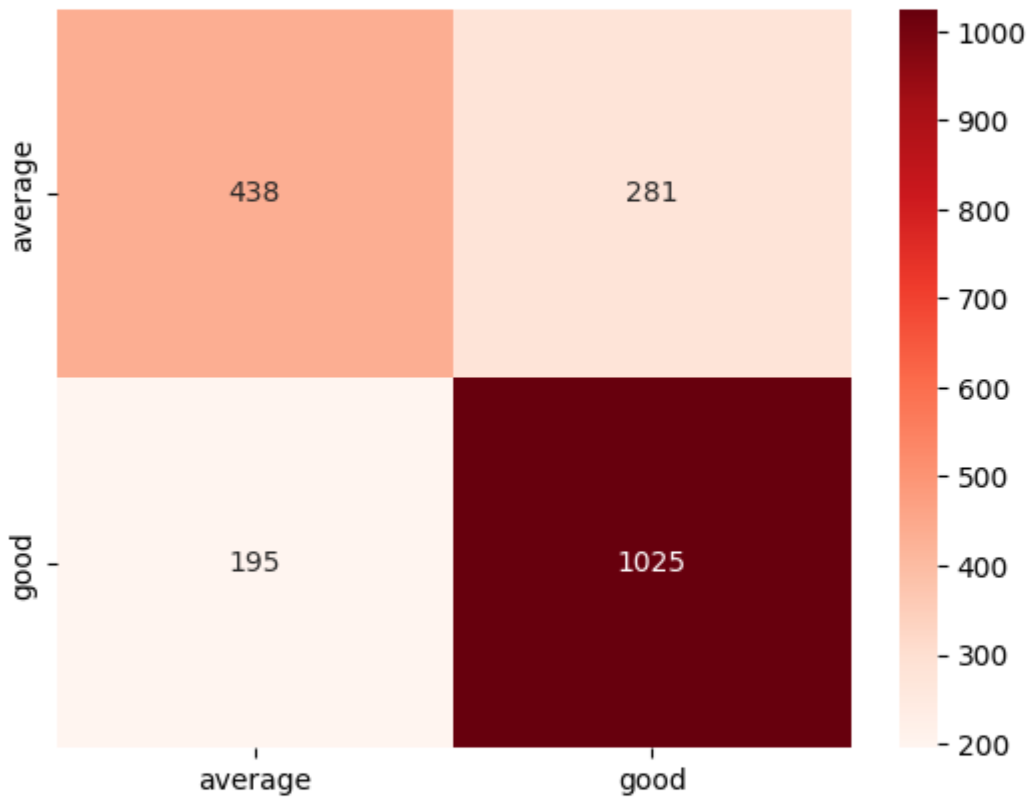
In [404]:
```
model = Sequential()
model.add(Dense(3, activation='tanh', input_dim=(x_train.shape[1])))
model.add(Dense(4, activation='tanh'))
model.add(Dense(2, activation='sigmoid'))

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy
```

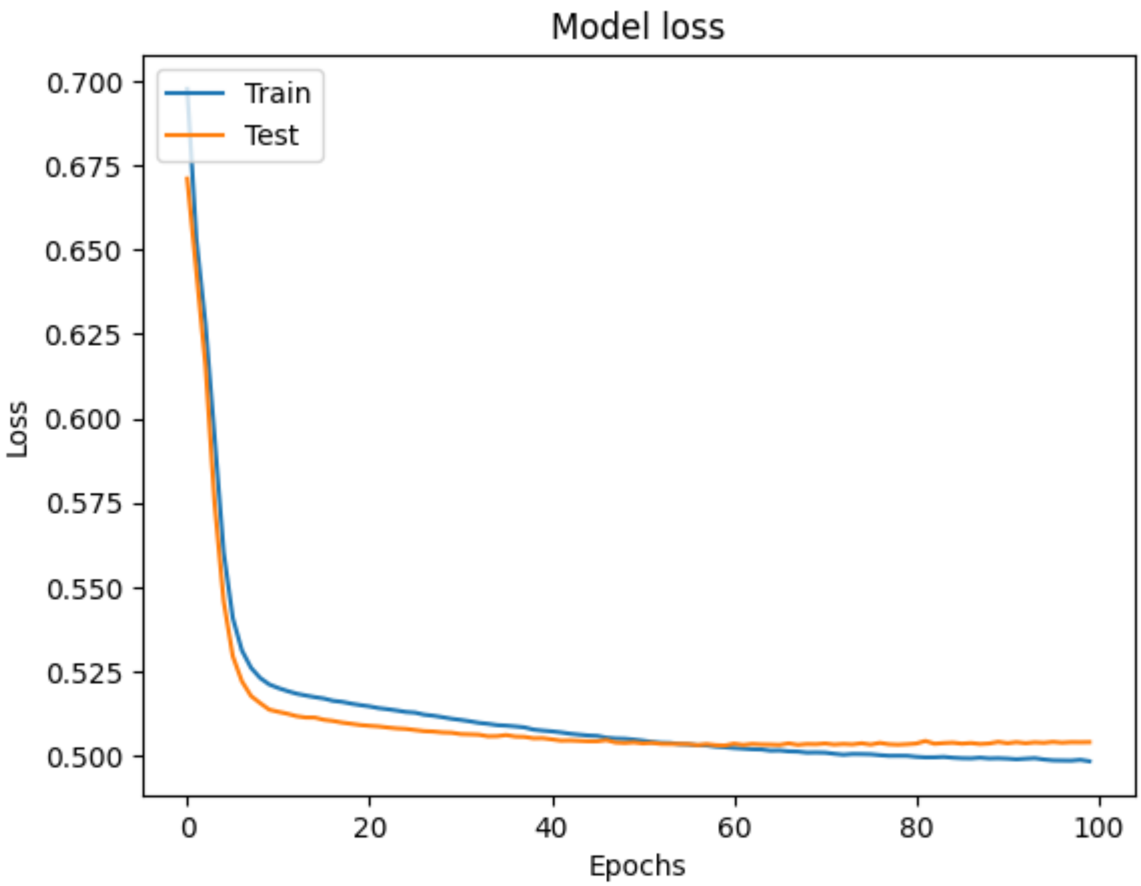
In [405]:
```
pred = model.predict(x_test)
pred = np.argmax(pred, axis=-1)
```

```
61/61 [==============================] - 0s 1ms/step
Accuracy:  0.7545126353790613
```

In [406]:


average
good
438
281
195
1025
average
good
1000
900
800
700
600
500
400
300
200

In [407]:
```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend(['Train', 'Test'], loc='upper left')
```



Another dataset processing

This time, instead of deleting rows with empty values, we extract the mean for them

In [408]:
```python
different_df = df
different_df['fixed acidity'] = different_df['fixed acidity'].fillna(different
different_df['volatile acidity'] = different_df['volatile acidity'].fillna(dif
different_df['citric acid'] = different_df['citric acid'].fillna(different_df[
different_df['residual sugar'] = different_df['residual sugar'].fillna(differe
different_df['chlorides'] = different_df['chlorides'].fillna(different_df['chl
different_df['pH'] = different_df['pH'].fillna(different_df['pH'].mean())
different_df['sulphates'] = different_df['sulphates'].fillna(different_df['sul

different_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 6497 entries, 5339 to 63
Data columns (total 13 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   type                  6497 non-null   object 
 1   fixed acidity         6497 non-null   float64
 2   volatile acidity      6497 non-null   float64
 3   citric acid           6497 non-null   float64
 4   residual sugar        6497 non-null   float64
 5   chlorides             6497 non-null   float64
 6   free sulfur dioxide   6497 non-null   float64
 7   total sulfur dioxide  6497 non-null   float64
 8   density               6497 non-null   float64
 9   pH                    6497 non-null   float64
 10  sulphates             6497 non-null   float64
 11  alcohol               6497 non-null   float64
 12  quality               6497 non-null   int64  
dtypes: float64(11), int64(1), object(1)
memory usage: 710.6+ KB
```

In [409]:
```python
different_df['quality'].value_counts()
```

Out[409]:
```
6    2836
5    2138
7    1079
4     216
8     193
3      30
9       5
Name: quality, dtype: int64
```

Preparing the dataset is the same as in the previous example with one exception

The only difference is normalization, where we use MinMaxScaler instead of StandardScaler, which converts all values to values between 0 and 1

In [410]:
```
X = different_df.drop(columns="quality")
y = different_df['quality']
```

Out[410]:

	type	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphate
5339	red	11.9	0.40	0.65	2.15	0.068	7.0	27.0	0.99880	3.06	0.6
3217	white	5.8	0.33	0.23	5.00	0.053	29.0	106.0	0.99458	3.13	0.5
3992	white	6.7	0.19	0.32	3.70	0.041	26.0	76.0	0.99173	2.90	0.5

In [411]:
```
bins = [0, 5.5, 10]
labels = ["average", "good"]
y = pd.cut(y, bins=bins, labels=labels)
```

Out[411]:
```
5339       good
3217    average
3992       good
2215       good
87         good
1868       good
1376       good
2188       good
20         good
192        good
2702    average
3882    average
1513       good
6069       good
6459    average
Name: quality, dtype: category
Categories (2, object): ['average' < 'good']
```

In [412]:
```
le = LabelEncoder()
y = le.fit_transform(y)
```

Out[412]:
```
array([1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0])
```

In [413]:
```
X['type'] = le.fit_transform(X['type'])
```

Out[413]:

	type	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates
5339	0	11.9	0.40	0.65	2.15	0.068	7.0	27.0	0.99880	3.06	0.68
3217	1	5.8	0.33	0.23	5.00	0.053	29.0	106.0	0.99458	3.13	0.52
3992	1	6.7	0.19	0.32	3.70	0.041	26.0	76.0	0.99173	2.90	0.57
2215	1	8.5	0.28	0.34	13.80	0.041	32.0	161.0	0.99810	3.13	0.40
87	1	6.8	0.25	0.31	13.30	0.050	69.0	202.0	0.99720	3.22	0.48

In [414]:
```
sc = MinMaxScaler()
X = sc.fit_transform(X)
```

Out[414]:
```
(6497, 12)
```

Data split into test and training set 30 - 70%

In [415]:

Decision trees

We use two decision trees - one without constraints and the other with a maximum depth of three levels

In [416]:
```
clf = tree.DecisionTreeClassifier()
```

In [417]:
```
clf = clf.fit(x_train, y_train)
```

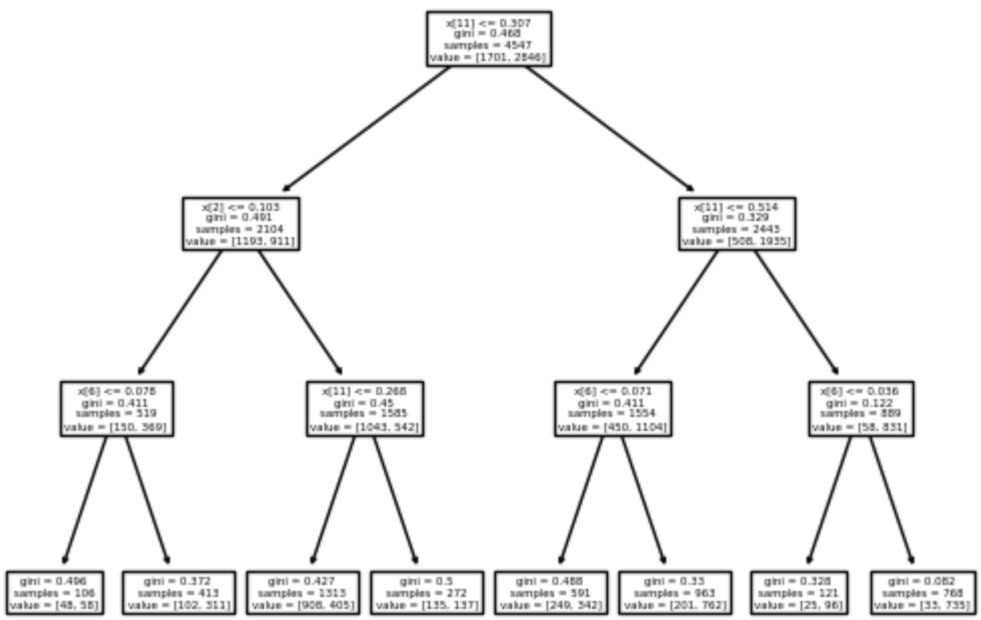
In [418]:

Out[418]:
```
[Text(0.5299146938094853, 0.9791666666666666, 'x[11] <= 0.307\ngini = 0.468\n
samples = 4547\nvalue = [1701, 2846]'),
 Text(0.21382269985330807, 0.9375, 'x[2] <= 0.103\ngini = 0.491\nsamples = 21
04\nvalue = [1193, 911]'),
 Text(0.04529257530955461, 0.8958333333333334, 'x[6] <= 0.078\ngini = 0.411\n
samples = 519\nvalue = [150, 369]'),
 Text(0.014045462945851045, 0.8541666666666666, 'x[4] <= 0.038\ngini = 0.496\
nsamples = 106\nvalue = [48, 58]'),
 Text(0.0073923489188689705, 0.8125, 'x[5] <= 0.092\ngini = 0.459\nsamples =
56\nvalue = [36, 20]'),
 Text(0.005913879135095177, 0.7708333333333334, 'x[8] <= 0.126\ngini = 0.426\
nsamples = 52\nvalue = [36, 16]'),
 Text(0.0029569395675475884, 0.7291666666666666, 'x[6] <= 0.043\ngini = 0.49
7\nsamples = 28\nvalue = [15, 13]'),
 Text(0.0014784697837737942, 0.6875, 'gini = 0.0\nsamples = 10\nvalue = [10,
0]'),
 Text(0.004435409351321382, 0.6875, 'x[5] <= 0.069\ngini = 0.401\nsamples = 1
8\nvalue = [5, 13]'),
 Text(0.0029569395675475884, 0.6458333333333334, 'x[6] <= 0.068\ngini = 0.23
```

In [419]:

Out[419]:

```
[Text(0.5, 0.875, 'x[11] <= 0.307\ngini = 0.468\nsamples = 4547\nvalue = [170
1, 2846]'),
 Text(0.25, 0.625, 'x[2] <= 0.103\ngini = 0.491\nsamples = 2104\nvalue = [119
3, 911]'),
 Text(0.125, 0.375, 'x[6] <= 0.078\ngini = 0.411\nsamples = 519\nvalue = [15
0, 369]'),
 Text(0.0625, 0.125, 'gini = 0.496\nsamples = 106\nvalue = [48, 58]'),
 Text(0.1875, 0.125, 'gini = 0.372\nsamples = 413\nvalue = [102, 311]'),
 Text(0.375, 0.375, 'x[11] <= 0.268\ngini = 0.45\nsamples = 1585\nvalue = [10
43, 542]'),
 Text(0.3125, 0.125, 'gini = 0.427\nsamples = 1313\nvalue = [908, 405]'),
 Text(0.4375, 0.125, 'gini = 0.5\nsamples = 272\nvalue = [135, 137]'),
 Text(0.75, 0.625, 'x[11] <= 0.514\ngini = 0.329\nsamples = 2443\nvalue = [50
8, 1935]'),
 Text(0.625, 0.375, 'x[6] <= 0.071\ngini = 0.411\nsamples = 1554\nvalue = [45
0, 1104]'),
 Text(0.5625, 0.125, 'gini = 0.488\nsamples = 591\nvalue = [249, 342]'),
 Text(0.6875, 0.125, 'gini = 0.33\nsamples = 963\nvalue = [201, 762]'),
 Text(0.875, 0.375, 'x[6] <= 0.036\ngini = 0.122\nsamples = 889\nvalue = [58,
831]'),
 Text(0.8125, 0.125, 'gini = 0.328\nsamples = 121\nvalue = [25, 96]'),
 Text(0.9375, 0.125, 'gini = 0.082\nsamples = 768\nvalue = [33, 735]')]
```



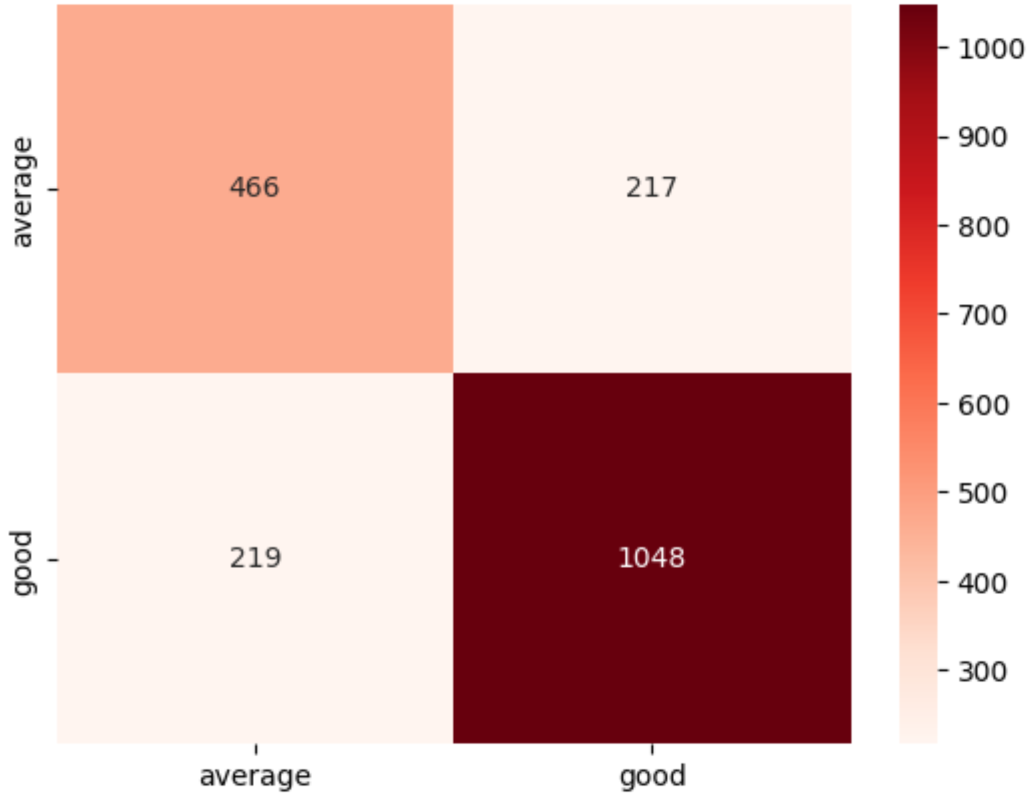
In [420]:
```
prediction = clf.predict(x_test)
prediction_smaller = clf_smaller.predict(x_test)

print("Accuracy on test data set with bigger tree: ", accuracy_score(predictio
```

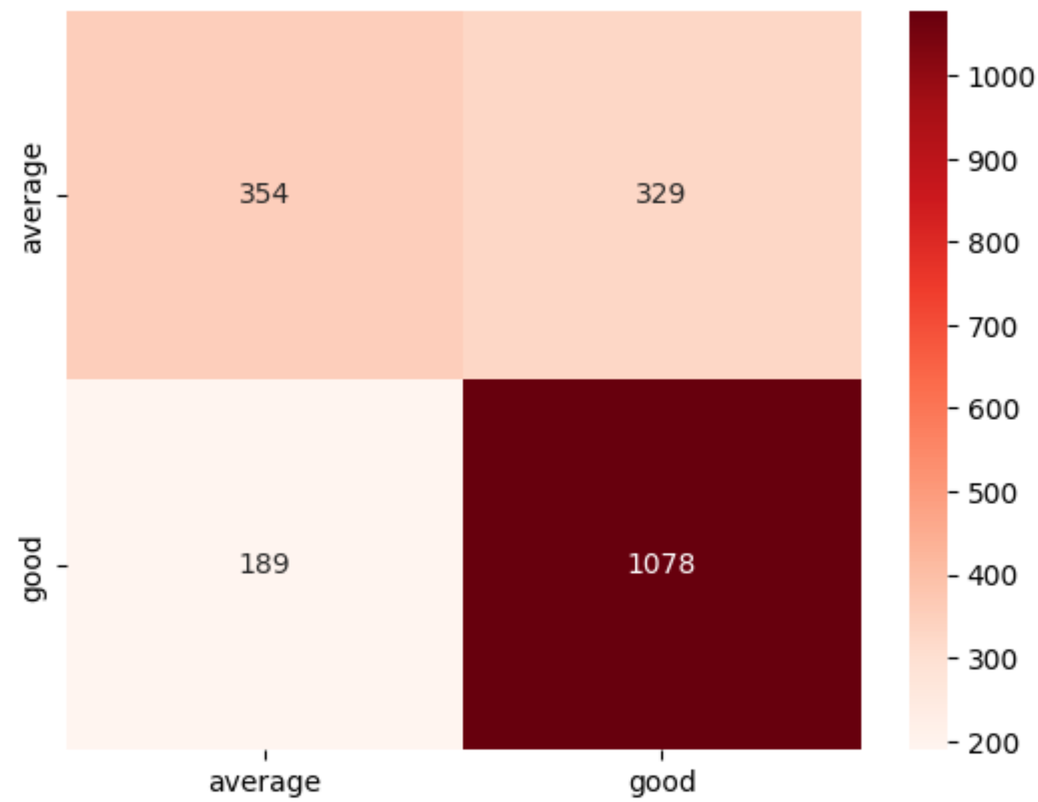
```
Accuracy on test data set with bigger tree:  0.7764102564102564
Accuracy on test data set with smaller tree:  0.7343589743589743
```

As we can see, a deeper decision tree performs better, but it takes more time

In [421]:



In [422]:
```
ax = sns.heatmap(confusion_matrix(y_test, prediction_smaller), annot=True, fmt
```

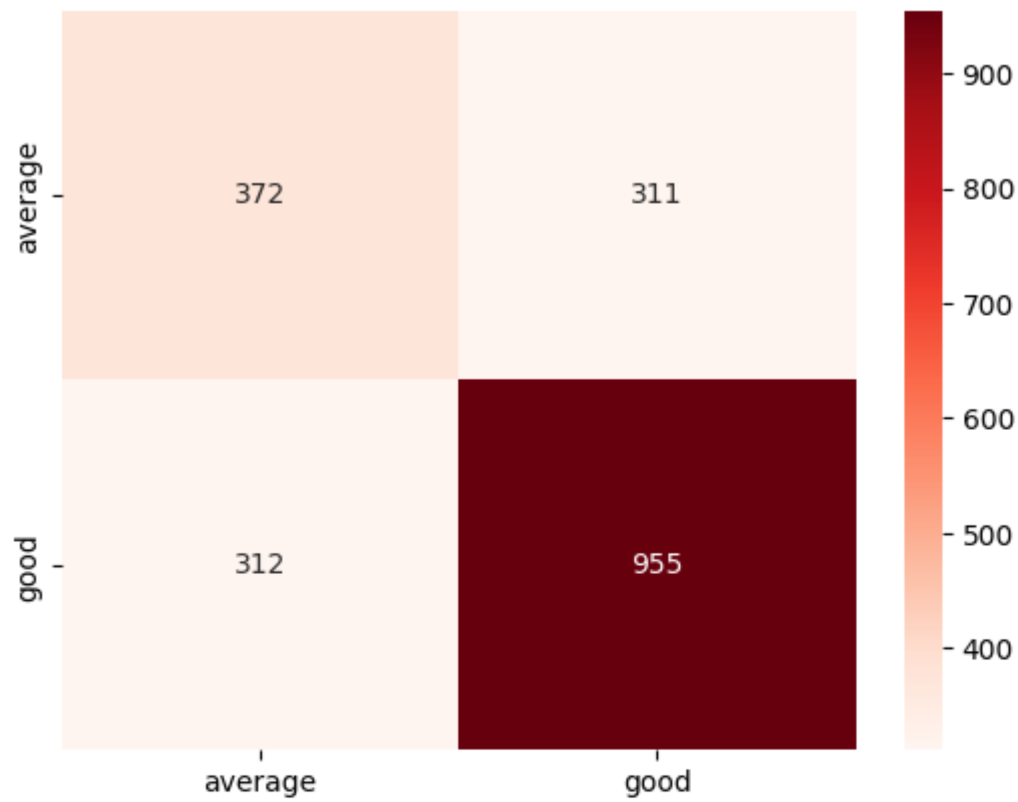


Naive-Bayes

In [423]:
```
model = GaussianNB()
model.fit(x_train, y_train)
```

```
Accuracy on test data set:  0.6805128205128205
```

In [424]:
```
prediction = model.predict(x_test)
```



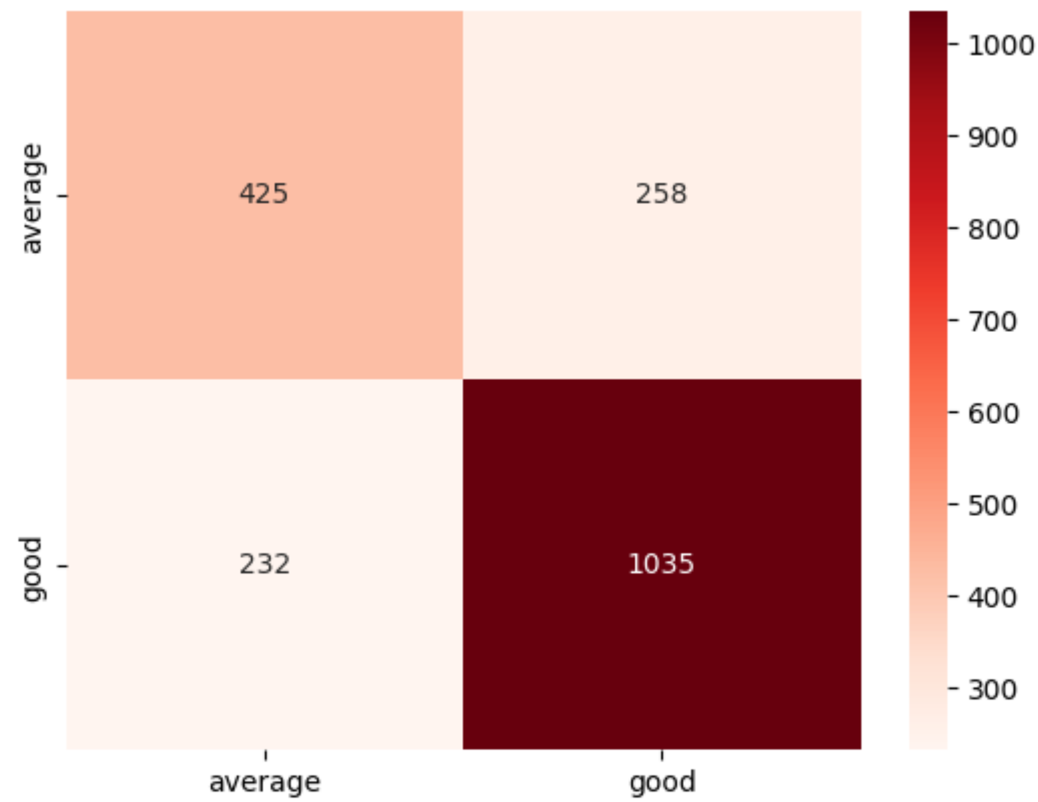
K-nearest neighbors

First try with three neighbors

In [425]:
```
knn = KNeighborsClassifier(n_neighbors=3, metric='euclidean')
knn.fit(x_train, y_train)
```

```
Accuracy on test data set:  0.7487179487179487
```

In [426]:
```
prediction = knn.predict(x_test)
```

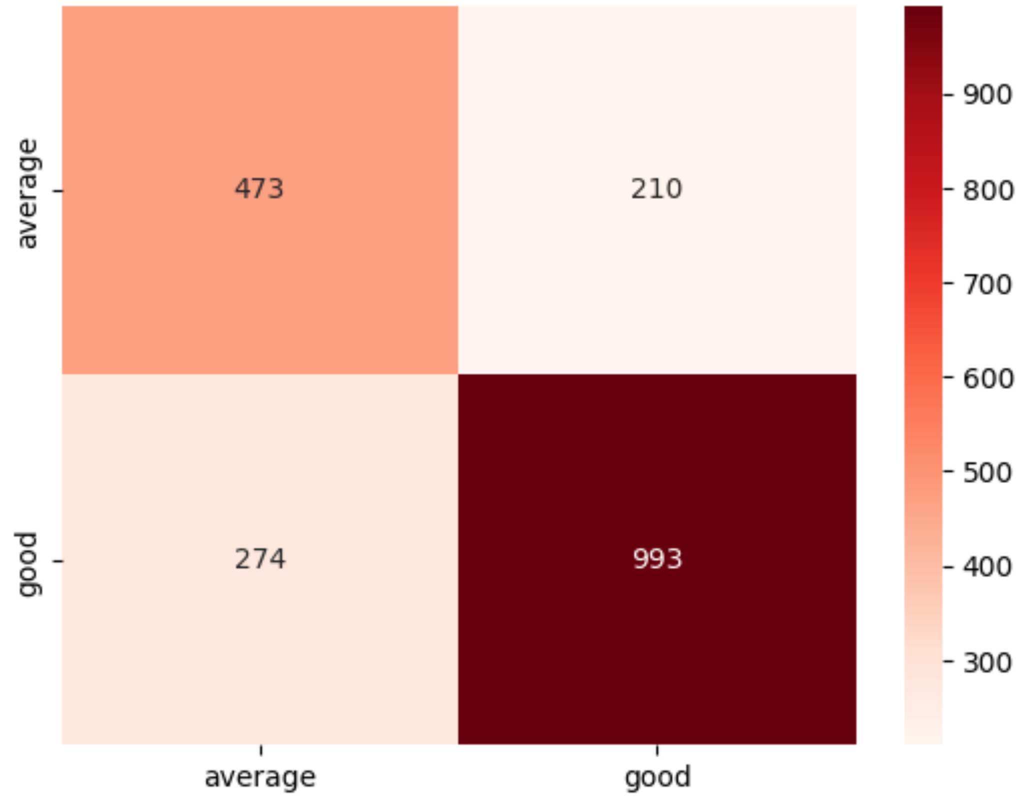


Six neighbors

In [427]:
```
knn = KNeighborsClassifier(n_neighbors=6, metric='euclidean')
knn.fit(x_train, y_train)
```

```
Accuracy on test data set:  0.7517948717948718
```

In [428]:
```
prediction = knn.predict(x_test)
```

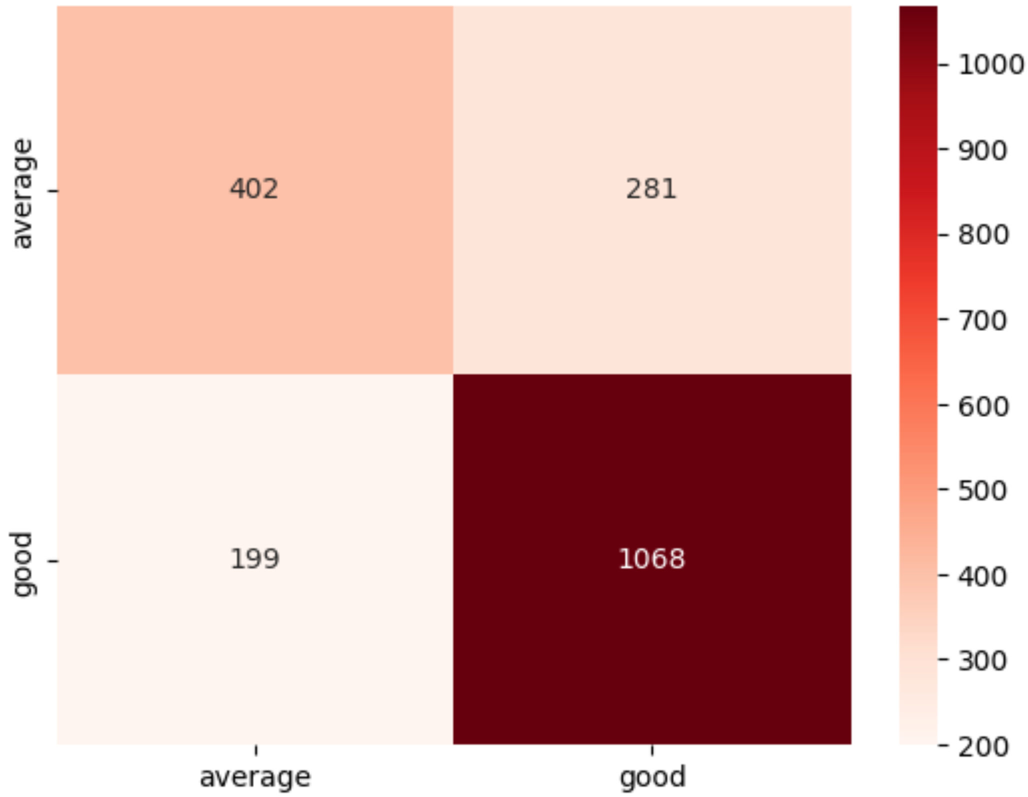


Nine neighbors

In [429]:
```
knn = KNeighborsClassifier(n_neighbors=9, metric='euclidean')
knn.fit(x_train, y_train)
```

```
Accuracy on test data set:  0.7538461538461538
```

In [430]:
```
prediction = knn.predict(x_test)
```

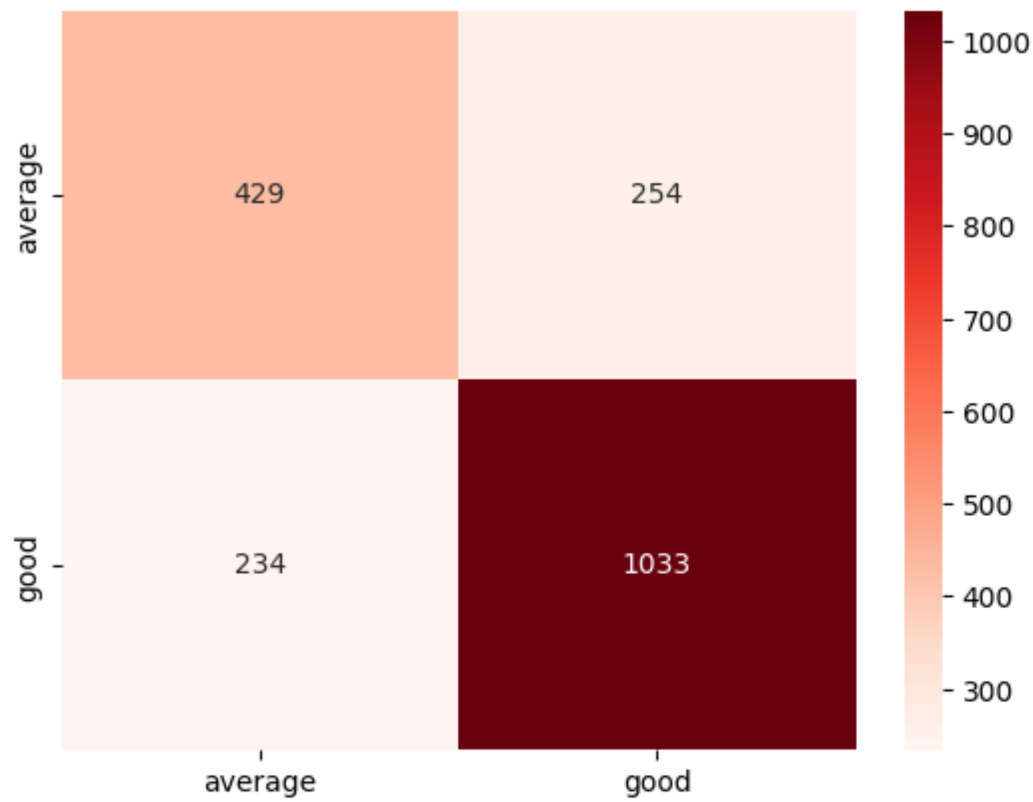


Twelve neighbors

In [431]:
```
knn = KNeighborsClassifier(n_neighbors=12, metric='euclidean')
knn.fit(x_train, y_train)
```

```
Accuracy on test data set:  0.7497435897435898
```

In [432]:
```
prediction = knn.predict(x_test)
```



As we can see, there is no big difference in the results

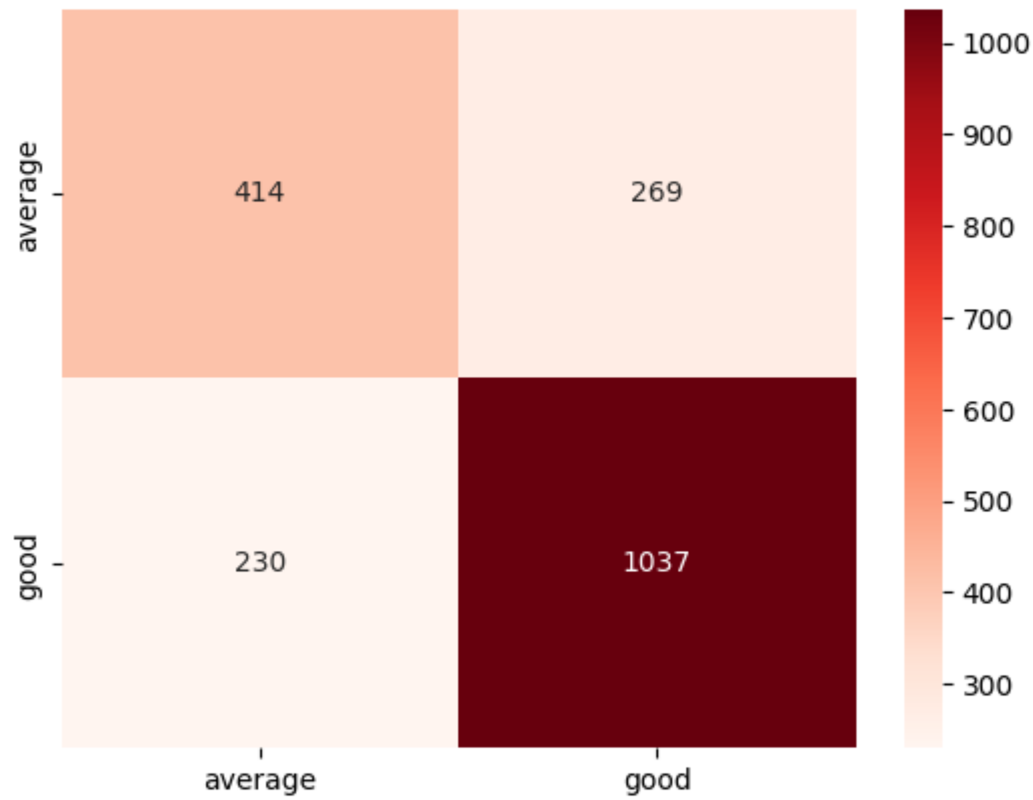
Neural networks

First try with MLPClassifier from sklearn with structure 6, 3, relu activation function, solver adam

In [433]:
```
clf = MLPClassifier(solver='adam', alpha=1e-5, hidden_layer_sizes=(6, 3), rand
clf = clf.fit(x_train, y_train)
prediction = clf.predict(x_test)
```

```
Accuracy on test data set:  0.7441025641025641
```

In [434]:



Preparation of the dataset for neural networks with keras

In [435]:

```
y_train = keras.utils.to_categorical(y_train, num_classes=2)
y_test_to_validation = keras.utils.to_categorical(y_test, num_classes=2)
```

Out[435]:

```
array([[0., 1.],
       [1., 0.],
       [0., 1.],
       [0., 1.],
       [0., 1.]], dtype=float32)
```

First try with 3, 4, 2 structure, tanh and sigmoid activation functions, solver adam

In [436]:

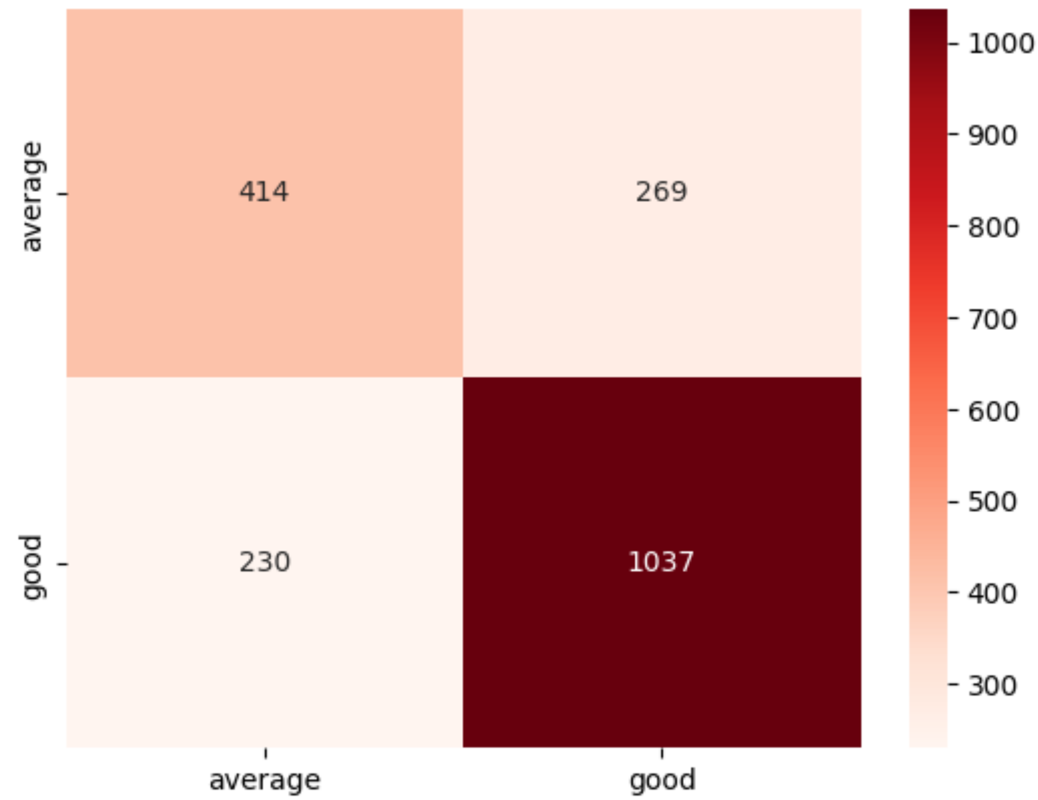
```
model = Sequential()
model.add(Dense(3, activation='tanh', input_dim=(x_train.shape[1])))
model.add(Dense(4, activation='tanh'))
model.add(Dense(2, activation='sigmoid'))

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy
```

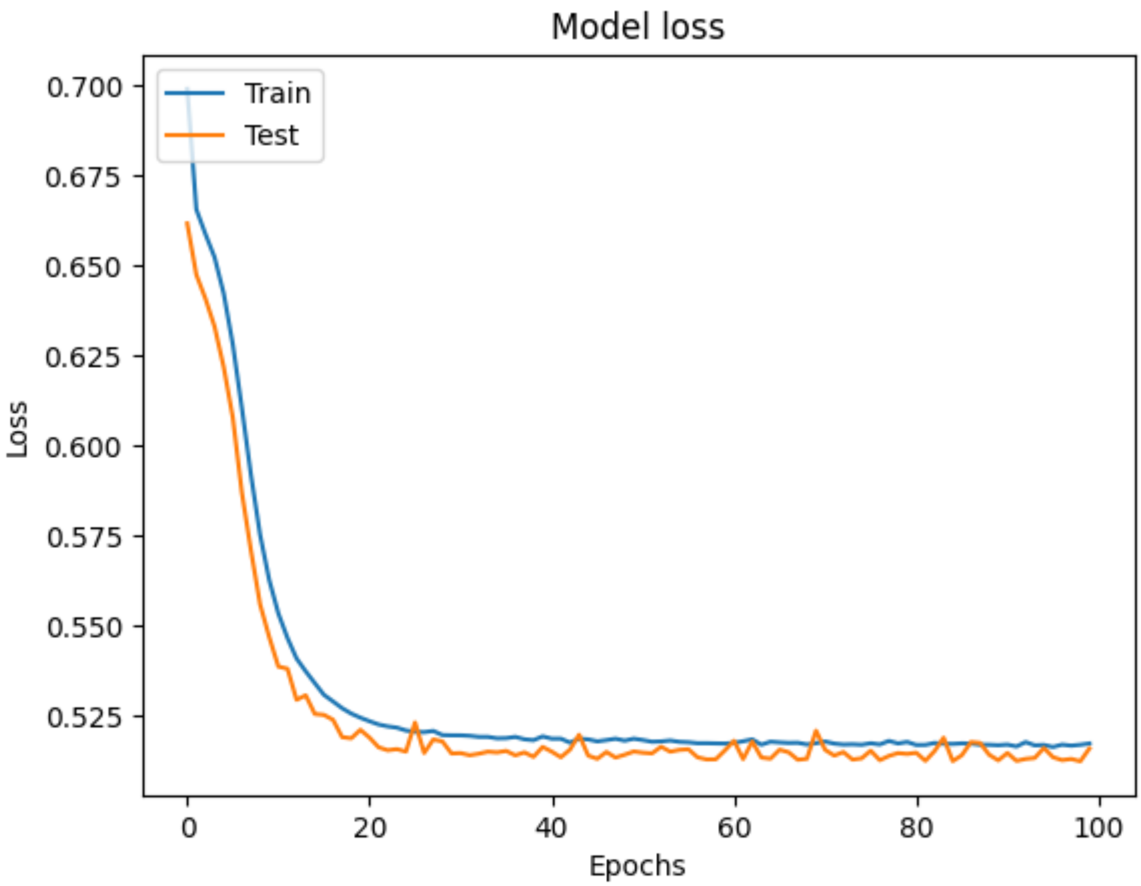
In [437]:
```
pred = model.predict(x_test)
pred = np.argmax(pred, axis=-1)
```

```
61/61 [==============================] - 0s 1ms/step
Accuracy:  0.7384615384615385
```

In [438]:



In [439]:
```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend(['Train', 'Test'], loc='upper left')
```



Second try with 3, 6, 2 structure, elu and sigmoid activation functions, solver adam

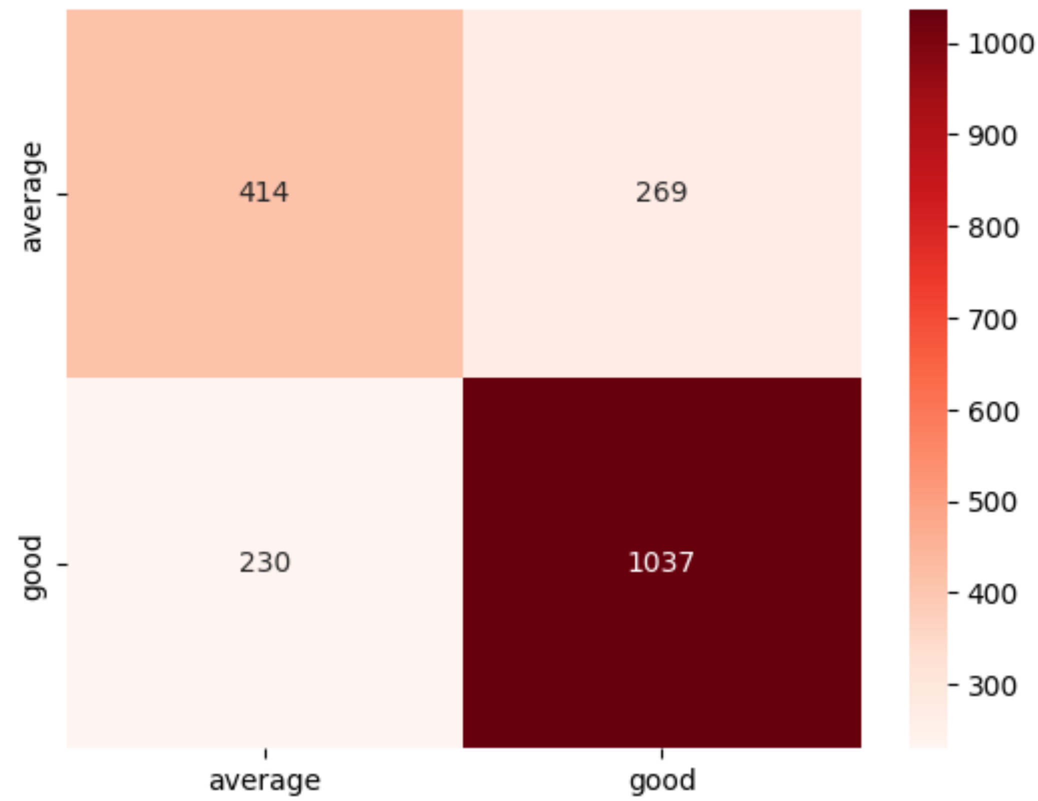
In [440]:
```
model = Sequential()
model.add(Dense(3, activation='elu', input_dim=(x_train.shape[1])))
model.add(Dense(6, activation='elu'))
model.add(Dense(2, activation='sigmoid'))

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy
```

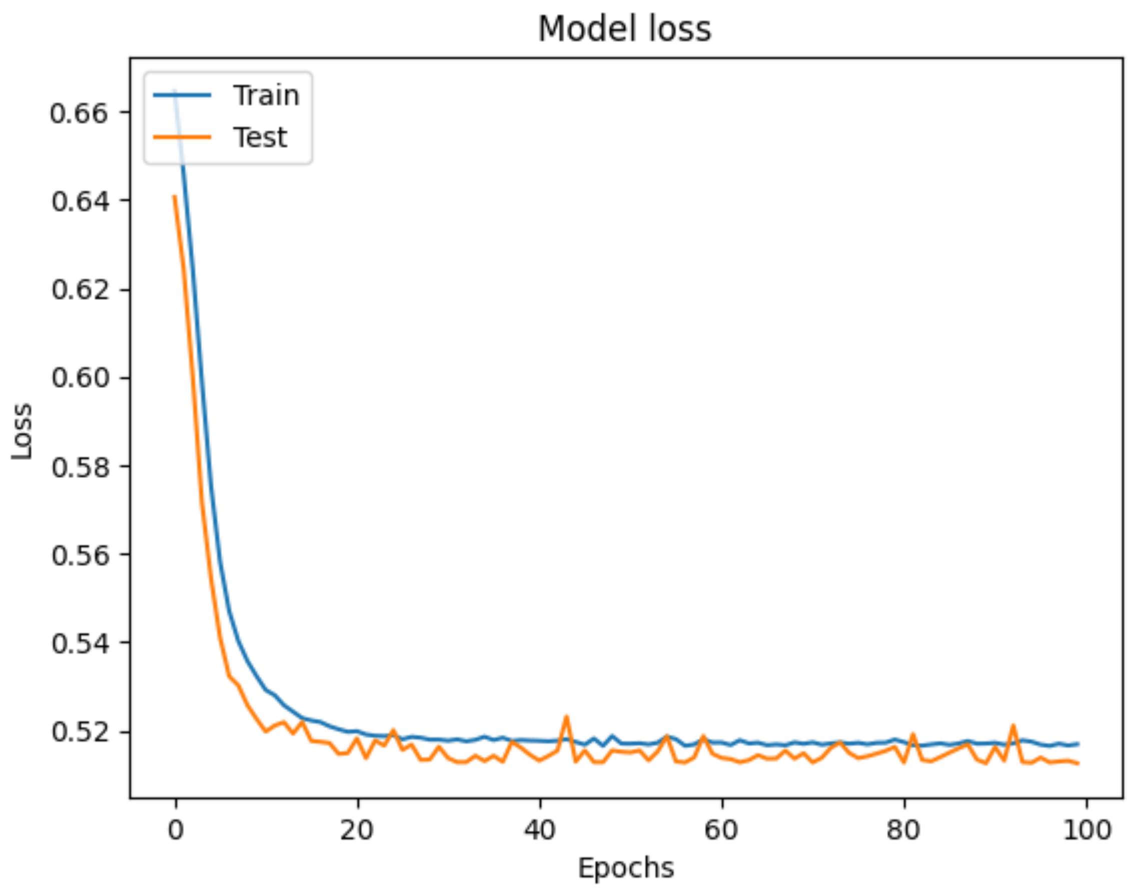
In [441]:
```
pred = model.predict(x_test)
pred = np.argmax(pred, axis=-1)
```

```
61/61 [==============================] - 0s 1ms/step
Accuracy:  0.7492307692307693
```

In [442]:


414
269
230
1037
average
good
average
good
1000
900
800
700
600
500
400
300

In [443]:
```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend(['Train', 'Test'], loc='upper left')
```



Third try with 3, 6, 2 structure, relu and sigmoid activation functions, solver adam

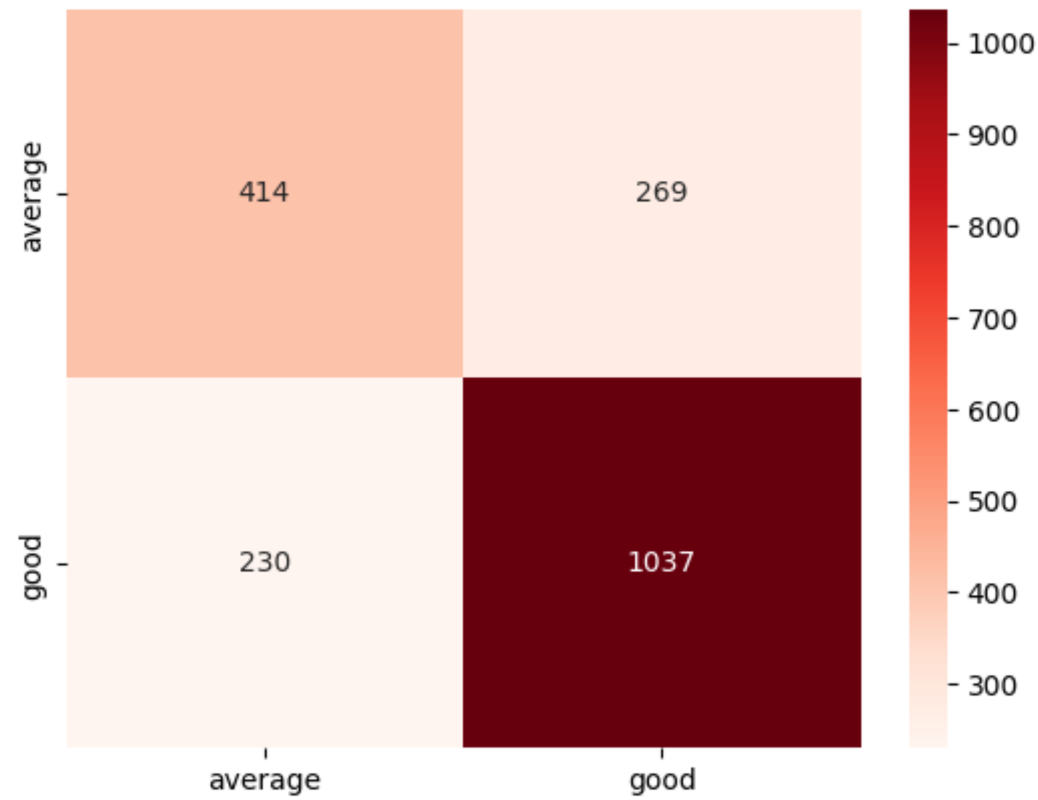
In [444]:
```
model = Sequential()
model.add(Dense(3, activation='relu', input_dim=(x_train.shape[1])))
model.add(Dense(6, activation='relu'))
model.add(Dense(2, activation='sigmoid'))

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy
```

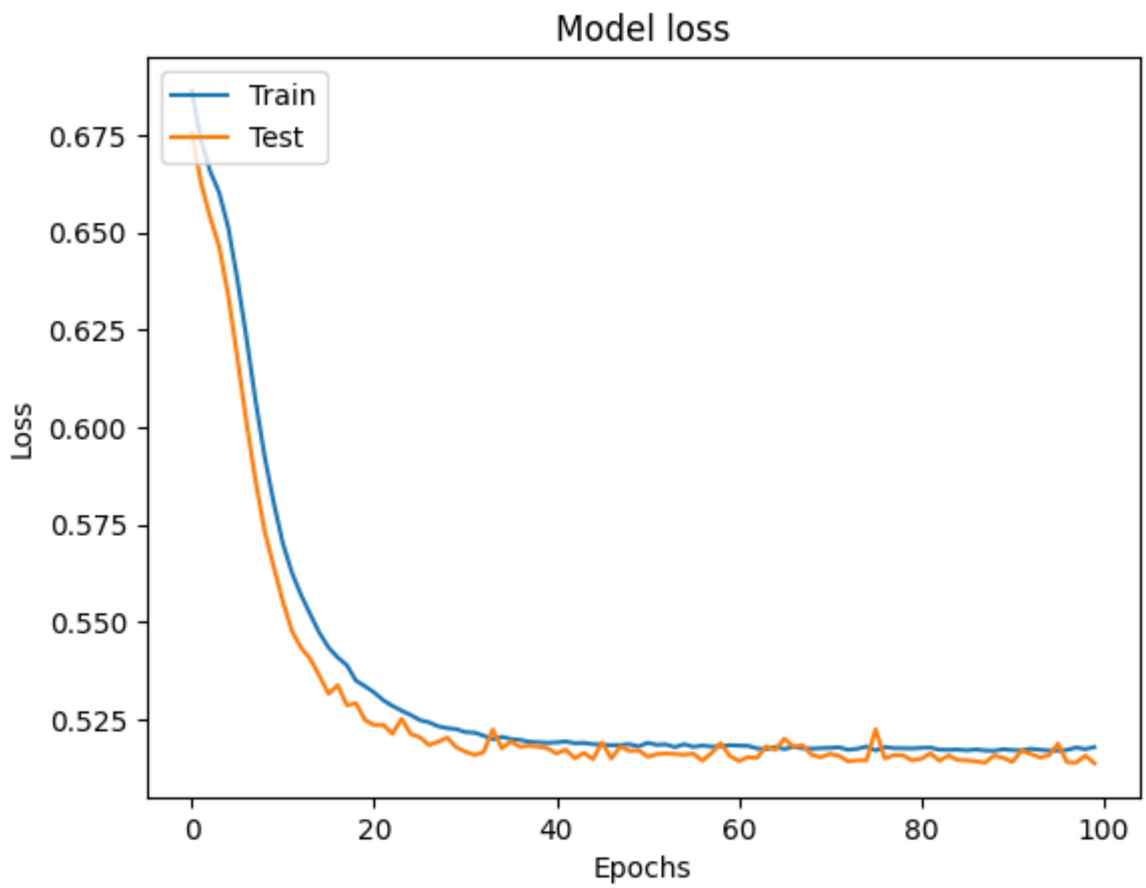
In [445]:
```
pred = model.predict(x_test)
pred = np.argmax(pred, axis=-1)
```

```
61/61 [==============================] - 0s 2ms/step
Accuracy:  0.7502564102564102
```

In [446]:


414
269
230
1037
average
good
average
good
1000
900
800
700
600
500
400
300

In [447]:
```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```



Summary

All tested classifiers obtained similar final results on both processing of the adatset. The results were around 75%, but the highest percentage was obtained by the deep decision tree, which with the second version of the dataset was about 77%. The worst was the naive bayes classifier, which in both cases obtained < 70%

Bibliography

[https://www.kaggle.com/datasets/rajyellow46/wine-quality (https://www.kaggle.com/datasets/rajyellow46/wine-quality)](https://www.kaggle.com/datasets/rajyellow46/wine-quality)