ML4H Project 1

Sven Gutjahr, Christian Bertsch, Richard Danis.

Disclaimer

We have created the report within our code snippets, each question is answered at the end of the section. We tried to be as clean as possible. Our code can be run cell after cell. And should lead to almost the same results (differences might occur different GPU hardware.)

Part 1

In [1]:
```python
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import f1_score, balanced_accuracy_score
from sklearn.tree import DecisionTreeClassifier
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
import torch.optim as optim
import shap
from tqdm import tqdm
from scipy import stats
```

In [2]:
```python
np.random.seed(42)
torch.manual_seed(42)
device = "cpu"
```

Load Data

In [3]:
```python
train_df = pd.read_csv('heart_failure/train_val_split.csv')
y = train_df.loc[:,'HeartDisease']
train_df = train_df.drop(columns=['HeartDisease'])
```

In [4]:
```python
test_df = pd.read_csv('heart_failure/test_split.csv')
y_test = test_df.loc[:,'HeartDisease']
test_df = test_df.drop(columns=['HeartDisease'])
```

Q1 Exploratory Data Analysis

Columns

In [5]:
```python
train_df.columns
```

Out[5]:
```
Index(['Age', 'Sex', 'ChestPainType', 'RestingBP', 'Cholesterol', 'FastingBS',
       'RestingECG', 'MaxHR', 'ExerciseAngina', 'Oldpeak', 'ST_Slope'],
      dtype='object')
```

Checking for Nans

In [6]:
```python
train_df.isnull().sum()
```

Out[6]:
```
Age               0
Sex               0
ChestPainType     0
RestingBP         0
Cholesterol       0
FastingBS         0
RestingECG        0
MaxHR             0
ExerciseAngina    0
Oldpeak           0
ST_Slope          0
dtype: int64
```

As one can see we have no missing data and therefore no data imputation is required.

One-hot encode categorical features

We observed that there are 5 feature columns with string values.
These need to be converted to numerical values in order to use them.
This done by one-hot encoding each of these features.

Sex: M,F
ChestPainType: 'ATA', 'ASY', 'NAP', 'TA'
RestingECG: 'Normal', 'ST', 'LVH'
ExerciseAngina: 'N', 'Y'
ST_Slope: 'Up', 'Flat', 'Down'

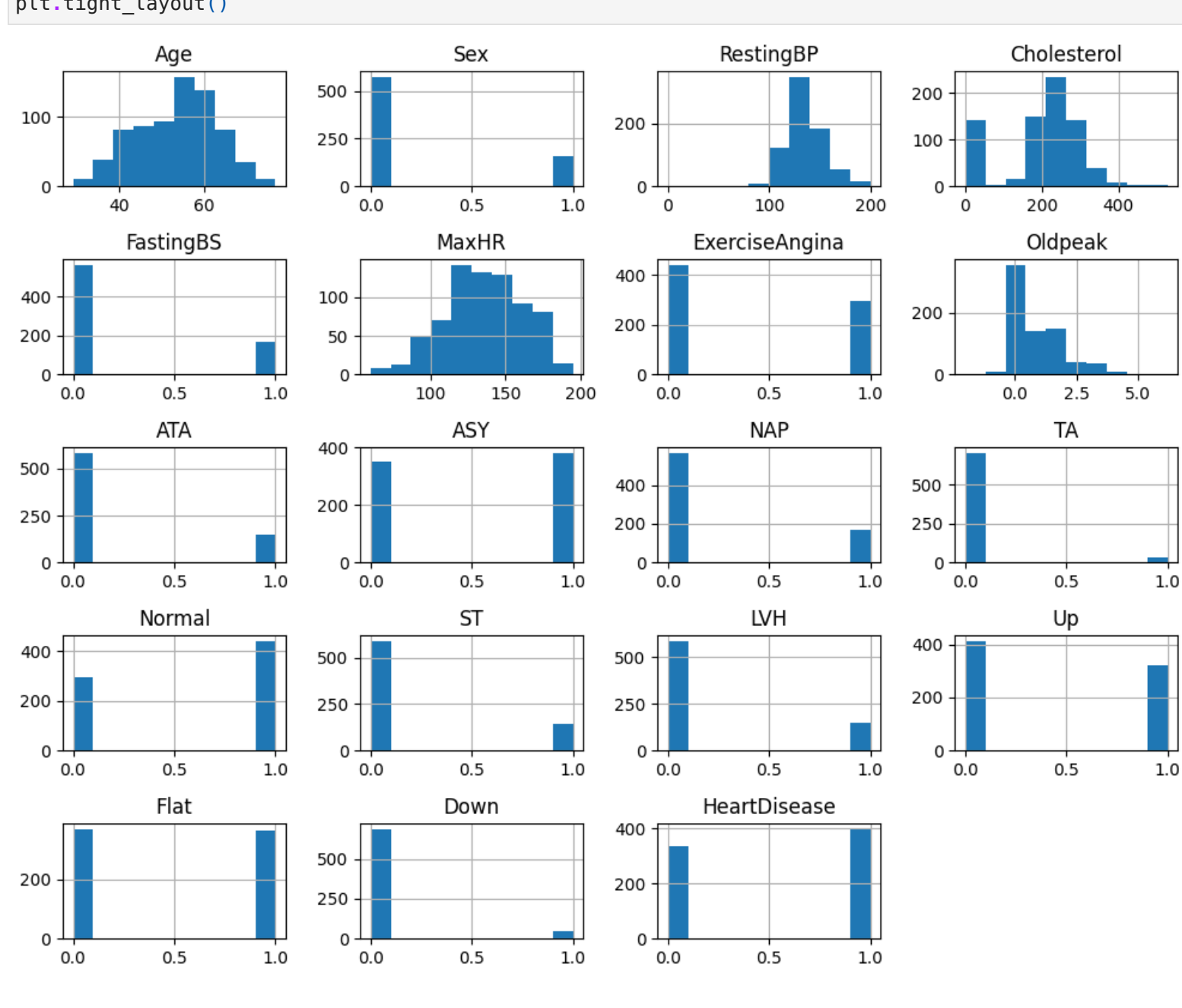
In [7]:
```python
def one_hot(df):

    df = df.replace({'M': 0.0, 'F': 1.0, 'N': 0.0, 'Y': 1.0})

    ata = (df.loc[:,'ChestPainType'] == 'ATA').to_numpy().astype(float)
    asy = (df.loc[:,'ChestPainType'] == 'ASY').to_numpy().astype(float)
    nap = (df.loc[:,'ChestPainType'] == 'NAP').to_numpy().astype(float)
    ta = (df.loc[:,'ChestPainType'] == 'TA').to_numpy().astype(float)

    df['ATA'] = ata
    df['ASY'] = asy
    df['NAP'] = nap
    df['TA'] = ta
    df = df.drop(columns=['ChestPainType'])

    normal = (df.loc[:,'RestingECG'] == 'Normal').to_numpy().astype(float)
    st = (df.loc[:,'RestingECG'] == 'ST').to_numpy().astype(float)
    lvh = (df.loc[:,'RestingECG'] == 'LVH').to_numpy().astype(float)

    df['Normal'] = normal
    df['ST'] = st
    df['LVH'] = lvh
    df = df.drop(columns=['RestingECG'])

    up = (df.loc[:,'ST_Slope'] == 'Up').to_numpy().astype(float)
    flat = (df.loc[:,'ST_Slope'] == 'Flat').to_numpy().astype(float)
    down = (df.loc[:,'ST_Slope'] == 'Down').to_numpy().astype(float)

    df['Up'] = up
    df['Flat'] = flat
    df['Down'] = down
    df = df.drop(columns=['ST_Slope'])

    return df
```

In [8]:
```python
train_df = one_hot(train_df)
test_df = one_hot(test_df)
```

In [9]:
```python
train_df['HeartDisease'] = y
```

Histograms

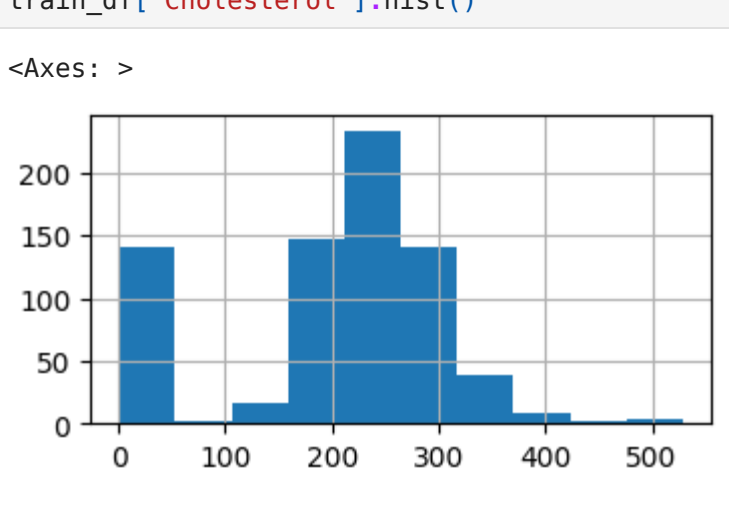
In [10]:
```python
plt.rcParams["figure.figsize"] = (10,8)
hist = train_df.hist()
plt.tight_layout()
```



One can observe nonsensical data in the feature cholesterol which shows many zero-value entries. Since Cholesterol is a very important bio-molecule/metabolite, having many vital functions in physiological processes, zero-values make no sense. In order to not change the shape of the distribution, we set the zero-values to the mean of our dataset. Furthermore, there is no significant class imbalance within the HeartDisease column as one can clearly see in our visualized histogram. Therefore, we do not have to take care of this potential problem.

In [11]:
```python
plt.rcParams["figure.figsize"] = (4,2)
train_df['Cholesterol'].hist()
```

Out[11]: `<Axes: >`

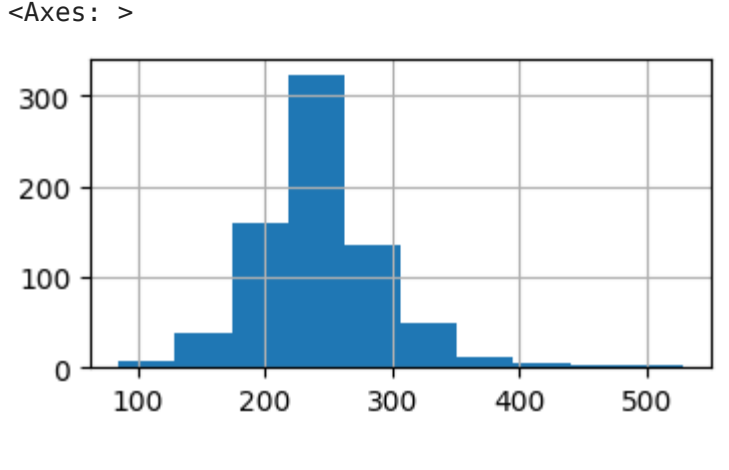


Fixing Cholesterol Distribution

In [12]:
```python
chol_mean = train_df['Cholesterol'].replace(0, np.NaN).mean()
chol_std = train_df['Cholesterol'].replace(0, np.NaN).std()
train_df['Cholesterol'] = train_df['Cholesterol'].replace(0, chol_mean)
test_df['Cholesterol'] = test_df['Cholesterol'].replace(0, chol_mean)
```

In [13]:
```python
plt.rcParams["figure.figsize"] = (4,2)
train_df['Cholesterol'].hist()
```

Out[13]: `<Axes: >`



Outlier removal

There are 5 distribution which are roughly normal.
We use the Z-Score to remove outliers in these distributions, that is we remove any sample with a Z-Score above 3 in any column.

In [14]:
```python
names = ['Age', 'RestingBP', 'Cholesterol', 'MaxHR', 'Oldpeak']
old_shape = train_df.shape
y = y[(np.abs(stats.zscore(train_df[names])) < 3).all(axis=1)]
train_df = train_df[(np.abs(stats.zscore(train_df[names])) < 3).all(axis=1)]
print("Total samples removed", old_shape[0] - train_df.shape[0])
```

```
Total samples removed 23
```

Normalizing

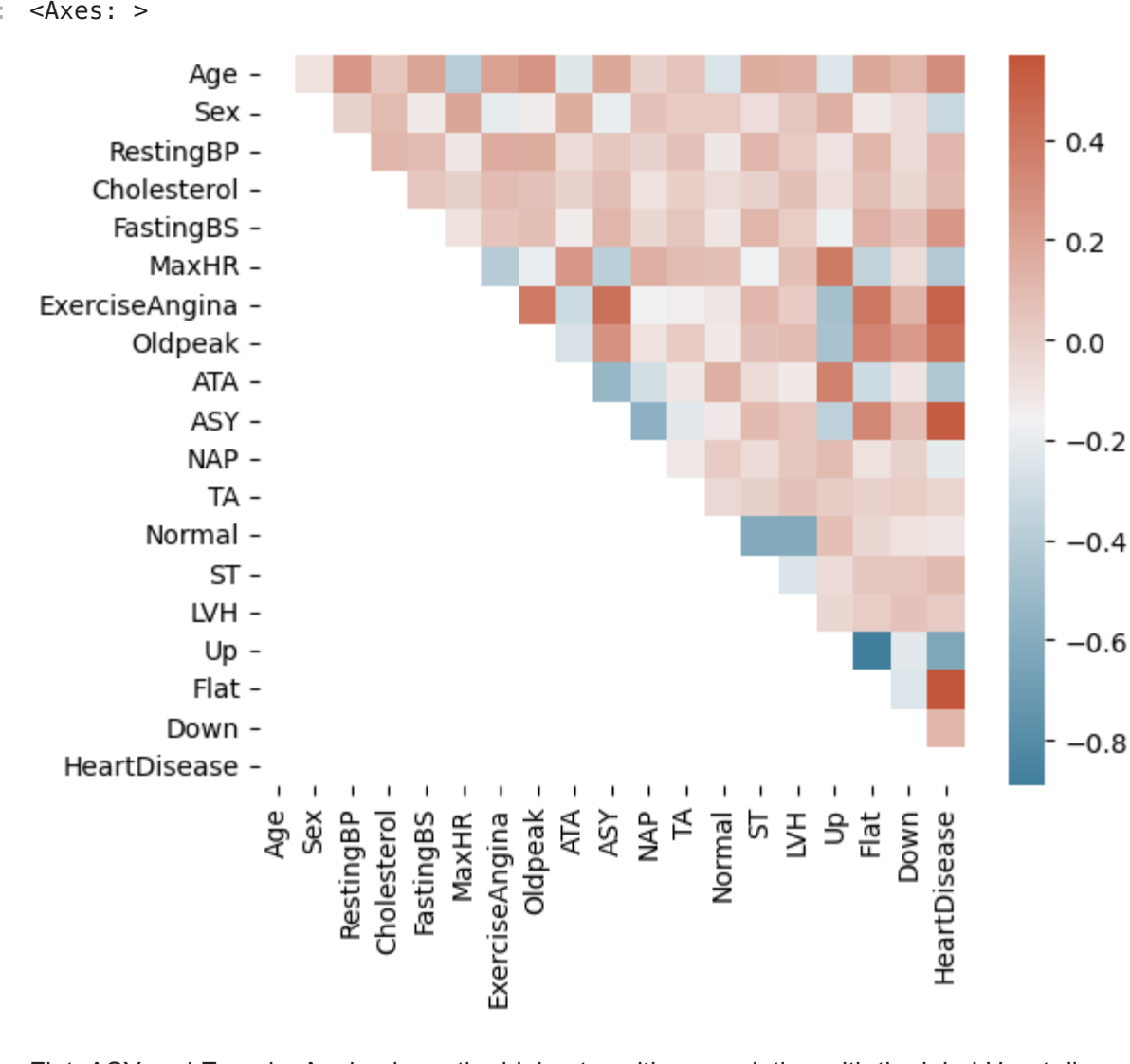
We normalize the non-categorical features Age, RestingBP, Cholesterol, MaxHR and Oldpeak.

In [15]:
```python
for name in names:
    if name == 'Cholesterol':
        test_df[name] = (test_df[name] - train_df[name].mean()) / chol_std
        train_df[name] = (train_df[name] - train_df[name].mean()) / chol_std
    else:
        test_df[name] = (test_df[name] - train_df[name].mean()) / train_df[name].std()
        train_df[name] = (train_df[name] - train_df[name].mean()) / train_df[name].std()
```

Correlation Matrix

In [16]:
```python
plt.rcParams["figure.figsize"] = (6,5)
mask = np.tril(np.ones_like(train_df.corr(), dtype=bool))
cmap = sns.diverging_palette(230, 20, as_cmap=True)
sns.heatmap(train_df.corr(), mask=mask, cmap=cmap)
```

Out[16]: `<Axes: >`



Flat, ASY and ExerciseAngina have the highest positive correlation with the label Heart disease within our correlation matrix. On the other hand, one can rarely observe negative correlated features with Up having the strongest negative correlation within the matrix. As a result, these features may be important for linear prediction of heart disease.

In [17]:
```python
train_df = train_df.drop(columns=['HeartDisease'])
```

Q2 Logistic Regression

What preprocessing step is crucial to ensure comparability of feature coefficients?

Normalizing as done in Q1 is crucial, as otherwise the feature values have different magnitudes and therefore the feature coefficients will have different magnitudes as well.

Fit Classifier

In [18]:
```python
log_clf = LogisticRegression(penalty='l1', solver='liblinear').fit(train_df, y)
```

Evaluate Metrics

In [19]:
```python
y_pred = log_clf.predict(test_df)
```

F1 Score

In [20]:
```python
f1_score(y_test, y_pred)
```

Out[20]: `0.8634361233480177`

Balanced Accuracy

In [21]:
```python
balanced_accuracy_score(y_test, y_pred)
```

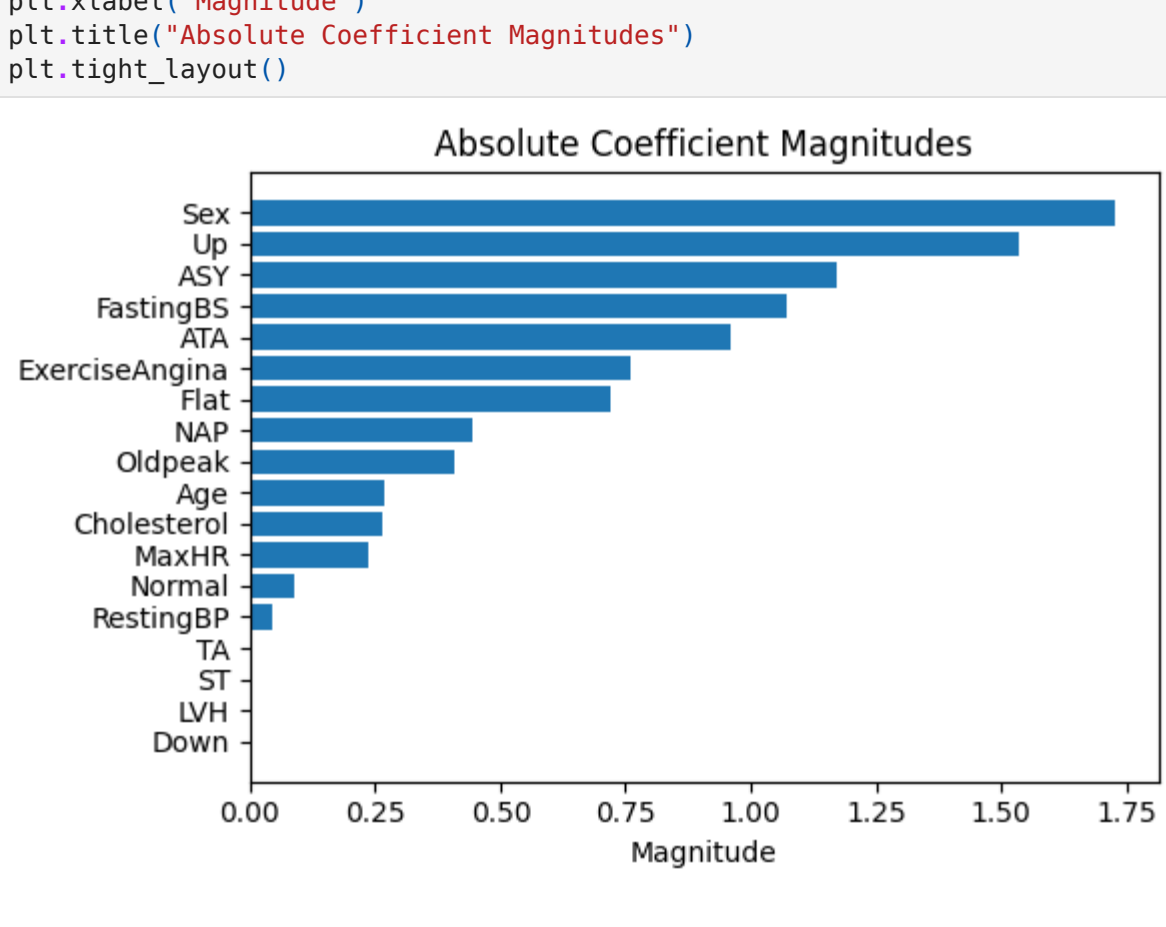
Out[21]: `0.817076167076167`

Plot absolute magnitude of feature coefficients

In order to visualize the importance of different features, we plot the absolute values of the corresponding coefficients ordered by magnitude.

In [22]:
```python
log_coef = np.abs(np.squeeze(log_clf.coef_))
sorted_indices = np.argsort(log_coef)
```

In [23]:
```python
plt.rcParams["figure.figsize"] = (6,4)
plt.barh(train_df.columns[sorted_indices], log_coef[sorted_indices])
plt.xlabel("Magnitude")
plt.title("Absolute Coefficient Magnitudes")
plt.tight_layout()
```



Finally, argue for or against fitting a logistic regression using only the important variables, as determined by the Lasso model, to arrive at the final coefficients instead of keeping the coefficients of the Lasso model.

The Lasso model only captures linear dependencies of the labels on the input features. Discarded features with small coefficients could have strong interactions with other features, but those too go undetected. Methods such as random forest importance are more suited for this task

Q3 Decision Tree

Fit Classifier

In [24]:
```python
tree_clf = DecisionTreeClassifier().fit(train_df, y)
```

Evaluate Metrics

In [25]:
```python
y_pred = tree_clf.predict(test_df)
```

F1 Score

In [26]:
```python
f1_score(y_test, y_pred)
```

Out[26]: `0.8090909090909091`

Balanced Accuracy

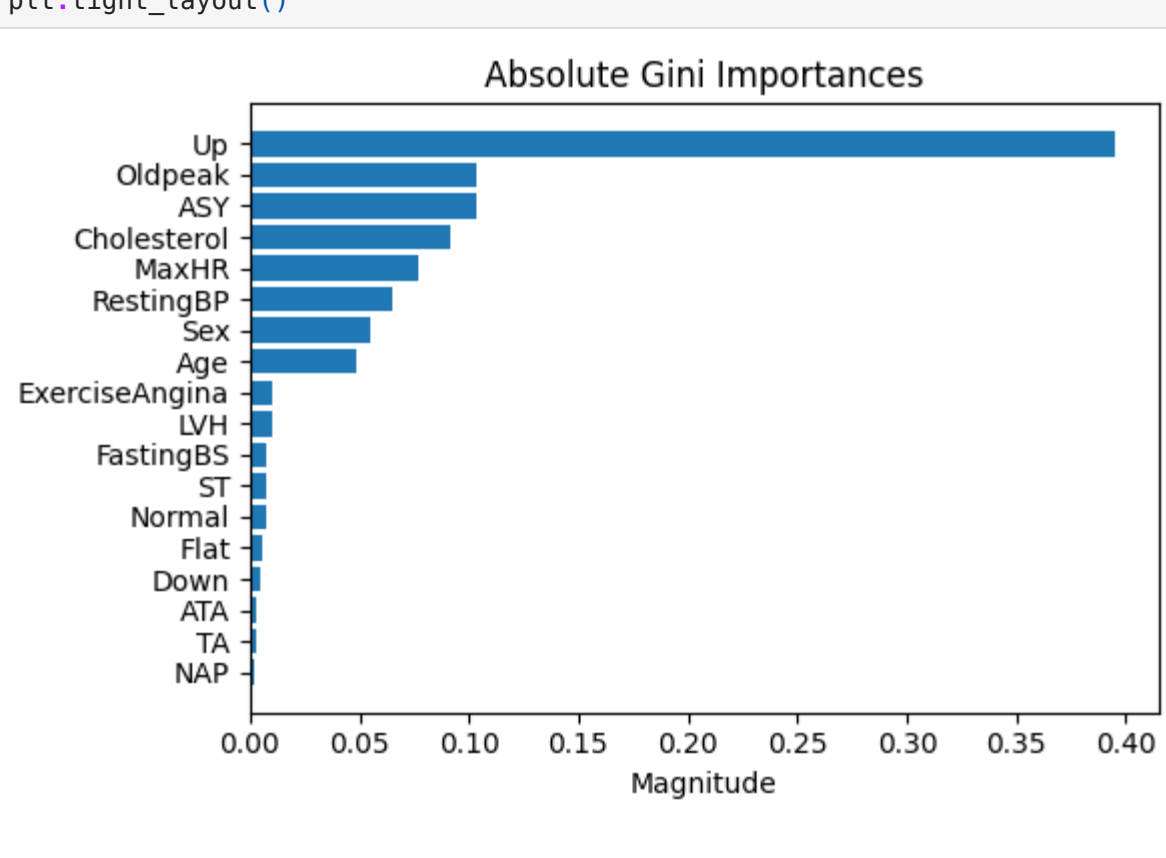
In [27]:
```python
balanced_accuracy_score(y_test, y_pred)
```

Out[27]: `0.7626535626535627`

Plot Gini Importances

We plot the sorted gini importances.

In [28]:
```python
sorted_indices = np.argsort(tree_clf.feature_importances_)
gini_importances = tree_clf.feature_importances_[sorted_indices]
plt.rcParams["figure.figsize"] = (6,4)
plt.barh(train_df.columns[sorted_indices], gini_importances)
plt.title("Absolute Gini Importances")
plt.xlabel("Magnitude")
plt.tight_layout()
```



Q4 Multi-Layer Perceptron

Defining a 2 layer Neural Network

We use a simple 2 layer Neural Network with a ReLU activation function between the two layers.

In [29]:
```python
class MLP(nn.Module):

    def __init__(self, dim=18):

        super().__init__()

        self.layers = nn.Sequential(
            nn.Linear(dim, dim),
            nn.ReLU(),
            nn.Linear(dim, 1)
        )

    def forward(self, x):

        return self.layers(x)
```

Dataset class for loading batches

In [30]:
```python
class HeartDataset(Dataset):

    def __init__(self, features, labels):

        self.features = features
        self.labels = labels

    def __len__(self):
        return self.features.shape[0]

    def __getitem__(self, idx):

        return torch.tensor(self.features[idx], dtype=torch.float32), torch.tensor(self.labels[idx]
```

Split into train and validation datasets

For the validation we create an 80-20 train-val data split.

In [31]:
```python
X_train, X_val, y_train, y_val = train_test_split(train_df, y, test_size=0.2, random_state=42)
```

DataLoader

In [32]:
```python
train_dataset = HeartDataset(X_train.values, y_train.values)
full_dataset = HeartDataset(train_df.values, y.values)
val_x = torch.tensor(X_val.values, dtype=torch.float32)
val_y = torch.tensor(y_val.values, dtype=torch.float32)
```

In [33]:
```python
full_dataloader = DataLoader(full_dataset, batch_size=32, shuffle=True, num_workers=0)
train_dataloader = DataLoader(train_dataset, batch_size=32, shuffle=True, num_workers=0)
```

Loss Function

In [34]:
```python
loss_fn = nn.BCEWithLogitsLoss()
```

Validation

In [35]:
```python
def train_mlp(model, epochs, dataloader, val=True):

    optimizer = optim.Adam(model.parameters(),
                           lr=5e-3)

    for epoch in range(epochs):

        train_loss_cum = 0
        cum = 0

        for x,y in dataloader:

            optimizer.zero_grad()
            out = torch.squeeze(model(x))
            loss = loss_fn(out, y)
            loss.backward()
            optimizer.step()

            train_loss_cum += x.shape[0] * loss
            cum += x.shape[0]

        if val:

            with torch.no_grad():

                out = torch.squeeze(model(val_x))
                val_loss = loss_fn(out, val_y)

        train_loss = train_loss_cum/cum

        if epoch % 5 == 0:

            if val:

                print(f'Epoch {epoch} | '
                      f'Train loss: {train_loss:.4f} | '
                      f'Val loss: {val_loss:.4f}')

            else:

                print(f'Epoch {epoch} | '
                      f'Train loss: {train_loss:.4f}')
```

In [36]:
```python
mlp = MLP()
train_mlp(mlp, epochs=31, dataloader=train_dataloader)
```

```
Epoch 0 | Train loss: 0.6161 | Val loss: 0.5032
Epoch 5 | Train loss: 0.3068 | Val loss: 0.2812
Epoch 10 | Train loss: 0.2815 | Val loss: 0.2801
Epoch 15 | Train loss: 0.2656 | Val loss: 0.2829
Epoch 20 | Train loss: 0.2480 | Val loss: 0.2929
Epoch 25 | Train loss: 0.2368 | Val loss: 0.3050
Epoch 30 | Train loss: 0.2254 | Val loss: 0.3135
```

Train on full training dataset

In [37]:
```python
mlp = MLP()
```

In [38]:
```python
train_mlp(mlp, epochs=11, dataloader=full_dataloader, val=False)
```

```
Epoch 0 | Train loss: 0.5946
Epoch 5 | Train loss: 0.2990
Epoch 10 | Train loss: 0.2793
```

Evaluate Metrics

In [39]:
```python
test_x = torch.tensor(test_df.values, dtype=torch.float32)
```

In [40]:
```python
with torch.no_grad():
    y_pred = torch.sigmoid(mlp(test_x)).numpy().squeeze()
```

In [41]:
```python
y_pred = np.round(y_pred)
```

F1 Score

In [42]:
```python
f1_score(y_test, y_pred)
```

Out[42]: `0.8634361233480177`

Balanced Accuracy

In [43]:
```python
balanced_accuracy_score(y_test, y_pred)
```

Out[43]: `0.817076167076167`

Shap values

In [44]:
```python
train_x = torch.tensor(X_train.values, dtype=torch.float32)
```

Append Sigmoid to the model to get shap values with respect to the probability values.

In [45]:
```python
mlp = nn.Sequential(mlp, nn.Sigmoid())
explainer = shap.DeepExplainer(mlp, train_x)
```

Take the first 4 negative and first 4 positive samples

In [46]:
```python
positive = np.where(y_pred == 1)[0][:4]
negative = np.where(y_pred == 0)[0][:4]
```

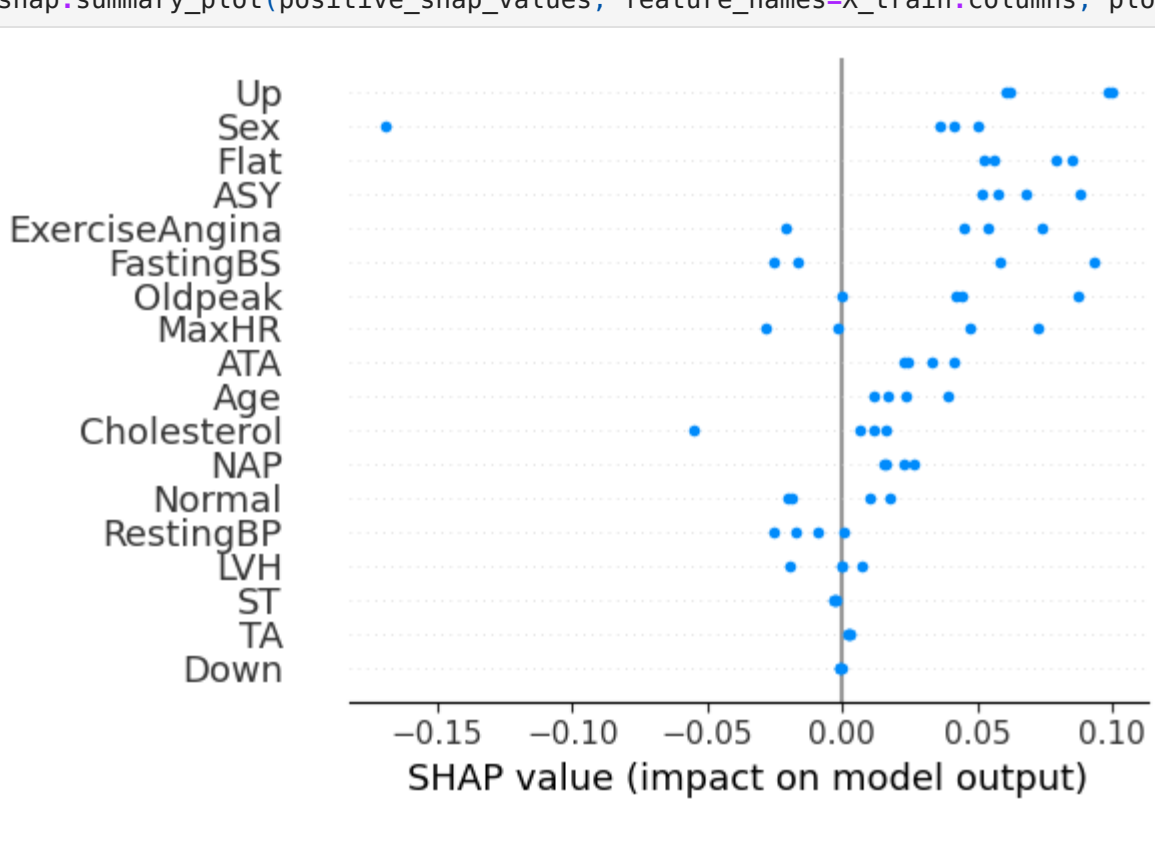
In [47]:
```python
positive_x = test_x[positive]
negative_x = test_x[negative]
```

Shapley Plot for 4 positive samples

In [48]:
```python
positive_shap_values = explainer.shap_values(positive_x)
```

```
Using a non-full backward hook when the forward contains multiple autograd Nodes is deprecated and
will be removed in future versions. This hook will be missing some grad_input. Please use register
_full_backward_hook to get the documented behavior.
```

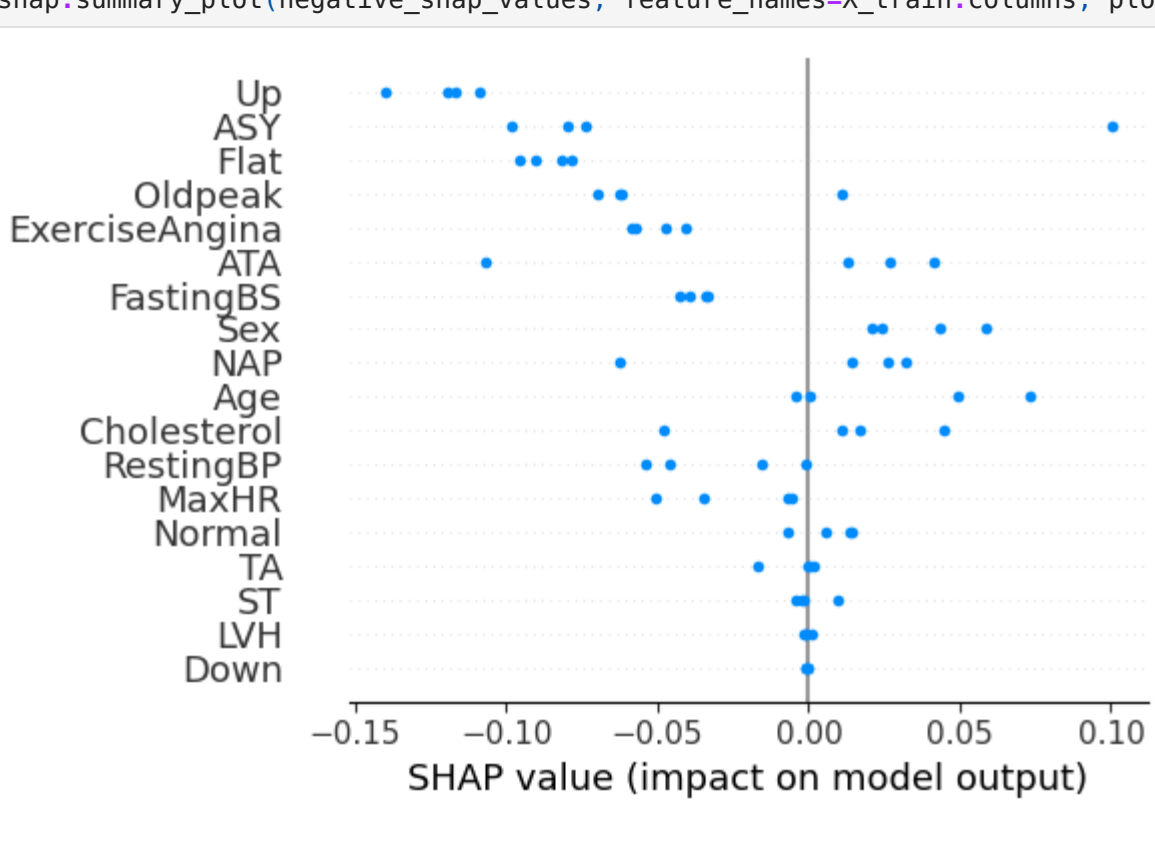
In [49]:
```python
shap.summary_plot(positive_shap_values, feature_names=X_train.columns, plot_size=(6,4))
```



Shapley Plot for 4 negative values

In [50]:
```python
negative_shap_values = explainer.shap_values(negative_x)
```

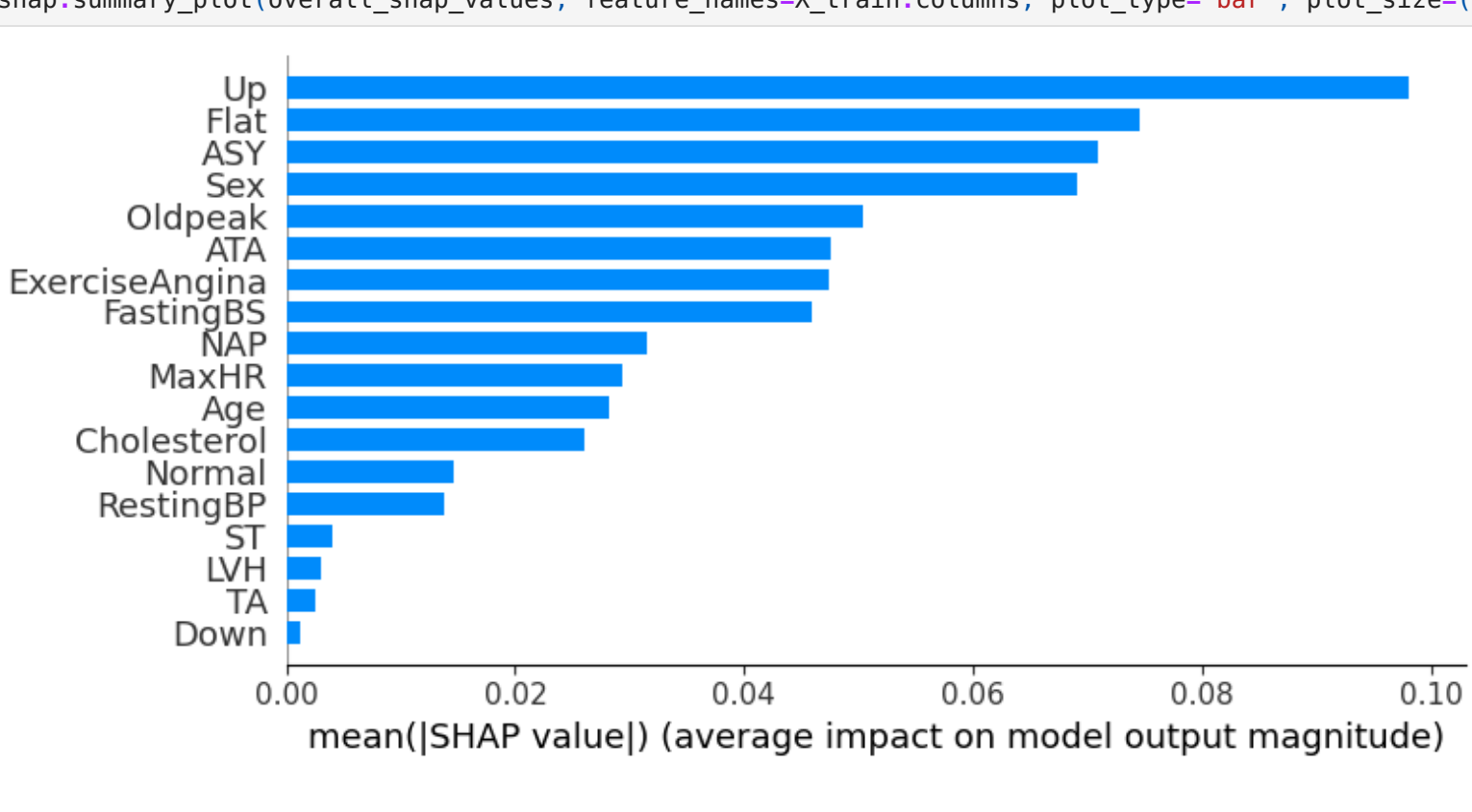
In [51]:
```python
shap.summary_plot(negative_shap_values, feature_names=X_train.columns, plot_size=(6,4))
```



Shapley Plot for all train samples

In [52]:
```python
overall_shap_values = explainer.shap_values(train_x)
```

In [53]:
```python
shap.summary_plot(overall_shap_values, feature_names=X_train.columns, plot_type='bar', plot_size=(8
```



Are feature importances consistent across different predictions and compared to overall importance values? Elaborate on your findings!

The feature importances are mostly consistent, that is most of them are positive for the positive samples and negative for the negative samples. But we do notice some outliers. For example Sex has a highly negative impact for one of the positive samples and ASY has a highly positive impact for one of the negative examples. Considering their absolute magnitude on the other hand, both of these outliers are in line with the overall absolute magnitude

Challenge 1: Neural Additive Models

In [54]:
```python
class FeatureNN(nn.Module):

    def __init__(self):

        super().__init__()
        self.l1 = nn.Linear(1, 64)
        self.l2 = nn.Linear(64, 64)
        self.l3 = nn.Linear(64, 32)
        self.l4 = nn.Linear(32, 1)
        self.ReLU = nn.ReLU()

    def forward(self, x):

        out = self.l1(x)
        out = self.ReLU(out)
        out = self.l2(out)
        out = self.ReLU(out)
        out = self.l3(out)
        out = self.ReLU(out)
        out = self.l4(out)

        return self.ReLU(out)


class NAM(nn.Module):

    def __init__(self, dim=18):

        super().__init__()

        self.fnns = nn.ModuleList([
            FeatureNN() for i in range(dim)
        ])

        self.dim = dim
        self.feature_out = None
        self.bias = torch.nn.Parameter(data=torch.zeros(1))

    def forward(self, x):

        out = torch.empty((x.shape[0], self.dim, 1), dtype=torch.float32)
        for i in range(self.dim):
            a = torch.unsqueeze(x[:,i], dim=1)
            a = self.fnns[i](a)
            out[:, i] = a

        self.feature_out = out

        return torch.sum(out, dim=1) + self.bias
```

In [55]:
```python
nam = NAM()
```

Loss Function and Optimizer

In [56]:
```python
def featureLoss(x):

    return torch.mean(torch.mean(torch.square(x), dim=0))
```

In [57]:
```python
loss_fn = nn.BCEWithLogitsLoss()
optimizer = optim.Adam(nam.parameters(),
                       lr=1e-3, weight_decay=1e-6)
output_penalty = 1e-3
```

In [58]:
```python
full_dataloader = DataLoader(full_dataset, batch_size=32, shuffle=True, num_workers=0)
train_dataloader = DataLoader(train_dataset, batch_size=32, shuffle=True, num_workers=0)
```

Validation

In [59]:
```python
epochs = 51

for epoch in range(epochs):

    train_loss_cum = 0
    cum = 0

    for x,y in train_dataloader:

        optimizer.zero_grad()
        out = nam(x)
        out = torch.squeeze(out)
        loss = loss_fn(out, y) + output_penalty * featureLoss(nam.feature_out)
        loss.backward()
        optimizer.step()

        train_loss_cum += x.shape[0] * loss
        cum += x.shape[0]

    with torch.no_grad():

        out = nam(val_x)
        out = torch.squeeze(out)
        val_loss = loss_fn(out, val_y)

    train_loss = train_loss_cum/cum

    if epoch % 1 == 0:

        print(f'Epoch {epoch} | '
              f'Train loss: {train_loss:.4f} | '
              f'Val loss: {val_loss:.4f}')
```

```
Epoch 0 | Train loss: 0.6573 | Val loss: 0.5725
Epoch 1 | Train loss: 0.5701 | Val loss: 0.5351
Epoch 2 | Train loss: 0.5574 | Val loss: 0.5274
Epoch 3 | Train loss: 0.5502 | Val loss: 0.5222
Epoch 4 | Train loss: 0.5476 | Val loss: 0.5159
Epoch 5 | Train loss: 0.5435 | Val loss: 0.5140
Epoch 6 | Train loss: 0.5399 | Val loss: 0.5127
Epoch 7 | Train loss: 0.5386 | Val loss: 0.5088
Epoch 8 | Train loss: 0.5352 | Val loss: 0.5077
Epoch 9 | Train loss: 0.5324 | Val loss: 0.5026
Epoch 10 | Train loss: 0.5308 | Val loss: 0.5008
Epoch 11 | Train loss: 0.5284 | Val loss: 0.4978
Epoch 12 | Train loss: 0.5274 | Val loss: 0.4978
Epoch 13 | Train loss: 0.5234 | Val loss: 0.4928
Epoch 14 | Train loss: 0.5239 | Val loss: 0.4897
Epoch 15 | Train loss: 0.5207 | Val loss: 0.4909
Epoch 16 | Train loss: 0.5179 | Val loss: 0.4869
Epoch 17 | Train loss: 0.5161 | Val loss: 0.4844
Epoch 18 | Train loss: 0.5148 | Val loss: 0.4838
Epoch 19 | Train loss: 0.5131 | Val loss: 0.4817
Epoch 20 | Train loss: 0.5135 | Val loss: 0.4815
Epoch 21 | Train loss: 0.5094 | Val loss: 0.4758
Epoch 22 | Train loss: 0.5089 | Val loss: 0.4751
Epoch 23 | Train loss: 0.5061 | Val loss: 0.4735
Epoch 24 | Train loss: 0.5048 | Val loss: 0.4704
Epoch 25 | Train loss: 0.5019 | Val loss: 0.4694
Epoch 26 | Train loss: 0.5022 | Val loss: 0.4680
Epoch 27 | Train loss: 0.4992 | Val loss: 0.4679
Epoch 28 | Train loss: 0.4977 | Val loss: 0.4644
Epoch 29 | Train loss: 0.4971 | Val loss: 0.4624
Epoch 30 | Train loss: 0.4956 | Val loss: 0.4636
Epoch 31 | Train loss: 0.4937 | Val loss: 0.4598
Epoch 32 | Train loss: 0.4937 | Val loss: 0.4581
Epoch 33 | Train loss: 0.4902 | Val loss: 0.4571
Epoch 34 | Train loss: 0.4898 | Val loss: 0.4542
Epoch 35 | Train loss: 0.4896 | Val loss: 0.4553
Epoch 36 | Train loss: 0.4859 | Val loss: 0.4518
Epoch 37 | Train loss: 0.4860 | Val loss: 0.4506
Epoch 38 | Train loss: 0.4849 | Val loss: 0.4490
Epoch 39 | Train loss: 0.4825 | Val loss: 0.4470
Epoch 40 | Train loss: 0.4828 | Val loss: 0.4458
Epoch 41 | Train loss: 0.4809 | Val loss: 0.4462
Epoch 42 | Train loss: 0.4813 | Val loss: 0.4431
Epoch 43 | Train loss: 0.4785 | Val loss: 0.4433
Epoch 44 | Train loss: 0.4769 | Val loss: 0.4414
Epoch 45 | Train loss: 0.4766 | Val loss: 0.4399
Epoch 46 | Train loss: 0.4752 | Val loss: 0.4402
Epoch 47 | Train loss: 0.4743 | Val loss: 0.4399
Epoch 48 | Train loss: 0.4740 | Val loss: 0.4365
Epoch 49 | Train loss: 0.4727 | Val loss: 0.4366
Epoch 50 | Train loss: 0.4725 | Val loss: 0.4338
```

Train on full dataset

During validation we observed that different runs varied strongly in perfomance. To counteract this, we decided to train 10 models and take the one performing best on the training set.

In [60]:
```python
epochs = 51
best_model = None

for i in range(10):

    best_loss = None
    curr_loss = None

    nam = NAM()
    optimizer = optim.Adam(nam.parameters(),
                           lr=1e-3, weight_decay=1e-6)
    output_penalty = 1e-3

    for epoch in tqdm(range(epochs)):

        train_loss_cum = 0
        cum = 0

        for x,y in full_dataloader:

            optimizer.zero_grad()
            out = nam(x)
            out = torch.squeeze(out)
            loss = loss_fn(out, y) + output_penalty * featureLoss(nam.feature_out)
            loss.backward()
            optimizer.step()

            train_loss_cum += x.shape[0] * loss
            cum += x.shape[0]


        train_loss = train_loss_cum/cum

        if epoch == epochs - 1:

            curr_loss = train_loss

    if best_loss is None or curr_loss < best_loss:

        best_loss = curr_loss
        best_model = nam

print(best_loss)
```

```
100%|██████████| 51/51 [00:40<00:00,  1.27it/s]
100%|██████████| 51/51 [00:41<00:00,  1.22it/s]
100%|██████████| 51/51 [00:33<00:00,  1.53it/s]
100%|██████████| 51/51 [00:31<00:00,  1.61it/s]
100%|██████████| 51/51 [00:31<00:00,  1.60it/s]
100%|██████████| 51/51 [00:31<00:00,  1.61it/s]
100%|██████████| 51/51 [00:31<00:00,  1.61it/s]
100%|██████████| 51/51 [00:39<00:00,  1.29it/s]
100%|██████████| 51/51 [00:38<00:00,  1.33it/s]
100%|██████████| 51/51 [00:37<00:00,  1.35it/s]
tensor(0.4602, grad_fn=<DivBackward0>)
```

In [61]:
```python
nam = best_model
```

Evaluate Metrics

In [62]:
```python
with torch.no_grad():
    y_pred = nam(test_x)
    y_pred = torch.sigmoid(y_pred).numpy().squeeze()
```

In [63]:
```python
y_pred = np.round(y_pred)
```

In [64]:
```python
f1_score(y_test, y_pred)
```

Out[64]: 0.853448275862069

In [65]:
```python
balanced_accuracy_score(y_test, y_pred)
```

Out[65]: 0.7945945945945946

Shap values

In [66]:
```python
wrapper = nn.Sequential(nam, nn.Sigmoid())
```

In [67]:
```python
explainer = shap.DeepExplainer(wrapper, train_x)
```

In [68]:
```python
positive = np.where(y_pred == 1)[0][:4]
negative = np.where(y_pred == 0)[0][:4]
```

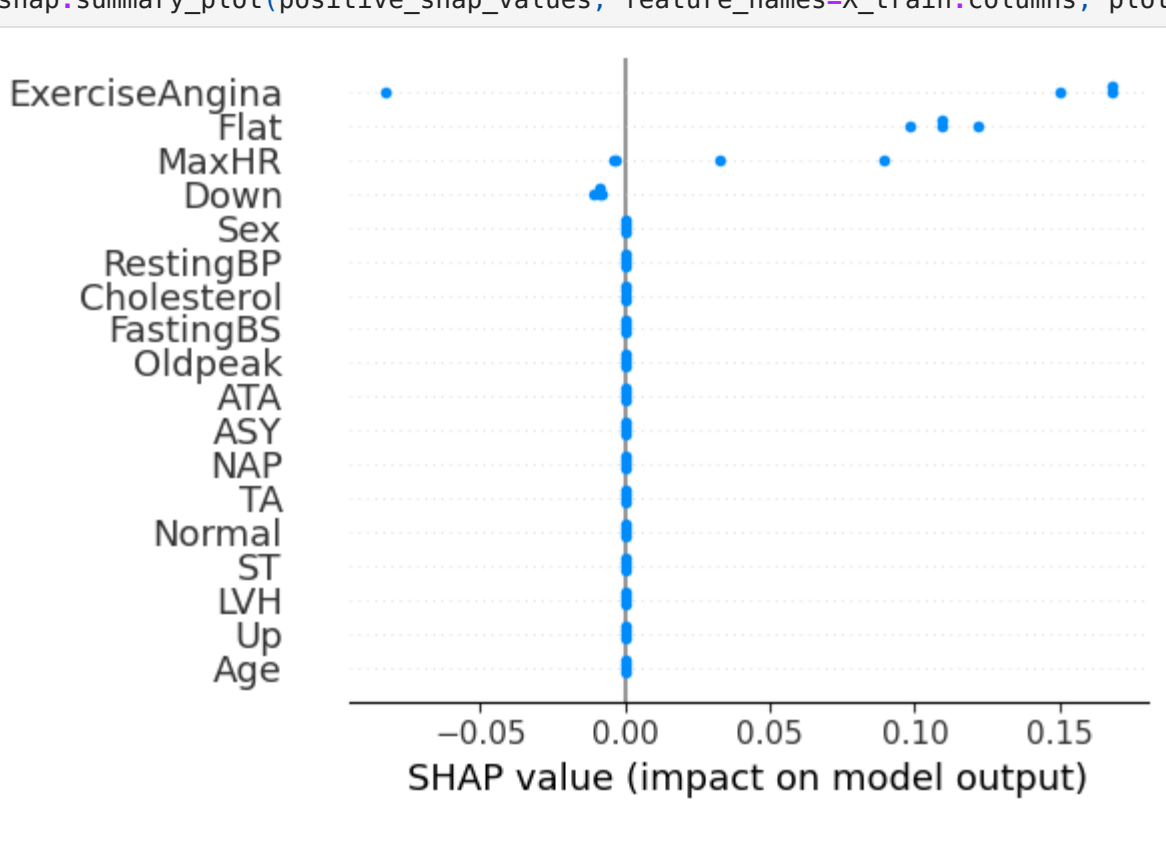
In [69]:
```python
positive_x = test_x[positive]
negative_x = test_x[negative]
```

In [70]:
```python
positive_shap_values = explainer.shap_values(positive_x)
negative_shap_values = explainer.shap_values(negative_x)
overall_shap_values = explainer.shap_values(train_x)
```

```
Using a non-full backward hook when the forward contains multiple autograd Nodes is deprecated and will be removed in future versions. This hook will be missing some grad_input. Please use register_full_backward_hook to get the documented behavior.
```

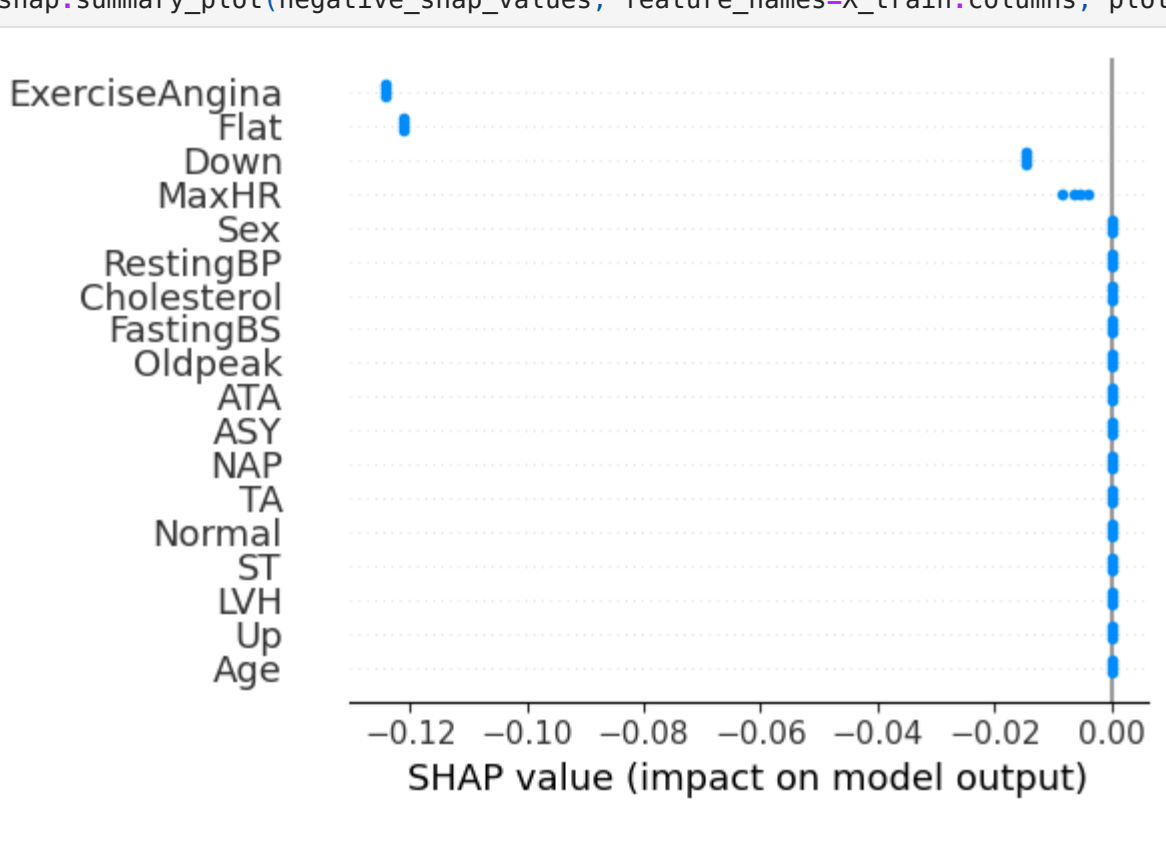
Shapley Plot for 4 positive samples

In [71]:
```python
shap.summary_plot(positive_shap_values, feature_names=X_train.columns, plot_size=(6,4))
```



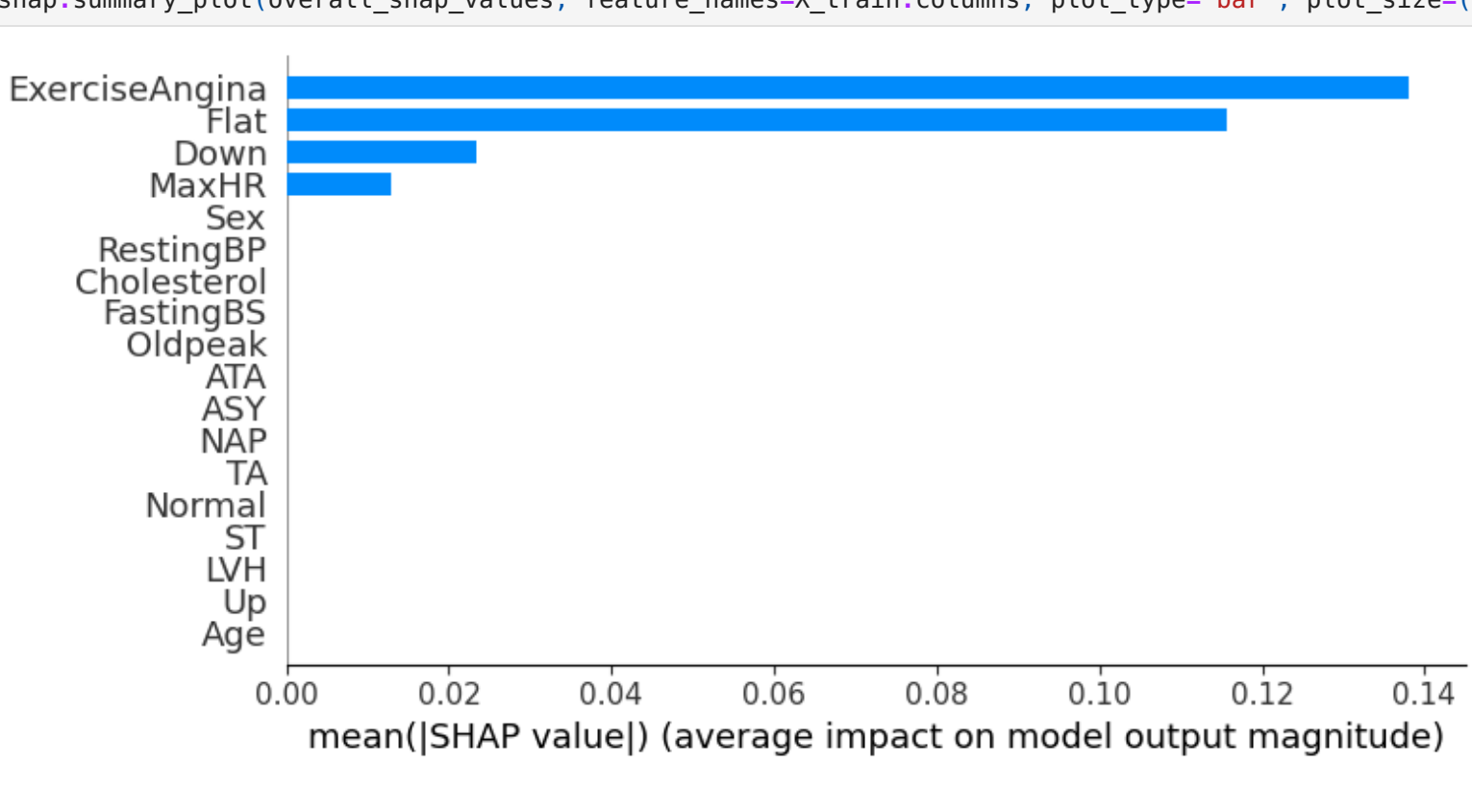
Shapley Plot for 4 negative values

In [72]:
```python
shap.summary_plot(negative_shap_values, feature_names=X_train.columns, plot_size=(6,4))
```



Shapley Plot for all train samples

In [73]:
```python
shap.summary_plot(overall_shap_values, feature_names=X_train.columns, plot_type='bar', plot_size=(8
```



Are the feature importances similar for different samples?

The plots show that the shapley values are now much more similiar across samples from the same class.

Do the feature importances found with NAMs differ from the ones in Q2-4, or are they consistently the same?

In contrast to Q2-4, there are only 4 features for NAMs which have non-zero importances.
Interestingly the feature "Down" has basically zero importance for Q2-4, but for the NAM it has moderately high importance.

Conceptually, how does the model compare to Logistic Regression and MLPs?

The NAM architecture is a mixture of Logistic Regression and MLPs.
First for each feature we use one MLP to extract a good non-linear representation of that feature.
With these new features, we essentially do logistic regression, as we are only linearly combining them.

Why are NAMs more interpretable than MLPs despite being based on non-linear neural networks?

Because different features ultimately only interact linearly as we simply sum them.
Thus we can more clearly separate their influences on the output.