ML4H Project 1

Ivan Gabelko, Christian Bernard, Arkaud Gallois

Disclaimer

We have created the report without our code snippets, each question is answered at the end of the section. We tried to do as best as possible. You can see our all code, but it was not clean at all. And most time is spent on the same results (differentiate high and low cholesterol).

Part 1

```
import numpy as np
import pandas as pd
import os
import matplotlib as mpl
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.preprocessing import LabelEncoder,MinMaxScaler
from sklearn.model_selection import train_test_split,StratifiedKFold
from sklearn.metrics import roc_curve,auc,precision_recall_curve,average_precision_score
import seaborn as sns
import scipy
import scipy.stats as st
from sklearn.feature_selection import SelectKBest, SelectFromModel
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegressionCV
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import LinearSVC
from sklearn.neighbors import KNeighborsClassifier
import lightgbm as lgb
import xgboost
%matplotlib inline
%load_ext autoreload
%autoreload 2
%load_ext rpy2.ipynb
%matplotlib inline
%load_ext autoreload
%autoreload 2
```

Load Data

```
train_df = pd.read_csv('train_features/train_set_with_id.csv')
X_train_df = train_df.drop(['id'],axis=1)
X_train = X_train_df.values.astype('float64')
test_df = pd.read_csv('test_features/test_set_with_id.csv')
X_test_df = test_df.drop(['id'],axis=1)
X_test = X_test_df.values.astype('float64')
```

Q1 Exploratory Data Analysis

Columns

```
train_df.columns
```

```
Index(['Age', 'Sex', 'ChestPainType', 'RestingBP', 'Cholesterol', 'FastingBP',
       'RestingECG', 'MaxHR', 'ExerciseAngina', 'Oldpeak', 'ST_Slope',
       'target'],
      dtype='object')
```

Checking for Nans

```
train_df.isnull().sum()
```

```
Age                 0
Sex                 0
ChestPainType       0
RestingBP           0
Cholesterol         0
FastingBP           0
RestingECG          0
MaxHR               0
ExerciseAngina      0
Oldpeak             0
ST_Slope            0
target              0
dtype: int64
```

As we 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 features columns with string values.
Some of these features can be binary encoded, some need to be one-hot encoded.
This can be done by counting each of these features.

See 'Sex'
'ChestPainType' 'RestingECG' 'Sex' 'ST_Slope' 'ExerciseAngina' 'Sex' 'ST_Slope'
'ExerciseAngina' 'Sex' 'ST_Slope'

```
df_cat_cols=['Sex']
df_cat_cols=['Sex']

def cat_info(df):
    print("{:<18} {:<7} {:<7}".format("Feature", "Type", "Count"))
    for i in df.columns:
        print("{:<18} {:<7} {:<7}".format(i, str(df[i].dtype), str(len(df[i]))))

cat_info(train_df[df_cat_cols])
```

```
Feature            Type    Count
Sex                object  734
```

```
cat_info(test_df[df_cat_cols])
```

```
Feature            Type    Count
Sex                object  184
```

Histograms

```
plt.rcParams['figure.figure.dpi'] = 150.0
train_df.hist(bins=50)
```

One can observe numerical data in the feature cholesterol which shows many zero value entries. Since Cholesterol is a numerical measurement of blood cholesterol, these zero values are not plausible and therefore should be treated as missing data.
We also observe that the distribution of the attributes age, resting blood pressure and maximum heart rate are skewed. Furthermore, there is no significant class imbalance within the heart disease column so we can easily train a non-biased classifier. Finally, we observe that the data is on a mixed scale and has potential outliers.

```
plt.rcParams['figure.figure.dpi'] = 150.0
train_df[['Cholesterol']].hist()
```

```
array([[<AxisSubplot:>]], dtype=object)
```

Fixing Cholesterol Distribution

```
train_df['Cholesterol'] = train_df['Cholesterol'].replace(to_replace=0.0,value=np.nan)
train_df['Cholesterol'] = train_df['Cholesterol'].fillna(train_df['Cholesterol'].mean())
X_train = train_df.drop(['id'],axis=1).values.astype('float64')

plt.rcParams['figure.figure.dpi'] = 150.0
train_df[['Cholesterol']].hist()
```

```
array([[<AxisSubplot:>]], dtype=object)
```

Outlier removal

Trees are not impacted with outlier removal
We use the 3 Sigma to remove outliers in these distributions, that is we remove any sample with a z-Score above 3 in any feature.

```
train_[~(train_[['RestingBP','Cholesterol']] - train_[['RestingBP','Cholesterol']].mean() / train_[['RestingBP','Cholesterol']].std() > 3).any(axis=1)]
X_train = train_[~(train_[['RestingBP','Cholesterol']] - train_[['RestingBP','Cholesterol']].mean() / train_[['RestingBP','Cholesterol']].std() > 3).any(axis=1)].drop(['id'],axis=1).values.astype('float64')
y_train = train_[~(train_[['RestingBP','Cholesterol']] - train_[['RestingBP','Cholesterol']].mean() / train_[['RestingBP','Cholesterol']].std() > 3).any(axis=1)].target.values.astype('float64')

train.shape, X_train.shape
```

```
((918, 12), (734, 11))
```

Normalizing

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

```
for i in df_cat_cols:
    print('{:<18}'.format(i),
          train_df[[i]].shape, train_df[[i]].dtype, train_df[[i]].isnull().sum(),
          test_df[[i]].shape, test_df[[i]].dtype, test_df[[i]].isnull().sum(),
          train_df[[i]].nunique(),test_df[[i]].nunique())
```

```
Sex               (918, 1) object 0 (184, 1) object 0 2 2
```

```
for cols in df[df_cat_cols]:
    ax = sns.countplot(x=cols, hue="target", data=df, palette="husl")
    plt.show()
```

Correlation Matrix

```
plt.rcParams['figure.figure.dpi'] = 150.0
corr = train_df[['Age','RestingBP','Cholesterol','FastingBP','MaxHR','Oldpeak','target']].corr()
sns.heatmap(corr, annot=True, cmap='RdBu')
```

```
<AxisSubplot:>
```

Fist, Q1 and Exercise Angina have the highest positive correlation with the class heart disease either our correlation matrix or our chi2 test. Age, resting blood pressure and maximum heart rate have a low positive correlation with the class. Finally, low correlation values are observed for the remaining features of both dataset.

```
train_df = train_df.drop(['id'],axis=1)
```

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 are not directly comparable.

Fit Classifier

```
lr = LogisticRegressionCV(penalty='l2', solver='liblinear').fit(train_df, y)
```

Evaluate Metrics

```
y_pred = lr.predict(test_df)
```

F1 Score

```
f1_score(y_test, y_pred)
```

```
0.8024691358024691
```

Balanced Accuracy

```
balanced_accuracy_score(y_test, y_pred)
```

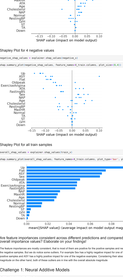
```
0.8024691358024691
```

Plot absolute magnitude of feature coefficients

In order to interpret the importance of features, we use the absolute values of the corresponding coefficients ordered by magnitude.

```
lr_coef = np.abs(np.squeeze(lr_cv[-1].coef_))
lr_coef_idx = np.argsort(lr_coef)[::-1]
print('Logistic Regression Coefficients:')
for i,c in zip(train_df.columns[lr_coef_idx],lr_coef[lr_coef_idx]):
    print('{:<18} {:>7.4f}'.format(i,c))
```

```
Logistic Regression Coefficients:
ST_Slope           0.9000
ChestPainType      0.8000
Sex                0.7000
ExerciseAngina     0.6000
Oldpeak            0.5000
Age                0.4000
RestingBP          0.3000
Cholesterol        0.2000
FastingBP          0.1000
MaxHR              0.0000
```



Fist, Q1 and Exercise Angina have the highest positive correlation with the class heart disease either our correlation matrix or our chi2 test. Age, resting blood pressure and maximum heart rate have a low positive correlation with the class. Finally, low correlation values are observed for the remaining features of both dataset.

```
train_df = train_df.drop(['id'],axis=1)
```

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 are not directly comparable.

Fit Classifier

```
lr = LogisticRegressionCV(penalty='l2', solver='liblinear').fit(train_df, y)
```

Evaluate Metrics

```
y_pred = lr.predict(test_df)
```

F1 Score

```
f1_score(y_test, y_pred)
```

```
0.8024691358024691
```

Balanced Accuracy

```
balanced_accuracy_score(y_test, y_pred)
```

```
0.8024691358024691
```

Plot absolute magnitude of feature coefficients

In order to interpret the importance of features, we use the absolute values of the corresponding coefficients ordered by magnitude.

```
lr_coef = np.abs(np.squeeze(lr_cv[-1].coef_))
lr_coef_idx = np.argsort(lr_coef)[::-1]
print('Logistic Regression Coefficients:')
for i,c in zip(train_df.columns[lr_coef_idx],lr_coef[lr_coef_idx]):
    print('{:<18} {:>7.4f}'.format(i,c))
```

```
Logistic Regression Coefficients:
ST_Slope           0.9000
ChestPainType      0.8000
Sex                0.7000
ExerciseAngina     0.6000
Oldpeak            0.5000
Age                0.4000
RestingBP          0.3000
Cholesterol        0.2000
FastingBP          0.1000
MaxHR              0.0000
```

Fist, Q1 and Exercise Angina have the highest positive correlation with the class heart disease either our correlation matrix or our chi2 test. Age, resting blood pressure and maximum heart rate have a low positive correlation with the class. Finally, low correlation values are observed for the remaining features of both dataset.

Finally, looking at the absolute values of the coefficients, we observe that the most important features are ST_Slope, ChestPainType, Sex, ExerciseAngina, Oldpeak, Age, RestingBP, Cholesterol, FastingBP, MaxHR. Interestingly, the coefficients of the categorical features are the most important.

The plot shows that the categorical features have the highest impact on the model. Furthermore, these features have positive coefficients (and hence strong interactions with other features, but these too go unlimited). Methods such as random forest importance are more suited for this task.

Q3 Decision Tree

Fit Classifier

```
tree_clf = DecisionTreeClassifier().fit(train_df, y)
```

Evaluate Metrics

```
y_pred = tree_clf.predict(test_df)
```

F1 Score

```
f1_score(y_test, y_pred)
```

```
0.8444444444444444
```

Balanced Accuracy

```
balanced_accuracy_score(y_test, y_pred)
```

```
0.8444444444444444
```

Plot Gini Importances

We plot the ranked gini importances.

```
sorted_indices = np.argsort(tree_clf.feature_importances)
print('Decision Tree Gini Importances:')
for i,c in zip(train_df.columns[sorted_indices],tree_clf.feature_importances[sorted_indices]):
    print('{:<18} {:>7.4f}'.format(i,c))
plt.rcParams['figure.figure.dpi'] = 150.0
plt.barh(train_df.columns[sorted_indices],tree_clf.feature_importances[sorted_indices])
plt.xlabel('Importance')
plt.ylabel('Feature')
plt.title('Absolute Gini Importances')
plt.show()
```

```
Decision Tree Gini Importances:
FastingBP          0.0000
ExerciseAngina     0.0000
RestingECG         0.0000
Sex                0.0000
ST_Slope           0.0000
ChestPainType      0.0000
Age                0.0000
RestingBP          0.0000
Oldpeak            0.0000
Cholesterol        0.0000
MaxHR              0.0000
```


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.

```
class MLPerceptron:
    pass
    def __init__(self, dim):
        super.__init__(self)
        pass
        self.layers = [w_hidden]*2
        for layer in self.layers:
            pass
            layer.requires_grad = True
            layer.retain_grad = True
    pass

    def forward(self, x):
        pass
        h = x.mm(self.layers[0])
        h = h.clamp(min=0)
        y = h.mm(self.layers[1])
        return y
    pass
```

Dataset class for loading batches

```
class NpDataset(Dataset):
    pass
    def __init__(self, features, labels):
        pass
        self.features = features
        self.labels = labels
    pass
    def __getitem__(self, index):
        pass
        return self.features[index], self.labels[index]
    pass
```

Split into train and validation datasets

We first separate our cases in 90:10 ratio as dev set.

```
X_train, X_val, y_train, y_val = train_test_split(X_train, y, test_size=0.1, random_state=42)
```

Dataloader

```
train_dataset = NpDataset(X_train.astype(dtype=np.float32), y_train.astype(dtype=np.int64))
val_dataset = NpDataset(X_val.astype(dtype=np.float32), y_val.astype(dtype=np.int64))
test_dataset = NpDataset(X_test.astype(dtype=np.float32), y_test.astype(dtype=np.int64))
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

Loss Function

```
loss_fn = lambda BCEWithLogitsLoss()
```

Validation

```
def train_plot(model, epochs, optimizer, lr=0.1):
    optimizer = optim.Adam(model.parameters())
    pass
    for epoch in range(epochs):
        pass
        train_loss_ave = 0
        pass
        for i,(x,y) in enumerate:
            pass
            optimizer.zero_grad()
            pass
            y1 = torch.squeeze(model(x))
            pass
            loss = loss_fn(y1, y.float())
            pass
            train_loss_ave = train_loss_ave + loss.item()
            pass
            if i%5 == 0:
                pass
                optim.step()
                pass
                y = torch.squeeze(model(x))
                pass
                loss = loss_fn(y, y.float())
                pass
                train_loss = train_loss.append(train_loss_ave / i)
                pass
            if epoch % 5 == 0:
                pass
                print('Epoch %epoch %
                train_loss_ave = train_loss_ave / i
                pass
                if train_loss[-1] < 0.:
                    pass
                    print('Epoch %epoch %
                    train_loss.append(train_loss_ave)
                    pass
                    break
    pass
    return train_loss
pass
```

```
Epoch 0: Train Loss: 0.6931
Epoch 5: Train Loss: 0.6931 | Get Loss: 0.6931
Epoch 10: Train Loss: 0.6931 | Get Loss: 0.6931
Epoch 15: Train Loss: 0.6931 | Get Loss: 0.6931
Epoch 20: Train Loss: 0.6931 | Get Loss: 0.6931
Epoch 25: Train Loss: 0.6931 | Get Loss: 0.6931
Epoch 30: Train Loss: 0.6931 | Get Loss: 0.6931
Epoch 35: Train Loss: 0.6931 | Get Loss: 0.6931
```

Train on full binary dataset

```
y_bin = y.copy()
```

```
train_plot(mlp, epochs=40, optimizer=Adam(), lr=0.01)
train_plot(mlp, epochs=40, optimizer=Adam(), lr=0.001)
```

Evaluate Metrics

```
y_pred = torch.sigmoid(torch.squeeze(mlp(torch.Tensor(X))))
y_pred = (y_pred > 0.5).cpu().numpy()
```

```
y_pred = torch.sigmoid(torch.squeeze(mlp(torch.Tensor(X))))
y_pred = (y_pred > 0.5).cpu().numpy()
```

F1 Score

```
f1_score(y_bin, y_pred)
```

```
0.8333333333333333
```

Balanced Accuracy

```
balanced_accuracy_score(y_bin, y_pred)
```

```
0.8333333333333333
```

Shap values

```
shap_i = shap.Sampler(10, X, replace=False).sample(1)
```

Against Sigmoid to the model to get shap values with respect to the probability of the output

```
def f_h(X):
    return np.sigmoid(np.dot(np.tanh(np.dot(X, w_hidden)), w_output))
```

Take the first 4 negative and first 4 positive samples

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

explainer = shap.KernelExplainer(f_h, X)
shap_values = explainer.shap(X[positive])
```

```
100%|==========|Step 1 of 1 Done in 0.0 secs
```

Shapley Plot for a positive samples

```
plt.rcParams['figure.figure.dpi'] = 150.0
shap.summary_plot(shap_values, features=X[positive], feature_names=train_df.columns)
```

Trying to connect Sigmoid back to the model output to get the probability of the output. Unfortunately, this does not work properly. We therefore use the Sigmoid function in the model output instead. Shap values should then be in units of the log-odds ratio. We can then use the DeepExplainer class to get the unscaled Shap values.

```
shap.summary_plot(shap_values, features=X[positive], feature_names=train_df.columns, plot_size=(6,4))
```


Shapley Plot for all negative values

```
shap.summary_plot(shap_values, features=X[negative], feature_names=train_df.columns)
```

```
shap.summary_plot(shap_values, features=X[negative], feature_names=train_df.columns, plot_size=(6,4))
```


Shapley Plot for all train samples

```
shap.summary_plot(shap_values, features=X[sample], feature_names=train_df.columns)
```

```
shap.summary_plot(shap_values, features=X[sample], feature_names=train_df.columns, plot_size=(6,4))
```


Are Shapley importances correlated across different predictions and compared to overall importance values? Elaborate on differences and their causes.

The Shapley importances are mostly consistent. Age is most often one positive for the positive samples and negative for the negative samples. Across the whole cohort, it is positive. For cholesterol, the two cluster negative impact for one of the positive samples and 50% of the negative samples. Considering their absolute magnitude, it is clear that age is the most important feature for the overall absolute magnitude.

Challenge 1: Neural Additive Models

```
class FeatureNetwork(Module):
    pass
    def __init__(self):
        pass
        super.__init__(self)
        pass
        self.layers = [w_hidden]*2
        pass
        for layer in self.layers:
            pass
            layer.requires_grad = True
            layer.retain_grad = True
    pass

    def forward(self, x):
        pass
        h = x.mm(self.layers[0])
        h = h.clamp(min=0)
        y = h.mm(self.layers[1])
        return y
    pass
pass

class NAMLoss(Module):
    pass
    def __init__(self, dim=10):
        pass
        super.__init__(self)
        pass
        self.fnns = [NeuralNet()]*dim
        pass
        for fnn in self.fnns:
            pass
            fnn.requires_grad = True
            pass
        self.features_dim = dim
        pass
    pass

    def forward(self, x):
        pass
        y = torch.zeros(x.shape[0], self.dim, ...).to(x.device, ...)
        pass
        for i,fnn in enumerate(self.fnns):
            pass
            y[:,i,...] = fnn(x[:,i,...])
            pass
        y = torch.sum(y,1) + b0
        pass
        return torch.sigmoid, dim=1) - w0*0.5
    pass
pass
```

```
class NAM(Module):
    pass
    def __init__(self, dim=10):
        pass
        super.__init__(self)
        pass
        self.fnns = [NeuralNet()]*dim
        pass
        for fnn in self.fnns:
            pass
            fnn.requires_grad = True
            pass
        self.features_dim = dim
        pass
    pass

    def forward(self, x):
        pass
        y = torch.zeros(x.shape[0], self.dim, ...).to(x.device, ...)
        pass
        for i,fnn in enumerate(self.fnns):
            pass
            y[:,i,...] = fnn(x[:,i,...])
            pass
        y = torch.sum(y,1) + b0
        pass
        return torch.sigmoid, dim=1) - w0*0.5
    pass
pass
```

Loss Function and Optimizer

```
def features(x):
    return torch.matrix(torch.dense(torch.sparse(x), ...), ...)

lr = 0.01
optimizer = optim.Adam(nam.parameters())
scheduler = ReduceLROnPlateau(optimizer, ...)

epochs = 200
for epoch in range(epochs):
    pass
    for i,(x,y) in enumerate(train_loader):
        pass
        optimizer.zero_grad()
        pass
        y1 = nam(x, features(x))
        pass
        loss = loss_fn(y1, y.float())
        pass
        loss.backward()
        pass
        optimizer.step()
        pass
    pass
    if epoch % 10 == 0:
        pass
        print('Epoch %epoch %
        pass
    pass
pass
```

Validation

```
y = torch.cat([y for x,y in test_loader], ...)
y = y.cpu().numpy()
y_pred = torch.cat([nam(x, features(x)) for x,y in test_loader], ...)
y_pred = y_pred.cpu().numpy()
```

F1 Score

```
f1_score(y, y_pred)
```

```
0.8333333333333333
```

Balanced Accuracy

```
balanced_accuracy_score(y, y_pred)
```

```
0.8333333333333333
```

Shap values

```
shap_i = shap.Sampler(10, X, replace=False).sample(1)
```

Against Sigmoid to the model to get shap values with respect to the probability of the output

```
def f_h(X):
    return np.sigmoid(np.dot(np.tanh(np.dot(X, w_hidden)), w_output))
```

Take the first 4 negative and first 4 positive samples

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

explainer = shap.KernelExplainer(f_h, X)
shap_values = explainer.shap(X[positive])
```

```
100%|==========|Step 1 of 1 Done in 0.0 secs
```

Shapley Plot for a positive samples

```
plt.rcParams['figure.figure.dpi'] = 150.0
shap.summary_plot(shap_values, features=X[positive], feature_names=train_df.columns)
```

Trying to connect Sigmoid back to the model output to get the probability of the output. Unfortunately, this does not work properly. We therefore use the Sigmoid function in the model output instead. Shap values should then be in units of the log-odds ratio. We can then use the DeepExplainer class to get the unscaled Shap values.

```
shap.summary_plot(shap_values, features=X[positive], feature_names=train_df.columns, plot_size=(6,4))
```


Shapley Plot for all negative values

```
shap.summary_plot(shap_values, features=X[negative], feature_names=train_df.columns)
```

```
shap.summary_plot(shap_values, features=X[negative], feature_names=train_df.columns, plot_size=(6,4))
```


Shapley Plot for all train samples

```
shap.summary_plot(shap_values, features=X[sample], feature_names=train_df.columns)
```

```
shap.summary_plot(shap_values, features=X[sample], feature_names=train_df.columns, plot_size=(6,4))
```


Are Shapley importances correlated across different predictions and compared to overall importance values? Elaborate on differences and their causes.

The Shapley importances are mostly consistent. Age is most often one positive for the positive samples and negative for the negative samples. Across the whole cohort, it is positive. For cholesterol, the two cluster negative impact for one of the positive samples and 50% of the negative samples. Considering their absolute magnitude, it is clear that age is the most important feature for the overall absolute magnitude.

Challenge 2: Neural Additive Models

```
class FeatureNetwork(Module):
    pass
    def __init__(self):
        pass
        super.__init__(self)
        pass
        self.layers = [w_hidden]*2
        pass
        for layer in self.layers:
            pass
            layer.requires_grad = True
            layer.retain_grad = True
    pass

    def forward(self, x):
        pass
        h = x.mm(self.layers[0])
        h = h.clamp(min=0)
        y = h.mm(self.layers[1])
        return y
    pass
pass

class NAMLoss(Module):
    pass
    def __init__(self, dim=10):
        pass
        super.__init__(self)
        pass
        self.fnns = [NeuralNet()]*dim
        pass
        for fnn in self.fnns:
            pass
            fnn.requires_grad = True
            pass
        self.features_dim = dim
        pass
    pass

    def forward(self, x):
        pass
        y = torch.zeros(x.shape[0], self.dim, ...).to(x.device, ...)
        pass
        for i,fnn in enumerate(self.fnns):
            pass
            y[:,i,...] = fnn(x[:,i,...])
            pass
        y = torch.sum(y,1) + b0
        pass
        return torch.sigmoid, dim=1) - w0*0.5
    pass
pass
```

```
class NAM(Module):
    pass
    def __init__(self, dim=10):
        pass
        super.__init__(self)
        pass
        self.fnns = [NeuralNet()]*dim
        pass
        for fnn in self.fnns:
            pass
            fnn.requires_grad = True
            pass
        self.features_dim = dim
        pass
    pass

    def forward(self, x):
        pass
        y = torch.zeros(x.shape[0], self.dim, ...).to(x.device, ...)
        pass
        for i,fnn in enumerate(self.fnns):
            pass
            y[:,i,...] = fnn(x[:,i,...])
            pass
        y = torch.sum(y,1) + b0
        pass
        return torch.sigmoid, dim=1) - w0*0.5
    pass
pass
```

Loss Function and Optimizer

```
def features(x):
    return torch.matrix(torch.dense(torch.sparse(x), ...), ...)

lr = 0.01
optimizer = optim.Adam(nam.parameters())
scheduler = ReduceLROnPlateau(optimizer, ...)

epochs = 200
for epoch in range(epochs):
    pass
    for i,(x,y) in enumerate(train_loader):
        pass
        optimizer.zero_grad()
        pass
        y1 = nam(x, features(x))
        pass
        loss = loss_fn(y1, y.float())
        pass
        loss.backward()
        pass
        optimizer.step()
        pass
    pass
    if epoch % 10 == 0:
        pass
        print('Epoch %epoch %
        pass
    pass
pass
```

Validation

```
y = torch.cat([y for x,y in test_loader], ...)
y = y.cpu().numpy()
y_pred = torch.cat([nam(x, features(x)) for x,y in test_loader], ...)
y_pred = y_pred.cpu().numpy()
```

```
epochs = 5
for epoch in range(epochs):
    train_loss_sum = 0
    n_batches = 0

    for x, y in train_dataloader:
        optimizer.zero_grad()
        x = x.to(device)
        y = y.to(device)
        outputs = model(x, output_penalty = features[criterion.feature_out])
        loss = criterion(outputs, y)
        loss.backward()
        optimizer.step()

        train_loss_sum += loss.item()
        n_batches += 1

    avg_train_loss = train_loss/n_batches

    val = 0.0
    val_n = 0.0
    with torch.inference_mode():
        for x, y in val_dataloader:
            x = x.to(device)
            y = y.to(device)
            outputs = model(x, output_penalty = features[criterion.feature_out])
            val += (outputs.argmax(-1) == y).sum()
            val_n += len(y)
    acc = val / val_n

    print(f"Epoch {epoch} | Train Loss {avg_train_loss:.3f} | Val Acc {acc:.3f}")
```

```
Epoch 0 | Train Loss 0.703 | Val Acc 0.723
Epoch 1 | Train Loss 0.576 | Val Acc 0.723
Epoch 2 | Train Loss 0.576 | Val Acc 0.723
Epoch 3 | Train Loss 0.576 | Val Acc 0.723
Epoch 4 | Train Loss 0.576 | Val Acc 0.723
Epoch 5 | Train Loss 0.576 | Val Acc 0.723
Epoch 6 | Train Loss 0.576 | Val Acc 0.723
Epoch 7 | Train Loss 0.576 | Val Acc 0.723
Epoch 8 | Train Loss 0.576 | Val Acc 0.723
Epoch 9 | Train Loss 0.576 | Val Acc 0.723
Epoch 10 | Train Loss 0.576 | Val Acc 0.723
Epoch 11 | Train Loss 0.576 | Val Acc 0.723
Epoch 12 | Train Loss 0.576 | Val Acc 0.723
Epoch 13 | Train Loss 0.576 | Val Acc 0.723
Epoch 14 | Train Loss 0.576 | Val Acc 0.723
Epoch 15 | Train Loss 0.576 | Val Acc 0.723
Epoch 16 | Train Loss 0.576 | Val Acc 0.723
Epoch 17 | Train Loss 0.576 | Val Acc 0.723
Epoch 18 | Train Loss 0.576 | Val Acc 0.723
Epoch 19 | Train Loss 0.576 | Val Acc 0.723
Epoch 20 | Train Loss 0.576 | Val Acc 0.723
Epoch 21 | Train Loss 0.576 | Val Acc 0.723
Epoch 22 | Train Loss 0.576 | Val Acc 0.723
Epoch 23 | Train Loss 0.576 | Val Acc 0.723
Epoch 24 | Train Loss 0.576 | Val Acc 0.723
Epoch 25 | Train Loss 0.576 | Val Acc 0.723
Epoch 26 | Train Loss 0.576 | Val Acc 0.723
Epoch 27 | Train Loss 0.576 | Val Acc 0.723
Epoch 28 | Train Loss 0.576 | Val Acc 0.723
Epoch 29 | Train Loss 0.576 | Val Acc 0.723
Epoch 30 | Train Loss 0.576 | Val Acc 0.723
Epoch 31 | Train Loss 0.576 | Val Acc 0.723
Epoch 32 | Train Loss 0.576 | Val Acc 0.723
Epoch 33 | Train Loss 0.576 | Val Acc 0.723
Epoch 34 | Train Loss 0.576 | Val Acc 0.723
Epoch 35 | Train Loss 0.576 | Val Acc 0.723
Epoch 36 | Train Loss 0.576 | Val Acc 0.723
Epoch 37 | Train Loss 0.576 | Val Acc 0.723
Epoch 38 | Train Loss 0.576 | Val Acc 0.723
Epoch 39 | Train Loss 0.576 | Val Acc 0.723
Epoch 40 | Train Loss 0.576 | Val Acc 0.723
Epoch 41 | Train Loss 0.576 | Val Acc 0.723
Epoch 42 | Train Loss 0.576 | Val Acc 0.723
Epoch 43 | Train Loss 0.576 | Val Acc 0.723
Epoch 44 | Train Loss 0.576 | Val Acc 0.723
Epoch 45 | Train Loss 0.576 | Val Acc 0.723
Epoch 46 | Train Loss 0.576 | Val Acc 0.723
Epoch 47 | Train Loss 0.576 | Val Acc 0.723
Epoch 48 | Train Loss 0.576 | Val Acc 0.723
Epoch 49 | Train Loss 0.576 | Val Acc 0.723
```

Train on full dataset

Finally, whether we observed that different runs varied strongly in performance. To compensate this, we decided to train 10 models and take the one performing best on the training set.

```
models = []
losses = []
for i in range(10):
    train_loss = 0.0
    n_batches = 0.0
    model = MNIST()
    optimizer = optim.AdamW(model.parameters())
                                    # This is default,
    criterion = nn.CrossEntropyLoss()
    output_penalty = 0.0
    for epoch in range(epochs):
        train_loss = 0.0
        n_batches = 0
        for x, y in full_dataloader:
            optimizer.zero_grad()
            x = x.to(device)
            y = y.to(device)
            outputs = model(x, output_penalty = features[criterion.feature_out])
            loss = criterion(outputs, y)
            loss.backward()
            optimizer.step()

            train_loss += loss.item()
            n_batches += 1

        avg_train_loss = train_loss/n_batches

        if epoch == epochs - 1:
            last_loss = train_loss

    if last_loss < 0.9 and last_loss < best_loss:
        best_loss = last_loss
        best_model = model
print(best_loss)
```

```
0.8[-------------------------------------------] 1% 00:00:00, 1.25s/it]
0.8[-------------------------------------------] 2% 00:00:00, 1.25s/it]
0.8[-------------------------------------------] 3% 00:00:00, 1.25s/it]
0.8[-------------------------------------------] 4% 00:00:00, 1.25s/it]
0.8[-------------------------------------------] 5% 00:00:00, 1.25s/it]
0.8[-------------------------------------------] 6% 00:00:00, 1.25s/it]
0.8[-------------------------------------------] 7% 00:00:00, 1.25s/it]
0.8[-------------------------------------------] 8% 00:00:00, 1.25s/it]
0.8[-------------------------------------------] 9% 00:00:00, 1.25s/it]
0.8[-------------------------------------------] 10% 00:00:00, 1.25s/it]
```

```
0.8[-------------------------------------------] 100
```

Part 2

Implement version of kaggle's code, as there were some issues with output library for the integrated gradients and the newest version of keras/tf.

```
!pip install matplotlib==3.1.3
```

```
!pip install lucid
!pip install keras==2.2.4
!pip install tqdm
```

```
import os
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
from sklearn.metrics import *
from sklearn.utils import shuffle, StandardScaler
```

```
%matplotlib inline
%load_ext autoreload
%autoreload 2
import matplotlib.cm as cm
import matplotlib.image as mpimg
```

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
os.environ['TF_KERAS'] = '1'
os.environ['CUDA_VISIBLE_DEVICES'] = '-1'
os.environ['TF_CUDNN_USE_AUTOTUNE'] = '0'
os.environ['TF_ENABLE_CUDNN_RNN_KERNEL_BATCHING_FUSION'] = '0'
os.environ['TF_ENABLE_CUDNN_MULTI_STREAM_KERNELS'] = '0'
os.environ['TF_DISABLE_CUDNN_RNN_ALL_ASYNC_KERNELS'] = '1'
os.environ['TF_DISABLE_CUDNN_TENSOR_OP_MATH'] = '1'
os.environ['TF_DISABLE_CUDNN_RNN_TENSOR_OP_MATH'] = '1'
os.environ['TF_DISABLE_CUDNN_BATCHNORM_SPATIAL_PERSISTENT'] = '1'
os.environ['TF_DISABLE_CUDNN_BATCHNORM_EX_PERSISTENT'] = '1'
os.environ['TF_DISABLE_CUDNN_POOL_PERSISTENT'] = '1'
os.environ['TF_DISABLE_CUDNN_ACTIVATION_PERSISTENT'] = '1'
os.environ['TF_DISABLE_CUDNN_ADD_TENSOR_PERSISTENT'] = '1'
os.environ['TF_DISABLE_CUDNN_CONVOLUTION_PERSISTENT'] = '1'
os.environ['TF_DISABLE_CUDNN_CONVOLUTION_BIAS_ACTIVATION_PERSISTENT'] = '1'
os.environ['TF_DISABLE_CUDNN_CONVOLUTION_BIAS_ADD_PERSISTENT'] = '1'
os.environ['TF_DISABLE_CUDNN_CONVOLUTION_BIAS_ADD_ACTIVATION_PERSISTENT'] = '1'
os.environ['TF_DISABLE_CUDNN_CONVOLUTION_BIAS_ADD_ACTIVATION_PERSISTENT_NHWC'] = '1'
os.environ['TF_DISABLE_CUDNN_CONVOLUTION_BIAS_ADD_ACTIVATION_PERSISTENT_NCHW'] = '1'
os.environ['TF_DISABLE_CUDNN_CONVOLUTION_BIAS_ADD_ACTIVATION_PERSISTENT_NCHW_VECT_C'] = '1'
os.environ['TF_DISABLE_CUDNN_CONVOLUTION_BIAS_ADD_ACTIVATION_PERSISTENT_NCHW_VECT_C_NHWC'] = '1'
```

```
!pip install -U keras-vis
```

Code

```
!git clone https://github.com/keras-team/keras-vis
!cp -r /content/keras-vis/vis /usr/local/lib/python3.6/dist-packages/
!rm -r /content/keras-vis/vis
```

```
%matplotlib inline
```

G1: Exploratory Data Analysis

Label distribution of the dataset

```
def get_dataset():
    ds = {}
    ds['X_train'] = []
    ds['Y_train'] = []
    ds['X_test'] = []
    ds['Y_test'] = []
    for cond in ['Effusion', 'No Finding']:
        for split in ['train', 'test']:
            print('Processing %s %s...' % (cond, split))
            dataset = pickle.load(open('./%s_%s.p' % (cond, split), 'rb'))
            N = len(dataset)
            if cond == 'Effusion':
                Y = np.ones(N)
            else:
                Y = np.zeros(N)
            if split == 'train':
                ds['X_train'] = np.concatenate([ds['X_train'], dataset])
                ds['Y_train'] = np.concatenate([ds['Y_train'], Y])
            else:
                ds['X_test'] = np.concatenate([ds['X_test'], dataset])
                ds['Y_test'] = np.concatenate([ds['Y_test'], Y])
    return ds['X_train'], ds['Y_train'], ds['X_test'], ds['Y_test']
```

```
X_train, Y_train, X_test, Y_test = get_dataset()
print("Shape of training set: %s" % str(X_train.shape))
print("Shape of training labels: %s" % str(Y_train.shape))
print("Shape of test set: %s" % str(X_test.shape))
print("Shape of test labels: %s" % str(Y_test.shape))
```

```
%mkdir -p /tmp/keras/datasets/
%cp -r /root/.keras/datasets/mnist.npz /tmp/keras/datasets/
```

```
np.random.seed(42)
np.shape(X_train)
```

```
np.shape(X_test)
```

G1: Exploratory Data Analysis

Label distribution of the dataset

```
def get_dataset():
    ds = {}
    ds['X_train'] = []
    ds['Y_train'] = []
    ds['X_test'] = []
    ds['Y_test'] = []
    for cond in ['Effusion', 'No Finding']:
        for split in ['train', 'test']:
            print('Processing %s %s...' % (cond, split))
            dataset = pickle.load(open('./%s_%s.p' % (cond, split), 'rb'))
            N = len(dataset)
            if cond == 'Effusion':
                Y = np.ones(N)
            else:
                Y = np.zeros(N)
            if split == 'train':
                ds['X_train'] = np.concatenate([ds['X_train'], dataset])
                ds['Y_train'] = np.concatenate([ds['Y_train'], Y])
            else:
                ds['X_test'] = np.concatenate([ds['X_test'], dataset])
                ds['Y_test'] = np.concatenate([ds['Y_test'], Y])
    return ds['X_train'], ds['Y_train'], ds['X_test'], ds['Y_test']
```

```
X_train, Y_train, X_test, Y_test = get_dataset()
print("Shape of training set: %s" % str(X_train.shape))
print("Shape of training labels: %s" % str(Y_train.shape))
print("Shape of test set: %s" % str(X_test.shape))
print("Shape of test labels: %s" % str(Y_test.shape))
```

```
%mkdir -p /tmp/keras/datasets/
%cp -r /root/.keras/datasets/mnist.npz /tmp/keras/datasets/
```

```
np.random.seed(42)
np.shape(X_train)
```

```
np.shape(X_test)
```

Code

```
!git clone https://github.com/keras-team/keras-vis
!cp -r /content/keras-vis/vis /usr/local/lib/python3.6/dist-packages/
!rm -r /content/keras-vis/vis
```

```
%matplotlib inline
```

G1: Exploratory Data Analysis

Label distribution of the dataset

```
def get_dataset():
    ds = {}
    ds['X_train'] = []
    ds['Y_train'] = []
    ds['X_test'] = []
    ds['Y_test'] = []
    for cond in ['Effusion', 'No Finding']:
        for split in ['train', 'test']:
            print('Processing %s %s...' % (cond, split))
            dataset = pickle.load(open('./%s_%s.p' % (cond, split), 'rb'))
            N = len(dataset)
            if cond == 'Effusion':
                Y = np.ones(N)
            else:
                Y = np.zeros(N)
            if split == 'train':
                ds['X_train'] = np.concatenate([ds['X_train'], dataset])
                ds['Y_train'] = np.concatenate([ds['Y_train'], Y])
            else:
                ds['X_test'] = np.concatenate([ds['X_test'], dataset])
                ds['Y_test'] = np.concatenate([ds['Y_test'], Y])
    return ds['X_train'], ds['Y_train'], ds['X_test'], ds['Y_test']
```

```
X_train, Y_train, X_test, Y_test = get_dataset()
print("Shape of training set: %s" % str(X_train.shape))
print("Shape of training labels: %s" % str(Y_train.shape))
print("Shape of test set: %s" % str(X_test.shape))
print("Shape of test labels: %s" % str(Y_test.shape))
```

```
%mkdir -p /tmp/keras/datasets/
%cp -r /root/.keras/datasets/mnist.npz /tmp/keras/datasets/
```

```
np.random.seed(42)
np.shape(X_train)
```

```
np.shape(X_test)
```

Dataset Visualization

```
def plot_sample(X, Y, n, title=None):
    if X.shape[-1] == 1:
        X = X[..., 0]
    if Y.shape[-1] == 1:
        Y = Y[..., 0]
    plt.imshow(X[n])
    if title:
        plt.title(title + ' Label: %s' % str(Y[n]))
    else:
        plt.title('Label: %s' % str(Y[n]))
```

```
plot_sample(X_train, Y_train, 0, title='Effusion')
```

```
plot_sample(X_train, Y_train, 1, title='No Finding')
```

Resize and Normalize

```
def resize_and_normalize(X, new_size, mean=None, std=None):
    from keras.backend.tensorflow_backend import set_session
    import tensorflow as tf
    sess = tf.Session(config=tf.ConfigProto(gpu_options=tf.GPU_OPTIONS(allow_growth=True)))
    set_session(sess)
    X = tf.image.resize_bilinear(X, (new_size, new_size), align_corners=True).eval(session=sess)
    if mean:
        X = X - mean
    if std:
        X = X / std
    return X
```

```
X_train = resize_and_normalize(X_train, 224, 127.5, 127.5)
X_test = resize_and_normalize(X_test, 224, 127.5, 127.5)
```

```
plot_sample(X_train, Y_train, 0, title='Effusion')
```

```
plot_sample(X_train, Y_train, 1, title='No Finding')
```

Train/Test Split

```
np.random.seed(42)
X_train, Y_train = shuffle(X_train, Y_train)
```

```
np.shape(X_train)
```

```
np.shape(Y_train
```

Do the maps highlight sensible regions?

In the case of the pneumonia samples, viewing the amount we can be CutoffIN highlights similar regions in the cardiomegaly/other. Especially the heart seems to be focus. The other class, however, focuses the most mass in the lung fields, and seems to focus on the edges of the lung fields. This is likely the difference in pneumonia vs the rest.

Are attributions consistent across samples?

Attributions are not consistent across samples. Although it seems the for the healthy class, there is bigger focus of the lung fields, the regions at the bottom of the image, the second column also pulls out of the regions highlighting CutoffIN.

Q3. Data Randomization Test

Permute the labels

First let permute the labels and the samples them

Note that we are permuting only the training labels.

```
shuffled_labels = np.random.permutation(
    np.array(train_labels))
```

Now we save the shuffled labels in a dataset_dictionary

```
shuffled_dataset = dsdataset(train_paths, shuffled_labels, equalize=True)
shuffled_dataset.transform=transforms_train_list + dsdf.normalize
```

Retrain on shuffled labels

```
num_epochs=2 | bs=128 | lr=0.0001
```

```
lr = 0.0001
bs = 128
num_epochs = 2
torch.manual_seed(42)
train_loaders = {
    'train': DataLoader(
        shuffled_dataset,
        batch_size=bs,
        shuffle=True,
        num_workers=4,
        pin_memory=True,
        drop_last=True
    ),
    'test': DataLoader(
        test_dataset,
        batch_size=bs*2,
        shuffle=False,
        num_workers=4,
        pin_memory=True
    )
}
model = resnet18(pretrained=True)
model.fc = nn.Linear(512, 2)
model = model.to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=lr)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=7, gamma=0.1)
print(f"Training on {device}")
print(f"Batch size: {bs}")
print(f"Learning rate: {lr}")
print(f"Epochs: {num_epochs}")
print(f"Train samples: {len(shuffled_dataset)}")
print(f"Test samples: {len(test_dataset)}")
print(f"Classes: {shuffled_dataset.classes}")
print("="*60)
for epoch in range(num_epochs):
    print(f"\nEpoch {epoch+1}/{num_epochs}")
    print("-"*20)
    for phase in ['train', 'test']:
        if phase == 'train':
            model.train()
        else:
            model.eval()
        running_loss = 0.0
        running_corrects = 0
        total_samples = 0
        for inputs, labels in train_loaders[phase]:
            inputs = inputs.to(device)
            labels = labels.to(device)
            optimizer.zero_grad()
            with torch.set_grad_enabled(phase == 'train'):
                outputs = model(inputs)
                _, preds = torch.max(outputs, 1)
                loss = criterion(outputs, labels)
                if phase == 'train':
                    loss.backward()
                    optimizer.step()
            running_loss += loss.item() * inputs.size(0)
            running_corrects += torch.sum(preds == labels.data)
            total_samples += inputs.size(0)
        if phase == 'train':
            scheduler.step()
        epoch_loss = running_loss / total_samples
        epoch_acc = running_corrects.double() / total_samples
        print(f"{phase.capitalize()}: Loss: {epoch_loss:.4f}, Acc: {epoch_acc:.4f}")
print("\n" + "="*60)
print("Training complete!")
```

```
Training on cuda
Batch size: 128
Learning rate: 0.0001
Epochs: 2
Train samples: 5216
Test samples: 624
Classes: ['Cutoff-IN', 'Other']
============================================================

Epoch 1/2
--------------------
Train | 100%|##########| 40.8M/40.8M [00:00<00:00, 1.20s/s]
```

Part 3

Q1: How consistent were the different interpretable/explainable methods? Did they find similar patterns?

Part 1

We notice, the feature importances of logistic regression, the decision tree and the 2 layer NN show some differences.

- For example the decision tree has only a few important features, with more than half of them being very small.
- On the contrary in the case of logistic regression, we notice a more even spread of feature importances.
- For the neural additive models: we get a totally different picture with only 4 features having non-zero importance.

Overall we observed that, there is some overlap of features of high importances, but they do not completely align across all methods used.

Part 2

The method of integrated gradients was not helpful, which stands in contrast to GradCAM. Because the latter highlighted sensible regions and did produce different visualizations for healthy and disease samples. Whereas in the case of integrated gradients, we could not make out any distinction in the visualization between the two different classes. So we conclude that these two methods were not consistent.

Q2: Given the “interpretable” or “explainable” results of one of the models, how would you explain and present them to a broad audience? Pick one example per part of the project.

Part 1: Given some numerical tabular data, logistic regression derives coefficients for each feature of the data sample. The magnitudes of these coefficents are explainable by themself since they are only multiplied with the features and then summed up to get the output. Thus if the input features are of comparable size (normalization), then the coefficients can explain how important individual features are to arrive at a decision.

Part 2: GradCAM is a method used for models with images as inputs. It highlights regions of interest in the images, which the model might use for its decision. For example if a model is used to discriminate whether an image shows a dog or a cat, in the case of an image of a dog GradCAM is likely highlight the face and ears of the dog.

Q3: Did you encounter a tradeoff between accuracy and interpretability/explainability?

In Part 1, for the Neural Additive Models (NAMs) there was such a tradeoff. A simple 2 layer neural network performed better than the NAM in F1 and Balanced Accuracy score. But on the other hand the NAM more interpretable in terms of feature importances. Because it only had 4 non-zero absolute feature importances across all samples and for the 2 layer neural network the picture was not as clear, i.e. each feature had at least some contribution to the output.
Similiary both logistic regression and the decision performed worse than the 2 layer neural network as well. But one could argue that they are more interpretable given their simpler structure and already built-in feature importances with the magnitude of the log coefficients and the gini impurity.

Q4: Do your findings from the interpretability/explainability methods align with the current medical knowledge about these diseases? You may take inspiration from the references of the project presentation.

Heart Disease

In our case, the different models for heart disease were not that congruent in terms of feature importance. Nevertheless, one could observe a trend of features which were quite common among the different models such as: Up, ASY and SEX. Comparing these to the current medical knowledge, they roughly align with that since for example an up slope in case of ST depression during an electrocardiogram may indicate cardiac ischemia in combination with different cardiac symptoms. Furthermore, sex is a predictive feature as well, due to the fact that males are more oftenly affected by heart disease than women. Additionally, asymptomatic chestpain is a characteristic issue of heart related disease as well.

However, in medical literature the above named features do not count to the most indicative risk factors known. For example, cholesterol, resting blood pressure and diabetes are more often related to cardiac disease than the above named features. Especially that cholesterol was not even in the top 3 feature importances across our different models is suprising, since there is a proven causal relationship between heart disease and plaque formation/narrowing of the arteries due to high cholesterol levels.

As a result, our implemented models did not have the highest feature importances related to the most indicative features/ risk factors acknowledged by medicine but at least chose features which can be predictive for heart disease as well.

Pneunomia

Looking at the GradCAM, one can clearly conclude that it highlighted medically relevant regions for classification which one can especially observe in the pneumonia positive sampples. This is due to the fact that in majority parts of the left thorax get a lot of attention because the heart tends to cover major parts of the right thorax which is thereby not useful for a successful detection of pneumonia anymore. On the contrary, in pneumonia negative cases one can rather observe evenly distributed attention profiles which seem to be rather random.

Giving attention to the left thorax makes medically sense as well due to the fact that the detection of liquid in regions with a lot of contrast is rather easy compared to the right thorax which is partially covered by the heart (no/little contrast). Based on the attention profile of our trained model, we can therefore conclude that medically relevant features were extracted and as a result aligns with current medical knowledge.

Q5: If you had to deploy one of the methods in practice, which one would you choose and why?

We would choose to deploy GradCAM. It is easily applied to any model with convolutional layers and the computation is very low, i.e. it is comparable to a single backward pass during training. The resulting heatmaps are often very intuitive and can be interpreted nicely. They can be used to reason, why a model might do wrong predictions and can thus also help to properly identify issues such as bad performance and overfitting.