Review of case 1

Neural Networks for Health Tech Applications
Spring 2020
Sakari Lukkarinen
Helsinki Metropolia University of Applied Sciences

Contents

- Do
  - Clarity
  - Simplicity
  - Focus
- Don’t
  - Verbose = 2 (!!) ➔ use verbose = 0 for finals
  - Check the probelm ➔ Use proper settings
  - Be careful with data preprocessing

Case 1 - Heart Disease Classification

Your names here

31.1.2020
Neural Networks for Health Technology Applications
Helsinki Metropolia University of Applied Sciences

Introduction

In this exercise, the aim is to create and train a dense neural network to predict the presence of heart disease using the dataset at https://archive.ics.uci.edu/ml/datasets/Heart+Disease

In [540]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import RobustScaler
from sklearn.metrics import classification_report
from sklearn.metrics import roc_curve
from sklearn.metrics import auc
from sklearn.metrics import confusion_matrix
import tensorflow as tf
from scipy.signal import savgol_filter
```

Dataset

Let's load the UCI Heart Disease dataset and look further into its features.

The Cleveland database used here is a subset of this dataset. It has 14 attributes, contains 303 samples, and is commonly used in machine learning experiments. The presence of heart disease is distinguished by values 1, 2, 3 and 4. Value 0 denotes the absence of heart disease. It is not specified in the dataset documentation, which medical conditions are considered encompassed by "heart disease". The assumption is that coronary artery disease is the prevailing diagnosis in this data.

In [541]:
```
url = r'http://archive.ics.uci.edu/ml/machine-learning-databases/heart-disease/processed.cleveland.data'
col_names = ['age','sex','cp','trestbps','chol','fbs','restecg','thalac','exang','oldpeak','slope','ca','thal','num']
df = pd.read_csv(url,
                 sep = ',',
                 header = None,
                 index_col = None,
                 names = col_names,
                 na_values = '?')
df.describe()
```

Out[541]:

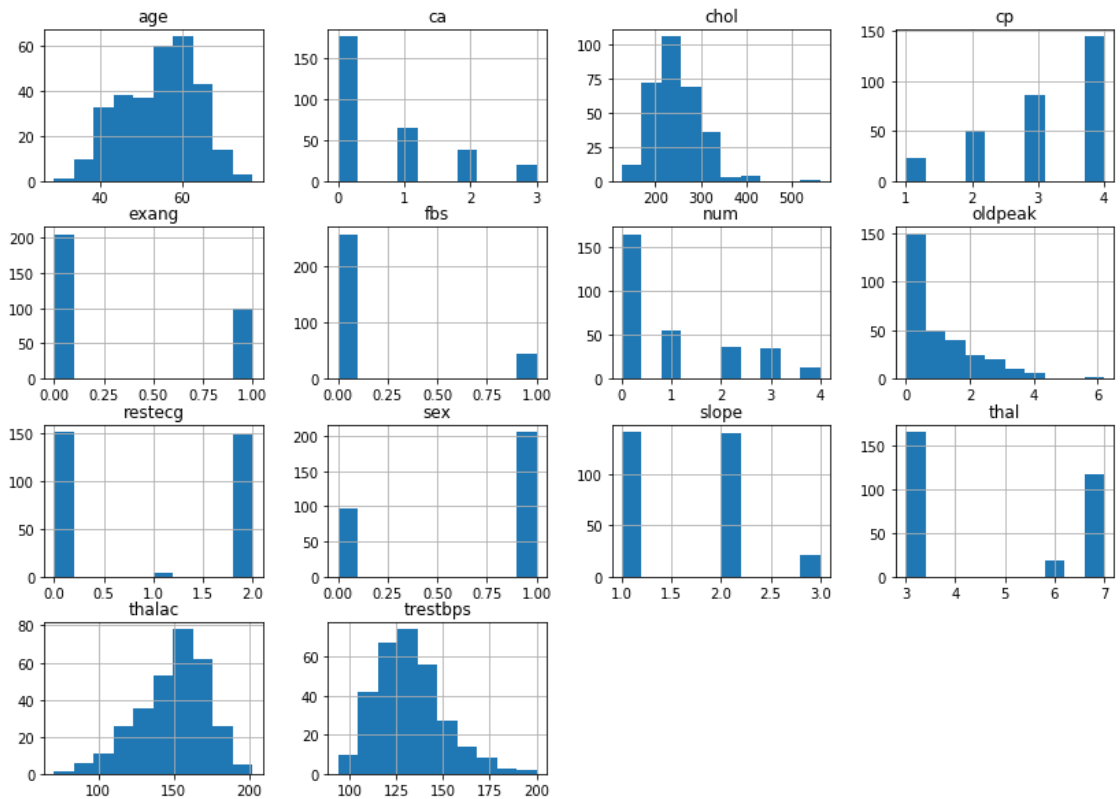
	age	sex	cp	trestbps	chol	fbs	restecg	thalac	exang	oldpeak	slope	ca	
count	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	299.000000	3
mean	54.438944	0.679868	3.158416	131.689769	246.693069	0.148515	0.990099	149.607261	0.326733	1.039604	1.600660	0.672241	
std	9.038662	0.467299	0.960126	17.599748	51.776918	0.356198	0.994971	22.875003	0.469794	1.161075	0.616226	0.937438	
min	29.000000	0.000000	1.000000	94.000000	126.000000	0.000000	0.000000	71.000000	0.000000	0.000000	1.000000	0.000000	
25%	48.000000	0.000000	3.000000	120.000000	211.000000	0.000000	0.000000	133.500000	0.000000	0.000000	1.000000	0.000000	
50%	56.000000	1.000000	3.000000	130.000000	241.000000	0.000000	1.000000	153.000000	0.000000	0.800000	2.000000	0.000000	
75%	61.000000	1.000000	4.000000	140.000000	275.000000	0.000000	2.000000	166.000000	1.000000	1.600000	2.000000	1.000000	
max	77.000000	1.000000	4.000000	200.000000	564.000000	1.000000	2.000000	202.000000	1.000000	6.200000	3.000000	3.000000	

Above is a summary containing statistical key figures of the dataset in use. The data is loaded into a pandas data frame.

Let's print a few key figures and make a quick histogram visualization of the data to get an idea of what we're dealing with.

In [542]:
```
data_size = df['num'].count()

empty = df.isna().sum().sum()
empty_p = (empty / data_size) * 100 # percentage

diseased = df['num'][df['num'] > 0].count()
diseased_p = (diseased / data_size) * 100 # percentage

healthy = data_size - diseased
healthy_p = (healthy / data_size) * 100 # percentage

print("Data size: %d rows, of which %d (%3.1f %%) contain empty values." %(data_size, empty, empty_p))
print("Healthy: %d (%3.1f %%) Diseased: %d (%3.1f %%)" %(healthy, healthy_p, diseased, diseased_p))

fig = df.hist(figsize=(14,10))
```

```
Data size: 303 rows, of which 6 (2.0 %) contain empty values.
Healthy: 164 (54.1 %) Diseased: 139 (45.9 %)
```

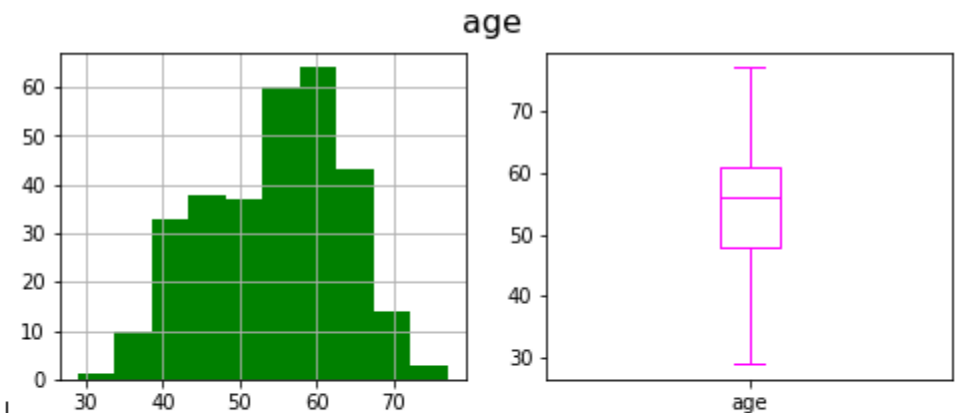


In [543]:
```
# define a helper function to visualize each column.
def plot_attribute(name):
    fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(8,3))
    fig.suptitle(name, fontsize=16)
    # histogram left
    df[name].hist(ax = axes[0], color='green')
    # box plot right - this may help spot outliers.
    df[name].plot(ax = axes[1], kind='box', color="magenta")
    plt.show()
```

Attribute Description

Now we will describe each attribute in more detail and consider their significance. We will also decide for each column, if they should be excluded from the analysis.

In [544]:
```
plot_attribute('age')
print("min:", df['age'].min(), "max:", df['age'].max())
```




```
min: 29.0 max: 77.0
```

Patient age in years. Increasing age is recognized as a major risk factor for coronary disease by American Heart Association.

Preprocessing

scale or normalize the variables
replace missing values with zeros, means/medians or random values

Imputation (handling of empty values)

There are six samples in the dataset containing empty values. This represents only 2 % of all the records, as mentioned earlier. The empty values occur in columns `ca` and `thal`, which are both problematic when it comes to inventing meaningful substitute data. Both are categorical variables containing actual clinical results, so all kinds of mean/interpolation methods will probably just create distortion and error. Therefore we choose to do listwise deletion, eliminating incomplete samples entirely.

In [557]:
```
# eliminate incomplete data
df = df.dropna()
```

Splitting the Data

We now split the data into two sets, setting aside 20 percent of it in a test set. The **test set** will only be used for final model performance evaluation, and its contents shall not be exposed to the model development process in any way.

The rest 80 % of the data will be used for training and validation. The data split is done following the 60-20-20 rule introduced in class.

In [558]:
```
df['num'] = (df['num'] > 0) * 1.0  # convert output labels to zeros and ones
train_data, test_data = np.split(df, [int(.8 * len(df)),])
```

Now that data has been split, let's examine if these subsets are actually representative of the entire data. We want both training and testing sets to contain both sick and healthy individuals in roughly the same proportion as the whole dataset.

In [559]:
```
train_counts = train_data['num'].value_counts()
test_counts = test_data['num'].value_counts()
total_counts = train_counts + test_counts
categories = train_data['num'].unique()

plt.bar(categories+0.125, total_counts / df['num'].count(), width=0.5, alpha=0.5)
plt.bar(categories, train_counts / train_data['num'].count(), width=0.1)
plt.bar(categories+0.25, test_counts / test_data['num'].count(), width=0.1)
plt.legend(['total', 'train', 'test'], loc=3)
plt.xticks([0.5, 1.5], labels=['healthy', 'sick'])
plt.title('Representation of target classes in split data sets')

plt.show()
```

Target class distribution seems fine in both subsets. Total sick-healthy-distribution is close to 50:50, which suggests, that *accuracy* will be a useful metric when evaluating the model.

Scaling

There are many ways to scale the data to even out each attribute's effect on the model. We could normalize it by subtracting mean and dividing by standard deviation, or standardize it by scaling its range down to -1...1. Both of these methods behave badly with outliers, compressing the inliers into a narrow range. Using sklearn's `RobustScaler` mitigates this problem.

The scaling parameters are fitted to the training/validation set. The training set and test set are both then scaled using these parameters.

In [560]:
```
# set aside output labels, while scaling input data
train_num = train_data['num'].values
test_num = test_data['num'].values

# using the RobustScaler that was advertised on Wednesday's lecture
scaler = RobustScaler()
# fit scaler parameters to the training/validation set and scale it
train_data = pd.DataFrame(scaler.fit_transform(train_data))
train_data.columns = col_names
# scale the test set using the same parameters
test_data = pd.DataFrame(scaler.transform(test_data))
test_data.columns = col_names

# replace original target labels
train_data['num'] = (train_num > 0) * 1.0
test_data['num'] = (test_num > 0) * 1.0
```

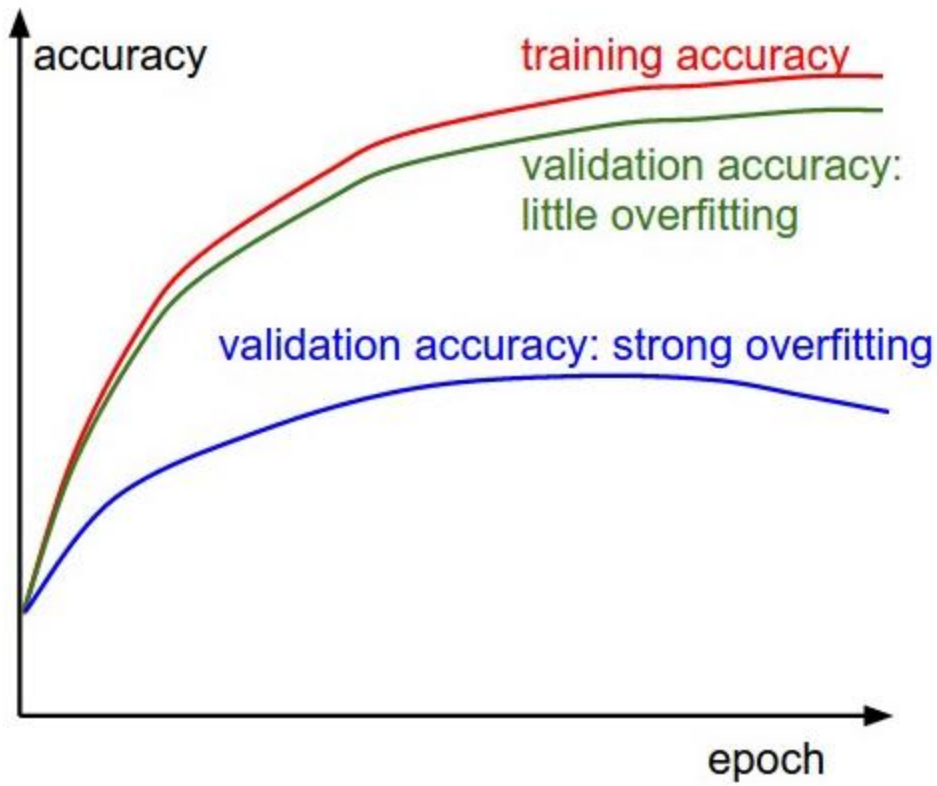
Model

Next, we will construct a feed-forward neural network to predict heart disease. We experiment by adjusting hyperparameters like number of layers, number of neurons, choice of optimizer and its learning rate, activation function, dropout, regularization, batch size and number of epochs.

We will inspect the learning curves to detect overfitting, maximize accuracy and minimize loss. Based on the learning curve, we choose a good number of epochs for early stopping.

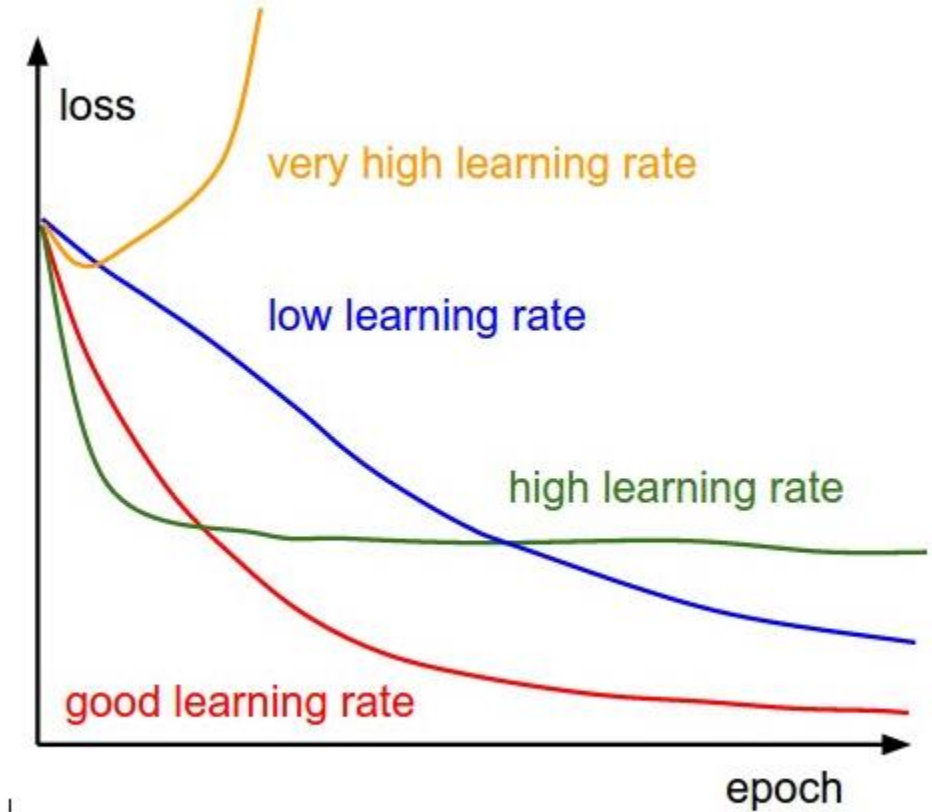
While iteratively improving our model, we employ 20 % of the data set (25 % of the training set) for validation. This allows us to detect overfitting. Diverging learning curves for training and validation is a sign of overfitting.

We use this material from Stanford cs231n course as a guideline for adjusting hyperparameters like learning rate and model complexity.



"The gap between the training and validation accuracy indicates the amount of overfitting." Source: Stanford University

In this case we can, for example, add regularization, decrease model complexity or add dropout.



"With low learning rates the improvements will be linear. With high learning rates they will start to look more exponential. Higher learning rates will decay the loss faster, but they get stuck at worse values of loss (green line)." Source: Stanford University

In [563]:
```
def create_model():
    reg = tf.keras.regularizers.l2(0.002)
    model = tf.keras.models.Sequential([
        tf.keras.layers.Dense(13, activation='sigmoid', kernel_regularizer=reg, input_shape=(train_X.shape[1],)),
        tf.keras.layers.Dropout(0.2),
        tf.keras.layers.Dense(6, activation='sigmoid', kernel_regularizer=reg),
        tf.keras.layers.Dense(1, activation='sigmoid')
    ])

    opt = tf.keras.optimizers.RMSprop(lr=0.001)

    model.compile(optimizer=opt,
                  loss='binary_crossentropy',
                  metrics=['accuracy'])
    return model

batch_size = 30
epochs = 300
```

In [564]:
```
model = create_model()
#print(model.summary())
```

In [565]:
```
history = model.fit(np.asarray(train_X),
                    np.asarray(train_y),
                    verbose=0,
                    batch_size=batch_size,
                    validation_split=0.25,
                    epochs=epochs)
```

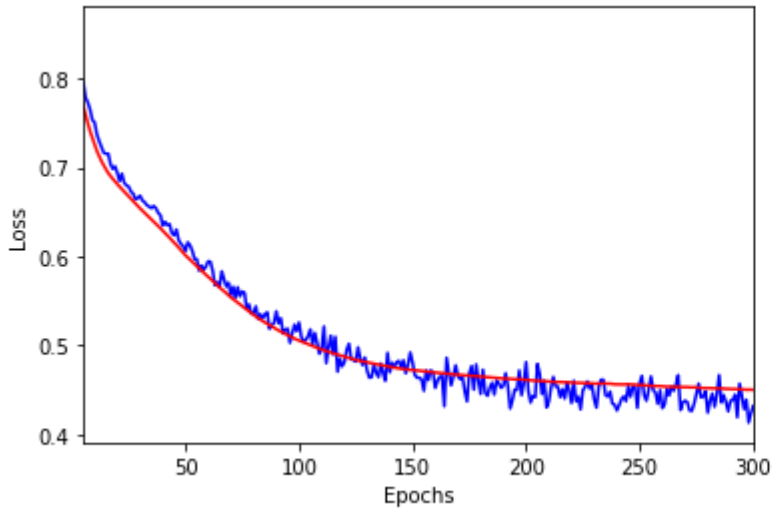
In [566]:
```
model.evaluate(np.asarray(train_X), np.asarray(train_y), verbose=2)
```

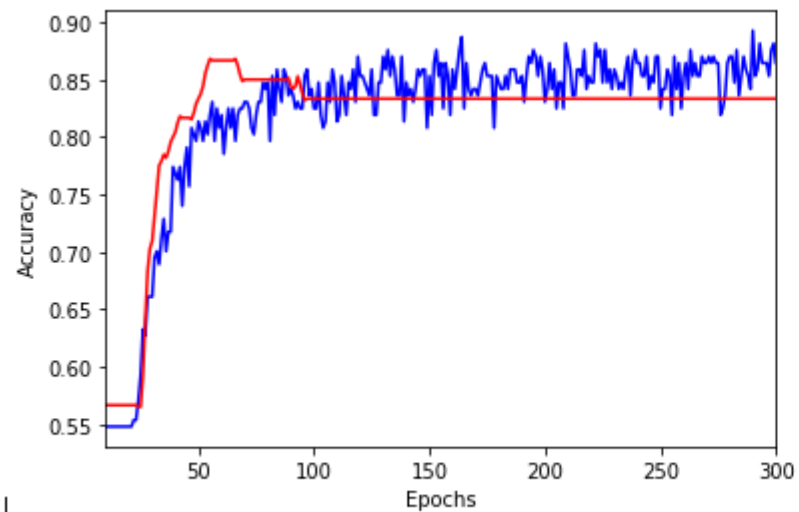
```
237/1 - 1s - loss: 0.3514 - accuracy: 0.8692
```

Out[566]:
```
[0.42446552210719274, 0.8691983]
```

In [567]:
```
# plot loss and accuracy learning curves for training and validation sets

print(history.history.keys())
loss = history.history['loss']
val_loss = history.history['val_loss']
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

time = range(1,len(loss)+1)

plt.plot(time, loss, 'b-')
plt.plot(time, val_loss, 'r-')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.xlim((5,epochs))
plt.show()

plt.plot(time, acc, 'b-')
plt.plot(time, savgol_filter(val_acc, 5, 3), 'r-')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.xlim((10,epochs))
plt.show()
```

```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```


0.8
0.7
0.6
0.5
0.4
Loss
50
100
150
200
250
300
Epochs


0.90
0.85
0.80
0.75
0.70
0.65
0.60
0.55
Accuracy
50
100
150
200
250
300
Epochs

The code from this point on should only be used in final evaluation

If we run this code **and** make modifications to the model based on its results, information will leak from the test set into our model. We have tucked away the test set to do final testing with data previously unseen to our model, and taken measures to prevent data leakage at any point in development.

A steady stream of fresh data would eliminate the need to guard this test set so closely.

Final Evaluation

Next, we create a fresh model and retrain it from scratch, now using the entire training set without reserving data for validation. Then we evaluate the model against the previously unseen test set to see how well it actually performs.

If we see much poorer performance (accuracy) here, compared to training and validation, our model is probably overfitted, i.e. it has become an expert in reciting the training/validation set, but gets confused when encountering unseen samples.

In [568]:
```
# create a fresh model for final evaluation
model = create_model()
# train model with early stopping based on learning curve, to avoid overfitting
# use all data: no validation set
_ = model.fit(np.asarray(train_X), np.asarray(train_y), verbose=0, batch_size=batch_size, epochs=170)
```

In [569]:
```
loss, acc = model.evaluate(np.asarray(test_X), np.asarray(test_y), verbose=0)
print("loss:", loss, "accuracy:", acc)
```

```
loss: 0.5868688106536866 accuracy: 0.75
```

Results

Let's use various **metrics** to gain insight into our model's performance.

In [570]:
```
y_pred = np.array(model.predict(np.asarray(test_X))).flatten()
target_names = ['Healthy', 'Sicko']
print(classification_report(test_y, y_pred.round(), target_names=target_names))
```

```
              precision    recall  f1-score   support

     Healthy       0.69      0.86      0.77        29
       Sicko       0.83      0.65      0.73        31

    accuracy                           0.75        60
   macro avg       0.76      0.75      0.75        60
weighted avg       0.77      0.75      0.75        60
```

The **classification report** shows accuracy, precision, recall and f1-score.

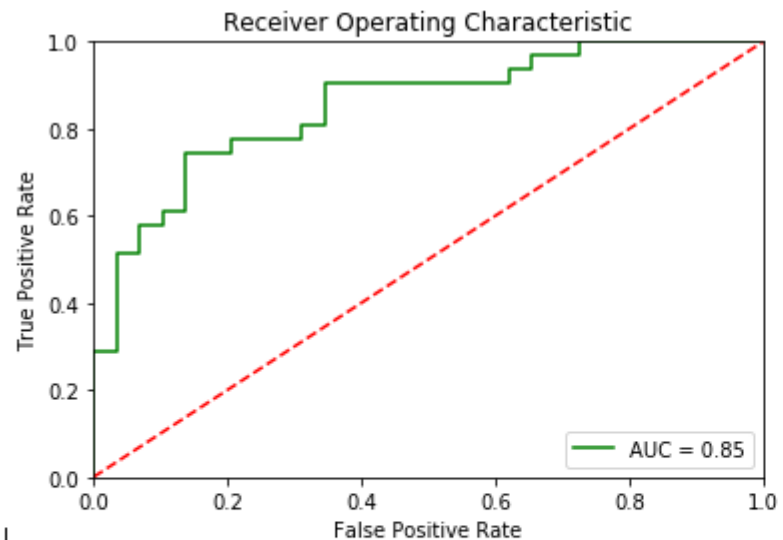
In [571]:
```
confusion_matrix(test_y, y_pred.round()).T
```

Out[571]:
```
array([[25, 11],
       [ 4, 20]], dtype=int64)
```

This **confusion matrix** contains the following values:

```
true positives    false positives
false negatives   true negatives
```

In [572]:
```
# calculate the fpr and tpr for all thresholds of the classification
probs = model.predict_proba(np.asarray(test_X))
preds = probs[:,0]
fpr, tpr, threshold = roc_curve(test_y, preds)
roc_auc = auc(fpr, tpr)

plt.title('Receiver Operating Characteristic')
plt.plot(fpr, tpr, 'b', color='green', label = 'AUC = %0.2f' % roc_auc)
plt.legend(loc = 'lower right')
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```



The **ROC curve** plots true positive rate against false positive rate at different discrimination thresholds. A high AUC (Area Under Curve) is a measure of good model performance.

The ROC curve should travel as close to upper left corner as possible.

Conclusions

Designing a viable ANN classifier is tedious work for an inexperienced novice.

[ ]:

TO IMPROVE

Use verbose = 0 for final

```
In [10]: network = model.fit(data, labels, epochs = 50, batch_size = 8, validation_split = 0.10)

Train on 272 samples, validate on 31 samples
Epoch 1/50
272/272 [==============================] - 2s 8ms/sample - loss: 0.7224 - accuracy: 0.5404 - val_loss:
0.6266 - val_accuracy: 0.6774
Epoch 2/50
272/272 [==============================] - 0s 614us/sample - loss: 0.6263 - accuracy: 0.6875 - val_loss:
0.5730 - val_accuracy: 0.7742
Epoch 3/50
272/272 [==============================] - 0s 612us/sample - loss: 0.5505 - accuracy: 0.7868 - val_loss:
0.5356 - val_accuracy: 0.8065
Epoch 4/50
272/272 [==============================] - 0s 611us/sample - loss: 0.4873 - accuracy: 0.8162 - val_loss:
0.5053 - val_accuracy: 0.8065
Epoch 5/50
272/272 [==============================] - 0s 611us/sample - loss: 0.4410 - accuracy: 0.8346 - val_loss:
0.4864 - val_accuracy: 0.8387
Epoch 6/50
272/272 [==============================] - 0s 622us/sample - loss: 0.4093 - accuracy: 0.8419 - val_loss:
0.4858 - val_accuracy: 0.8387
Epoch 7/50
272/272 [==============================] - 0s 614us/sample - loss: 0.3895 - accuracy: 0.8456 - val_loss:
0.4782 - val_accuracy: 0.8387
Epoch 8/50
272/272 [==============================] - 0s 616us/sample - loss: 0.3750 - accuracy: 0.8603 - val_loss:
0.4726 - val_accuracy: 0.8387
Epoch 9/50
272/272 [==============================] - 0s 696us/sample - loss: 0.3643 - accuracy: 0.8640 - val_loss:
0.4697 - val_accuracy: 0.8387
Epoch 10/50
272/272 [==============================] - 0s 635us/sample - loss: 0.3543 - accuracy: 0.8713 - val_loss:
0.4668 - val_accuracy: 0.8065
Epoch 11/50
272/272 [==============================] - 0s 631us/sample - loss: 0.3459 - accuracy: 0.8750 - val_loss:
0.4735 - val_accuracy: 0.8065
Epoch 12/50
272/272 [==============================] - 0s 623us/sample - loss: 0.3410 - accuracy: 0.8640 - val_loss:
0.4632 - val_accuracy: 0.8065
Epoch 13/50
272/272 [==============================] - 0s 616us/sample - loss: 0.3349 - accuracy: 0.8824 - val_loss:
0.4607 - val_accuracy: 0.8065
Epoch 14/50
272/272 [==============================] - 0s 605us/sample - loss: 0.3291 - accuracy: 0.8824 - val_loss:
0.4643 - val_accuracy: 0.8065
Epoch 15/50
272/272 [==============================] - 0s 642us/sample - loss: 0.3236 - accuracy: 0.8860 - val_loss:
0.4642 - val_accuracy: 0.8065
Epoch 16/50
272/272 [==============================] - 0s 655us/sample - loss: 0.3196 - accuracy: 0.8860 - val_loss:
0.4640 - val_accuracy: 0.8065
Epoch 17/50
272/272 [==============================] - 0s 617us/sample - loss: 0.3137 - accuracy: 0.8897 - val_loss:
0.4664 - val_accuracy: 0.8065
Epoch 18/50
272/272 [==============================] - 0s 625us/sample - loss: 0.3092 - accuracy: 0.8897 - val_loss:
0.4652 - val_accuracy: 0.8065
Epoch 19/50
272/272 [==============================] - 0s 640us/sample - loss: 0.3055 - accuracy: 0.8897 - val_loss:
0.4653 - val_accuracy: 0.8065
Epoch 20/50
Epoch 25/50
272/272 [==============================] - 0s 615us/sample - loss: 0.2811 - accuracy: 0.8971 - val_loss:
0.4613 - val_accuracy: 0.8065
Epoch 26/50
272/272 [==============================] - 0s 620us/sample - loss: 0.2784 - accuracy: 0.9007 - val_loss:
0.4575 - val_accuracy: 0.8065
Epoch 27/50
272/272 [==============================] - 0s 605us/sample - loss: 0.2746 - accuracy: 0.9044 - val_loss:
0.4508 - val_accuracy: 0.8065
Epoch 28/50
272/272 [==============================] - 0s 618us/sample - loss: 0.2713 - accuracy: 0.9007 - val_loss:
0.4536 - val_accuracy: 0.8065
Epoch 29/50
272/272 [==============================] - 0s 612us/sample - loss: 0.2691 - accuracy: 0.9081 - val_loss:
0.4548 - val_accuracy: 0.8065
Epoch 30/50
272/272 [==============================] - 0s 618us/sample - loss: 0.2665 - accuracy: 0.9081 - val_loss:
0.4534 - val_accuracy: 0.8065
Epoch 31/50
272/272 [==============================] - 0s 618us/sample - loss: 0.2628 - accuracy: 0.9081 - val_loss:
0.4586 - val_accuracy: 0.8065
Epoch 32/50
272/272 [==============================] - 0s 624us/sample - loss: 0.2608 - accuracy: 0.9154 - val_loss:
0.4641 - val_accuracy: 0.8065
Epoch 33/50
272/272 [==============================] - 0s 603us/sample - loss: 0.2585 - accuracy: 0.9081 - val_loss:
0.4619 - val_accuracy: 0.8065
Epoch 34/50
272/272 [==============================] - 0s 616us/sample - loss: 0.2550 - accuracy: 0.9228 - val_loss:
0.4669 - val_accuracy: 0.8065
Epoch 35/50
272/272 [==============================] - 0s 600us/sample - loss: 0.2545 - accuracy: 0.9081 - val_loss:
0.4557 - val_accuracy: 0.8065
Epoch 36/50
272/272 [==============================] - 0s 620us/sample - loss: 0.2499 - accuracy: 0.9154 - val_loss:
0.4585 - val_accuracy: 0.8065
Epoch 37/50
272/272 [==============================] - 0s 604us/sample - loss: 0.2506 - accuracy: 0.9154 - val_loss:
0.4565 - val_accuracy: 0.8065
Epoch 38/50
272/272 [==============================] - 0s 609us/sample - loss: 0.2467 - accuracy: 0.9191 - val_loss:
0.4573 - val_accuracy: 0.8065
Epoch 39/50
272/272 [==============================] - 0s 638us/sample - loss: 0.2446 - accuracy: 0.9154 - val_loss:
0.4564 - val_accuracy: 0.8065
Epoch 40/50
272/272 [==============================] - 0s 604us/sample - loss: 0.2412 - accuracy: 0.9228 - val_loss:
0.4694 - val_accuracy: 0.8065
Epoch 41/50
272/272 [==============================] - 0s 611us/sample - loss: 0.2412 - accuracy: 0.9228 - val_loss:
0.4591 - val_accuracy: 0.8065
Epoch 42/50
272/272 [==============================] - 0s 617us/sample - loss: 0.2380 - accuracy: 0.9228 - val_loss:
0.4600 - val_accuracy: 0.8065
Epoch 43/50
272/272 [==============================] - 0s 627us/sample - loss: 0.2359 - accuracy: 0.9265 - val_loss:
0.4576 - val_accuracy: 0.8065
Epoch 44/50
272/272 [==============================] - 0s 623us/sample - loss: 0.2335 - accuracy: 0.9301 - val_loss:
0.4686 - val_accuracy: 0.8065
Epoch 45/50
272/272 [==============================] - 0s 601us/sample - loss: 0.2315 - accuracy: 0.9265 - val_loss:
0.4626 - val_accuracy: 0.8065
Epoch 46/50
272/272 [==============================] - 0s 602us/sample - loss: 0.2307 - accuracy: 0.9301 - val_loss:
0.4644 - val_accuracy: 0.8065
Epoch 47/50
272/272 [==============================] - 0s 619us/sample - loss: 0.2275 - accuracy: 0.9301 - val_loss:
0.4601 - val_accuracy: 0.8065
Epoch 48/50
272/272 [==============================] - 0s 607us/sample - loss: 0.2261 - accuracy: 0.9301 - val_loss:
0.4584 - val_accuracy: 0.8065
Epoch 49/50
```

Check the problem

Model 1

In [4]:
```
# Construct the model 1, no regularization

model = Sequential()
model.add(Dense(50, activation='relu', input_shape=(data_train.shape[1],)))

model.add(Dense(1))

model.compile(optimizer='rmsprop', loss='mse', metrics=['mae'])

history = model.fit(partial_data_train, partial_labels_train,
                    epochs=130, batch_size=64, verbose=0,
                    validation_data = (data_val, labels_val))
```

Be careful with data preprocessing

```
# since we have few instances with missing value we will remove them from the dataset
data_rfc = data.dropna()

# Convert categorical attributes into dummy/indicator variables (One hot encoding)
data_rfc = pd.get_dummies(data_rfc, columns = ['sex', 'cp', 'fbs', 'restecg', 'exang', 'slope', 'ca', 'thal'])

# scale non categorical attributes
standardScaler = preprocessing.StandardScaler()
columns_to_scale = ['age', 'trestbps', 'chol', 'thalach', 'oldpeak']
data_rfc[columns_to_scale] = standardScaler.fit_transform(data_rfc[columns_to_scale])

# split data
samples= data_rfc.iloc[:,0:13].to_numpy()

# simplifying the target column to 1 or 0 meaning a person have disease or not
labels = 1.0 * (data_rfc.iloc[:, 13] > 0).to_numpy()

# splitting data with 30 percent for test and 70 percent for trainning, setting random state of keras to easily reproduce on re-
train_samples,test_samples,train_labels,test_labels = train_test_split(samples, labels, test_size=0.3, random_state=10)

# print one hot encoded and scaled data
data_rfc.head()
```

Out[8]:

	age	trestbps	chol	thalach	oldpeak	num	sex_0.0	sex_1.0	cp_1.0	cp_2.0	...	slope_1.0	slope_2.0	slope_3.0	ca_0.0	ca_1.0	ca_2.
0	0.936181	0.750380	-0.276443	0.017494	1.068965	0	0	1	1	0	...	0	0	1	1	0	
1	1.378929	1.596266	0.744555	-1.816334	0.381773	2	0	1	0	0	...	0	1	0	0	0	
2	1.378929	-0.659431	-0.353500	-0.899420	1.326662	1	0	1	0	0	...	0	1	0	0	0	
3	-1.941680	-0.095506	0.051047	1.633010	2.099753	0	0	1	0	0	...	0	0	1	1	0	
4	-1.498933	-0.095506	-0.835103	0.978071	0.295874	0	1	0	0	1	...	1	0	0	1	0	

5 rows × 29 columns