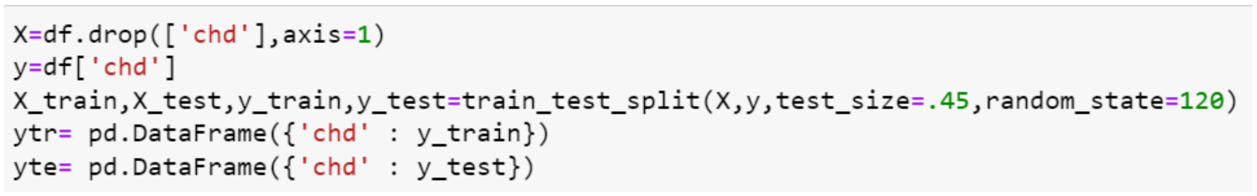
Data-splitting decisions:

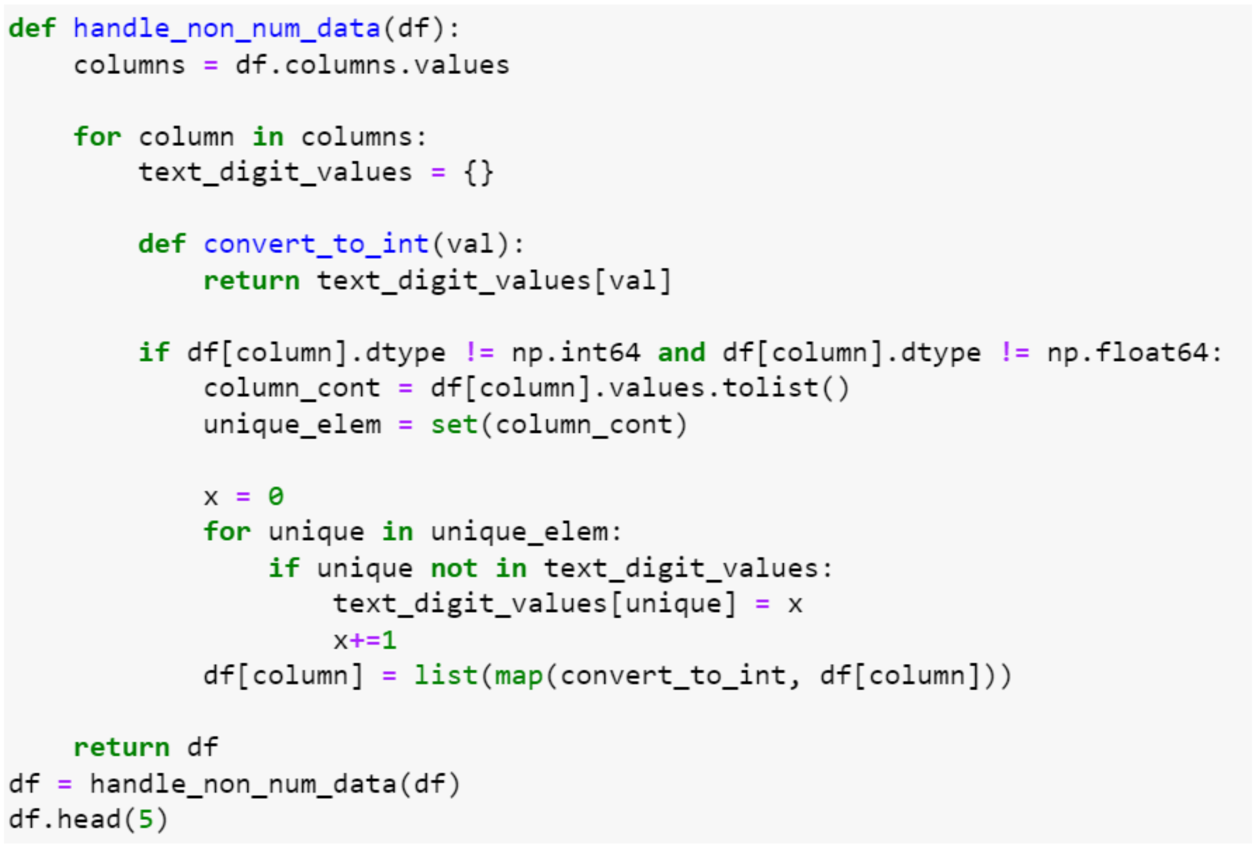
First, I divided the data into 20% for testing and 80% for training. However, the accuracy for testing was very bad compared with the training's accuracy. To solve this problem, I decided to give 45% for testing and the rest for training. In this case, we have increased the amount of tested data, which will increase the accuracy of the final result. For example, testing 100 records would lead to more precise results than testing 1 record with 100% accuracy because the 1 record is not enough to generalize.



```
X=df.drop(['chd'],axis=1)
y=df['chd']
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=.45,random_state=120)
ytr= pd.DataFrame({'chd' : y_train})
yte= pd.DataFrame({'chd' : y_test})
```

Non-numerical input data:

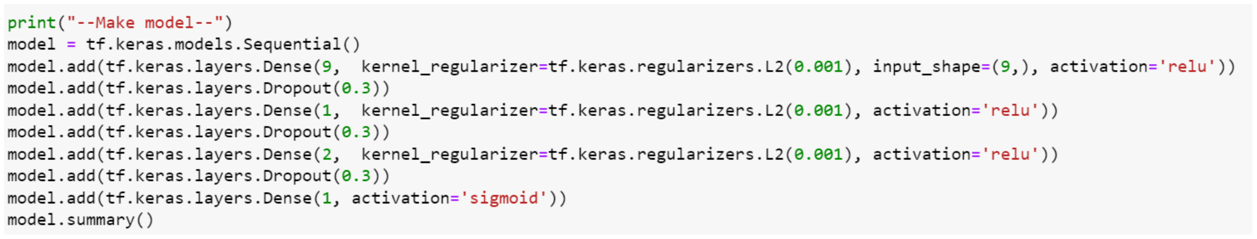
There is 1 string column in our dataset, called famhist. I decided to go with Binary Encoding since the whole column has only two distinct values (Present, Absent) . I wrote a function in Python called handle_non_num_data to convert the string to either 0 for Present or 1 for Absent.



```
def handle_non_num_data(df):
    columns = df.columns.values

    for column in columns:
        text_digit_values = {}

        def convert_to_int(val):
            return text_digit_values[val]

        if df[column].dtype != np.int64 and df[column].dtype != np.float64:
            column_cont = df[column].values.tolist()
            unique_elem = set(column_cont)

            x = 0
            for unique in unique_elem:
                if unique not in text_digit_values:
                    text_digit_values[unique] = x
                    x+=1
            df[column] = list(map(convert_to_int, df[column]))

    return df
df = handle_non_num_data(df)
df.head(5)
```

Dealing with overfitting/underfitting:

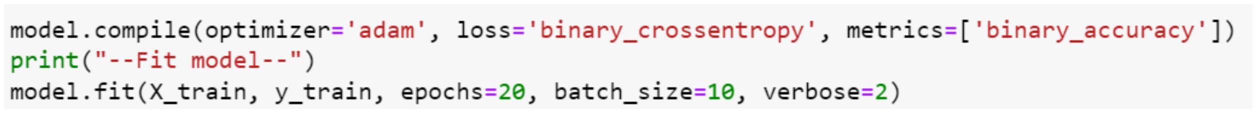
Due to the small size of the dataset, we have an overfit. The accuracy of trained data outweighs the test's accuracy. First, I decided to increase the amount of tested data. I used tf.keras.layers.Dropout(Probability), which leads to ignoring random numbers of neurons. Dropping them temporarily for a random number of epochs. Second, I used regularization to add a penalty to the error function. Third, decrease the number of neurons instead of having a large network to eliminate the overfitting problem.



```
print("--Make model--")
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Dense(9,  kernel_regularizer=tf.keras.regularizers.L2(0.001), input_shape=(9,), activation='relu'))
model.add(tf.keras.layers.Dropout(0.3))
model.add(tf.keras.layers.Dense(1,  kernel_regularizer=tf.keras.regularizers.L2(0.001), activation='relu'))
model.add(tf.keras.layers.Dropout(0.3))
model.add(tf.keras.layers.Dense(2,  kernel_regularizer=tf.keras.regularizers.L2(0.001), activation='relu'))
model.add(tf.keras.layers.Dropout(0.3))
model.add(tf.keras.layers.Dense(1, activation='sigmoid'))
model.summary()
```

Hyper-parameters and results:

The neural network has 4 layers. Number 1 is the input. Number 2 and 3 are the hidden layers. Number 4 consists of one node and represents the output layer. The Dropout value is 0.3. The optimizer is 'adam' and the loss is 'binary_crossentropy'. The epochs' value is 12. The result for training data is 62.6%, and Test Accuracy is 68.8%.



```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['binary_accuracy'])
print("--Fit model--")
model.fit(X_train, y_train, epochs=20, batch_size=10, verbose=2)
```